A High-Level Target Language for the Compilation of Dataflow Programs

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria
Tel: $*(7236)3231-66$, Fax: $*(7236)3338-30$
E-Mail: `schreine@risc.uni-linz.ac.at`

November 1991

Abstract

We present a new model $\mathcal{DF}$ for the implementation of non-strict but non-lazy functional languages (dataflow languages) on conventional parallel hardware. $\mathcal{DF}$ consists of a small set of functions that can be added to any imperative language and allow to express dataflow behaviour in a simple and efficient way. The purpose of this model is to provide a high-level and machine-independent target language to which dataflow languages can be compiled for parallel execution. A binding of the $\mathcal{DF}$ model to the C language has been implemented on a multi-transputer system and shows promising results.

Keywords: Parallel Programming Paradigms; Implementation of Parallel Languages; Data Flow.

1 Introduction

The programming of parallel computers in functional languages gains more and more interest. A functional program is a set of mutually recursive function definitions where the order of function applications is not specified. Functions that do not depend on the result of each other may be executed in any order, in particular they may be executed in parallel (*horizontal parallelism*, e.g. between g and h in $f(g(e_1), h(e_2)))$. Additional parallelism is provided by functional languages with *non-strict semantics* where a function can execute in parallel with the evaluation of its arguments (*vertical parallelism*, e.g. between f and g in $f(g(e)))$. This makes sense if a function does not immediately need its arguments to proceed execution, in particular if an argument is a compound data structure that can be produced in parallel with its consumption.

The *dataflow model* is a technique to exploit the parallelism of non-strict functional languages where each expression is evaluated when the required data are available (data-driven execution). This model is often connected with special purpose architectures (dataflow machines) but it can also serve as a basis for the compilation of functional languages for conventional parallel computers. For instance, a compiler might generate code for an abstract dataflow machine [5] that is is then boiled down to actual machine code. As an alternative approach, also a conventional imperative language might be chosen as the compilation target. The compiler for the functional language would then become simpler and the imperative target program could be executed on any architecture by recompilation with conventional compilers.

Of course, there are several features of dataflow languages that cannot be directly compiled into an imperative form. Moreover, for each parallel dialect of an imperative language, there exist different extensions that are not portable at all. One solution of this problem is to encapsulate all machine-dependent features into a small set of library functions that offer the necessary facilities to express the parallel behaviour of the functional program. This library has to be ported to each parallel architecture, while the compiler and the target programs may remain unchanged.

In this paper, we present our programming model $\mathcal{DF}$ (for dataflow) consisting of a few primitives that allow to express dataflow behavior in any imperative language. A binding of this model to the C language has been implemented on a multi-transputer system and several dataflow programs have been manually translated into this form. After the presentation of $\mathcal{DF}$, we describe its implementation and the results we gained from several experiments. An outlook on our further work concludes this paper.

2 The Programming Model $\mathcal{DF}$

2.1 Required Features

In this section, we gradually develop an imperative parallel programming model that may serve as a portable and efficient target for the compilation of a dataflow language

```
-- sort list
quicksort ::  [int] -> [int]
quicksort l = qsort l [ ]

-- sort h:t and append r
qsort ::  [int] -> [int] -> [int]
qsort [ ] r   =  r
qsort h:t r   =  qsort a h:(qsort b r)
                 where (a,b) = split h t

-- partition h:t by e into two sublists
split ::  int -> [int] -> ([int],[int])
split e [ ]   =  ([ ], [ ])
split e h:t   =  (h:a,b), if h < e,
              =  (a,h:b), else
                 where (a,b) = split e t
```

Figure 1: Quicksort

on conventional parallel machines. The properties that such a model must fulfill shall be demonstrated by a concrete example: consider the function `qsort` in Figure 1 that represents a generalized version of the Quicksort algorithm that sorts the first input list and appends the second one (and by this generalization avoids an additional append operation). Figure 2 shows the dataflow graph of the recursive branch of the function.

`split` partitions the first input list t by its head h into two sublists a and b such that a contains all elements less than h and b the remaining ones. The first recursive invocation of `qsort` sorts b and appends r and passes this list to the second recursive `qsort` invocation that concatenates it with the sorted version of a. The function does not contain any horizontal parallelism, since the second `qsort` invocation depends on the result of the first one which itself depends on the result of `split`. Hence, in a strict execution model the program would be executed in a totally sequential fashion.

