Parallel Functional Implementations: Graphbased vs. Stackbased Reduction

Guido Hogen and Rita Loogen***

Abstract

The *meshed stack* is a technique for the management of control structures in distributed implementations of dynamic process systems. In the environment of a parallel abstract machine for the execution of functional programs, we compare this approach to the classical graphbased reduction technique. Experimental results will show, that the stackbased reduction is more efficient in space and time, especially in relation to garbage collection. Also, the stack technique is a natural extension of sequential reduction machines.

1 Introduction

In parallel implementations of functional languages, graph reduction is a well-known technique. Whereas purely sequential implementations use a *runtime stack* for the control of recursive function calls, parallel implementations allocate runtime structures in a graph or heap. Because of the dynamic process behaviour, the use of a single global stack is no longer advisable. Since each parallel task or process needs its own stack structure, a global central stack would become a bottleneck. A *graph organization* has the advantage of *decentralization* which is especially suited for distributed systems.
Furthermore, a suspension mechanism must be implemented in order to save the state of tasks waiting for the result of some parallel subtask. In a graph, context switches can simply be achieved by redirecting the pointer to the currently active process. The runtime stack of each process is implemented as a linked list of activation blocks in the heap. Spawning of parallel processes yields a tree structure of such lists, representing a so-called *cactus stack*.
In [Hogen, Loogen 94] we have presented an alternative management of the runtime structures which tends to support parallel and sequential evaluations equally well. Instead of representing the cactus stack as a tree of activation blocks, we allocate the activation blocks within each processor element of the parallel system on a single *meshed stack*, which is organized as a contiguous array, but shared by several processes. The heap is used only for the representation of migrated parallel processes, structured data, closures and suspensions. Evaluation is controlled by the stack.
In this paper we present a detailed comparison of both techniques. We do this by incorporating a graphbased and a stackbased approach in the *same* parallel environment. We

*RWTH Aachen, Lehrstuhl für Informatik II, Ahornstraße 55, D-52056 Aachen, Germany, email: ghogen@zeus.informatik.rwth-aachen.de

**Philipps-Universität Marburg, Fachgebiet Informatik, Hans Meerwein Straße, Lahnberge, D-35032 Marburg, Germany, email: loogen@informatik.uni-marburg.de

will describe the parallel environment as well as the two reduction techniques (namely the PAM [Loogen et al. 89] and the PASTEL-machine [Hogen, Loogen 94]) in detail. Since both machines run in the same environment, we are able to make detailed comparisons and runtime measurements.
Our experimental results will show that the stackbased approach has its main advantages in the reduced organizational overhead for the *deallocation* (garbage collection) of runtime information. It also has some more nice properties: First, purely sequential evaluations within processes are handled as in sequential stack machines, second, the stack technique is a quite natural extension of a sequential stack machine by adding the ability of multitasking. We still have the distinction between control and data.
This paper is organized as follows: In the next section, we first give an overview on the project and the general structure of the parallel abstract machine. Then we describe in detail the graphbased and the stackbased reduction, especially the meshed stack technique. In Section 3 we focus on the comparison of these approaches; the differences are mainly due to process creation and process termination (garbage collection). In Section 4 we show experimental results for the *quicksort*-function on both machines in conjunction with two different heap garbage collection strategies. We are going to show, that the stack technique is more efficient in runtime and memory usage. A discussion of related work and conclusions in Section 5 completes the paper.

2 Parallel Implementation of Functional Languages

We have developed a first order monomorphically typed functional language with algebraic data structures as the basis of our implementation. Programs are expected to be annotated with parallelism information. These annotations can be done automatically by program transformation, see e.g. [Hogen et al. 92]. The functional programs are translated into abstract machine code, which is interpreted on the abstract machine. Instead of going into detail on the functional language and its transformation functions, we will only give a sample program (see Figure 1). For more details on this topic see [Hogen, Loogen 94].

$$
\begin{array}{llll}
\mathit{append}(l_1, l_2) := & \mathbf{case}\ l_1\ \mathbf{of} & \mathit{nil}: & l_2; \\
 & & \mathit{cons}: & \mathit{cons}(\mathbf{proj}(1, l_1), \mathit{append}(\mathbf{proj}(2, l_1), l_2)) \\
 & \mathbf{end} & & \\
\mathit{filter}(f, l) := & \mathbf{case}\ l\ \mathbf{of} & \mathit{nil}: & \mathit{nil}; \\
 & & \mathit{cons}: & \mathbf{let}\ h = \mathbf{proj}(1, l) \\
 & & & \mathbf{in\ if}\ (f\ h)\ \mathbf{then}\ \mathit{cons}(h, \mathit{filter}(f, \mathbf{proj}(2, l))) \\
 & & & \qquad\qquad \mathbf{else}\ \mathit{filter}(f, \mathbf{proj}(2, l)) \\
 & \mathbf{end} & & \\
\mathit{qsort}(l) := & \mathbf{case}\ l\ \mathbf{of} & \mathit{nil}: & \mathit{nil}; \\
 & & \mathit{cons}: & \mathbf{let}\ h = \mathbf{proj}(1, l); \\
 & & & \qquad t = \mathbf{proj}(2, l) \\
 & & & \mathbf{in\ letpar}\ y = \mathit{qsort}(\mathit{filter}((\geq\ h), t)) \\
 & & & \quad \mathbf{in}\ \mathit{append}(\mathit{qsort}(\mathit{filter}((<\ h), t)), \mathit{cons}(h, y)); \\
 & \mathbf{end} & &
\end{array}
$$

