The Meta Transformation Tool for Skeleton-Based Languages

Marco Aldinucci

Computer Science Dept.
University of Pisa
Pisa – Italy

Skeletons

- Structured programming models
- Skeletons
  $\rightarrow$ ***known, reusable, parallelism exploitation patterns***
  $\rightarrow$ ***Think to the analogy with seq. prog.*** `(while do, for ...)`
- Programmers concentrate on qualitative aspects of parallelism
- Tools deal with implementation details and quantitative aspects
  $\rightarrow$ ***load-balancing, parallelism degree, messages size, etc.***
- Specification of the software architecture

Why skeletons ?

- Parallel programming is difficult and error prone
  → Skeletons have a clear, functional and parallel semantic
- Performance portability
  → ***Only performance can justify HPC high costs***
  → ***Performance heavily depends on the "matching" of the program with the architecture***
- Several MPI implementation of the same algorithm

	seq	farm	farm + pipe	pipe + 2*farm	pipe + farm
T_s (sec)	6.03	0.39	1.30	0.72	4.99
T_c (sec)	1207.76	84.50	286.62	151.67	1004.69
#PE	1	20	20	20	20
$\mathcal{E}(\%)$		75.52	23.08	41.93	6.04

(Aldinucci & Danelutto. IASTED PDCS'99, MIT, Boston, USA)

Common skeletons

- **Task parallel skeletons**
  - **parallelism in the computation of stream tasks**
  - **E.g.** pipeline, task farm, **etc.**
- **Data parallel skeletons**
  - **parallelism in the computation of a single task**
  - **E.g.** map **(independent forall),** reduce, $\mathcal{D}\&\mathcal{C}$, **etc.**
- **Sequential skeletons**
  - **the degenerate case: no parallelism at all**
  - **sometime used to wrap functions written in a guest language**

The methodology

- **Ingredients**
  - **A skeleton-based language**
  - **A cost calculus to foresee program performance**
  - **A set of semantic-preserving rewriting rules**
- **Methodology**
  1. **Write an initial specification/program**
  2. **Evaluate its performance**
  3. **Transform it (until the performance is satisfactory)**
- **Wish list**
  - **A compiler for the language**
  - **A tool to transform (optimize) programs**

Outline

- Framework & Motivation

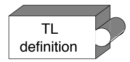
→ Outline

- The Meta transformation tool
  - A short introduction
  - Dealing with languages and rules
  - How it work
  - The architecture and the implementation
  - Running the tool
- Conclusions

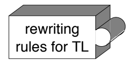
The Meta transformation tool: What does it do ?

Given:

1. **A Target Language (TL)**



2. **A set of (sound) rewriting rules (L $\rightarrow$ R) for TL**



3. **A program written in TL**

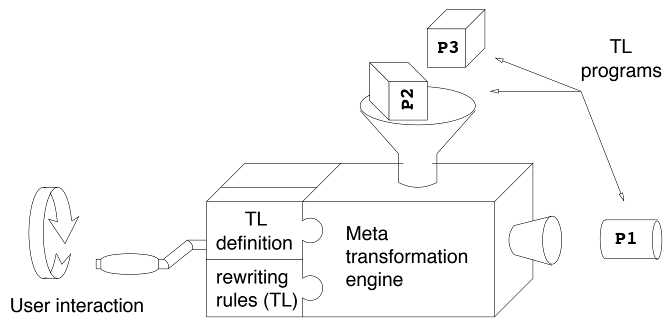
the Meta tool:

- **locates applicable transformations**
- **provides performance estimates**
- **(possibly) transforms the program accordingly with the (user) chosen rule**

Meta basically implements a (meta) term-rewriting system

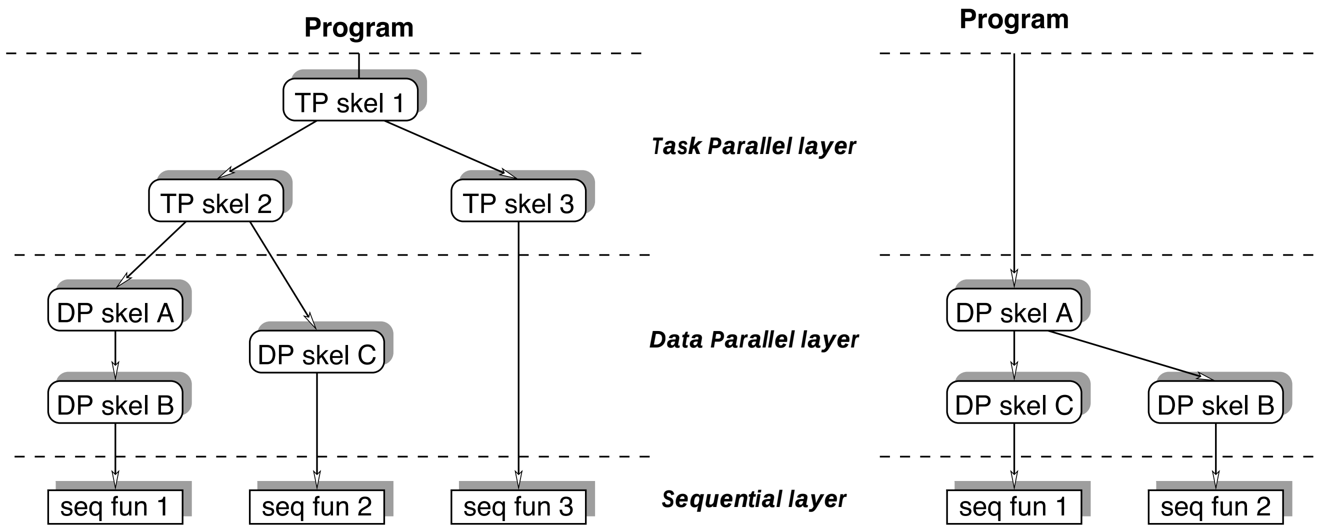
The Meta transformation tool: properties

