Resource Analysis of Distributed and Concurrent Programs

Elvira Albert
Complutense University of Madrid (Spain)

DICE-FOPARA 2017

April 22-23, 2017, Uppsala, Sweden

The aim of RESOURCE ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P

The aim of ~~static~~ RESOURCE ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P ***without actually executing P*** *static*

The aim of RESOURCE ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P ***without actually executing*** ***any*** P

automatic *static*

The aim of RESOURCE ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P ***without actually executing P*** ~~without~~ ***any***

automatic *static*

The aim of ~~RESOURCE~~ ANALYSIS is to bound the resource consumption (aka cost) of executing a given program *P* ***wi~~thout~~*** *any* ***actually executing*** *P*

automatic *static*

The aim of RESOURCE ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P ***without actually executing P*** ***any***

- Upper Bounds (***worst case***)
- Lower Bounds (***best case***)

automatic
static

The aim of RESOURCE ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P *without actually executing* *any* P

automatic static

The aim of RESOURCE ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P *without ~~actually~~ any executing P*

- Execution steps
- Visits to p
- Memory

automatic *static*

The aim of RESOURCE ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P *without actually executing* *any* P

- Execution steps
- Visits to p
- Memory

non-cumulative

automatic static

The aim of RESOURCE ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P *without actually executing* any *P*

- Execution steps
- Visits to p
- Memory
- Time? Energy?

automatic *static*

The aim of RESOURCE ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P ~~*without*~~ *any* *actually executing* P

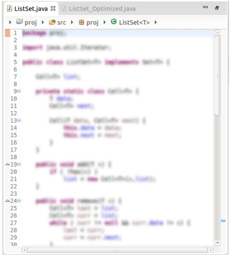
- Execution steps
- Visits to p
- Memory
- ~~Time? Energy?~~ *platform dependent*

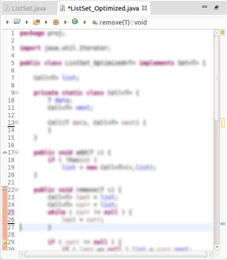
WHAT IS IT USEFUL FOR?

- Traditional applications

WHAT IS IT USEFUL FOR?

- Traditional applications
  - Program optimization

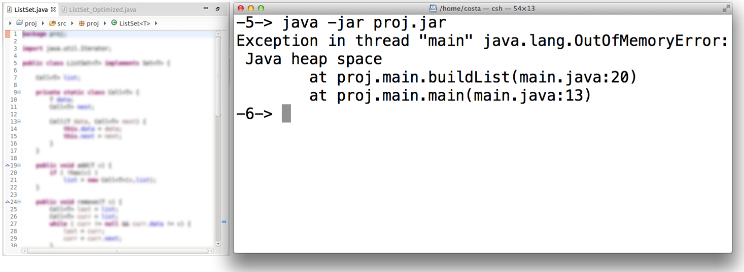




$$\mathcal{U}_1 \preceq \mathcal{U}_2$$

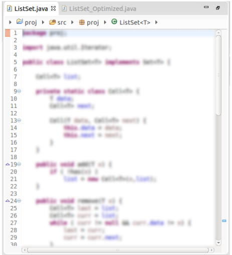
WHAT IS IT USEFUL FOR?

- Traditional applications
  - Program optimization
  - Verification: resource guarantees



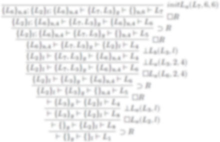
WHAT IS IT USEFUL FOR?

- Traditional applications
  - Program optimization
  - Verification: resource guarantees
  - Certification: resource usage certificates



$$\mathcal{U} = 2n^2 + 3n$$

Proof:

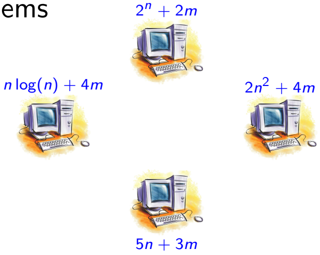


WHAT IS IT USEFUL FOR?

- Traditional applications
  - Program optimization
  - Verification: resource guarantees
  - Certification: resource usage certificates
- New applications for distributed systems

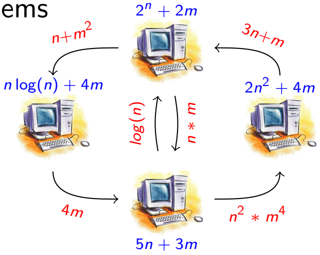
WHAT IS IT USEFUL FOR?

- Traditional applications
  - Program optimization
  - Verification: resource guarantees
  - Certification: resource usage certificates
- New applications for distributed systems
  - Load balance



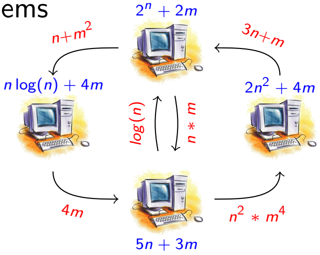
WHAT IS IT USEFUL FOR?

- Traditional applications
  - Program optimization
  - Verification: resource guarantees
  - Certification: resource usage certificates
- New applications for distributed systems
  - Load balance
  - Amount of data transmitted



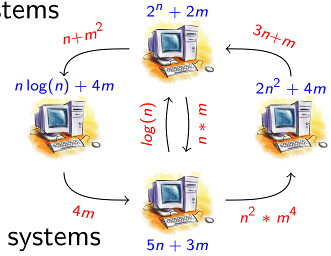
WHAT IS IT USEFUL FOR?

- Traditional applications
  - Program optimization
  - Verification: resource guarantees
  - Certification: resource usage certificates
- New applications for distributed systems
  - Load balance
  - Amount of data transmitted
  - Explotation of parallelism



WHAT IS IT USEFUL FOR?

- Traditional applications
  - Program optimization
  - Verification: resource guarantees
  - Certification: resource usage certificates
- New applications for distributed systems
  - Load balance
  - Amount of data transmitted
  - Explotation of parallelism
  - Model and dimension distributed systems



Plan of the talk

- Part 1: Cost analysis in sequential programs
  - Generation of cost relations
  - Inference of upper bounds

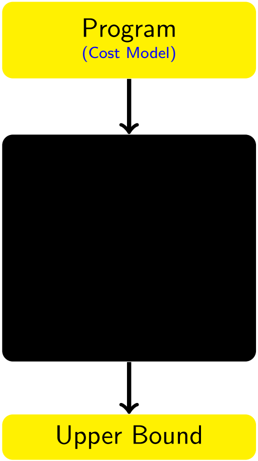
Plan of the talk

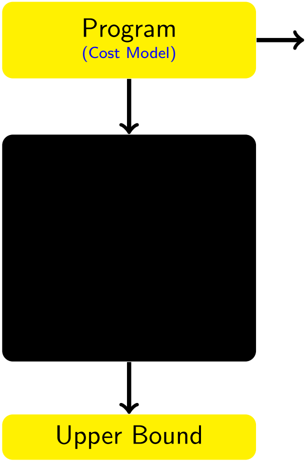
- Part 1: Cost analysis in sequential programs
  - Generation of cost relations
  - Inference of upper bounds
- Part 2: Cost analysis in concurrent programs
  - Loops with concurrent interleavings
  - May-happen-in-parallel analysis
  - Rely-guarantee reasoning

Plan of the talk

- Part 1: Cost analysis in sequential programs
  - Generation of cost relations
  - Inference of upper bounds
- Part 2: Cost analysis in concurrent programs
  - Loops with concurrent interleavings
  - May-happen-in-parallel analysis
  - Rely-guarantee reasoning
- Part 3: Cost analysis of distributed systems
  - Dynamic distributed locations
  - Resource analysis with cost centers
  - New performance indicators
  - Parallel and peak cost

Sequential Programs

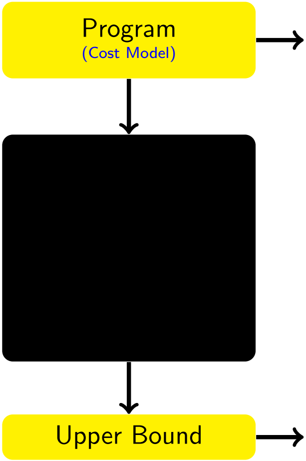

Program
(Cost Model)
Upper Bound




```
while (l != null) {
  l = l.next;
  new C();
}
```

(number of instructions)

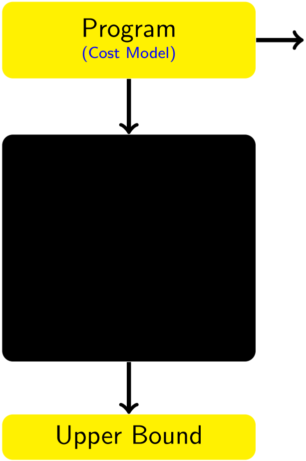
Resource Analysis of Sequential Code




```
while (l != null) {
  l = l.next;
  new C();
}
```

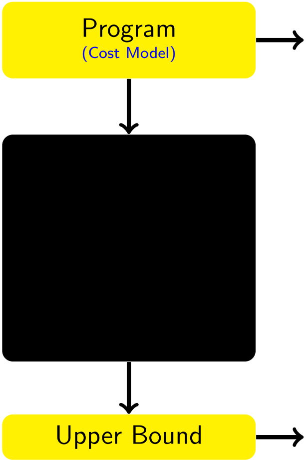
(number of instructions)

$1 + 3*size(l)$




```
while (l != null) {
   l = l.next;
   new C();
}
```

(number of visits to a program point)

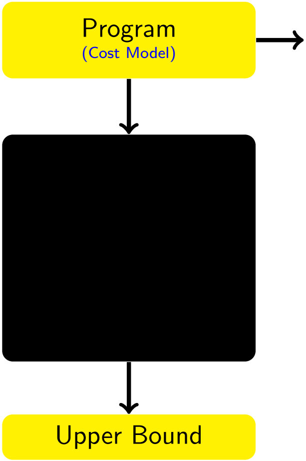



```
while (l != null) {
   l = l.next;
   new C(); ⇐ program point
}
```

(number of visits to a program point)

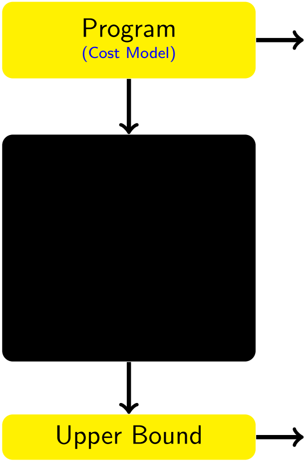
$size(l)$

Resource Analysis of Sequential Code




```
while (l != null) {
  l = l.next;
  new C();
}
```

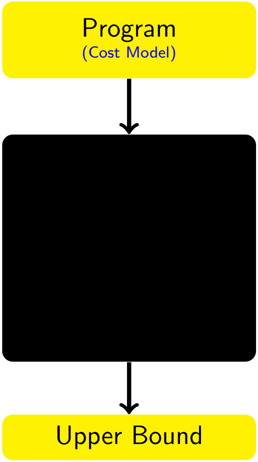
(memory)




```
while (l != null) {
  l = l.next;
  new C(); ⇐ memory
}
```

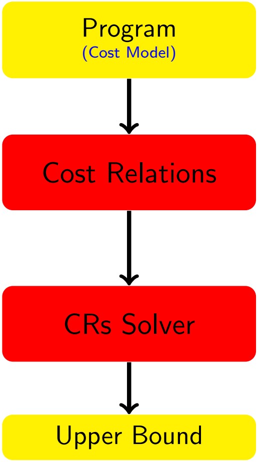
(memory)

$size(l) * size(C)$



A Classical approach [Wegbreit'75] to cost analysis consists of:

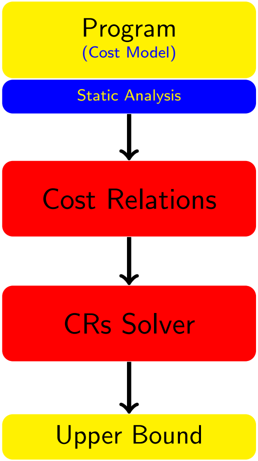
1. expressing the cost of a program by means of *recurrence relations*.
2. solving the relations by obtaining a *closed-form upper bound* (a function of the input data sizes).



