A Segment-Swapping Approach for Executing Trapped Computations

Pablo Chico de Guzman[1] Amadeo Casas[2] Manuel Carro[1,3]
Manuel V. Hermenegildo[1,3]

[1]School of Computer Science, Technical University of Madrid, Spain
[2]Samsung Research, USA
[3]IMDEA Software Institute, Spain

January 23, 2012

Logic programming and parallelism

- (C)LP is a very interesting framework for parallelism:
  - Program closer to problem.
  - Notion of control provides more flexibility.
  - *Central parallelization challenges still there:*
    - dependencies, heap, pointers, aliasing, ...

    but cleaner semantic setting (e.g., pointers exist, but are declarative).
- Two main types of parallelism:
  - *Or-Parallelism*: explore in parallel **alternative search paths**.
  - *And-Parallelism*: execute in parallel **parts of each execution path** (statements, procedure calls, ...)
    - Traditional parallelism: parbegin-parend, loop parallelization, divide-and-conquer, etc.

Background: parallel execution and independence

- **Correctness:** same results as sequential execution.
- **Efficiency:** ideal execution time $\leq$ than seq. program (no slowdown).

```
main :-                 |    p(X) :- X = [1,2,3].
     s1    p(X),        |
     s2    q(X),        |    q(X) :- X = [], large computation.
           write(X).    |    q(X) :- X = [1,2,3].
```

- Fundamental issue: `p` *affects* `q` (prunes its choices).
  - `q` ahead of `p` is *speculative* $\rightarrow$ **not independent**.

Background: parallel execution and independence

- **Correctness:** same results as sequential execution.
- **Efficiency:** ideal execution time $\leq$ than seq. program (no slowdown).

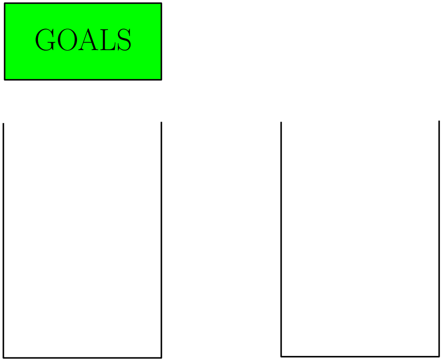
```
main :-                       p(X) :- X = [1,2,3].
    s1    p(X),
    s2    q(X),               q(X) :- X = [], large computation.
          write(X).           q(X) :- X = [1,2,3].
```

- Fundamental issue: `p` *affects* `q` (prunes its choices).
  - `q` ahead of `p` is *speculative* $\rightarrow$ **not independent**.
- We will focus herein on **independent** and-parallelism (IAP).
  - Many interesting issues in parallelization: dependency analysis (pointer sharing, determinacy, non-failure, ...), granularity/ovhd. analysis, ...

we assume program parallelized (using `&/2`: simple fork-join, nested).

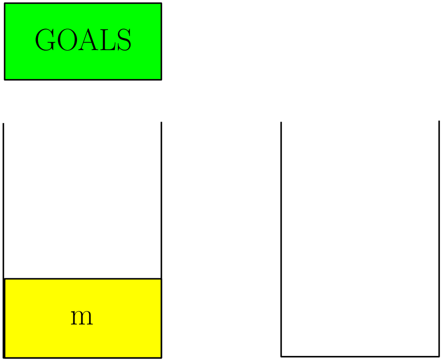
IAP: deterministic example

```
m :- a(X) & a(Y).

a(1).
```



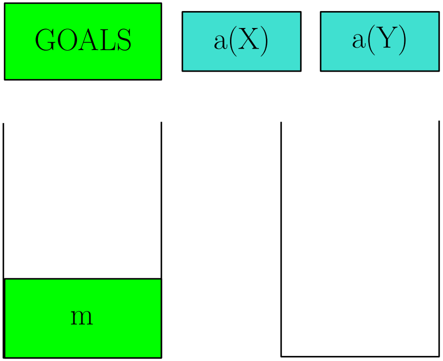
IAP: deterministic example

```
m :- a(X) & a(Y).

a(1).
```



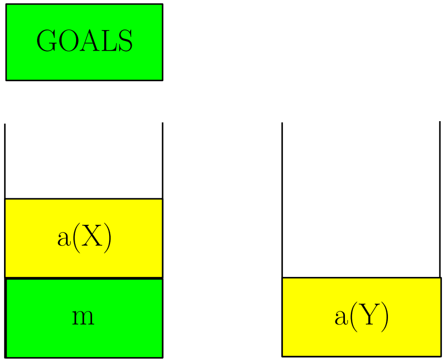
IAP: deterministic example

```
m :- a(X) & a(Y).

a(1).
```



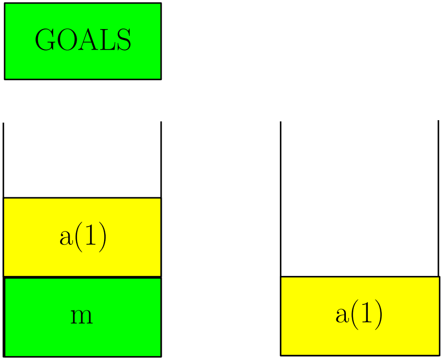
IAP: deterministic example

```
m :- a(X) & a(Y).

a(1).
```



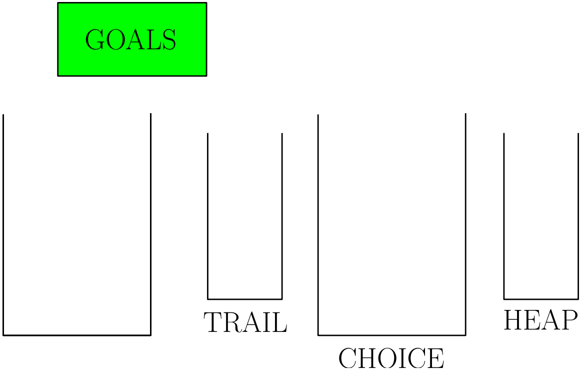
IAP: deterministic example

```
m :- a(X) & a(Y).

a(1).
```



