The HDG-machine: a highly distributed graph-reducer for a transputer network*

HUGH KINGDON, DAVID R. LESTER† AND GEOFFREY L. BURN

GEC-Marconi Ltd, Hirst Research Centre, East Lane, Wembley, HA9 7PP

Distributed implementations of programming languages with implicit parallelism hold out the prospect that the parallel programs are immediately scalable. This paper presents some of the results of our part of Esprit 415, in which we considered the implementation of lazy functional programming languages on distributed architectures.

A compiler and abstract machine were designed to achieve this goal. The abstract parallel machine was formally specified, using Miranda.‡ Each instruction of the abstract machine was then implemented as a macro in the Transputer Assembler. Although macro expansion of the code results in non-optimal code generation, use of the Miranda specification makes it possible to validate the compiler before the Transputer code is generated.

The hardware currently available consists of five T800-25s, each board having 16 Mbytes of memory. Benchmark timings using this hardware are given. In spite of the straightforward code-generation, the resulting system compares favourably with more sophisticated sequential implementations, such as that of LML.

Received September 1990

1. INTRODUCTION

Popular mythology about implementations of lazy functional languages is that they are slow when compared with more traditional languages such as C and Pascal. Early implementations *were* slow, for two reasons:

- they were largely interpretive; and
- the lazy semantics of the languages requires that arguments to functions are not evaluated until their values are needed; thus imposing time and memory overheads, and restricting any parallelism in an implementation.

Compiler technology has now advanced sufficiently so that lazy functional programs run respectably fast when compared with those written in more traditional languages, overcoming the overheads due to interpretation. An excellent collection of papers, which includes ones on implementation techniques, can be found in the April 1989 issue of this journal, the special issue on Functional Programming.

Our work began by solving the second problem. Realising that some functions needed to evaluate their arguments, so that the overheads of passing them unevaluated were unnecessary, we worked on a semantically sound technique for determining when this was the case. This resulted in the *evaluation transformer* model of reduction, which is able to capitalise on the information about how functions use their arguments in order to obtain more efficient sequential and parallel implementations.[8,9,14]

The evaluation transformer model says how the normal model of reduction can be modified in order to allow evaluated arguments to be passed to functions in a sequential implementation, and argument expressions to be evaluated in parallel in a parallel implementation. Being a modification of lazy evaluation, all the compiler technology for sequential implementations can be used as a basis for an implementation using the evaluation transformer model of reduction, on both sequential and parallel machines. It can be used on a parallel machine because such an implementation is best constructed using the best sequential compiler technology and placing it in a harness which supports task management and communication (cf. the observations made for parallel prolog implements in Ref. 29 and the implementation of a combined logic and functional language on parallel machines in Ref. 4).

At the time we were developing our parallel reduction model, most implementations were described by defining an abstract machine and showing how to compile functional languages to that abstract machine–see Refs 2, 17, 18, 20 and 21 for example – and then compiling the abstract machine code to machine code for a real computer. In fact, the LML compiler, which produces some of the most efficient code for functional programs, works in this way. More recently, implementations have begun to be described in terms of more conventional compiler technology, reflecting the growing understanding of the implementation of lazy functional language – see Refs 6, 7, 26 and 27 for example.We described how the evaluation transformer information could be used to compile parallel code for functional languages in terms of an abstract machine.[11,12] This was didactically convenient, and we note that the ideas can be adapted to implementations using more standard compiler technology.

In functional language systems, the compiler organises for memory to be allocated to store the structures representing unevaluated arguments and data objects. Typically this information is kept in a graphical data structure, so the compiler is unable to determine when the allocated store should be released. Therefore implementations include a *garbage collector* which periodically reclaims the storage occupied by data that is no longer

* Research undertaken while the authors were employed by GEC Hirst Research Centre, and partially funded by ESPRIT Project 415: 'Parallel Architectures and Languages for AIP – A VLSI-Directed Approach'.

† To whom all enquiries about this paper should be addressed.

‡ Miranda is a trade mark of Research Software,

being used by the program. We designed a novel garbage collection algorithm which is well-suited to parallel language implementations.[23]

Our abstract machine specification gave each process a stack, which therefore formed part of the state of the process. When a process is switched, its state must be saved, and the state of the new process loaded. To have the entire stack as part of the state is a large overhead at task switching time. Instead, we represent the stack as a linked list of stack frames, and keep a pointer to the stack frame currently being used by a process.[24] Saving the state of the stack then reduces to storing the stack frame pointer. A similar solution has been adopted in.[2] In some ways, this is very natural on the transputer, which has the concept of a *workspace*, the area of memory where the data for the current computation resides; and the *workspace pointer*, which points to the base address of that space.

With all these techniques in place, the time came when we had to try them out on a real parallel machine. We had access to a multi-transputer system on which it could be developed. The implementation had two goals:

- to test out our ideas by providing a simple prototype implementation; and
- explore some of the issues concerning implementing functional languages in parallel on the transputer architecture.

The main purpose of this paper is to describe some of our techniques, and record some of our experiences.

Being a simple prototype implementation, negative results about implementation speeds would not necessarily be conclusive. Nevertheless, our implementation achieved speeds comparable with those that might be obtained from LML, and so there is definite hope that a real implementation using our ideas could run significantly fast.

The next two sections discuss the framework of the transputer implementation in more detail, and the rest of the paper is devoted to discussing particularities of it. In the next section, we describe in a bit more detail the evaluation transformer model of reduction, our garbage collection algorithm, and our way of handling stacks in a parallel, distributed machine. Those who are interested in a more general overview of the work of our project are referred to the survey article,[12] and again we refer the reader to the April 1989 issue of this journal which contains a number of excellent papers on the analysis, use and implementation of functional languages.