Figure 1: The *quicksort*-program

The main feature of the language w.r.t. parallelism is the **letpar**-construct. A **letpar**-

expression **letpar** $...y_i = f_i(e_{i1}, \ldots, e_{in_i})...$ **in** e means that the function applications $f_i(e_{i1}, \ldots, e_{in_i})$ *may be* evaluated in parallel with the main stream of evaluation, e. Whether the parallel evaluation is executed in parallel, depends on the workload of the distributed system. Evaluation of a **letpar**-expression first leads to the parallel activation of the function applications. Then the evaluation of the body e proceeds until the result of a parallel subexpression, represented by y_j, is needed but not available yet. In this case, the evaluation of e will be suspended until the parallel subprocess sends its result. In the meantime, the processor may evaluate some other processes.

The resulting process system is hierarchical. Starting with the evaluation of a main expression, **letpar**-expressions may generate parallel subprocesses, which can be executed on other processor elements. Execution of parallel processes may generate further processes. As soon as a process terminates, its result is communicated to the parent process, i.e. the process that generated the subprocess, and all other processes waiting for this result.

2.1 General Structure of the Parallel Abstract Machine

The parallel reduction machine consists of a finite number of identical processor elements, each with a local storage. An interconnection network enables all processor elements to communicate by exchanging messages. Each processor element consists of two independent processing units, which communicate via message exchanges in a local shared memory (see Figure 2, the dashed stack component will be added later).

The *communication unit* handles the organization of the parallel aspects of the reduction process. The *reduction unit* executes the functional program. Basically, it is a sequential reduction machine extended by multitasking and message handling.

The main task of the communication unit is the distribution of parallel processes through the net and the organization of message traffic (e.g. routing). We will not go into the details of the communication processor. We assume a simple static routing scheme and a passive process distribution strategy, i.e. processor elements without work have to ask their neighbours.

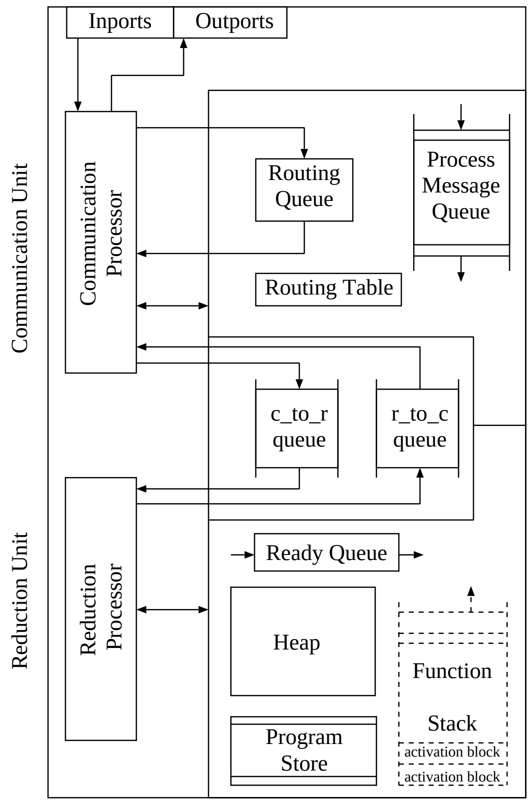

Figure 2 Structure of a Single Processor Unit

The main message types are *request messages* to ask for data stored on another processor element, *answer messages* to communicate the result of a parallel process or requested data, and *process messages* to distribute parallel processes. Since we assume a passive distribution scheme, each communication processor maintains a process queue which contains parallel processes which have been generated by the reduction unit, but not been invoked yet.

The structure of the reduction component depends on the chosen reduction *technique*. As for the reduction *strategy* we will restrict our presentation to an eager strategy; the machine can be easily extended to handle lazy evaluation. Since we want to compare two reduction techniques, we will first describe the graphbased approach and then the changes made for a stackbased implementation, before we will compare both techniques in a separate section.

2.2 Graphbased Reduction

The first technique presented here is program execution by *distributed code-directed graph reduction* [Goldberg, Hudak 86], [Loogen et al. 89], [Augustsson, Johnsson 89], [Kingdon et al. 91]. Each reduction unit has a copy of the abstract machine code in its program store. As one can already see in Figure 2 the main runtime structure is the *heap* or *graph* component. We distinguish between different heap entries:

- *Task nodes:* A task node represents a function call. It contains all status information to evaluate this call, i.e. a local instruction pointer, the argument list, a local data stack, and the address of the invoking task node.
- *Terminal nodes:* We distinguish further between data, constructor and functional nodes (partial applications).
- *Placeholder nodes*: These reserve space for the result of a parallel process executed on another processor element; or for terminal nodes, which are to be transferred from another processor element.

