Organizing Speculative Computations in Rule–Based Systems

Raimund Schroeder and Werner Kluge
Department of Computer Science
University of Kiel
D–24098 Kiel, Germany

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 search problems on average decidedly faster than sequentially.*

The paper addresses, in the context of term rewrite systems, the measures necessary to organize speculative computations in multiprocessing environments. They primarily concern task management and scheduling, ensuring fair progress of all speculative tasks, and resolving the conflict between fairness and bounded numbers of speculative tasks.

Keywords: Speculative evaluation, term rewriting, concurrency

1 Introduction

This paper addresses some essential aspects of organizing concurrent speculative computations and of their benefits as regards runtime improvements. Such computations can be conveniently realized in functional (or function–based), term-rewrite or logic-based (PROLOG) systems, in which program execution is conceptually governed by sets of rewrite rules [8, 9, 7, 3, 11, 10]. The basic idea is to evaluate several alternative such rule applications 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 contribute to which extent to problem solutions eventually. A generous supply of processing sites provided, this approach can be expected to compute solutions of np–hard search problems on average decidedly faster than sequential searches.

The organizational problems primarily relate to the coexistence in such systems of speculative and vital tasks which in some orderly form must share limited resources. To prevent the monopolization of the system predominantly with computations which eventually turn out to be useless, vital tasks (which are bound to contribute to solutions) must be given priority over speculative tasks, and the latter must be scheduled preemptively under some fairness regime to move them ahead at about the same pace and to turn processing sites over as soon as possible to newly emerging vital tasks. Additional measures are necessary to strike a reasonable balance between fair progress of all speculative tasks on the one hand and finite resources on the other hand, which requires some throttle on the creation of speculative tasks.

The paper outlines conceptual solutions for these organizational problems and gives some representative performance figures obtained from an implementation that extends an existing graph reduction system which sofar supported only concurrent computations conservative (divide–and–conquer) style [2].

2 Language constructs

The language constructs that are of primary interest in this paper are collections of term rewrite rules (or pattern matching clauses) that have the syntactical form:

```
CASE
   pattern_1 →    term_1
                  ...
   pattern_n →    term_n
OTHERWISE         term_0
END_CASE
```

In a functional interpretation, the CASE–construct specifies a complex unary function. When applied to an argument, all patterns are tried in the order from top to bottom, and the value of the first term whose pattern matches is returned as function value. If none of the patterns matches, *term_0* is taken as an escape hatch. This evaluation order guarantees determinacy of results even in the presence of several (potentially) matching patterns.

However, there are many interesting applications, specifically search problems, where several matching patterns cannot be given a unique ordering with respect to a best possible choice of a solution. This choice may have to be made further down the road as more information to this effect becomes available (and some of the alternatives can safely be discarded), say by repeated application of the same or other CASEs. To speed things up, all matching clauses may then be computed truly concurrently on a speculative basis, provided that conceptually they cause no side effects amongst each other (which is guaranteed if the rewrite rules specify context–free substitutions of equals by equals).

3 Controlling speculative concurrent computations

The idea of concurrent program execution conservative style is to make sure that new tasks are spawned only for program parts which are bound to contribute to results. Such vital (or mandatory) tasks may be scheduled for processing in any order and non–preemptively, i.e., they may yield processing sites only if they terminate or suspend themselves.

Things become decidedly more complicated with speculative tasks entering the game. To commit processing capacity with top priority to useful computations, no speculative task may be scheduled for processing as long as there vital tasks ready to run. Moreover, scheduling speculative tasks on the remaining processing sites 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. Since there are usually more speculative tasks than processing sites they can run on, the tasks must inevitably be scheduled in a round robin fashion to enforce fair progress.

A suitable fairness regulation mechanism for varying numbers of speculative tasks may be based on the the concept of synchronic distances in conjunction with some loose form of barrier synchronization. Synchronic distances, roughly speaking, define upper bounds on the number of operational steps by which several competing tasks (or threads of control) may at most get ahead of each other before the others must catch up. In rule–based computations these steps can easily be counted in terms of rule applications performed.