2. BACKGROUND

2.1 The evaluation transformer model of reduction

Having to build data structures to pass unevaluated arguments to functions is an overhead that is not present in a system which passes its arguments by value. It also restricts any parallelism in an implementation. The key points of the evaluation transformer model of reduction are that:

- some functions definitely need to evaluate their arguments; and
- the amount of evaluation that is needed of an argument expression may depend on the amount of evaluation required of the function application of which it is a subexpression.

For example, the function + needs to evaluate both of its arguments. consider further the function `append`:

```
append []     ys=ys
append (x:xs) ys=x:append xs ys
```

which concatenates two lists together. The consumer of the output of this function can request varying amounts of the result to be produced, and the amount requested affects the amount of evaluation that must be done to the argument expressions. For example, if the first element of the result must be obtained, then only the first argument needs any evaluation, and only needs its first element to be evaluated. If however, the consumer needs to evaluate all of the elements of the result list, then all of the elements of both argument lists also need to be evaluated.

In a sequential implementation, this information is used to evaluate the argument to the required extent, before applying the function, and so saving the cost of building a data structure for the argument in the heap. The information is used in a parallel implementation by creating a parallel process to evaluate the argument in parallel with the function application.

The information about how functions use their arguments can be determined using a semantically sound analysis technique, such as abstract interpretation[8,9,14] or projection analysis.[13,28] We refer the reader to these papers for further details of the model and the compilation of programs using evaluation transformer information.

2.2 Supporting the evaluation transformer model of reduction

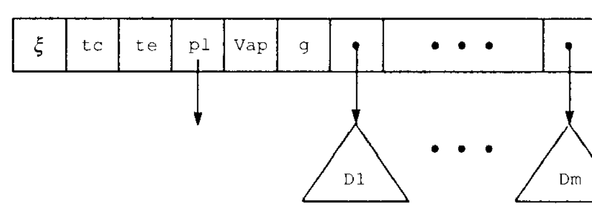

Figure 1. The `Vap` **node for the application** (`g D1...Dm`).

In our parallel implementation, function applications are stored as graphs. The graph of the application is shown schematically in Fig. 1. It has four status fields:

ξ the amount of evaluation requested of the expression so far,

`tc` true if and only if a task has been created to evaluate the expression,

`te` true if and only if the evaluation of the expression has begun (although it may be temporarily suspended), and

`pl` a pointer to the *pending list*, a list of tasks waiting for the evaluation of the expression to be completed,

the tag `Vap` to indicate that it is storing a function application, a way of accessing the code for the function being applied (pictorially represented by putting g in the box after the tag), and pointers to the graphs for the argument expressions (pictorially represented by pointers to triangles containing the name of the expression).

19-2

These fields are motivated in Ref. 10 and a complete specification of the abstract machine given in Ref. 25.

2.3 Specifying the abstract machine

Most specifications for abstract machines have represented the state of the machine as a tuple, and described the instructions by giving the modified state. We found this method to be unwieldy when used for a parallel machine, and instead wrote our specification in a functional language. The resultant specification was much easier to write and read, compare the difference between Ref. 11 and Ref. 25, and had the further advantage that it was executable. It may seem odd to talk about debugging specifications, but this is precisely the reason that we wanted an executable specification. It is possible that a formal proof of correctness for our implementation could be given, perhaps adapting the result of Ref. 22. Like most other projects, we ducked the responsibility, and went for debugging the specification instead!

2.4 A stackless implementation

The traditional way of implementing functional languages (and indeed, any language with a function call mechanism) is to have each activation record kept on a stack, and so the stack is part of the state of a process. Such a stack, consisting of three activation records, is represented in the first part of Fig. 2. The heavy black

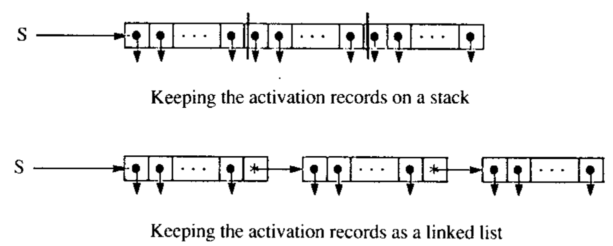

Figure 2. Two different ways of keeping activation records for a process (`g D1...Dm`).

lines indicate the end of one activation record and the start of another. In a parallel machine, it is more convenient to store the activation records in the heap, and link each activation record into a list. We keep a pointer to the current activation record, and each record points to the one which was activated immediately before it (see the second part of Fig. 2). Process switches can now be very fast, because saving the state of the stack only involves saving the pointer to its current activation record. Furthermore, we make sure the `Vap` nodes that are created to store a function application are big enough to be the activation records when that expression is evaluated, and the evaluation of the expression takes place in the space used by the `Vap` node. The only problem with this is determining how much space is needed. Lester developed an analysis technique which gives this information.[24] A simpler approach has been taken in Ref. 3, where a `Vap` node may occasionally need to be extended.

2.5 Garbage collection

Bevan and the two Watsons independently discovered a very elegant garbage collection algorithm for distributed implementations of languages with heap allocated storage.[5,30] It consisted of giving each object a reference count, and each pointer a weight; the sum of the weights of the pointers to an object is equal to the reference count of that object. When a pointer is copied, its weight is shared evenly between the new copy and the original pointer (with an indirection node being introduced when the weight reaches one). Dealing with reference counts in this way means that only decrement reference count messages are needed, removing the need for expensive protocols that are used in other reference counting algorithms to handle races between increment and decrement messages.

Unfortunately, copying garbage collectors are inherently more efficient,[1] as collection takes time proportional to the number of live objects. In contrast, any mark/scan algorithm takes time proportional to the heap memory size. Hartel has shown experimentally that this asymptotic behaviour occurs even in quite small heaps.[19] Furthermore, at least some of the efficiency of the G-machine[21] may be attributed to the ease with which heap may be allocated in a copying scheme. Efficiency considerations therefore dictate that we choose a collector with the following properties: locally it should do copying collection, for inter-processor references it should do weighted reference counting.