At each time, a reduction unit executes at most one task, indicated by the *active task pointer*. During execution a task may generate sequential sub-tasks, which are executed immediately (like subroutine calls). After termination of such subtasks, the parent task can proceed directly.

A task also may spawn parallel processes. These are encoded into a process message and sent to the communication unit. A placeholder node for the result of the parallel task is generated.

If a task encounters a situation where specific data is not available, it is suspended. If data from another processor element is needed, a request message is sent and a placeholder node is generated. This placeholder node may have been generated before, making the request message superfluous, e.g. when the task is waiting for the result of a parallel subtask. Attached to the placeholder node is a list of waiting tasks. By adding its own node address, the task is suspended. When the result becomes available, the list is moved to the *ready queue*, a list of ready-to-run tasks.

A context switch, triggered by a task termination or suspension, can easily be maintained by inspecting the ready queue for another task. If the queue is empty, the communication unit is asked for a new process. Once started, a parallel process remains on the same processor element and does not migrate.

2.3 Stackbased Reduction

In the previous section we have seen the basic concepts of the graph based machine. We have seen that all control and data information is stored together in the graph component. A graph organisation of control information is well suited for parallel implementations (due to the decentralisation), but especially for sequential computations the graph representation is not straightforward; too much knowledge about the call structure of runtime information is lost. Therefore we tried to separate control and data, and introduced a stack component (see Figure 2), which is known to be efficient in sequential implementations.

On the implementation level we expect better measurements in runtime as in space. On the conceptual level we claim that it is more natural to separate control and data.
The central component of the reduction machine is now a function stack which contains the activation blocks of recursive function calls. The heap only contains terminal and placeholder nodes.

The Meshed Stack Technique

An activation block of the stack component contains mainly the same information as a task node: some status information, the argument list and some space for local computations (we will call this the local data stack of the activation block). The space demands can be determined at compiletime, so pushing data on the local data stack will never "leave" this activation block.
In order to evaluate a function application, the addresses of the graph representations of the arguments are written on top of the local data stack and the corresponding function call is executed. This is done by creating a new activation block on top of the function stack, moving the pointers to the function arguments from the (old) data stack to the (new) function block, reserving cells for local variables and local computations, writing the value of the instruction pointer as return address and the dynamic link to the current activation block into the new activation block and finally jumping to the corresponding program address.
Thus, each process works on its own private function stack area. For evaluating function calls, new activation blocks are always generated on top of the stack. Parallel processes are spawned by generating a placeholder node and a process message as described before. Our meshed stack keeps the runtime information of several parallel processes on a single stack. Its main difference with respect to an ordinary stack is that the current activation block need not be on top of the stack. New activation blocks will always be generated on top of the stack, but if a function call terminates and its activation block is not on top of the stack, it can not be removed immediately. If a parallel process is suspended due to locally not available information, its activation blocks are kept on the stack until its evaluation can continue. In the meantime, the reduction machine can proceed with the evaluation of another parallel process, whose activation blocks will simply be allocated on top of the activation blocks of the suspended process. When the new process is suspended or terminates and the former process is reactivated, newly created activation blocks of this evaluation will be allocated on top of the activation blocks of the previous evaluation. Thus we will get a mixture of activation blocks belonging to different parallel processes on the stack, which is the reason for calling it a meshed stack. This explains the main idea of the meshed stack technique.

3 Graphbased vs. Stackbased Reduction

In Figure 3 a cactus stack and its implementation with a meshed stack is shown. In this example, process 1 (a) spawns two parallel processes 2 and 3 and continues with a local computation (b). Process 1 (b) needs data from another processor element and has to be suspended. In the meantime, process 2 has been transferred to processor II, but process 3 is still available and can be executed. Process 3 also has to wait for a result and is suspended. Process 1 (b) can be reactivated because the data has become available. 1 (b) now calls the local computation 1 (c), etc.

