Organizing Speculative Computations in Functional Systems

Raimund Schroeder and Werner Kluge
Department of Computer Science
University of Kiel
D–24098 Kiel, Germany
E–mail: wk@informatik.uni-kiel.de

Abstract. *Speculative evaluation relates to computing several (alternative) threads of control of large programs concurrently without knowing in advance which of them contribute to which extent to final results. This approach may be used to advantage to compute, at the expense of deploying considerable processing power, solutions of np–hard search problems on average a lot faster than sequentially.*
This paper addresses the organizational measures necessary to perform speculative computations concurrently in a distributed memory multiprocessor system. They primarily concern task management and scheduling, a fairness regulation scheme which ensures progress of all speculative tasks at about the same pace, and the conflict between fairness and bounded numbers of speculative tasks. Though these measures are discussed in the context of functional languages and systems, they are in principle applicable in the imperative world as well.

1 Introduction

Programs of functional languages are known to be perfectly suited for concurrent processing. Conceptually, program execution is a process of meaning–preserving program transformations based on a set of rewrite rules which for all semantically meaningful programs eventually terminates with a result which is itself a program. Since all rewrite rules perform context–free substitutions of equals by equals, creating no side effects elsewhere in the program, they may be applied in any order without affecting the determinacy of results.

Of the various concepts of executing functional (or function–based) programs concurrently, speculative evaluation is the most challenging one from an organizational point of view. However, at the expense of committing considerable resources, it may also be the least rewarding one in terms of performance gains. The idea is to evaluate several sub–terms of a functional program concurrently without having, at the time the respective tasks or threads of control are being created, sufficient information at hand to decide which of them may contribute to which extent to the normal form of the entire program eventually.

Under a lazy regime, speculative evaluation is often employed to evaluate some or all function arguments in advance and possibly beyond the point actually needed, provided sufficient processing power can be made available that would otherwise be idling (though this so–called eager–beaver approach runs somewhat counter to the idea of laziness since more than absolutely necessary is usually done to compute normal forms) [Jon87, Par91, Mat93, Che94].

This paper is on the speculative evaluation of sets of rewrite rules, specified as pattern matching clauses embedded in `case` (or switch) constructs. Pattern matching, generally speaking, is to abstract specific (sub–)terms from given structural contexts (the argument terms to which the `case`s are applied) and to substitute them into specific syntactical positions of other contexts (the body terms of the matching clauses). As several patterns of a `case` may be overlapping, they may produce as many matches on given arguments. Under a purely functional interpretation, the patterns are applied in the order in which they are specified in the `case`, and the first matching clause is the one that is being picked as the result of the entire `case` application. In compliance with this execution order, the clauses are usually arranged so that the patterns covering special argument features precede those that cover the more general (structural) features.

However, rule based applications such as term rewriting (logic reasoning) or many search problems are typically of a nature where several clauses feature overlapping patterns which can not be given a unique ordering. If more than one of these patterns matches a particular argument, the respective clauses may have to be evaluated speculatively since sufficient information as to which of them will lead to some desired result (problem solution) eventually may become available only further down the road as more pattern clauses (rules) are being applied. Unfortunately, such trial–and–error computations generally feature an exponential complexity which renders them intractable for large problem sizes, particularly when doing them sequentially. However, evaluating all matching clauses of `case` applications concurrently on a speculative basis may considerably improve the chances of computing solutions of such problems decidedly faster, a generous supply of processing power provided.

Of primary concern in this paper are the organizational measures necessary to support in some orderly form this kind of speculative concurrency in a multiprocessor system both effectively and efficiently. To do so, the system must

– distinguish between vital tasks (or threads of control) whose results are bound to contribute to problem solutions and speculative tasks (threads) whose results may or may not be required;
– treat all vital tasks with higher priority than speculative tasks, and possibly distinguish several priority levels among the latter, to prevent the monopolization of the system with computations of which most are known to be superfluous;
– abort speculative tasks or lift their status to vital as soon as decisions to this effect can be made;
– apply a fair scheduling discipline to all speculative tasks originating from the same `case` application to ensure that all of them proceed at about the same pace as long as no clues are available as to which of them have the best chances of succeeding; otherwise too many resources may be committed to

just the wrong computations while those that produce useful solutions are left starving;

– strike an acceptable compromize between limitation of resources, specifically processing power, on the one hand and fair progress of potentially unbounded numbers of speculative tasks on the other hand.

The paper discusses conceptual solutions for these organizational problems which have been successfully implemented as extensions of an existing concurrent graph reduction system π-RED [Klu83, BHK94, GK96]. This implementation has been extensively tested and validated by means of a parameterizable program which simulates searches of a spider for exits in a maze.

The paper is organized as follows: the next section introduces some essential language constructs, section 3 outlines the basic principles of organizing speculative computations, section 4 describes the measures that sustain reasonable fairness while limiting the number of speculative tasks, section 5 reports on some performance results, and section 6 discusses some related work.