Lester designed an algorithm which has the advantages of both algorithms.[23] In his algorithm, reference counting is only used for interprocessor references. Two new node types are added to the machine:

Output indirections which point to non-local objects and have a weight, and

Input indirections which are pointed at by non-local references and point to local object; they have a reference count.

The heap therefore is divided into three logically distinct parts, as shown in Figure 3. Input indirections

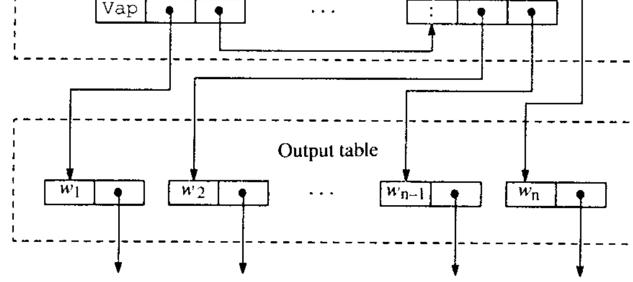

Figure 3. The logical structure of the heap.

are kept in the input table, output indirections in the output table, and all pointers in the third part of the heap, the local graph structure, are local. doing things in this way has some further advantages:

- the output indirections are a natural place to store a copy of the remote expression when it has been produced, so that all pointers in a local store that point to the remote node share the copy; and
- the local nodes and output indirection nodes can be kept in a heap which is managed by a semi-space allocator and copying collector.

This concludes our necessarily brief overview of the higher-level details of our implementation; we now look at the details of the tasks that are used by the processors.

3. HDG-MACHINE TASKS

We refer to the function application illustrated in Fig. 1 as a `Vap` (Variable size APlication) node of the graph. It is the job of a task to evaluate a `Vap` node until it reaches a result – such as the integer 3. Things are slightly more complicated for tasks that result in data structure values, because we use the evaluator to control the amount of evaluation requested.

One of the key ideas of the HDG-Machine is that the `Vap` nodes can be used to implement function calls. When a function call is attempted we open a new `Vap` node. We store the old program counter on the old `Vap` node, and insert a return pointer on the newly opened `Vap`; this points back to the old `Vap` node. The mechanism is shown in abstract form in Fig. 2.

3.1 Generating tasks

As an alternative to the sequential evaluation of new `Vap` nodes, we can instead create a task to perform this evaluation. We refer to this operation as the *spawning* of a task. This is how new tasks are generated. To provide parallelism, such tasks will be exported to processors with insufficient work.

3.2 The task pools

When program execution begins a single task exists. As the program executes more tasks are created; these tasks are store in a task pool. The task pool is implemented as three separate task pools.

Migratable. A newly created task is initially placed here. These tasks are the only candidate for migration to other processors.

Active. Tasks received from other processors are placed here. When a task is blocked it moves to the blocked task pool. When the active task pool is empty, tasks are moved from the migratable task pool. If this too is empty, tasks are requested from a neighbouring processor's migratable task pool. Active tasks may not be migrated.

Blocked. Whenever a task is blocked, because it is waiting for a result from some other task, it is placed here. When the result is available the task is placed in the active task pool.

Tasks in the migratable task pool are kept distinct from other tasks that can be executed immediately, because their state is small. This means that exporting them is not going to involve transferring an unbounded amount of state information to the remote processor.

The migratable task pool is implemented as a doubly linked-list. Tasks are exported from one end (the oldest task is exported) and moved to the local active task pool from the other (the youngest task is moved).

The task migration strategy used is a simple one: processors which do not have any executable tasks request tasks from each of their neighbours in turn. Each neighbour either rejects the request for tasks on the grounds that it has no tasks to donate or it passes back a task, in this case the task requests are stopped. In order to prevent tasks cycling around the machine without performing useful work, when a task is received from a remote processor, it is placed in the active task pool, thus ensuring that it will be executed on the receiving processor.

The specification of the HDG-machine contains a blocked task pool. There is no blocked task pool in the implementation: instead blocked tasks are referenced only from the node which caused the task to be blocked. When the evaluation of a task has completed, the corresponding node is investigated to see if any tasks were waiting on the result of the evaluation; if this is the case the tasks are restarted by placing them in the active task pool again.

3.3 Transputer details for HDG-machine tasks

There are only two real registers on the transputer. They are `Iptr`, the program counter, and `Wptr`, the workspace pointer. We use the first as the program counter and the other points to the current `Vap` node (see Subsection 2.4). This gives us fast access to the contents of the node at the expense of:

- some extra state on each node, and
- an awkward access pattern to global quantities.

The first of these points means that we must have two words immediately below the workspace for storing state when descheduled. The second point is only a problem because the occam linker does not permit the absolute placing of code or data in the transputer's memory. Ideally, we would like to access the global quantity at absolute address `x` by:

```
mint          /* Load constant
                 -(2^31) */
ldnl x-(2^31) /* Load indirect
                 with offset to x */
```

However, the value of `x` is not a link-time constant to the occam linker and so the above code is not permitted.

When we initially implemented the machine we placed the local state of the node at negative locations from the tag position, and the *S* stack in the positive locations. An experiment was conducted, in which the order of items in the state was reversed. Because the loading small negative offsets is more costly than loading small positive offsets, faster register access is obtained in the modified scheme.* The speed-up was 11 per cent.

We now move on to consider the low-level details involved in using controlled shared memory access by concurrent processes in a transputer.

* In the transputer, loading and storing at locations with offsets between zero and 15 can occur in one instruction. Two instructions are required to access locations with offsets between −1 and −15.

4. TRANSPUTER PROCESSES

4.1 Overview

Each transputer emulates one processing element from the abstract machine. These processing elements can be thought of as consisting of three parts.

- A garbage collected heap.
- An input and an output process for each channel; there are therefore three of each on each transputer.
- An evaluation process.

The heap is used to store a number of data structures.

- Those necessary for running the evolution process, e.g. Vap nodes, integers, etc.
- The heap is also used to hold the task pools, as these are dynamic data structures, and may grow arbitrarily large. The active task pool must be shared between the reduction process and the I/O processes, as each needs to extract tasks from this data structure.
- Message queues of unsent messages.