- **Interactive**
- **Graphical**
- **Fast**
- **Language independent**
- **Rule independent**
- **Manages mixed data and task parallel languages**



Meta ***can be instantiated with a broad class of*** TL ***(three-tier languages) and sets of rules for*** TL**.**

Skeleton-Based (up to) three-tier languages



Lower levels skeletons cannot call upper level ones

"The exploitation of task parallelism is (often) orthogonal with respect to the exploitation of data parallelism"

Three-tier languages: testbeds

FAN: Functional Abstract Notation

(Aldinucci, Gorlatch, Lengauer, Pelagatti. Parallel Algorithms & Applications (to appear), Gordon & Breach.)

- **Data parallel skeleton language**
- FAN **cost calculus**

Skel-BSP: Skeletons on top of the H-BSP

(Zavanella. Ph.D. Thesis, University of Pisa)

- **Task and data parallel skeleton language**
- **Implemented on top of (hierarchical) BSP**
- **BSP-like cost calculus**

Notice I don't present here neither new languages nor new rules

Indeed, they have been presented and validated elsewhere

Three-tier languages: example Skel-BSP

TL_prog ::= TP | DP
TP ::= farm (TP) | pipe {$TPlist$} | DP
DP ::= map Seq | scanL Seq | reduce Seq | Seq | comp (**out** Var, **in** $Varlist$) {$DPlist$}...
Seq ::= $\langle$ *a sequential C function* $\rangle$
...

comp.$name$ (out $outvar$, in $invars$){

$outvar_1$ = dp.1 Op_1 $invars_1$

$\vdots$

$outvar_n$ = dp.n Op_n $invars_n$}

comp definition follows the *single-assignment* rule: there is at most one equation defining each variable. Skeletons into comp are executed in sequence on a single set of PEs.

Rewriting rules

A rule is a pair $L \to R$ where

- L **and** R **are fragments of** TL **programs with variables** $\nu_0, \nu_1, \ldots$
- $\nu_0, \nu_1, \ldots$ **act as placeholders for any piece of program**
- **Every variable occurring in** R **must occur also in** L
- L **is not a variable**

Rewriting rules: examples (1)

TSk $\overset{\rightarrow}{\leftarrow}$ **farm (TSk)**

A **farm** replicates **TSk** without changing the function it computes

pipe { **DSk**$_1$ Op_1, **DSk**$_2$ Op_2, $< \cdots >_1$ **DSk**$_n$ Op_n}	$\rightarrow$	**comp (out** z, **in** a) { b = **DSk**$_1$ Op_1 a, c = **DSk**$_2$ Op_2 b, $< \cdots >_1$ z = **DSk**$_n$ Op_n y}

Same functional (sequential) semantic. They differ in the execution model. pipe stages run on on different set of PEs, comp stages run in sequence on the same set of PEs

Rewriting rules: examples (2)

