An Abstract Machine for Parallel Graph Reduction

Lal George
Gary Lindstrom

UUCS-89-003
January, 1989

An Abstract Machine for Parallel Graph Reduction

Lal George
Gary Lindstrom

Department of Computer Science
University of Utah
Salt Lake City, UT 84112

January 25, 1989

Abstract

An abstract machine suitable for parallel graph reduction on a shared memory multiprocessor is described. Parallel programming is plagued with subtle race conditions resulting in deadlock or fatal system errors. Due to the nondeterministic nature of program execution the utilization of resources may vary from one run to another. The abstract machine has been designed for the efficient execution of normal order functional languages. The instructions proposed related to parallel activity are sensitive to load conditions and the current utilization of resources on the machine. The novel aspect of the architecture is the very simple set of instructions needed to control the complexities of parallel execution. This is an important step towards building a compiler for multiprocessor machines and to further language research in this area. Sample test programs hand coded in this instruction set show good performance on our 18 node BBN Butterfly as compared to a VAX 8600.

1 Introduction

We define an abstract machine suitable for parallel graph reduction on a shared memory multiprocessor machine. This provides a level of abstraction that is an important step towards building a compiler. The machine is intented for an ML like language with *compound datatypes* executed lazily by default. Our interest in lazy functional languages for multiprocessors is motivated by several reasons:

1. Awkward annotations for synchronization such as those found in Flat Concurrent Prolog (FCP) are not required. In FCP over specification could result in deadlock and underspecification could result in a *runaway unification*. The synchronization of functional programs is under the jurisdiction of the runtime system that blocks or suspends on access to an object that is unevaluated or in the process of being evaluated. The programmer need not be aware of such events.

2. Parallelism in functional programs is obtained from the evaluation of strict arguments to functions and the evaluation of *anticipatory work* (§ Section 5) in parallel with *mandatory work*. The strict arguments to a function can be obtained from strictness analysis or user annotations. These annotations are much simplier when compared to the *future construct* of multilisp for example. It is not obvious what expressions should have a future construct wrapped around them to obtain efficient execution.

3. Due to the side effect free nature of functional languages, expressions can be executed in parallel without fear of having violated data dependencies. Maintaining data dependencies in an optimistic evaluation strategy may require the need for *rollback* in the computation, while an excessive amount of communication traffic can be generated in a conservative strategy.

4. Normal order evaluation which is not easily amenable to efficient execution on sequential machines turns out to be of great value on parallel machines. The *producer consumer* property afforded by normal order reduction implies that the consumer may begin execution as soon as data has been produced by the producer thus generating concurrent activity.

5. Functional languages are readily amenable to static analysis and program transformation which are often non-trivial for imperative languages.

Since little is known about programming *general purpose* multiprocessors the benefits afforded by functional languages makes them a better *starting point* compared to other alternatives.

2 Intended Architecture

It is straightforward to map the abstract machine we propose onto a shared memory MIMD machine where the abstract machine is is emulated on each node of the processor. We are interested in tightly coupled shared memory machines like the switch connected BBN Butterfly or a bus connected Sequent multiprocessor[1]

3 Source Language

The abstract machine is intended for a language like ML which is a strongly typed functional language with compound datatypes, i.e., types expressed as a *product* and *sum* of types. The default evaluation strategy is normal order implemented via graph reduction. Parallelism is obtained from :

1. Evaluation of strict arguments to a function. Such information is derived from strictness analysis or user annotations.

2. Evaluation of anticipatory work from the top level print function.

4 Memory Management

Since the heap is a shared memory space, migrating tasks is cheap and convenient. Each abstract machine has a segment of the total heap that is local to that machine. Accesses to the local segment of the heap is *usually* faster than accesses to nonlocal segments. Each abstract machine makes allocations out of its local heap segment and when exhausted will try and allocate from a remote

[1] The distinction between a shared memory machine and a distributed machine is beginning to pale. In the extreme one can consider for example the computers connected by the arpa net as a shared memory machine where addresses are $site + location + offset$.

heap segment. This means that the heap allocation routine must be a critical section. Since functional programs tend to be memory intensive this is a bottle neck as locking would be required for every allocation. The heap allocation can be optimized by locally managing a sufficiently large buffer space allocated out of the heap. Test programs showed an improvement of between 7-17%.

