Parallel Logic Programs on the HP Mayfly

John S. Conery

CIS-TR-90-22
December 7, 1990

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 a coprocessor 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 coprocessor implements kernel level functions in OM, with particular attention to the operations that support task switching. 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 coprocessor can effectively build the context for the next thread.

Department of Computer and Information Science
University of Oregon

1 Introduction

OPAL, the Oregon Parallel Logic 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 [5,6]. 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.

A prototype Mayfly system with seven nodes (there will eventually be 19) interconnected temporarily via an HP-IB bus is now running at HP Labs. The version of OPAL that runs on this system uses three different dynamic task allocation schemes. A future paper will describe the task allocators and how well parallel programs run on the Mayfly. The main focus of this paper is 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 RISC chips in each node as a *kernel coprocessor*, offloading as much of the message passing overhead as possible from the main logic program processor.

The first two sections of the paper provide some background information on OPAL and on the Mayfly. Following that are some details on how this implementation of OPAL takes advantage of Mayfly hardware structures. The paper concludes with some timing data and a list of future projects.

2 The OPAL Language and Virtual Machine

2.1 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* [4]. A process that is expecting a message from other processes is a continuation for that message. For example, an AND process that starts an OR process to solve a goal is a failure continuation that will be invoked by the OR if it fails to unify any its clause heads with the goal. Continuations can be passed as arguments in messages to allow processes to respond directly to processes other than their parents [9]. When an OR process matches a call with the head of a nonunit clause and starts an AND process for the body, the OR's own success continuation is passed to the new AND along with the initial bindings for body variables and other parameters. When the AND finds a solution and invokes its success continuation, the message is sent directly to the ancestor process that made the original call. 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 [10] 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] A few more complex operations, such as I/O, can be 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.* [7]).

Figure 1 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 5.2. The second is an OR-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

[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".  There are 16 arcs, leading to 51 solutions,
% in the actual program (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/2 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 1: Example OPAL Programs

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 its siblings created for other clauses in this procedure. The compiler also exploits AND parallelism automatically. It uses mode information extracted from an abstract interpreter to order the goals in a data dependency graph which will control the order of execution at run-time. The order may change if a goal does not bind input variables to ground terms [11]. Since the procedures in the examples presented here always bind their arguments to ground terms the programs have a fixed execution order determined at compile time.

2.3 OM Virtual Machine

OPAL programs are compiled into instructions of a byte-coded virtual machine named OM (for *OPAL Machine*). The 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 OR processes. 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 (every process has the same type of ports, in the same relative location, at the head of its code). 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 literal failed, and then branches to a section of code that determines which predecessor(s) of the failed literal need to be sent redo messages in order to get different bindings for the variables in the failed literal.

OM is faintly reminiscent of the Warren Abstract Machine for standard Prolog implementations [12]. To solve a goal, arguments are loaded into argument registers by `put` instructions. Unification is performed in the head of a clause by a series of `get` instructions that compare an argument register with a constant or the value of a variable. The main differences between OM and WAM are in the representation of variable bindings and the control instructions.

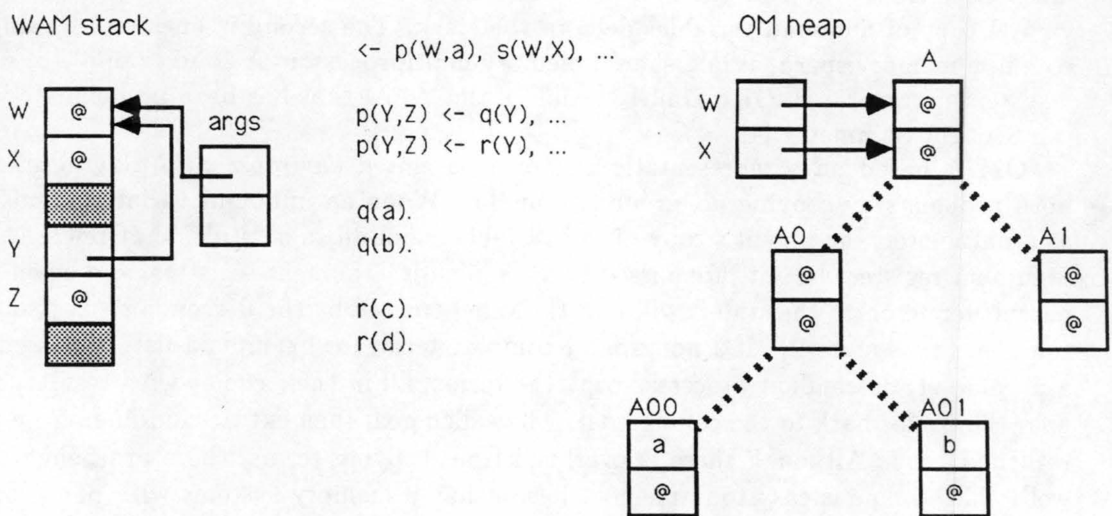
Most logic programming systems, including WAM and OM, represent a logical variable by a cell that contains a pointer to (address of) the variable's value. Unbound

