Parallel Logic Programs on the Mayfly

John S. Conery (conery@cs.uoregon.edu) [*]
Department of Computer and Information Science
University of Oregon

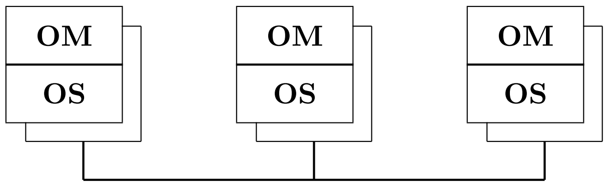
Abstract. The Mayfly, a parallel processor being built at HP Labs in Palo Alto, has architectural support for several important aspects of the OM virtual machine for parallel logic programs. Each node has an extra processor that is able to relieve the main processor of a significant amount of the "housekeeping" work of memory management, task switching, and message handling. This paper describes how the second processor implements kernel level functions in OM, with particular attention to the operations that support task switching and task allocation. The paper includes detailed timing data from a program with interleaved parallel threads to show that while the main processor is busy in one thread the kernel processor can effectively build the context for the next thread, significantly reducing task switching time.

1. Introduction

OPAL, the *O*regon *PA*rallel *L*ogic language, is based on the AND/OR Process Model, an abstract model for parallel logic programs with an operational semantics defined by asynchronous objects that communicate solely via messages. This paper describes an implementation of OPAL on the Mayfly, a parallel processor for symbolic computation currently being developed at Hewlett-Packard Laboratories in Palo Alto [4]. Each node of the machine contains special purpose hardware that provides significant support for an object-oriented, message-passing style of computation, so the machine should provide an ideal execution environment for OPAL programs.

The main focus of this paper is on operations within a single node of the system and how the hardware that supports fast task switching and efficient message handling implements these low level operations in OPAL. Our main concern is in using one of the two general purpose HP-PA RISC chips in each node as a *kernel coprocessor*, offloading as much overhead as possible from the main logic program processor. The kernel processor is used to build contexts for the main processor, which reduces the overall task switching time and allows the main processor to spend more of its time in the user program. It also implements dynamic task allocation via "hooks" installed in various kernel functions.

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



Each PE has its own copy of the virtual machine (OM) and the OS. All host-dependent operations, including interprocessor communication, are done by the kernel.

Figure 1. Modules in an OPAL Implementation

The paper begins with an overview of the OPAL language and how it is implemented on systems in general. Following that are sections that explain how the critical low level operations – task allocation and task switching – are implemented on the Mayfly. In the final section we list some interesting future projects that are enabled by this machine.

2. The OPAL Language and Virtual Machine

Figure 1 shows the software modules in OPAL. When implemented on a multiprocessor such as the Mayfly, we make a complete copy of each module for each PE. The top layer is the OPAL machine (OM), a byte-code interpreter that executes compiled user programs. The registers, instructions, and other aspects of the machine are all concerned with the execution of a single process, known as the *current process*. The OS module implements inter-process operations; for example, if the current process executes an instruction that sends a message, it causes a trap to the OS level which creates the message and appends it to a queue.

The OM and OS modules are machine-independent. Supporting them both is a host-dependent kernel which implements memory management, inter-processor message passing, and any other functions that depend on the architecture of that particular host. To continue the example, the OS will check to see if the recipient of the new new message is on the same PE; if so, it appends it to the local message queue, otherwise it traps to a kernel routine that handles the message. On the Mayfly, the kernel breaks the message into packets and sends them to the Post Office for transmission to the other PEs.

The following sections give some background information on the OPAL language, its execution model, and the OM virtual machine in preparation for the main part of the paper, which explains how the OPAL kernel for the Mayfly implements task switching and task allocation.

2.1. The Execution Model

OPAL programs are executed according to the AND/OR Process Model, an abstract framework for parallel logic programs [2]. Under this model, programs are collections of asynchronous objects communicating via messages. A process will update its internal state only when it receives a message from another process, and process updates are nonpreemptible operations.

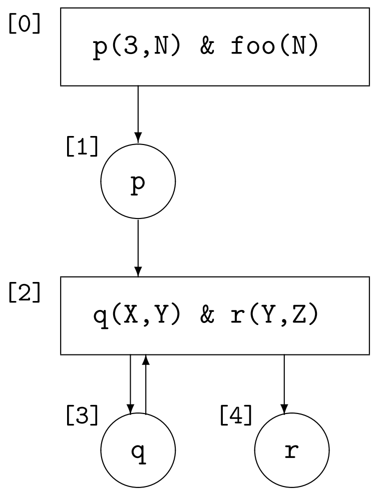
AND processes in this model are charged with finding the solution to a goal statement, which is a collection of one or more goals from the body of a clause. A computation is started by creating a top level AND process for the user's query. If AND parallelism is exploited by the system, an AND process may solve more than one of its goals at a time by starting parallel descendant processes for those goals. The order in which goals are solved is determined by the compiler, which generates a data dependency graph that is used at runtime to coordinate the solution of body goals.

OR processes are created by AND processes to solve a single goal. An OR process performs the unifications of the goal with the heads of candidate clauses. If no heads unify, the goal fails. When a goal matches the head of a nonunit clause (a clause with one or more goals in the body), the OR process starts an AND process to solve the body. If the goal matches the head of a unit clause (an assertion), the OR process can send a success message to the calling AND process.

