Some implementation aspects of Concurrent Clean on distributed memory architectures.

John H. G. van Groningen

Faculty of Mathematics and Computer Science,
University of Nijmegen,
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
E-mail: johnvg@cs.kun.nl

October 1992

Abstract.

We have implemented a code generator and runtime system that can be used to simulate parallel execution of Concurrent Clean programs on a single Macintosh computer. This code generator generates machine code and is an extension of the (sequential) ABC code generator for the MC680x0 and SPARC processors.
This implementation is discussed briefly. The largest part of the paper describes two aspects of the implementation in detail. The first one is how to let a process wait until a node is overwritten with its head normal form. Several possible solutions are described. The other one is a description of an efficient graph copying algorithm. This graph copying algorithm can be used to send graphs to remote processors using a more compact representation. Finally some results are presented.

Introduction.

We have made an implementation of Concurrent Clean that can be used to simulate parallel execution on a single Macintosh computer. Concurrent Clean is an experimental lazy functional programming language (Brus (1987), Nöcker et al. (1991)). This simulation is not done using an interpreter, but by generating efficient machine code and simulating several processors on one machine.
To simulate Concurrent Clean programs, they are first compiled to (P)ABC code (Smetsers (1989), Koopman et al. (1990)). The resulting code is then translated by the code generator to machine code. This code generator is based on the (sequential) code generator for the MC680x0 and SPARC processors (Groningen (1990)). This implementation will be extended to be able to execute parallel programs on several Macintosh (or SPARC) computers connected to an ethernet network. Currently, it is already possible to do this, but only with two Macintosh computers. And it has not been tested yet on an ethernet network, but only on a (slow) local talk network.

Below we will give a brief description of the current implementation (the simulator, not the parallel implementation, which is the same except for the runtime system):

- The code generated is very similar to the code generated for sequential programs, except for what is discussed below. The node representation is the same (Groningen et al. (1991)).
- Stacks are checked for overflows, and if necessary enlarged by copying the stack to a larger memory block.
- Every process has two stacks. The PABC machine has 3 stacks, but the code generator has been changed, so that it can merge the B (basic) stack and the C (control) stack. This has been done to implement checking for stack overflows more efficiently. This merging of the stacks is now also used by the sequential implementation and the implementation on the transputer.
- Suspending and resuming processes is described below.
- Processes are scheduled round robin. This is implemented by decrementing a counter before every jump and subroutine call instruction and descheduling

the process if the counter was zero. For the jump instruction this is implemented very efficiently on the MC680x0 processor by using the DBRA instruction. In this way there is hardly any overhead when no descheduling is necessary. For the subroutine call the overhead is small. Currently descheduling occurs after 100 jump or subroutine call instructions.
- Processors are simulated using a timer. A simulated processor is descheduled when one if its processes is descheduled and the timer has expired. Currently this is done 500 times a second.
- All processors share the same heap. When the heap is full, all simulated processors are suspended while the garbage collection occurs. The processors are copied to the other semi-space one at a time by the copying collector (with copying a processor we mean copying the local nodes referenced by the processor). The (one space) mark sweep collector (Groningen et al. (1991)), which is sometimes used by the sequential implementation, cannot be used yet. While copying one processor, the garbage collector doesn't examine memory belonging to other processors. Instead global garbage collection (between processors) is done using one bit reference counting, which of course doesn't reclaim all unused memory.
- Simulated processors communicate using message passing. The messages are read by communication processes. Every simulated processor has such a communication process.
- Graphs are send using the copying algorithm described in this paper.

Waiting processes.

One of the problems that has to be solved to implement a parallel functional language is how to let a process wait until a node is in head normal form.