However, a non-strict model offers lots of inherent vertical parallelism: since `qsort` processes its first input list one element after the other and only in its final step requires the second list, `split` can work in an overlapping way with both `qsort` invocations. While `split` produces the elements of a and b one after the other, both `qsort` branches can independently process these elements; synchronization between both branches takes only place when finally one `qsort` needs the result of the other one.

This program demonstrates very well various principles that an imperative programming model must obey to be a suitable target for a dataflow language:

- **Processes:** Functions play the role of processes, i.e. the unit of parallelization is the user-defined function (function-level parallelism, macro-dataflow).

- **Channels:** Shared variables represent the communication medium between processes. Communication is directed and one-to-many.

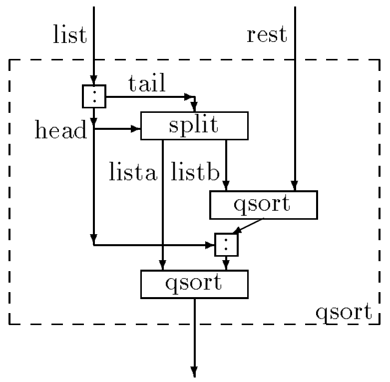

Figure 2: Dataflow Graph of `qsort`

- **Non-Strict Functions:** Functions can start execution independently of their arguments. Only when arguments are accessed, synchronization takes place.
- **Non-Strict Data Structures:** Data structures can contain empty slots. When such slots are accessed, synchronization must take place.
- **Garbage Collection:** Data structures are not explicitly deallocated. If a structure is no more referenced, it must therefore be automatically reclaimed.

2.2 The $\mathcal{DF}$ Kernel

In this subsection, the kernel of our $\mathcal{DF}$ model is presented that consists of the following types, constants and procedures:

REF denotes the type of all references to $\mathcal{DF}$-structures (structures in short). This type is word-sized and includes the constant NIL.

PROCNUM is the number of processors in the system (each of which has a unique address in the range from 0 to PROCNUM-1).

ACTPROC is the address of the current processor, i.e. the processor on which the current process is executed.

NIL denotes a special constant of type REF that is different from any valid structure reference.

par($\downarrow$*fun*, $\downarrow$*proc*, $\downarrow$*n*, $\downarrow arg_1$, $\downarrow arg_2$, ..., $\downarrow arg_n$) starts on processor *proc* a process executing *fun*(arg_1, arg_2, ..., arg_n). *proc* must be either some processor address or -1 (in that case an automatic strategy for process distribution is applied). *fun* must not return any value, not reference any global environment (except the constants described above) and not cause any side-effects (besides the operations described below).

new($\downarrow$*size*, $\uparrow$*ref*) allocates a structure with *size* slots and returns a reference *ref* to this structure. All slots are numbered from 0 to $size-1$ and may contain any word-sized value. All slots are initialized to the special constant $\perp$ (read "empty") that is different from any value that can be stored in that slot.

get($\downarrow$*ref*, $\downarrow$*index*, $\uparrow$*value*) reads the slot at position *index* of the structure denoted by *ref* and returns the word-sized *value* stored there. *ref* must be a structure reference different from NIL and *index* must be a valid slot position in this structure. If the slot contains $\perp$, the current process is blocked until another process writes some value into that slot (which is then returned as *value*).

put($\downarrow$*ref*, $\downarrow$*index*, $\downarrow$*value*) writes into the slot at position *index* of the structure denoted by *ref* the given word-sized *value*. *ref* must be a structure reference different from NIL and *slot* must be the position of a slot in this structure that contains $\perp$. All processes that have been blocked on an attempt to read this slot are released.

The $\mathcal{DF}$ model introduces two basic concepts:

- **Parallel Processes** created by the procedure par and
- **$\mathcal{DF}$-Structures** generated and accessed by new, get and put.

$\mathcal{DF}$-structures are just arrays with some properties that allow them to serve as a tool for communication and synchronization between parallel processes: Each slot may contain the special constant $\perp$ ("the slot is empty") and any process that tries to read an empty slot is blocked. Furthermore, a process may only write values into empty slots ("single assignment rule"), which releases all processes blocked on this slot. Actually, $\mathcal{DF}$-structures represent a modified version of I-structures [1] that has been redesigned for the practical application in an imperative language.