comp (**out** *outvar*, **in** *invars*) $\{$

$< \cdots >_1$

q = **map** $Op_1\ p$,

$< \cdots >_2$

r = **map** $Op_2\ q$,

$< \cdots >_3$ $\}$

$\rightarrow$
$\leftarrow$

comp (**out** *outvar*, **in** *invars*) $\{$

$< \cdots >_1$

q = **map** $Op_1\ p$,

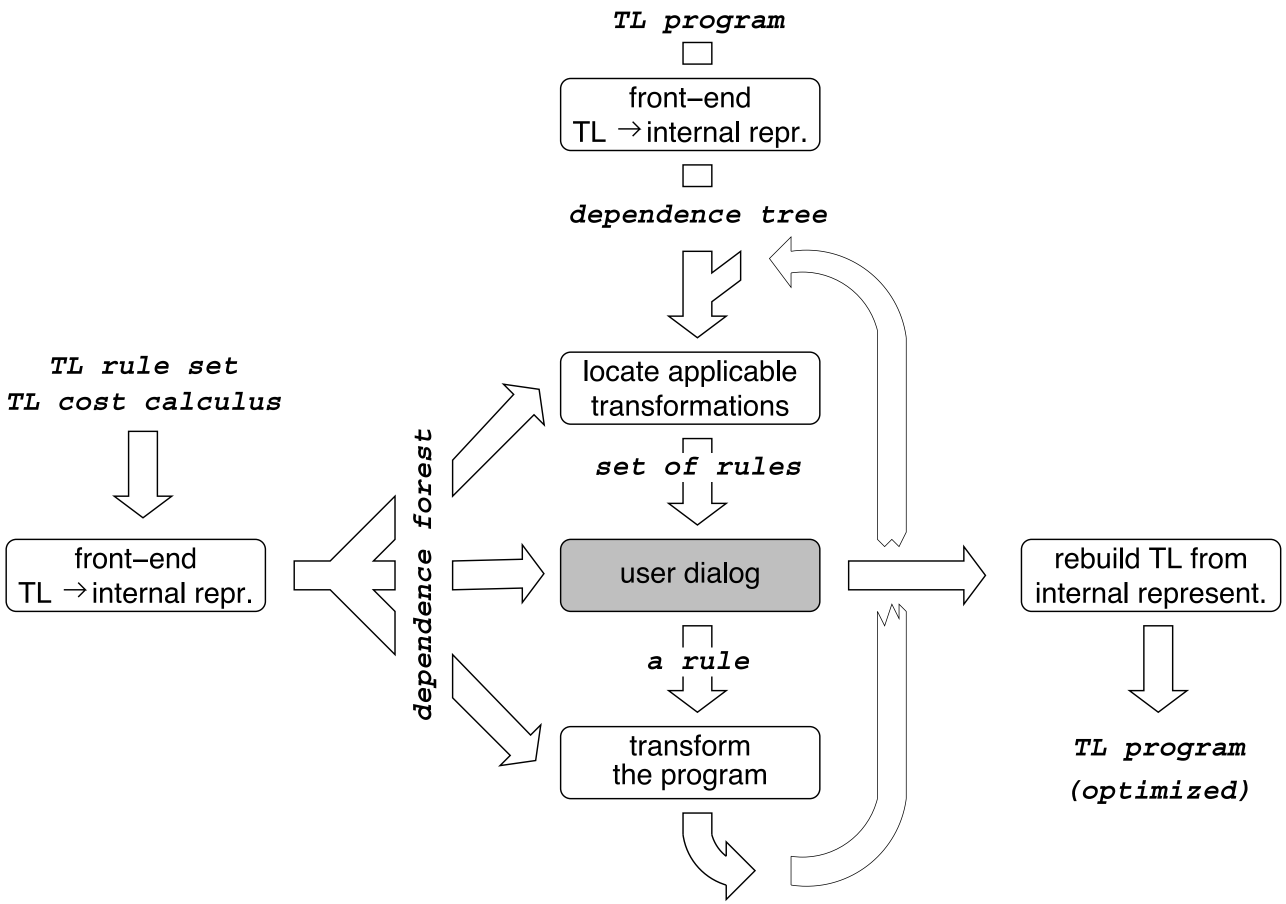
r = **map** $(Op_2 \circ Op_1)\ p$,

$< \cdots >_2$

$< \cdots >_3$ $\}$

map (backwards) distribution through functional composition. We do not require the two **map**s to be adjacent in the program code. Meta provides the program with the additional assignment only if the intermediate result q is referenced into $< \cdots >_2$ or $< \cdots >_3$

Tool behaviour



Representing program and rules

Both program and rules are represented by ***dependence trees***

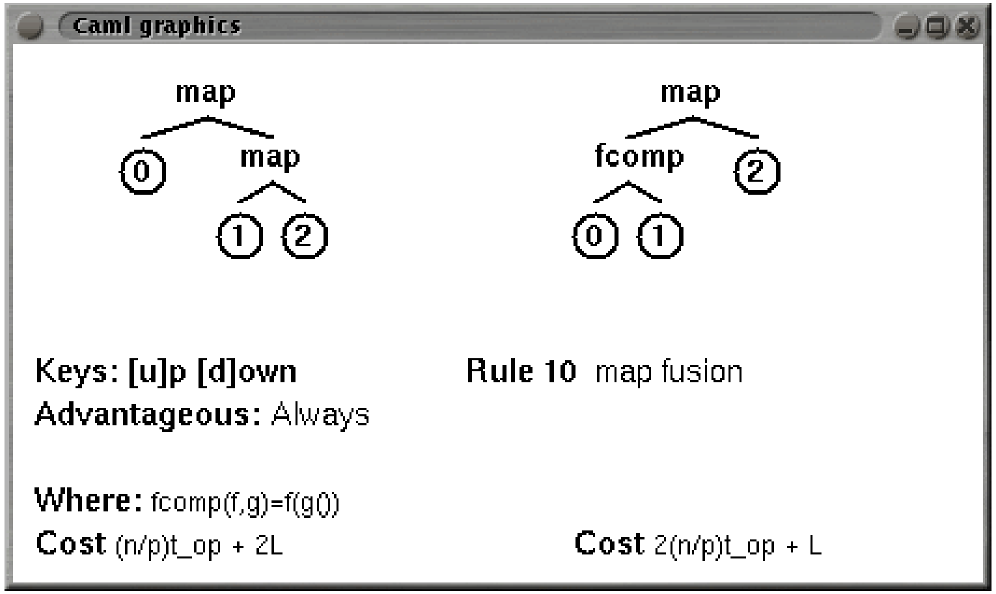
- A ***dependence tree*** (DT) is a labelled tree
- DT directly represents the data dependence among skeletons
- if `Sk1` directly uses data produced by `Sk2`
  then they are adjacent in DT
- DT is built starting from parse tree (PT) and
  data flow graph (DFG) of the program

☺ both PT and DFG can be build using standard tools

☺ Since program and patterns (L) are trees, the search for applicable rules reduces to subtree matching

Representing program and rules (cont'ed)

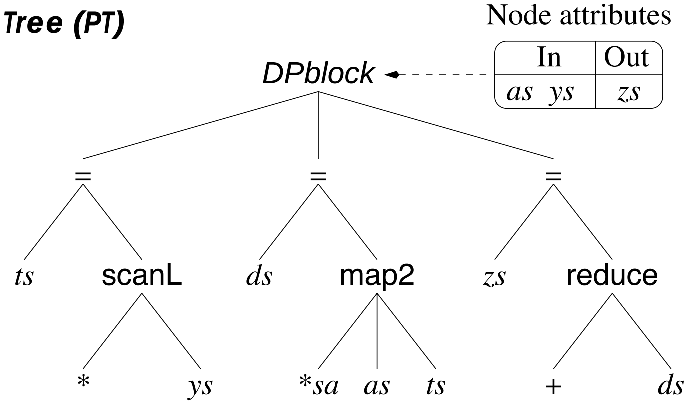
A rule is represented by a dependence tree pair