This waiting occurs in three situations:
1. A process wants to evaluate an indirection node to a node on a remote processor. On a distributed memory architecture communication between the processors is then necessary to fetch the result. In such a case the process should be suspend after sending a request. The process should be resumed when the node has been overwritten with the head normal form. This head normal form is send by the remote processor as answer to this request.
2. If a request arrives from a process on a remote processor for a node that is not yet in head normal form. Then the process that is responsible for answering the request has to wait until the node is overwritten with its head normal form before the result can be send.
3. A process wants to evaluate a node that is already being evaluated by another local process. In such a situation the process should be suspended, and resumed when the other process has overwritten the node with its head normal form.

To be able to do this, there should be a way to detect if a graph is being evaluated. Most implementations do this by marking the node in some way when evaluation of this node is started. For example by using a special evaluation address (in the PABC machine, Nöcker et al. (1991), Kesseler (1991)) or using a bit in the tag (<ν,G> machine, Augustsson and Johnsson (1989)).
This marking is called reserving, or locking. A node which has been reserved is called a reserved node. Reserving is not only necessary (for some nodes) for parallel evaluation, but is also very useful for sequential evaluation. It is useful because it can be used to prevent some space leaks and to detect a cycle in the spine during evaluation. Although this reserving slows down program execution, the benefit of having fewer space leaks is usually more important than the cost of executing one or a few more instructions for every closure. So we do not consider this reserving as a problem.

Resuming a process after a node has been overwritten by its head normal form is more difficult. Several possible solutions are discussed below.

Waiting lists.

One way to implement suspending and resuming processes is to add a list of waiting processes to every reserved node. When a process needs to evaluate a reserved node, the process is removed from the scheduling list, and added to the waiting list of the reserved node. When this node is overwritten (by a node in head normal form), the processes in the waiting list are resumed.

The disadvantages of this method are:

1. Before overwriting a node with its head normal form (an update), it is necessary first to the check if the node contains a waiting list. And if so, all processes in the waiting list have to be resumed. Because the number of updates performed during execution of a program is usually large, the cost in execution time is high. And because more instructions are generated for an update, programs get larger too.

2. Reserved nodes should contain a waiting list or some indication whether or not the node contains a waiting list.
   Storing a waiting list in all reserved nodes would mean that every time the evaluation of a closure is started, an empty waiting list should be stored in the node. Because the number of closures which are evaluated during execution is large, this cost is high.
   Another possibility is to store a waiting list in every closure. Then no empty waiting list has to be stored when a node is evaluated, but closures are larger and more expensive to build, so this is even worse. This method is used at the moment by the transputer implementation of the PABC machine (Kesseler 1991).
   A better way is to have some indication whether the node contains a waiting list, and only store the waiting list if there is one. If we combine this information with reserving a node, then there is no additional overhead when the evaluation of a closure is started. This approach is used in the <ν,G> machine (Augustsson and Johnsson (1989)).
   In the PABC machine a node is reserved by overwriting the evaluation address in the closure by the address of code which suspends a process and inserts it in the waiting list. Because there are only a few of those addresses, we can easily make two versions of those addresses. One that is used when there is no waiting list and the other when there is a waiting list. For example on the transputer we could store one version at an even address and the other one at an odd address. Then the least significant bit indicates that there is a waiting list. Because these nodes only contain this evaluation address, there is free space in these nodes to store a waiting list.

3. The garbage collector should be able to copy waiting lists.

Optimising waiting lists.

The most important of these disadvantages is that when a node is updated we have to test if there is a waiting list, and resume all processes in the waiting list if so. For some implementations it is however possible to prevent this overhead when the waiting list is empty. How this can be done depends on how closures are represented and how updates are performed.

For example, in the implementation of Concurrent Clean on ZAPP (Goldsmith, McBurney and Sleep (1991)) a closure is represented by a pointer into a symbol table and the arguments of the closure. The symbol table contains an entry to code which is executed when an update has to be performed. In this implementation testing for waiting lists is prevented by making two symbol table entries for every function. One that contains update code which is used when there is no waiting list. And one that contains update code which resumes the processes in the waiting list. Usually closures contain a pointer to the first table entry, but when a process should be moved to the waiting list, the pointer in the closure is changed to the second table entry.
A disadvantage of this method is that an extra indirection is necessary to evaluate a closure. This extra indirection is required to fetch the evaluation address from the symbol table. Some other implementations (see below) don't have such an indirection.