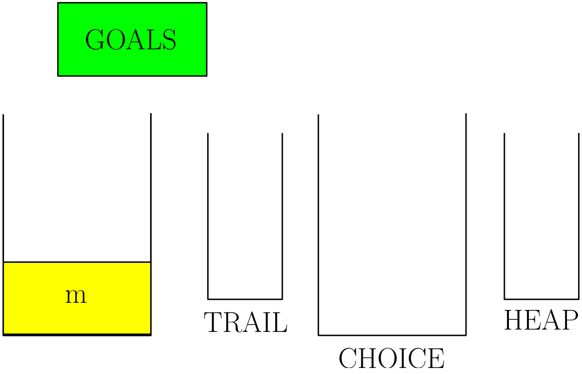
IAP: non-deterministic example

```
m :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



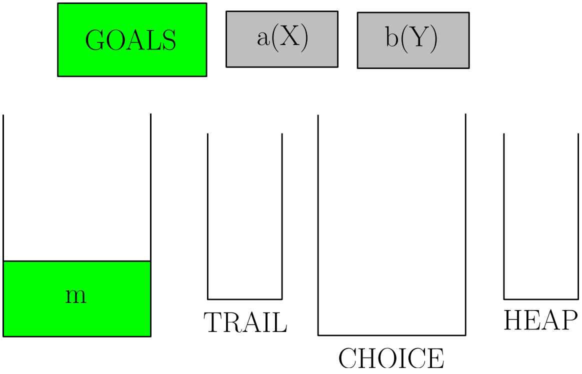
IAP: non-deterministic example

```
m :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



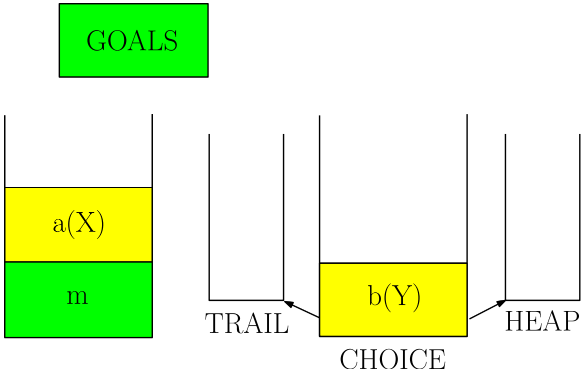
IAP: non-deterministic example

```
m :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



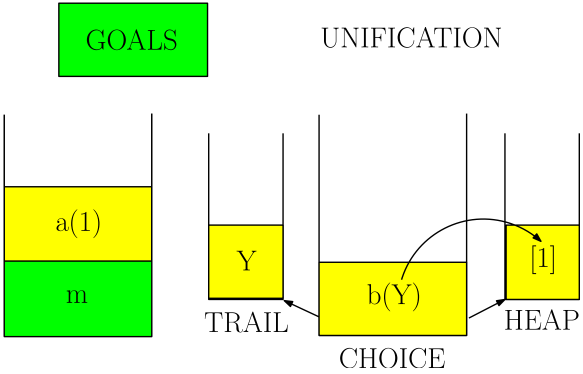
IAP: non-deterministic example

```
m :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



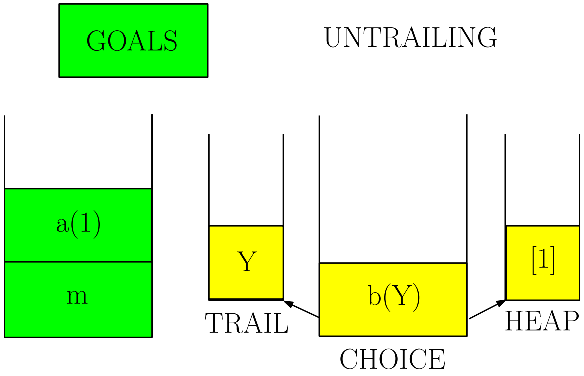
IAP: non-deterministic example

```
m :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



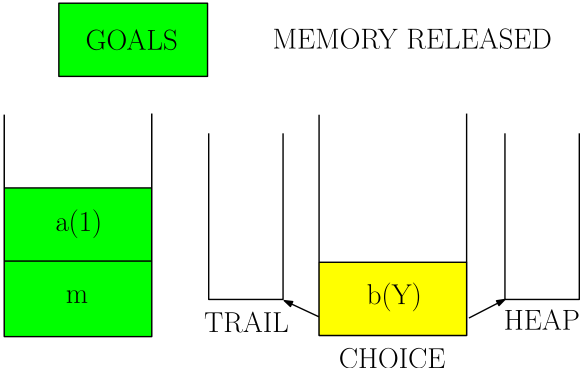
IAP: non-deterministic example

```
m :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



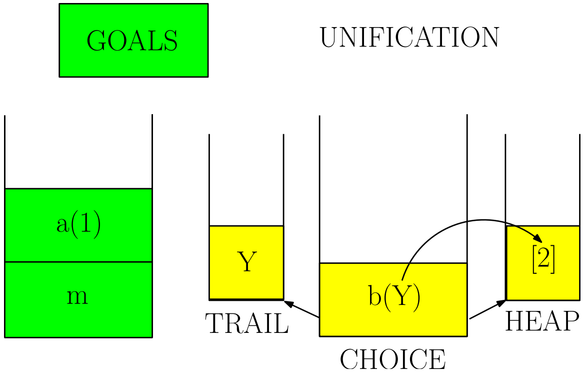
IAP: non-deterministic example

```
m :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



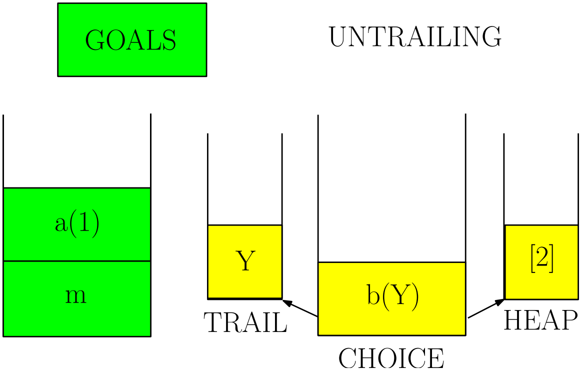
IAP: non-deterministic example