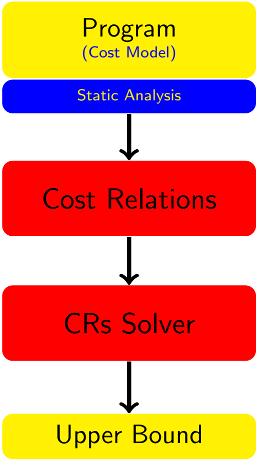
A Classical approach [Wegbreit'75] to cost analysis consists of:

1. expressing the cost of a program by means of *recurrence relations*.
2. solving the relations by obtaining a *closed-form upper bound* (a function of the input data sizes).

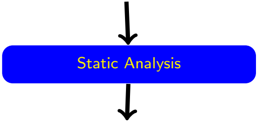
Resource Analysis of Sequential Code



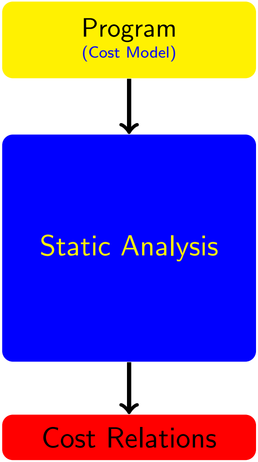
Resource Analysis of Sequential Code



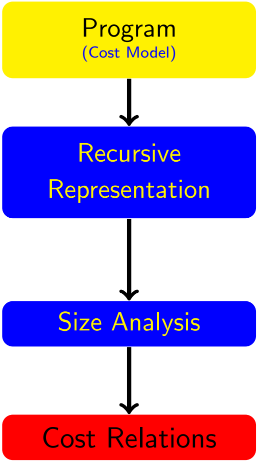
while (l != null) l = l.next;

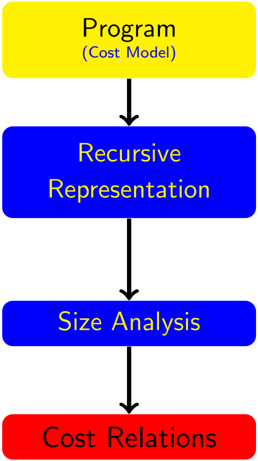


$$\begin{array}{lcll} while(l) & = & k_1 & \{l{=}0\} \\ while(l) & = & k_2{+}while(l'') & \{l{>}0, l{>}l''\} \end{array}$$


Program
(Cost Model)
Static Analysis
Cost Relations

Resource Analysis of Sequential Code



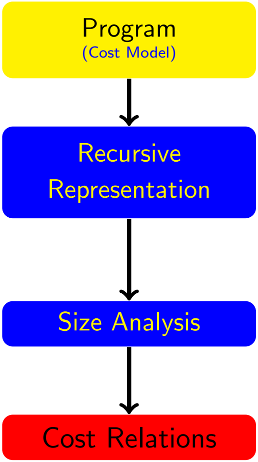


while (l != null) l = l.next;

↓

$$
\begin{array}{lll}
while(l,l) & \leftarrow & l{=}null. \\
while(l,l') & \leftarrow & l{\neq}null, \\
 & & l''{=}l.next, \\
 & & while(l'',l').
\end{array}
$$

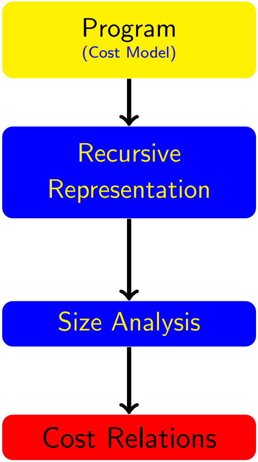
Resource Analysis of Sequential Code



while (l != null) l = l.next;

↓

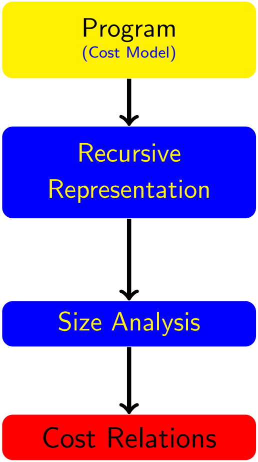
$$
\begin{array}{lll}
\mathit{while}(l, l) & \leftarrow & l{=}\mathit{null}. \\
\mathit{while}(l, l') & \leftarrow & l{\neq}\mathit{null}, \\
 & & l''{=}l.\mathit{next}, \\
 & & \mathit{while}(l'', l').
\end{array}
$$




```
while (l != null) l = l.next;
```

↓

$$
\begin{array}{lll}
\mathit{while}(l, l) & \leftarrow & l{=}\mathit{null}. \\
\mathit{while}(l, l') & \leftarrow & l{\neq}\mathit{null}, \\
 & & l''{=}l.\mathit{next}, \\
 & & \mathit{while}(l'', l').
\end{array}
$$

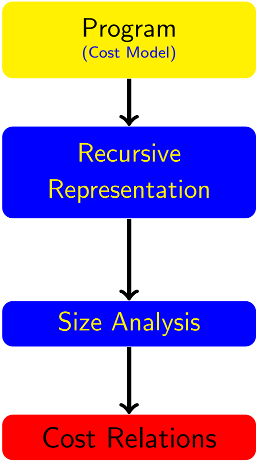


while (l != null) l = l.next;

$\downarrow$

$$
\begin{aligned}
while(l, l) &\leftarrow l{=}null.\\
while(l, l') &\leftarrow l{\neq}null,\\
&\quad\; l''{=}l.next,\\
&\quad\; while(l'', l').
\end{aligned}
$$

Resource Analysis of Sequential Code



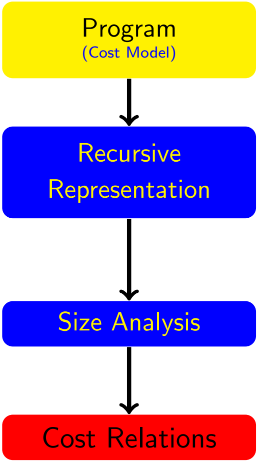

```
while (l != null) l = l.next;
```

$\Downarrow$

$$
\begin{array}{lll}
\mathit{while}(l, l) & \leftarrow & l{=}\mathit{null}. \\
\mathit{while}(l, l') & \leftarrow & l{\neq}\mathit{null}, \\
 & & l''{=}l.\mathit{next}, \\
 & & \mathit{while}(l'', l').
\end{array}
$$

$\Downarrow$

$$
\begin{array}{lll}
\mathit{size}_1 & \leftarrow & \{l{=}0\} \\
\mathit{size}_2 & \leftarrow & \{l{>}0, l{>}l''\}
\end{array}
$$



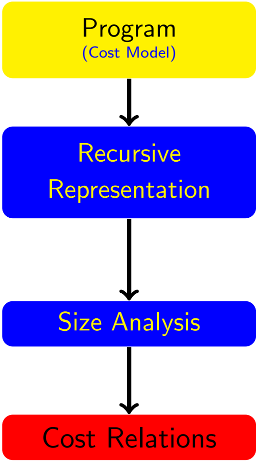
while (l != null) l = l.next;

$\downarrow$

$$
\begin{array}{lll}
while(l,l) & \leftarrow & l{=}null. \\
while(l,l') & \leftarrow & l{\neq}null, \\
 & & l''{=}l.next, \\
 & & while(l'',l').
\end{array}
$$

$\downarrow$

$$
\begin{array}{lll}
size_1 & \leftarrow & \{l{=}0\} \\
size_2 & \leftarrow & \{l{>}0, l{>}l''\}
\end{array}
$$



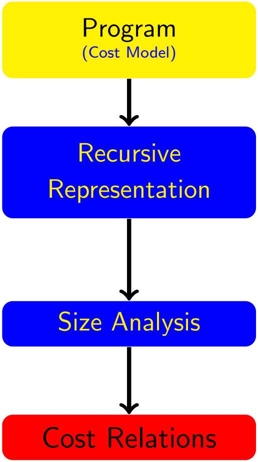
while (l != null) l = l.next;

↓

$$\begin{aligned} &while(l, l) \leftarrow && l{=}null. \\ &while(l, l') \leftarrow && l{\neq}null, \\ & && l''{=}l.next, \\ & && while(l'', l'). \end{aligned}$$

↓

$$\begin{aligned} &size_1 \leftarrow && \{l{=}0\} \\ &size_2 \leftarrow && \{l{>}0, l{>}l''\} \end{aligned}$$



while (l != null) l = l.next;

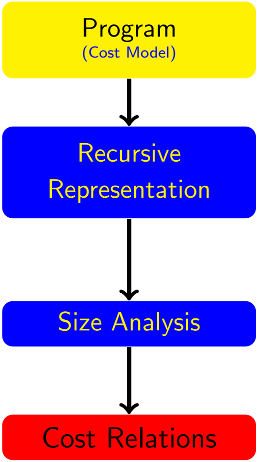
↓

$$
\begin{array}{lll}
while(l, l) & \leftarrow & l{=}null. \\
while(l, l') & \leftarrow & l{\neq}null, \\
 & & \color{red}{l''{=}l.next}, \\
 & & while(l'', l').
\end{array}
$$

↓

$$
\begin{array}{lll}
size_1 & \leftarrow & \{l{=}0\} \\
size_2 & \leftarrow & \{l{>}0, \color{red}{l{>}l''}\}
\end{array}
$$