For the PABC machine we cannot use this solution, because after evaluating a node execution continues at the return address on top of a stack, and not at an address which is found in a symbol table (in the PABC machine a closure contains a pointer to the code to be executed when the closure has to be evaluated, instead of a pointer into the symbol table). For implementations like this we can use one the methods described below.

These methods prevent unnecessary testing for waiting lists by changing the return address when a waiting list is added to a node. This is done when a process starts to wait for a node which does not yet contain a waiting list. This return address should point to code which reactivates the processes in the waiting list and then jumps to the original return address.
Then the process will be resumed after the node has been updated. And there is no overhead in the common case that a node doesn't contain a waiting list, because a normal return address doesn't have to check for one.

One way to do this is to store a pointer to the stack element containing the return address in a reserved node. We do this when we start to evaluate the node. This would mean an extra instruction has to be executed for every evaluation. But this is still faster than testing for a waiting list. Also the address of the stack element can be changed when the stack is reallocated because it has become too small (or by the garbage collector in some implementations). Then the address of the stack entry has to be relocated too. Doing this is quite complicated, so this doesn't seem to be a good solution. But this is not a problem when linked or fixed size stacks are used.

A more elegant solution was proposed by Marco Kesseler. This solution is not to store the return address on a stack, but in the node which is going to be evaluated. To do this the return address is passed as an argument to the evaluation code. The evaluation code stores this return address in the node after copying the argument(s) of the node on the stack. And before updating the node with the head normal form, the return address is loaded from the node and then jumped to.

Unfortunately some frequently used processors, like the Motorola MC680x0 and the Intel 80x86, cannot store(load) the return address in(from) a node as efficiently as in(from) the stack. For example the MC680x0 needs 5 instructions instead of 2 (a JSR and a RTS) to accomplish this. So for these processors this optimisation is probably not worthwhile. But for the transputer and most RISC processors this could work well. Another advantage of this method is that less stack space is used, because the return address is now stored in the node.

For both approaches there is still another problem: where to store the original return address and the process identification when a process has to be suspended. A possible solution is to store this information in a new node or in the process table entry (or process node). The machine code to be executed after the node has been updated should then be stored in that place as well. Then this code can find the original return address and the process identification. The return address on the stack should then point to this code after suspending the process. Unfortunately, this could cause some small problems for the garbage collector.

Yet another problem for both solutions is discussed in the section 'Creating fewer interleaved reducers'.

Using sharing information.

Only nodes which are being shared by more than one process (and are not yet in head normal form), can contain a waiting list. So to optimise the check for waiting lists, we could try do this check only when a node might be shared by several processes.

This is probably similar to trying to prevent updates when nodes cannot be shared. This has been suggested for the Spineless Tagless G-machine (Peyton Jones and Salkild (1989)) and TIM (Fairbairn and Wray (1987)). They propose to use some form of static analysis to detect sharing, and then to use this information to generate update frames (STG-machine) or markers (TIM) only when necessary. Unfortunately, to use this approach many functions have to do argument satisfaction checks or check for markers. This is probably not much faster than testing for waiting lists. Also, we don't know how effective and expensive this sharing analysis would be.

Polling processes.

Another way to implement suspending and resuming of processes, is to let a 'suspended' process poll every time it is scheduled. So, if a process wants to evaluate a node which is being reduced by another process, it deschedules itself and doesn't remove itself from the scheduling list, but enters a polling state. When a process in a polling state is scheduled, it checks if the node has been updated with its head normal form. If this is not the case, it deschedules itself and doesn't change its state so that it will do the same actions the next time it is scheduled. But if the node has been updated, the process leaves the polling state, and resumes normal execution.