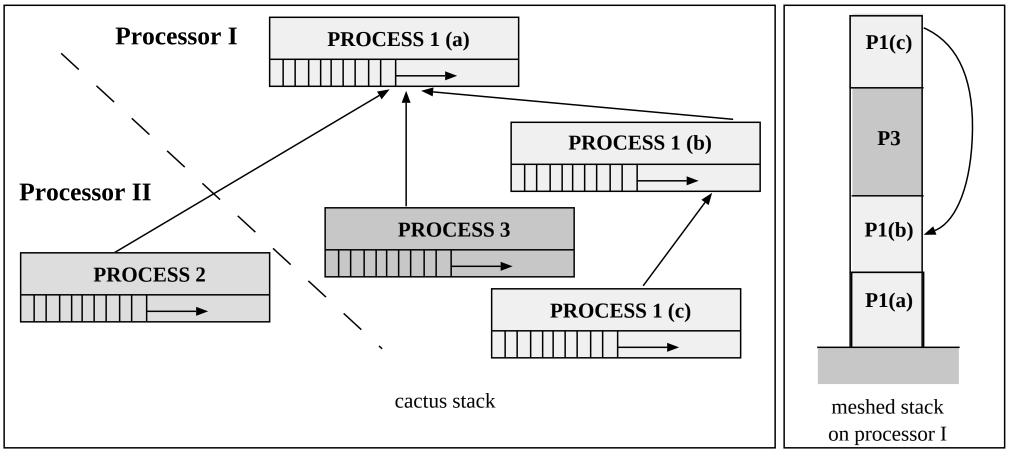

Figure 3: Implementation of a Cactus Stack on a Meshed Stack

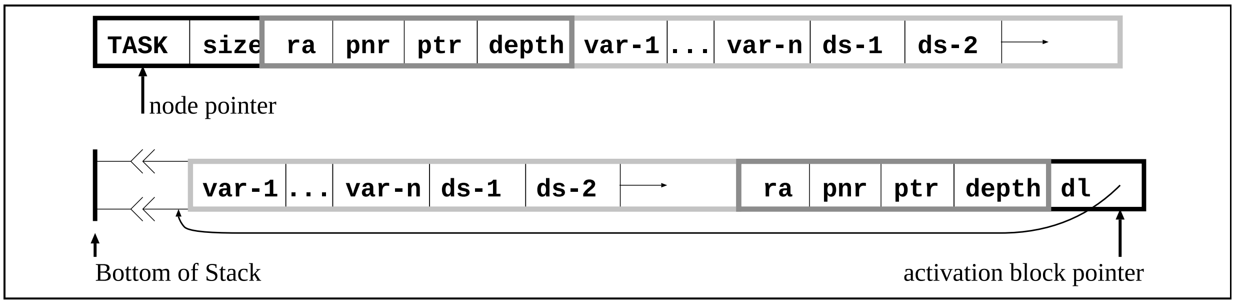

Figure 4: Task Node and Activation Block Structure

If we look at the corresponding meshed stack of processor I, we find the activation blocks of these processes interleaved, since the growth of the meshed stack reflects the temporal behaviour of the call and suspension structure.
After we have explained the basic techniques, we now compare the two approaches. First of all, we will take a closer look at the contents of a task node and an activation block, which leads us to the topic of process creation.

3.1 Process Creation / Allocation

In Figure 4 the structure of a task node and an activation block is shown. In both blocks, there are cells for the parameter list, space for local variables and for local computations (local data stack). Both representations need to know the return address (*ra*, which is zero, if the function was invoked in parallel), the processor number (*pnr*) and local address of the parent of the process (*ptr*, in the global case the address of a placeholder node, in the local case the address of the calling block), the current depth of the local data stack (*depth*, has to be updated when the process is suspended) and the size of the block (*size* in the task node, *dl* as the dynamic link in the activation block). Since only the task node has to be tagged, it needs one more entry.
On the implementation level, there is no vital difference between allocation of an activation block on the stack and allocation of a task node on the heap at first sight. Both "blocks" carry virtually the same information and therefore have equal memory demands.
Nevertheless, there is a quite subtle difference on the conceptual level: Heap allocation *can* be done as efficiently as stack allocation by the incrementation of a heap pointer [Appel 87]. This efficient heap allocation depends on the underlying memory management, and especially the underlying garbage collection technique. E.g. if one choses a *mark-and-scan* garbage collector, free lists have to be maintained. So if one writes an allocation

procedure for stack blocks, one can use the knowledge of the special structure simply by writing information to the memory address indicated by the Top_Of_Stack pointer and increasing the pointer. On the other hand, if one wants to create a task node on the heap, there is no general allocation strategy. A special heap-allocation procedure has to be invoked, which returns a free heap address. This step was eliminated by stack allocation.

3.2 Process Suspension

Examining process suspension we do not find any differences. Both representations, stack block or task node, contain all necessary information to evaluate the function call. Therefore the corresponding process is characterized by one pointer to the stack block or the node, and can be suspended simply by saving this pointer.

3.3 Process Termination / Garbage Collection

The handling of terminated processes, especially the reuse of these memory cells (garbage collection) is the most important topic of our comparison. We will explain the garbage collection technique for the stack furtheron and will examine it in the context of a *mark-and-scan* and a *mark-and-copy* heap collector. Of course, the advantage of the stack technique in contrast to a graph approach depends heavily on the efficiency of the (heap) garbage collector; more sophisticated collectors (e.g. generational garbage collectors) may outweigh the efficiency of the stack collector. Nevertheless, our stack collector is quite simple and easy to implement.

Due to the distinction between *control information* (in the stack component) and *data information* (stored in the heap component), one can handle each component with an appropriate garbage collection mechanism. E.g., the problem of data sharing needs only to be handled by the garbage collector of the heap component and causes no unnecessary administration overhead in the stack garbage collector. One major advantage of our approach is the fact, that the knowledge of the special stack structure can be used to develop a simple and efficient garbage collector for this component.

Garbage-Collecting the Meshed Stack

In general, activation blocks can only be released immediately from the stack if they are on top. Underlying blocks can be released subsequently until a non-garbage block is encountered. Experimental results show, that most blocks are executed on top of the stack, and therefore can be released immediately after execution. Within the stack, the termination of a process will cause a “hole” (see Figure 5). Suppose there is sufficient stack space, the above strategy would eventually release all blocks. Nevertheless, the stack may run out of space while there are still garbage holes within the stack, so a garbage collector must be provided.