Resource Analysis of Sequential Code




```
while (l != null) l = l.next;
```

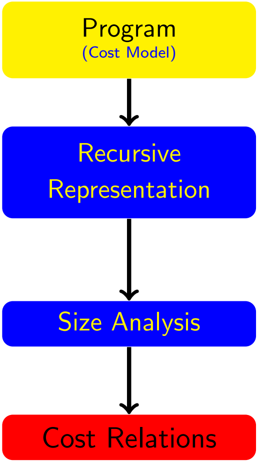
↓

$$\begin{array}{lll} while(l, l) & \leftarrow & l{=}null. \\ while(l, l') & \leftarrow & l{\neq}null, \\ & & l''{=}l.next, \\ & & while(l'', l'). \end{array}$$

↓

$$\begin{array}{lll} size_1 & \leftarrow & \{l{=}0\} \\ size_2 & \leftarrow & \{l{>}0, l{>}l''\} \end{array}$$

Resource Analysis of Sequential Code



while (l != null) l = l.next;

$\downarrow$

$$
\begin{array}{lll}
while(l,l) & \leftarrow & l{=}null. \\
while(l,l') & \leftarrow & l{\neq}null, \\
 & & l''{=}l.next, \\
 & & while(l'',l').
\end{array}
$$

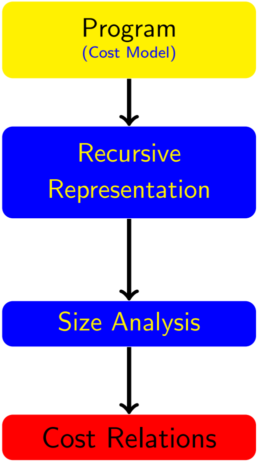
$\downarrow$

$$
\begin{array}{lll}
size_1 & \leftarrow & \{l{=}0\} \\
size_2 & \leftarrow & \{l{>}0, l{>}l''\}
\end{array}
$$

$\downarrow$

$$
\begin{array}{ll}
while(l) = k_1 & \{l{=}0\} \\
while(l) = k_2 + while(l'') & \{l{>}0, l{>}l''\}
\end{array}
$$

Resource Analysis of Sequential Code



while (l != null) l = l.next;

↓

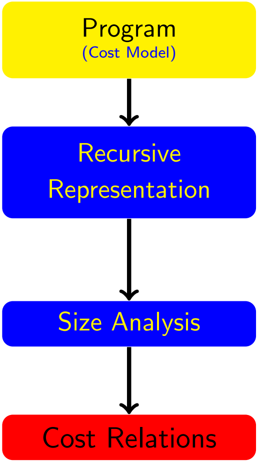
$$
\begin{array}{lll}
while(l,l) & \leftarrow & \color{red}{l{=}null}. \\
while(l,l') & \leftarrow & l{\neq}null, \\
 & & l''{=}l.next, \\
 & & while(l'',l').
\end{array}
$$

↓

$$
\begin{array}{lll}
size_1 & \leftarrow & \{l{=}0\} \\
size_2 & \leftarrow & \{l{>}0, l{>}l''\}
\end{array}
$$

↓

$$
\begin{array}{ll}
while(l) = \color{red}{k_1} & \{l{=}0\} \\
while(l) = k_2{+}while(l'') & \{l{>}0, l{>}l''\}
\end{array}
$$



while (l != null) l = l.next;

↓

$$\begin{aligned} while(l,l) &\leftarrow \; l{=}null. \\ while(l,l') &\leftarrow \; l{\neq}null, \\ &\quad\;\; l''{=}l.next, \\ &\quad\;\; while(l'',l'). \end{aligned}$$

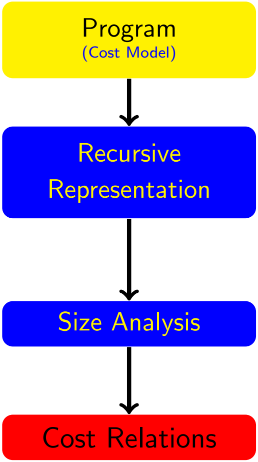
↓

$$\begin{aligned} size_1 &\leftarrow \{l{=}0\} \\ size_2 &\leftarrow \{l{>}0, l{>}l''\} \end{aligned}$$

↓

$$\begin{aligned} while(l) &= k_1 & \{l{=}0\} \\ while(l) &= k_2 + while(l'') & \{l{>}0, l{>}l''\} \end{aligned}$$

Resource Analysis of Sequential Code



while (l != null) l = l.next;

↓

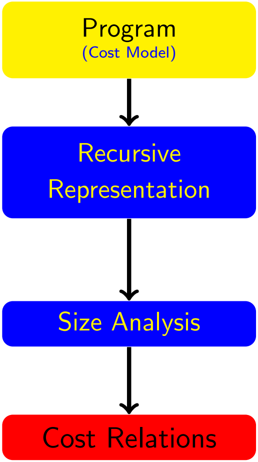
$$
\begin{array}{lll}
while(l, l) & \leftarrow & l{=}null. \\
while(l, l') & \leftarrow & l{\neq}null, \\
 & & l''{=}l.next, \\
 & & while(l'', l').
\end{array}
$$

↓

$$
\begin{array}{lll}
size_1 & \leftarrow & \{l{=}0\} \\
size_2 & \leftarrow & \{l{>}0, l{>}l''\}
\end{array}
$$

↓

$$
\begin{array}{ll}
while(l) = 1 & \{l{=}0\} \\
while(l) = k_2{+}while(l'') & \{l{>}0, l{>}l''\}
\end{array}
$$



while (l != null) l = l.next;

$\downarrow$

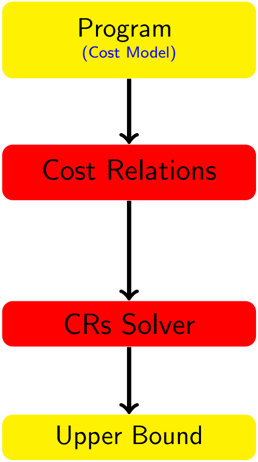
$$
\begin{array}{lll}
\mathit{while}(l,l) & \leftarrow & l{=}\mathit{null}. \\
\mathit{while}(l,l') & \leftarrow & {\color{red} l{\neq}\mathit{null}}, \\
 & & {\color{red} l''{=}l.\mathit{next}}, \\
 & & \mathit{while}(l'',l').
\end{array}
$$

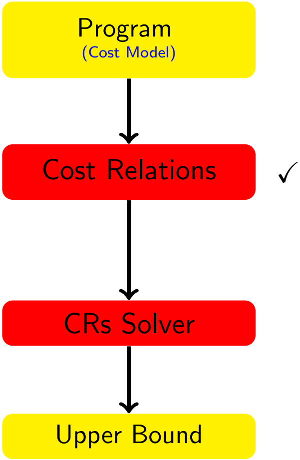
$\downarrow$

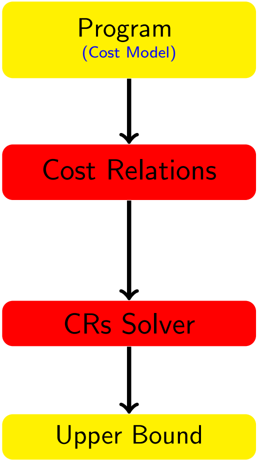
$$
\begin{array}{lll}
\mathit{size}_1 & \leftarrow & \{l{=}0\} \\
\mathit{size}_2 & \leftarrow & \{l{>}0, l{>}l''\}
\end{array}
$$

$\downarrow$

$$
\begin{array}{ll}
\mathit{while}(l) = 1 & \{l{=}0\} \\
\mathit{while}(l) = {\color{red} 2}{+}\mathit{while}(l'') & \{l{>}0, l{>}l''\}
\end{array}
$$


Program
(Cost Model)
Cost Relations
CRs Solver
Upper Bound


Program
(Cost Model)
Cost Relations
✓
CRs Solver
Upper Bound



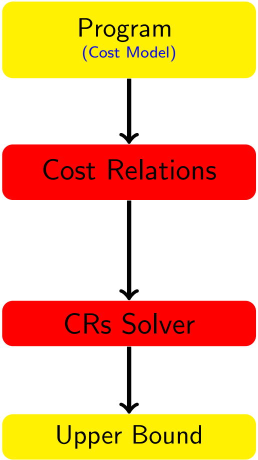
✓

while (l != null) l = l.next;

↓

$$\begin{array}{lll} while(l) = k_1 & \{l{=}0\} \\ while(l) = k_2 + while(l'') & \{l{>}0, l{>}l''\} \end{array}$$

✓



✓

while (l != null) l = l.next;

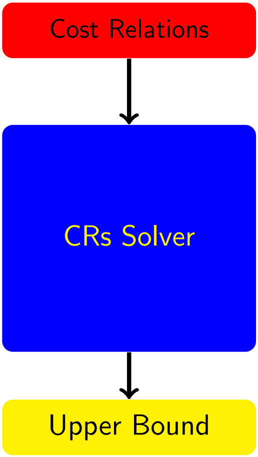
↓

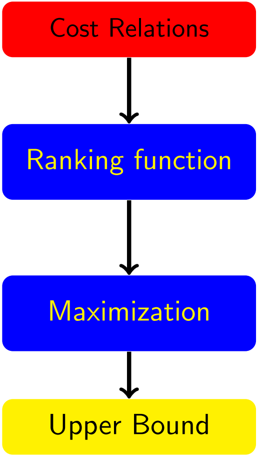
$$\begin{array}{lll} while(l) = k_1 & \{l{=}0\} \\ while(l) = k_2{+}while(l'') & \{l{>}0, l{>}l''\} \end{array}$$ ✓

↓

$$k_1 + k_2 * l$$

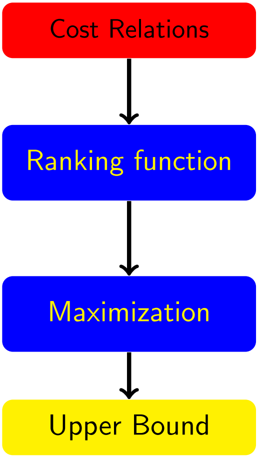
Resource Analysis of Sequential Code





$$
\begin{array}{ll}
\mathit{while}(l) = k_1 & \{l{=}0\} \\
\mathit{while}(l) = k_2 + \mathit{while}(l'') & \{l{>}0, l{>}l''\}
\end{array}
$$

Resource Analysis of Sequential Code

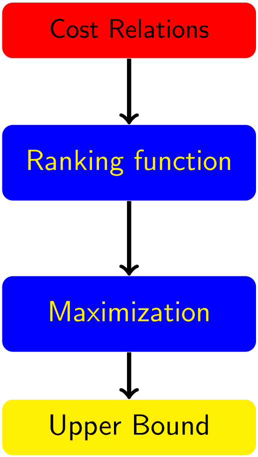


$$\begin{array}{ll} while(l) = k_1 & \{l{=}0\} \\ while(l) = k_2 + while(l'') & \{l{>}0, l{>}l''\} \end{array}$$