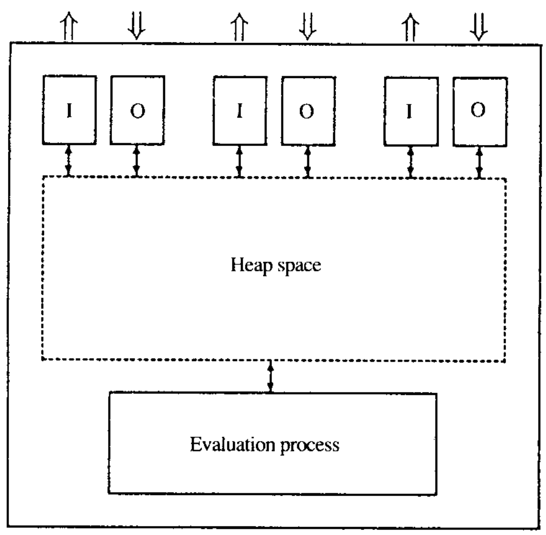

Figure 4. An example of the state of a local processor.

Because all of the processes – reduction and I/O – require access to the heap to read and modifiy objects there, we must make sure that this access is controlled. That is, we would like to ensure that the heap is in a stable state when a process performs an operation on it. The natural CSP model for this would be to have a heap process, to which each I/O process and the reduction process would send requests using messages. The responses to these messages would then be returned using messages. The only problem with this approach – and for us it was a serious one – is that the reduction process spends a significant time accessing the heap. For this reason we chose to have the heap as a shared resource, *under strict control.*

4.2 I/O processes

To ensure that messages are sent as soon as possible it is important to run the I/O processes at high priority. This further complicates the access to the heap; now we must control access from processes running at two different priorities.

Each output process has a queue of messages awaiting transmission; the queue is stored in the heap. The activation of the output process is controlled by a counting semaphore. The sequence of actions that occur when a message is sent are listed below.

1. When a task is evaluating an expression, the reduction process needs to send a message to another processor. This can occur when an expression pointed to by an output indirection is required evaluated by the task.
2. The reduction process must now create a message holder which contains the type of the message, some operands to the message and the address of any expressions to be sent.
3. Depending on the destination, the reduction process places the message holder in one of the three output queues. A semaphore is signalled to indicate that another message has been placed on the queue.
4. The reduction process may be able to continue evaluation of the current task, or may have to begin evaluation of a different task.
5. An output process is restarted by the semaphore action. It creates an exportable version of the message, storing the result in a buffer. The message is transmitted from the buffer across the INMOS link in a single block.
6. The input process of the receiving transputer reads the message into an input buffer. The action of the input process depends on the type of message received, but typically it involves some access to the heap and perhaps the creation of a task.

In order to export a task we need to make *exportable copies* of the Vap node associated with a task. The only complication here is that we must create input indirections for each of the pointers in the Vap node. The input process of the receiving processor, creates the corresponding output indirection nodes.

4.3 Garbage collection and heap consistency

In this section we investigate some of the practical problems with the transputer implementation of the garbage collection algorithm from,[23] described in Section 2.5. There are two major problems to be overcome. Firstly we must organise the data structures, so that the problems of overflow are minimised. Secondly we must maintain a consistent heap in the presence of multiple concurrent processes, all of which share the heap as a common resource. We deal with these problems in order.

Because we have a local semi-space collector, it is possible for high-priority and low-priority allocation to occur from opposite ends of the active semi-space. This can be seen in Fig. 5. The low priority reduction process allocates from the part labelled R, the high-priority I/O processes allocate from the part labelled C. The input indirections are held in I_A and I_B.

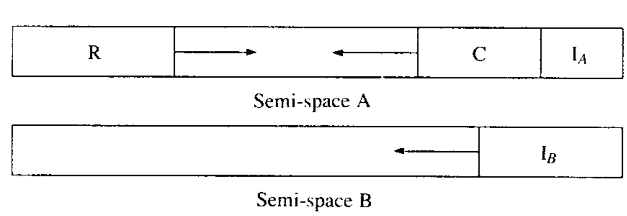

Figure 5. Heap allocation in a dual-priority system.

A garbage collection is induced whenever the memory labelled R gets too close to that labelled C in Fig. 5. Because the high-priority process may have interrupted the reduction process anywhere, there may be a partially filled-in node in the heap. We must therefore resume the low-priority process before initiating a garbage collection. Fortunately, a result from Ref. 24 allows us to deduce the largest heap allocation that may be performed by the reduction machine and we can therefore place an upper bound on the uncertainty in position of the bottom of heap pointer. Provided there is room for both the largest possible allocation by the reduction process and the size of the block required by the high-priority process, the heap allocation can succeed.

The output indirection nodes may be placed in the heap – provided that we link them all together. This linked list is searched after the copying phase of the semi-space collector. Any output indirection nodes that have not been copied are now garbage; a decrement reference count message must therefore be sent to the relevant processor.

The input indirection table is kept in two parts, labelled I_A and I_B in Fig. 5, at the top of each semi-space. There is a free-list from which input indirection nodes are allocated. If this is empty, the input indirection table may be extended in the inactive semi-space (as shown in Fig. 5, where I_B is shown growing downwards). Input indirection nodes are deleted when their reference count falls to zero – the locations are then chained into the free-list. After the copying phase of the garbage collector the free-list is examined in an attempt to return as much as possible of the input indirection table to the heap semi-spaces. The input indirection nodes must have fixed addresses because they are referenced externally. The alternative would be to broadcast an input indirection nodes new location.

The second problem is to maintain heap consistency with the above data structures. The heap must be in a stable state when a garbage collection is initiated. There are two parts to this.

1. There must be no nodes in the heap which are only partially filled in.
2. The registers of the descheduled process must be in a consistent state. To be in such a state, it must be known which machine registers are pointers.