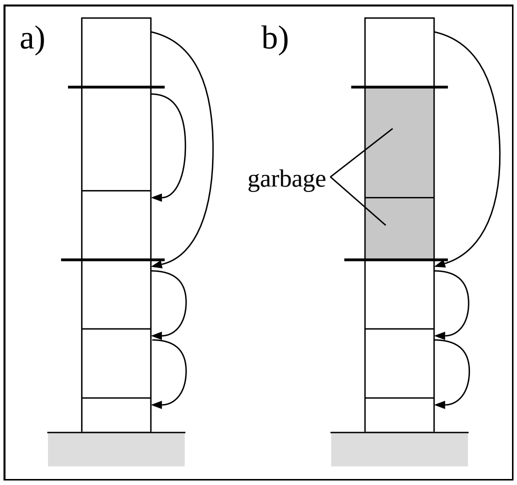

Figure 5 Function Stack Structure

The idea is to crunch the stack *in place*, when a stack overflow occurs. Since blocks get relocated, we have to update all references to the blocks. Therefore we introduce

```
procedure crunch_stack;
{ to = from = bottom_of_stack;                  % start at bottom
  while (from < top_of_stack)
     { while (garbage(from))                    % garbage blocks:
           { release_table_entry(id(from));     % release id and
               from = nextblock(from) }         % skip holes
         mem_copy(from, to, size(from));        % copy from-block
         rewrite_table_entry(id(from), to);     % new address entry
         to += size(from);
         from = nextblock(from) }               % examine block above
  top_of_stack = to                             % new Top of Stack
}
```

Figure 6: The Main Procedure for Garbage Collecting the Stack

an address-mapping table by adding an individual block id to the status information of activation blocks[1]. All references to the activation block are made through this id. The actual address of the block can be found in the table which maps the id's to stack addresses. If the garbage collector is invoked, the stack is simply crunched *in place*, deleting the holes.

The main procedure is outlined in Figure 6. Starting from the bottom, garbage blocks are skipped and their id's are released. Once a valid block is encountered, it is moved "down" the stack and the address entry in the table is updated. Since we traverse the stack "bottom up", the block lengths are only available on "the other side" of the blocks (dynamic links), therefore the mapping table also has to provide the block sizes. This process is iterated until the top of the stack is reached.

On termination of a process that is executed on an inner block, this garbage block can be melted with adjacent "holes". This results in a "normal form" of the stack: between two active blocks there is at most one garbage block. In the above procedure, the inner `while`-loop could be replaced by an `if`.

The indirection introduced by the mapping table causes no significant overhead, since the mapping of id's to actual addresses is only resolved during context switches: References to the stack occur *only* locally as *waiting task id's*, attached to the placeholder nodes. Only when a waiting task is reactivated, it looks up its actual stack position in the table.

The garbage collection process is completely transparent to the rest of the environment and can be done independently. More details on the stack garbage collector are given in [Schmidt 93].

As already mentioned, the success of the stack technique depends on the overall garbage collection technique. Therefore we will compare the stack and graph approaches in conjunction with different heap collectors.

4 Experimental Results

We have implemented an interpreter version of both reduction techniques on a transputer system under Helios C. For the global garbage collection we have implemented the

[1]For the id entry we can "reuse" the task node tag, so now an activation block and the corresponding task node have the same length. As for the garbage marker, we can simply reuse the processor number (*pnr*) entry by overwriting it with an invalid number (e.g. 0) on termination.

distributed garbage collector of [Lester 89]. It is a hybrid scheme with a (global) reference counting garbage collector. The weights are maintained by in- and outtables on each processor element. This technique makes the overall garbage collector transparent to the local machine and allows us to examine different local collection strategies on each reduction unit. It would even be possible to run diffenent garbage collectors on different processor elements.
In this section we will show some experimental results based on the evaluation of the *quicksort*-function for a list of 150 elements. We will discuss the two most common garbage collectors, namely *mark-and-scan* and *mark-and-copy*. In the future we will also implement other collectors, like reference counting and generational collection.

4.1 *mark-and-scan*

In a *mark-and-scan* garbage collector all active nodes are marked. The heap is scanned form bottom to top and all garbage nodes are released by adding the addresses to a free list (possibly a couple of free lists for different node sizes). Allocation must be done via the free lists and may be expensive (if the free list of the appropriate size is empty, a first-fit or best-fit strategy has to be used), and may cause fragmentation of the heap. The main advantage of this collector is that the nodes are not relocated, so there is no overhead in relocating pointers to the structures.
We now give statistical data for a sample run of *quicksort* with a list of 150 elements on two processors. Each reduction unit has 20k cells of heap memory (see Table 1)[2].