5 Top Level Print

A functional program compiled with no strictness information or annotations may exhibit parallelism from the *top level print* function. The purpose of the top level print function is to perform IO of the top level expression being evaluated. Parallelism is obtained by spawning *if possible* the remaining components of the expression *only if* they are in unevaluated form. We call this work *anticipatory* work since its need is anticipated. *Mandatory* work on the other hand is related to whatever is currently being printed. Functionally the top level print is defined by[2]

```
print (int x) true = {output x; return true}
print [x|xs] true = print (spawn xs) (print x true)
```

`Output x` sends the integer represented by `x` to the output stream. The way `print` is defined above does not yield much parallelism since parallel activity is only generated when the second statement of the `print` definition is matched. It would be desirable if the unevaluated subcomponents of `xs` were evaluated in anticipation of their use. In our implementation each task has an *exhaustiveness bit* that when set indicates that any *unevaluated subcomponents* may be spawned on available processors. This is low priority work and is only performed if resources are available. The exhaustiveness bit is propagated down to the unevaluated components. Figure 1 shows a stream being evaluated where each component of the stream is implemented as a fixed delay to represent some computation being performed. The only parallelism is that obtained from anticipatory work.

6 Abstract Machine

The abstract machine is derived from Johnsson's G-machine[2] but modified for parallel execution. The abstract machine running on every node is described by the tuple `<S, C, G, F, D>` where:

- `S` = evaluation stack or pointers to heap nodes.

[2]Assumes that the only basic types are integers and lists. This is easily extended to generalised types. `int` refers to a datatype constructor or tag.

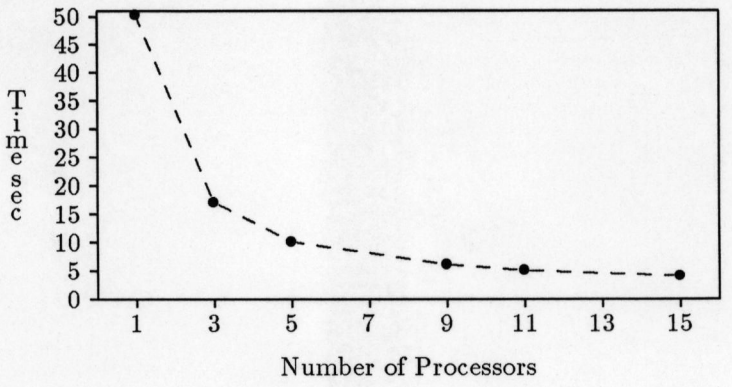

Figure 1: Parallelism from Anticipatory Work

- `C` = code sequence being executed.
- `G` = heap space shared by all processors.
- `F` = a status register with fields (currently only one!) that get set/reset by specific instruction.
- `D` = sequence of return or continuation points and saved stack segments.
- `T` = task queue.

The abstract machine is mapped onto each node of the multiprocessor. The only component shared among all processors is the heap and task queue. All others are local to the node and may reside in its local store.

7 Function Evaluation

A task has the following components:

- `TAG` → A tag value that may either be `CLOSURE` or `BUSY`.
- `f` → A code pointer.
- `wc` → A wait count for synchronization.
- `nc` → A notification chain consisting of pointers to closure
- `env` → A pointer to an environment/argument block.

There are certain other components that we have omitted for compactness in this exposition. These include the **exhaustiveness bit** mentioned in Section 5, and a **lock bit**.

We use the following conventions when representing the state of the machine. The evaluation stack will normally be represented by $s_0 : ...s_k : ... : s_n$ where s_0, sk and s_n are references to objects on the heap. The code sequence is represented as a list within [and]. We will use $\mapsto$ to dereference a pointer. Rather than displaying the entire heap, only the references of interest will be shown within { and }, and all other references can be assumed to be unchanged. A tag with an accent such as $\widehat{Tag}$ will indicate that the tag is locked. Absence of the accent would indicate that it may or may not be locked.

As in the original G-machine, the **S** stack is used to *cache* the arguments of a closure and maintain an environment during execution of the closure. Prior to the execution of the code pointer associated with the closure, the argument block is unwound or copied onto the **S** stack. The state of the machine immediately after the unwind would be represented by :

$$< s_0 : s1 : ...s_n, [f], \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ wc\ nc\ env \\ env \mapsto s1 : ...s_n \end{array} \right\}, \mathbf{F}, D, T >$$

8 Instructions

The following discussion is motivated by presenting typical programming situations and the corresponding code generated. A thorough description of the instructions related to parallel evaluation is then presented. It will be shown in the following sections that the generation and synchronization of parallel activity can be largely achieved by the use of two instructions, **demand** and **block**.