OPAL augments the basic AND/OR model through the use of *continuations*. We use this term the same way it is used in Actors models: a continuation is the ID of a process (actor) that will handle a response to a message [7]. Continuations can be passed as arguments in messages to allow processes to respond directly to processes other than their parents. Consider the following goal statement and clause:

```
        <- p(3,N) & foo(N).
 p(X,Z) <- q(X,Y) & r(Y,Z).
```

The top level goal will start an OR process to solve the call to `p` (see Figure 2). The unification of the call with the clause head will succeed, and the OR process will start an AND process for the body of the clause. Later, when that AND process receives a solution for `q`, it starts an OR process for `r`. In the pure AND/OR model, the OR process for `r` would send its response to its parent (process 2 in the figure). However, that AND process would just turn around and pass



When the call to `r(Y,Z)` *succeeds, process 4 can use a success continuation to respond directly to process 0 instead of process 2.*

Figure 2. Using Success Continuations

the result back up the process tree. It would be much more efficient to pass the ID of the top level goal as the *success continuation* in the call to `r`, so that if `r` succeeds it sends its results directly to the top level goal. Start messages carry two continuations, one for success and one for failure; currently, the failure continuation of each new process is its parent.

This use of success continuations allows the introduction of last call optimization, a generalization of tail recursion optimization in systems that use message passing control structures (see [8] for a description of tail recursion and last call optimization in logic programs).

2.2. The OPAL Language

The OPAL programming language is little more than 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 taking the functor of a term.[1]

Figure 3 shows three simple OPAL programs. The first is a trivial program that will be used to illustrate the detailed interaction between components of a Mayfly processor in Section 5. The second is an OR-

[1] Success and fail continuations are used only in the implementation, and are not constructs in the programming language.

```
% Program 1:  "small"

goal <- foo(A,B).

foo(X,Y) <- p(X) & q(Y).

p(0).
p(1).

q(a).
q(b).

% Program 2: "path".  The actual program has 16 arcs,
% leading to 51 solutions (only 3 arcs are shown here).

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).

% Program 3: "color".  There are 12 clauses for 'next' in the
% actual program (only 3 are shown here).

goal <- color(A,B,C,D,E).

color(A,B,C,D,E) <-
    next(A,B) & next(C,D) & next(A,C) & next(A,D) &
    next(B,C) & next(B,E) & next(C,E) & next(D,E).

next(green,yellow).
next(green,blue).
next(green,red).
```

Figure 3. Example OPAL Programs

parallel program that searches for all even-length paths in an acyclic graph. The parallelism is a form of pipelining, where the system is searching for longer paths (via the second clause in the procedure for `ep`) while shorter paths are being reported as solutions. The third is an AND-parallel program that generates all possible colorings of a map with five regions and eight interior borders. The parallelism comes from testing the validity of some colorings in parallel (the predicate `next` colors a region if the name of the region is unbound in a call, and checks the coloring of two regions if both parameters are bound). Since the tests are very simple, there is actually very little parallelism here; this program is mainly used to test the system's execution of nondeterministic AND-parallel programs.

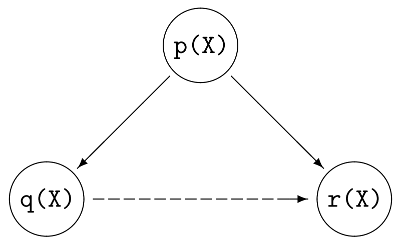
The OPAL compiler automatically creates OR-parallel goals wherever it can. When a procedure is called, the resulting OR process will attempt to unify the call with every clause head. Each successful match with a nonunit clause leads to a new AND process that will run in parallel with siblings created for other clauses in this procedure.

The compiler also exploits AND parallelism automatically. It builds a data dependency graph using information about variable bindings extracted from an abstract interpreter. The data dependency graph, which is implicit in the structure of the compiled code for the AND process, controls the order of execution of body goals at run-time. Two goals are connected if they have a variable in common and solution of one goal creates bindings used by the other goal. The bodies of the clauses for `epath` in Program 2 of Figure 3 are examples of data dependencies. In this case, the call to `arc(B,C)` depends on the value of `B` created in the call to `arc(A,B)`, so the goals are executed sequentially. The goals in the body of the procedure for `foo` in Program 1 are independent since they have no variables in common; the calls to `p(X)` and `q(Y)` will be executed in parallel.

Sometimes data dependencies are dynamic. Consider the following three goals in the body of a clause:

```
  <- ... p(X) & q(X) & r(X) ...
```

If the compiler determines that every solution to `p` will bind `X` to a ground term (a term with no unbound variables) then the data dependency graph will have only two arcs, one connecting `p` to `q` and the other connecting `p` to `r` (Figure 4). After `p(X)` succeeds, `q(X)` and `r(X)` would be solved in parallel. However, if some solution to `p` binds `X` to a term containing another variable, this solution introduces a dependency between the calls to `q` and `r`, and the call to `r` will have to be postponed until `q` is solved. The arc connecting `q` to `r` is conditional, and depends on the binding produced by the call to `p(X)`. The analyses performed by



Data dependency graph for the goal `<- p(X) & q(X) & r(X).`

Figure 4. Data Dependency Graphs

the compiler and the instructions used to handle dynamic dependencies are defined in a recent paper by Meyer and Conery [10].