```
m :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



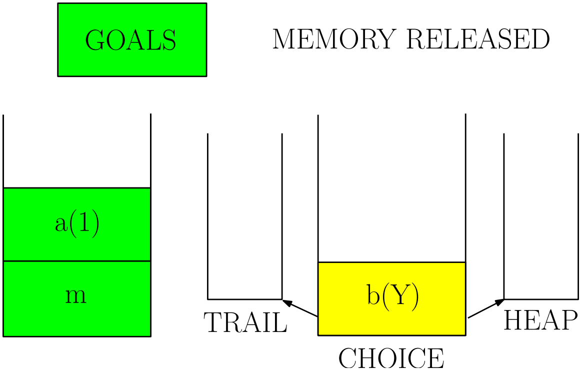
IAP: non-deterministic example

```
m :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



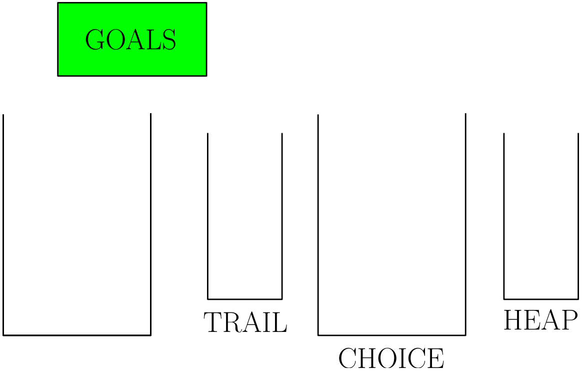
IAP: non-deterministic example

```
m :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



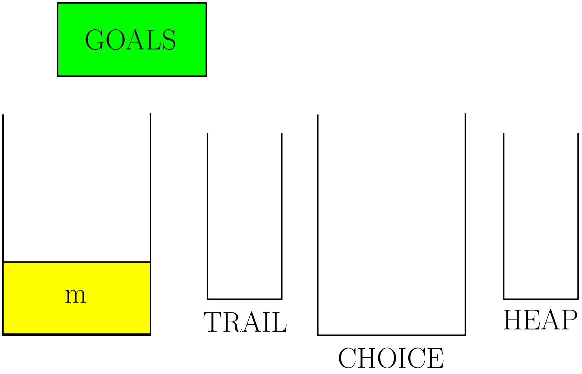
IAP: non-deterministic example with trapped goals

```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



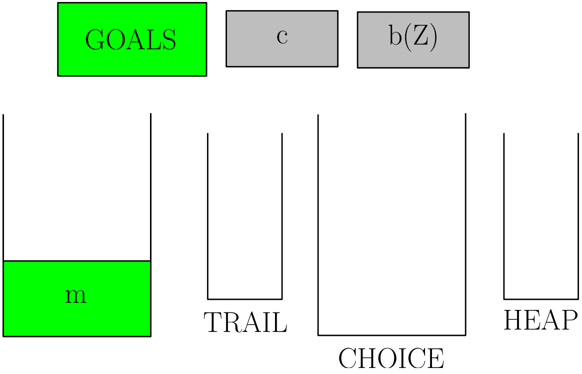
IAP: non-deterministic example with trapped goals

```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



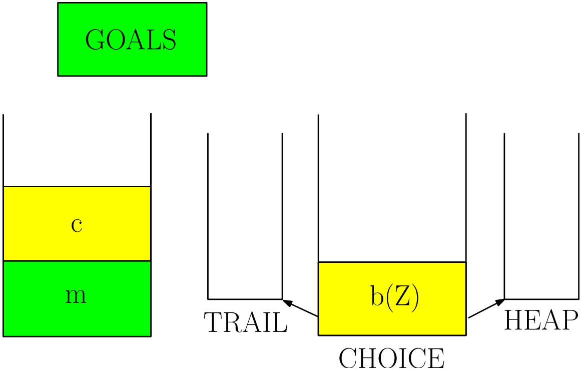
IAP: non-deterministic example with trapped goals

```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



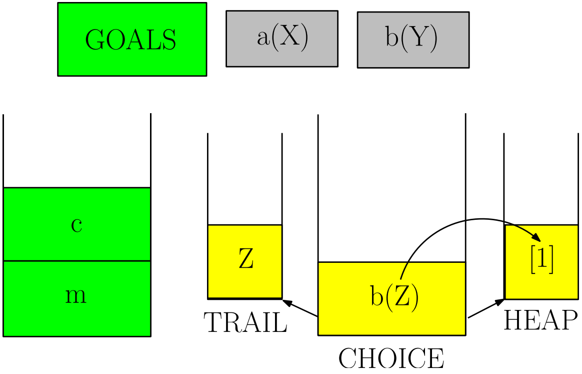
IAP: non-deterministic example with trapped goals

```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



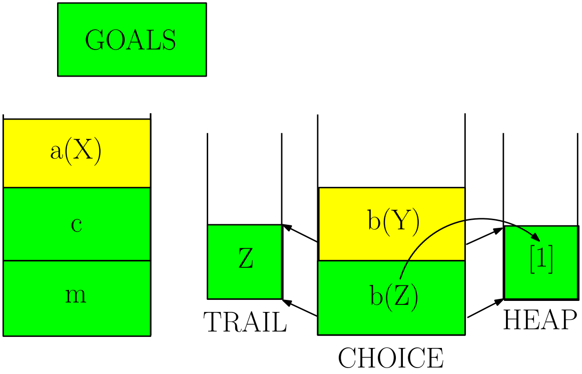
IAP: non-deterministic example with trapped goals

```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



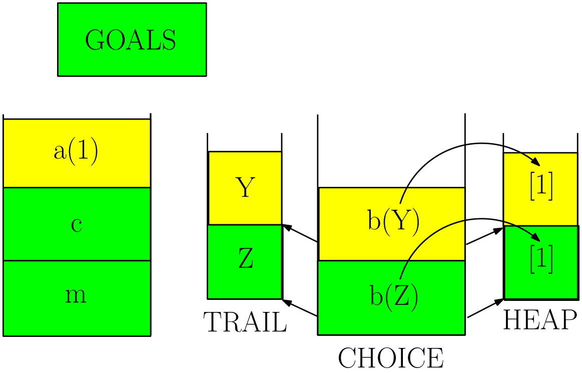
IAP: non-deterministic example with trapped goals