2 The language

For the purpose of this paper it suffices to consider a simple dynamically typed and strict functional kernel language with a reduction semantics [Berk75, Klu94], i.e., program execution is governed by a set of term rewrite rules. The program terms are recursively constructed as follows:

$$
\begin{array}{rl}
e = & const \mid var \mid prim\_fun \\
 & \mid (\, e\_0\, e\_1 \,\ldots\, e\_n \,) \\
 & \mid \text{IF}\; e\_0 \;\text{THEN}\; e\_1 \;\text{ELSE}\; e\_2 \\
 & \mid \text{LET}\; u\_1 = e\_1 \ldots u\_n = e\_n \;\text{IN}\; e\_0 \\
 & \mid \;<\; e\_1, \ldots, e\_n \;> \\
 & \mid \text{DEFINE}\; \ldots, f\; u\_1 \ldots u\_n = e\_f, \ldots \;\text{IN}\; e\_0 \quad . \\
 & \mid \text{CASE}\; pat\_1 \to e\_1,\; \ldots,\; pat\_i \to e\_i,\; \ldots, pat\_n \to e\_n \;\; \text{END\_CASE} \\
pat = & const \mid var \mid \_ * pat\_ * \mid \; < \; \ldots,\; pat\_k,\; \ldots \; >
\end{array}
$$

These terms denote, from top to bottom, constant values, variables, primitive functions such as $+$, $-$, $\ldots gt$, le ... etc., applications of terms e_0 in function position to n argument terms e_1 , ... , e_n, IF_THEN_ELSE clauses, LET terms, $n-$ tuples (n–ary lists) of terms, and sets of mutually recursive function definitions, with $f, u\_1, \ldots, u\_n$ and e_f respectively denoting a function identifier, the formal parameters of the function, and the function body term (which computes the function value). The term e_0, which may call upon any of the functions defined, computes the value of the entire DEFINE construct.

The terms of primary interest in this paper are CASE–constructs of some n pattern matching clauses $pat\_i \to e_i$ which may be used to define sets of rewrite rules for tuple terms. The patterns may be composed of constants, variables, wild card patterns (denotes as $\_ * pat * \_$), and tuples of these items, including recursively tuples as tuple components.

A tuple pattern *pat* is said to match an argument term if it has an identical tuple structure, each pattern constant literally equals a constant value in an identical tuple position of the argument, and each pattern variable matches a (sub–)structure in an identical argument position[1]. This being the case, each occurrence of a pattern variable in the body term e_i is substituted by the respective argument component, and the term thus instantiated is evaluated.

The entire CASE–construct in fact specifies a complex unary function. When applied to an argument value, a strictly functional interpretation requires that all clauses be tried in the order from left to right, and the value of the first clause whose pattern matches be returned as function value. This evaluation order guarantees determinacy of results even in the presence of several (potentially) matching patterns.

If none of the patterns matches, the CASE application may simply return itself as its own value since it can obviously not be re-written into anything else Alternatively, the application may be considered undefined and be replaced by the bottom symbol $\perp$.

Thus, the meaning (or the semantics) of a CASE application may be defined by an evaluator function EVAL as:

$$\text{EVAL}[\ (\ \text{CASE}\ \ldots,\ pat\_i \rightarrow e\_i, \ldots\ \text{END\_CASE}\ e\_a\ )\ ]$$

$$= \begin{cases} \text{EVAL}[\ e\_i[\Leftarrow]\ ] \text{ if MATCH } (pat\_i,\ e\_a) = \text{SUCC} \\ \qquad\qquad\quad \text{and } (\forall\, j \in\ \{1, \ldots, n\})\ \text{MATCH } (pat\_j,\ e\_a) = \text{FAIL} \\ \\ \qquad\qquad\quad (\ \text{CASE}\ \ldots,\ pat\_i \rightarrow e\_i, \ldots\ \text{END\_CASE}\ e\_a\ ) \\ \qquad\qquad\quad (\text{or}\ \ \perp\ )\ \text{ otherwise} \end{cases}$$

where MATCH(pat_i, e_a) returns SUCC (for succeed) if pat_i matches the argument term e_a, and FAIL otherwise[2]; $e\_i[\Leftarrow]$ denotes the instantiation of occurrences of the pattern variables in the body term e_i by the (sub-)terms extracted from the matching positions of the argument term e_a.

However, there are many interesting applications, specifically search problems, where clauses (rewrite rules) with overlapping patterns cannot be given a unique ordering with respect to a best possible choice of a solution. There may be several promising alternatives to pursue, and the choice may have to be made further down the road as more information to this effect becomes available, say by repeated application of the same or other CASEs, and some of the alternatives can safely be discarded (e.g., those that produce a FAIL). Until such choices can

[1] Trivial constant patterns match identical constant arguments, and trivial variable patterns match all legitimate argument terms. Wild card patterns $\_ * pat * \_$ may only occur as components of tuple patterns. They match sequences of $geq1$ tuple components in the arguments, of which one must match the pattern *pat*. If the wild card _ preceding or succeeding *pat* is missing, then *pat* must match the first or the last component, respectively, of the matching sequence.

[2] Note that both SUCC and FAIL are not elements of the functional language proper but are values of the function MATCH used by the evaluator EVAL.

be made, evaluating matching clauses remains inevitably speculative as it is not a priori known which of them will fail or succeed eventually. However, to speed things up, several or all matching clauses may be computed concurrently since, in a functional setting, they do not inflict side effects on each other.

Denoting a CASE construct as S_CASE if it is to be evaluated speculatively, the meaning of an S_CASE application may be defined as:

$$\mathrm{EVAL}[\ (\ \mathrm{S\_CASE}\ \ldots,\ pat\_i \to e\_i, \ldots\ \mathrm{END\_CASE}\ e\_a\ )\ ]$$

$$= \begin{cases} (\ \mathrm{S\_CASE}\ \ldots,\ pat\_i \to e\_i, \ldots\ \mathrm{END\_CASE}\ e\_a\ ) \quad (\text{or}\ \perp) \\ \quad \text{if}\ \forall\, i \in \{1, \ldots, n\}\ \mathrm{MATCH}\ (pat\_i,\ e\_a) = \mathrm{FAIL} \\ \\ \{\ \mathrm{EVAL}\ [\ e\_i[\Leftarrow]\ ]\ |\ \mathrm{MATCH}\ (pat\_i,\ e\_a) = \mathrm{SUCC}\ \}\ \ \text{otherwise} \end{cases}$$

i.e., the value of such an application, again, is the application itself (or the bottom symbol $\perp$) if none of the patterns matches, and the set of values of the instantiated body terms of all clauses whose patterns match the argument value e_a otherwise.

