

Continuation Based Control in the Implementation of Parallel Logic Programs

John S. Conery (conery@cs.uoregon.edu) *

Department of Computer and Information Science

University of Oregon

Abstract. Continuations are used to define the flow of messages between low level tasks in a parallel logic programming language. A combination of compiler and runtime operations reduces message traffic by up to 50% when success continuations are passed as parameters in messages that start new processes. Continuations are also the key to fast task switching, a critical operation in this fine grain parallel system. Data from sample programs shows the effectiveness of continuations in reducing message traffic and the speed with which task switches are performed on a typical host architecture.

Keywords: continuation, logic programming, implementation, parallel, message passing

Conery, J.S. Continuation-based control in the implementation of parallel logic programs. *Lisp and Symbolic Computation* 7(111-134), 1994.

* Supported by NSF Grant CCR-8707177 and grants from Motorola, Inc, and Hewlett-Packard Corp.

1. Introduction

The notion of a continuation is very well known in programming language research. Abstract continuations have long been used in descriptions of programming language semantics, and the concept was elevated to first-class citizenship in languages such as Actors [1] and Scheme [3].

This paper describes how continuations have been applied to the implementation of control structures for a parallel language. The language, OPAL, is a logic programming language based on the AND/OR process model [4]. In this model, programs are compiled into a set of actor-like processes that communicate solely via messages. Continuations are not first-class citizens in OPAL since programmers cannot manipulate continuations to achieve different patterns of control. Rather we use continuations as a formal framework for implementation techniques that have greatly improved performance. The use of continuations has improved the implementation of OPAL in two ways. Organizing message passing via continuations uncovered several improvements in message passing patterns, improvements we are able to exploit through a combination of runtime decisions and compiler optimizations. Continuations also provide the key to a very efficient task switching technique that is well-suited to distributed memory multiprocessors.

The first part of the paper is a brief overview of the OPAL language and the underlying AND/OR process model. Following that is a description of how continuations are used to improve control, with some data from a few small benchmarks to confirm the intuition that continuations achieve up to a 50% reduction in the number of messages. Section 4 is on the implementation of continuations, explaining the representation of messages and continuation IDs, with some measurements of how fast a system can switch between processes when continuations are used to control message passing. The final section is a summary and discussion of future research.

2. OPAL and the AND/OR Model

The name OPAL is an acronym for Oregon *PAR*allel Logic. OPAL is a pure Horn clause language augmented with predicates for arithmetic and simple operations such as testing to see if a term is an unbound variable or finding the functor of a term. A few more complex operations, such as I/O, are implemented via a front-end host machine, but they are “cavalier” operations and no attempt is made to serialize programs through these constructs (for an example of a system that

Example 1: epath/2. The complete program has 16 arcs; only three are shown here. There are 51 solutions to the top level goal in this highly nondeterministic example.

```
goal <- epath(X,Y).

epath(A,C) <- arc(A,B) & arc(B,C).
epath(A,D) <- arc(A,B) & arc(B,C) & epath(C,D).

arc(0,1).
arc(0,2).
arc(0,4).
```

Example 2: append/3. This is the common list processing demo, which is the same in OPAL as it is in Prolog. With this pattern of parameters in the top level goal the program is deterministic.

```
goal <- append([a,b,c,d,e],[f,g,h],L).

append([],Y,Y).
append([A|X],Y,[A|Z]) <- append(X,Y,Z).
```

Example 3: some rules from a symbolic differentiation program.

```
d(U*V,X,P) <- functor(U,_,N) & product(N,U,V,X,P).

product(0,C,U,X,C*DU) <- C \== X & d(U,X,DU).
product(N,U,V,X,DV*U+DU*V) <- N > 0 & d(U,X,DU) & d(V,X,DV).
```

Figure 1. Three OPAL Programs

delays side effect predicates until all preceding side effects have been executed, see [7]). Other built-in operations that depend on side effects, such as Prolog's internal database, are not part of the language.

Figure 1 shows three simple OPAL programs. The first is an OR-parallel program that searches for all even length paths in an acyclic graph. The parallelism comes from multiple arcs leaving a node that is being searched, and the overall effect is of a pipeline where shorter paths are reported first. The second program is the familiar `append` benchmark which shows how to concatenate lists in a logic programming language. It is often used to demonstrate the conversion of tail recursion into iteration, and we will use it to illustrate how continuations achieve a similar effect in OPAL. The third example is part of a

program for symbolic differentiation and shows a combination of AND and OR parallelism. The two recursive calls to $\mathbf{d}/\mathbf{3}$ ¹ in the body of the rule that differentiates products can be done in parallel since \mathbf{X} , the only common variable, will be bound when the body is invoked.