↓

$$RF(l) = l$$

(linear expression on l)



$$
\begin{array}{ll}
while(l) = k_1 & \{l{=}0\} \\
while(l) = k_2 + while(l'') & \{l{>}0, l{>}l''\}
\end{array}
$$

↓

$$RF(l) = l$$

(linear expression on l)

↓

Maximization remains the same

k_1 and k_2 are constants

Cost Relations

$$while(l) = k_1 \quad \{l{=}0\}$$
$$while(l) = k_2 + while(l'') \quad \{l{>}0, l{>}l''\}$$

↓

Ranking function

$$RF(l) = l$$

(linear expression on l)

↓

Maximization

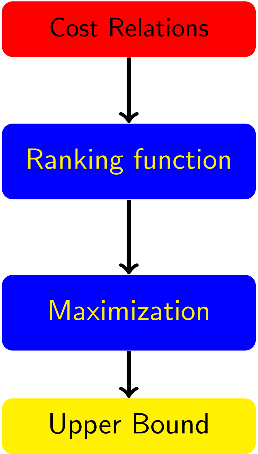
Maximization remains the same

k_1 and k_2 are constants

↓

Upper Bound

$$while^+(l) = cost^+_{bc} + RF(l) * cost^+_{loop}$$



$while(l) = k_1 \quad \{l{=}0\}$
$while(l) = k_2 + while(l'') \quad \{l{>}0, l{>}l''\}$

↓

$$RF(l) = l$$

(linear expression on l)

↓

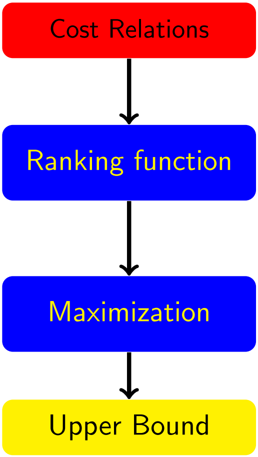
Maximization remains the same

k_1 and k_2 are constants

↓

$while^{+}(l) = cost^{+}_{bc} + RF(l) * cost^{+}_{loop}$
$while^{+}(l) = k_1 +$

Resource Analysis of Sequential Code



$$while(l) = k_1 \qquad \{l{=}0\}$$
$$while(l) = k_2 + while(l'') \; \{l{>}0, l{>}l''\}$$

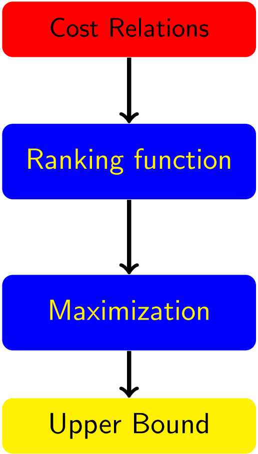
$$\color{red}{RF(l) = l}$$

(linear expression on l)

Maximization remains the same

k_1 and k_2 are constants

$$while^+(l) = cost^+_{bc} + \color{red}{RF(l)} * cost^+_{loop}$$
$$while^+(l) = k_1 + \color{red}{l} *$$



$while(l) = k_1 \quad \{l{=}0\}$
$while(l) = k_2 + while(l'') \quad \{l{>}0, l{>}l''\}$

$$RF(l) = l$$

(linear expression on l)

Maximization remains the same
k_1 and k_2 are constants

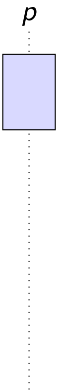
$while^+(l) = cost^+_{bc} + RF(l) * cost^+_{loop}$
$while^+(l) = k_1 + l * k_2$

Summary Sequential

- The process involves a series of transformations and analyses:
  - Transformation into recursive form
  - Size analysis
  - Generation of cost relations
  - Ranking functions and maximization
- We cover polynomial, exponential, logarithmic complexities
- From now on: given task m, we assume cost $\mathcal{U}_m$
- Main references: **ESOP'07**, **SAS'08**
- Handling fields: **SAS'10**, **FM'11**

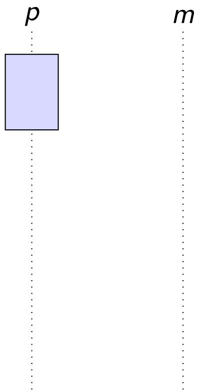
Concurrent Programs

Adding Concurrency



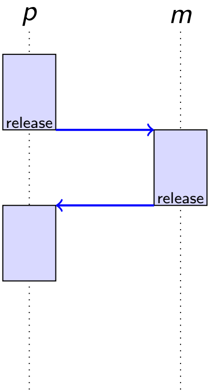
- Different tasks interleave execution in the same processor

Adding Concurrency



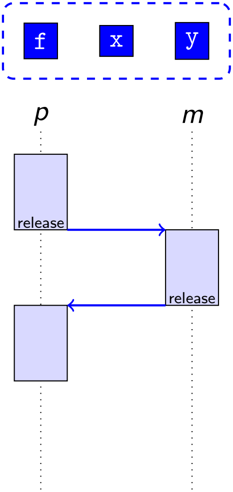
- Different tasks interleave execution in the same processor
- Asynchronous task invocations $\mathtt{m}(\bar{x})$

Adding Concurrency



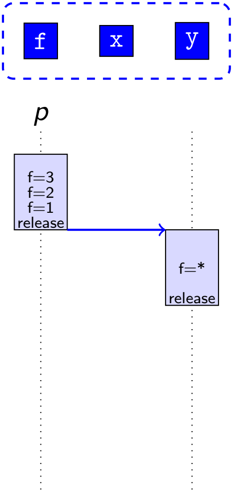
- Different tasks interleave execution in the same processor
- Asynchronous task invocations `m(x̄)`
- Non-preemptive concurrency by explicitly releasing the processor `release`

Adding Concurrency



- Different tasks interleave execution in the same processor
- Asynchronous task invocations `m`$(\bar{x})$
- Non-preemptive concurrency by explicitly releasing the processor `release`
- Shared memory among the different tasks

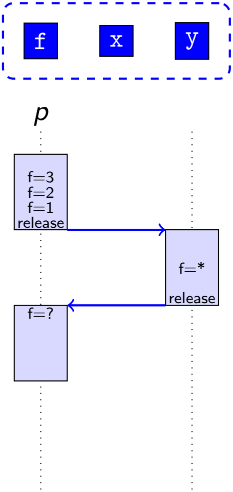
Resource Analysis with interleavings (I)




```
while (f>0){
  ...
  f = f-1;
  release;
}
```

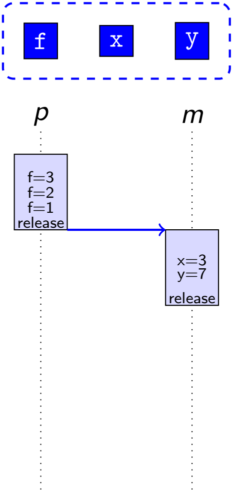
- 1^{st} **approach**: assume that shared memory changes after every `release`

Resource Analysis with interleavings (I)




```
while (f>0){
  ...
  f = f-1;
  release;
}
```

- **1st approach**: assume that shared memory changes after every `release`
- Loss of information, poor results → loops based on shared variables cannot be bound.

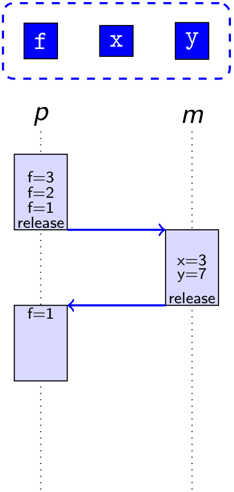


p()
```
1 while (f>0){
2    ...
3   f = f-1;
4   release;
5 }
```

m()
```
6 x = 3
7 y = 7;
```

- **2^{nd} approach**: use a *May-Happen-in-Parallel* analysis to infer instructions pairs that can interleave: $\ldots(4,6),(4,7)\ldots$

Resource Analysis with interleavings (II)

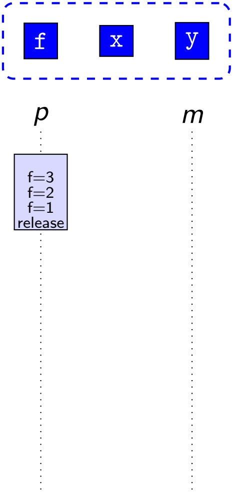


p()
```
1 while (f>0){
2   ...
3   f = f-1;
4   release;
5 }
```

m()
```
6 x = 3
7 y = 7;
```

- **2^{nd} approach**: use a *May-Happen-in-Parallel* analysis to infer instructions pairs that can interleave: $\ldots(4,6),(4,7)\ldots$
- Shared memory can only change if an update can interleave with `release` $\rightarrow$ improve results

Resource Analysis with interleavings (III)

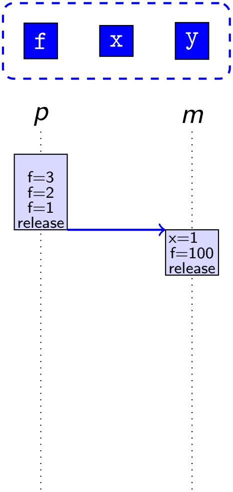



```
p()
while (f>0){
  ...
  f = f-1;
  release;
}
```

```
m()
while (x>0){
  x = x-1;
  f = 100;
  release;
}
```

- **3rd approach**: interleavings that modify shared memory are safe if they can only happen a *finite* number of times

Resource Analysis with interleavings (III)

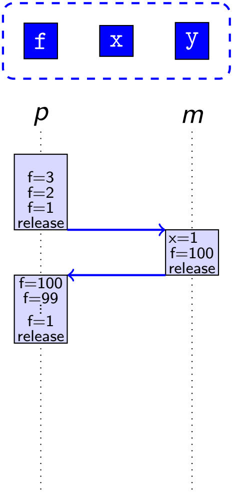


p()
```
while (f>0){
   ...
  f = f-1;
  release;
}
```

m()
```
while (x>0){
  x = x-1;
  f = 100;
  release;
}
```

- **3rd approach**: interleavings that modify shared memory are safe if they can only happen a *finite* number of times

Resource Analysis with interleavings (III)

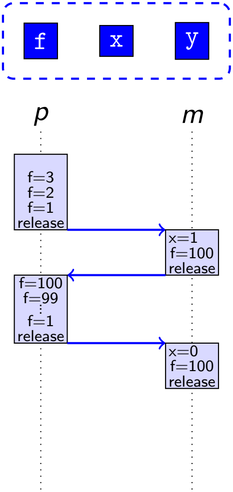


p()
```
while (f>0){
   ...
  f = f-1;
  release;
}
```

m()
```
while (x>0){
  x = x-1;
  f = 100;
  release;
}
```

- **3rd approach**: interleavings that modify shared memory are safe if they can only happen a *finite* number of times