2.3 A Sample $\mathcal{DF}$ Program

Figure 3 shows a translation of the `qsort` function of subsection 2.1 into the $\mathcal{DF}$ kernel bound to some imperative language. This translation is guided by the attempt to construct the program's dataflow graph as illustrated in Figure 2:

1. **Allocate Channels:** Channels (that denote the arcs of the dataflow graph) are created to establish the communication between the parallel processes. These channels are actually empty structures into which some processes write their results while other processes read their inputs from them.
2. **Spawn Processes:** Then the processes (that denote the nodes in the dataflow graph) are started with those structures as input arguments that correspond to input or output channels. The processes read their input values from these structures and write their results into them.

3. **Local Operations:** Finally, several operations are performed by the current process. These operations can be considered as an optimization that makes the creation of an additional process superfluous by inlining its code into the body of the current procedure.

The $\mathcal{DF}$ version of `qsort` in Figure 3 demonstrates these three phases: First the communication channels are allocated, then one `split` and two `qsort` processes are spawned and finally several local operations are performed that connect the input channels of the procedure with the input channels of the spawned processes: The head and the tail of the input list are received and written into the input structures of the `split` process. Furthermore, the head is linked with the result list of the second `qsort` process and a reference to this new list is written into an input structure of the first `qsort` invocation.

In this program, all process arguments are references to structures that will (but might not yet) contain the input arguments or will receive the result values, respectively. They are *not* the input arguments themselves! This feature allows to model-non strict behaviour in an imperative language with a call by value (i.e. strict) argument passing mechanism. A process can start execution even if one of the input arguments has not yet been written into the input structure. On an attempt to read the empty structure slot (because the input argument is necessary for further computation), the process is automatically blocked until the required value is available.

Only because of this feature, parallelism between both `qsort` branches is possible. `qsort` reads the value stored in its second argument *rest* only in the the base case of the recursion, i.e. if the input *list* is NIL and the value of *rest* has to be written into the *result* structure. In the recursive branch, *rest* is just forwarded to one `qsort` process without demanding its contents. Hence, it makes sense to start two `qsort` processes to execute in parallel even if the result of the one (*resultb*) is input to the other one (in the form of the *cons* argument that represents the list *hd:resultb*).

However, in most cases a clever compiler may optimize this non-strict form of argument passing to the more efficient call-by-value mechanism. E.g. the *head* and *tail* structures that are passed to the `split` process are totally superfluous. They will receive the head and tail of the input *list* after all processes are spawned. An automatic strictness analyzer could tell us that that without these values neither `split` nor one of the `qsort` processes can start execution. Hence we might as well get these values before `split` is spawned and pass them directly to the process.

From this example, we can check that the $\mathcal{DF}$ kernel already fulfills four of the five properties demanded in Subsection 2.1:

1. **Processes:** The par statement creates a parallel process that executes a particular function, which is exactly the desired property.

2. **Channels:** Communication is performed by a shared $\mathcal{DF}$-structure that is allocated in advance and passed to all participating processes.

3. **Non-Strict Functions:** The non-strictness of functions is achieved by the non-strictness of data-structures that contain the actual input values.

```
------------------------------------------
-- result := qsort(list, rest)
------------------------------------------
proc qsort(REF list, REF rest, REF result)

  REF head, tail, lista, listb;
  REF cons, resultb, tl;
  INT hd;

  -- get actual list reference
  get(list, 1, ↑list);

  -- base case
  if list == NIL then
    get(rest, 1, ↑rest); -- result := rest
    put(result, 1, rest);
    return;
  end;

  -- allocate channels
  new(2, ↑head); new(2, ↑tail);
  new(2, ↑lista); new(2, ↑listb);
  new(2, ↑cons); new(2, ↑resultb);

  -- start processes
  par(split, ACTPROC, 4, head, tail, lista, listb); -- (lista,listb) := split(head,tail)
  par(qsort, ACTPROC, 3, lista, cons, result);      -- result := qsort(lista, cons)
  par(qsort, next(), 3, listb, rest, resultb);      -- resultb := qsort(listb, rest)

  -- local computations
  get(list, 0, ↑hd); put(head, 1, hd);              -- head := list.hd
  get(list, 1, ↑tl); put(tail, 1, tl);              -- tail := list.tl
  put(cons, 1, resultb); put(resultb, 0, hd);       -- cons := hd:resultb

end qsort;
```

Figure 3: $\mathcal{DF}$ Version of Quicksort

4. **Non-Strict Data Structures:** The crucial point of $\mathcal{DF}$-structures is that synchronization automatically occurs on empty structure slots.