8.1 Example 1

There are several situations where the environment needs to be pre-evaluated to some degree before some computation can proceed. Consider the merge[3] program below

```
1.      merge [ ] L = L
2.      merge L [ ] = L
3.      merge (l1 as x::xs) (l2 as y::ys) =
            if (x > y) then
                x ::  merge xs l2
            else y ::  merge l1 ys
```

Before the body of the third definition can be executed it is necessary to pre-evaluate or demand **x** and **y**, since their value is needed in the relational test,

[3] :: is infix **cons** operator

(x > y). The following scenarios may be present before entering the body of statement 3.

1. Both x and y are *evaluated.* In this situation the control should jump immediately into the code for the body of the definition.

2. Both x and y are *unevaluated.* A simple analysis shows that it is worthwhile to retain the larger computation in the sequential thread of execution and spawn off the smaller ones. In this situtation we *cannot* know which is the larger of the two, so we must arbitrarily spawn one off and retain the other in the sequential thread of execution. Since both executions may block a closure must be built representing the continuation that will be notified upon completion of both tasks. The last task to perform the notification **awakens** the continuation. This allows the continuation to be executed on any processor that happens to be available.

3. Only one of x or y is evaluated or under evaluation. In this case the unevaluated one is reduced locally. The same comments regarding the continuation and notification above apply.

4. Both x and y are under evaluation. In this case the machine must find some other work to do after setting up the proper continuation and notification chains.

In all of the above cases where parallel activity is generated, it may be the case that all processors are busy and the task pool T is also full. In this case the object demanded must be evaluated *inline.* This could result in deadlock as explained in Section 8.7.

The code that is generated in this very simple example must be able to handle all the cases mentioned above. The code to be executed on our abstract machine before entering the body of statement 3 of the merge program is shown below.

```
set_wtcnt(0, 3);      % wait count on redex = 3
Reset_ResrvClsr();    % Reset flag

push(1, 3);           % l1 to offset 3
hd(3);                % replace with head of l1
demand(3);            % evaluate

push(2, 4);           % l2 to offset 4
hd(4) ;               % replace with head of l2
demand(4);            % evaluate

block(5, 2, g_merge);    % barrier synchronization
```

The subcomponents, **x** and **y** are accessed onto locations on the stack and demanded. **g_merge** is a code pointer for the body of statement **3**.

In general, the nature of code generated has the following structure :

1. Set the wait count on the root redex.
2. Reset flag, **F**.
3. One or more occurences of task creation (typically closure building) or subcomponent access of a structure, followed by the **demand** instruction.
4. Creation of an environment on the top of the stack for the continuation followed by the **block** instruction.

We do not attempt to describe immediately how the instructions handle and all the cases mentioned above. This is deferred to a more thorough rendering in Section 8.6 where each instruction is described in detail. In the above code sequence we have assumed that the arguments must have been previously evaluated to at least a **pair** with possibly unevaluated components. Hence we are at liberty to access the head without checking for a pair. This sort of information can be determined from strictness analysis or user annotations.

8.2 Example 2

The same requirements occur when a closure is built representing a delayed computation. The imports to the closure from the outer environment are captured in a new environment. The code pointer associated with the closure may need to pre-evaluate some of these imports before proceeding. This case is analogous to the situation described in example 1. Again it is not difficult to determine what needs to be pre-evaluated from static analysis or user annotations.

8.3 Example 3

Having evaluated the imports or arguments of a closure we can obtain further parallelism by evaluating the strict arguments of functions in parallel. For example, consider the fibonnaci function:

```
1.      fib 0 = 1
2.      fib 1 = 1
3.      fib n = fib(n-1) + fib(n-2)
```

Since the + is strict in both its arguments, we can do them in parallel. Unlike the previous case, we know this time exactly what the parallel computations are, namely `fib (n - 1)` and `fib (n - 2)`. We can thus apply heuristics as to which one to spawn off and which to retain in the sequential thread of execution. The code generated is similar to that above and would look like:

```
set_wtcnt(0, 3);          % wait count on redex = 3
Reset_ResrvClsr();        % reset flag

push(1, 3);               % n to offset 3
mkClosure(fib1, 3);       % make closure with n as argument
demand(3);                % evaluate

push(1, 4);               % n to offset 4
mkClosure(fib2, 4);       % make closure with n as argument
demand(4);                % evaluate

block(5, 2, g_add);       % barrier synchronization
```

`fib1` computes $fib(n-1)$, `fib2` computes $fib(n-2)$ and `g_add` is a function that adds two numbers. In this case it so happens that we have opted to perform `fib (n-1)` in the sequential thread of execution; we could have opted otherwise (§Section 8.6).