The operational semantics of OPAL programs is defined by the AND/OR Process Model. In this model, programs are executed by collections of asynchronous objects that communicate solely via messages. There are two types of objects: AND processes and OR processes. An AND process solves the set of goals in the body of a clause, and an OR process coordinates the solution of a single goal with multiple definitions.

Messages are used to start new processes for subgoals, report results, or perform control operations such as asking for another result or canceling a subgoal. The four basic message types are:

- *Start*, which creates a new process.
- *Succeed*, which sends the result of a procedure call back to a calling process.
- *Redo*, sent by a parent to a descendant process when it needs another solution.
- *Fail*, used to report the failure of a goal or goal statement.

A procedure call in Prolog corresponds to an OR process in OPAL. When the system needs to solve a single goal, it creates an OR process for the goal and sends a start message to the new process. When an OR process successfully matches the call with the head of a clause, it starts an AND process for the body and sends it a start message.

The redo message is the key to nondeterministic execution. When a procedure has multiple solutions, the corresponding OR process will generate more than one success message. These messages are buffered at the OR process and sent back to the calling process on demand. The first one is always returned immediately, since the start message serves as a demand for the first solution, but the rest are saved and returned one at a time in response to redo messages. The actual implementation is more efficient than this because it bundles up multiple responses into one machine level message, but at the level of the abstract model the system operates on one result at a time.

The OPAL implementation is based on a byte-coded virtual machine named the OPAL Machine (OM). The OPAL compiler generates OM

¹ In the logic programming literature the notation \mathbf{p}/\mathbf{n} denotes a procedure named \mathbf{p} with \mathbf{n} arguments.

Figure 2. Message Passing in the AND/OR Model

code for AND and OR processes from the clauses of the user program. It creates a code block for an AND process from the body of each nonunit clause in the program, and collects all the clause heads of a procedure into the code block for an OR process. At runtime, an instance of a process is represented by a state vector. When a process is scheduled for execution, the machine will install frequently used parts of the state vector in OM registers, branch to the code block, and execute the instructions in the block.

The compiler automatically creates parallel goals wherever it can. When a procedure is called, the resulting OR process will attempt to unify the call with every clause head, and each successful match with a nonunit clause leads to a new AND process. In AND-parallel goals, the compiler uses information about the sharing and independence of variables derived from an abstract interpreter to order the goals in a data dependency graph which will control the order of execution at run-time [12]. Two calls can be executed in parallel if they have no unbound variables in common, i.e. OPAL exploits independent AND parallelism.

Figure 3. Message Passing in OPAL

In the basic AND/OR model, a process communicates only with its parent or its immediate children. The processes and their communication channels define an AND/OR tree, as shown in Figure 2 for the first few process steps in the execution of the even length path program. Downward pointing arrows indicate start messages, and upward pointing arrows are success messages. Redo messages are sent backwards along success arcs, and fail messages flow backwards along the start arcs. The AND process in charge of solving the top level goal statement has created an OR process for `epath/2`. When a procedure contains unit clauses, the OR process performs all the unifications and reports the results of these unifications to its parent AND process; the calls to `arc/2` are examples of this situation. When a procedure has nonunit clauses, the OR process starts an AND process for each body. In the even path program, `epath/2` is defined by two clauses, and the OR process starts two descendant AND processes. It is possible for a procedure to be defined by a combination of unit and nonunit clauses, in which case the OR process would report the results from matching the unit clauses at the same time it starts descendant AND processes.

Communication patterns in OM improve the basic model by allowing a process to send a success message to an ancestor, bypassing intervening steps that merely pass the result back up the process tree. The actual message passing patterns used when OM executes the even length path program are shown in Figure 3. The general rule is that results of a procedure call are always sent directly to the AND process that makes the procedure call. In the message marked (1) in the figure, the leftmost OR process for `arc/2` sends its results to its parent as it would in the basic AND/OR model. The OR process marked (2) sends its results to the top level goal because once this process has succeeded, its parent is done. The OM message pattern requires two fewer messages to get the result back to the top level. The message labeled (3) shows that this pattern is recursive and that lower level calls can send results all the way to the top of the process tree.

The key to this more efficient message pattern in OM is the use of continuations. When a procedure call is made, the AND process making the call hands a success continuation to the new OR process. Each time the OR process or one of its descendants generates a result, it invokes the success continuation in order to pass the results back up the tree. The mechanisms for passing continuations and the cases where more efficient patterns are exploited will be discussed in detail in the next section.