```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



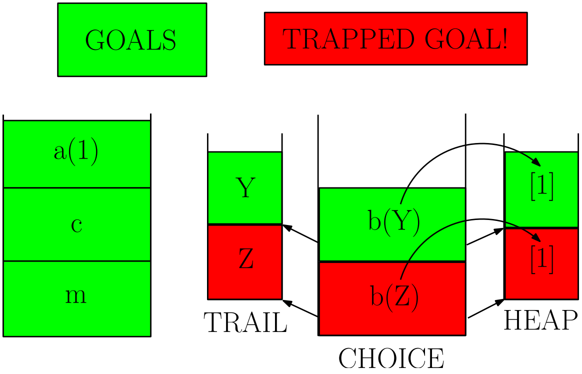
IAP: non-deterministic example with trapped goals

```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



IAP: non-deterministic example with trapped goals

```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



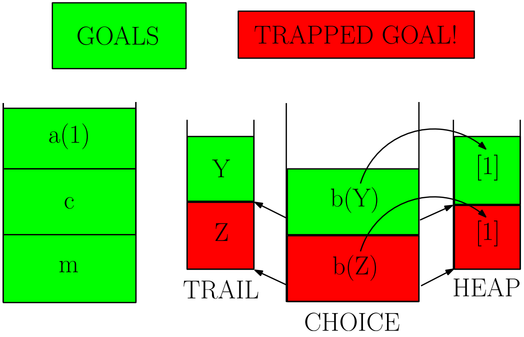
Solving the trapped goals problem

- Classical approaches:
  - Implement execution over discontiguous segments.
  - Avoid trapped goals through scheduling limitations.
- Our proposal: segment-swapping operation.

Dealing with trapped goals: discontiguous execution segments

- Large changes to the WAM implementation ⇒ overhead!
- Low-level WAM extensions need to be revisited.
- Difficult maintainance (more complex than the classical one).

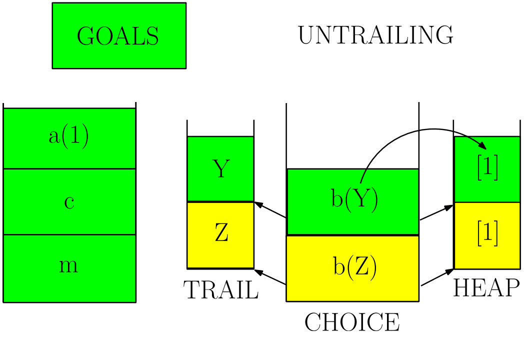
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals: discontiguous execution segments

- Large changes to the WAM implementation ⇒ overhead!
- Low-level WAM extensions need to be revisited.
- Difficult maintainance (more complex than the classical one).

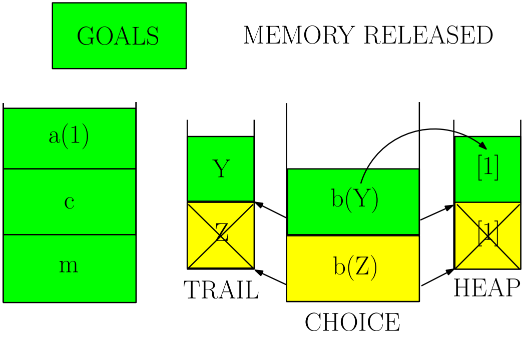
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals: discontiguous execution segments

- Large changes to the WAM implementation ⇒ overhead!
- Low-level WAM extensions need to be revisited.
- Difficult maintainance (more complex than the classical one).

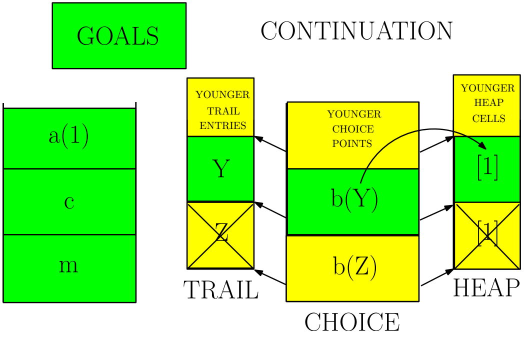
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals: discontiguous execution segments

- Large changes to the WAM implementation ⇒ overhead!
- Low-level WAM extensions need to be revisited.
- Difficult maintainance (more complex than the classical one).

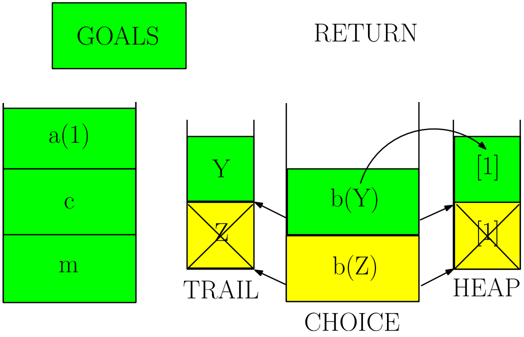
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals: discontiguous execution segments

- Large changes to the WAM implementation ⇒ overhead!
- Low-level WAM extensions need to be revisited.
- Difficult maintainance (more complex than the classical one).

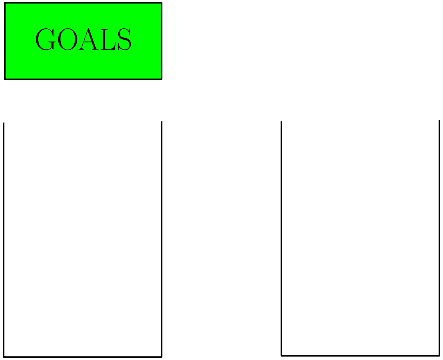
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals: imposing scheduling limitations

- Solution: do not select "older" goals if they will stack over "younger"
  - Precedence analysis.
    - Goal age: `a(X)` older than `b(Y)` older than `b(Z)`.
  - Less parallelism: `b(Y)` cannot be executed on top of `b(Z)`.

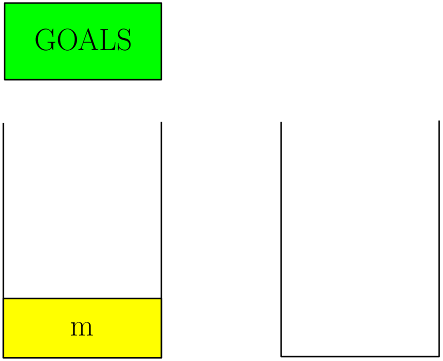
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals: imposing scheduling limitations

- Solution: do not select "older" goals if they will stack over "younger"
  - Precedence analysis.
    - Goal age: `a(X)` older than `b(Y)` older than `b(Z)`.
  - Less parallelism: `b(Y)` cannot be executed on top of `b(Z)`.