We now proceed to describe the instructions used in more detail.

8.4 Set_wtcnt k v

Sets the wait count on a closure to a specific value. **k** is the depth on the **S** stack of the closure and **v** is the value of the new wait count.

$$< s_0 : ..s_k..s_n, [(set\_wtcnt\ k\ v) \mid C], \{\ \ s_k \mapsto CLOSURE\ f\ wc_k\ nc_k\ env_k\ \ \}, \mathbf{F}, D, T > \Rightarrow$$
$$< s_0 : ..s_k..s_n, [C], \{\ \ s_k \mapsto CLOSURE\ f\ v\ nc_k\ env_k\ \ \}, \mathbf{F}, D, T >$$

It should be noted that for proper synchronization in the above examples the wait count should be set to one more than the number of **demand** instructions in that block; or *one more than the number of processes that need to be synchronized.* Thus when the wait count is one, all the parallel activity that needs to be synchronized has been completed.

8.5 Reset_ResrvClsr

This instruction resets the flag associated with the machine. The flag being set is used during the synchronization phase to indicate that some work has been retained for the sequential thread of execution, (§Section 8.8).

$$< S, [(Reset\_ResrvClsr) \mid C], \{\ G\ \}, \mathbf{F}, D, T > \Rightarrow$$
$$< S, [C], \{\ G\ \}, \mathbf{0}, D, T >$$

8.6 Demand k

The **demand** instruction generates a *child task* by *possibly* spawning it off to another processor. The offset of the object on the S stack is **k**. The object demanded is assumed to be a child of the root redex.

If the object being demanded has already been evaluated then the wait count **wc**, associated with the root redex is decremented. All operations on the wait count are assumed to be performed atomically. Otherwise (**else** part), the redex is locked so that the proper notification can be set up and a case analysis performed. **whnf** is an instruction that checks if its argument is in weak head normal form, WHNF[4].

$$< so : ..s_k : ..s_n, [(demand\ k) \mid C], \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ wc\ nc\ env \\ s_k \mapsto xxx \end{array} \right\}, \mathbf{F}, D, T > \Rightarrow$$

if whnf(s_k) **then**

$$< so : ..s_k : ..s_n, [C], \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ (wc-1)\ nc\ env \\ s_k \mapsto xxx \end{array} \right\}, \mathbf{F}, D, T >$$

else

$$< so : ..s_k..s_n, \left[\begin{array}{l} (lock\ k) : \\ (demand^*\ k) \mid \\ C \end{array} \right], \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ wc\ nc\ env \\ s_k \mapsto xxx \end{array} \right\}, \mathbf{F}, D, T >$$

In attempting to **lock** the object, there is a potential for a race condition in which the object may have got evaluated before the lock was obtained. *For this reason it is important that the lock bit is at the same position on the closure and its evaluated form.* If it was evaluated we merely decrement the wait count atomically as before.

$$< so : ..s_k..s_n, [(demand^*\ k) \mid C], \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ wc\ nc\ env \\ s_k \mapsto \widehat{xxx} \end{array} \right\}, \mathbf{F}, D, T > \Rightarrow$$

if whnf(s_k) **then**

$$< s_0 : ..s_k : ..s_n, [(unlock\ k) : C], \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ (wc-1)\ nc\ env \\ s_k \mapsto \widehat{xxx} \end{array} \right\}, \mathbf{F}, D, T >$$

If the object is not in WHNF then if it is under evaluation indicated by a **BUSY** tag, we merely set up the notification, decrement the wait count on the root redex and unlock the demanded object.

$$< s_0 : ..s_k : ..s_n, [demand^*\ k \mid C], \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ wc\ nc\ env \\ s_k \mapsto \widehat{BUSY}\ f_k\ wc_k\ nc_k\ env_k \end{array} \right\}, \mathbf{F}, D, T > \Rightarrow$$

[4] A object is in weak head normal form if it is a basic type (i.e. integer, boolean, char), a nullary construct, or a product type with possibly unevaluated components.

$$< so : ..s_k..s_n, [(unlock\ k) \mid C], \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ (wc-1)\ nc\ env \\ s_k \mapsto \widehat{BUSY}\ f_k\ wc_k\ (s_0 : nc_k)\ env_k \end{array} \right\}, \mathbf{F}, D, T >$$