3. Continuations in OPAL

There are two types of continuations in OPAL, *success continuations* and *failure continuations*. Both are similar to continuations in Scheme even though they are created implicitly and are not first class citizens. Every procedure call in OPAL defines one success continuation and one failure continuation. Both continuations are passed as hidden parameters in the procedure call, to be invoked when the called procedure succeeds or fails. If the called procedure is nondeterministic (capable of producing more than one set of outputs for any given input combination) then the success continuation will be invoked several times, once for each result. The failure continuation is invoked only once, either if the called procedure fails to produce any results or after it produces its last result.

As an example, consider the following clause:

$$p(X,Z) \text{ <- } q(X,Y) \ \& \ r(Y,Z) \ \& \ s(X,Y).$$

Suppose `X` is the input argument, `Z` is the output argument, and `q/2` is nondeterministic, i.e. for any given value of `X` it may produce several

different bindings for Y . When the AND process for the body of this clause calls $q/2$, it creates a success continuation that consists of the remaining two goals and an environment in which X has the value passed to $p/2$, Z is unbound, and Y is an input parameter. When $q/2$ or one of its descendants succeeds after unifying its second argument with a term t , it calls the success continuation with t as a parameter.

Since it is not known (and not knowable) in advance when and where $q/2$ will succeed, the success continuation is passed as a parameter to $q/2$ and will be invoked each time $q/2$ succeeds. For example, $q/2$ might be defined as follows:

```
q(a,b).
q(X,Y) <- t(X,Y).

t(a,c).
```

Given this definition, the success continuation will be invoked twice. One invocation occurs when the first clause for $q/2$ unifies its output argument with b . The second clause for $q/2$ passes the success continuation on to $t/2$ and the continuation is invoked when $t/2$ unifies its output argument with c . The mechanisms for forwarding success continuations will be explained below. Note that the two invocations of the success continuation can occur in any order; for example, the success message that carries the binding $Y=b$ may be hung up in traffic so the message with $Y=c$ is delivered first.

Every procedure call also implicitly defines a failure continuation. As described in [11] one of the distinguishing features of the OM virtual machine is that instead of handling all failures the same way, OM executes a specially constructed failure code sequence for each procedure call. This code sequence implements a “semi-intelligent” backtracking scheme which is more efficient than Prolog’s simple chronological backtracking. Consider again the clause

```
p(X,Z) <- q(X,Y) & r(Y,Z) & s(X,Y).
```

Since in this example bindings for Y are created by calls to $q/2$, any procedure that “consumes” Y should request new bindings from $q/2$. If $s/2$ fails, Prolog simply backtracks into $r/2$. In OPAL, however, the compiler constructs a failure continuation for $s/2$. If $s/2$ fails, the failure continuation is invoked to undo the binding for Y , cancel the execution of $r/2$ if it is still running, and wait until $q/2$ or one of its descendants invokes its success continuation with a new binding for Y .

A continuation ID (CID) is represented in OM by a pair (p, a) where p is a process state vector and a is a code address. CIDs are passed as parameters in messages and define how the operation represented by

Figure 4. OM Message Formats

the message will be handled. At a minimum each message has a TO continuation that specifies the recipient of the message. To initiate a message handling operation, the system extracts the TO continuation ID (p, a) from the message, installs process p as the current process by loading values from the state vector of p into OM machine registers, and then branches to code location a (this operation is explained in more detail in the next section). Other CIDs passed as parameters in a message specify how to respond to the message. Up to two additional CIDs can be supplied in each message.

In the remainder of this section we will define the formats of each basic message type and then show how the system manipulates the CID fields of messages in order to achieve the control patterns shown in Figure 3.

3.1. MESSAGE FORMATS

The contents of the four main types of messages in OPAL are shown in Figure 4. The first field in each message is a one-byte message type.

Start messages (type S) contain two additional continuation IDs and a set of arguments. The three shaded fields are the CIDs. The argument field contains pointers to binding environments and other parameters being passed to the new process. *SUCC* identifies the continuation to invoke if the new process succeeds. For example, if the new process is an OR process for a procedure such as `arc/2` from Figure 1, it will collect the bindings from the successful matches with the goal and send them in a success message to the success continuation. *FAIL* is the ID of the continuation that receives a message when the new process fails.

Success messages (type A) contain only one additional CID, which identifies a redo continuation. The redo continuation is the continuation that should be invoked if the receiver of the success message ever needs additional results. The argument field contains the results of the call, for example a pointer to the set of bindings made to variables passed as parameters in the procedure call.