Resource Analysis with interleavings (III)

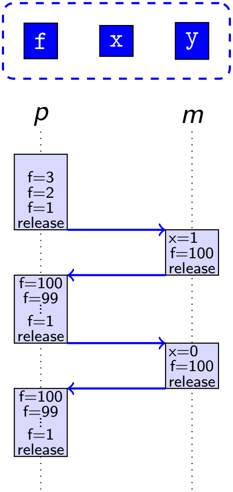


p()
```
while (f>0){
  ...
  f = f-1;
  release;
}
```

m()
```
while (x>0){
  x = x-1;
  f = 100;
  release;
}
```

- **3rd approach**: interleavings that modify shared memory are safe if they can only happen a *finite* number of times

Resource Analysis with interleavings (III)



p()

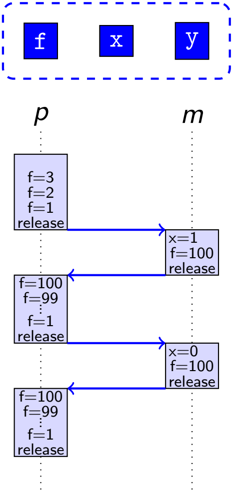
```
while (f>0){
  ...
  f = f-1;
  release;
}
```

m()

```
while (x>0){
  x = x-1;
  f = 100;
  release;
}
```

- **3^{rd} approach**: interleavings that modify shared memory are safe if they can only happen a *finite* number of times

Resource Analysis with interleavings (III)



p()
```
while (f>0){
  ...
  f = f-1;
  release;
}
```

m()
```
while (x>0){
  x = x-1;
  f = 100;
  release;
}
```

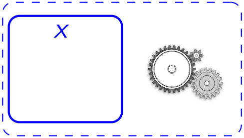
- **3rd approach**: interleavings that modify shared memory are safe if they can only happen a *finite* number of times
- Rely-guarantee reasoning: $max(f) \times (max(x) + 1)$

Summary Concurrent Programs

- Basic resource analysis for sound results **APLAS'11**
- May-happen-in-parallel analysis **FORTE'12, LPAR'13, SAS'15**
- Rely-guarantee reasoning **ATVA'13, JAR'17**
- From now on: given a concurrent task m, we assume cost $\mathcal{U}_m$

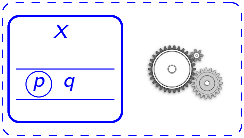
Distributed Systems

Adding distribution



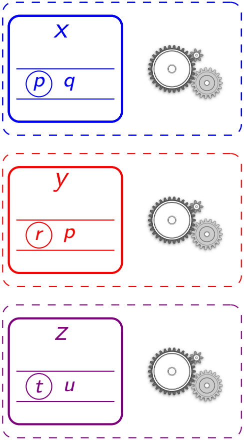
- `X = newLoc` to create a distributed location

Adding distribution



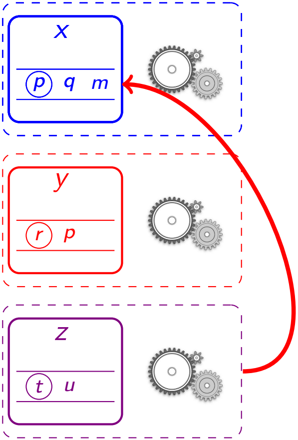
- X = newLoc to create a distributed location
- A location has a queue of pending tasks and one active task

Adding distribution

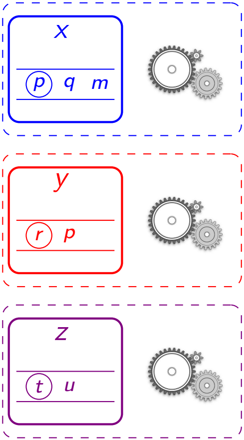


- `X = newLoc` to create a distributed location
- A location has a queue of pending tasks and one active task
- Multiple locations can be created dynamically `y=newLoc; z=newLoc`

Adding distribution

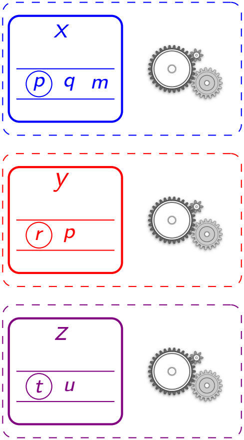


- `X = newLoc` to create a distributed location
- A location has a queue of pending tasks and one active task
- Multiple locations can be created dynamically `y=newLoc; z=newLoc`
- Asynchronous tasks can be added among locations: `x.m(w)` (in `z`)

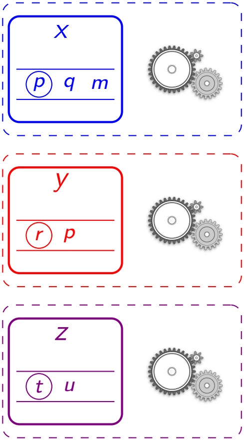


- Using cost analysis so far:
  $C = \mathcal{U}_p + \mathcal{U}_q + \mathcal{U}_m + \mathcal{U}_r + \mathcal{U}_p + \cdots + \mathcal{U}_t + \mathcal{U}_u$

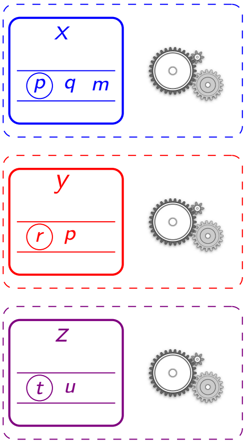
Resource analysis with cost centers



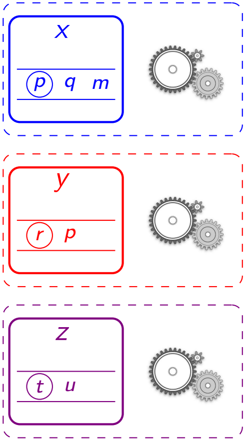
- Using cost analysis so far:
  $C = \mathcal{U}_p + \mathcal{U}_q + \mathcal{U}_m + \mathcal{U}_r + \mathcal{U}_p + \cdots + \mathcal{U}_t + \mathcal{U}_u$
- We aim at having the cost at the level of distributed components
  $C_x = \mathcal{U}_p + \mathcal{U}_q + \mathcal{U}_m \quad C_y = \mathcal{U}_r + \mathcal{U}_p \quad \ldots$



- Using cost analysis so far:
  $C = \mathcal{U}_p + \mathcal{U}_q + \mathcal{U}_m + \mathcal{U}_r + \mathcal{U}_p + \cdots + \mathcal{U}_t + \mathcal{U}_u$
- We aim at having the cost at the level of distributed components
  $C_x = \mathcal{U}_p + \mathcal{U}_q + \mathcal{U}_m \quad C_y = \mathcal{U}_r + \mathcal{U}_p \quad \ldots$
- Idea: use cost centers to separate the cost
  $c(x), c(y), c(z)$

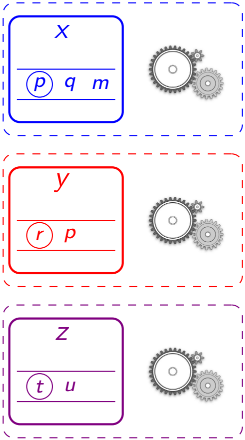


- Using cost analysis so far:
  $C = \mathcal{U}_p + \mathcal{U}_q + \mathcal{U}_m + \mathcal{U}_r + \mathcal{U}_p + \cdots + \mathcal{U}_t + \mathcal{U}_u$
- We aim at having the cost at the level of distributed components
  $C_x = \mathcal{U}_p + \mathcal{U}_q + \mathcal{U}_m \quad C_y = \mathcal{U}_r + \mathcal{U}_p \quad \ldots$
- Idea: use cost centers to separate the cost
  $c(x), c(y), c(z)$
- When we analyze an instruction i, its cost C_i is added to the cost center of the x component: $c(x) \cdot C_i$



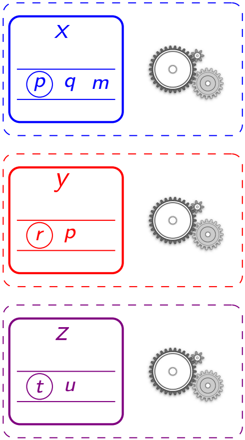
- Using cost analysis so far:
  $C = \mathcal{U}_p + \mathcal{U}_q + \mathcal{U}_m + \mathcal{U}_r + \mathcal{U}_p + \cdots + \mathcal{U}_t + \mathcal{U}_u$
- We aim at having the cost at the level of distributed components
  $C_x = \mathcal{U}_p + \mathcal{U}_q + \mathcal{U}_m \quad C_y = \mathcal{U}_r + \mathcal{U}_p \quad \ldots$
- Idea: use cost centers to separate the cost
  $c(x), c(y), c(z)$
- When we analyze an instruction i, its cost C_i is added to the cost center of the x component: $c(x) \cdot C_i$
- Global cost expression:

$$c(x)\cdot(\mathcal{U}_p+\mathcal{U}_q+\mathcal{U}_m)+c(y)\cdot(\mathcal{U}_r+\mathcal{U}_p)+c(z)\cdot(\mathcal{U}_t+\mathcal{U}_u)$$



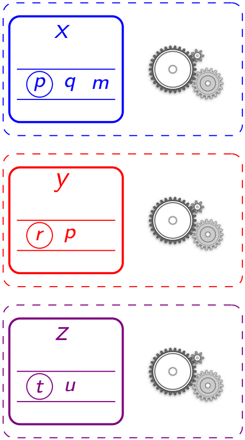
- Using cost analysis so far:
  $C = \mathcal{U}_p + \mathcal{U}_q + \mathcal{U}_m + \mathcal{U}_r + \mathcal{U}_p + \cdots + \mathcal{U}_t + \mathcal{U}_u$
- We aim at having the cost at the level of distributed components
  $C_x = \mathcal{U}_p + \mathcal{U}_q + \mathcal{U}_m \quad C_y = \mathcal{U}_r + \mathcal{U}_p \quad \ldots$
- Idea: use cost centers to separate the cost
  $c(x), c(y), c(z)$
- When we analyze an instruction i, its cost C_i is added to the cost center of the x component: $c(x) \cdot C_i$
- Global cost expression:

$$c(x)\cdot(\mathcal{U}_p+\mathcal{U}_q+\mathcal{U}_m)+\cancel{c(y)\cdot(\mathcal{U}_r+\mathcal{U}_p)}+\cancel{c(z)\cdot(\mathcal{U}_t+\mathcal{U}_u)}$$