Based in this definition, it may be further qualified what a (best possible) problem solution (result) that is to be chosen from this set should be. There are basically three options available:

- One could be content with just one of possibly several results, e.g., the one returned first. However, as such result may depend on execution orders chosen by the underlying system, it is generally non–determinate and thus violates the functional semantics.
- To guarantee determinacy, the criterion for selecting a single out of several possible results has to be made dependent upon an algorithmic or application–specific property. Such a criterion could be the least number of rule applications performed to arrive at a result, and if there is more than one result that meets it, then one could pick the leftmost of the particular S_CASE.
- Another alternative would be to ask for the full set of solutions. Unfortunately, it cannot generally be decided by the system whether this set can be computed at all since the computation may not terminate in all branches. However, the user could specify an upper bound on the number of rule applications to be executed in each speculative branch, and accept as a solution the subset of results that can be computed within this limit.

Selecting one of these options may be specified by means of distinct key words N_CASE (for non–determinate single solutions) D_CASE (for determinate single solutions) and M_CASE (for multiple solutions) which control appropriate interpretation or compilation to machine code.

The need to count rule applications in the latter two cases goes hand in hand with the need to halt in some orderly form runaway computations effected, say, by recursive function calls that fail to meet termination conditions. This may be accomplished by a system-supported count variable which prior to each program run is initialized with some user-specified integer value. Upon each application of a rewrite rule this count value is decremented by one. The computation is halted either if the program term has reached normal form or if the count value is down to zero, whichever occurs first [Berk75, GK96].

This count variable may be smoothly integrated into the definition of the evaluator EVAL, which for S_CASEs then takes the form

$$\text{EVAL}[\ (\ \text{S\_CASE}\ \ldots,\ pat\_i \rightarrow e\_i, \ldots\ \text{END\_CASE}\ e\_a\ )\mid k\ ]$$

$$= \begin{cases} (\ \text{S\_CASE}\ \ldots,\ pat\_i \rightarrow e\_i, \ldots\ \text{END\_CASE}\ e\_a\ )\mid k \quad (\text{or}\ \perp) \\ \quad \text{if}\ \forall\, i \in \{1, \ldots, n\}\ \text{MATCH}\ (pat\_i,\ e\_a) = \text{FAIL} \\ \quad \text{or if}\ \ k = 0; \\ \\ \{\ \text{EVAL}\ [\ e\_i[\Leftarrow] \mid k-1\ ] \mid \text{MATCH}\ (pat\_i,\ e\_a) = \text{SUCC}\ \}\ \ \text{otherwise} \end{cases}$$

Assuming that the count variable has value k upon entering the evaluation of an S_CASE application, it remains unchanged (or is replaced by the bottom symbol $\perp$ [3]) if none of the patterns matches the argument or k is down to zero at this point, but is decremented by one if at least one pattern matches, whereupon the body terms of all matching clauses continue computing with their own copies of the count value $k-1$.
Using these count values, the value (or normal form) of

- a D_CASE application can be defined as the value of the clause selected by a successful pattern match whose associated count variable k has the largest remaining value (and thus evaluated the least number of applications);
- an M_CASE application can be defined as the set values of those clauses selected by successful pattern matches whose associated count variables have values $k > 0$.

3 Controlling speculative concurrent computations

Executing functional programs concurrently is usually based on a simple divide–and–conquer scheme which recursively spawns new tasks for program terms whose values are bound to contribute to results. The tasks at the leaves of the emerging task tree may be scheduled for processing in any order and non–preemptively since all of them are vital (or mandatory). There are also fairly simple measures at hand to prevent the creation of tasks far beyond the number of processing sites [Klu83].

Things become decidedly more complicated when speculative tasks enter the game. To commit processing capacity with top priority to useful computations, vital tasks must be given preference over speculative tasks, i.e., no speculative

[3] Note that the bottom symbol is used in the actual system implementation to easily identify speculoative computations that have failed to produce a (partial) problem solution and therefore can be aborted.

task may be scheduled for processing as long as there are vital tasks ready to run. Moreover, vital tasks, once running, may yield processing sites only if they terminate or suspend themselves. Scheduling speculative tasks on the processing sites that are left over must be preemptive to prevent the monopolization of the systems resources with computations that may turn out to be useless. Processing sites may have to be turned over as soon as possible to newly emerging vital tasks, and all speculative tasks ought to be moved ahead at about the same pace as long as it cannot be decided which of them are going to succeed. If there are more speculative tasks than processing sites they can run on, they must inevitably be scheduled in a round robin fashion similar to time slicing to enforce fair progress (which in fact realizes a breadth–first evaluation strategy).

A suitable fairness regulation mechanism may be based on the notion of synchronic distances as introduced in Petri net theory [GLT80], in combination with some loose form of barrier synchronization. Synchronic distances, roughly speaking, define upper bounds on the number of operational steps by which one of several competing tasks (or threads of control) can at most get ahead of the others before they must catch up. These steps can easily be counted in terms of rewrite rules performed, which nicely blends in with the count variable k introduced in the preceding section as part of the definition of the evaluator EVAL, whose primary purpose is to halt runaway computations.