I/O and a few other complex operations are implemented on the front-end host machine, but they are "cavalier" operations and no attempt is made to serialize programs through these constructs (*c.f.* [5]). For example, the calls to `write` in the bodies of the clauses in this procedure are executed asynchronously:

```
foo(N) <- bar(N,M) & write(M).
foo(N) <- write(M).
```

Aurora (an OR-parallel version of the WAM [1].) guarantees that side-effect predicates will be executed in the same order as they would in Prolog; in this example, the call to `write` in the second clause would be blocked until the call to `write` in the first clause is done, and the user would see the results printed in the same order as in Prolog. In OPAL, however, the results would be sent to the host processor in a random order by the AND processes for the bodies of the two clauses.

2.3. The OM Virtual Machine

The OPAL compiler generates a code sequence for an AND process from each clause body, and collects the heads of clauses in each procedure to generate the code for an OR process for that procedure. At the front of the block of code for a process is a set of *port* instructions which act as branches into the rest of the body. When the system delivers a message to a process, it installs the process state in the virtual machine registers and branches to the port that handles that type of message. The port instruction then branches to a location that is a function of the contents of the message and the state of the process. For example, the `and_fail_port` instruction examines the message to find out which body goal failed, and then branches to a section of code that determines which other processes need to be sent redo messages in order to get different bindings for the variables in the failed goal.

OM is faintly reminiscent of the Warren Abstract Machine (WAM) for standard Prolog implementations [12]. One of the main differences between OM and WAM is in the representation of variable bindings. Most logic programming systems, including WAM and OM, represent a logical variable by a cell that contains a pointer to (address of) the variable's value. Unbound variables are pointers to themselves. When two unbound variables are unified, one is bound to a pointer to the other. A situation where this happens is when an unbound variable is passed as a parameter to a procedure:

```
<- ... p(X) ...
p(Z) <- ... q(Z) ...
```

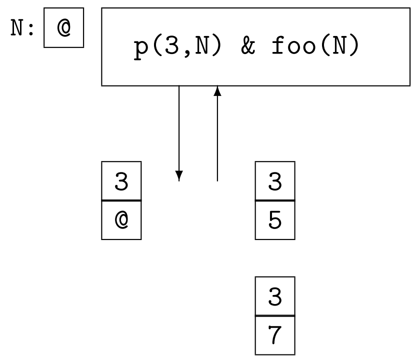
If `X` is unbound when `p` is called, the variable `Z` in the environment of the called procedure is bound to a pointer to the variable `X` from the environment of the call.

When a variable is passed unbound through several levels of calls in a Prolog program, the resulting stack will contain a pointer from the top back to a cell nearer the bottom. This leads to two problems in OR-parallel systems. First, OR-parallel calls may try to bind the same cell to conflicting bindings. To continue the example above, `q` may be defined by two clauses:

```
q(a) <- ...
q(b) <- ...
```

An OR-parallel system should be able to execute the bodies of both clauses in parallel, and in order to do that it must be able to unify `Z` (and thus `X`) with the constant occurring in the corresponding clause head. There are many effective solutions to this problem based on giving each OR-parallel thread a virtual copy of unbound variables deep in the stack [13].

The second problem is that the two unified variables (`X` and `Z` in this case) may be in environments that are allocated in two different nodes of a nonshared memory multiprocessor. Of course the problem is finessed in a shared memory multiprocessor, for example Aurora on the the Sequent Symmetry [1]. However, we intend to run OPAL on large-scale nonshared memory multiprocessors, and there will be situations where a processor needs to refer to a value stored in another processor's memory. The AND process for the body of the clause for `p` might run on a different node than the AND process for the top level goal, and we do not want to have to make a nonlocal memory reference each time we need the value of `Z`.



An AND process with one variable. When `p(3,N)` *is called,* `N` *is unbound. Solutions returned to the AND process are copies of the argument registers. Environment closing instructions will extract bindings for* `N` *from the returned argument frames.*

Figure 5. Message Arguments

OM is based on a representation, known as *closed environments*, that addresses both problems – shared ancestor variables and nonlocal accesses – by copying nonground terms [3]. When an unbound variable is passed as a parameter, the parent's copy of the variable is bound to a "stub" that refers to an argument register. If a nonground complex term, for example a list, is passed as a parameter, descendant processes copy the terms, fill in their copies with results, and pass the copies back to the calling goal. Results are passed back as filled-in argument registers, and when the parent dereferences the stub it will find the value created by the descendant that solved the goal (see Figure 5). The caller then extract bindings from the returned copy. Although there is overhead from copying terms and extracting results, there are benefits as well: this scheme is easy to implement in nonshared memory systems with partitioned address spaces, and since there is no need to maintain stacks (with concomitant worries over buried goals [6]) we can use a heap-based memory allocator and let each node do local garbage collection. Eventually we hope to demonstrate that the copying overhead is not much worse than the overhead in Aurora and other systems that provide virtual copies of unbound ancestor variables.