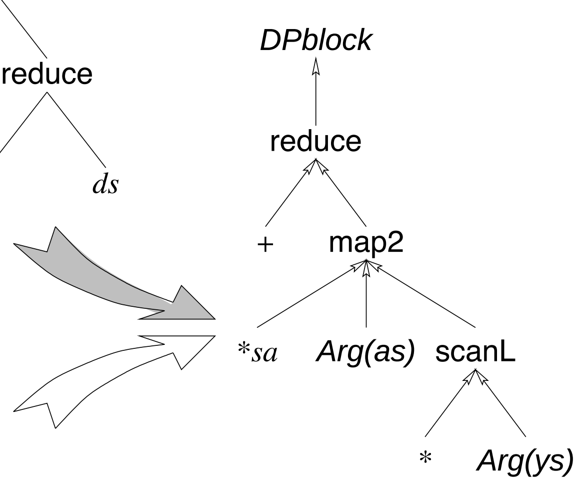
- Circled figures represents variables
- fcomp is special node representing functional composition

Building up the *Dependence Tree*

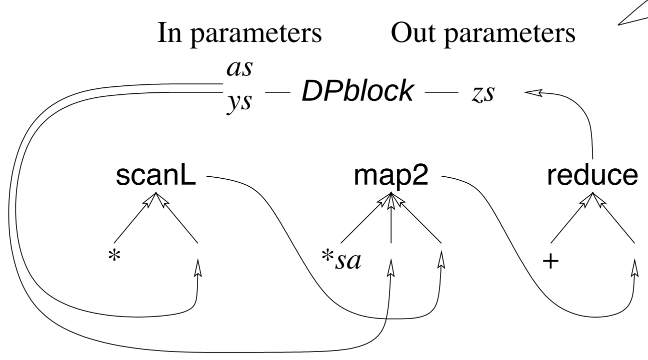
Parse Tree (PT)



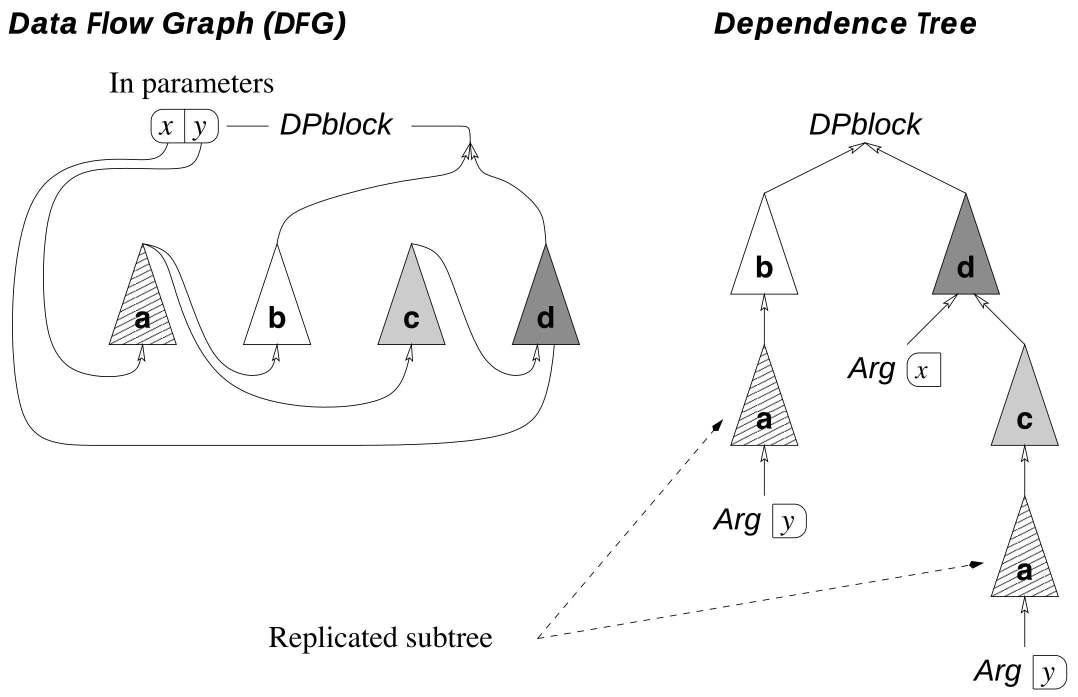
Dependence Tree



Data Flow Graph (DFG)



Building up the *Dependence Tree* (cont'ed)



shared subtrees: keep them shared or replicate them ?