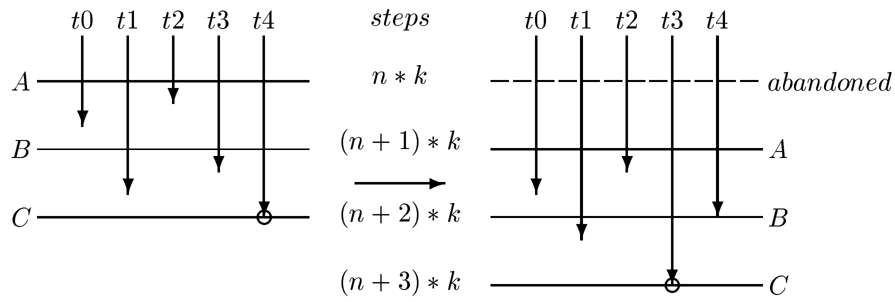

Fig. 1. Fairness regulation scheme with three–barrier synchronization

The fairness mechanism which, after experimenting with several alternative solutions, has emerged as the most effective one for supporting speculative computations in a distributed implementation of π-RED involves three successive barriers A, B and C which are synchronic distances of some k rewrite steps away from each other (see fig. 1). Assuming that some n speculative tasks (there are five named $t0 \ldots t4$ shown in the figure) participating in the race have just managed to cross barrier A, they are supplied with a fuel of some k rewrite steps to reach barrier B. Tasks arriving at barrier B (and thus having exhausted their fuel) may cross and be refueled with another k rewrite steps to move on towards barrier C. There they are blocked until all n tasks have crossed barrier

B (the left of fig. 1). This being the case, the three barriers are moved k steps ahead, i.e., the current barrier A is abandoned, barriers B and C become the new barriers A and B, respectively, and a new barrier C is set up k steps down the road from the current one (the right of fig. 1), and so on.

This three–barrier mechanism keeps many of the tasks alive and computing most of the time, particularly if the number of tasks only marginally exceeds the number of available processing sites. As it must exercise control over all speculative tasks across all processing sites of a distributed system, it may be realized as an attachment to the root task of the program, which for this purpose must be kept vital and running in some unique site. The root task maintains a data structure which contains pointers to the context blocks of all speculative tasks participating in the fairness regulation. This structure is updated upon receiving control messages pertaining to the spawning of new speculative tasks and to tasks changing their status from speculative to vital, terminated or aborted. Information about barriers reached (and possibly blocked) or crossed by individual tasks is held in the respective context blocks and updated whenever signals to this effect arrive.

Another sophisticated mechanism concerns the creation of speculative tasks, their registration as new parties of the fairness mechanism, and actually starting them. A vital or speculative task that executes, say, a CASE–application may become a parent task that creates a speculative child task for every matching clause, generally in some other processing site, whereupon the parent usually suspends itself since there may be little else to do but to wait for results returned by the children. The children, in turn, have to be registered with the fairness regulation mechanism and supplied with fuel to start running. The communications necessary to do so are depicted in fig. 2. After having created in sequence speculative tasks for all matching clauses (two in the example), the parent continues until it receives acknowledge messages from the new children, together with their identities. The children are then immediately suspended until further notice. The parent registers the new children with the fairness regulation mechanism and suspends itself [4]. The fairness regulation, in turn, confirms the registration and sends out start signals, together with the fuel left by the parent to reach the next barrier.

This rather complex scheme is necessary to catch asynchronous messages which may interleave with these communications. For instance, while the parent is in the process of spawning children or registering them with the fairness regulation, an incoming message might signal the parent that the next barrier has been opened and more fuel is available, which may be immediately passed on to the children.

Termination or abortion of speculative tasks depends to some extent on the type of the desired result. As mentioned in section 2 this could be just any one of possibly several problem solutions, e.g., the one the system returns first, a unique single solution, say, the one that requires the least number of rewrite steps to

[4] If the parent is a speculative task itself, it temporarily resigns from the fairness regulation as well.

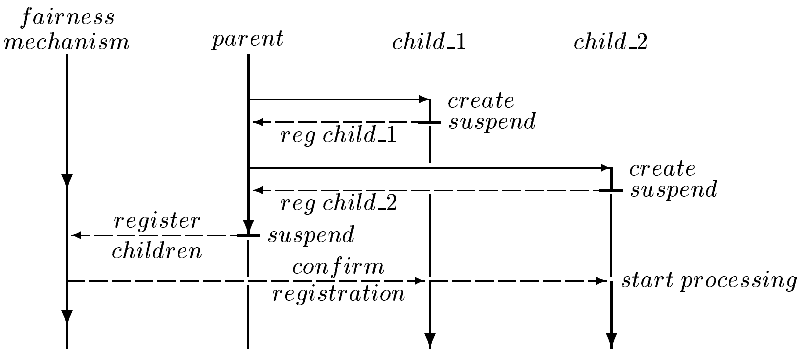

Fig. 2. Spawning new speculative tasks

arrive at, or all possible solutions requiring less than some pre–specified number of rewrite steps – the total fuel – to compute [5]. In both of the latter cases, the computations would produce determinate results irrespective of actual execution orders. In the former case, the result would be non–determinate, or dependent on actual task scheduling.