The other major difference between OM and the WAM is that OM uses asynchronous parallel control instructions. In the WAM, once the argument registers are filled with parameters to a called procedure, a `call` instruction invokes the procedure. The called procedure then adds structures to the stack, if required, and solves the goal; if it is successful, control resumes in the calling procedure at the point where the call was made. In OM, procedures are invoked by an asynchronous `start_or` instruction which directs the system to create a new process object and send it a start message containing the argument registers as parameters. Control remains with the AND process, however, since it may want to start up several OR processes during an AND-parallel execution. An AND process gives up control when it executes a control instruction that causes a task switch. For example, the operand of `check_solved` is a bit-encoded set of goal indices; if this set is not a subset of the goals that have been solved, a task switch occurs.

Code blocks for OR processes also have asynchronous control instructions. The last `get` instruction in the head of a nonunit clause is followed by a `start_and` instruction. This creates a new AND process for the body of the clause, and a start message is added to the system message queue. The machine continues execution of the OR process, which begins unification of the next clause. OR-parallelism will be exploited in programs that have more than one head that matches a call. The OR process switches out after finishing (successfully or not) the unification of the last clause head in the procedure.

The control instruction that follows the last argument of a unit clause is `proceed`, which is analogous to the same instruction in the WAM since it invokes the current success continuation. In OM, this is done by adding the argument registers (which, if they contained unbound variables at the beginning of the unification now have output bindings) to the set of results being generated for this call. After the last clause head has been tried, the OR process returns this set to its success continuation, which is an AND process that will use the results one at a time.

Figure 6 shows part of the compiled code for the example program named `small`. It shows the port instructions at the beginning of the code segment for two processes, the asynchronous control instructions that start processes, and unification instructions in the code for one of the OR processes.

2.4. The Kernel and System Layers

Perhaps the most important set of functions implemented in the OS and kernel are concerned with task switching. Processes are the fundamental building blocks in OM, and for it to be anywhere near as efficient as a Prolog implementation the system must be able to switch from one process to another about as quickly as Prolog can make a procedure call. Consider the operations that take place between the call to `p` and

```
foo2a1:   and_fail_port
          and_redo_port     foo2a1_Redo
          and_success_port
          and_start_port    3
          make_args         1, 1
          put_val           1, 1
          start_or          p1, p1_SC07, p1_FC08
          check_indep_else  [1,2], 0, foo2a1_L09
          make_args         2, 1
          put_val           2, 1
          start_or          q1, q1_SC0B, q1_FC0C
          check_solved      [1,2]

q1_SC0B:  cget_val          2, 1
          set_solved        2
          check_solved      [1,2]
          succeed           1

q1:       or_fail_port
          or_redo_port
          or_success_port
          or_start_port
q1o2:     next_alternative  q1o1
          store_args
          get_const         b, 1
          proceed
q1o1:     last_alternative
          restore_args
          get_const         a, 1
          proceed
```

Part of the compiled code for `small`*. The first two sections are part of the "forward execution" code for the AND process for the body of* `foo`*. The last section is the OR process for* `q`*.*

Figure 6. Code Block

the execution of the first step in the body of the procedure in this example:

```
?- ... p(a,X) & s(X) ...

p(U,V) <- q(U,W) ...
```

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

A Prolog system executes a procedure call instruction, which builds a choice point (since there are two alternatives for `p`) and then starts unification in the first clause head. If the patterns match, the machine continues execution in the body of the first clause. When the last goal in the body of that clause is solved, execution resumes in the top level goal where the system sets up the call to `s` using the values for `X` created during the call to `p`. Later on, if the first clause fails, the machine will use the choice point to restore the state of variables as they were before the first unification was made, and then start unifying the second clause head.

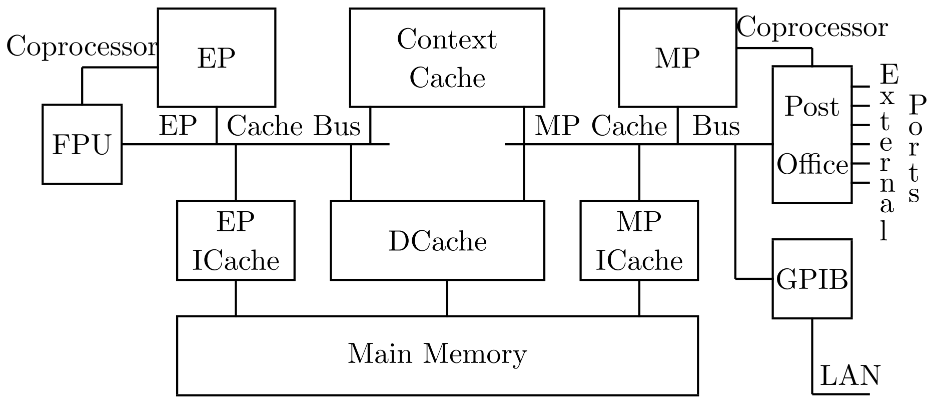
When OM makes its equivalent of a procedure call, it must build an OR process. Some time later, a task switch will install the OR process as the current process. It will perform both unifications and create two new AND processes for the bodies, and then switch out. When one of the new AND processes is at the front of the queue, another task switch occurs and execution resumes in the body of that clause. Finally, when results are sent back to the AND process that made the call, there is a third task switch, and the value of `X` returned by the call is extracted from the argument registers. Thus where Prolog does one procedure call and return OM must do up to three task switches plus the extra work of extracting a binding. For deterministic procedures, such as `append`, OM will make two process switches for every recursive procedure call – one for the OR process and one for the AND process – and then one final switch to return the result.