The essential idea of the $\mathcal{DF}$ model is that all the central issues of communication, synchronization and non-strictness can be expressed by a single concept, namely non-strict structures.

2.4 Indirections

A problem with our $\mathcal{DF}$ version of `qsort` is the fact that due to limited memory resources we can actually create processes only down to a certain level in the "recursion tree". Beyond this level, the `qsort` calls must be executed in a recursive and thus sequential manner. However, then a subtle difficulty arises: Each `qsort` process calls `split` and then steps down to the next level until for the first time the base case is reached (the process is than in one "corner" of the recursion tree). In this situation, `qsort` needs the argument `rest` from the next process:

```
if list == NIL then
  get(rest, 1, ↑rest);
  put(result, 1, rest);
  return;
end;
```

Unfortunately, that process will deliver this value only after it has traversed its whole recursion tree and so all processes must basically execute in sequence.

This situation is annyoing, since the implicit concatenation of results essentially contributes to the efficiency of a large class of functional programs. The problem did not arise in our original solution, since there each leaf of the recursion tree was a process of its own and all solutions could be computed in parallel. In a coarse-grain dataflow model, parallelism is often seriously restricted, because processes have to wait for values that are part of the total solution although they are not really "interested" in them. To break this vicious circle, we introduce the concept of *indirections* in $\mathcal{DF}$:

puti(↓*ref*, ↓*index*, ↓*ivalue*) This procedure behaves like *put* but may be only applied on a reference value *ivalue* (called *indirection*). Any get operation on slot *index* will not return *ivalue* itself but the contents of the slot *index* of the structure referenced by *ivalue*.

Now we can rewrite the non-recursive branch of `qsort` as follows:

```
if list == NIL then
  puti(result, 1, rest);
  return;
end;
```

The *result* cell does not receive the reference stored in *rest* but a reference to *rest* itself. Thus no get must be performed and the current process is not blocked.

Summarizing, the application of indirections allows to decouple processes such that the result of one may be used by the other without that the receiving process is blocked. Actually, an indirection is nothing but a cheap replacement for an additional process that is spawned to connect two $\mathcal{DF}$ structures without blocking the father process.

2.5 Garbage Collection

The $\mathcal{DF}$ model supports garbage collection based on the *weighted reference counting* method defined in [2]. The algorithm is simple to understand and perfectly suitable for a parallel system. Its basic idea is to associate a weight to each reference and to count in each structure the sum of the weights of all references to it. If this count drops to zero, the structure is not referenced any more and may be reclaimed. If a reference is copied, both copies get half of the original weight, which does not change the count in the referenced structure (this is the main advantage of this method in a distributed memory environment). If a reference is deleted, the count of the referenced structure is decreased by that weight.

Our version of this algorithm is based on three additional $\mathcal{DF}$ operations:

putr($\downarrow$*ref*, $\downarrow$*index*, $\downarrow$*rvalue*) behaves the same way as put but is used for writing a reference value into a structure slot. put itself is restricted to uninterpreted atomic values (integers, floating points, ...).

cop($\updownarrow$*ref*) halves the weight of the reference *ref* (if this weight is a power of two) or set it to zero (if the weight is one). This operation is usually performed before the reference is applied (see the description below).

del($\updownarrow$*ref*) deletes the reference *ref*. *ref* must not be applied any more (see the description below).

By the operation new, an empty structure is allocated and a reference to it is returned with an initial weight that is some power of two. The current process and all the processes to which the reference is communicated are now in charge of the correct management of the weight of the reference.

If the reference is *applied* (i.e. it references structures in get or put operations), the count in the corresponding structure is automatically decreased by the weight of the reference (i.e. it is assumed that the reference will not be applied any more). However, if we perform a cop before the actual application, the weight of the reference is halved and the reference count of the structure is only decreased by this value. Hence, further applications of the reference (with half of its original weight) are possible.