If the object being demanded is unevaluated, then we can either retain this for the sequential thread of execution or spawn it off to another processor. This decision is made on the basis of the flag. If the flag is reset, then it is set to the offset of the object being demanded otherwise the latter is spawned off to another processor (Section 8.7).

$$< s_0 : ..s_k..s_n, [demand^*\ k \mid C], \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ wc\ nc\ env \\ s_k \mapsto \widehat{CLOSURE}\ f_k\ wc_k\ nc_k\ env_k \end{array} \right\}, \mathbf{0}, D, T > \Rightarrow$$
$$< so : ..s_k..s_n, [(unlock\ k) \mid C], \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ wc\ nc\ env \\ s_k \mapsto \widehat{CLOSURE}\ f_k\ wc_k\ (s_0 : nc_k)\ env_k \end{array} \right\}, \mathbf{k}, D, T >$$

$$< s_0 : ..s_k..s_n, [demand^*\ k \mid C], \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ wc\ nc\ env \\ s_k \mapsto \widehat{CLOSURE}\ f_k\ wc_k\ nc_k\ env_k \end{array} \right\}, \mathbf{k}, D, T > \Rightarrow$$
$$< so : ..s_k..s_n, \left[\begin{array}{l} (unlock\ k) : \\ (spawn\ k\ n) \mid \\ C \end{array} \right], \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ wc\ nc\ env \\ s_k \mapsto \widehat{CLOSURE}\ f_k\ wc_k\ (s_0 : nc_k)\ env_k \end{array} \right\}, \mathbf{k}, D, T >$$

8.7 Spawn k n

Spawn tries to enqueue a reference onto the task queue. **TrySpawn** is a boolean function that returns immediately with **true** if the object could be enqueued and **false** otherwise. If the object could not be enqueued then the computation is performed inline by saving the status of the machine.

$$< s_0 : ..s_k..s_n, [(spawn\ k\ n) \mid C], \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ wc\ nc\ env \\ s_k \mapsto xxx \end{array} \right\}, \mathbf{F}, D, T > \Rightarrow$$

if (TrySpawn k n) **then**

$$< s_0 : ..s_k..s_n, [C], \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ wc\ nc\ env \\ s_k \mapsto xxx \end{array} \right\}, \mathbf{F}, D, (s_k : T) >$$

else

$$< so : ..s_k..s_n, \left[\begin{array}{l} (save\_machine\ n) : \\ (eval\ k) : \\ (restore\_machine) \mid \\ C \end{array} \right], \left\{ \begin{array}{l} s_0 \mapsto BUSY\ f\ wc\ nc\ env \\ s_k \mapsto xxx \end{array} \right\}, \mathbf{F}, D, T >$$

Note that **save_machine** saves the status of the machine on the dump **D**. This in practice is a limited resource and the state transition is not as simple as indicated above. If the **save_machine** instruction is unsuccessful then the machine enters a trapped state where it must persistently try to enqueue the object spawned onto the task pool **T**. It could be the case that all processors are in this state which is a deadlock situation. Our implementation uses a simple

counter to detect deadlock at which point a fatal error is reported. *The inline evaluation of the object results in a sequential thread of execution based on the system load.*

8.8 Block n m f

The `block` instruction is where the synchronization of all the parallel activity takes place. Control may :

- Branch directly to the continuation if all the parallel activity has completed.
- May continue with some sequential thread of control if such exists
- Find something else to do.

`n` is the current top of the stack, `m` is the number of arguments for the continuation[5] and `f` is the code pointer for the continuation. When the wait count on the redex being reduced is 1, then a tail recursion optimization is performed to the the function `f` assuming that the arguments (`m` in number) were previously created on top of the evaluation stack. As mentioned in Section 8.4 the wait count must be set to one more than the number of processes being synchronized.

$$< s_0 : s1 : ..s_n, [(block\ n\ m\ f) \mid C], \{\ s_0 \mapsto BUSY\ g\ 1\ nc\ env\ \}, \mathbf{F}, D, T > \Rightarrow$$
$$< s_0 : s_{n-m+1} : ..s_n, [f \mid C], \{\ s_0 \mapsto BUSY\ g\ 0\ nc\ env\ \}, \mathbf{F}, D, T >$$

When the wait count is not one, then the root redex is locked to perform the appropriate case analysis.

$$< s_0 : ..s_n, [(block\ n\ m\ f) \mid C], \{\ s_0 \mapsto BUSY\ g\ wc\ nc\ env\ \}, \mathbf{F}, D, T > \Rightarrow$$
$$< s_0 : ..s_n, \begin{bmatrix} (lock\ 0) : \\ (block^*\ n\ m\ f) \mid \\ C \end{bmatrix}, \{\ s_0 \mapsto BUSY\ g\ wc\ nc\ env\ \}, \mathbf{F}, D, T >$$