Because efficient task switching is such a critical operation, careful attention was paid to the representation of process states in the definition of OM and its OS. Later in Section 5 the basic task switching mechanisms common to all implementations will be explained, along with a description of how we use the Mayfly hardware to improve on the generic task switching operations.

3. Overview of OPAL on the Mayfly

The hardware structure of a single Mayfly processing element (PE) is discussed in another article in this issue [4]. Figure 7 is a portion of a figure from that article that shows those parts of the PE that play the most significant roles in the implementation of OPAL.

The most interesting aspect of the Mayfly as far as the OPAL project is concerned is the fact that there are two independent HP-PA RISC processor chips per PE. The division of labor used to implement OPAL on the Mayfly is to use the execution processor (EP) as the processor



The main components and data path within a single PE of the Mayfly (from [4]).

Figure 7. Structure of a Mayfly PE

that runs user programs, performing unifications and executing control instructions in OM, and to use the message processor (MP) as a kernel coprocessor which tries to offload the low level overhead from the EP.

Two basic strategies characterize this first implementation of OPAL on the Mayfly. First, we moved a minimum number of operations to the MP. Even though execution profiles of a single processor executing an entire OPAL program show that the MP is underutilized compared to the EP, this could change drastically depending on the task allocation methods used in multiprocessor executions. The MP will be interrupted whenever packets containing tasks newly allocated to this PE or messages destined to processes already running on this PE arrive from the Post Office.

The second basic strategy is to implement one Mayfly task per OPAL process. In other words there is a one-to-one mapping of AND and OR processes to Mayfly contexts, and a single process step in the AND/OR model is implemented by the EP dequeueing a context, performing the step defined by the message that triggered the step, and then sending descriptions of new messages and processes back to the MP in an output context. This strategy will undoubtedly create far too many tasks, since the basic execution model defines small to medium grain tasks for every goal that needs to be solved. Compounding the problem is the simple task scheduler within a PE, which generates a breadth-first expansion of the process tree. Some highly nondeterministic programs cannot possibly run on the system as it stands (an example is a simple program for the Knight's Tour chess problem, which does a blind search of a tree that has roughly 10^{45} leaf nodes).

We will solve the problem of too many small grain tasks in two ways. Once we have measured the amount of time it takes to load a task into a context, transfer a task to another node, and other parameters that affect decisions about grain size, we will start combining several OPAL process steps into larger grain Mayfly tasks. For example, what are now two separate processes when a procedure call is unified with a clause head and then execution begins in the body, we could combine into a single operation, avoiding the time to save and later restore the virtual machine register that points to the environment of the clause and other task switching overhead. A second way to avoid too many tasks is to build a more intelligent scheduler, so that the global search of the process tree resembles several independent depth-first searches. Here again the structure of the Mayfly PE will enable interesting new strategies, because techniques that might be too expensive in a "traditional" multiprocessor might be feasible if the calculations are done in the MP without interfering with the execution of the user program in the EP.

The primary task of the MP in its role as kernel coprocessor is to control task switching. The MP maintains the local message queue. When a process step is to be executed, the MP builds a description of the task it wants executed in one of the contexts and sends it to the EP via the context cache. The EP does the process update and then returns descriptions of new processes and messages, also via the context cache. Details of task switching and the interaction between EP and MP are presented in Section 5.

Since the MP is connected to the Post Office, it is the obvious choice for interprocessor message routing and other jobs that require interaction with other PEs. The hooks in the kernel that are used to implement dynamic task allocation functions will be described in Section 4. The interconnection topology and other physical attributes of message routing are hidden from the language implementer. The primitive functions that transmit messages are procedures that send and receive fixed length (32-word) packets. OPAL messages are packed into message buffers and then transmitted one packet at a time to the receiving PE. The lower levels of the system guarantee delivery of a packet, but not their arrival order. When the receiving PE has reassembled a complete message, it is installed in the local queue.

Both the EP and the MP must be able to allocate and deallocate memory blocks. These are kernel level functions because the memory systems of the underlying hosts can be quite different. The EP needs to be able to allocate new binding environments, message structures, and fields of process states as it executes a process step. The MP also has to be able to allocate messages and frames when it receives them as parameters in messages from other PEs; for example, if a call to `p(X)` is mapped to another PE, and that PE returns a success message, the binding for `X` that is part of the success message must be allocated locally. We want the MP to be in charge of deallocating unused blocks, since this is overhead that can easily be done outside the main stream of execution.

The heap-based memory allocator in OM is based on a "fast-fit" algorithm [11]. A vector of pointers P keeps track of free blocks. All free blocks of size i are linked together in a chain, and $P[i]$ points to the head of the chain. To allocate a block of size i, either remove the first block in chain $P[i]$ or, if the chain is empty, carve a new block from free space. To deallocate a block of size i simply put it at the front of chain $P[i]$. Since both the EP and MP allocate blocks, access to the pointer vector is protected by a spin lock. An atomic instruction tests the state of a memory word while setting it to a nonzero value. When a processor needs to allocate or deallocate a block, it cycles until the lock has a nonzero value, proceeds with its operation, and then clears the lock.