The advantages of this method are that it is very easy to implement and that there is no overhead for sequential execution and if no processes are waiting. A disadvantage is that is very inefficient if many processes are waiting.

Optimising polling processes.

It is however possible to prevent polling in many cases. We already explained that there are three situations in which a process starts to wait (and thus poll).

In the first case (an indirection node to a node on a remote processor has to be evaluated), the process can only be resumed if a message arrives with the head normal form of the node. So it is not necessary to put the process in a polling state, but the process can be suspended (removed from the scheduling list), and resumed when the message with the head normal form arrives.
To implement this, we added a list of waiting processors to channel nodes. Channel nodes are indirection nodes which point to a remote processor. The code which is called when a channel node is evaluated, sends a request (but only if no request has been send yet for this channel node), adds the process to the waiting list and removes it from the scheduling list. When the message with the head normal form arrives, the processes in the list are resumed.

In the second case (a request arrives from a process on a remote processor for a node which is not yet in head normal form), a naive implementation would start a waiting process for every request which cannot be answered immediately. A more efficient

way would be to create a list of all nodes for which a request has arrived (and not yet answered). And let a communication process check whether these nodes have been updated to head normal form every time the process is scheduled. In this way the scheduling overhead is smaller. And only one check is necessary per node, even if more requests arrive for the same node.
To do this, we added a list of requests to nodes for which a request can arrive (global nodes). If a request arrives for a node which is not yet in head normal form, the request is added to the list of requests for this node. And the node is added to the list of global nodes which are checked by the communication process (except if it is already in this list) Then, if the communication process detects a node for which an answer can be send, it sends an answer to all processes for which a request is in the request list, and removes this node from the list.
Note that when no message can be send, because the communication process hasn't finished sending the previous message, no polling is necessary.

The cost of optimised polling and waiting lists.

The current implementation uses polling with the optimisations described in the previous section. We have run several programs to determine the overhead of this optimised polling. The overhead was determined by counting the number of polls. Two counters were used. One counter counted the number of polls by normal processes. The other counter counted the number of polls by the communication process. The counters were not incremented when all processes were polling.

The cost of every poll was estimated. Polling by normal processes was estimated to cost 90 clock cycles for every poll. The time it takes to deschedule the process is included in this number. Polling by the communication process was estimated to cost 32 clock cycles for every poll.

The programs where run a Macintosh IIfx. The results of these measurements are: (all times in seconds)

	total simulation time:	number of process polls:	number of communication process polls:	total overhead of polling:	overhead of polling (%):
quicksort	26.68	140	26100	0.021	0.079
queens	25.67	0	106500	0.085	0.33
nfib	4.73	0	946	0.0008	0.016
fast fourier	17.27	0	270	0.0002	0.001
warshall 1	3.47	5500	5400	0.017	0.48
warshall 2	6.17	6400	4900	0.018	0.30
sieve 1	43.80	0	214000	0.17	0.39
sieve 2	40.92	20	3000	0.0025	0.006
sieve 3	38.32	143000	6800	0.33	0.85

So for all these programs the overhead for polling is less than 1 percent. Most of these programs use course grain parallelism, except the sieve programs. Sieve 1 and 2 use fine grain parallelism.

The same programs were run to determine the overhead of testing for waiting lists. We have assumed waiting lists are indicated by setting a bit in the tag. We have estimated the cost of testing this tag bit to be 14 clock cycles.

	total simulation time:	number of tests for waiting lists:	total overhead of waiting lists:	overhead of waiting lists (%):	average idle time (%):
quicksort	26.68	280000	0.098	0.37	75
queens	25.67	1380000	0.48	1.88	22
nfib	4.73	233	0.00001	0.0002	11
fast fourier	17.27	69600	0.024	0.14	51
warshall 1	3.47	42100	0.015	0.43	56
warshall 2	6.17	40700	0.014	0.23	46
sieve 1	43.80	235000	0.082	0.19	49
sieve 2	40.92	51200	0.018	0.04	79
sieve 3	38.32	837000	0.29	0.76	61

So for these programs the overhead for testing for waiting lists is less than 2 percent. This is lower than we expected. The reasons for this low overhead are that the processors spend too much time doing nothing (see idle times in table) and copying graphs. Also our estimate of 14 clock cycles is only accurate if the instructions which test for waiting lists are in the instruction cache of the processor.