Locating applicable rules: subtree matching

Problem: Match a set of patterns against many subject trees

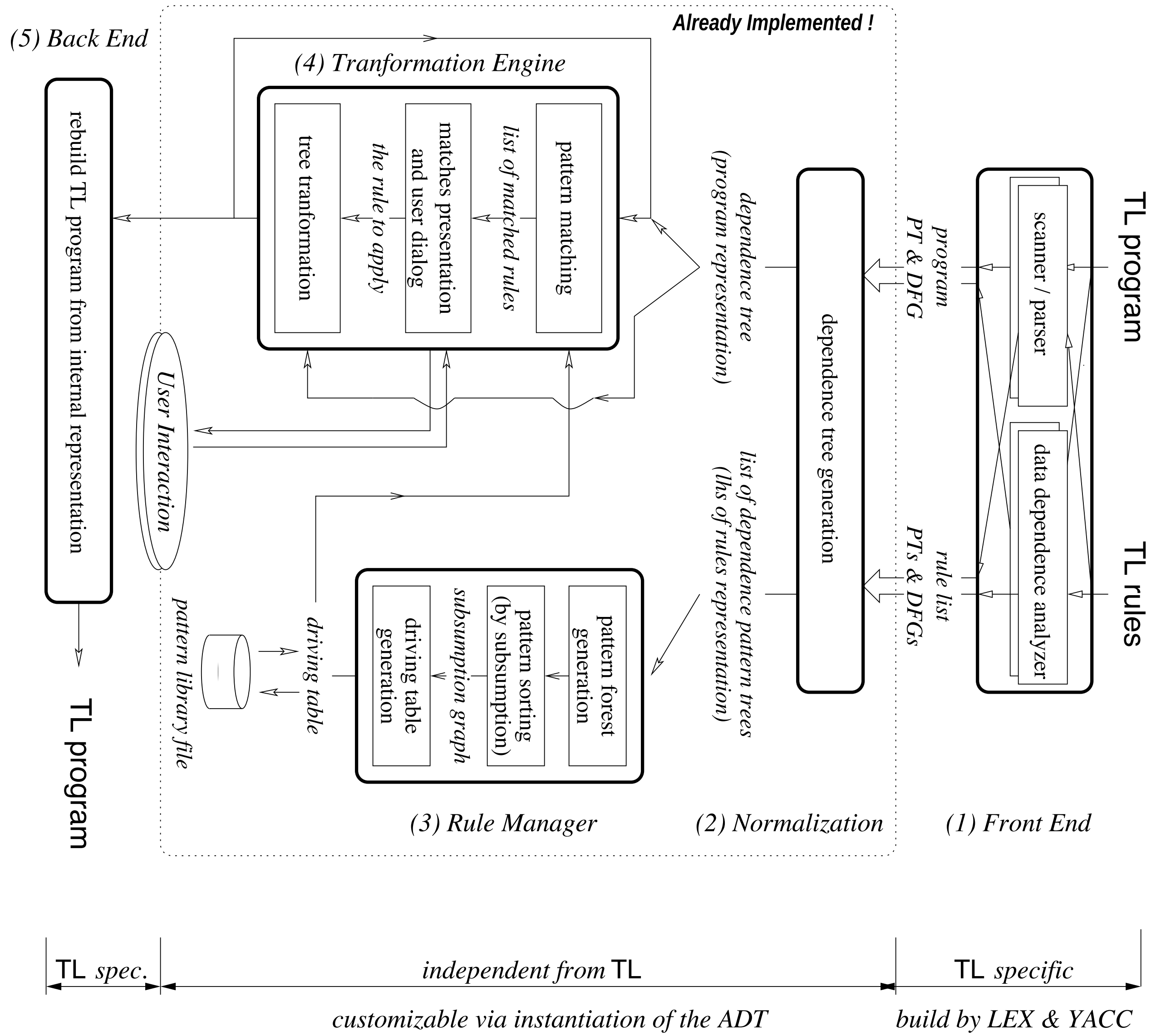
Solution: Hoffman-O'Donnell bottom-up algorithm:
Two phases: 1) preprocessing of the rules, 2) matching

Good news:

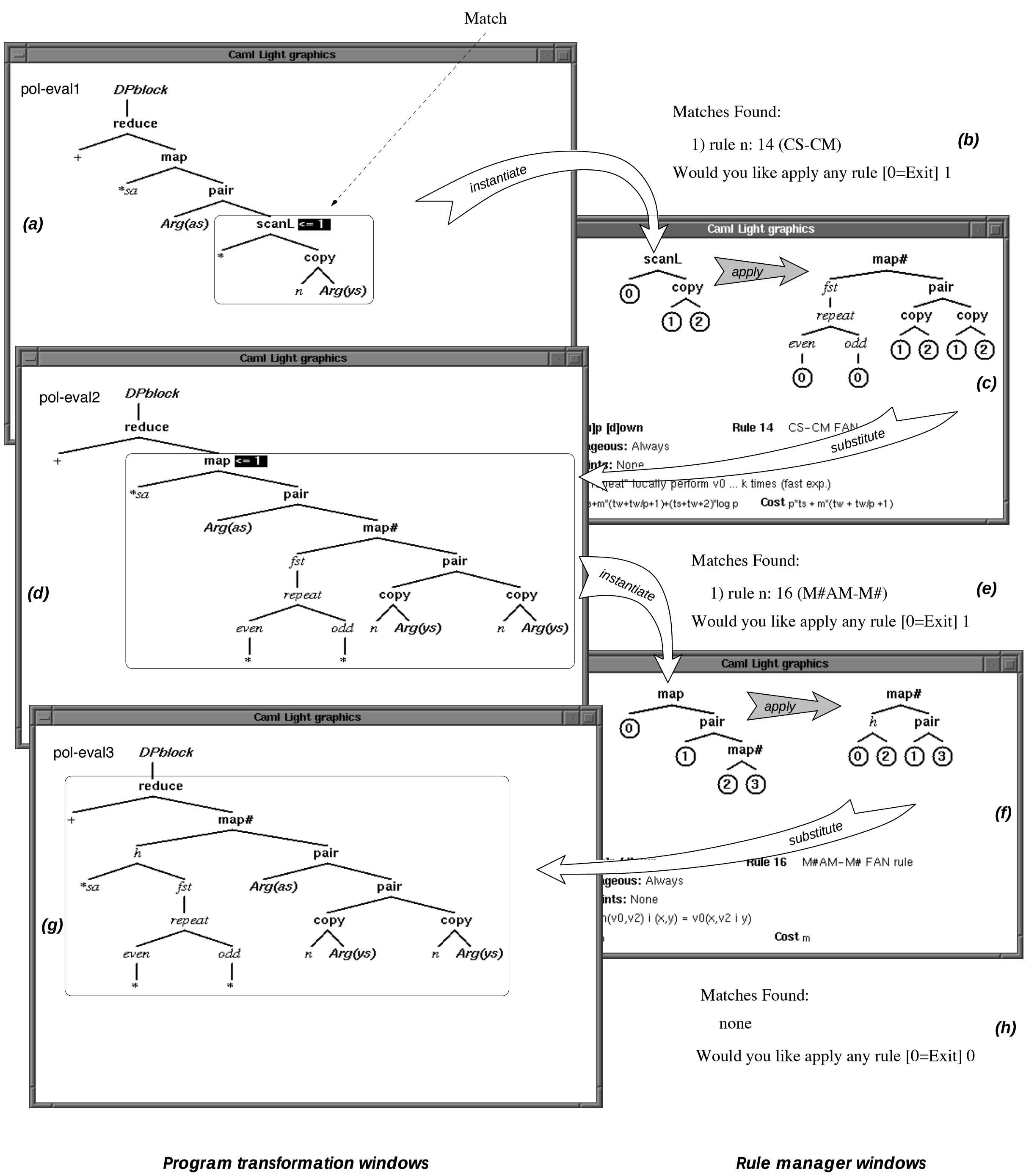
- **The preprocessing phase have to be repeated only if either the rules or the language have been changed**
- **Matching really fast (even in practice): a single traversal of T**

Bad news:

- **The preprocessing phase may be expensive**
- **but, it is fast for a broad class of pattern sets**
- **Details in the paper**



Already Implemented !
(5) Back End
(4) Tranformation Engine
rebuild TL program from internal representation
pattern matching
list of matched rules
matches presentation and user dialog
the rule to apply
tree tranformation
User Interaction
dependence tree (program representation)
dependence tree generation
program PT & DFG
TL program
scanner / parser
data dependence analyzer
TL rules
rule list PTs & DFGs
list of dependence pattern trees (lhs of rules representation)
pattern forest generation
pattern sorting (by subsumption)
subsumption graph
driving table generation
driving table
pattern library file
TL program
(3) Rule Manager
(2) Normalization
(1) Front End
TL spec.
independent from TL
TL specific
customizable via instantiation of the ADT
build by LEX & YACC



Match
Caml Light graphics
pol-eval1
DPblock
reduce
map
pair
scanL <= 1
copy
Arg(as)
Arg(ys)
(a)
Matches Found:
1) rule n: 14 (CS-CM)
Would you like apply any rule [0=Exit] 1
(b)
instantiate
apply
map#
fst
repeat
even
odd
Rule 14 CS-CM FAN
substitute
Cost p*ts + m*(tw + tw/p +1)
(c)
pol-eval2
map <= 1
(d)
Matches Found:
1) rule n: 16 (M#AM-M#)
Would you like apply any rule [0=Exit] 1
(e)
Rule 16 M#AM-M# FAN rule
Cost m
(f)
pol-eval3
(g)
Matches Found:
none
Would you like apply any rule [0=Exit] 0
(h)
Program transformation windows
Rule manager windows

MSS example in Skel-BSP (9 out of 20 formulations)

```
pipe.mss {
  map pair,
  scanL Op+,
  map P1,
  reduce max }
```

farm i/e

```
pipe.mss {
  map pair,
  farm(scanL Op+),
  map P1,
  reduce max }
```

farm i/e

```
pipe.mss {
  map pair,
  farm(scanL Op+),
  farm(map P1),
  reduce max }
```

pipe→comp

farm i/e

farm i/e

```
comp.mss (out r, in x) {
  y=map pair x,
  s=scanL Op+ y,
  v=map P1 s,
  r=reduce max v}
```

```
pipe.mss {
  map pair,
  scanL Op+,
  farm(map P1),
  reduce max }
```

farm i/e

```
pipe.mss {
  map pair,
  scanL Op+,
  farm(map P1),
  farm(reduce max) }
```

SAR–ARA

```
comp.mss (out r, in x) {
  a=map pair x,
  b=map pair a,
  c=reduce
    Op3(max,Op+) b,
  d=map P1 c,
  r=map P1 d }
```