If we know that a reference is applied for the last time, the preceding cop operation can be omitted and the reference count is decreased by the whole weight of the reference (which means that the reference is considered to be deleted). However, if we miss the point of the last application (i.e. a reference is available but of no use any more), a del operation can be performed that has no other effect but decreasing the reference count in the structure. The application history of a reference *ref* therefore looks like

ref enters process;
cop(*ref*); *apply ref;*
cop(*ref*); *apply ref;*
...
cop(*ref*); *apply ref;*
apply ref;

where the final application (and only this one) may be a del statement.

The main advantage of our scheme is that in most cases the explicit deletion of references can be avoided and no additional access to the referenced tuple is necessary (which in a distributed memory environment probably involves a message to another processor). If the last application of a reference can be statically (i.e. at compile time) determined, the access operations get and put(i/r) themselves are in charge of decreasing the reference counts, since the messages submitted by these operations additionally carry the reference weight information.

If a reference has a weight of one that can not be halved any more, this weight is decreased to zero. Hence, one weight unity is lost and the reference count of the corresponding structure will never become zero again. If the initial weight is sufficiently large, this will be very rarely the case (moreover, for such structures the probability is very high that they will be alive during the whole runtime of the program). However, any implementor of the $\mathcal{DF}$ model is free to choose other strategies, e.g. the automatic insertion of indirection cells as suggested in [2].

3 Implementation of C-$\mathcal{DF}$

In this section, we present our implementation of C-$\mathcal{DF}$, a binding of $\mathcal{DF}$ to the C language, on a parallel system of 16 T800 transputers [3]. C-$\mathcal{DF}$ was written using the Logical Systems Transputer Toolset that contains a compiler for a parallel version of C with the posssibility of inlining assembly code [4]. The implementation consists of some 1500 lines of C code and several inline assembly instructions. The $\mathcal{DF}$-library that has to be linked to the user program is about 15 KB large.

3.1 Process Creation

The transputer has two special registers that determine the process under execution by the address of the next instruction and the address of the process workspace, respectively. Furthermore, there exists a queue of processes that are temporarily descheduled and wait for execution. If a process is descheduled, the address of its next instruction is written

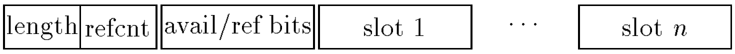

Figure 4: A $\mathcal{DF}$-Structure

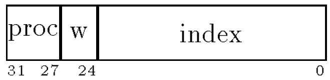

Figure 5: A $\mathcal{DF}$-Reference

into its workspace whose address is stored at the end of the process queue. Then the process at the front of the queue is scheduled for execution by loading the workspace and the instruction register with the corresponding values.

For starting a parallel process with the par routine, a new process workspace (whose size is determined by the user-definable variable WORKSPACE) is allocated from the heap. The workspace is initialized with the address of the function to be executed, its arguments and the return address of the process when it terminates. The process is then started by queuing its workspace into the list of descheduled processes.

3.2 $\mathcal{DF}$-Structures

A $\mathcal{DF}$-structure is constructed as depicted in Figure 4 (on transputers, the standard wordlength is 32 bits): The first two halfwords contain the sructure's length and its reference count, respectively. The next word contains pairs of available/reference bits: bit $2 * i$ of this word is set, if slot i of the structure contains a value, bit $2 * i + 1$ signals whether this value is a reference to another structure or not. The following words denote the structure slots. Since there is only one word reserved for slot managment, a structure may contain up to 16 slots.

Structure references are constructed as depicted in Figure 5: The most significant 5 bits denote the number of the processor that the referenced structure is stored on. The next

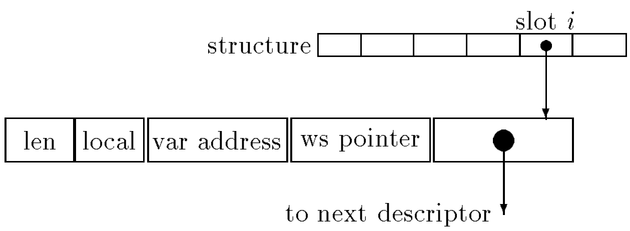

Figure 6: A Process Descriptor

3 bits contain the weight of the reference represented as its binary logarithm plus one (a value of zero denotes zero weight). The least significant 24 bits denote the position of the referenced structure relative to the start of the heap (in units of words). The kernel operations on $\mathcal{DF}$ structures are implemented as follows:

- new allocates a structure, stores its length and sets the reference count to the maximum weight. The available and reference bits are reset and the slots are initialized with NULL. The weight of the returned reference is set to its maximum.
- get delivers, if the slot is not empty, the stored value into the given variable (if the value is a reference, its weight is halved before). Otherwise, the slot points to to a list of blocked process descriptors. Then a new descriptor is allocated as depicted in Figure 6 and linked into the list. The process is descheduled leaving the address of the next instruction in the workspace.
- put traverses the list of process descriptors linked into the slot. For each process, the desired value is stored at the given address, and the process workspace is linked into the scheduling queue. Finally, the value is written into the slot and its available bit is set (as well as the reference bit in the case of putr).