- Using cost analysis so far:
  $C = \mathcal{U}_p + \mathcal{U}_q + \mathcal{U}_m + \mathcal{U}_r + \mathcal{U}_p + \cdots + \mathcal{U}_t + \mathcal{U}_u$
- We aim at having the cost at the level of distributed components
  $C_x = \mathcal{U}_p + \mathcal{U}_q + \mathcal{U}_m \quad C_y = \mathcal{U}_r + \mathcal{U}_p \quad \ldots$
- Idea: use cost centers to separate the cost
  $c(x), c(y), c(z)$
- When we analyze an instruction i, its cost C_i is added to the cost center of the x component: $c(x) \cdot C_i$
- Global cost expression:

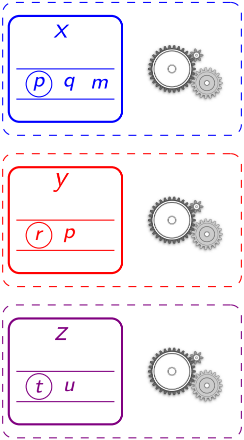
~~$c(x)\cdot(\mathcal{U}_p+\mathcal{U}_q+\mathcal{U}_m)$~~ $+\ c(y)\cdot(\mathcal{U}_r+\mathcal{U}_p) +$ ~~$c(z)\cdot(\mathcal{U}_t+\mathcal{U}_u)$~~



- Using cost analysis so far:
  $C = \mathcal{U}_p + \mathcal{U}_q + \mathcal{U}_m + \mathcal{U}_r + \mathcal{U}_p + \cdots + \mathcal{U}_t + \mathcal{U}_u$
- We aim at having the cost at the level of distributed components
  $C_x = \mathcal{U}_p + \mathcal{U}_q + \mathcal{U}_m \quad C_y = \mathcal{U}_r + \mathcal{U}_p \quad \ldots$
- Idea: use cost centers to separate the cost
  $c(x), c(y), c(z)$
- When we analyze an instruction i, its cost C_i is added to the cost center of the x component: $c(x) \cdot C_i$
- Global cost expression:

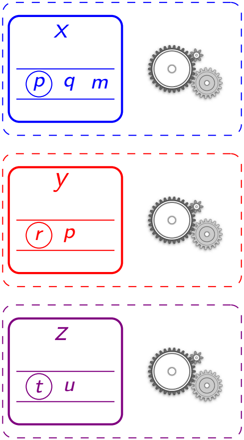
~~$c(x) \cdot (\mathcal{U}_p + \mathcal{U}_q + \mathcal{U}_m)$~~ $+$ ~~$c(y) \cdot (\mathcal{U}_r + \mathcal{U}_p)$~~ $+ c(z) \cdot (\mathcal{U}_t + \mathcal{U}_u)$

Cost centers of different types



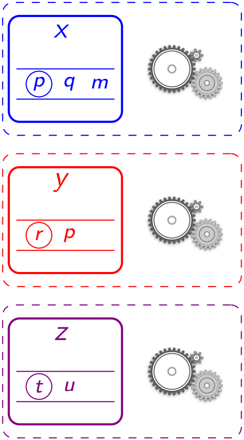
- Cost centers are a general concept that allows us to distinguish within the UB different aspects:

Cost centers of different types



- Cost centers are a general concept that allows us to distinguish within the UB different aspects:
- **Component cost centers:** $c(x)$, $c(y)$..

Cost centers of different types

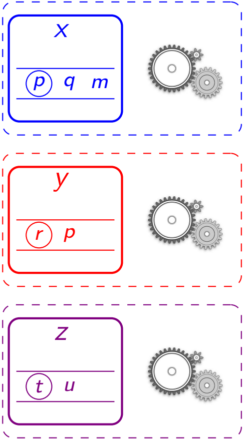


- Cost centers are a general concept that allows us to distinguish within the UB different aspects:
- **Component cost centers:** $c(x)$, $c(y)$..
- **Program point cost centers:** cost center `c(pp)` per `pp:acquire(e)`

```
for (x=0;x<n;x++) pp:acquire(e)
```

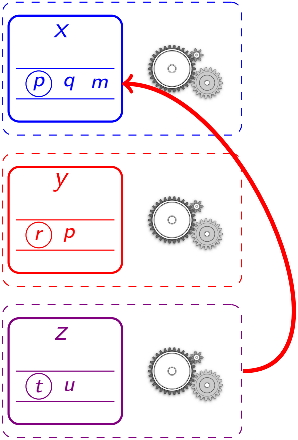
$c(pp) * n * max(e) + c(pp2) * \ldots$

Cost centers of different types



- Cost centers are a general concept that allows us to distinguish within the UB different aspects:
- **Component cost centers:** $c(x)$, $c(y)$..
- **Program point cost centers:** cost center `c(pp)` per `pp:acquire(e)`
  `for (x=0;x<n;x++) pp:acquire(e)`
  $c(pp) * n * max(e) + c(pp2) * \ldots$
- **Task level centers:** cost center `c(m)` per method
  $c(m) * C_m + c(p) * \ldots$

Cost centers of different types



- Cost centers are a general concept that allows us to distinguish within the UB different aspects:
- **Component cost centers:** $c(x)$, $c(y)$..
- **Program point cost centers:** cost center `c(pp)` per `pp:acquire(e)`
  `for (x=0;x<n;x++) pp:acquire(e)`
  $c(pp) * n * max(e) + c(pp2) * \ldots$
- **Task level centers:** cost center `c(m)` per method
  $c(m) * C_m + c(p) * \ldots$
- **Multi-component cost centers:** cost centers of the form $c(z, x)$, i.e., when we find an instruction `x.m(w)` in z we do $c(z, x) * size(w)$

Parallel Cost

Parallel Cost

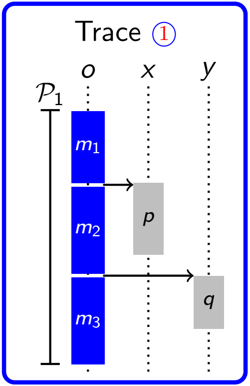
- Serial cost: accumulate costs from different locations
- Limitation: ignore the parallelism of the distributed execution model.
- New analysis: infer the parallel cost of distributed systems (maximum cost between parallel tasks)
- Use: know if an application succeeds in exploiting the parallelism of the distributed locations, overall resource consumption

Parallel Cost

```
void m (int n){
  ... // m1
  x.p(n);
  ... // m2
  y.q(n);
  ... // m3
}
```

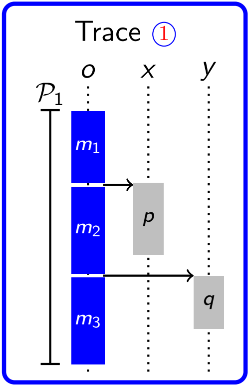
Parallel Cost

```
void m (int n){
   ... // m1
   x.p(n);
   ... // m2
   y.q(n);
   ... // m3
}
```



Parallel Cost

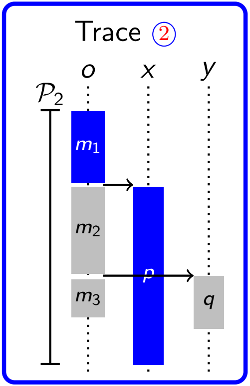
```
void m (int n){
  ... // m1
  x.p(n);
  ... // m2
  y.q(n);
  ... // m3
}
```



$$\mathcal{P}_1 = \mathcal{U}_{m_1} + \mathcal{U}_{m_2} + \mathcal{U}_{m_3}$$

Parallel Cost

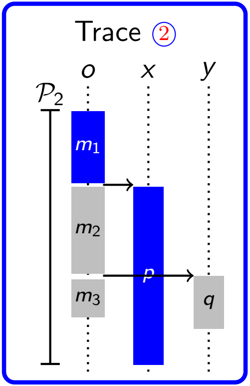
```
void m (int n){
  ... // m1
  x.p(n);
  ... // m2
  y.q(n);
  ... // m3
}
```



$\mathcal{P}_1 = \mathcal{U}_{m_1} + \mathcal{U}_{m_2} + \mathcal{U}_{m_3}$

Parallel Cost

```
void m (int n){
  ... // m1
  x.p(n);
  ... // m2
  y.q(n);
  ... // m3
}
```

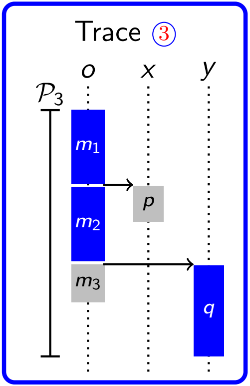


$\mathcal{P}_1 = \mathcal{U}_{m_1} + \mathcal{U}_{m_2} + \mathcal{U}_{m_3}$

$\mathcal{P}_2 = \mathcal{U}_{m_1} + \mathcal{U}_p$

Parallel Cost

```
void m (int n){
  ... // m1
  x.p(n);
  ... // m2
  y.q(n);
  ... // m3
}
```

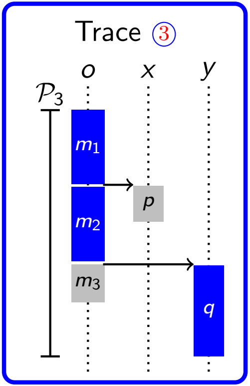


$\mathcal{P}_1 = \mathcal{U}_{m_1} + \mathcal{U}_{m_2} + \mathcal{U}_{m_3}$

$\mathcal{P}_2 = \mathcal{U}_{m_1} + \mathcal{U}_p$

Parallel Cost

```
void m (int n){
   ...  // m1
   x.p(n);
   ...  // m2
   y.q(n);
   ...  // m3
}
```



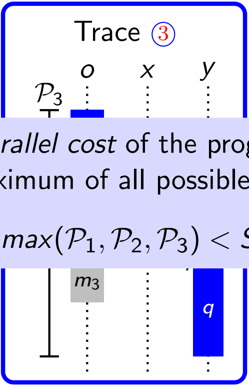
$\mathcal{P}_1 = \mathcal{U}_{m_1} + \mathcal{U}_{m_2} + \mathcal{U}_{m_3}$

$\mathcal{P}_2 = \mathcal{U}_{m_1} + \mathcal{U}_{p}$

$\mathcal{P}_3 = \mathcal{U}_{m_1} + \mathcal{U}_{m_2} + \mathcal{U}_{q}$

Parallel Cost

```
void m (int n){
  ...  // m1
  x.p(n);
  ...  // m2
  y.q(n);
  ...  // m3
}
```



The *parallel cost* of the program is the maximum of all possible traces:

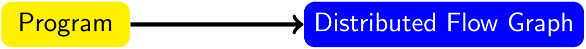
$$\mathcal{P} = max(\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3) < Serial$$

$\mathcal{P}_1 = \mathcal{U}_{m_1} + \mathcal{U}_{m_2} + \mathcal{U}_{m_3}$

$\mathcal{P}_2 = \mathcal{U}_{m_1} + \mathcal{U}_p$

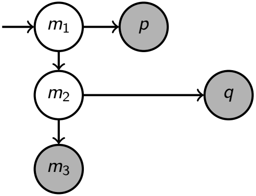
$\mathcal{U}_{m_1} + \mathcal{U}_{m_2} + \mathcal{U}_q$