The first criterion means that a node never contains bad pointers, and that therefore we are always permitted to follow pointers in the heap. The second means that we may consistently update the pointers in the descheduled process's register set.

The use of two priority levels creates extra problems: the reduction process may be interrupted at any point, including the few actions which must be completely indivisible. These actions can only be made indivisible by also running at high priority; to achieve this a routine which changes the priority of a process is used; this is the subject of the next subsection.

4.4 The change-priority routines

In this subsection we will look at two ways of efficiently changing a low-priority process into a high-priority process and back again. We will consider the general solution to this problem, and then look at a faster solution which is suitable for implementing certain kinds of critical sections.

To understand the solution, we must look at the process-scheduling methods employed by the transputer. There are two transputer scheduling instructions that we make use of: `runp` and `stopp`. The first, `runp`, causes a process to be added to the relevant transputer process queue. The workspace of this process is pointed to by the transputer `Areg` ANDed with -2, the least significant bit being used to determine the priority level (low priority is 1 and high priority is 0). The new program counter is taken from the first negative location of this new workspace. The `stopp` instruction stops the current process saving the program counter in the first negative location of the old workspace.

When a low-priority process is interrupted by a high-priority process, its state is preserved in locations near the bottom of memory. We can therefore recover them by loading from these locations if we desire. It is also possible to set up these locations for passing values from the high-priority process to the low-priority process, *provided that we know which low-priority process is currently suspended.* We now look at codes that will perform a general change of the priority of the executing process from low to high. This will work even if the high-priority process is suspended whilst waiting on communications channels.

```
/* code for a general Lo-Hi change */
   ldc (L2-L1); ldpi
/* load PC ... */
L1: stl -1; ldlp 0
/* ... into workspace */
   runp
L2:
   mint; ldnlp KillIptr
/* a pre-initialised stopp instruction.*/
   mint; stnl IptrSaveLoc
/* currently 12 */
   mint; ldnlp KillWptr
/* a dummy work space */
   mint; stnl WptrSaveLoc
/* currently 11 */
   mint; ldnl BregSaveLoc
/* currently 14 */
   mint; ldnl AregSaveLoc
/* currently 13 */
```

The final two lines are optional and can be used to pass the values of the `Areg` and `Breg` from the low-priority process to the high-priority process. It relies on a pre-initialised area of memory which contains a `stopp` instruction, and an area in which it can put the program counter on executing this instruction. The code length and execution time for the above code are given in the following table. The extra code size and execution time are associated with the dummy workspace and the execution of a `stopp` instruction. The first column represents the number of transputer registers passed.

Parameters	Code length (bytes)	Execution time (no wait states)
0	16+5	28+11
1	18+5	31+11
2	20+5	34+11

At the end of the high-priority code we can resume execution at low priority by executing the following code sequence.

```
/* code for a general Hi-Lo change */
    ldlp 0; adc 1; runp; stopp
```

In this case it is impossible to pass any registers to the low-priority process. The code length and execution time are:

Parameter	Code length (bytes)	Execution time (no wait states)
0	4	23

If we can guarantee that the high-priority process never deschedules, then a faster solution is possible. It relies on the fact that the suspended low-priority process will never be executed.

```
/* code for a fast Lo-Hi change */
    ldc (L2-L1); ldpi
/* load PC ... */
L1: stl-1; ldlp 0
/* ... into workspace */
    runp
L2:
    mint; ldnl BregSaveLoc
/* currently 14 */
    mint; ldnl AregSaveLoc
/* currently 13 */
```

It is again possible to pass the transputer registers from the low-priority process to the high-priority process. The code length and execution times are given in the following table.

Parameters	Code length (bytes)	Execution time (no wait states)
0	6	16
1	8	19
2	10	22

To change back to low priority we may use the following code.

```
/* code for a fast Hi-Lo change */
    mint; stnl AregSaveLoc
/* currently 13 */
    mint; stnl BregSaveLoc
/* currently 14 */
    ldc (L4-L3); ldpi
/* load resumption PC */
L3: mint; stnl IptrSaveLoc
/* Currently 12 */
    stopp
L4:
```

This time it is possible to pass some of the registers back to the low-priority process. The code lengths and execution times are:

Parameters	Code length (bytes)	Execution time (no wait states)
0	5	17
1	7	20
2	9	23

This concludes a presentation of the low level details associated with changing the priority levels of currently executing processes. We are mainly interested in the second version, as it permits interlocking, *via* semaphores, of processes at different priorities.

4.5 Testing tags

Unlike traditional languages, tags are not used to determine types at run time, but are used to distinguish different sorts of objects of the same type. We may think of them as selectors for a union type in the 'C' programming language. In a lazy language they are also used to mark closures. Closures may be thought of as recipes that tell us how to compute a value, when we have not already done so.

Tag testing is a common operation in a lazy functional language, because many instructions depend on the type of data that is an argument to the instruction.

One of the current debates within the functional programming community concerns the representation of these tags. It is common ground that some form of object oriented approach is required. The tag is then a pointer to a table of addresses. Each entry in the table corresponds to an operation to be performed. For example the first entry might evaluate the object, the second might print it, and so on.

Augustsson and Johnsson content that one should test the least significant bit of the tag, which indicates whether a node is already evaluated.[3] They claim that some 70 per cent of the time that this bit is tested, the nodes *are* evaluated. An alternative view is that of Peyton Jones and Salkild.[26] They claim that it is easier to always jump through the tag, in a similar way to the traditional G-machine.

In theory Augustsson and Johnsson are right. Their scheme results in at most one pipe-line break and, with the right sort of hardware, it is possible than an intelligent prefetch could hide most of this penalty. In practice, on traditional hardware, Peyton Jones and Salkild are correct. This is the case even though there are always two pipe-line breaks.