As usual a check must be made after the lock to ensure that the wait count has not been reduced to one during the lock attempt. If it has then the root is unlocked and the tail recursion optimization performed.

$$< s_0 : s1 : ..s_n, [(block^*\ n\ m\ f) \mid C], \left\{\ s_0 \mapsto \widehat{BUSY}\ g\ 1\ nc\ env\ \right\}, \mathbf{F}, D, T > \Rightarrow$$

$$< s_0 : s_{n-m+1} : ..s_n, \begin{bmatrix} (unlock\ 0) \\ f \mid \\ C \end{bmatrix}, \left\{\ s_0 \mapsto \widehat{BUSY}\ g\ 0\ nc\ env\ \right\}, \mathbf{F}, D, T >$$

[5] We assume that the arguments to the continuation have been built on top of the stack.

The interesting case is when the wait count is not 1. It is either the case that a a closure was reserved to continue the sequential thread of execution or all objects that were demanded were under evaluation. We discover this information from the flag, F. In either case the wait count is decremented, and the continuation built.

$$< s_0 : s1 : ..s_n, [(block^*\ n\ m\ f) \mid C], \{\ s_0 \mapsto \widehat{BUSY}\ g\ k\ nc\ env\ \}, \mathbf{F}, D, T > \Rightarrow$$
$$< [], \begin{bmatrix} (unlock\ 0) : \\ (continue) \mid C \end{bmatrix}, \begin{Bmatrix} s_0 \mapsto \widehat{BUSY}\ f\ (k-1)\ nc\ env_{new} \\ env_{new} \mapsto s_{n-m+1} : ..s_n \end{Bmatrix}, \mathbf{F}, D, T >$$

If the flag is still reset then we go back to the top level **eval** which fetches another task from the task pool T. This means that nothing was kept for the sequential thread of execution.

$$< s_0 : s1 : ..s_n, [(continue) \mid C], \{\ s_0 \mapsto BUSY\ g\ k\ nc\ env\ \}, 0, D, T > \Rightarrow$$
$$< [], [C], \{\ s_0 \mapsto BUSY\ g\ k\ nc\ env\ \}, \mathbf{F}, D, T >$$

If the flag is set the the top of the evaluation stack is replaced with the object demanded and a return to the top level **eval** is performed which proceeds with the unwind, etc.

$$< s_0 : ..s_k : ..s_n, [(continue) \mid C], \{\ s_0 \mapsto BUSY\ g\ k\ nc\ env\ \}, \mathbf{k}, D, T > \Rightarrow$$
$$< s_k, [C], \{\ s_0 \mapsto BUSY\ g\ k\ nc\ env\ \}, \mathbf{k}, D, T >$$

9 Optimizations

It is possible to optimize the basic model to improve locality and memory usage.

9.1 Data Caching

ML objects are typically represented as *boxed* or *unboxed* structures. An unboxed structure is used to represent nullary constructors and small integers that can be represented in a word of the machine (in our case 32 bits). Boxed structures are used to represent product types and have a descriptor (32 bits) followed by a vector of the component types. If all references are to word boundaries and all allocation are multiples of a word, then the least significant two bits of a reference will always be zero. These can be used as a tag to discriminate between refrences, boxed and unboxed structures. This further means that unboxed structures can be created on the S stack, rather than on the heap as in the traditional G-machine. The advantages are:

- Reduction in memory requirements since unboxed objects may be built on the S stack.