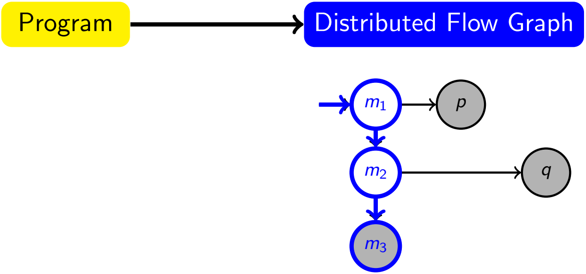
Parallel Cost Analysis



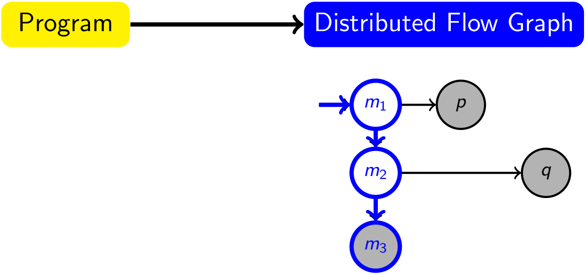
Parallel Cost Analysis

Program → Distributed Flow Graph



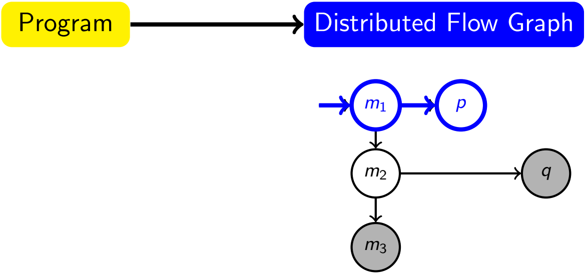

Program
Distributed Flow Graph
m_1
p
m_2
q
m_3

Parallel Cost Analysis



$N_1 = \{m_1, m_2, m_3\}$

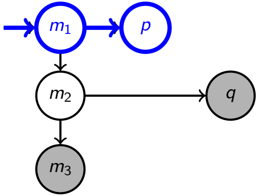
Parallel Cost Analysis



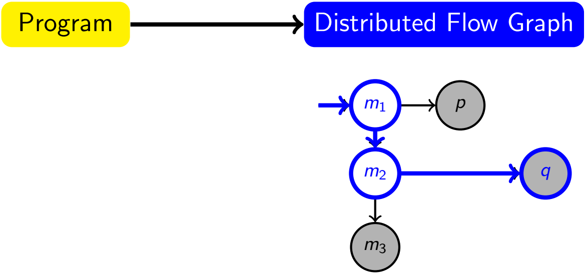
$N_1 = \{m_1, m_2, m_3\}$

Parallel Cost Analysis

Program → Distributed Flow Graph



$N_1 = \{m_1, m_2, m_3\}$ $N_2 = \{m_1, p\}$

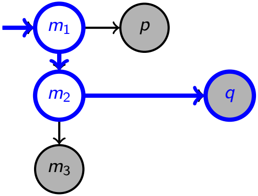


$N_1 = \{m_1, m_2, m_3\}$

$N_2 = \{m_1, p\}$

Parallel Cost Analysis

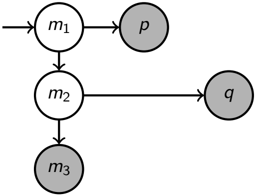
Program → Distributed Flow Graph



$N_1 = \{m_1, m_2, m_3\}$ $N_2 = \{m_1, p\}$ $N_3 = \{m_1, m_2, q\}$

Parallel Cost Analysis

Program → Distributed Flow Graph

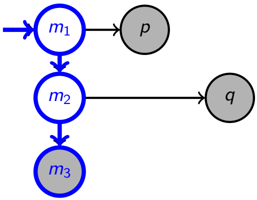


$N_1 = \{m_1, m_2, m_3\}$ $N_2 = \{m_1, p\}$ $N_3 = \{m_1, m_2, q\}$

$$Serial = c(m_1) \cdot \mathcal{U}_{m_1} + c(m_2) \cdot \mathcal{U}_{m_2} + c(m_3) \cdot \mathcal{U}_{m_3} + c(p) \cdot \mathcal{U}_p + c(q) \cdot \mathcal{U}_q$$

Parallel Cost Analysis

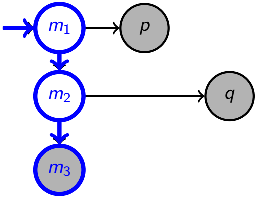
Program → Distributed Flow Graph



$N_1 = \{m_1, m_2, m_3\}$ $N_2 = \{m_1, p\}$ $N_3 = \{m_1, m_2, q\}$

$$Serial = c(m_1)\cdot\mathcal{U}_{m_1} + c(m_2)\cdot\mathcal{U}_{m_2} + c(m_3)\cdot\mathcal{U}_{m_3} + c(p)\cdot\mathcal{U}_p + c(q)\cdot\mathcal{U}_q$$

Program → Distributed Flow Graph



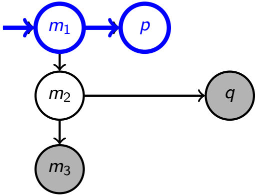
$N_1 = \{m_1, m_2, m_3\}$ $N_2 = \{m_1, p\}$ $N_3 = \{m_1, m_2, q\}$

$$Serial = c(m_1)\cdot\mathcal{U}_{m_1} + c(m_2)\cdot\mathcal{U}_{m_2} + c(m_3)\cdot\mathcal{U}_{m_3} + \cancel{c(p)\cdot\mathcal{U}_p} + \cancel{c(q)\cdot\mathcal{U}_q}$$

$$UB|_{N_1} = \mathcal{U}_{m_1} + \mathcal{U}_{m_2} + \mathcal{U}_{m_3}$$

Parallel Cost Analysis

Program → Distributed Flow Graph



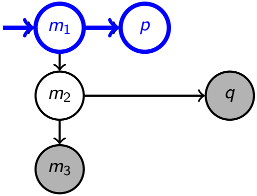
$N_1 = \{m_1, m_2, m_3\}$ $N_2 = \{m_1, p\}$ $N_3 = \{m_1, m_2, q\}$

$Serial = c(m_1) \cdot \mathcal{U}_{m_1} + c(m_2) \cdot \mathcal{U}_{m_2} + c(m_3) \cdot \mathcal{U}_{m_3} + c(p) \cdot \mathcal{U}_p + c(q) \cdot \mathcal{U}_q$

$UB|_{N_1} = \mathcal{U}_{m_1} + \mathcal{U}_{m_2} + \mathcal{U}_{m_3}$

Parallel Cost Analysis

Program → Distributed Flow Graph



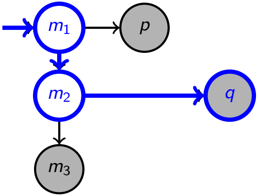
$N_1 = \{m_1, m_2, m_3\}$ $N_2 = \{m_1, p\}$ $N_3 = \{m_1, m_2, q\}$

$Serial = c(m_1) \cdot \mathcal{U}_{m_1} + c(m_2) \cdot \mathcal{U}_{m_2} + c(m_3) \cdot \mathcal{U}_{m_3} + c(p) \cdot \mathcal{U}_p + c(q) \cdot \mathcal{U}_q$

$UB|_{N_1} = \mathcal{U}_{m_1} + \mathcal{U}_{m_2} + \mathcal{U}_{m_3}$ $UB_{N_2} = \mathcal{U}_{m_1} + \mathcal{U}_p$

Parallel Cost Analysis

Program → Distributed Flow Graph



$N_1 = \{m_1, m_2, m_3\}$

$N_2 = \{m_1, p\}$

$N_3 = \{m_1, m_2, q\}$

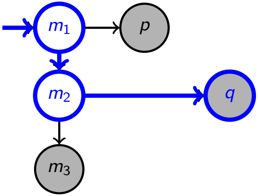
$$Serial = c(m_1) \cdot \mathcal{U}_{m_1} + c(m_2) \cdot \mathcal{U}_{m_2} + c(m_3) \cdot \mathcal{U}_{m_3} + c(p) \cdot \mathcal{U}_p + c(q) \cdot \mathcal{U}_q$$

$UB|_{N_1} = \mathcal{U}_{m_1} + \mathcal{U}_{m_2} + \mathcal{U}_{m_3}$

$UB_{N_2} = \mathcal{U}_{m_1} + \mathcal{U}_p$

Parallel Cost Analysis

Program → Distributed Flow Graph

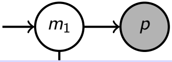


$N_1 = \{m_1, m_2, m_3\}$ $N_2 = \{m_1, p\}$ $N_3 = \{m_1, m_2, q\}$

$$Serial = c(m_1) \cdot \mathcal{U}_{m_1} + c(m_2) \cdot \mathcal{U}_{m_2} + \cancel{c(m_3) \cdot \mathcal{U}_{m_3}} + \cancel{c(p) \cdot \mathcal{U}_p} + c(q) \cdot \mathcal{U}_q$$

$UB|_{N_1} = \mathcal{U}_{m_1} + \mathcal{U}_{m_2} + \mathcal{U}_{m_3}$ $UB_{N_2} = \mathcal{U}_{m_1} + \mathcal{U}_p$ $UB_{N_3} = \mathcal{U}_{m_1} + \mathcal{U}_{m_2} + \mathcal{U}_q$

Program → Distributed Flow Graph



The *parallel cost* of the program is the maximum of all possible UB's:

$$UB^{\mathcal{P}} = max(UB_{N_1}, UB_{N_2}, UB_{N_3}) < Serial$$

$N_1=\{m_1, m_2, m_3\}$ $N_2=\{m_1, p\}$ $N_3=\{m_1, m_2, q\}$

$$UB=c(m_1)\cdot\mathcal{U}_{m_1} + c(m_2)\cdot\mathcal{U}_{m_2} + c(m_3)\cdot\mathcal{U}_{m_3} + c(p)\cdot\mathcal{U}_p + c(q)\cdot\mathcal{U}_q$$

$UB|_{N_1}=\mathcal{U}_{m_1}+\mathcal{U}_{m_2}+\mathcal{U}_{m_3}$ $UB_{N_2}=\mathcal{U}_{m_1}+\mathcal{U}_p$ $UB_{N_3}=\mathcal{U}_{m_1}+\mathcal{U}_{m_2}+\mathcal{U}_q$

Demo SACO

Peak Cost

Motivation

- **Non-cumulative resources**: are acquired and then released
- **New notion of cost**: infer the **peak** cost vs. the **total** cost
- **Technical difficulty**: not enough to reason on the final state of the execution, the upper bound on the cost can happen at any intermediate step
- **Key feature**: framework can be instantiated to measure any type of non-cumulative resource that is acquired and (optionally) freed.