We now perform a *post hoc* justification of the approach we took, which is that of Peyton Jones and Salkild. To do so, we will give the code and timings for each approach.

```
    ldnl  0      /* load tag */
    ldnl  offset /* offset into table */
    gcall        /*jump to code */
/*  If the item is unevaluated, we
    immediately return: */
    gcall        /* which returns */
```

Provided that `offset` lies between 0 and 16 and assuming no memory fetch cycles, this code executes in 10 machine cycles. The in-line part of the code is 3 bytes long. As an alternative, we will look at codes which tests the least significant bit of the tag.

```
    ldnl  0      /* load tag */
    dup          /* create a
                    copy */
    ldc   1
    and
    cj    $L     /* jumps on
                    bit clear */
    adc   -1     /* clears
                    bit in pointer */
    ldnl  offset /* offset
                    into table */
    gcall        /* jump to
                    code */
$L:
/* code as before */
```

The in-line code size is now 9 bytes. If the branch is taken the above executes in 9 cycles. If it is not taken the code executes in 15 + 4 cycles, the extra four cycles being required to return. The expected time for this code, using Augustsson's 70 per cent rule, is then 12 cycles.

Therefore, the expected time to execute is the same in both cases, although Augustsson's code is 4 bytes larger.

4.6 Making tag testing more efficient

In the paper specifying the parallel abstract machine,[23] each node in the program graph has a number of fields – an evaluator, a pending list, a task-executing flag, and a tagged object. The `Vap` nodes require all three extra fields. To speed up the selection of the correct operation, the evaluator and the task-executing flag are made part of the object's tag. This means there are more tag tables, for `Vap` nodes, but there is no need for run-time tests on these extra flags.

We also note – as discussed in the specification paper – that some of these fields are not needed on some types of objects. For example, integers do not need to have evaluators, pending lists, or a task-executing flag. The HDG-machine therefore does not have these fields for integers.

We now discuss the performance of the resulting system.

5. PERFORMANCE OF THE SYSTEM

After describing the hardware and software used in the benchmarking, we analyse the results obtained from the HDG-machine. The following points are to be stressed.

- There is no limit to the grain size of a task, i.e. a task can be arbitrarily small.
- The only way to introduce parallelism is via annotations for evaluation transformers.

This is clearly less than optimal. For example we could re-write any of the benchmark programs so that only tasks of a reasonable size were created. It is also the case that hand annotation of the programs for parallelism would result in better performance.*

The following restrictions on the applicability of the results should also be borne in mind.

- The implementation uses only four Transputers. It may be that the parallelism does not scale.
- The abstract machine is based on a slightly outdated technology (similar to the $\langle v, G \rangle$-machine).[3] This may mean that the costs of exporting work have been understated.
- The code generation is naïve, again causing an understatement of the costs of exporting work. Better code generation will result in an increase in the relative cost of the parallelism overheads.

The first restriction is the most worrying. The ZAPP project[15] found that divide-and-conquer parallelism scaled to at least 40 transputers.* The other two items are less significant, as we will hopefully be able to increase the granularity of tasks sufficiently to overcome the problem.

5.1 Hardware

The hardware used was a network of T800-25 transputer boards – developed mainly for ESPRIT 1219 (PADMAVATI) – which had an unusually large memory size (16 Mbyte DRAM). The DRAM memory access time is three wait-states. The large size enables efficient execution of functional and symbolic applications, because the larger the heap space available, the less frequently garbage collections are required.

5.2 Network

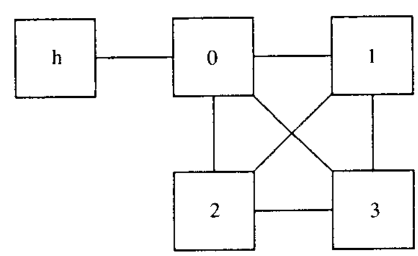

Figure 6. Network used for benchmarking.

The fully connected network of Fig. 6 was used. There were two reasons for this.

- It was readily realisable at that point in both our project and PADMAVAI.
- We were reluctant to write through-routing software for the T800. The PADMAVATI consortium was to provide fast global communication via a dynamic network (available since 1990 as Thomson's DYNET), and INMOS were rumoured to be working on their own solution (the HI).

5.3 Compilation

Functional programs were compiled to transputer machine code by first compiling them to parallel HDG-machine code,[11,14,23] and then macro-expanding this code to transputer machine code.

5.4 Load distribution

The load distribution mechanism was very simple –

* The `nfib` benchmark – discussed later – gives a clue as to the expense involved.

* The problem of shared data access does not figure in the ZAPP results.

when a processor had no work, it cyclically requested work from its neighbours until it was given a task. A neighbour would send a task if it had at least two tasks, and at least one of them had not been started yet. This idea was borrowed from ZAPP.[15] Even though it is a fairly primitive algorithm, the theoretical results from[16] seem to show that any algorithm is within a constant fraction of an optimal distribution strategy.*

5.5 Analysis of `nfib`

Table 1. Timings for `nfib`

	Time (in s) for processors			
Program	1	2	3	4
nfib 20	1.284	0.675	0.478	0.373
nfib 20*	0.996			
nfib 20†	0.813			0.832

* Parallel code with BSPAWN instructions deleted.
† Purely sequential code generated.

```
>nfib n=1,  n<2
>       =1+nfib (n-2)+nfib (n-1),
>             otherwise
```

The number of `nfib` calls per second when running on four processors is 58689. Relative to the parallel code running on a single processor the efficiency using four processors is 86 per cent and the speed-up is 3.4.

The second row in Table 1 shows what happens when the parallel code without BSPAWN instructions is executed. The version in the third row differs from that in the second, because it is sometimes able to avoid building graph.

For comparison, the purely sequential code is given in the third row. It runs in 0.813 s, giving 26926 function calls per second. Also included in this row is the same program – purely sequential – run with four processors. The slight slow down is because the root transputer must respond to messages requesting work.

In Table 2 we include the result for a more sophisticated compiler. The graph operations are the same as those used by the simple compiler; the change is that code is generated to make use of the transputer temporary registers to hold intermediate values.