4. Task Allocation

An instance of an AND or OR process is defined by a *state vector* allocated from the local heap by one of the PEs. The size of a state vector is a function of the number of variables and body goals in an AND process, and the number of clause heads in an OR process. When a new process is created, it is represented by a *seed*, which is simply a start message that contains all the information necessary to expand it into a full state vector. When the start port instruction for a new process is executed, it allocates room for the state vector and assigns the new process its system-wide unique ID.

There is one invariant property of each task allocation strategy for OPAL: seeds may be transferred freely around the system, but once they are "sprouted" to form the root of a new process tree, the resulting state vector is never moved. The ID of the new process is a combination of the PE ID and the local address of the state vector.

An immediate benefit of this policy is that the amount of information passed between PEs when moving a task is minimized, since seeds are much smaller than process states. Furthermore, we do not need to implement a forwarding address scheme. Except for the start message that creates a process, all messages to a process P are responses to earlier messages sent by P: success and fail messages are responses to start messages sent to new descendants, and redo messages are a response from above to successes generated by P. Whenever P sends out a message which it expects will lead to a response, it includes its ID in a continuation field of the message. When another process finally creates the response message, it invokes the continuation by putting P's ID in the receiver field of the message. Since P cannot have been moved since it sent the original message, the new message will go directly to the PE where P resides.

In order to implement a variety of task allocation strategies, hooks were placed throughout the kernel. Collectively, the functions called when the hooks are invoked define the task allocation strategy. The hooks are:

`init_mapper` Called when the system is initialized before the start of each new query, this procedure resets the data structures used by the current strategy.

`assign_task` Called when a PE creates a new seed; some strategies immediately route the new task to another PE, others install it in the local queue, where it may or may not be moved to another PE.

`no_tasks` Called when the MP has no messages in the local message queue; used in strategies that send out requests to neighboring PEs when the local supply of seeds is exhausted.

`request_task` Called when a request for work arrives from another PE.

`new_task` Called when a seed arrives from another PE.

Three strategies implemented so far are named *random*, *round-robin*, and *request*. The first two are similar, in that they send a new task out to another PE as soon as it is created. *random* sends it to a random PE, while *round-robin* distributes the tasks evenly to each other PE in the system (keeping each N^{th} task for itself, where N is the number of PEs). The `no_tasks`, `request_task`, and `new_task` hooks are null functions in these two strategies.

The *request* strategy is a variation of the gradient method of Lin and Keller [9]. When a PE has an empty message queue, it sends a request for work to other PEs, and a PE that receives a request for work sends a seed to the requesting PE. To avoid sending repeated requests, a PE sets an internal flag R_i when it sends out a request to PE i. When it runs out of work, it sends a request message to a PE only if the corresponding flag is not set. Flag R_i is reset when a seed arrives from PE i. Several parameters can be modified to tune this allocation strategy. Some examples: a PE can keep all seeds, regardless of incoming requests, until it reaches some threshold value; a seed can be restricted to some maximum number of transfers or to a maximum distance from its originating PE; more than one seed can be sent to a PE in response to a request.

5. Task Switching

The invariant property of task allocation described in the previous section – that a process does not move once it has been expanded from a seed – is used to define the task switching algorithm used in every implementation of OPAL. Since processes do not move, we use a combination of PE number and local heap address to define a process ID. A process ID is assigned when the process is expanded from a seed.

The object of task switching is for the kernel to take the next message from its local queue, set up virtual machine registers for the next process step, and then set the program counter so the virtual machine starts executing code in the process in order to handle the message. The virtual machine registers set by the kernel are: P, which points to the state vector of the current process; M, which points to the message that triggered the current step (but once the process starts, any new messages it constructs are pointed to by M, since the process refers to only one message at a time); and PC, the program counter. Recall from Section 2.3 that the code block for each process begins with a set of port instructions; the task switching algorithm sets the PC to the address of one of these ports. Once control is returned to the virtual machine, the port instruction finishes restoring registers, loading more information from the message and process into other virtual machine registers as needed. For example, the `or_start_port` instruction sets the A register, which points to the parameters of the current procedure call, to the address of a block of terms contained in the message.

A message header has, among other things, the process ID of the receiving process and a two-bit encoding of the message type. The code address of the port that will process the message is a simple function of the type of the message and the beginning of the code block for the process, which is stored in the state vector.

Using the notation `R.x` to stand for the field named `x` in the structure pointed to by register `R`, the part of a task switch executed by the kernel is:

```
M = next_message();
P = M.to.addr;
PC = M.kind + P.codeloc;
```

The parts of the task switching operations described so far are common to all implementations of OPAL. The machine-dependent steps are contained in the procedure that gets the next message from the local queue and the `no_tasks` hook described in the previous section that is called when there are no messages in the local queue.