Having fulfilled their task, get, put and putr decrease the reference count of the affected structure by the weight of the given reference (this is also the only purpose of the del procedure). If the count decreases to zero, the structure is reclaimed. The references stored in the reclaimed structure are deleted by decreasing the counts of the referenced structures which might now be recursively reclaimed, too.

3.3 The Communication System

The transputer is equipped with four links each of which allows transmission of messages to other processors. The counterpart of these physical links are software channels that allow uni-directional, unbuffered and point-to-point communication between processes. Each link represents two channels, one into each direction. Since the $\mathcal{DF}$ structure space is distributed among all processors, a system for communication between all processors had to be implemented consisting of one input process and one output process per link (see Figure 7).

- An **input process** receives a block of messages and stores it into a local buffer. Each message is managed according to its address: if a message has reached its destination, it is stored into a queue where a server process will manage it. Otherwise, a routing table is inspected fo forward the message through an output link.
- An **output process** moves all messages stored in the corresponding link buffer into a local send buffer and signals on the link that it is ready for communication. As soon as the input process on the other side has become ready too, the block of messages is transmitted and the output process can continue execution.

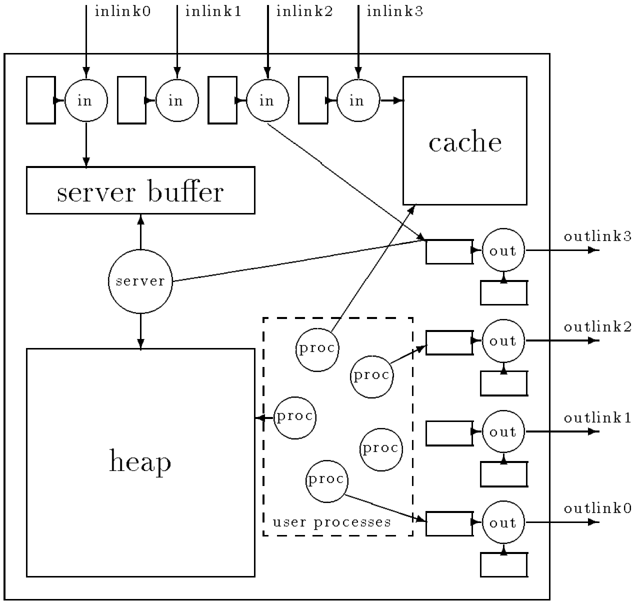

Figure 7: The Communication System

The communication scheme is free from deadlock as long as the link buffers do not overflow. Our experiences show that for sufficiently large buffers (about 32 messages) this case does never occur. The only topology-dependent part of our scheme is the routing table that can be be easily adapted to new interconnection networks.

On each processor, a server process manages all messages that have been sent from other processors. The input processes store these messages into a buffer from where the server can receive them. The messages denote either requests from remote user processes or answers to requests that have been issued by local user processes. Basically, there are six types of messages that are all of equal size (four words), with the exception of one double length message for process creation.

A problem is to provide the server and the link processes with enough computing time. If there are many user processes that compete with the system processes for execution, the overall system performance can be extremely degraded. Since the transputer's possibilities for priorizing processes are rather poor, we artificially decrease the average time that user processes are scheduled for execution by inserting additional “rescheduling points” at certain $\mathcal{DF}$ instructions. Each time that one of these points is reached, the user process is rescheduled and allows the next process to be executed.