A speculative task may be aborted if

- by some criteria, typically a CASE–application with no matching clauses, it produces the bot symbol $\perp$ which indicates failure of the particular clause to produce a problem solution (compare section 2);
- it receives a signal that a solution has been produced somewhere else and the task has thus become irrelevant.

In either case the task immediately resigns from the fairness regulation, detaches itself from its parent, and releases its context block.

A task terminates regularly if it produces a solution. All other tasks

- may be aborted if just any one solution is required;
- that have already consumed more rewrite steps are signaled to abort at the next barrier if the desired solution must have consumed the least number of rewrite steps;

When computing all solutions up to the point of exhausting the total fuel, all tasks that are still alive at this point must be aborted.

However, as results must be returned to the root task, the tasks that are on the path from a terminating task to the root must, of course, in all three cases stay alive until the results have been passed through.

[5] Without such an upper bound the computation could get trapped in endless recursions and thus never terminate.

4 Fairness versus limited resources

There is a fundamental conflict of interest between the demand for fair progress of all speculative tasks on the one hand and finite resources on the other hand.

With no information at hand to set priorities, all opportunities for speculative evaluation ought to be treated on an equal basis, i.e., driven ahead at about the same pace, as otherwise too much processing power might be wasted on just the wrong computations. Given a recursively unfolding search problem, a breadth–first search would require spawning unbounded numbers of speculative tasks as there is generally no upper limit on the size of the search space. However, spawning tasks far beyond the number of processing sites would not only create considerable management overhead without any further performance gains but also consume decidedly more memory for runtime structures and heap space than, say, a strictly sequential search at the other extreme.

To overcome this problem, a compromize must be made between unbounded demands for resources on the one hand and enforcing strict fairness among speculative computations as outlined in the preceding section on the other hand. The idea is to provide a pool of concessions (or tokens) – in a distributed system evenly spread out over its processing sites – to spawn new tasks [Klu83]. Whenever such an opportunity arises, the (local) pool is checked for the availability of a concession, and if so, the concession is taken and a new task is created; otherwise this opportunity is ignored and the sub–term that was to be evaluated by the new task is instead processed sequentially by the calling task. A terminating task recycles to the (same local) pool the concession it has taken, which may be immediately picked up to create another task. Thus, at any time there are at most as many tasks alive in the system as there were initially concession tokens in the pool.

Ignoring opportunities to spawn new tasks in fact implies switching from a breadth–first to a depth–first execution mode. If this approach would be applied to speculative tasks, fairness would clearly be violated: several matching clauses of a CASE application could only be evaluated in sequence, say from left to right, and depth–first, i.e., further CASE applications deeper down would recursively

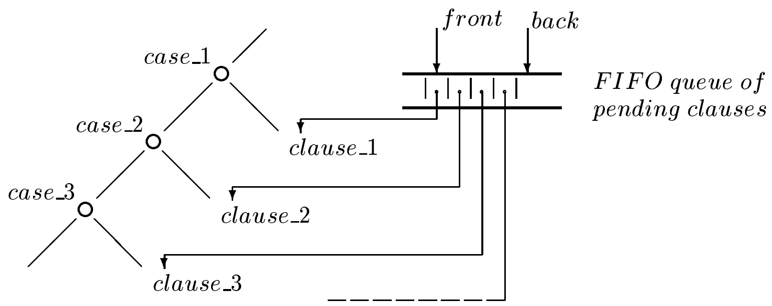

Fig. 3. Building a FIFO queue of pending clauses

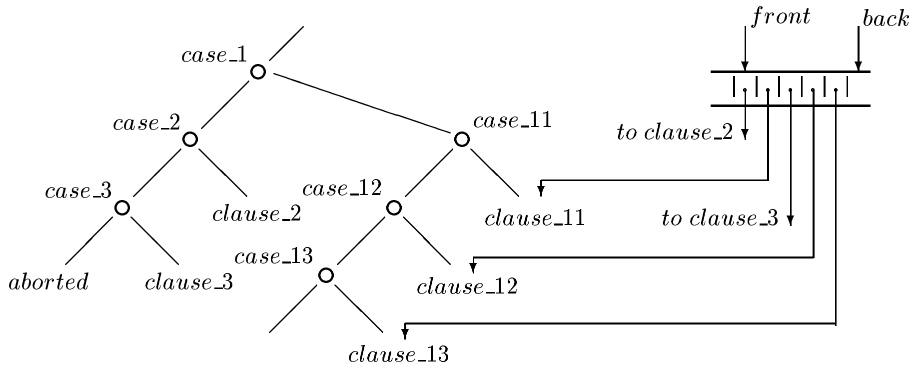

Fig. 4. Keeping pending clauses in the queue in descending order

be treated in the same way.

The way out of this dilemma consists in placing, on the way down, into a task–specific *FIFO*–queue all matching clauses of a CASE application that for the time being must be left pending, as indicated in fig. 3 for a recursive nesting of three CASEs that produce just two matches each. Whenever a task produces a bottom symbol $\perp$ in the leftmost branchs, indicating that the search for a solution has failed, e.g., at a CASE application with no matching pattern, it continues with the first alternative at the front–end of the queue which is the one highest up in the unfolding computational tree. Likewise, if concessions to spawn new tasks become available again later on, the queues of all existing tasks are inspected and the pending clauses highest up across all tasks residing in a particular site are chosen for the creation of new tasks.

Since a task, after having produced a bottom symbol in its leftmost branch, is supposed to backtrack to the topmost alternative clause and this clause is to be taken from the front–end of the queue, the queue cannot strictly be operated in *FIFO* order. As illustrated in fig. 4, the pending clauses of the new branch must be sorted into the existing queue so that clauses of higher levels always precede clauses of lower levels of the computational tree to keep the computation breadth–first as much as possible.