Redo messages (type R) are simply signals to send another result. The TO continuation here is a process that succeeded once before and is now being asked to supply a new result. Conceivably a redo message could contain success and fail continuation IDs, just as a start message does, but in OPAL a process saves the success and fail continuations handed to it when it was started and uses them to respond to redo messages. Thus every success generated by a process goes to the same continuation, as specified in the start message, and the failure of the process is reported to the fail continuation whether the failure happens immediately or after some number of successes.

Fail messages (type F) are also very simple. There is no need for parameters or for a continuation that would respond to a failure; in fact, in the current implementation a process deallocates itself when it fails and there is nothing left to handle a response to a fail message.

3.2. BASIC CONTROL PATTERNS

When an AND process p_i calls a procedure, it creates a new OR process for the call and sends it a start message with success CID (p_i, sc) and failure CID (p_i, fc) . The code addresses sc and fc are unique locations within the code block of the AND process.

Figure 5 shows the two basic control patterns generated when a process p_0 needs to solve goal $p/1$. It starts OR process p_1 and sends it a start message. Case (a) arises when the procedure is defined only by unit clauses. If the OR process cannot match the passed parameters with the compiled parameters for any clause head, it sends a fail message to (p_0, fc) . If it succeeds in matching the heads of one or more unit clauses, it sends the results to (p_0, sc) .

In Figure 5b, the OR process matches the head of a nonunit clause, so it needs to create a new AND process p_2 to solve the body of the clause. The start message to p_2 has success continuation ID (p_0, sc) and failure continuation ID (p_1, fc) . Note that the fail continuation is in the OR process p_1 . The reason is that in general there can be many matching nonunit clauses and p_2 will have one sibling for each additional matching nonunit clause (for example the OR process for the call to `epath/2` in Figure 2 has two descendants, one for each clause matching the call). p_0 should be sent a fail message only after p_2 and

Figure 5. Basic Message Patterns in OPAL

all its siblings fail. The OR process keeps track of fail messages arriving at (p_1, fc) and invokes its own fail continuation, which has ID (p_0, fc) , when all its descendants have failed.

It is possible for a procedure to be defined by a combination of unit and nonunit clauses. In these situations, the OR process reports successes from the unit clauses at the same time it sets up AND processes for the nonunit clauses. OPAL exploits true “don’t care” nondeterminism: results may be returned in any order, and the calling process must be able to handle the results in the order they return. Note that all results, whether they come from the OR process or from processes lower in the tree, are all handled by the same success continuation, identified by (p_0, sc) .

In order for this scheme to work, a slight adjustment needs to be made to the basic AND/ OR model. In the abstract model, an OR process is in charge of routing messages one at a time to the calling AND process, and lower level AND processes return their successes to the OR process for handling. For the control scheme outlined here, the messages go directly to the calling AND process. Thus the queuing mechanism that receives messages and extracts them one at a time on demand must now be part of the AND process.

3.3. LAST CALL OPTIMIZATION IN OPAL

A crucial aspect of any high performance implementation of a symbolic programming language is the conversion of tail recursion into iteration. Warren showed how techniques used to optimize tail recursion in Prolog can be generalized to the last call in any clause, not just those that are immediately tail recursive [13]. The recent text by Kogge [9] provides a good introduction to last call optimization and other implementation techniques for logic programs.

In an AND-parallel system such as OPAL, which can execute calls in the body of a clause in parallel, there may be no “last call” and it may not be possible to turn tail recursion into iteration. As an example, consider the following clause:

$$p(X,Y,Z) \text{ <- } q(X,Y) \ \& \ r(X,Z) .$$

Here X is an input parameter, Y and Z are outputs, and the calls to $q/2$ and $r/2$ can be executed in parallel. The AND process for the body of this clause must be a synchronization barrier: it must wait until both calls succeed before invoking its own success continuation, and neither call is a “last call” that can send a success message directly to an ancestor goal.

This section explains how the OPAL compiler generates parallel goals with no last calls for clauses such as the above example, yet is able to exploit tail recursion and last call optimization for `append/3` and similar procedures.

There are two separate aspects to last call optimization in a non-deterministic language. The first is space optimization, in which the space for a tail-recursive call overlays the space used in the previous call so the procedure runs in constant space. The second is the control optimization, implemented in OPAL by continuations, which avoids returning to a procedure body after calling the last goal.