We can also see that polling is much faster for queens and quicksort, and a little bit slower for warshall and sieve. Polling is only slower for nfib and fast fourier, but for these programs the overhead is very small in both cases. So polling with a few simple optimisations is more efficient than waiting lists for our test programs.

A combination of polling and waiting lists.

A further improvement is possible because the node a process is waiting for is very often the root node of another process. In this situation we can use waiting lists efficiently, because we only have to check such a root node for a waiting list when the process finishes reducing. Then in this situation, we no longer have to poll for these nodes, and there is no cost for sequential execution and only a small cost for course grain parallelism.

If all the optimisations are implemented, then polling is used only for nodes for which another local process is waiting or for which a request has arrived and which are not the root node of a process. All other process/requests waiting for a node are resumed/answered when a process terminates or an answer to a request arrives.

Creating fewer interleaved reducers.

In Concurrent Clean {I} annotations are used in programs to start a new process on the same processor. Such a process is called an interleaved reducer. An interleaved reducer reduces the graph to head normal form.
In the Concurrent Clean programs we have written, we very often use functions which evaluate the head of a list using a strictness annotation and the tail of this list using an {I} annotation. This tail is very often a recursive call of the same function. For every call of such a function a new interleaved reducer is created, and the old one dies, except possibly for the first call. Creating interleaved reducers is expensive, so this inefficient.
To eliminate this inefficiency, we want to change the compiler so that it is possible to let the old process continue with the work which would otherwise have been done by the process started with the {I} annotation. Then no new process has to be started, and the reduction goes much faster.
So the code for such a function would first compute the head of the list, then update the root node, and then continue with the reduction of the tail of the list.

If we use waiting lists, we also have to check for a waiting list every time the function is called. And the optimisations to prevent overhead for waiting lists described in the section 'optimising waiting lists' cannot be used in this situation.
Because many Concurrent Clean programs contain similar functions, this is a serious disadvantage of using waiting lists. Implementations using polling do not have this problem. So for these implementations efficient code can be generated for such functions.

Waiting lists or polling for Concurrent Clean ?

The current implementation of polling uses less than 1 percent (on average 0.27%) of the execution time for the programs we tested and is easy to implement. No extra code has to be generated by the compiler and no extra information has to be stored in the nodes.
The naive form of waiting lists is slower for these test programs, the overhead is less than 2 percent and on average 0.45 percent.
An optimised form of waiting lists could be faster than optimised polling. But these solutions are difficult to implement and generate more code. Some of the overhead eliminated by the optimisations would be introduced again if we change the compiler to compile some functions as described in the previous section.
At the moment the overhead caused by checking for stack overflows and descheduling is much higher (see below) than the overhead caused by polling. So at the moment the efficiency of suspending and resuming processes is more than sufficient.

Graph copying.

When a parallel process is created on a distributed memory architecture usually a copy of a graph has to be send to another processor. Many implementations do this by first creating a local copy which is represented in the same way as a normal graph. And then this local copy is send to the remote processor. This approach is used by Concurrent Clean on ZAPP (Goldsmith, McBurney and Sleep (1991)) and by the PABC machine implementation on the transputer (Kesseler (1991)).
In this way, the processor which receives this graph has to do very little work to start the new process. Unfortunately, this representation is not the most compact representation of the graph.