variables are pointers to themselves. When two unbound variables are unified, one is bound to a pointer to the other.

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 (the shared ancestor variable problem), and second, in nonshared memory multiprocessors there may be a situation where a processor needs to refer to a value stored in another processor's memory. There are many effective solutions to the first problem, based on giving each OR-parallel thread a virtual copy of unbound variables deep in the stack. The second is finessed by using a common memory space, as in a shared memory multiprocessor. A good example of such a system is Aurora, an OR-parallel version of the WAM that has been implemented on the Sequent Symmetry [1].

OM is based on a representation, known as *closed environments*, that addresses both problems 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. 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 2). 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. The called goal then extract bindings from the returned copy. Although there is overhead from copying terms, 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 concomittant worries over buried goals [8]) 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. Copying also simplifies garbage collection, which will be a severe problem in large parallel programs.

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 **`check_solved`** instruction that



A simple program and the binding environments it creates in the WAM and OM. In the WAM, pointers to arguments are put in the A registers, and unification binds the original variables. In OM, the original variables are bound to "stubs" which are later dereferenced to values that are returned in copies of the original argument registers.

Figure 2: WAM Stacks *vs.* Closed Environments

causes a task switch because its argument literals have not yet been solved.

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. Execution resumes in the OR process, which sets up the unification of the next clause. OR-parallelism will be exploited in programs that have multiple matching heads.

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 3 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

Figure 4 shows the layers of abstraction in an OPAL implementation. The top layer is the OM virtual machine. The registers, instructions, and other aspects of this layer are all concerned with execution of a single process, known as the "current process." The bottom two layers are the "OS" and kernel, which implement inter-process interactions. The distinction between the OS and kernel is that the OS implements machine-independent functions. For example, when an OR process starts a new descendant, it executes a trap to the kernel layer which creates the "seed" of a new AND process. This is a kernel function because seeds may be transferred to other processors in the system, and there are hooks in this function that call host-dependent task mapping procedures. When the seed is scheduled for execution, the first instruction it executes is `and_start_port`, which will initialize a state vector for the new process. This operation is implemented by a trap to the OS layer, since the new state is machine independent.

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

```
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_SCOB, q1_FCOC
          check_solved       [1,2]

q1_SCOB:  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 3: Code Block and Resulting Process Tree

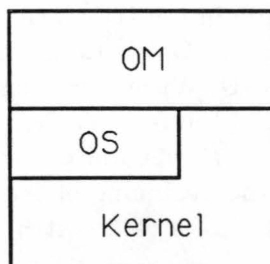


Figure 4: Modules in an OPAL Implementation

```
?- ... 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 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. OM must build an OR process and then execute a task switch instruction to invoke it. The OR process performs both unifications and creates two new AND processes for the bodies. Next there is another task switch to begin execution of the body in one of the new AND processes. Finally, before execution resumes in the body of the first AND process, 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 OM must do up to three task switches. There are several ways to address this extra overhead. This paper reports on two of them: efficient representations of tasks and using a coprocessor to implement the kernel level functions involved in building and switching among tasks. Other ways to reduce overhead are through better compilation and by using more efficient message passing patterns. An example of a compiler improvement is one we are now working on. If the compiler knows there will be no choice point in the called procedure, there is no need to create an OR process, and the goal can be solved by a Prolog-style procedure call. Tail recursion and last call optimization are examples of more efficient message passing patterns; we hope to come up with additional creative uses of success and fail continuations.