The distinction between control and space optimizations can be seen in the following goal statement and clause:

$$\dots p(a,X) \ \& \ s(X), \dots$$

$$p(Y,Z) \text{ <- } q(Y,W) \ \& \ \dots \ r(W,Z) .$$

The last thing to do in a call to $p/2$ is to call $r/2$. The control optimization is to have the code for $r/2$ branch directly to the program point between the calls to $p/2$ and $s/1$ instead of returning to the clause for $p/2$. The space optimization is to reclaim the environment of $p/2$ before calling $r/2$. If this clause is tail recursive, i.e. the last call

is to $p/2$ itself and not $r/2$, the space optimization is what converts recursion to iteration.

The control and space optimizations are independent in a non-deterministic program since there are situations where it may be possible to exploit the control optimization without reclaiming the space used by the call. Consider the following goal statement and procedure definition:

```

... p(X,Y) & r(Y,Z) ...

p(a,a).
p(W,Z) <- q(W,X) & ... p(Y,Z).

```

At first glance the second clause for $p/2$ defines an iterative loop. However, if $q/2$ is nondeterministic, the state of that call must be preserved in order to allow backtracking or some other mechanism to extract the additional results. In this case the recursive calls consume progressively more space since they cannot reuse their parents' environments. The control optimization is still possible, though; when the base case of the recursion is activated, control passes immediately to the call to $r/2$.

In order to exploit the space optimization, which turns $p/2$ into an iterative procedure, Prolog programmers often use the cut operation to ensure no unwanted “choice points” are on the stack. The recursive call in this version of the clause (rewritten in Prolog syntax) is guaranteed to run in constant space:

```
p(W,Z) :- q(W,X), ... !, p(Y,Z).
```

OPAL programs are able to exploit the control optimization that results from last calls and tail recursion by having an AND process forward its success continuation to the OR process for a last call. For example, in

```

... p(X,Y) & s(Y) ...

p(X,Z) <- q(X,Y) & r(Y,Z).

```

the compiler will classify the call to $r/2$ as a last call. Instead of creating a new success continuation for the call to $r/2$, the AND process for $p/2$ will forward the success continuation it was handed by the top level call. When $r/2$ succeeds the result will be passed directly back to the top level to be used in the call to $s/1$.

The OPAL compiler arranges the goals in the body of a clause in a data dependency graph and then generates code that creates calls according to the constraints defined by the graph [12]. The structure of the data dependency graph is the key to whether or not a clause has a last call: *a procedure call in OPAL is a last call if and only if the call is a single node on the last level of the data dependency graph for the clause.*

Figure 6 shows two example clauses and the corresponding data dependency graphs. In the first example, the compiler has determined (from examining the rest of the program, to see how $p/2$ is called) that X and Y will always be unbound when the procedure is called. The graph

Figure 6. Data Dependency Graphs for Two Clauses

shows the two body goals are independent and can be solved in parallel since they are at the same level in the graph. In the second example, X will be an input term, and Z will be output. No matter what the parameters are, the two body goals will always have unbound variable Y in common, so they are not independent. In this graph, the compiler has determined that $q/2$ will be called first, and after it has been solved $r/2$ is called.

In the first procedure, there is no last call, and the relationship between the AND process for this clause and its two OR descendants will be the same as that shown in Figure 5a. The AND process set up for the body of this procedure cannot pass its own success continuation to either of the body goals because it must wait for both to be solved, and it does not know which will succeed first.

The second procedure in Figure 6 does have a last call, since $r/2$ is the only goal on the last level of the data dependency graph. Once $r/2$ is solved, the entire clause is solved, so the success continuation passed to $p/2$ can be passed on to $r/2$. Figure 7 shows the AND/OR tree that will result. Note that the OR process for $r/2$ passes its results directly back to the top level AND process.

In order to implement the last call optimization, the start message sent to the OR process for a last call uses a different set of continuations

Figure 7. Last Call Optimization in OPAL

than other calls made by an AND process. Suppose, as in Figure 7, an AND process p_2 is sending a start message to a new OR process for a call to $r/2$. Part of the state vector for p_2 are the two continuation IDs (p_0, sc_0) and (p_1, fc_1) given to it by its parent. As defined in the previous section, if the new OR process is not a last call, the success and failure CIDs in the start message are (p_2, sc) and (p_2, fc) , respectively, and p_2 will handle the success messages from the call to $r/2$. However, if the new OR process is a last call, the two CIDs passed to $r/2$ are (p_0, sc) and (p_2, fc) . In other words, the OR process for a last call and all its descendants will send their results back to p_2 's ancestor p_0 instead of to p_2 itself.

When the compiler decides that a goal can be solved with the last call control optimization, it generates a `start_last_or` instruction instead of a `start_or` instruction. Both virtual instructions trap to the kernel to create a new process and send it a start message; they just differ in the value of the success CID put into the start message.