This compiler was *not* implemented. It can be implemented as a peephole optimizer to our current version.

Table 2. Timings for `fnfib`

	Time (in s) for processors			
Program	1	2	3	4
fnfib 20	1.002	0.524	0.367	0.294
fnfib 20*	0.709			
fnfib 20†	0.575			

* Parallel code with BSPAWN instructions deleted.
† Purely sequential code generated.

* There could of course be problems with shared data structures which become distributed over the machine.

5.6 Analysis of `tak`

Table 3. Timings for `tak`

	Time (in s) for processors			
Program	1	2	3	4
tak 18 12 6	5.215	2.672	1.858	1.433

```
>tak x y z=z,  x<=y
>         =tak (tak (x-1) y z)
>              (tak (y-1) z x)
>              (tak (z-1) x y)),
>              otherwise
```

We have included the `tak` benchmark for comparison with LISP systems. The code generated for this problem is not ideal, as the current evaluation transformer analysis is not sophisticated enough to spot that the term `(z-1)` can be evaluated strictly. The `tak` benchmark partitions very well into tasks of roughly equal size. On four processors we are therefore obtaining an efficiency of 91 per cent, and a speed-up of 3.6.

5.7 Analysis of `queens`

```
>queens n          =queens' n n []
>queens' p 0       =1
>queens' p (n+1) b=sum [queens' p n (t:b)|
>                       t<-[1..p];
>                       safe t 1 b]
>safe q n []       =True
>safe q n (p:ps)   =q ~=p &
>                   q+n ~= p &
>                   q-n ~=p &
>                   safe q (n+1) ps
```

This benchmark calculates the number of ways to place n queens on to an $n \times n$ chessboard, such that no queen checked any other. In particular we have calculated values for $n = 4$ and $n = 6$.

Table 4. Timings for `queens`

	Time (in s) for processors			
Program	1	2	3	4
queens 4	0.012	0.011	0.009	0.009
queens 6	0.210	0.119	0.086	0.076

Table 5. Profile of function calls for `queens`

	Calls	
Function	queens 4	queens 6
queens	1	1
queens'	17	153
sum*	75	1043
generate†	75	1043
safe	100	2168

* This assumes that `sum` is defined recursively.
† The auxiliary function that is compiled for the list comprehension.

We can clearly see the effect of having a problem that is too small. The message overhead to export a task and retrieve its result is approximately 6 ms. In `queens 4` this means that each task that is exported is taking about 3 ms, and the maximum possible speed-up is achieved with three processors.* On a more positive note we see that as soon as the problem size is expanded to `queens 6` we achieve reasonable speed-ups. The speed-up on four processors is 2.8

This concludes our presentation of the results; we now discuss some of the issues that we feel deserve further investigation.

6. FURTHER WORK

In the course of the project we became aware of a number of features of the system that we would have liked to investigate, but had insufficient time to do so. We discuss some of them in this section.

6.1 Caching remote graph

In the current implementation it is possible for a piece of graph to have two different input indirection nodes pointing to it. When it is copied to a remove processor via one route, the remote processor does not know that it already has the value if it is accessed by the second route.

To correct this we must implement a hashing scheme to common-up remote references. In large-scale applications this will be vital.

6.2 Limitations of evaluation transformers

There are practical limits to the complexity of the analysis that can be performed by any abstract interpreter. In the scheme we used – with only four evaluators – we are unable to compile good code for the `concat` function.

```
>concat []     =[]
>concat (x:xs)=x++concat xs
```

If we need to evaluate each element in the list returned as the answer to `concat xs`, then will need to evaluate all of the elements in each list of `xs`. The problem here is that – working with a fourth-point domain – the abstract interpreter is unable to prove this fact for us. A more sophisticated abstract interpreter could solve this for us, but eventually there is a limit to what this smarter interpreter can do as well.

6.3 Constants on distributed machines

A problem with the distributed implementation of any language is the trade-off between copying data and re-creating it. This surfaces most clearly in functional languages when we consider *Constant Applicative Forms* (abbreviated CAFs). A simple example presenting an obvious choice is

```
>x=7
```

* It is conjectured that the rather slower time for two processors is related to the particular timing sequence of task requests to the root transputer.

In this case we should make a copy on each processor. A slightly larger CAF is

```
>y=[1..100]
```

This is the list of integers between 1 and 100. In this case it is probably worthwhile to have a copy on each processor.

```
>y=[0..]
```

This is the infinite list of integers, and we would probably wish to have a single copy which is exported when required.

6.4 Trees *vs.* lists

Programs using lists as their main data-structures generally do not parallelise well. The use of trees (when they are balanced) should result in a better partition of the data-structure over the distributed memory.

6.5 The I/O bottleneck

As implemented our machine uses a single transputer to access the file store, resulting in a bottleneck. If the hardware were adapted to support a number of file interface points in the network, multiple I/O operations could become significantly faster.

6.6 Large parallel applications

Although we have shown that the techniques we used work for small problems, we have still to demonstrate that the same techniques work for 'real' applications.

7. CONCLUSIONS

Within the constraints imposed on our project – given at the beginning of Section 5 – we have demonstrated that lazy functional programming can be efficiently implemented on a distributed architecture machine. This has been achieved without user annotations. This is a feature that we feel will be increasingly important as the size of parallel programs increases: at some stage it is likely to be infeasible to manually place annotations for parallelism.

The specification of our parallel abstract machine in a functional language turned out to be a very important tool in developing our implementation on the transputer network. Firstly, it enabled us to debug the specification of the parallel abstract machine. Secondly, each abstract machine instruction was implemented in the specification as a function from one state of the machine to another; complex instructions were specified and the composition of simpler subfunctions, each of which modelled a simple action in a real implementation. Therefore, the translation from our specification to a real implementation involved giving a sequence of transputer instructions for each function. On reflection, the hardest part of our implementation on the transputer network was the communication system, probably because it was specified at a much higher level than the rest of the abstract machine.