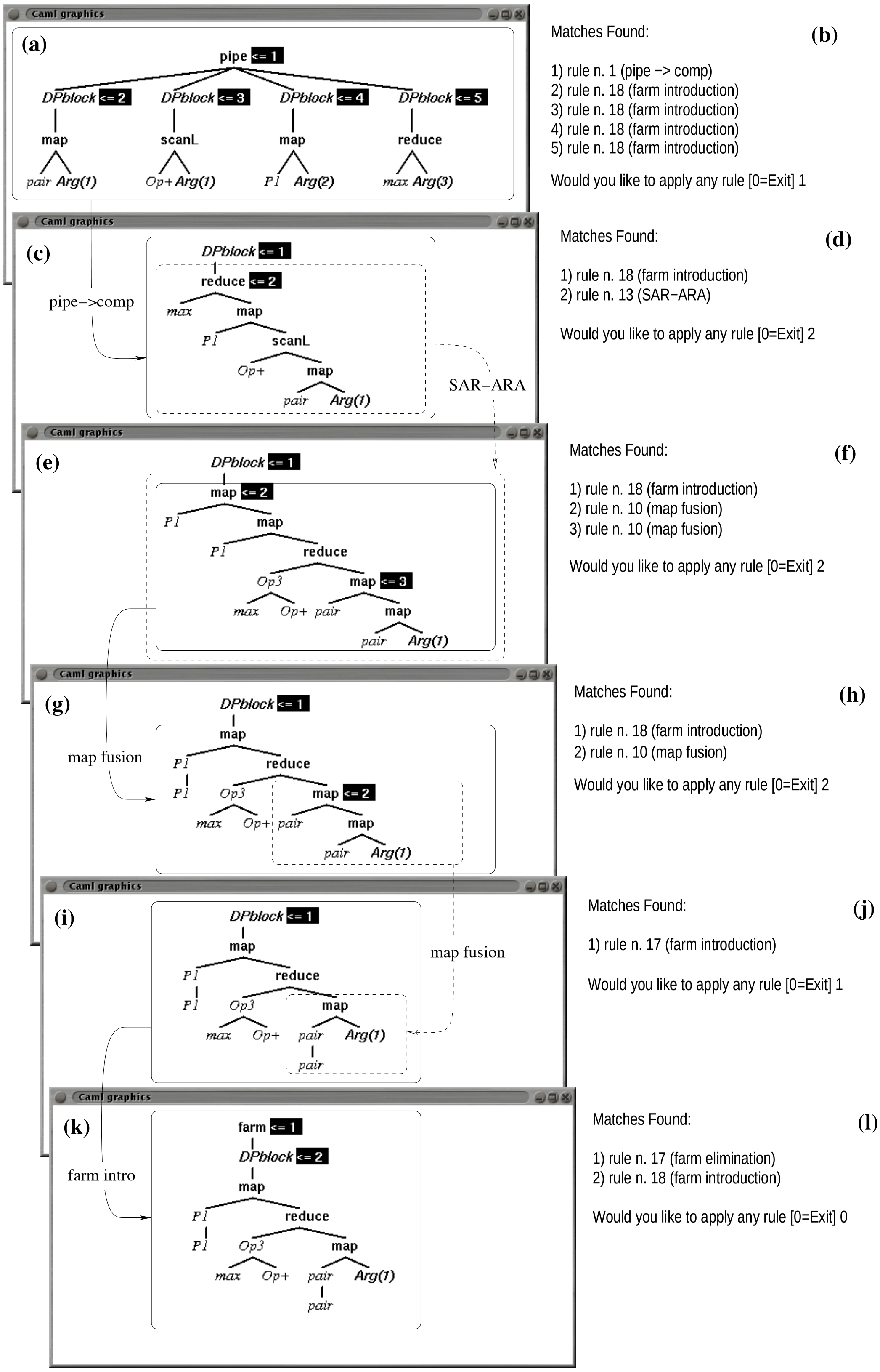
map fusion

map fusion

```
comp.mss (out r, in x) {
  a=map (pair ∘ pair) x,
  b=reduce
    Op3(max,Op+) a,
  r=map (P1 ∘ P1) b }
```

farm i/e

```
farm.mss (
  comp (out r, in x) {
    a=map (pair ∘ pair) x,
    b=reduce
      Op3(max,Op+) a,
    r=map (P1 ∘ P1) b })
```


(a)
pipe <= 1
DPblock <= 2
DPblock <= 3
DPblock <= 4
DPblock <= 5
map
scanL
map
reduce
pair Arg(1)
Op+ Arg(1)
P1 Arg(2)
max Arg(3)
Matches Found:
(b)
1) rule n. 1 (pipe −> comp)
2) rule n. 18 (farm introduction)
3) rule n. 18 (farm introduction)
4) rule n. 18 (farm introduction)
5) rule n. 18 (farm introduction)
Would you like to apply any rule [0=Exit] 1
(c)
pipe−>comp
DPblock <= 1
reduce <= 2
max map
P1 scanL
Op+ map
pair Arg(1)
SAR−ARA
Matches Found:
(d)
1) rule n. 18 (farm introduction)
2) rule n. 13 (SAR−ARA)
Would you like to apply any rule [0=Exit] 2
(e)
DPblock <= 1
map <= 2
P1 map
P1 reduce
Op3 map <= 3
max Op+ pair map
pair Arg(1)
Matches Found:
(f)
1) rule n. 18 (farm introduction)
2) rule n. 10 (map fusion)
3) rule n. 10 (map fusion)
Would you like to apply any rule [0=Exit] 2
(g)
map fusion
DPblock <= 1
map
P1 reduce
P1 Op3 map <= 2
max Op+ pair map
pair Arg(1)
Matches Found:
(h)
1) rule n. 18 (farm introduction)
2) rule n. 10 (map fusion)
Would you like to apply any rule [0=Exit] 2
(i)
DPblock <= 1
map
P1 reduce
P1 Op3 map
max Op+ pair Arg(1)
pair
map fusion
Matches Found:
(j)
1) rule n. 17 (farm introduction)
Would you like to apply any rule [0=Exit] 1
(k)
farm intro
farm <= 1
DPblock <= 2
map
P1 reduce
P1 Op3 map
max Op+ pair Arg(1)
pair
Matches Found:
(l)
1) rule n. 17 (farm elimination)
2) rule n. 18 (farm introduction)
Would you like to apply any rule [0=Exit] 0