Note that fail messages from the new OR process for $r/2$ will come back to AND process p_2 . To implement the space saving dimension of last call optimization, with immediate garbage collection of previous goals, p_2 would have to send (p_1, fc_1) , or better yet (p_0, fc_0) , as the failure CID to the new OR process. However, in order to do this p_2

Table I. Basic Control vs. Continuation Control

	color		epath		append	
	<i>basic</i>	<i>cont</i>	<i>basic</i>	<i>cont</i>	<i>basic</i>	<i>cont</i>
<i>#procedures</i>	876	876	375	375	54	54
<i>#messages</i>	2647	2137	1584	974	214	108
<i>#instructions</i>	53170	51719	18522	15977	1600	858

would have to be sure there were no more successes possible from $q/2$. In Prolog this is a simple decision, since control frames are allocated from a stack and it is easy to see if there are any choice points on the stack. In OPAL, however, we would have to tag success messages with information that indicates whether or not more results are possible; this is the subject of a current project.

3.4. MEASUREMENTS

To measure the effects of continuation-based control patterns, we counted the number of messages generated by two versions of the OPAL machine, one with the straightforward control regime defined by the abstract model and the other with the optimized control strategy enabled by using continuations. The results are shown in Table I.

The three programs used to collect the data were a map coloring program (72 solutions), the program that searched for even length paths in a graph (51 solutions), and a program that made 26 recursive calls to the deterministic append procedure. We recorded the number of procedure calls, the number of messages generated, and the total number of virtual machine instructions executed for each program. Two numbers shown for each program are the values recorded using the basic control strategy and the continuation based control.

As expected, the total number of procedure calls is unchanged since the same process tree is created for each program. However, the number of messages goes down. Map coloring is a highly nondeterministic program, and the main AND process does not allow last call optimization. The small improvement in the number of messages is a result of the fact

that OR processes do not handle success messages. In the even path program, one clause in the procedure `epath/2` is tail recursive, and the reduction in messages is more pronounced since the tail recursive calls end up sending their results directly to the top level goal. `append` is completely deterministic. The AND/OR graph for this program is a single long branch with two nodes (one for the OR process and one for the AND process for the body) for each call to `append/3`. As expected, the number of messages is cut in half since the result is returned straight to the top level without bubbling back up the tree. If we could tag success messages to show no more results are possible, and achieve constant-space iteration, the number of messages would be cut in half again since fail messages would not have to bubble back up the tree.

4. Implementation

The use of continuations in message passing not only enables more efficient control strategies, it is the key to highly efficient task switching in OPAL. In this section we will describe how continuation IDs are represented and how the system kernel uses these IDs to effect a task switch. Timing data from a simulator that builds and traverses random AND/OR trees is presented in the final section.

4.1. REPRESENTATIONS

From the beginning, one of the main goals for OM was an efficient implementation of parallel logic programs for scalable (i.e. distributed memory) parallel processors. In a parallel system, the AND/OR tree of process states will be spread across the local memories of the nodes in the system. In Section 3 we defined a continuation ID to be a pair (p, a) where p is a process ID and a is a code address. An efficient representation of a CID thus depends on representations of p and a in a distributed memory.

One assumption we made was that every node would have the same copy of the compiled user program, so the code address a can simply be the offset from the beginning of the code space. This allows us to create a CID with the address of the procedure that will handle the message without worrying where the code for a resides on the processor that will receive the message – it is at the same location in the receiving node as it is in the transmitting node. For example, when an AND process wants to solve a procedure `p/n`, it would send a start message containing a TO CID of the form $(-, a)$ where a is the address of the first instruction in the procedure. The code address will be valid no matter which node eventually executes the procedure.

The process ID p is more complicated. A key to the representation is that once a process begins executing it does not move around in the system. When a process is first created, its state vector is a “seed” that is little more than the location of the first instruction in the code for the process. The first instruction in the code block for a process causes the seed to “sprout” and be expanded into a full process state. The task allocation mechanisms are all allowed to send seeds to other nodes in order to balance the load, but none can relocate a full process state vector.

Since a process does not move, the system can use the local heap address of the state vector in the representation of the process ID. Currently a process ID is a two-word record containing the ID of the processor that “owns” the process, the status of the process (seed, active, terminated, etc.), and the local address of the state vector.

The control information in a CID is thus a tuple $\langle\langle I, HA \rangle, C\rangle$ where I is the ID of the processor that will handle the message, HA is the heap address of the state vector of the receiving process in the local memory of processor I , and C is a constant used to compute the code address of a block of instructions that will be executed in order to handle the message.