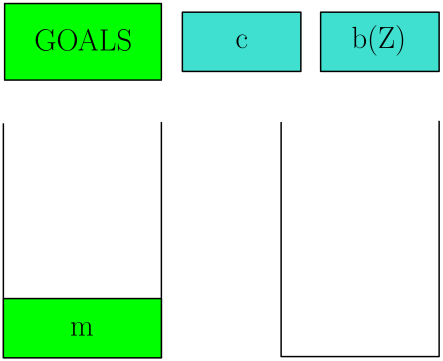
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals: imposing scheduling limitations

- Solution: do not select "older" goals if they will stack over "younger"
  - Precedence analysis.
    - Goal age: `a(X)` older than `b(Y)` older than `b(Z)`.
  - Less parallelism: `b(Y)` cannot be executed on top of `b(Z)`.

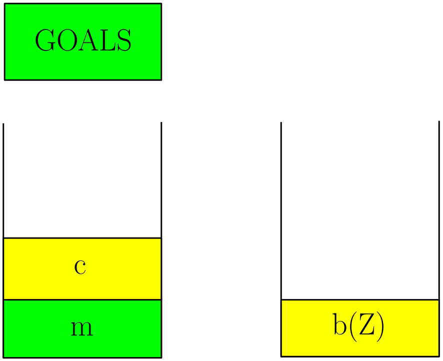
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals: imposing scheduling limitations

- Solution: do not select "older" goals if they will stack over "younger"
  - Precedence analysis.
    - Goal age: `a(X)` older than `b(Y)` older than `b(Z)`.
  - Less parallelism: `b(Y)` cannot be executed on top of `b(Z)`.

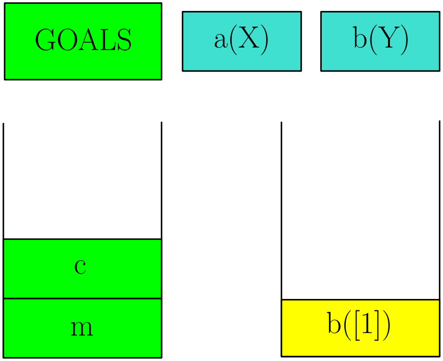
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals: imposing scheduling limitations

- Solution: do not select "older" goals if they will stack over "younger"
  - Precedence analysis.
    - Goal age: `a(X)` older than `b(Y)` older than `b(Z)`.
  - Less parallelism: `b(Y)` cannot be executed on top of `b(Z)`.

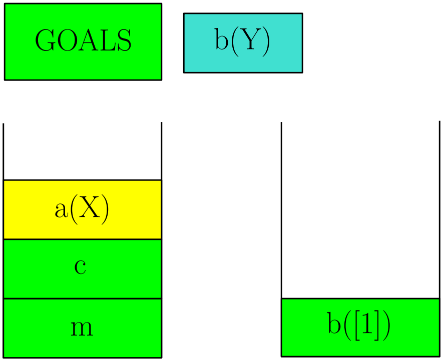
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals: imposing scheduling limitations

- Solution: do not select "older" goals if they will stack over "younger"
  - Precedence analysis.
    - Goal age: `a(X)` older than `b(Y)` older than `b(Z)`.
  - Less parallelism: `b(Y)` cannot be executed on top of `b(Z)`.

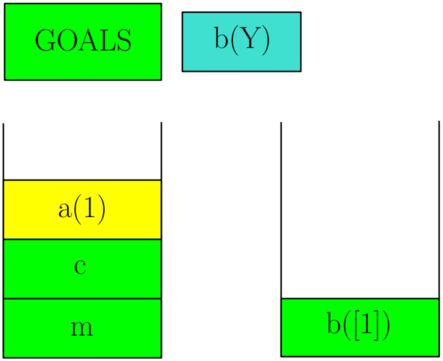
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals: imposing scheduling limitations

- Solution: do not select “older” goals if they will stack over “younger”
  - Precedence analysis.
    - Goal age: `a(X)` older than `b(Y)` older than `b(Z)`.
  - Less parallelism: `b(Y)` cannot be executed on top of `b(Z)`.

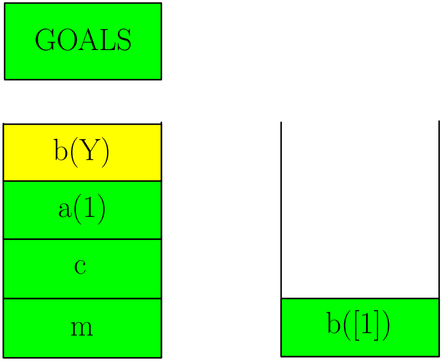
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals: imposing scheduling limitations

- Solution: do not select "older" goals if they will stack over "younger"
  - Precedence analysis.
    - Goal age: `a(X)` older than `b(Y)` older than `b(Z)`.
  - Less parallelism: `b(Y)` cannot be executed on top of `b(Z)`.

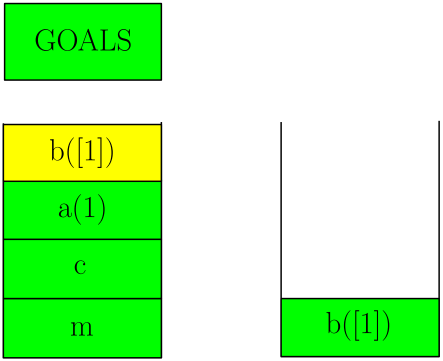
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals: imposing scheduling limitations

- Solution: do not select "older" goals if they will stack over "younger"
  - Precedence analysis.
    - Goal age: `a(X)` older than `b(Y)` older than `b(Z)`.
  - Less parallelism: `b(Y)` cannot be executed on top of `b(Z)`.

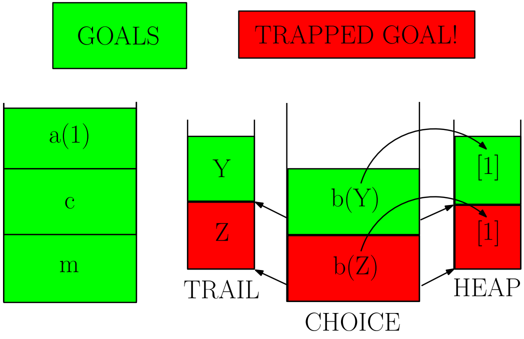
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals (our solution): segment-swapping approach

- Minimal changes to the WAM ⇒ sequential execution unaffected.
- Trapped goals allowed ⇒ potential parallelism unaffected.
- **Segment-swapping** of trapped goals to recover WAM invariants:
  - Complex operation, but it is local and modular.

