Design and Implementation of Parallel TRAM

Kazuhiro Ogata, Masaru Kondo, Shigenori Ioroi and Kokichi Futatsugi

JAIST, JAPAN ({ogata, m-kondo, ioroi, kokichi}@jaist.ac.jp)

1 Introduction

Algebraic specifications, introduced in the mid 70's as a method of modeling and specifying abstract data types, has been widely attracting attention since they have exact semantics and the ability to verify and reason about specifications of software systems. In addition, since specifications written in algebraic specification languages can be executed on stock hardware, the specification languages can be used as tools for rapid prototyping.

TRAM[7] is an abstract machine for order-sorted conditional term rewriting systems (OSCTRSs). The OSCTRSs[5] can serve as a general computation model for advanced algebraic specification languages such as *OBJ*[6] and *CafeOBJ*[2]. TRAM adopts the *E-strategy*[6] as its reduction strategy. Parallel TRAM is a parallel variant of TRAM that is designed to be executed on shared-memory multiprocessors. In Parallel TRAM, parallelism directives are specified by using the *Parallel E-strategy* that is an extension of the E-strategy. The Parallel E-strategy may control parallelism suitably by combining conditions. In this paper, we describe the design and implementation of Parallel TRAM and assess the current implementation on OMRON LUNA-88K^2.

Up to the present, several researches on designing rewrite engines for algebraic specification languages on parallel architectures have been done[4, 8]. Massively parallel computers have been mainly focused as the target architectures so that much faster rewritings by far could be achieved. But, almost all the rewrite engines have been gone no further than having been designed. Shared-memory multiprocessors have been chosen as our target architecture since we think the multiprocessors will undoubtedly become standard for the future workstations.

2 TRAM: Term Rewriting Abstract Machine

TRAM Programs. TRAM programs are rewrite (equational) programs that are similar to OBJ's modules. For example, the program defining a function that returns the nth element of (infinite) natural numbers' lists is as follows:

```
sorts: Zero NzNat Nat List .
order: Zero < Nat    NzNat < Nat .
ops: 0 : -> Zero    s : Nat -> NzNat
     cons : Nat List -> List { strat: (1 0) }
     inf : Nat -> List    nth : Nat List -> Nat .
vars: X Y : Nat  L : List .
```

rules: $inf(X) \rightarrow cons(X, inf(s(X)))$
$nth(0, cons(X, L)) \rightarrow X$
$nth(s(X), cons(Y, L)) \rightarrow nth(X, L)$.

TRAM adopts the *E-strategy* that lets each operation have its own *local strategy*. The local strategies indicate the order of rewritings of terms whose top operations have the strategies. The order is specified by using lists of numbers ranging from zero to the number of the arguments. Non-zero number n and zero in the lists are intended to reduce (evaluate) nth arguments of the terms and the terms themselves to a variant of normal forms respectively. We call the variant of normal forms *E-normal forms* (*ENFs*). Arguments whose numbers are not in the lists might or might not be evaluated lazily. The operation *cons* has the local strategy (1 0) that indicates a term whose top operation is *cons* is tried to be rewritten to another after evaluating the first argument to ENF when the term is evaluated. If the term is rewritten to another, the new one will be evaluated according to the local strategy of its top operation. The second argument might or might not be evaluated lazily. The eager local strategy (1 2...0) is attached to each operation to which explicit local strategies are not specified. Figure 1 shows the reduction sequence for $nth(s(0), inf(0))$ using the above program.

TRAM Architecture. TRAM consists of six regions (DNET, CR, CODE, SL, STACK and VAR) and three processing units (the rule compiler, the term compiler and the TRAM interpreter). DNET is the region for *discrimination nets* [3] encoded from the lefthand sides of rewrite rules. The righthand sides of rewrite rules (RHSs) are compiled and allocated on CR. Matching programs compiled from subject terms are allocated on CODE. SL contains strategy lists for subject terms. STACK is the working place for pattern matching. VAR contains substitutions.

In TRAM, subject terms are compiled into sequences of abstract instructions (*matching programs*) that are self modifying programs.

Definition 1. The matching program L_T for a term T whose top operation is f of arity n is as follows:

```
L_T: match_sym idx_f
     L_1          // idx_f is the index for f.
     :            // L_i(i = 1,...,n) is the label of
     L_n          // the ith argument's matching program.
```

Figure 1 shows some terms and the corresponding matching programs. All applicable rewrite rules for T are gained by executing the program. The program is called by jumping the label L_T after pushing a continuation (a return address) onto STACK. `match_sym` tests whether f is in the root node of (a sub-tree of) DNET. If f is in the root, the continuations (the arguments) $L_n, \ldots, L_2$ are pushed onto STACK and the control is transferred to L_1. Backtracking is used so as to find all applicable rewrite rules for the term. The method for backtracking used in WAM [1] for Prolog is adopted.