In this paper we have given most of the building blocks for constructing an efficient implementation of lazy functional languages on a transputer network. Although we adopted a simplistic way of generating code, macro-expanding the abstract machine instructions into sequences of transputer instructions, our experimental results have been better than expected, so much so that they compare favourably with much better engineered implementations. This encourages us to pursue the work further. Specifically, three things need further investigation:

- better code generation;
- structuring the system in a better way; and
- extending the system to a larger transputer network.

We briefly discuss each of these in turn.

Over recent years, compiling code for lazy functional languages has become a well-understood problem – see Refs 6, 26 and 27 for example. Better code generation will therefore involve combining these techniques with some of our insights concerning the transputer architecture.

One of the real difficulties in constructing the transputer implementation was the lack of good tools, and a suitable assembler. Not only was the assembler unreliable, but it forced us to structure the implementation in an unnatural way. This also had significant affects on the speed of the system, as 'global' values could only be accessed using quite complex procedures, and furthermore, required the storing of extra information on each node in the graph. Hopefully these inadequacies can be overcome as better tools become available.

We made a deliberate decision not to implement through-routeing, which would have allowed us to use a bigger transputer network. Now that the H1 chip has arrived, with hardware through-routeing, we should be able to try out our ideas on larger transputer networks, seeing how things scale.

Our experimental results have been very encouraging. We look forward to developing a better-engineered, more general system, and being able to analyse it.

Acknowledgements

Our colleagues at the GEC Hirst Research Centre, John Robson and Krste Asanovic, were very helpful in assisting us with our implementation on a transputer network. Their knowledge of the transputer architecture and help with various tools was most appreciated. David Bevan and Rajiv Karia also performed much valuable foundational work when they were working on this project. We would like to thank Simon Peyton Jones and the GRIP team for some very stimulating discussions on the parallel implementation of lazy functional languages.

Thanks are due to Sebastian Hunt for allowing us to use his implementation of abstract interpretation. We are also grateful to Bruce Cameron and Andrew Stitcher for permitting us to rewire and use PADMAVATI to run some further experiments after we have left GEC.

This work was performed whilst the authors worked for GEC Hirst Research Centre and was partially funded by ESPRIT Project 415, 'Parallel Architectures and Languages for AIP – A VLSI-Directed Approach'.

REFERENCES

1. A. W. Appel, Garbage collection can be faster than stack allocation. *Information Processing Letters* **25** (4), 275–279 (1987).
2. L. Augustsson, Compiling lazy functional languages, Part II. Doctoral thesis, Chalmers Tekniska Högskola, Göteborg, Sweden (1987).
3. L. Augustsson and T. Johnsson, The $\langle v, G\rangle$-machine: An abstract machine for parallel graph reduction. In *ACM Proceedings of the Functional Programming Languages and Computer Architecture Conference 11–13 September 1989*, edited D. B. MacQueen, pp. 202–213.
4. G. P. Balboni, P. G. Bosco, C. Cecchi, R. Melen, C. Moiso and G. Sofi, Implementation of a parallel logic + functional language. In *Parallel Computers: Object-Oriented, Functional Logic*, edited P. C. Treleaven, chapter 7, pp. 175–214. Wiley (1990).
5. D. I. Bevan, Distributed garbage collection using reference counting. In *PARLE Parallel Architectures and Languages Europe, Eindhoven, The Netherlands*, volume 2, edited J. W. de Bakker, A. J. Nijman and P. C. Treleaven, pp. 176–187. Lecture Notes in Computer Science [LNCS], Springer, Heidelberg (1987).
6. A. Bloss, P. Hudak, and J. Young, Code optimizations for lazy evaluation. *Lisp and Symbolic computation: An International Journal* **1** (2), 147–164 (1988).
7. A. Bloss, P. Hudak, and J. Young, An optimising compiler for a modern functional language. *The Computer Journal* **32** (2), 152–161 (1989).
8. G. L. Burn, Evaluation transformers – A model for the parallel evaluation of functional languages (extended abstract). In *Proceedings of the Functional Programming Languages and Computer Architecture Conference, September 1987*, edited G. Kahn, pp. 446–470. LNCS 274, Springer, Heidelberg.
9. G. L. Burn, *Abstract Interpretation and the Parallel Evaluation of Functional Languages*. Doctoral thesis, Imperial College, University of London (1987).
10. G. L. Burn, Developing a distributed memory architecture for parallel graph reduction. In *Proceedings of CONPAR 88, Manchester, UK*.
11. G. L. Burn, A shared memory parallel G-machine based on the evaluation transformer model of computation. In *Proceedings of the Workshop on the Implementation of Lazy Functional Languages, Aspenäs, Göteborg, Sweden, 5–8 September 1988*.
12. G. L. Burn, Overview of a parallel reduction machine project II. In *Proceedings of PARLE 89, Eindhoven, The Netherlands*, volume 1, edited E. Odijk, M. Rem and J.-C. Syre, pp. 385–396. LNCS 365, Springer, Heidelberg.
13. G. L. Burn, Using projection analysis in com[piling lazy functional programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pp. 227–241.
14. G. L. Burn, *Lazy Functional Languages: Abstract Interpretation and Compilation.* Research Monographs in Parallel and Distributed Computing, Pitman in association with MIT Press (1991).
15. F. W. Burton and M. R. Sleep, Executing functional programs on a virtual tree of processors. In *Proceedings of the First Conference on Functional Programming and Computer Architecture, Portsmouth, New Hampshire, October 1982*, pp. 187–194.
16. D. L. Eager, E. D. Lazowska, and J. Zahorjan, Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering* **SE-12** (5), 662–675 (1985).
17. J. Fairbairn and S. C. Wray, Code generation techniques for functional languages. In *Proceedings 1986 ACM Conference on Lisp and Functional Programming, Cambridge, Massachusetts, USA*, pp. 94–104.