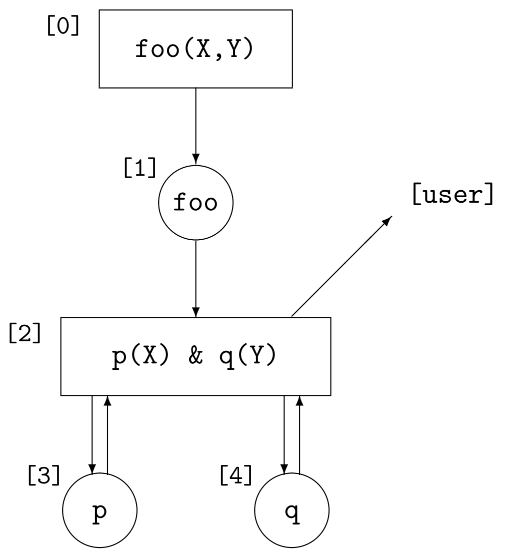
When OPAL is running on the Mayfly, the EP is in a perpetual loop that waits for a context to appear in its input FIFO. The context will hold values of the virtual machine registers P, M, and PC. When it receives a context, the EP starts the fetch-decode-execute loop using these values of the virtual machine registers. Once it starts the virtual machine, the EP no longer needs any information in the current context, so it is free to write into it. As the EP executes the instructions that carry out the process transformation, it modifies data structures in the shared memory. The process state vector is updated in place in the DRAM, and new process and message blocks are allocated from the heap. For each new message generated, the EP puts a pointer to the message into the current context. When the process step is done, the context is enqueued, and the EP goes back into a busy-wait for the next context.

The MP is also in a perpetual loop, but it is looking for incoming packets from other PEs and the host as well as output contexts from the EP. Incoming packets can be control packets (stop the machine, reboot, report runtime stats, *etc*) or data packets containing part of a message that is being routed to this PE. When all packets of an incoming message have arrived, the message is stored in the local heap and enqueued in the local message queue.

When the MP receives an output context from the EP, it examines each message generated in the recently completed step. If the message is to a process that resides on another PE, it is broken into packets and mailed to that PE. Otherwise it is stored in the local queue.

The local message queue is implemented by the FIFO that connects the MP to the EP. The MP keeps track of which contexts are free; since the EP never uses a context that was not sent via the FIFO, the MP is the only processor that creates contexts. To enqueue a message, it writes the values of the three registers (P, M, and PC) into the context and inserts the context into the FIFO. If all 32 contexts are in use, then a pointer to the message is put in a linked list which implements an overflow queue.

Clearly, as long as there is only one execution thread this scheme will not speed things up very much; in fact, as a result of the overhead of loading a state into the context buffer it might even slow down the overall execution. However, when there are two or more threads in the



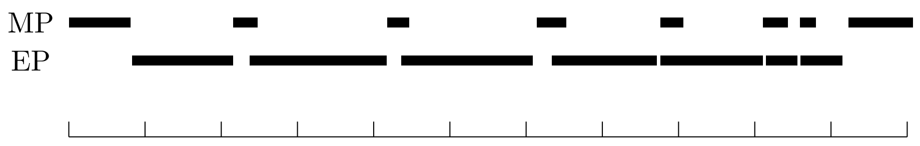
Tree of process states generated during execution of `small`*.*
Figure 8. Process Tree

local PE, they can be overlapped, with the MP setting up information for one thread while the EP executes a process step in another thread.

The program named `small` from Figure 3 has two parallel threads. The tree of processes generated when this program is executed is shown in Figure 8, and a Gantt chart that shows when the EP and MP were active during the first few steps of the program is shown in Figure 9. The tick marks at the bottom represent intervals of one millisecond (these measurements were taken from a prototype of the PE which did not have a data cache, so the times are slower than they will be on the final machine). The top line of the Gantt chart is for the MP, and the bottom line is for the EP. Dark sections of the time line represent periods when the corresponding processor is active.

The first event in the plot is when the MP receives a start message from the host. The MP builds a context for the top level goal, enqueues it for the EP, and goes into its busy-wait loop. Since the EP is waiting for a context, it is activated almost immediately. The top level goal builds a call to the procedure `foo` and then switches out.

The first three events in the EP occur during the execution of a single thread. The first step is the AND process step for the top level goal. The second step is the OR process that unifies the call, and the



Times were recorded by calling a procedure that returned a time stamp (contents of an internal processor register) at the beginning and end of each event. MP events are the processing of an incoming data packet or a context from the EP. EP events correspond to a single OPAL process step.

Figure 9. Gantt Chart for `small`

third step is the start of the AND process for the body of `foo`. Since `p(X)` and `q(Y)` will be solved in parallel, the third step in the EP makes two OR processes, one for `p` and one for `q`.

At this point, approximately 6.3 ms into the program, we can see a small bit of overlap in the Gantt chart. As soon as the MP builds a context for the start message to the OR process for `p`, it enqueues it. Then, since there is another message, it continues and builds a context for the start message to `q`. The EP resumes with the OR process for the call to `p` while the MP is building the context for the call to `q`. When the EP is done with the unifications in the call to `p` (its fourth step), the context for the call to `q` is waiting, and the EP switches immediately to this next thread. While the EP is executing the instructions for the head of `q` (its fifth step) the MP is checking the output context from the call to `p`; there aren't any, so this MP step is over quickly. The last two steps of the EP are for the AND process for the body of `foo`, as it gets success messages from both the calls. The last event shown in the figure is the MP building a success message to send to the host.