A suitable mechanism to this effect may involve three successive barriers A, B and C which are some k 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 steps to reach barrier B. Tasks arriving at barrier B (and thus having exhausted their fuel) may cross and be refueled with another k 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 another 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. Tasks blocked at barrier C may yield

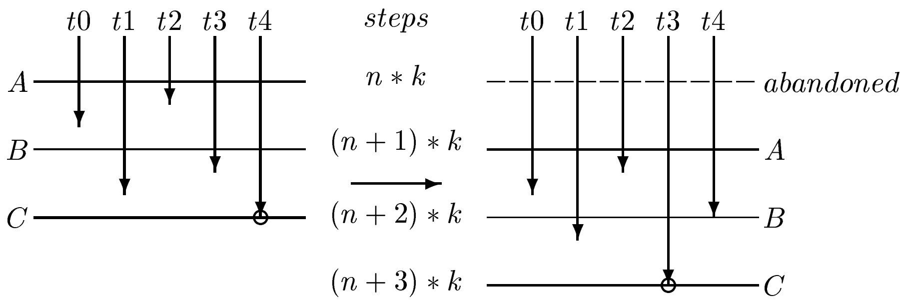

Figure 1: Fairness regulation scheme with three–barrier synchronization

their processing sites either to other speculative tasks that must catch up in the race or to newly created vital tasks. This three–barrier mechanism, which in fact realizes a breadth–first evaluation strategy with a synchronic distance of $2 * k$, ensures good load balancing as it can be expected to keep most of the time as many speculative tasks alive and computing as there are available processing sites (rather than halting all but one of them at barrier C, as a strict barrier synchronization would require).

Since this fairness regulation mechanism must exercise control over all speculative tasks across all processing sites, it may be realized as part of the root task of a program. This task maintains a data structure which is updated whenever new speculative tasks are being spawned, tasks change their status from speculative to vital, terminated or aborted, and barriers are reached or crossed (and the tasks are possibly blocked or provided with more fuel).

Another sophisticated mechanism as depicted in fig. 2 is necessary to have some parent task spawn new speculative children, to register them with the fairness regulation, and to finally start them. Having created in succession speculative tasks, say, for two alternative rule applications, the parent continues until it receives acknowledge messages from the new children, together with their identities, whereupon the children are immediately suspended. The parent registers the new children with the fairness mechanism and suspends itself. The fairness mechanism, in turn, confirms the registration and sends out start signals, together with the fuel left by the parent to reach the next barrier.

Termination or abortion of speculative tasks depends to some extent on the type of the desired result. This could be just any one of possibly several problem solutions, e.g., the one the system returns with first, a unique single solution, say, the one that requires the least number of rewrite steps to arrive at, or all possible solutions requiring less than some pre–specified number of rule applications (the total fuel) to compute [1]. 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 it either produces some FAIL condition by itself or it receives a signal that a solution has been found somewhere else and the task has thus become irrelevant. In either case the task immediately resigns from the fairness mechanism and detaches itself from its parent.

A task terminates regularly if it produces a solution, in which case all other tasks may be aborted if just any one solution is required or, if the desired solution must have consumed the least number of rule application steps, all tasks that have already consumed more steps are signaled to abort at the next barrier. When computing all solutions up to the point of ex-

[1] Without such an upper bound the computation could get trapped in endless recursions.

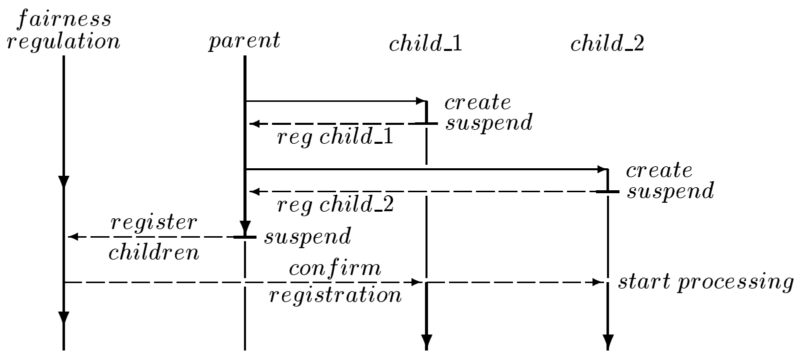

Figure 2: Spawning new speculative tasks

hausting the total fuel, all tasks that are still alive at this point must be aborted.