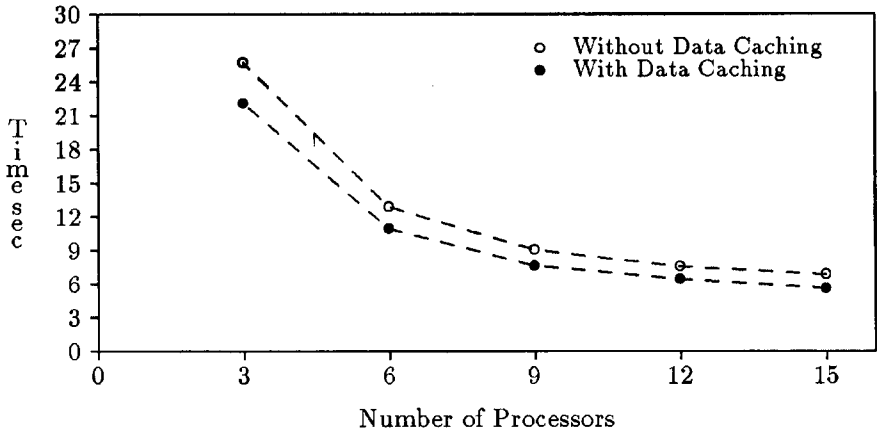

Figure 2: Sieve Program with Data Caching

- When a closure is created an argument block is formed by grouping refrences on the S stack. If data caching is used, then the data goes directly into the argument block and when unwound on some remote processor becomes a local access. This should be compared to exporting a global reference in the argument block. On a shared memory machine this results in less switch traffic.

The main drawback is that a check now has to be made each time a reference is followed from the S stack.

All programs tested showed an improvement ranging from 11 - 20 % keeping everything else constant (see Figures 9.1 and 9.1).

9.2 Argument Block Reusage

When a continuation is created a new argument block is created on the heap as indicated by the transition in Section 8.6. If the old argument block can accommodate the new, then it can be reused. While this saves on the memory used, there is an advantage to creating a new environment block; namely the new environment will be allocated on the local processor and the subseqent writes will be local.

9.3 Two Level Scheduling

To avoid contention on the centralized task pool, a small local task pool can be maintained. Thus excess work spills over from the local task pool to the

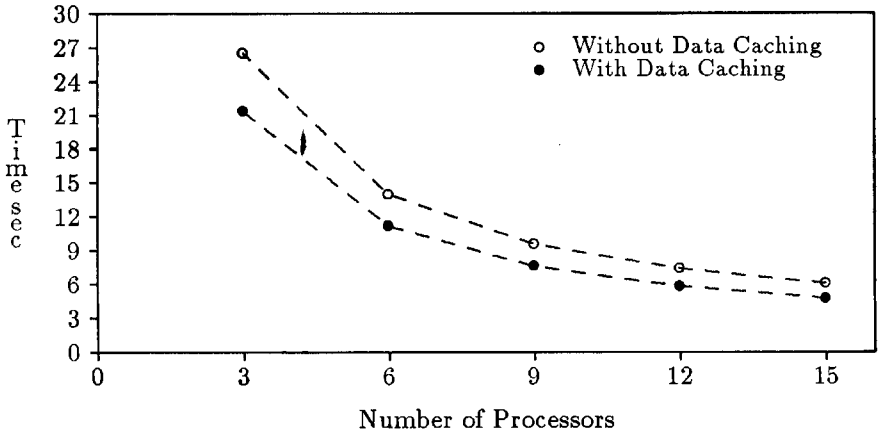

Figure 3: 8 Queens Program with Data Caching

centralized task pool. Search for work begins at the local task pool and ends at the centralized task pool. We have found in the examples tried that a local task pool greater than 1 degrades the performance. Figure 9.3 and 9.3 shows the results obtained.

10 Future Directions

- The above implementation was developed in the context of the Chrysalis operating system on the BBN Butterfly. It will be interesting to see how this would turn out in the context of the Mach[1] operating system.
- Under Mach it should be possible to kill unwanted tasks by sending them a signal. It should be possible to implement a model for speculative computation.
- It was observered that the node on which the task pool resided performed approximately half the number of reductions that the other nodes were able to do. This may suggest that a distributed task pool may work a little better, with a load balancing scheme to migrate tasks around.

11 Concluding Remarks

In this Chapter we have described an abstract machine and some simple instructions that control the complexities of parallel execution. This is an important