Pr #	t	cells total	cells used	nodes used	used %	Δt (ms)
1	0.29	20430	4301	318	21.1	20
1	0.59	20050	6020	473	30.0	29
1	1.03	20386	6216	486	30.5	51
2	1.21	20436	2542	201	12.4	69
1	1.49	20386	7175	566	35.2	47
2	1.84	19670	3349	265	17.0	41
1	1.91	20386	6229	496	30.6	44
2	2.22	19670	3066	269	15.6	34
1	2.34	20386	7000	567	34.3	49

Table 1: *quicksort(150)* on two processors, 20k heap cells, *mark-and-scan*

In total, 8596 nodes were allocated on the heap, whereof 4526 were tasknodes (= 53%). For parallel evaluation 140 tasks were created (= 3% of the task nodes), but only 40 were evaluated on the other reduction unit.
The runtimes of the collection cycles do not seem to have any connection with the number of nodes (in the marking phase all valid nodes are traversed, in the scan phase *all* nodes are investigated). The total time is mainly influenced by communication times of the global reference counting collector, since the release of structured data often triggers decrement messages to other processors. This effect can be seen especially on processor 2, since the original structure to sort was generated entirely on processor 1. When structured data is sent to another processor element, only the root node is sent, the components remain on the original processor.

[2] t is the absolute runtime in seconds, when the collection cycle was completed.

For a fair comparison with the stack technique, we have to allocate some space for the stack (4k cells), yielding a rest of 16k heap cells. Note, that in the graph version, over fifty percent of the nodes were task nodes, but we do allocate only a small amout of the total space for the stack. In Table 2 we see, that garbage collection has decreased dramatically.

		heap collection					stack collection			
Pr #	t	cells total	cells used	nodes used	used %	Δt (ms)	cells total	cells used	blocks used	used %
2	0.82						4012	1718	78	42.8
1	1.57	16384	5781	509	35.3	85				
2	1.57						4016	728	33	18.1

Table 2: *quicksort(150)* on two processors, 16k heap cells + 4k stack cells, *mark-and-scan*

In total, 4119 nodes were allocated on the heap. 4526 activation blocks were created (that is exactly the number of task nodes). For parallel evaluation 146 activation blocks were created (= 3% of all blocks), 46 were evaluated on the other reduction unit. It is interesting to see, that most of the activation blocks have been released immediately after termination (3982 blocks = 88%), and only 111 blocks (= 2.5%) had to be copied during stack crunching.
In the pure heap version, 8498 nodes were examined during garbage collection (= 1062 nodes per collection), here we yield a total of 1728 nodes in the only collection. That is, because data nodes are much shorter than task nodes and so more nodes fit in the heap before an overflow occurs.
Also the runtimes were satisfying: in this particular example the graph version took 2.52 seconds, the stack version took 2.10 seconds. You might say we should have called the program *slowsort*, but please notice that this is an interpreter version, including statistical procedures and monitor output of the results. If we press the *turbo* button of our machine, we yield 2.06 and 1.98 seconds, respectively. This is a speedup of 5% in favour of the stack technique, which is quite reasonable.

4.2 *mark-and-copy*

For a *mark-and-copy* garbage collector, the heap is divided into two semispaces (*fromspace* and *tospace*) [Fenichel, Yochelson 69]. All heap cells are allocated in the fromspace by incrementing the top-of-heap pointer. If the garbage collector is invoked, the fromspace is scanned and all active nodes are copied to the tospace (leaving indirections for later depedencies). When the copy phase is finished, fromspace and tospace are flipped.
Note that this principle can also be found in the stack collector. However, the stack collector differs in many ways. First of all, marking is done already during reduction (more precise, on termination of an activation block). Also, we have no problems with breadth-first or depth-first search strategies, since the stack structure is linearised. Furtheron, it is not necessary to divide the memory into two semispaces, since the crunching of the stack is done "overlapped" on one space. So one can see the stack collector as *one-space linearised mark-and-copy* collector.
Again, we run *quicksort(150)* on two processors with 20k heap cells (see Table 3), so one semispace now has 10k heap cells. In total, 8515 nodes were allocated on the heap, whereof 4526 were tasknodes (= 53%). For parallel evaluation 142 tasks were created (= 3% of the task nodes), 40 were evaluated on the other reduction unit.

Pr #	t	cells total	cells used	nodes used	used %	Δt (ms)
1	0.14	9747	3971	244	40.7	16
1	0.25	9760	4541	308	46.5	19
1	0.35	9745	4781	356	49.0	21
1	0.45	9744	4871	398	50.0	23
1	0.57	9751	5469	462	56.1	27
2	0.75	9743	2988	207	30.7	21
1	0.91	9761	5558	474	56.9	32
2	0.99	9746	3428	253	35.2	27
2	1.26	9742	3112	244	31.9	31
2	1.30	9753	3641	314	37.3	20
1	1.36	9744	5452	471	56.0	34
2	1.47	9742	2823	251	29.0	25
1	1.54	9741	5838	499	59.9	36
1	1.69	9747	5737	492	58.9	37
2	1.85	9746	3825	296	39.3	24
2	2.00	9755	2926	273	30.0	32
1	2.05	9743	5196	443	53.3	33
1	2.19	9749	5402	468	55.4	37
1	2.34	9746	5329	467	54.7	34

Table 3: *quicksort(150)* on two processors, 20k heap cells, *mark-and-copy*

For a comparison with the stack technique, we allocate, as before, 4k cells for the stack, yielding a rest of 16k heap cells (8k for each semispace). Again, garbage collection is decrased significantly (see Table 4).