This strategy does depth–first evaluation whenever, after exhaustion of all concessions to spawn speculative tasks, there is no alternative left, but gives preference to breadth–first evaluation as soon as concessions are (or become again) available, i.e., it is reasonably fair without involving elaborate priority schemes, given the constraints of limited numbers of tasks.

5 Some performance experiments with a search problem

The concepts described in the preceding sections have been implemented as extensions of a concurrent graph reduction system π–RED [GK96, BHK94]. This system interprets abstract machine code to which high–level programs of the functional language KIR [Klu94] (whose syntax closely resembles the one given in section 2) are compiled. Other than identifying S_ CASE–constructs as candidates for speculative evaluation, these programs do not contain any annotations pertaining to concurrent execution. π–RED is installed on an nCUBE/2 distributed multiprocessor system of 32 processing sites which can be configured to form sub–cubes of some $2^k \mid k \leq 5$ sites. Each site runs a micro kernel nCX which supports a single π–RED process. This process in fact realizes a small kernel of its own which handles π–RED–specific task management and communication, including signals and messages from and to the fairness mechanism.

Program code is downloaded from some front–end workstation into each nCUBE/2 processing site; program execution starts with a single task at a unique initial site (which also runs the fairness mechanism) from where it recursively spreads out over all sites of the particular configuration. The initial task in that site eventually also assembles the result of the computation and returns it to the front–end. As of now, the system interprets compiled abstract machine code as it was easier at this level to implement, modify and experiment with the pieces of code that control the creation, synchronization and termination of speculative tasks, communicate with the fairness regulation mechanism, effects status changes, etc.. For programs which heavily use pattern matching, code interpretation is about three times slower than compiled target machine code.

To systematically investigate performance enhancements due to speculative evaluation relative to sequential program execution, we have chosen a program which searches for exits in a maze. Though this problem can be elegantly and efficiently solved by encoding the maze into a set of higher–order functions, of which an application of the function that represents the starting point produces the desired result, we have deliberately chosen a decidedly more primitive algorithm. It has the advantage of creating by relatively simple means and in a controlled way sufficient computational complexity and opportunities for speculative computations.

The search is simulated by a spider walking step by step through the maze, represented as a matrix which marks walls as 1s and the aisles in between as 0s, and checking at each position, by means of pattern matches, how many alternatives are available to make a next step. If there is more than one direction to follow, the computation splits up into as many speculative searches. The search along a particular path succeeds if an exit can be reached and fails if, at a particular position, there is no alternative left but to move backward. Each search path is kept track of by generating a list (tuple) of coordinate pairs (positions) where the spider changed directions or branched out into two or three different directions. The paths that led to exits are returned as output and those that led into dead ends are discarded. The program can be parameterized with respect to size and layout (complexity) of the maze, number and positions of exits, starting position and width of the steps taken by the spider.

The program basically centers around a function *explore* as shown in fig. 5 which, given a certain position of the spider in the maze in terms of its coordinates x, y, moves one step ahead in the direction specified by the parameter h

(which may assume one of the values $'east'$, $'west'$, $'north'$, $'south'$) and finds out where to move next from there.

```
explore x y h =
   LET xx = (new_x x h),
       yy = (new_y y h),
   IN IF(is_exit xx yy)
       THEN << xx, yy >>
       ELSE LET hs = (scan xx yy h)
       IN ...
      ( S_CASE
          < _* 'east' _* > → (path < xx, yy > (explore xx yy 'east'))
          < _* 'west' _* > → (path < xx, yy > (explore xx yy 'west'))
          < _* 'north' _* > → (path < xx, yy > (explore xx yy 'north'))
          < _* 'south' _* > → (path < xx, yy > (explore xx yy 'south'))
       END_CASE hs )
```

Fig. 5. Code for the function *explore* that controls the movement of the spider

From top to bottom, this function computes, by means of the functions new_x and new_y, the new pair of coordinates xx, yy, checks whether this new position is an exit (in which case the function terminates by returning the pair of coordinates), and, if not, computes by means of the function *scan* the directions in which the spider may move next, and collects these directions in a tuple returned as function value which instantiates the LET bound variable hs. This tuple is then taken as an argument of an S_CASE function whose patterns [6] check it for occurrences of the four possible directions. In case of a match (of which there may be up to three since the spider is not allowed to move backward), the coordinate pair $< xx,\ yy >$ is appended to the list(s) of travel positions computed so far and prepended to the list of positions still to be computed by the recursive application of the function *explore* to these coordinates and to the direction identified by the pattern match.

The first problem investigated was the search for a single exit in a maze featuring the regular shape of a balanced binary tree of 128 leaves at the base (the left of fig. 6 shows a tree of eight leaves). The search starts at the root of the tree, and terminates successfully at a leaf which is selected as the single exit. This exit position is moved from left to right along the leaves. The program execution times on the right of this figure confirm what can be expected: when doing a depth–first and left–to–right sequential search on a single processing site, the runtimes increase linearly from the exit being in the leftmost to being in the rightmost position. With speculative searches, runtimes drop considerably with increasing numbers of processing sites and become invariant against changing

[6] The patterns are specified by means of wild cards since the particular direction may be in any of at most three tuple positions.