The remainder of this section describes the representation of tasks and the task switching algorithm implemented in the OS layer. Following a brief overview of the

Mayfly in the next section will be a description of multiprocessor task switching functions implemented in the Mayfly version of the OPAL kernel.

The key to fast task switching in OM is to use the local address of a process state vector as one field in its process ID. When a message M is being sent to a process P, the OS uses the address portion of the receiver field of M to locate the state vector of P. The code address of the port in P that will process the message is a simple function of the type of the message and the beginning of the code block for P, which is stored in the state vector. The algorithm for a task switch is:

```
M = next_message();          % trap to OS to get message from queue
P = M.to.addr;               % get process state vector from receiver ID
PC = M.kind + P.codeloc;     % set address of port instruction
```

When the fetch-decode loop resumes, the next instruction executed will be the port instruction that will restore other registers and resume execution in the receiving process.

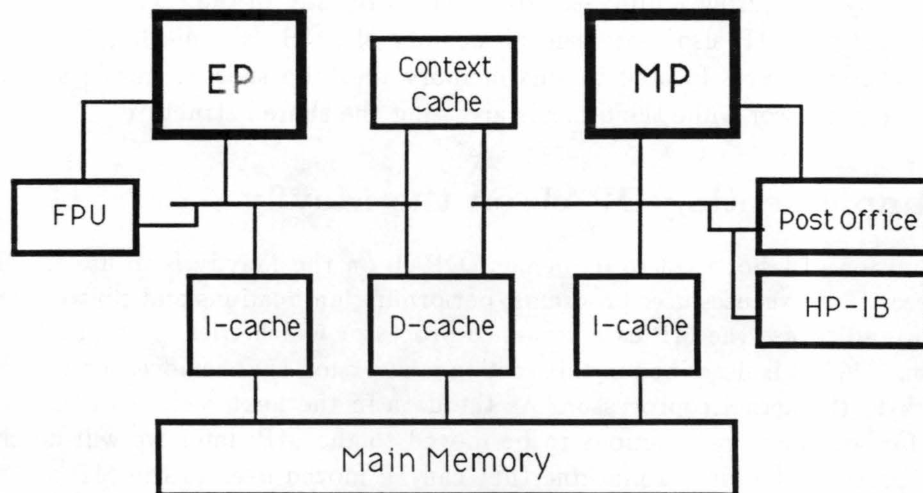
In order to use memory addresses in process IDs, the system must assure that processes are not relocated, either within the memory space of the processor that created the process or to another node, once it has started. This constraint is enforced by the task allocation functions and by the OS functions that deallocate process states (we must be careful not to deallocate a process if there are any other processes that might still send it a message).

3 A Language Implementer's View of the HP Mayfly

Figure 5 shows the data path of a single processing element (PE) in the Mayfly. 19 PEs are connected via their post office chips in a hexagonal toroidal mesh to form an "E3 processing surface," several of which may be connected to form larger multiprocessors [5].

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 assembled a complete message, it is installed in the local heap.

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 PEs are known as the *execution processor* (EP) and *message processor* (MP). The MP



The main components and data path within a single PE of the Mayfly (from [5]).

Figure 5: Structure of a Mayfly PE

is not involved in inter-PE packet routing; all low-level packet handling is done by the "post office" processor. The MP is interrupted to handle a packet only when this PE is the final destination for that packet.

The EP and MP communicate with each other through a dual-ported memory known as the *context cache*. This cache contains 32 256-word contexts, each identified by a unique 5-bit tag. A processor has access to one context at a time, which is at a fixed location in its address space. Addresses in the range designated for use by the context cache are combined with the ID of the current context to form a reference to the actual context cache word. As long as the MP and EP have different current contexts, they can be operating on blocks in the cache simultaneously, without fear of collisions. Memory accesses to the current context are very fast – one cycle, with no wait states from arbitration.