In total, 4041 nodes were allocated on the heap. 4526 activation blocks were created (fortunately again the number of task nodes). For parallel evaluation 143 activation blocks were created, 48 were evaluated on the other reduction unit. These figures are quite similar to our observations in the previous paragraph. Again, the immediate release of most of the the activation blocks (4044 blocks = 89%) is responsible for the avoidance of garbage collection cycles. Only 106 blocks (= 2.3%) had to be copied during stack crunching.

In the pure heap version, 6920 nodes were copied during garbage collection, here we yield

		heap collection					stack collection			
Pr #	t	cells total	cells used	nodes used	used %	Δt (ms)	cells total	cells used	blocks used	used %
2	0.77						4016	1584	72	39.4
1	1.06	8151	5057	479	62.0	38				
2	1.35	8157	2120	204	26.0	50				
2	1.45						4018	742	34	18.5
1	1.50	8159	4692	433	57.5	38				
2	1.94	8151	1935	187	23.7	41				
1	2.00	8157	3603	332	44.2	29				

Table 4: *quicksort(150)* on two processors, 16k heap cells + 4k stack cells, *mark-and-copy*

a total of 1741 copies. So, although in principle the stack cruncher is a copy collector, we avoid almost totally the copying of task nodes resp. activation blocks, and this by using only a fifth of the total heap size[3]!
As for the runtimes, in this example the graph version took 2.69 seconds, the stack version took 2.42 seconds. Again, if we hurry up a bit, we yield 2.17 and 2.07 seconds, respectively. This is also a speedup of 5%.

4.3 A Stack Profiler

We implemented a small stack profiler, which gives us an insight on the structure of the stack throughout the reduction process. In Figure 7 the profile of the stacks on both processors for a sample run of *quicksort(150)* is shown. In y-direction, valid activation blocks are indicated by small vertical black lines, garbage blocks are not drawn. So it is easy to see where garbage blocks build up and how the release of a top-of-stack block also enables the release of huge garbage sections.

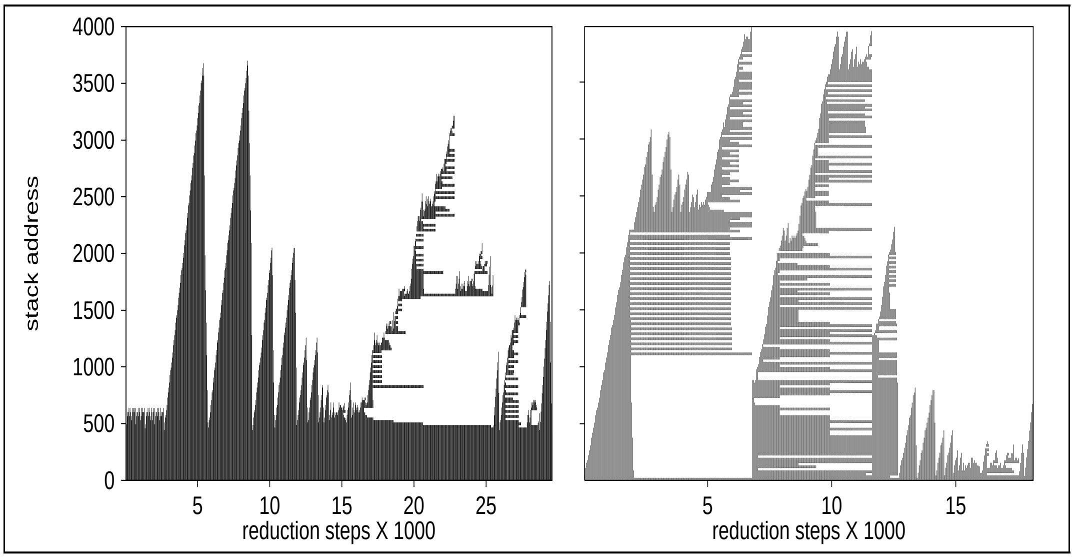

Figure 7: Stack Profiles on Processor 1 & 2

A closer examination of the stack structure on processor 1 shows the general call structure of the *quicksort*-program (cf. Figure 1): In the first 3000 steps, a list to sort is built, this is purely sequential work. The first peak shows the recursive calls of the *filter*-function as argument of the parallelised *quicksort* call. This task is executed on processor 2 (so as for absolute timing, the first reduction step on processor 2 coincides with reduction step 6000 on processor 1). The second peak belongs to the recursive calls for the other *filter*-function as argument of the local *quicksort* call. This structure can be found subsequently for shorter lists and therefore smaller peaks. At the end of the computations, larger peaks are due to the execution of *append* which glues together the results. On processor 2 the structure is not that regular, since the entire data structure to sort was build on the first processor element. So data has to be requested, which leads to more suspensions.
In this example, stack crunching occured twice on processor 2: at reduction step 7000 and again at 12000. While in the first case most activation blocks were in the upper half of the stack, in the second case they were distributed equally over the stack.

[3]Note that task nodes occupied about half of the heap.

This stack profiler gives an interesting idea of the progress of reduction in a parallel system, which is normally hidden to the user. The profiler can easily be extended (e.g. by adding colours) to show which blocks belong to which function calls.