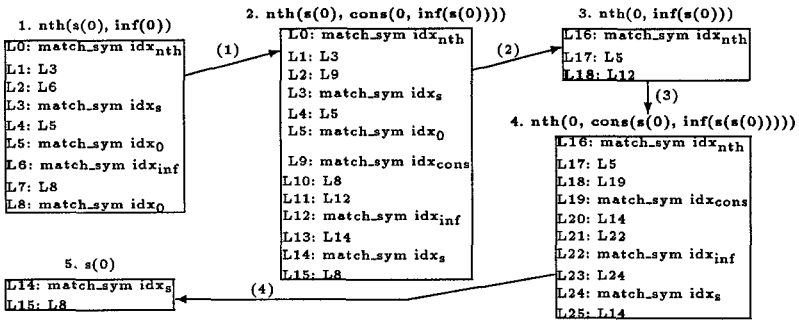

Fig. 1. Reduction sequence for *nth*(*s*(*0*), *inf*(*0*))

Matching program templates are compiled from RHSs and are instantiated when the corresponding rules are used. The matching program at L9 through L15 in Fig. 1 is the instantiated one of the matching program template of *inf*(X) -> *cons*(X, *inf*(X)) that is used in the rewriting (1) in Fig. 1.

Rewriting Machinery. The implementation of the E-strategy in TRAM is based on *strategy lists*. A strategy list for a term t is basically a sequence of all eager positions (reachable nodes) in t. The order of the strategy list corresponds to the order of the evaluation of t. Elements of strategy lists in TRAM are triples $\langle label, pslot, skip \rangle$ of matching programs' labels, parent slots holding the labels, and the number of elements to be skipped. The last element of the strategy lists is the *BINGO* triple $\langle$*BINGO, subject,* _$\rangle$. *BINGO* is the label of the instruction `bingo` that is executed when an ENF is got. The second of the *BINGO* triple is the parent slot of subject terms. The TRAM interpreter executes the matching program to reduce a subject term according to the strategy list until the BINGO triple. The strategy list for *nth*(*s*(*0*), *inf*(*0*)) in Fig. 1 is [$\langle$*L5, L4, 0*$\rangle$, $\langle$*L3, L1, 0*$\rangle$, $\langle$*L8, L7, 0*$\rangle$, $\langle$*L6, L2, 0*$\rangle$, $\langle$*L0, RESULT, 0*$\rangle$, $\langle$*BINGO, L0,* _$\rangle$].

When an applicable rule is got by executing the matching program for a subterm of a term, the strategy list template of the RHS is instantiated and the instantiated list is appended to the remaining strategy list of the term so that the strategy list for the new term is gained after replacing the subterm (redex) with the corresponding contractum. [$\langle$*L9, L2, 0*$\rangle$] is the instantiated one for the strategy list template of *inf*(X) -> *cons*(X, *inf*(*s*(X))) that is used in the rewriting (1) in Fig. 1. The strategy list for *nth*(*s*(*0*), *cons*(*0*, *inf*(*s*(*0*)))) is [$\langle$*L9, L2, 0*$\rangle$, $\langle$*L0, RESULT, 0*$\rangle$, $\langle$*BINGO, L0,* _$\rangle$] after the rewriting (1) is done.

The TRAM interpreter executes the following instruction sequence when it begins to interpret the matching programs to reduce terms:

```
        init          // initializes TRAM's registers
LOOP:   next          // pops a label from SL and puts it at L_Dummy
        jump L_Dummy
        go_ahead
        select        // selects one among the applicable rules
```

```
rewrite        // replaces the redex with its contractum
jump LOOP
```

`next` also pushes the `go_ahead`'s label onto STACK. `go_ahead` is executed when an applicable rule is found. If there is no applicable rule, the control is transferred to *LOOP*. `go_ahead` triggers off backtracking if there is a choice point frame [1]. Otherwise it transfers the control to `select`. `rewrite` also appends the instantiated strategy list of the used rule to one for the subject term.

3 Parallelization for TRAM

The Parallel E-strategy. Since rewrite programs have parallelism inherently, explicit parallelization directives are not necessary for executing them in parallel [8]. Generally speaking, however, we can gain better performance of a program when its restricted subtasks with sufficiently large amount of work are only assigned to parallel processes than when its all subtasks are assigned to parallel processes. The ability to control parallelism is very desirable for this reason. Especially, it is indispensable to control parallelism of rewrite programs so that they are executed efficiently in parallel on a multiprocessor with a small number of processors.