The context cache has two FIFOs which are used in inter-processor communication within the PE. The EP pushes context IDs into one FIFO, and the MP pushes IDs into the other. A processor can build a data structure in its current context, put the ID of the context into its FIFO, and go off to another task. When it wants information from the other processor, it checks to see if there is a context in the other FIFO. Pushing and popping contexts are atomic operations implemented by the context cache itself,

activated by reads from a processor to a control register of the cache.

The EP and MP also share a local memory of 8MB dynamic RAM. On occasions when data structures located in this memory must be shared, simple spin-locks can block one processor while the other is accessing the shared structure.

4 Implementing OPAL on the Mayfly

The division of labor used to implement OPAL on the Mayfly is to use the EP as the processor that executes user programs, performing unifications and control instructions in OM, and to use the MP as a kernel co-processor which tries to offload the low level overhead. What is described in this section is a version that removes a minimal amount of work to the kernel coprocessor. As the data in the next section will show, there is room for several more functions to be moved to the MP; later we will describe some of the parts of the virtual machine that can be moved over to the MP and issues we need to take into account. If too much work is done by the kernel processor, it may end up being the bottleneck in a larger parallel machine when it is handling more inter-PE traffic.

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 contains the values of four virtual machine registers:

`P` points to the state vector of the process that will be updated in this step.

`M` points to the message that triggered this update.

`PC` is the address of the port instruction that will start the update.

`CP` is the continuation pointer, the address of another instruction in the current process (irrelvent for this paper).

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 output 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 (the MP is not involved

in routing messages to other PEs; that is handled at lower levels, by the post office). 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 (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 four registers (P, M, PC, and CP) 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 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. This situation occurs early in the execution of the demo program named `small`, and data that shows faster task switching between parallel threads is given in the next section.

An important aspect of the closed environment representation for binding environments should be mentioned at this point. Since all bindings required by a process 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. There are no chains of variable references that lead to a frame that is currently in DRAM (or on another PE), and thus there is no danger that the EP could be making bindings to variables that have already been loaded into a context by the MP. The closed environment model was designed to allow a process step to be executed by any processor at any place in the system. As a result, we can have the EP working with one environment while the MP is loading others into contexts.

There is one place in the system where the EP and MP need to coordinate access to shared data structures in DRAM. Both processors need to be able to allocate new data blocks in memory. The EP needs to be able to allocate new binding environments, message structures, and fields of the process state as it executes a process step. The MP 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. Also, we want the MP to be in charge of deallocating unused blocks. For these reasons, the data structures used by the memory allocation module

are shared, and access to them 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.

5 Preliminary Performance Data

Two sets of timings are presented in this section. The first compares OM running on an HP 9000/835 workstation with SICStus Prolog running in the same environment. The second is a detailed analysis of operations within a single PE, to see how well the EP and MP are working together on OPAL programs.

5.1 OPAL Compared to SICStus

As a "reality check" to see how close OPAL comes to state of the art Prolog systems, we compared our OM programs with similar Prolog programs running compiled under SICStus Prolog. We chose to compare to SICStus because it uses the same basic implementation technology – byte-code interpreter written in C – and because it is widely acknowledged to be the best such Prolog system available.

The `small` program was too short to measure accurately with either system (the clock on this workstation has a 10 msec resolution). To find all 51 solutions to the `path` program required an average of .078 seconds in OPAL and .020 seconds in SICStus. To find all 72 solutions of the `color` program took an average of .224 seconds in OPAL compared to .044 seconds for SICStus.[2]

The difference in speed comes from several sources. SICStus is much more mature than OPAL, and is a more efficient program as a result. One example is in the way the systems select candidate clauses for unifications during procedure calls. We have not yet implemented clause indexing in OPAL, which by itself will improve our performance on these two programs since the alternatives can be distinguised by examining the first argument register. SICStus uses sophisticated "try chains", below the level of the virtual machine, since compiled Prolog programs can call interpreted procedures or procedures written in C.