5 Related Work and Conclusions

Most parallel implementations of functional languages rely on the graph-based model of reduction ($\langle \nu, G \rangle$-machine of [Augustsson, Johnsson 89], the HDG-machine of [Kingdon et al. 91], ALFALFA [Goldberg, Hudak 86], PAM [Loogen et al. 89]). There have been attempts to incorporate stack techniques ([Koopman et al. 90], [Peyton-Jones 91]), but only on the sequential level. We presented an alternative extended stack technique for the management of control structures in distributed implementation of dynamic process systems. The technique has been presented in the context of functional languages, but in principle it is independent of the chosen programming language.
The basic idea of the meshed stack technique dates back to 1973, when Bobrow and Wegbreit [Bobrow, Wegbreit 73] proposed a single stack that can run also activation blocks inside the stack. If the block claims more memory, and there is no space immediately above the block, it has to be copied on top of the stack. We avoid this copying by calculating the maximum storage demands at compiletime. The authors already pointed out possible applications like backtracking and multitasking. Indeed, this idea is also realised in the WAM [Warren 83].
A similar approach has been taken in [Hofman et al. 92], but there only activation blocks on top of the stack are allowed to be executed. So inner activation blocks may be blocked by the top-of-stack task. Note that they consider a fork/join task model. In our environment, where sharing of subtasks is allowed, this approach could result in deadlock situations. Nevertheless, their approach yields a good cache behaviour, which should also be observed in our machine.
Our technique allows for a special efficient garbage collection of control structures. We have seen, that we get garbage collection of control structures almost for free, since most of this structures can be released immediately. This results in less memory usage. If stack crunching is necessary, only a small percentage of the stack blocks is alive, so the proposed *in-place copy collector* is very efficient.
Nevertheless, more sophisticated heap collection techniques could outweigh these advantages, but the stack approach has still the advantage of conceptual clarity: control and data are seperated, and the extension of the stack technique of the sequential implementations to the parallel case seems more natural.

Acknowledgements

The authors would like to thank Christian Queinnec for drawing our attention to the paper of Bobrow and Wegbreit [Bobrow, Wegbreit 73], and Markus Mohnen for careful proofreading of the draft version of this paper.

References

[Appel 87] A.W. Appel: *Garbage Collection can be faster than stack allocation*, Information Processing Letters, 21(4):275–279, 1987.

[Augustsson, Johnsson 89] L. Augustsson, Th. Johnsson: *Parallel Graph Reduction with the $\langle \nu, G \rangle$-Machine*, Conf. on Functional Progr. Lang. and Comp. Arch. (FPCA), Addison-Wesley 1989.

[Bobrow, Wegbreit 73] D.G. Bobrow, B. Wegbreit: *A Model and Stack Implementation of Multiple Environments*, Communications of the ACM 16 (10), 1973.

[Fenichel, Yochelson 69] R.R. Fenichel, J.C. Yochelson : *A LISP garbage collector for virtual-memory computer systems*, Communications of the ACM 12 (11), 1969.

[Goldberg, Hudak 86] B. Goldberg, P. Hudak: *ALFALFA — Distributed Graph Reduction on a Hypercube Multiprocessor*, Workshop on Graph Reduction, LNCS 279, Springer 1986.

[Hogen et al. 92] G. Hogen, A. Kindler, R. Loogen: *Automatic Parallelization of Lazy Functional Programs*, Europ. Symp. on Progr. (ESOP '92), LNCS 582, Springer 1992.

[Hogen, Loogen 94] G. Hogen, R. Loogen: *Efficient Organization of Control Structures in Distributed Implementations*, Compiler Construction (CC '94), LNCS 786, Springer 1994.

[Hofman et al. 92] R. Hofmann, K. Langendoen, W.G. Vree: *Scheduling Consequences of Keeping Parents at Home*, Proceedings of ICPADS'92, December 1992.

[Kingdon et al. 91] H. Kingdon, D. Lester, G. Burn: *The HDG-Machine: a highly distributed graph-reducer for a transputer network*, Computer Journal 34 (4), 1991.

[Koopman et al. 90] P. Koopman, M. van Eekelen, E. Nöcker, J. Smeetsers, R. Plasmeijer: *The ABC-machine: A Sequential Stack-based Machine for Graph Rewriting*, Technical Report 90–22, University of Nijmegen 1990.

[Lester 89] D. Lester: *An Efficient Distributed Garbage Collection Algorithm*, Conf. on Parallel Arch. and Lang. Europe (PARLE) 1989, LNCS 365, Springer 1989.

[Loogen et al. 89] R. Loogen, H. Kuchen, K. Indermark, W. Damm: *Distributed Implementation of Programmed Graph Reduction*, Conf. on Parallel Arch. and Lang. Europe (PARLE) 1989, LNCS 365, Springer 1989.

[Peyton-Jones 91] S.L. Peyton-Jones: *The Spineless tagless G-machine: Second Attempt*, Proceedings of the Workshop on Parallel Impl. of Functional Lang. , Technical Report CSTR 91–07, University of Southhampton 1991.

[Schmidt 93] A. Schmidt: *Garbage Collection of Meshed Stacks*, Diploma Thesis (in german), RWTH Aachen 1994.

[Warren 83] D.H.D. Warren: *An abstract Prolog instruction set*, Technical Report 309, SRI International 1983.