The chart shows what we expected to see. During periods when the machine had just one thread to execute, EP and MP events were "interlocked" where an event in one processor enabled just one step in the other. The EP was idle an average of 220 microseconds between steps during this phase. As soon as two execution threads were available, the EP was able to work on one thread while the MP built the next message for the other thread. The EP was idle an average of 45 microseconds between steps in this phase.

The chart also illustrates the fact that the EP is doing most of the work, and the MP is idle a significant amount of time in between process steps. For the `color` and `path` programs, where there is a lot of work done at the start of each new OR process, the EP does roughly three times as much work as the MP overall.

In a parallel system, with more than one Mayfly PE, the MP will be interrupted to handle incoming packets from other PEs, and the MP time line will show externally triggered events in the middle of what are now fairly large gaps. The challenge for future versions of OPAL will be to migrate just enough work from the EP to the MP, but leaving ample free time to handle inter-PE packets. The main goal for the task allocators will be to create several threads on each PE, and keep inter-PE messages to a minimum during periods of high activity.

6. Future Work

One of the most time consuming functions in the virtual machine is dealing with the possibility that a goal can be canceled. Again, we are exploring two approaches. One, we can try to get the compiler to tell us when a goal is speculative and subject to cancellation. If a goal is not speculative, then we can avoid the overhead associated with connecting these goals to their parents. The other approach is to move the cancel operation and the housekeeping that links speculative children to their parents to the kernel coprocessor.

Execution profiles show that cleaning up after failed processes is also fairly time consuming. The low level block allocator and deallocator are efficient enough. In the map coloring program, over 3600 blocks are allocated and all are eventually deallocated. The allocator accounted for roughly 1% of the execution time, and the deallocator was too short to measure (it didn't show up in the execution graph). However, deallocating processes and messages often requires traversing complex structures, and when this time is included the memory management operations account for almost 5% of the execution time.

The most likely next step will be to have the MP implement all port instructions, so that the EP starts right in at the instruction branched to from the port. Among the advantages of this scheme are the fact that some steps are done in a single port instruction, *i.e.* the machine does a task switch at the end of the port instruction. These steps will be done entirely in the MP, leaving the EP free for more complex operations. A disadvantage is that this might mean more context for the EP to load into its internal registers from the context buffer.

Since all bindings required by a process in a closed environment system are localized when the process is created, the only variables modified by the EP during a process step are those that occur in frames owned by the process. The closed environment model was designed to allow a process step to be executed by any processor at any place in the system. What this means for the Mayfly implementation of OPAL is that we can have the MP load more of a process state into a context than merely the M, P, and PC virtual machine registers. We are considering putting the top level of the argument registers and probably the local stack frame of an AND process into the context so they can be accessed from fast memory by the EP. Since there are no chains of variable references leading from a frame of the current context, there is no danger that the EP could be making bindings to variables that have already been loaded into a different context by the MP. Performance will improve not only because context cache words have lower access times, but we will also avoid the cache misses that occur when the EP starts to access the data structures of its new context.

The most difficult question to answer will be how much of the context of the new process should be put into the context buffer by the MP. The more that is there, the easier it will be to access, but the longer it will take to load into the buffer. Even if the values are not loaded into the context, it may be worthwhile to preload the D-cache for the EP by having the MP traverse structures and touch terms so they are read into the shared data cache.

Acknowledgements

Early development of the OM virtual machine was supported by NSF grant CCR-8707177. The project was also supported by a generous equipment donation from Hewlett-Packard and a research grant from Motorola. Implementation of OM on the Mayfly would not have been possible without the patient support of Al Davis, Robin Hodgson and the other members of the Mayfly group at HP Labs.

References

1. M. Carlsson. *Design and Implementation of an OR-Parallel Prolog Engine.* PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 1990. Report TRITA-CS-9003.
2. J. S. Conery. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, Boston, MA, 1987.
3. J. S. Conery. Binding environments for parallel logic programs in non-shared memory multiprocessors. *Int. J. Parallel Programming*, 17(2):125–152, April 1988.
4. A. L. Davis. Mayfly: A general-purpose, scaleable, parallel processing architecture. xx(xx):xx–xx, xx xx.

5. B. Hausman. *Pruning and Speculative Work in OR-Parallel Prolog.* PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 1990. Report TRITA-CS-9002.
6. M. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, University of Texas, Austin, TX, 1986.
7. C. E. Hewitt, G. Attardi, and H. Lieberman. Specifying and proving properties of guardians for distributed systems. In G. Kahn, editor, *Semantics of Concurrent Computation*, number 70 in Lecture Notes in Computer Science, pages 316–336. Springer-Verlag, New York, NY, 1979.
8. P. M. Kogge. *The Architecture of Symbolic Computers.* McGraw-Hill, New York, NY, 1991.
9. Frank C. H. Lin and Robert M. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, SE-13(1):32–38, January 1987.
10. D. M. Meyer and J. S. Conery. Architected failure handling for AND-parallel logic programs. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 633–653, 1990.
11. T. A. Standish. *Data Structure Techniques.* Addison-Wesley, Reading, MA, 1980.
12. D. H. D. Warren. An abstract Prolog instruction set. Tech. Note 309, SRI International, October 1983.
13. D. H. D. Warren. The SRI model for OR-parallel execution Prolog – Abstract design and implementation. In *Proceedings of the 1987 IEEE Symposium on Logic Programming*, 1987.