4.2. TASK SWITCHING

During the execution of a user program, the registers of the OM virtual machine contain information about one process, known as the “current process.” When the last OM instruction in the current process causes a task switch, the system traps to the underlying kernel. The kernel loads a new process state into the virtual machine registers and then returns control to the virtual machine level.

When a process sends a message, a virtual machine instruction executes a trap which invokes the kernel. The kernel examines the TO continuation ID in the message. If the PID field of the continuation ID equals the processor’s ID, the message is inserted into the local message queue, otherwise the message is handed over to the interprocessor message router.

The three registers updated during a task switch are M , P and PC . The message register M holds a pointer to the message that activated the current process. The process register P holds a pointer to the state vector of the current process. PC is the usual program counter register, containing the code address of the next virtual machine instruction to execute. For registers such as M and P that point to complex structures, the notation $R[X]$ refers to field X of the structure pointed to by register R .

The three steps executed by the kernel to perform a task switch are:

```

M := next_message;
P := M[to[HA]];
PC := P[code] + M[type];

```

The call to `next_message` removes a message from the local message queue; if the queue is empty, it waits until a message arrives from another node. The second step sets the `P` register to the ID of the process specified in the `TO CID`. The third step uses `P` to find the address of the first virtual instruction in the code block for the process.

The first four instructions in each process are “port instructions” that are always arranged in the same order. The third step above causes the virtual machine to branch to the port that will handle the given message type. The port instruction will carry out the final step of using the `C` field of the `TO CID` to branch to the code for the program point that will handle the message. For example, the `and_success_port` instruction is a multi-way branch instruction that uses a combination of the `C` field and the state vector to branch to the success continuation code for the procedure that was just solved.

4.3. MEASUREMENTS

The steps outlined in the previous section are compiled into very few host machine instructions, almost all of which are simple indexed loads, so task switching should be very efficient. To measure the efficiency of the representations and the task switching code, we wrote a simple program that grows a random AND/OR tree. Parameters of the simulation are the branching factor for each type of node, the maximum depth of the tree, and the probability (which decreases with depth) that an OR node will become the root of the next round of expansion. The simulator was used to exercise the heap-based memory manager as well as the message handling and task switching logic.

When run on an HP 9000/835 workstation, the simulator processed between 75,000 and 98,000 messages per second. Included in this figure is the time to allocate and deallocate the nodes of the tree, do “process updates” to figure out where and how to expand the next node, and generate messages to carry out the next expansion. We could not measure the actual distribution of CPU time between task switching, process updates, message queue operations, and memory management since the procedures were too small for accurate measurement by statistical profiling tools that sample the host machine program counter at periodic intervals. We estimate that roughly 1/10th of the time spent in the simulator is used by the task switcher.

We can draw two conclusions from the figure of almost 100,000 messages per second. First, at roughly 10 microseconds per node expansion step, the continuation based control is quite efficient. If 1/10th of the time in each simulated step is devoted to task switching, it takes on the order of 1 microsecond to remove a message from the queue and update the M, P and PC registers.

Second, this simulator provides a reasonable estimate of the overhead incurred by executing programs according to the AND/OR process model. With the current implementation using two processes per procedure call in the append benchmark, we have an upper bound of 50,000 LIPS, which is over two times slower than SICStus Prolog on this machine. A big improvement will be seen if we can cut the number of messages down to one per call through the use of redo continuations, both because there will be fewer messages and because the program will take less space and may even execute entirely in cache. Further improvements will come from compiler and runtime optimizations that increase the granularity of tasks or new, more innovative, message passing patterns.

Measurements of actual programs show that in some nondeterministic programs OPAL is fairly close to SICStus in execution times on a single processor. The best so far is a program that computes the opening bid for a hand in Bridge; this program is only 1.6 times slower. Other nondeterministic programs are from four to five times slower, and deterministic programs such as naive reverse and quicksort are around ten times slower. A more detailed discussion of these measurements and of where we expect to improve OM can be found in [5].

5. Summary and Discussion

This paper has presented an application of continuations in the development of efficient control structures for parallel logic programs that have an Actors style operational semantics. By viewing process states as continuations and passing two continuation IDs – one for success and one for failure – to each new process, message passing can be reduced by up to 50%.

Our use of continuations is similar in spirit to the Mach 3.0 kernel of Draves, et al [6]. Their goal was to redesign the implementation of certain kernel operations without modifying the specifications of the procedures; from the user's point of view, nothing changed but performance. Our goal was the same, since we wanted to implement the basic operational semantics of the AND/OR process model using fewer messages and more efficient task switches.