3.4 Structure Caches

On distributed memory systems, the ratio between the processor power communication bandwith is still very unbalanced. Hence, a main concern in the design of the $\mathcal{DF}$ model

$\mathcal{DF}$-Operation	μs	C-Operation	μs	Factor
`new(s, r)`	55.9	`r = malloc(4*s)`	29.1	1.9
`put(r, i, v)`	21.6	`r[i] = v`	1.1	19.0
`get(r, i, v)`	25.0	`v = r[i]`	1.0	25.7
`par(f, 0, 6,` a_1`, ...,` a_6`)`	42.0	`f(`a_1`, ...,` a_6`)`	3.3	12.7

Figure 8: Speed of C-$\mathcal{DF}$ Operations

has been to minimize the number of messages to be exchanged (e.g. process creation requires only one message). However, the main load of the interconnection network is caused by the access to remote structures. If a structure is stored on one processor and demanded by a process on another processor, for each field of of this structure one request has to be sent and one message has to be returned.

In order to decrease network traffic, we introduced in our C-$\mathcal{DF}$ implementation the concept of a *cache*: if one field of a remote structure is accessed, also the other fields will be probably demanded. For example, if a list is processed, usually both the head and the tail field of each list cell are read. Thus, we transmit the whole cell on the first demand of one of its fields and store it in a cache on the requesting processor. If then the other field is demanded, no more network message is necessary. Moreover, if other processes on that processor demand this list cell too, they can immediately read the cell from the cache. Currently, we use cache entries with two slot fields, since the most-used data structures are lists with cells of two fields.

However, a cache reduces the number of messages only if both values of the cached section can be transmitted in one message and are both really demanded. Moreover, for correct garbage collection unfortunately one additional message is required, if a cache entry has to be replaced. Therefore the cache is actually not of vital importance for structures that are read only once but for those that are used by many processes. In this case, the cache is able to considerably reduce the amount of network traffic.

4 Experiments

4.1 Absolute Performance

It is common practice to restrict the presentation of implementations of parallel languages to to the relative speedups that can be achieved. However, we think that one should also speak frankly about the absolute performance of a system. We have therefore measured the average time of the $\mathcal{DF}$ kernel operations (all executed on local structures) and compared the results to the time required for performing corresponding operations of the C language itself. The figures are only the outcome of one particular test but can give a rough feeling about the range in that the performance of our implementation is settled.