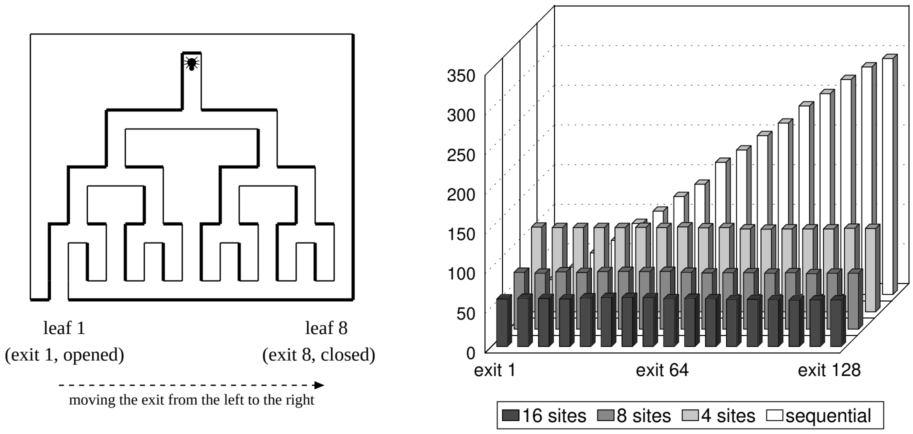

Fig. 6. Searching for an exit in a maze shaped as a binary tree

exit positions as the tree is well balanced. The speculative search on 16 processing sites clearly outperforms the sequential search for all but the 8 leftmost of the 128 possible exit positions. Thus, at the expense of involving considerable processing capacity, one may win in some (most) situations but lose in others.

Similar experiments have been conducted with several square–shaped mazes of highly irregular internal structures of aisles and walls, with exits placed on each of the four borders, and with different lengths of the search paths leading to these exits. A typical such maze is shown on the left of fig. 7. The searches were done

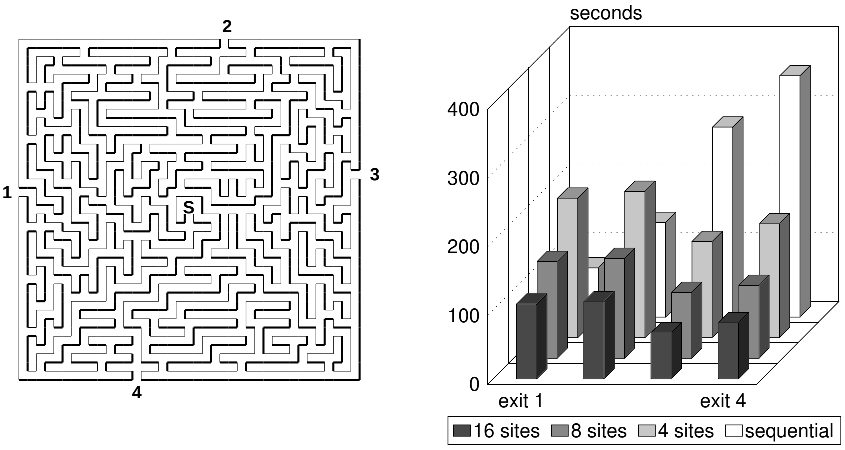

Fig. 7. Searching for exits in an arbitrary maze

with only one of the four or with all exits open, and with the paths to one or all four exits as results. Though performance figures to some extent depend on the particularities of the maze structure and also on the machine configuration (e.g., on the chosen synchronic distance of the fairness mechanism or on upper bounds on the number of speculative tasks), it again turned out that for all mazes and system parameters investigated the speculative searches on 16 processing sites did decidedly better than all but the best case sequential searches, and that with 8 or 4 processing sites the speculative searches outperformed at least the worst case sequential searches. Representative for these results are the execution times on the right of fig. 7.

An interesting question concerns the overhead inflicted by the organizational measures necessary to support speculative evaluation as described. This question is difficult to answer quantitatively since it would require appropriate instrumentation of the runtime system and profiling of several application programs with different system configurations (numbers of processing sites, number of concessions to spawn speculative tasks, synchronice distances of the fairness mechanism, etc.), which we simply did not have the resources to do. Comparing wallclock execution times with speculative evaluation activated and de-activated would not be very conclusive as they relate to different modes of program execution with completely different dynamic program behavior. However, looking at the performance diagrams of figs. 6 and 7, which compare speculative evaluation of the same program on different numbers of processing sites, it can at least be concluded that the overhead must be quite substantial (i.e., in the order of 30% to 50%): doubling the number of processing sites (from 4 to 8 and then to 16) reduces program execution times merely by about a factor of 2/3.

Conclusive wallclock measurements were only possible with respect to the effect on program execution times of varying synchronic distances enforced by the fairness regulation mechanism. Representative are the data collected from a maze program that takes roughly 350.000 rule applications to find a single exit. It was run on a system configuration with 8 processing sites and unbounded numbers of speculative tasks. Changing the synchronic distance from 5000 to 50.000 rule applications (in increments of 5000) led to a nearly linear reduction of control messages from some 260 to 24 (i.e., by about an order of magnitude) and of the program runtime by about 10%.

6 Related work

In the functional domain, speculative evaluation primarily relates to non–strict arguments under a lazy regime, with HASKELL as the language of choice [Par91, Mat93]. In [Che94] are described several priority schemes for speculative tasks, up– and down–grading, and both explicit and implicit abortion of tasks that have become irrelevant. This work also addresses the problem of non–determinacy and suggests introducing the notion of bags in which solutions are collected without any ordering to maintain a functional semantics.