In Parallel TRAM, we adopt the *Parallel E-strategy*, that is an extension of the E-strategy and is a subset of the *Concurrent E-strategy* [4], in order to control parallelism of rewrite programs. The Parallel E-strategy lets each operation have its own *parallel local strategy*. A parallel local strategy for an operation f of arity n is specified by using a list defined by the following extended BNF notation:

Definition 2.

⟨ParallelLocalStrategy⟩ ::= ( ) | (⟨SerialElem⟩* 0)
⟨SerialElem⟩ ::= 0 | ⟨ArgNum⟩ | ⟨ParallelElem⟩
⟨ArgNum⟩ ::= 1 | 2 | ... | n
⟨ParallelElem⟩ ::= { ⟨ArgNum⟩$^+$ }

⟨ParallelElem⟩ specifies some arguments of a term whose top operation is f that are reduced in parallel. Each element of ⟨ParallelLocalStrategy⟩ are evaluated in sequence from left. For example, a TRAM program computing Fibonacci numbers in parallel can be defined as follows:

```
sorts: Nat .
order: .
ops: 0 : -> Nat    s : Nat -> Nat
     padd : Nat Nat -> Nat { strat: ({1 2} 0) }
     pfib : Nat -> Nat .
vars: X Y : Nat .
rules: padd(X, 0) -> X    padd(X, s(Y)) -> s(padd(X, Y))
       pfib(0) -> 0    pfib(s(0)) -> s(0)
       pfib(s(s(X))) -> padd(pfib(s(X)), pfib(X)) .
```

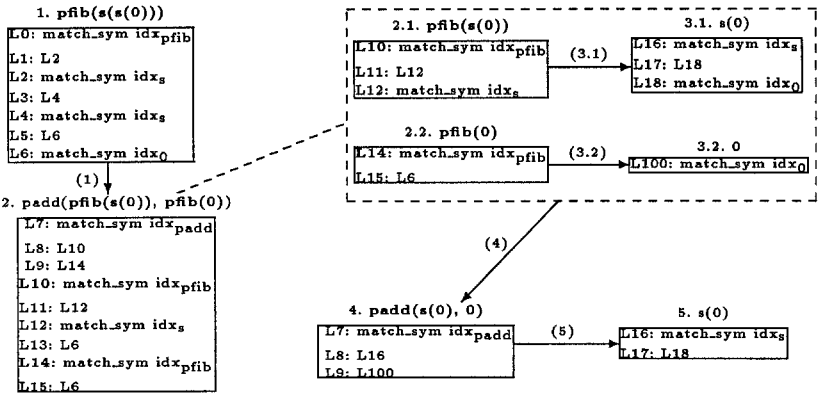

Fig. 2. Parallel reduction sequence for $pfib(s(s(0)))$

The operation *padd* has the parallel local strategy ({1 2} 0) that indicates a term whose top operation is *padd* is tried to be rewritten to another after evaluating the first and second arguments to ENFs in parallel when the term is evaluated. Figure 2 shows the parallel reduction sequence for $pfib(s(s(0)))$. The two rewritings enclosed with the dash frame in Fig. 2 are done in parallel.

The Parallel E-strategy may control parallelism more suitably by combining conditions. Instead of $pfib(s(s(X))) \rightarrow padd(pfib(s(X)), pfib(X))$, the following rules can be defined for gaining a more efficient program:

$$pfib(s(s(X))) \rightarrow padd(pfib(s(X)), pfib(X)) \textbf{ if } threshold(X) = false$$
$$pfib(s(s(X))) \rightarrow add(fib(s(X)), fib(X)) \textbf{ if } threshold(X) = true$$

threshold returns (is reduced to) *false* if its argument is larger than a number. Otherwise it returns *true*. *add* is a sequential addition and *fib* computes Fibonacci numbers in sequence by using *add*.

Parallel TRAM Architecture. In Parallel TRAM, each processor has its own TRAM interpreter and four mutable regions. Two stable regions are shared with all processors. One of the processors is also an interface processor that also plays a role of the user interface by using the rule compiler and the term compiler. A strategy list represents a process queue. A chunk of consecutive elements of the strategy list represents a process. Each process has two possible states: *active* and *pending*. Each process queue contains zero or more pending processes and zero or one active process. If there is an active process in a process queue, it is at the top of the queue. A processor whose process queue contains an active process is in *busy*. Otherwise it is in *idle*.