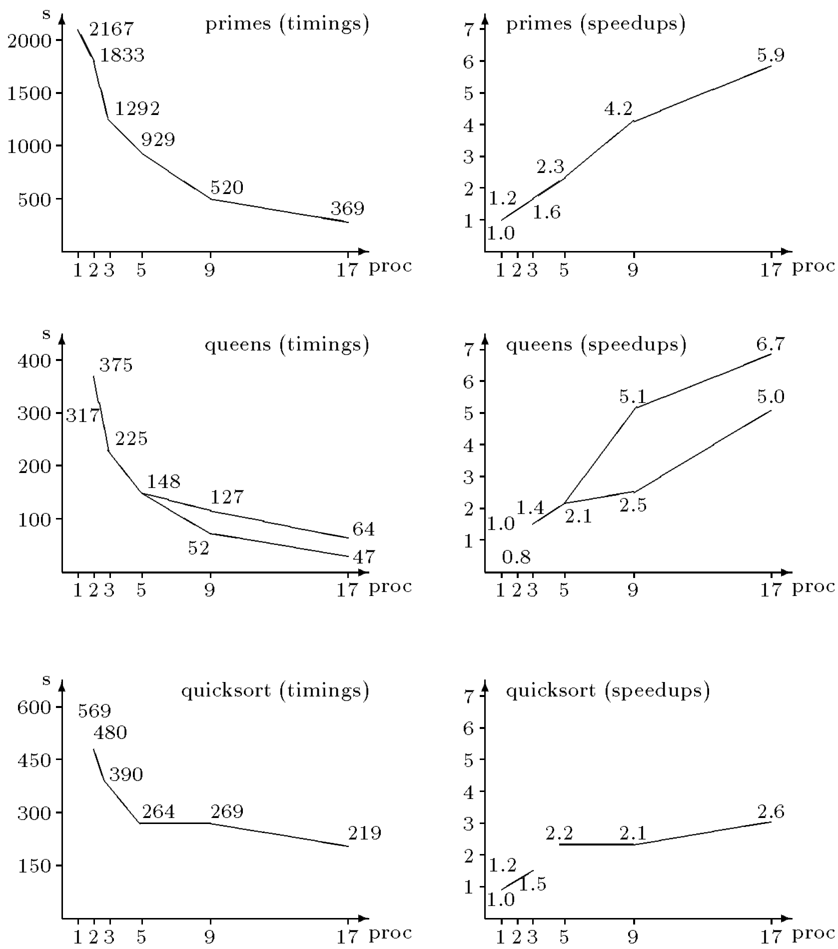

Figure 9: Benchmark Results

We think that these results (show in Figure 8) are quite satisfying. Due to lack of human resources, we concentrated our efforts during the implementation of C-$\mathcal{DF}$ mainly on portability (almost the whole code is written in C). For a really efficient implementation, several parts would have to be rewritten in assembly language which would increase the speed of the $\mathcal{DF}$ operations by a factor of 2–3. Moreover, many applications are not only time-bound but actually memory-bound. Since our implementation is very memory-effective, it gives us the possibility to experiment with programs that solve problems of considerable complexity.

4.2 Relative Speedups

We measured the performance of our transputer implementation of C-$\mathcal{DF}$ with three programs that represent the $\mathcal{DF}$ versions of typical parallel functional algorithms (see below). The transputer system was configured to hypercube topologies of dimensions 0 to 4 where an additional root processor was linked into one edge of the hypercube. Figure

9 shows the absolute times of these experiments and the relative speedups compared to the time that each program required on the root transputer. Since this processor runs at a higher clock frequency than the hypercube processors, all timings were adjusted to a common basis. The benchmars yielded the following results:

- `primes` computes all primes up to 1000000. The algorithm is based on vertical parallelism where each odd number is tested by a parallel process. The performance of this program scaled very well with the number of processors and lead to a maximum speedup of 5.9. The program profited much from the caching mechanism, since the same global list of primes is accessed many times by all processes on all processors.
- `queens` computes all solutions of the queens problem on a size 10 board. The program is based on a parallel traversal of the search tree; parallel processes are only created in the first level, respectively in the first two levels for the 17 processor configuration. Beginning with 9 processors, the automatic scheduling strategy performed significantly worse (speedup 5.0) than a manual process placement (speedup 6.7).
- `quicksort` is the version of the Quicksort algorithm described in Subection 2.1 applied to a list 64000 random numbers. Processes are created up to that level in the recursion tree such that all processors are satured with processes. In general, the achieved speedups were rather poor (up to 2.6) because of the large amount of communication necessary to transmit the intermediate lists between processors.

On the one hand, the results of these experiments are rather promising (we achieved a maximum speedup of 6.7 with 17 processors), on the other hand, they also show the limits of our implementation: $\mathcal{DF}$ is built upon a shared structure space, while the transputer is based on a distributed memory model. For parallel algorithms where the amount of inter-process communication is low (or can be artificially decreased by caching, as in the `primes` program), this difference is not so important. However, if processes are very tightly coupled (as in the `quicksort` program), communication between processors becomes a significant bottleneck.

5 Conclusions

The $\mathcal{DF}$ model is only a first step in the parallel implementation of a functional language. Starting from functional specifications, we are able to generate $\mathcal{DF}$ programs that can be efficiently executed on a transputer system (we will soon port the C-$\mathcal{DF}$ model to a shared memory machine, too). However, it still remains to prove that this tranlation can be performed in an automatic way. Thus, our future work will concentrate on the development of a parallelizing compiler. Different from other approaches that create many fine-grained tasks, we will try to detect the main source of parallelism in the program in order to spawn only few coarse-grained processes. We believe that most algorithms are based on a few parallelization principles and that a compiler can detect these principles in concrete programs. The application of functional languages for parallel programming should then become feasible.

References

[1] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-Structures: Data Structures for Parallel Computing. Computation Structures Group Memo 269, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, February 1987.

[2] D.I. Bevan. Distributed Garbage Collection Using Reference Counting. In *PARLE, Parallel Architectures and Languages Europe, Volume 2: Parallel Languages*, pages 176–187, Eindhoven, The Netherlands, June 15–19, 1987. Volume 259 of Lecture Notes in Computer Science, Springer, Berlin.

[3] Inmos Limited, Bristol, UK. *The Transputer Databook*, November 1988.

[4] Logical Systems, Corvallis, Oregon. *Transputer Toolset Version 89.1*, January 1990.

[5] Wolfgang Schreiner. ADAM & EVE — An Abstract Dataflow Machine and Its Programming Language, Diploma Thesis, Johannes Kepler University, Linz, Austria, October 1990.