For example a copy of the graph CONS (INT 1) (CONS (INT 2) NIL) would be represented by:

0: CONS	1: 3	2: 5	3: INT	4: 1	5: CONS	6: 8	7: 10	8: INT	9: 2	10: NIL

If all nodes in the graph are referenced only once, we can leave out all the pointers (3,5,8 and 10 in the example), and are still able to reconstruct the graph. Then the copy becomes:

0: CONS	1: INT	2: 1	3: CONS	4: INT	5: 2	6: NIL

Now the size of the copy is only 7 words instead of 11 words.

And (CONS (INT 2) NIL, NIL, CONS (INT 4) NIL) becomes: (breadth first copy)

0:	1:	2:	3:	4:	5:	6:	7:	8:	9:
3 TUPLE	CONS	NIL	CONS	INT	2	NIL	INT	4	NIL

If a node is referenced more than once, we can still use pointers if we choose the representation of pointers and constructors in such a way that we can distinguish them. In our implementation this is done by setting the least significant bit of a pointer. (pointers and symbols are always on an even address) For the first reference to such node, we do not use a pointer. For example the copy of the graph CONS (a: INT 1) (CONS a (CONS a NIL)) is:

0:	1:	2:	3:	4:	5:	6:	7:
CONS	INT	1	CONS	1	CONS	1	NIL

If we assume nodes consist of just a symbol and arguments, the size of the copy is the number of unshared nodes + the number of pointers (arguments) to shared nodes.

If there is no sharing, every symbol in such a copy is followed by the root nodes of the arguments, so the copy is made in a breadth first manner. It can also be done in a depth first manner, then the symbol is followed by the whole subgraph of the first argument (if it has an argument), then the subgraph of the second arguments, etc. But this turned out to be more difficult.

There are algorithms to create such a packed copy and unpack it which are about as fast as creating a normal copy. These algorithms are discussed below. The advantages of using this way of copying are:

- Fewer information has to be send across the communication network. So messages are smaller. Or when large graphs are split into smaller messages, fewer messages have to be send. So communication will be faster. And if through-routing has to be performed by the processor, sending fewer messages also means that less processor time is wasted by routing.
- Less space is necessary to store a local copy on a sending processor. This is important if messages are buffered. This happens when messages are produced faster than can be send, or when the communication channel is unreliably and messages cannot be removed until an acknowledgement has been received.

Disadvantages are that more memory is required to unpack a copy, and that unpacking is a little bit slower.

The copy algorithm for the sending processor.

To describe the algorithms we will assume all nodes consist of just a symbol and arguments.

Creating a packed copy as described above can be done in a similar way as the scavenging algorithm used by a copying garbage collector. But instead of copying the whole node to the start of free memory (to-space), the symbol is copied to the beginning of memory, and the arguments (the pointers) of this node to the end. Except when a node has already been copied, then the address of the copy of this node is copied to the beginning of memory. But in such a way that it can be recognised as a pointer (and not a symbol).

The most important variables used by the algorithm are:

`symbol_p`:	pointer to symbol of node which is to be scavenged (of which the arguments should be copied).

`end_symbol_p`:	pointer to the end of the memory block containing symbols and pointers to shared nodes, where new symbols and pointers to shared nodes are created.
`argument_p`:	pointer to an argument of the node which is being scavanged.
`end_argument_p`:	pointer to the end of the memory block containing the arguments, where new arguments are copied to.
`free`:	the number of free words in the memory block.

Before the starting this scavenging like algorithm, first the root node has to be copied (evacuated). To prevent having to write evacuation code, this is done by branching directly into the loop after setting up the variables appropriately.

```
argument=address_of_root_node_of_the_graph_to_be_copied
arity = n = 1
symbol_p = end_symbol_p = begin_of_free_memory
argument_p = end_argument_p = end_of_free_memory
free = size_of_free_memory
GOTO begin_copy

WHILE symbol_p < end_symbol_p
      IF is_marked_pointer (*symbol_p)
            ++symbol_p
      ELSE
            arity = node_arity (*symbol_p++)
            FOR n=1 TO arity
                  argument = *--argument_p
begin_copy:       IF is_marked_pointer (*argument)
                        free -= 1
                        IF free < 0
                              GOTO garbage_collect
                        *end_symbol_p++ = *argument
                  ELSE
                        argument_arity=node_arity (*argument)
                        free -= 1 + argument_arity
                        IF free < 0
                              GOTO garbage_collect
                        *end_symbol_p = *argument
                        *argument++ = mark_pointer (end_symbol_p)
                        ++end_symbol_p
                        FOR m=1 TO argument_arity
                              *--end_argument_p = *argument++
```