Parallel Rewritings. It is not necessary to change matching programs even though terms are reduced in parallel. Strategy lists control parallel rewritings by holding labels at which new parallel instructions are stored. The new parallel instructions are `fork`, `join`, `exit`, `sleep` and `nop`. `fork` creates a new process

reducing a subterm in parallel and allocates the processor to it if there is an idle processor. Otherwise the caller processor reduces the new process, or the new process is *in-lined.* `join` delays its caller process until all of its child processes terminate and makes the caller processor idle. `exit` terminates its caller process and reports the termination to its parent process. After executing `exit`, the caller processor resumes the pending process at the top of its process queue if there are pending ones in the queue. Otherwise the processor becomes idle by executing `sleep`. `sleep` makes its caller processor idle. `nop` does nothing. `fork` creating an empty process may appear in a strategy list by instantiating a strategy list template. `nop` is replaced with such a wasteful `fork`.

There are two kinds of triples that are elements of strategy lists in TRAM. One is for matching programs, the other for `bingo`. In addition to these ones, Parallel TRAM has five kinds of triples as elements of strategy lists. These new ones correspond to the new five parallel instructions: $\langle FORK,\ SIZE,\ A_{\mathrm{J}}\rangle$, $\langle JOIN,\ PNUM,\ \_\rangle$, $\langle EXIT,\ A_{\mathrm{PJ}},\ PID\rangle$, $\langle SLEEP,\ \_,\ \_\rangle$ and $\langle NOP,\ \_,\ \_\rangle$. FORK is the label of `fork`. $SIZE$ is the size of the forked process. A_{J} points to the JOIN triple corresponding to the FORK triple. JOIN is the label of `join`. $PNUM$ is a number of the caller's child processes that do not finish their work. EXIT is the label of `exit`. A_{PJ} points to the JOIN triple of the caller's parent process. PID is the processor ID of the caller's parent process. SLEEP is the label of `sleep`. NOP is the label of `nop`. For example, the strategy list for $padd(pfib(s(0)),\ pfib(0))$ after rewriting (1) in Fig. 2 is $[\langle FORK,\ 2,\ A_{\mathrm{J}}\rangle, \langle L12,\ L11,\ 0\rangle, \langle L10,\ L8,\ 0\rangle, \langle L14,\ L9,\ 0\rangle, \langle JOIN,\ 1,\ \_\rangle, \langle L7,\ RESULT,\ 0\rangle, \langle BINGO,\ L7,\ \_\rangle]$. Suppose that there are two processors P0 and P1 that are in active and in idle respectively. After rewriting (1), P0 executes `fork`, and a new process is created and is allocated to P1. Then, P1's strategy list is $[\langle L12,\ L11,\ 0\rangle, \langle L10,\ L8,\ 0\rangle, \langle EXIT,\ A_{\mathrm{J}},\ P0\rangle, \langle SLEEP,\ \_,\ \_\rangle]$. The two processors reduces terms in parallel soon after.

4 Implementation of Parallel TRAM on Multiprocessor

Parallel TRAM has been implemented on OMRON LUNA-88K^2 in C. LUNA-88K^2 carries four MC88100 processors and adopts Mach 2.5 as its OS.

Processors are represented by using C structures in which four mutable regions and the related registers such as the instruction pointer are packed. Each structure also contains states of processors: *BUSY* and *IDLE*. Parallel execution is realized by giving each processor a Mach thread. Pointers to the structures are passed to C functions in which four mutable regions or the related registers are accessed so that processors (threads) can access them as quickly as possible. Idle processors are managed by using a processor idle queue *IdleQueue*. When `fork` is executed, its caller processor tries to get one of idle processors from IdleQueue and to allocate it to a newly created process. If IdleQueue is empty, the new process is in-lined.

Parallel TRAM adopts a sequential stop-and-copy algorithm as the garbage collection (GC). A processor detecting a necessity of a GC performs the GC after all other processor pause. In Parallel TRAM, one global creation space (where

Table 1. Experimental results in computing the 24th Fibonacci number

system	time (s)	rewritings	r/s	GCs	created processes	in-lined processes	speed
tram	13.63	514105	37719	6	–	–	base
ptram1	14.71	527377	35852	6	0	376	0.93
ptram2	8.84	527377	59658	6	8	368	1.54
ptram3	7.55	527377	69851	7	39	337	1.81
ptram4	6.04	527377	87314	6	20	356	2.26

ptram$i$ means the Parallel TRAM system running on i processors.
The threshold is 10 for parallel computations of Fibonacci numbers.

new matching programs are allocated) is divided into multiple chunks for the purpose of efficient usage of storage and each processor gets one chunk from the global space when necessary. A GC is triggered when there is no space left in the global space. In addition to *BUSY* and *IDLE*, *GC* is used as a processor state for a GC. After a GC, the GC processor resumes the processors pausing for the GC. It is necessary to distinguish processors pausing for the GC from idle ones at the moment. The state *GC* is given to processors that are the second or later to detect the necessity of the GC.