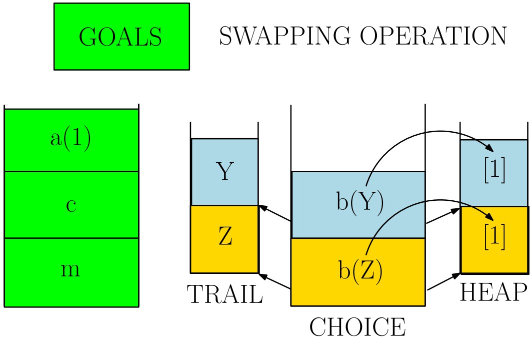
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals (our solution): segment-swapping approach

- Minimal changes to the WAM ⇒ sequential execution unaffected.
- Trapped goals allowed ⇒ potential parallelism unaffected.
- **Segment-swapping** of trapped goals to recover WAM invariants:
  - Complex operation, but it is local and modular.

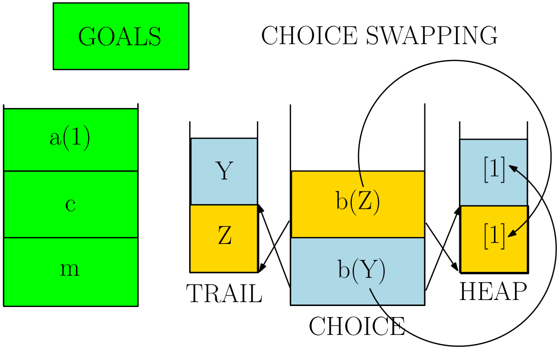
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals (our solution): segment-swapping approach

- Minimal changes to the WAM ⇒ sequential execution unaffected.
- Trapped goals allowed ⇒ potential parallelism unaffected.
- **Segment-swapping** of trapped goals to recover WAM invariants:
  - Complex operation, but it is local and modular.

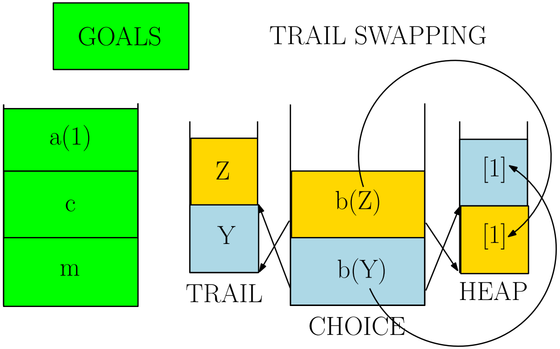
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals (our solution): segment-swapping approach

- Minimal changes to the WAM ⇒ sequential execution unaffected.
- Trapped goals allowed ⇒ potential parallelism unaffected.
- **Segment-swapping** of trapped goals to recover WAM invariants:
  - Complex operation, but it is local and modular.

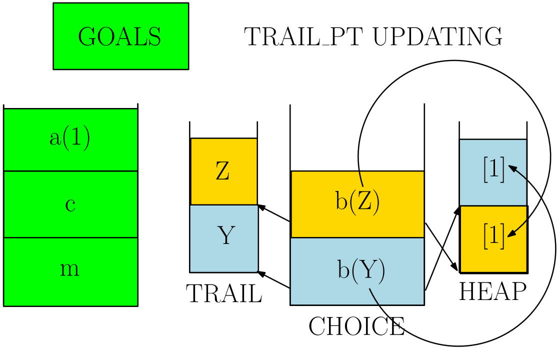
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals (our solution): segment-swapping approach

- Minimal changes to the WAM $\Rightarrow$ sequential execution unaffected.
- Trapped goals allowed $\Rightarrow$ potential parallelism unaffected.
- **Segment-swapping** of trapped goals to recover WAM invariants:
  - Complex operation, but it is local and modular.

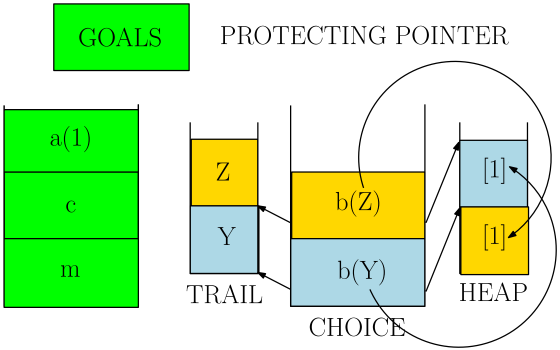
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals (our solution): segment-swapping approach

- Minimal changes to the WAM ⇒ sequential execution unaffected.
- Trapped goals allowed ⇒ potential parallelism unaffected.
- **Segment-swapping** of trapped goals to recover WAM invariants:
  - Complex operation, but it is local and modular.

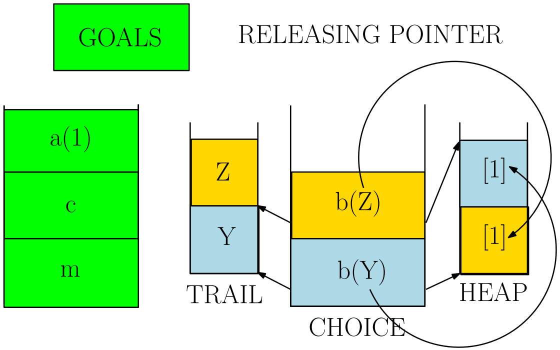
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals (our solution): segment-swapping approach

- Minimal changes to the WAM ⇒ sequential execution unaffected.
- Trapped goals allowed ⇒ potential parallelism unaffected.
- **Segment-swapping** of trapped goals to recover WAM invariants:
  - Complex operation, but it is local and modular.