Now the packed copy is stored in the memory area from the `begin_of_free_memory` to the `symbol_p`. During this copying the nodes of the original graph are overwritten by indirections to the copy of the node. The following code is used to restore the original graph:

```
restore_original_graph:
WHILE end_argument_p < end_of_free_memory
      node_p = *end_argument_p++
      IF is_marked_pointer (*node_p)
            *node_p = *unmark_pointer (*node_p)
```

If there is not enough memory available to copy the graph, the original graph is restored in the same way, the memory used during the copying is released, then the garbage collector is called and a new attempt is made to copy the graph.

The copy algorithm for the receiving processor.

Unpacking also uses an algorithm similar to scavenging. Now copying the nodes in done in the usual way, i.e. both the symbol and the arguments are copied to the beginning of the memory area. But reading the arguments to be copied is simpler, because the nodes are already stored in the right order. So the original is traversed from the beginning to the end.
When a symbol is copied, space is reserved for the arguments immediately. These spaces are filled in when the root node of the argument is copied.

The most important variables used by the algorithm are:

`node_p`:	pointer to the node which is to be scavenged (of which the arguments should be copied).
`end_node_p`:	pointer to the end of the memory block containing the already copied nodes, where new nodes are created.
`graph_p`:	pointer to the next argument which is to be copied.
`free`:	the number of free words in the memory block.

The same trick is used again to copy the root node:

```
graph_p = address_of_the_graph_to_be_copied
node_p = end_node_p = begin_of_free_memory
free = size_of_free_memory
arity = n = 1
GOTO begin_copy

WHILE node_p < end_node_p
      arity = node_arity (*node_p++)
      FOR n=1 TO arity
begin_copy: IF is_marked_pointer (*graph_p)
                  *node_p++ = *relocate_marked_pointer (*graph_p++)
            ELSE
                  argument_arity = node_arity (*graph_p)
                  free -= 1+ argument_size
                  IF free < 0
                        GOTO garbage_collect
                  *end_node_p = *graph_p
                  *node_p++ = end_node_p
                  *graph_p++ = end_node_p
                  end_node_p += 1 + argument_arity
```

Now the memory area from the `begin_of_free_memory` to `node_p` contains a normal copy of the graph. The original has been changed, but is not used anymore, so it doesn't have to be restored.

Except when there is not enough memory to copy the graph, then the original is restored using:

```
restore_original_graph:
p=begin_of_graph
WHILE p < graph_p
      IF is_not_a_marked_pointer (*p)
            *p=**p
      ++p
```

and then the memory used during the copy is released, the garbage collector is called and a new attempt is made to copy the graph.
Note that to be able to restore the graph, it should be possible to detect the difference between a marked pointer and the pointer stored by the line '`*graph_p++ =`

`end_node_p`'. In our implementation this is possible because for a marked pointer the least significant bit is set, and for a normal pointer it is not set.

Results.

To measure the overhead of checking for stack overflows and scheduling some simple programs were run on a Macintosh IIcx (16Mhz MC68030 without cache):

program:	time (s):	time including overhead (s):	overhead (%):
fast fourier	14.88	15.25	2.5
sieve	11.50	12.63	9.8
queens	42.45	49.50	16.6
reverse	52.25	52.26	0.1
twice	1.53	1.73	13.1
tak	6.60	7.58	14.8
integer nfib	6.11	7.86	28.6
real nfib	6.13	6.35	3.6

So the overhead is usually between 0 and 30 percent of the execution time without checking for stack overflows and scheduling.