4 Fairness versus limited resources

Unfortunately, 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 as otherwise too much processing power might be wasted on just the wrong computations. However, a recursively unfolding search problem would require spawning unbounded numbers of speculative tasks as there is generally no upper limit on the size of the search space. This would not only create management overhead without further performance gains (the opposite is generally the case as job granularities become finer, too) but would also consume decidedly more memory for runtime structures and heap space than a strictly sequential search at the other extreme.

Thus, an acceptable compromize must be made between unbounded demands for resources on the one hand and enforcing strict fairness among speculative computations on the other hand. A first measure to this effect is to provide a system–wide pool of concessions to spawn new tasks [6]. Whenever such an opportunity arises, the pool is checked for the availability of a concession, and if so, it is taken and a new task is created; otherwise this opportunity is ignored and the rule application that was to be evaluated by the new task is instead processed sequentially by the calling task. Terminating tasks recycle their concessions to the pool, which may be immediately picked up to spawn other tasks as long as such opportunities come about. Thus, at any time there are at most as many tasks alive in the system as there were initially concessions in the pool.

To save memory, this measure must be complemented by switching, within the tasks that exist after exhausting all concessions, from a breadth–first to a depth–first execution mode which, in a strict sense, violates fairness: several alternative rule applications for which no speculative tasks can be spawned can then only be evaluated in sequence and each of them depth–first.

The way out of this dilemma consists in placing into a task–specific $FIFO$–queue all rule applications which 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 the task produces a failure in the leftmost branch, 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 become available again later on, the queues of all existing tasks are inspected and the pending rule applications highest up

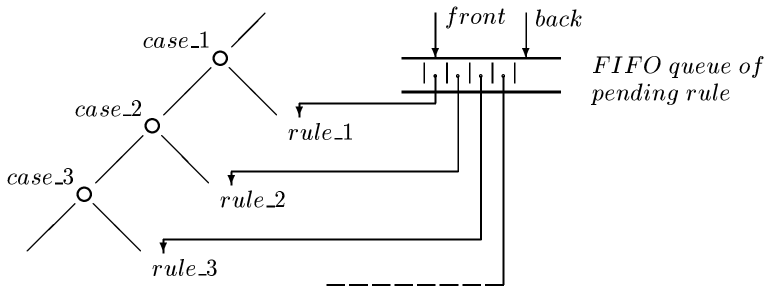

Figure 3: Building a FIFO queue of pending rule applications

across all tasks are chosen to spawn new tasks.

Since a task, after having produced a failure in its leftmost branch, backtracks to the topmost alternative rule application held at the front–end of the queue, the queue cannot strictly be operated in *FIFO* order. Instead, the pending rule applications of the new branch must be sorted into the existing queue so that rule applications of higher levels always precede rule applications 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 Performance experiments

The concepts described in the preceding sections, which are equally well suited for both shared and distributed memory systems, have been implemented as extensions of a concurrent version of a compiled graph reduction system π–RED [2, 4] which interprets abstract machine code. High–level programs, other than identifying CASE–constructs as candidates for speculative evaluation, do not contain any annotations pertaining to concurrent execution.

π–RED is installed on an nCUBE/2 distributed multiprocessor system of 32 processing sites, of which each is equipped with 16 MBytes of memory. The system can be configured to form sub–cubes of some $2^k \mid k \leq 5$ sites which are logically directly connected to each other. Each site runs a micro kernel nCX which supports a single π–RED process. This process realizes a small kernel of its own which handles π–RED–specific task management, scheduling, communication and also the local parts of the fairness regulation 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 regulation) 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.