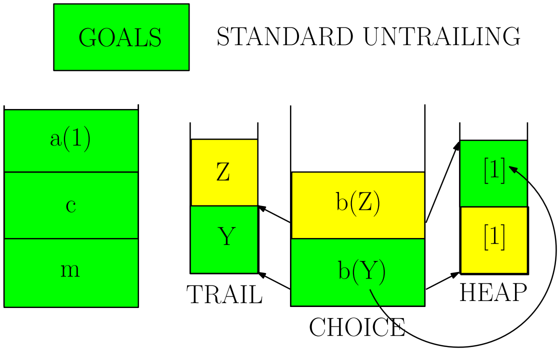
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals (our solution): segment-swapping approach

- Minimal changes to the WAM ⇒ sequential execution unaffected.
- Trapped goals allowed ⇒ potential parallelism unaffected.
- **Segment-swapping** of trapped goals to recover WAM invariants:
  - Complex operation, but it is local and modular.

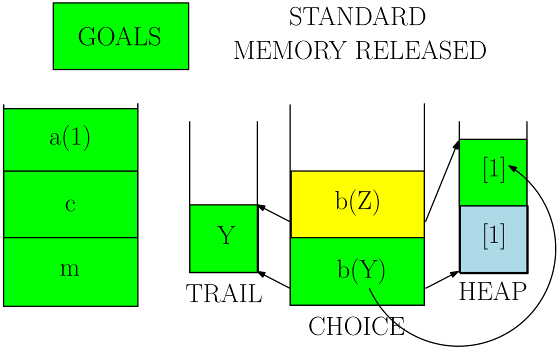
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals (our solution): segment-swapping approach

- Minimal changes to the WAM ⇒ sequential execution unaffected.
- Trapped goals allowed ⇒ potential parallelism unaffected.
- **Segment-swapping** of trapped goals to recover WAM invariants:
  - Complex operation, but it is local and modular.

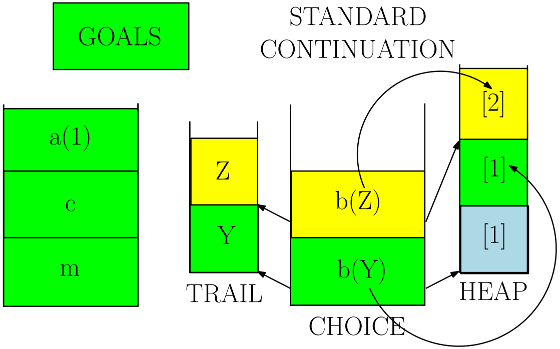
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals (our solution): segment-swapping approach

- Minimal changes to the WAM ⇒ sequential execution unaffected.
- Trapped goals allowed ⇒ potential parallelism unaffected.
- **Segment-swapping** of trapped goals to recover WAM invariants:
  - Complex operation, but it is local and modular.

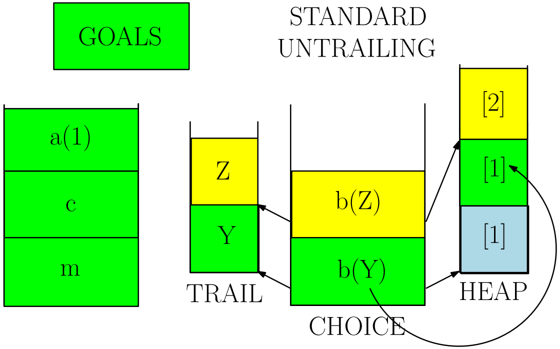
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals (our solution): segment-swapping approach

- Minimal changes to the WAM ⇒ sequential execution unaffected.
- Trapped goals allowed ⇒ potential parallelism unaffected.
- **Segment-swapping** of trapped goals to recover WAM invariants:
  - Complex operation, but it is local and modular.

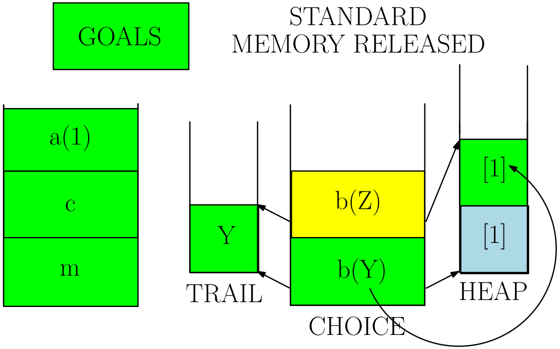
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals (our solution): segment-swapping approach

- Minimal changes to the WAM ⇒ sequential execution unaffected.
- Trapped goals allowed ⇒ potential parallelism unaffected.
- **Segment-swapping** of trapped goals to recover WAM invariants:
  - Complex operation, but it is local and modular.

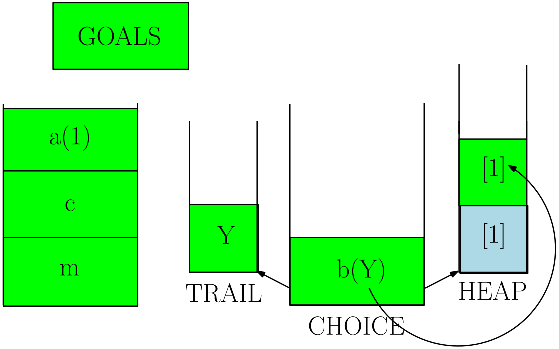
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals (our solution): segment-swapping approach

- Minimal changes to the WAM ⇒ sequential execution unaffected.
- Trapped goals allowed ⇒ potential parallelism unaffected.
- **Segment-swapping** of trapped goals to recover WAM invariants:
  - Complex operation, but it is local and modular.

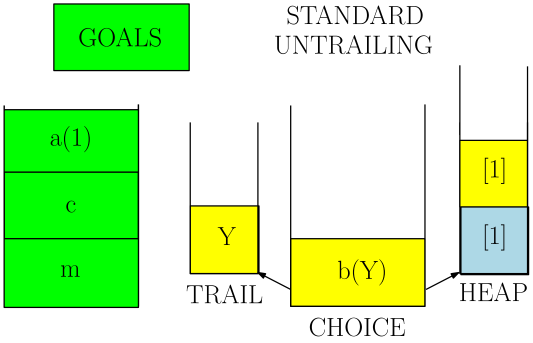
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Dealing with trapped goals (our solution): segment-swapping approach

- Minimal changes to the WAM ⇒ sequential execution unaffected.
- Trapped goals allowed ⇒ potential parallelism unaffected.
- **Segment-swapping** of trapped goals to recover WAM invariants:
  - Complex operation, but it is local and modular.