Most of the difference is undoubtedly accounted for by the different execution models: the fact that OM builds a new OR process for each procedure call and a new AND process for each clause body, and that each OR process makes a copy of the input argument register set when putting it in closed form after each successful unification.

[2]Both systems were I/O bound and ran at roughly the same speed when measurements were first taken. The numbers reported here were obtained by turning off I/O in OPAL and having Prolog time the final failure of a top level goal of the form `epath(X,Y), fail`.

A current project is investigating the possibility of postponing the environment closing operation until we know the descendant will actually run on another PE. The cost of building a choice point will be similar to the cost of building the OR process, but we will avoid copying nonground terms until we know the copies will be sent to another PE.

Given these differences and other places we hope to improve OM, we are close to our goal of making a single task switch roughly equivalent to a complex procedure call in Prolog.

5.2 OPAL Operations within a Mayfly PE

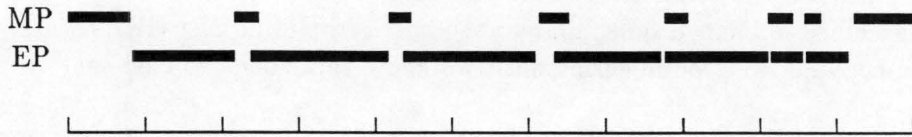
Figure 6 shows a Gantt chart of the first few steps in the execution of the `small` program. The top line is for the MP, and the bottom line is for the EP. Dark sections of the time line represent periods when the corresponding processor is active.

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 first step in the top level call builds a call to the procedure `foo` and then switches out.

The first three events in each processor show 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 third step is the AND process for the body of `foo`. Since this an AND-parallel goal (because `X` and `Y` are unbound in the call, `p(X)` and `q(Y)` can 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 6.3ms into the program, we can see a small bit of overlap in the Gantt chart. 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



The tick marks represent intervals of one millisecond. 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 6: Gantt Chart for `small`

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. The chart distorts the true situation a little, since later on in the computation EP events are shorter and MP events are longer. By the end of the `small` program, the EP and MP were active roughly the same amount of time. 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 events triggered externally in 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 cancelation. 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. Together these 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.

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. We are considering putting the top level of the argument registers and probably the local stack frame of an AND process into the context. 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 periodically touch terms so they are read into the cache. We know from the constraints of the closed environment model that no value loaded into the cache by the MP can be changed while the EP is busy in other execution threads, so the effort will not be wasted, and this might dramatically improve the cache hit ratio in the EP.

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] Carlsson, M. *Design and Implementation of an OR-Parallel Prolog Engine.* PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 1990. Report TRITA-CS-9003.

[2] Conery, J. S. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, Boston, MA, 1987.

[3] Conery, J. S. Binding environments for parallel logic programs in non-shared memory multiprocessors. *Int. J. Parallel Programming 17*, 2 (Apr. 1988), 125–152.

[4] Conery, J. S. Task switching in OPAL. In *Proceedings of the Workshop on Parallel Implementation of Languages for Symbolic Computation* (July 1990), Tech. Rep. CIS-TR-90-15, Univ. of Oregon.

[5] Davis, A. Mayfly: A general-purpose, scalable, parallel processing architecture. Hewlett-Packard Company, December, 1990.

[6] Davis, A. L., Coates, B., Hodgson, R., Schediwy, R., and Stevens, K. Mayfly System Hardware. Technical Report HPL-SAL-89-23, Hewlett-Packard Laboratories, April 1989.

[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] Hermenegildo, M. *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.

[9] Hewitt, C. E., Attardi, G., and Lieberman, H. Specifying and proving properties of guardians for distributed systems. In *Semantics of Concurrent Computation*, G. Kahn, Ed., no. 70 in Lecture Notes in Computer Science. Springer-Verlag, New York, NY, 1979, pp. 316–336.

[10] Kogge, P. M. *The Architecture of Symbolic Computers.* McGraw-Hill, New York, NY, 1991.

[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), pp. 633–653.

[12] Warren, D. H. D. An abstract Prolog instruction set. Tech. Note 309, SRI International, Oct. 1983.