This machinery has been used to systematically investigate performance enhancements due to speculative evaluation relative to sequential program execution. For this purpose we have chosen a program which searches for exits in a maze. The search is simulated by a spider walking step by step through the maze and checking at each position, by means of pattern matching, 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 some

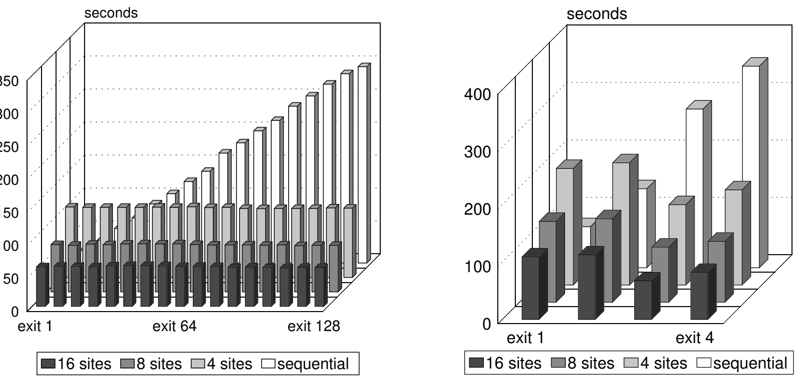

Figure 4: Performance results for speculative searches of exits in a tree–shaped maze (left) and in an square-shaped irregular maze (right)

position in the maze, there is no alternative left but to move backward. This program can be parameterized with respect to size and layout of the maze, number and positions of exits, starting position and width of the steps taken by the spider.

The first problem investigated is the search for a single exit in a maze featuring the regular shape of a balanced binary tree of depth 7 (or 128 leafs at the base). 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 leafs. The program execution times on the left of figure 4 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 exit positions as the search 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.

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 paths leading to these exits. The searches were done 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 synchronic distance of the fairness mechanism or on upper bounds on the number of speculative tasks), again 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 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. 4.

6 Related work

Related work in the functional domain concerns speculative evaluation as an optimization technique under a lazy regime, with HASKELL as the language of choice [9, 7]. In [3] are described several priority schemes for speculative tasks, up– and down–grading, and both explicit and implicit abortion of tasks that have become irrelevant.

A rather elaborate priority scheme for the speculative evaluation of LISP programs is reported in [8]. It uses a so–called sponsor model which favors the execution of speculative tasks with the best chances of succeeding. Sponsorships are dynamically adjusted as computations proceed and more information about possible outcomes become available.

In logic programming, the concepts closest to our approach relate to the exploitation of the so–called *OR*–parallelism in PROLOG–programs [10]. However, global fairness mechanisms are considered unnecessary as multiple tasks may share the same set of clauses. Scheduling strategies for *OR*– parallel processing are described in [1]. 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 [5].

Representative for related work on term–rewrite systems is [11]. It describes an interactive theorem prover which allows to launch several speculative attempts to prove conjectures with different strategies. Both *AND*– and *OR* parallelism are supported, and the number of competing tasks or the depth of the recursive expansion can be restricted. Whenever a proof succeeds, the system aborts all other attempts; unsuccessful attempts must be cleaned up by hand.

References

[1] A. Beaumont. Scheduling Strategies and Speculative Work. *LNCS*, Vol. 569, pp. 120–131, 1991.

[2] T. Buelck, A. Held, W. Kluge, C. Rathsack, S.-B. Scholz, R. Schroeder. Experience with the Implementation of a Concurrent Graph Reduction System on an nCUBE/2 Platform, *LNCS*, Vol. 854, pp. 497-508, 1994.

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

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

[5] B. Hausman. Pruning and Scheduling Speculative Work in OR-Parallel Prolog. *LNCS*, Vol. 366, pp. 133–150, 1989.

[6] W.E. Kluge. Cooperating Reduction Machines. *IEEE Transactions on Computers*, Volume C-32 (11) 1002–1012, 1983.

[7] 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.

[8] R.B. Osborne. Speculative Computation in Multilisp. *LNCS*, Vol. 441, pp. 103 – 137, 1990.

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

[10] E. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, Volume 21 (3), pp. 413–510, 1989.

[11] M.T. Vandevoorde, D. Kapur. Distributed Larch Prover (DLP): An Experiment in Parallelizing a Rewrite-Rule Based Prover. *LNCS*, Vol. 1103, pp. 420–423, 1996.