5 Performance of Parallel TRAM

Table 1 shows the experimental results in computing the 24th Fibonacci number in parallel with the Parallel TRAM system on LUNA-88K[2]. For comparison with TRAM, the results of the TRAM system are shown. Due to limitations of space, the overhead of the current implementation is only discussed here.

We calculate an ideal improvement in speed when the 24th number is computed in parallel on four processors. When the 24th number is computed in parallel, two processors add the computations of the 22nd and 21st numbers, and of the 21st and 20th numbers in parallel after four processors compute the 22nd, 21st, 21st and 20th numbers in parallel. Then, the two sums are added so as to get the 24th number. The computations of the 22nd, 21st, 21st and 20th numbers involve 186579, 112286, 112286 and 67596 rewritings, respectively. If these rewritings are fairly done in parallel with four processors, it takes a time in proportion to 119687 rewritings (the mean of the four rewritings) to compute the four numbers. The additions of the computations of the 22nd and 21st, and of the 21st and 20th numbers involve 10948 and 6697 rewritings, respectively. Since these two additions are done independently with two processors, it takes a time in proportion to 130635 rewritings (119687 + 10948) to compute the 23rd and 22nd numbers. It takes a time in proportion to 148348 rewritings (130635 + 17713) to compute the 24th number in parallel with four processors since the addition of the computations of the 23rd and 22nd numbers involve 17713 rewritings. Hence, the ideal improvement in speed is 3.47 (514105/148348) when the 24th number is computed in parallel on four processors.

Since the actual improvement in speed is 2.26, there are some considerable overheads in the current implementation. The overheads may depend on the

sequential GC, the process creation, the parallelization of TRAM and the architectural characteristic of LUNA-88K[2] such as shared bus. The process creation and the parallelization of TRAM may slightly contribute to the overhead judging from the experimental result of ptram1 in Table 1. It is currently unclear that how much the architectural characteristic of LUNA-88K[2] affects the overhead. We should implement Parallel TRAM on another multiprocessor so as to assess that thing. The computation of the 24th Fibonacci number was done in parallel with four processors on the Parallel TRAM system with enough size of the creation space to involve no GCs so that the overhead of the sequential GC was confirmed. It took 4.90 seconds. The overhead caused by the sequential GC is about 19% in the Parallel TRAM system while it is about 9% in the TRAM system. One of the main source of the overhead in the current implementation of the Parallel TRAM system is the sequential GC.

6 Conclusion and Future Work

We have described the design and implementation of Parallel TRAM where parallelism directives are specified by using the Parallel E-strategy. The Parallel TRAM system on LUNA-88K[2] was found to be about twice times faster than the TRAM system on the same workstation. Parallel TRAM will be implemented on other multiprocessors so that Parallel TRAM and its implementation technique will be improved and be made secure. One of the points that should be improved in the current implementation is to adopt a parallel or on-the-fly garbage collection.

References

1. Aït-Kaci, H.: Warren's Abstract Machine. A Tutorial Reconstruction. The MIT Press. 1991
2. CafeOBJ home page: http://ldl-www.jaist.ac.jp:8080/cafeobj
3. Christian, J.: Flatterms, Discrimination Nets, and Fast Term Rewriting. *Journal of Automated Reasoning.* **10** (1993) 95–113
4. Goguen, J., Kirchner, C. and Meseguer, J.: Concurrent Term Rewriting as a Model of Computation. Proc. of the Workshop on Graph Reduction. LNCS **279** Springer-Verlag. (1986) 53–93
5. Kirchner, C., Kirchner, H. and Meseguer, J.: Operational Semantics of OBJ-3. Proc. of the 15th International Colloquium on Automata, Languages and Programming. LNCS **317** Springer-Verlag. (1988) 287–301
6. Futatsugi, K., Goguen, J. A., Jouannaud, J. P. and Meseguer, J.: Principles of OBJ2. Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages. (1984) 52–66
7. Ogata, K., Ohhara, K. and Futatsugi, K.: TRAM: An Abstract Machine for Order-Sorted Conditional Term Rewriting Systems. Proc. of the 8th International Conference on Rewriting Techniques and Applications. (1997) (to appear)
8. Viry, P. and Kirchner, C.: Implementing Parallel Rewriting. Proc. of the International Workshop on Programming Language Implementation and Logic Programming. LNCS **456** Springer-Verlag. (1990) 1–15