A continuation is usually defined to be a function plus an environment, an object that can be activated in order to resume execution in a known state. This definition applies to OPAL continuations as well, even though we did not discuss in this paper the environment portion of OPAL continuations or the mechanism for passing parameters to continuations. A message contains all the binding information needed in the resumption of the clause, in the form of a set of closed frames. The handling of these frames is discussed in detail in [5].

Since the continuations represent independent functions, and their environments are captured completely in the closed frames, it is possible to invoke a continuation more than once. For example, if an OR process unifies its call with the heads of n unit clauses, it will eventually invoke its success continuation n times. In OPAL, all n solutions are sent on to the calling AND process in one message, but the AND process uses them one at a time. Conceptually, we could fork n independent invocations of the success continuation to run in parallel. This is in fact what happens in a pure OR parallel system such as Aurora [10] or Muse [2], which build a “cactus stack” of bindings in a shared memory space; an interior node with n branches is the stack frame for a call with n successful matches. Kalé’s Reduce-OR process model is an example of a process oriented system, similar to the AND/OR model, that can operate on all results simultaneously [8]. Kalé calls this form of AND parallelism “consumer instance parallelism”, reflecting the fact that one invocation of the continuation (the consumer) is made for each result sent.

We have experimented with implementing consumer instance parallelism in OPAL, but there are limited situations when it can be applied without major changes to the underlying model. If an AND process forks two or more OR processes at the same time, it becomes the success continuation for all of the processes and their descendants. Before the AND process can start goals in the next level of the data dependency graph, however, it must receive solutions from each branch, and the results may arrive in any order. We cannot simply make a copy of the AND process when a result arrives, because the result must be matched with results from sibling goals. Kalé’s system implements a join operation that is able to create multiple instances of the success continuation each time a new result arrives on one of the input channels.

Our immediate plans are to implement more intelligent redo continuations. Currently the redo continuation in a success message is the ID of the process that creates the result. By tagging success messages with a field that indicates no more results are possible, we should be able to avoid sending redo messages to processes that cannot generate any more successes. As described earlier, in purely deterministic programs

this new optimization will cut the number of message in half again, for a total savings of 75% over the basic AND/OR model.

It is clear that viewing sending and receiving processes as continuations has been the key conceptual framework that led to the improvements so far. We hope that as we gain experience with more and larger user programs we will be able to uncover additional optimizations.

References

1. Agha, G. and Hewitt, C. Actors: A conceptual framework for concurrent object-oriented programming. In Shriver, B. and Wegner, P., editors, *Research Directions in Object-Oriented Computing*, MIT Press, Cambridge (1987) 49–74.
2. Ali, Khayri A. M. and Karlsson, Roland. The Muse Or-Parallel Prolog model and its performance. In *Proceedings of the 1990 North American Conference on Logic Programming* (1990) 757–776.
3. Clinger, W. and Rees, J. *The Revised⁴ Report on the Algorithmic Language Scheme*. Tech. Rep. CIS-91-25, University of Oregon (February 1992). See also *IEEE Standard 1178-1990*.
4. Conery, J. S. *Parallel Execution of Logic Programs*. Kluwer Academic Publishers, Boston, MA (1987).
5. Conery, J. S. The OPAL machine. In Kacksuk, P. and Wise, M., editors, *Implementations of Distributed Prolog*, John Wiley and Sons, Ltd., London (1992) 159–185.
6. Draves, R. P., Bershad, B. N., Rashid, R. F., and Dean, R. W. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (1991) 122–136.
7. Hausman, B. *Pruning and Speculative Work in OR-Parallel Prolog*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden (1990). Report TRITA-CS-9002.
8. Kalé, L. V. and Ramkumar, B. The Reduce-OR process model for parallel logic programming on non-shared memory machines. In Kacksuk, P. and Wise, M., editors, *Implementations of Distributed Prolog*, John Wiley and Sons, Ltd., London (1992) 187–212.
9. Kogge, P. M. *The Architecture of Symbolic Computers*. McGraw-Hill (1991).
10. Lusk, E. L. *et al.* The Aurora OR-parallel Prolog system. In *Proceedings of the International Conference on Fifth Generation Computer Systems* (1988) 819–830.
11. Meyer, D. M. and Conery, J. S. Architected failure handling for AND-parallel logic programs. In *Proceedings of the Seventh International Conference on Logic Programming* (1990) 633–653.
12. Sundararajan, R. and Conery, J. S. An abstract interpretation scheme for groundness, fairness, and sharing analysis of logic programs. In *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science* (1992).
13. Warren, D. H. D. *An Abstract Prolog Instruction Set*. Tech. Note 309, SRI International (October 1983).