The programs below were run on a Macintosh IIfx (40 Mhz MC68030 with 32K external cache) with 8 M memory using system 7.0.1. (queens is not the same programs as above, the compilers used are different too) Sixteen processors were simulated. All execution times are in seconds.

program:	time (sequential code):	total simulation time:	simulation time / number of processors:	speedup: (16 processors)
fast fourier	4.88	17.3	1.08	4.5
queens	13.28	25.7	1.60	8.3
matrix	2.18	9.93	0.62	3.5
quicksort	1.31	26.7	1.67	0.78
nfib	2.48	4.73	0.30	8.4

Annotations are only used to create coarse grain parallelism. The fast fourier program also contains strictness annotations to speed up execution.
Fast fourier performs the fast fourier transformation of 8192 complex numbers.
Queens computed the number of solutions on a 10x10 board.
Matrix multiplies two 64x64 real matrices.
Quicksort sorts 10000 integers.
Nfib computes nfib 30.

References.

Brus T., Eekelen M.C.J.D. van, Leer M. van, Plasmeijer M.J. (1987), 'Clean - a Language for Functional Graph Rewriting', Proc. of the third International Conference on Functional Programming Languages and Computer Architecture (FPCA '87), Portland, Oregon, USA, Springer Lecture Notes on Computer Science 274, pp. 346-384.

Augustsson L., Johnsson T. (1989). 'The <ν,G>-machine: an abstract machine for parallel graph reduction', Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '89), Imperial College, London (11-13 September), pp 202-213.

Fairbairn J., Wray S (1987). 'TIM: A simple, lazy abstract machine to execute supercombinators', Proceedings of the Functional Programming Languages and Computer Architecture Conference (FPCA '87), pp 34-45, LNCS, 274, Springer, Heidelberg.

Goldsmith R., McBurney D.L. and Sleep M.R. (1991). 'Concurrent Clean on ZAPP', Proceedings of the Semagraph '91 Symposium on the Semantics and Pragmatics of Generalized Graph Rewriting, Nijmegen, the Netherlands, (10-12 December), to appear 1992.

Groningen J.H.G. van. (1990), 'Implementing the ABC-machine on M680x0 based architectures'. Master Thesis, University of Nijmegen, November 1990.

Groningen J.H.G. van, Nöcker E.G.J.M.H., Smetsers J.E.W. (1991). 'Efficient heap management in the concrete ABC machine', Proceedings of the Third International Workshop on the Parallel Implementation of Functional Languages, Southampton, June 1991.

Kesseler M.H.G. (1990). 'Concurrent Clean on Transputers', Master Thesis, University of Nijmegen, November 1990.

Kesseler M.H.G. (1991). 'Implementing the PABC machine on transputers', Proceedings of the Third International Workshop on the Parallel Implementation of Functional Languages', Southampton, June 1991.

Koopman P.W.M., Eekelen M.C.J.D. van, Nöcker E.G.J.M.H., Smetsers J.E.W., Plasmeijer M.J. (1990). 'The ABC-machine: A Sequential Stack-based Abstract Machine For Graph Rewriting'. Technical Report no. 90-22, December 1990, University of Nijmegen.

Nöcker E.G.J.M.H., Smetsers J.E.W., Eekelen, M.C.J.D. van, Plasmeijer (1991). 'Concurrent Clean', Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE '91), Eindhoven, The Netherlands, Lecture Notes on Computer Science, Springer Verlag, June 1991.

Peyton Jones S.L, Salkild J. (1989). 'The Spineless Tagless G-machine'. Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '89), Addison Wesley, pp 184 - 201.

Smetsers J.E.W., (1989). 'Compiling Clean to Abstract ABC-Machine Code', University of Nijmegen, Technical Report 89-20, October 1989.