Another interesting approach based on the speculative evaluation of LISP programs is reported in [Osb90]. It introduces a rather elaborate priority scheme based on a so–called sponsor model which favors the execution of speculative tasks with the best chances of succeeding. Sponsorships and, hence, priorities are dynamically adjusted as computations proceed and more information about possible outcomes become available. The sponsor model implicitely also gets around the fairness vs. limited resources problem.

In logic programming, the concepts closest to our approach relate to the exploitation of the so–called OR–parallelism in PROLOG–programs [Sha89], for which scheduling strategies are described in [Bea91]. Interestingly enough, speculative evaluation in PROLOG relates to computations that may be cut off by pruning operations such as the famous `cut`. Detecting this kind of speculative work is discussed in [Hau89].

Representative for related work on term–rewrite systems is [VK96]. It describes an interactive theorem prover called LARCH which allows the user to launch several speculative attempts to prove conjectures; unsuccessful attempts must be cleaned up by hand. Both AND– and OR parallelism are supported, and the number of competing tasks or the depth of the recursive expansion can be restricted.

7 Conclusion

The system concept described in this paper merely supports the measures that are considered absolutely essential to organize speculative computations in some orderly form. They mainly concern a reasonably efficient usage of resources (by giving mandatory tasks scheduling priority over speculative tasks and by throttling the number of speculative tasks that may participate in a computation), ensuring fair progress among all speculative tasks, and earliest possible status changes of speculative tasks either to aborted or to vital. The emphasis of this work was primarily on feasibility studies of various alternative solutions to these problems, less so on absolute performance figures (hence interpretation of abstract machine code which facilitated experimental implementations considerably). No attempts have been made to speculate, e.g., by means of more or less elaborate priority schemes, on the likely behavior of the program at hand, and to have task scheduling governed by these priorities, as for instance in the approaches reported in [Che94] and in [Osb90]. Concentrating on the essentials has nevertheless yielded encouraging results in terms of noticeable performance gains vs. sequential execution.

Further work primarily concerns more refined scheduling strategies for speculative tasks which take into account several dynamically changing priority levels, improved mechanisms for passing control messages up and down hierarchies of speculative tasks, and compilation to target machine code.

References

[Bea91] A. Beaumont. Scheduling Strategies and Speculative Work. In Anthony Beaumont and Gopal Gupta, editors, *Proceedings of Parallel Execution of Logic*

Programs (ICLP '91), volume 569 of *LNCS*, pages 120–131, Berlin, Germany, June 1991. Springer.

[Berk75] K.J. Berkling Reduction Languages for Reduction Machines *Proceedings 2nd Annual International Symposium on Computer Architecture*, ACM/IEEE 75CH0916-7C 1975, pages 133–140,

[BHK94] T. Bülck and A. Held and W.E. Kluge and S. Pantke and C. Rathsack and S.-B. Scholz and R. Schröder. Experience with the Implementation of a Concurrent Graph Reduction System on an nCUBE/2 Platform. In B. Buchberger and j. Volkert, editors, *Proceedings of CONPAR 94 - VAPP IV* volume 854 of *LNCS*, Springer 1994, pages 497 – 508,

[BK92] H. Bloedorn and W.E. Kluge. Organizing Speculative Computations in the Reduction System π-RED*. In *Proc. Workshop Parallel–Algorithmen und Rechnerstrukturen*, pages 138–152. Mitteilungen der GI ISSN 0177–0454, 1992.

[Che94] Iain G. Checkland. *Speculative Concurrent Evaluation in a Lazy Functional Language.* PhD thesis, University of York, Department of Computer Science, 1994.

[GK96] D. Gärtner and W.E. Kluge. π-RED$^+$—An Interactive Compiling Graph Reduction System for an Applied λ-Calculus. *Journal of Functional Programming*, 6(5):723–756, 1996.

[GLT80] H.J. Genrich and K. Lautenbach and P.S. Thiagarajan. Elements of General Net Theory In *Net Theory and Applications*, volume 84 of *LNCS*, Springer 1980, pages 21–163,

[Hau89] B. Hausman. Pruning and Scheduling Speculative Work in OR-Parallel Prolog. In E. Odijk, M. Rem, and J.-C. Syre, editors, *Proceedings of the Conference on Parallel Architectures and Languages Europe : Vol. 2*, volume 366 of *LNCS*, Springer 1989, pages 133–150

[Jon87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Computer Science. Prentice-Hall, 1987.

[Klu83] W.E. Kluge. Cooperating Reduction Machines. *IEEE Transactions on Computers*, C-32:1002–1012, 1983.

[Klu94] W.E. Kluge. A User's Guide for the Reduction System π–RED . Technical Report 9419, Institut für Informatik und praktische Mathematik, Universität Kiel, December 1994.

[Mat93] J. S. Mattson. *An Effective Speculative Evaluation Technique for Parallel Supercombinator Graph Reduction.* PhD thesis, Department of Computer Science and Engineering, University of California, San Diego, February 93.

[Osb90] Osborne. Speculative Computation in Multilisp. Technical Report DEC-CRL-90-1, Digital Equipment Corporation, Cambridge Research Lab, 90.

[Par91] A. Partridge. *Speculative Evaluation in Parallel Implementations of Lazy Functional Languages.* PhD thesis, Dept. of Computer Science, University of Tasmania, 1991.

[Sha89] Ehud Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, September 1989.

[VK96] M. T. Vandevoorde and D. Kapur. Distributed Larch Prover (DLP): An Experiment in Parallelizing a Rewrite-Rule Based Prover. volume 1103 of *LNCS*, Springer 1996, pages 420–423,