Implementation

- **Prototype in `Ocaml 2.02` (about 2000 lines of code)**
- **Tested under Windows'98 and Linux RH6.2**
- **Tested over 2 target languages and about 20 rules**
- **The implementation (except graphical interface) is based on a single ADT which describes the dependence tree and the function working on it**
- **The implementation can handle many Target Languages via instantiation of the ADT**

Discussion

1. **Why the tool is interactive ?**

   → Because the rewriting calculus of TL, in the general case, is not confluent in performance and the solution space (may) grow exponentially with the number rules

2. **Does the tool make any decision about the rule to apply ?**

   → No. But, it can be extended with your own heuristics, if you have them

   → Currently Meta optimises Skel-BSP data-parallel-free programs with a standard sequence of rules. Such "Normal Form" is proved to be the fastest among the semantic-equivalent formulations that can be reached with these rules (Aldinucci, Danelutto. IASTED PDCS'99, Boston, USA)

3. **Does the tool really optimise the program ?**

   → It really depend on the TL, the set of rule and the cost calculus, not on the tool. The tool make you happy playing with your new skeleton language, that's it.

Conclusions

Meta implements a (meta) rewriting system for skeleton-based languages

- It is independent from the target language and the rules
- It only requires a three-tier language schema
- Can be equipped with heuristics to make decisions on the rule to apply
- It is already implemented on a platform-independent language (Ocaml)
- It has a (simple) graphical interface and it is fast
- Is is easy to modify and to extend
- ...
- **It is free !**

FAN rules (1)

Rule SR-ARA

b = reduce $Op2$ (scanL $Op1$ a)
b = proj1 (reduce $Op3$ (pair (a,a)))
If $Op1$ distributes forward over $Op2$
$(a_1, b_1)\, Op3\, (a_2, b_2) \; = \; (a_1\, Op2\, (b_1\, Op1\, a_2),\, b_1\, Op2\, b_2)$

Rule AR-RA

b = reduce $Op1$ (proj1 a)
b = proj1 (reduce $Op2$ a)
$(a_1, b_1)\, Op2\, (a_2, b_2) \; = \; (a_1\, Op1\, a_2,\, b_1\, Op1\, b_2)$

Rule SAR-ARA

c = reduce $Op2$ (proj1 (scanL $Op1$ a))
c = proj1 (proj1 (reduce $Op3$ (pair(a,a))))
If $Op1$ distributes forward over $Op4$
$(a_1, b_1)\, Op3\, (a_2, b_2) \; = \; (a_1\, Op4\, (b_1\, Op1\, a_2)\,,\, b_1\, Op4\, b_2)$ $(a_1, b_1)\, Op4\, (a_2, b_2) \; = \; (a_1\, Op2\, a_2,\, b_1\, Op2\, b_2)$

Rule CS-CM

b = scanL Op (copy n a)
b = map$_{\#}$ f (copy n a)
$f\ i\ x\ = fst(repeat\ i\ (x, x))$ $repeat\ \ k\ x = \text{if } k = 0 \text{ then } x \text{ else } repeat\ (k \text{ div } 2)\ (\text{if } (k \bmod 2 = 0) \text{ then } e\ x \text{ else } o\ x)$ $e(t, u) = (t, u\ Op\ u),\ o(t, u) = (t\ Op\ u, u\ Op\ u)$

FAN rules (2)

Rule $M_{\#}$M-$M_{\#}$

b = map$_{\#}$ f a c = map g b
c = map$_{\#}$ h a
$h\ i\ x = g\ (f\ i\ x)$

Rule $M_{\#}$AM-$M_{\#}$

b = map$_{\#}$ f a c = map g (pair (d,b))
c = map$_{\#}$ h (pair (d,a))
$h\ i\ (x,y) = g\ (x,\ f\ i\ y)$

FAN Operation	Time required
map f x	$m * t_f$
proj1 x	0
pair (x,y)	$2 * m * t_{copy}$
copy n x	$p * t_s + \frac{m*(p-1)}{p} * t_w$
part (r, s) x	$2 * t_s + (r + s) * t_w$
reduce $(\oplus)$ x	$m * t_{\oplus} + \log p * (t_s + t_w + t_{\oplus})$
scanL $(\oplus)$ x	$2 * m * t_{\oplus} + \log p * (t_s + t_w + 2 * t_{\oplus})$

Rule	Time left hand side	Time right hand side	Improves if
SR-ARA	$3 * m + \log p * (2 * (t_s + t_w) + 3))$	$2 * m + \log p * (t_s + 2 * t_w + 2)$	always
AR-RA	$m + \log p * (t_s + t_w + 1)$	$2 * m + \log p * (t_s + 2 * (t_w + 1))$	never
SAR-ARA	$3 * m + \log p * (2 * (t_s + t_w) + 3)$	$5 * m + \log p * (t_s + t_w + 1)$	$(t_s + t_w + 2) * \log p > 2m$
CS-CM	$p * t_s + m * (t_w + \frac{t_w}{p} + 1)$ $+ \log p * (t_s + t_w + 2)$	$p * t_s + m * (t_w + \frac{t_w}{p} + 1)$	always
$M_{\#}$M-$M_{\#}$	$2 * m$	m	always
$M_{\#}$AM-$M_{\#}$	$4 * m$	m	always