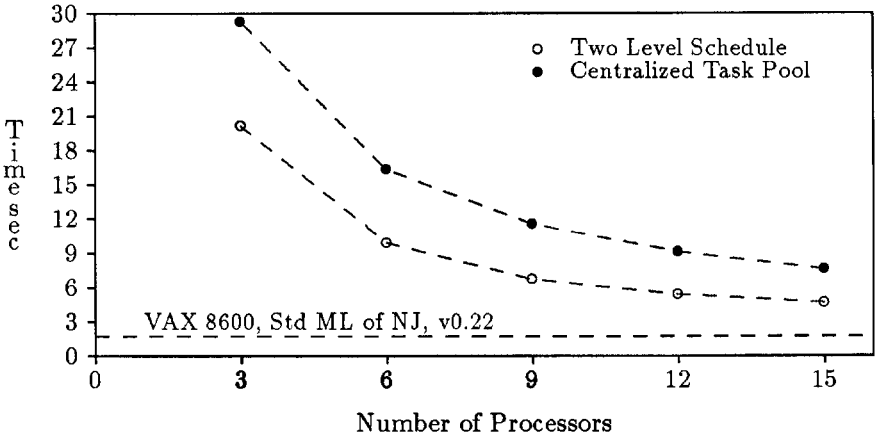

Figure 4: Two Level Scheduling and the Sieve Program

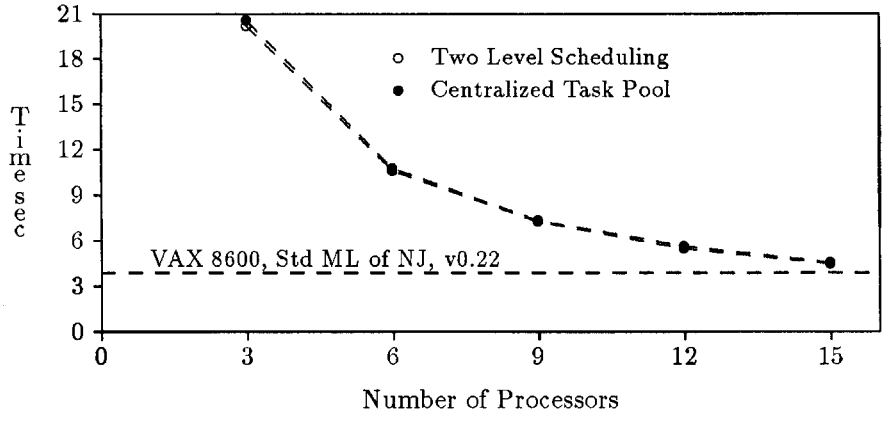

Figure 5: Two Level Schedule and the 8 Queens Program

step towards building a compiler for multiprocessor machines. Figures 9.3 and 9.3 show the performance against the Standard ML of New Jersey compiler, running on a VAX 8600. Section 12 shows the functional specification of the sieve program for prime numbers. At first sight it is not obvious where the major parallelism is or how it should be exploited or if parallelism exists in the first place or that the performance of this program dependents of how fast `(filter 2)` can execute. It is encouraging that a straightforward compilation to our abstract machine should show such significant speedup and performance matching the VAX 8600.

References

[1] Avadis Tevaninan, Jr., Rashid, R. F., Golub, D. B., Black, D. L., and Young, M. W. Mach threads and the unix kernel: the battle for control. Tech. Rep. CMU-CS-87-149, Carnegie-Mellon University, August 1987.

[2] Johnsson, T. Efficient compilation of lazy evaluation. In *Proc. Symp. on Compiler Construction* (Montreal, 1984), ACM SIGPLAN.

12 Sieve Program

12.1 Functional Specification

12.2 Code Generated

```
r_filter:
     set_wtcnt(0,2);
     Reset_ResrcClsr();
     demand(2);              % demand 2nd argument
     block(3, 2, g_filter);

g_filter:
     isNIL(2)
     jfalse L1
          updtNIL();
     L1:
          set_wtcnt(0, 3);
          Reset_ResrvClsr();
          demand(1);         % demand 1st argument
          push(2, 3);
          hd(3);
          demand(3);         % demand head of second argument
          block(3, 2, g_filter1);

g_filter1:
     push(2, 3);
     hd(3);                  % x
     push(2, 4);
     tl(4);                  % xs
     getInt(3);              % v[0] =x
     getInt(4);              % v[1] = p
     mod(1);                 % v[0] := v[0] mod v[1]
     eqConst(0, 0);          % v[0] == 0
     jfalse L2
          set_wtcnt(0, 2);
          Reset_ResrvClsr();
          push(1, 5);
          push(4, 6);
          demand(6, 0);      % demand xs
          block(7, 2, g_filter);
     L2:
          push(3, 5);
          push(1, 6);
```

```
            push(4, 7);
            mkArgs(2, 8);
            mkClosure(r_filter, 6);
            updtStrictCons(5); % spawns in anticipation its components


g_sieve()
      isNIL(1)
      jfalse L3
            updtNIL();
      L3:
            push(1, 2);
            hd(2);
            push(2, 3);
            push(1, 4);
            tl(4);
            mkArgs(2, 5);
            mkClosure(r_filter, 3);
            mkClosure(r_sieve, 3);
            updtStrictCons(2);

r_sieve():
      set_wtcnt(0, 2);
      Reset_ResrvClsr();
      demand(1);
      block(2, 1, g_sieve);
```