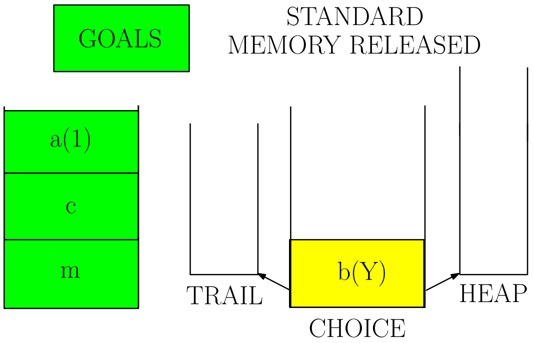
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



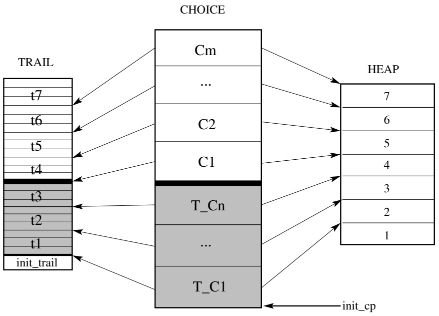
Dealing with trapped goals (our solution): segment-swapping approach

- Minimal changes to the WAM ⇒ sequential execution unaffected.
- Trapped goals allowed ⇒ potential parallelism unaffected.
- **Segment-swapping** of trapped goals to recover WAM invariants:
  - Complex operation, but it is local and modular.

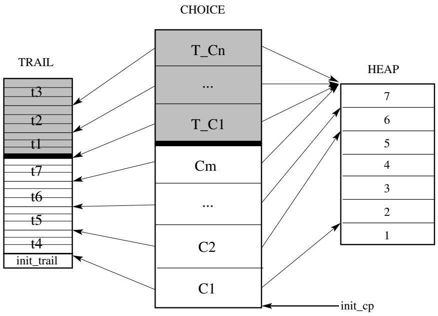
```
m :- c & b(Z).

c :- a(X) & b(Y).

a(1).

b([1]).
b([2]).
```



Segment-swapping operation: high level description



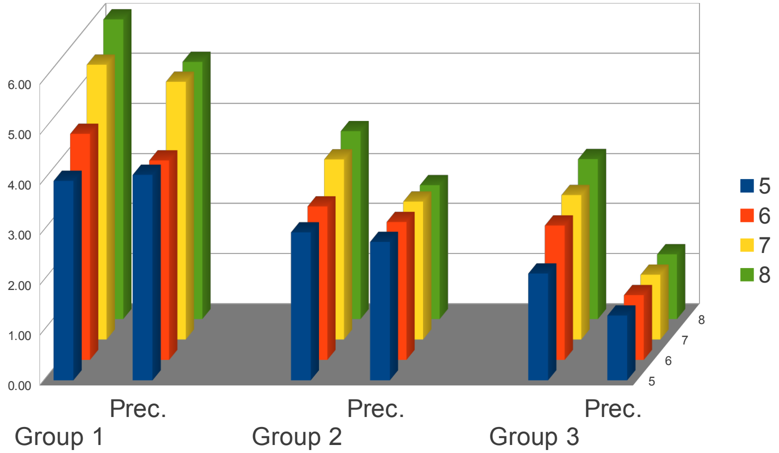
(a) Trapped computation



(b) After segment-swapping operation

Impact of restrictions on precedence

Speedups: swapping vs. precedence



- Precedence restriction limits parallelism and affects performance.

Segment-swapping overhead

	5		6		7		8	
	Freq	**Lost**	**Freq**	**Lost**	**Freq**	**Lost**	**Freq**	**Lost**
fft	0.05	0.00	0.09	0.00	0.10	0.00	0.15	0.00
fibo	0.03	0.00	0.03	0.01	0.04	0.01	0.06	0.02
han	0.02	0.00	0.02	0.00	0.04	0.00	0.04	0.00
han_dl	0.03	0.05	0.04	0.05	0.03	0.05	0.04	0.07
mmat	0.00	0.00	0.00	0.00	0.02	0.01	0.00	0.00
pal	0.00	0.00	0.02	0.00	0.01	0.00	0.03	0.00
qs	0.07	0.00	0.11	0.00	0.06	0.00	0.09	0.00
qs_dl	0.06	0.00	0.13	0.01	0.11	0.01	0.11	0.01
iqs	0.18	0.02	0.26	0.02	0.27	0.02	0.35	0.03
iqs_dl	0.15	0.02	0.20	0.02	0.28	0.03	0.36	0.03
tak	0.01	0.00	0.13	0.00	0.07	0.00	0.05	0.00
qs_nd	0.14	0.00	0.21	0.00	0.33	0.01	0.39	0.01

- **Freq**: Trapped backtracking fraction vs. total backtrackings.
- **Lost**: Swapping execution time fraction vs. total execution time.

Conclusions

- Previous solutions to trapped goal problem not ideal:
  - Precedence analysis limits parallelism.
  - Discontiguous execution segments is very complex.
- We propose segment-swapping approach:
  - Can be made to not affect sequential execution performance.
  - Easier to maintain due to its locality and modularity.
  - Good parallel performance.

Segment-swapping vs. precedence analysis

	5		6		7		8	
	SS	**Prec**	**SS**	**Prec**	**SS**	**Prec**	**SS**	**Prec**
fft	2.68	2.69	2.87	2.68	2.97	2.67	3.02	2.68
fibo	3.98	4.10	4.51	3.98	5.48	5.14	5.98	5.13
han	3.24	3.24	3.41	3.21	3.74	3.23	4.11	3.42
han_dl	2.95	2.76	3.06	2.75	3.59	2.75	3.75	2.67
mmat	3.72	3.67	4.35	4.11	4.97	4.68	5.63	5.42
pal	3.18	1.82	3.29	3.17	3.60	3.18	3.96	3.03
qs	2.69	2.29	2.84	2.29	3.42	2.29	3.73	2.29
qs_dl	2.65	2.19	3.13	2.19	3.25	2.19	3.32	2.16
iqs	2.27	1.33	2.43	1.33	2.80	1.33	3.02	1.33
iqs_dl	2.13	1.29	2.68	1.29	2.88	1.29	3.19	1.29
tak	3.50	3.50	3.54	3.54	4.47	3.54	4.57	3.74
qs_nd	1.93	1.59	2.01	1.59	2.34	1.59	2.54	1.66

- **SS**: segment-swapping approach.
- **Prec**: scheduling limitation approach.