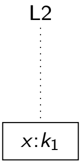
Handling resources

- Two instructions for handling resources:
  - `y = acquire(e)` allocates the amount of resources stated by expression `e`.
  - `release y` releases resources referenced by `y`.
- **resource leaks** when
  - Reusing a resource variable without releasing previous resources.
  - Reaching the end of a method without releasing a resource variable.

Peak Cost: Motivating Example

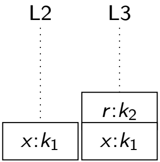
```
main (int s, int n){
    x = acquire(k1);
    r = acquire(k2);
    r = acquire(s);
    release r;
    y = acquire(n);
    release x;
}
```

```
main (int s, int n){
  x = acquire(k1);
  r = acquire(k2);
  r = acquire(s);
  release r;
  y = acquire(n);
  release x;
}
```



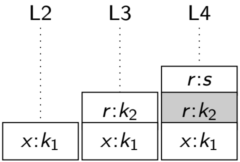
Peak Cost: Motivating Example

```
main (int s, int n){
    x = acquire(k1);
    r = acquire(k2);
    r = acquire(s);
    release r;
    y = acquire(n);
    release x;
}
```



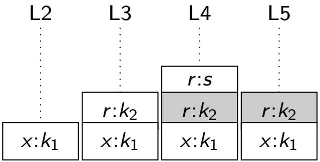
Peak Cost: Motivating Example

```
1 main (int s, int n){
2     x = acquire(k1);
3     r = acquire(k2);
4 ➡  r = acquire(s);
5     release r;
6     y = acquire(n);
7     release x;
8 }
```



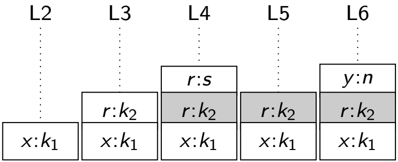
Peak Cost: Motivating Example

```
main (int s, int n){
    x = acquire(k1);
    r = acquire(k2);
    r = acquire(s);
    release r;
    y = acquire(n);
    release x;
}
```



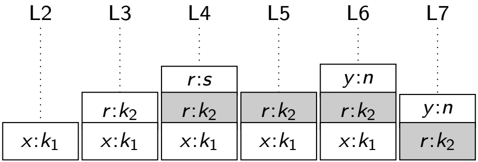
Peak Cost: Motivating Example

```
1 main (int s, int n){
2    x = acquire(k1);
3    r = acquire(k2);
4    r = acquire(s);
5    release r;
6 ➡ y = acquire(n);
7    release x;
8 }
```

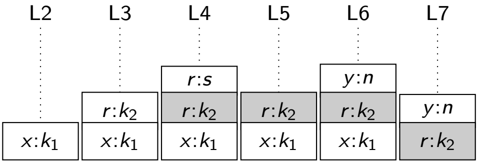


Peak Cost: Motivating Example

```
main (int s, int n){
    x = acquire(k1);
    r = acquire(k2);
    r = acquire(s);
    release r;
    y = acquire(n);
    release x;
}
```



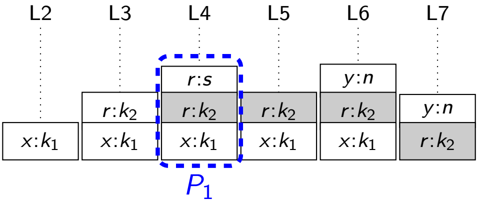
```
1 main (int s, int n){
2     x = acquire(k1);
3     r = acquire(k2);
4     r = acquire(s);
5     release r;
6     y = acquire(n);
7     release x;
8 }
```



$$Total = k_1 + k_2 + s + n$$

Peak Cost: Motivating Example

```
1 main (int s, int n){
2     x = acquire(k1);
3     r = acquire(k2);
4     r = acquire(s);
5     release r;
6     y = acquire(n);
7     release x;
8 }
```

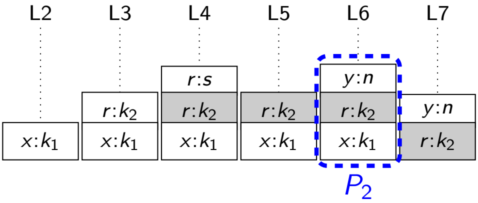


$$Total = k_1 + k_2 + s + n$$

$$P_1 = k_1 + k_2 + s$$

Peak Cost: Motivating Example

```
1 main (int s, int n){
2     x = acquire(k1);
3     r = acquire(k2);
4     r = acquire(s);
5     release r;
6     y = acquire(n);
7     release x;
8 }
```

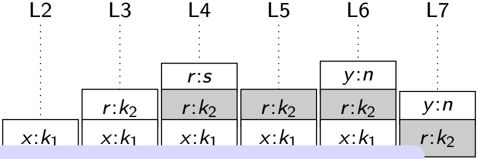


$Total = k_1 + k_2 + s + n$

$P_1 = k_1 + k_2 + s$

$P_2 = k_1 + k_2 + n$

```
1 main (int s, int n){
2    x = acquire(k1);
3    r = acquire(k2);
4    r = acquire(
5    release r;
6    y = acquire(
7    release x;
8 }
```



The *peak cost* is the maximum between them:

$$Peak = max(P_1, P_2) < Total$$

$Total = k_1 + k_2 + s + n$

$P_1 = k_1 + k_2 + s$

$P_2 = k_1 + k_2 + n$

Simultaneous resource analysis

```
main (int s, int n) {
    x = acquire(k1);
    r = acquire(k2);
    r = acquire(s);
    release r;
    y = acquire(n);
    release x;
}
```

```
main (int s, int n) {
    x = acquire(k1);
    r = acquire(k2);
    r = acquire(s);
    release r;
    y = acquire(n);
    release x;
}
```

$A_1 = \{a_2, a_3, a_4\}$

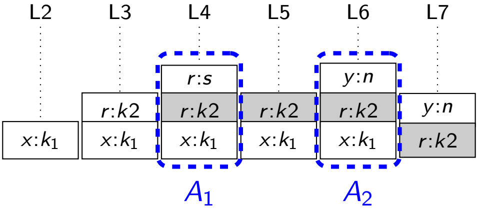
Simultaneous resource analysis

```
main (int s, int n) {
  x = acquire(k1);
  r = acquire(k2);
  r = acquire(s);
  release r;
  y = acquire(n);
  release x;
}
```

$A_1 = \{a_2, a_3, a_4\}$

$A_2 = \{a_2, a_3, a_6\}$

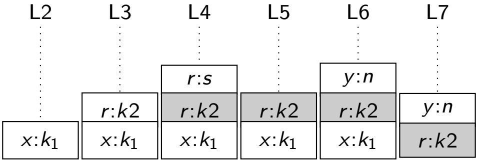
Simultaneous resource analysis



$A_1 = \{a_2, a_3, a_4\}$

$A_2 = \{a_2, a_3, a_6\}$

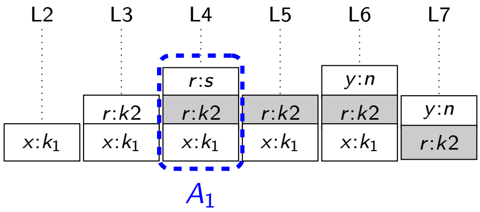
Simultaneous resource analysis



$A_1 = \{a_2, a_3, a_4\}$ $A_2 = \{a_2, a_3, a_6\}$

$$Total = c(a_2) \cdot k_1 + c(a_3) \cdot k_2 + c(a_4) \cdot s + c(a_6) \cdot n$$

Simultaneous resource analysis

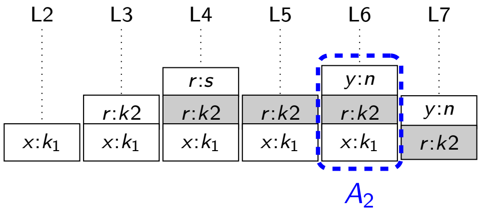


$A_1 = \{a_2, a_3, a_4\}$ $A_2 = \{a_2, a_3, a_6\}$

$Total = c(a_2) \cdot k_1 + c(a_3) \cdot k_2 + c(a_4) \cdot s + \cancel{c(a_6) \cdot n}$

$UB|_{A_1} = k_1 + k_2 + s$

Simultaneous resource analysis

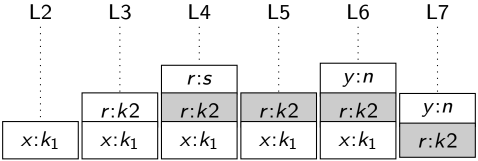


$A_1 = \{a_2, a_3, a_4\}$ $\quad$ $A_2 = \{a_2, a_3, a_6\}$

$$Total = c(a_2) \cdot k_1 + c(a_3) \cdot k_2 + \cancel{c(a_4) \cdot s} + c(a_6) \cdot n$$

$UB|_{A_1} = k_1 + k_2 + s$ $\quad$ $UB|_{A_2} = k_1 + n + k_2$

Simultaneous resource analysis



The UB on the *peak cost* of the program is the maximum of all UB's:

$$UB^{\mathcal{N}} = max(UB_{A_1}, UB_{A_2}) < Total$$

$$Total = c(a_2) \cdot k_1 + c(a_3) \cdot k_2 + c(a_4) \cdot n + c(a_6) \cdot s$$

$$UB|_{A_1} = k_1 + k_2 + s$$

$$UB|_{A_2} = k_1 + n + k_2$$

Demo SACO

Summary Distributed Systems

- Cost centers based resource analysis **APLAS'11**
- New performance indicators **iFM'13**
- Parallel cost analysis **SAS'15**
- Peak cost analysis **TACAS'15**

Conclusions

- Cost Analysis
  - research on cost analysis dates back to 1975
  - generating and solving different forms of recurrence relations

Conclusions

- Cost Analysis
  - research on cost analysis dates back to 1975
  - generating and solving different forms of recurrence relations
- From sequential to concurrent systems
  - Concurrent interleavings
  - May-happen-in-parallel based analysis
  - Rely-guarantee

Conclusions

- Cost Analysis
  - research on cost analysis dates back to 1975
  - generating and solving different forms of recurrence relations
- From sequential to concurrent systems
  - Concurrent interleavings
  - May-happen-in-parallel based analysis
  - Rely-guarantee
- From concurrent to distributed systems
  - New performance indicators
  - New notions of cost
    - Parallel cost
    - Peak cost

Conclusions

- Cost Analysis
  - research on cost analysis dates back to 1975
  - generating and solving different forms of recurrence relations
- From sequential to concurrent systems
  - Concurrent interleavings
  - May-happen-in-parallel based analysis
  - Rely-guarantee
- From concurrent to distributed systems
  - New performance indicators
  - New notions of cost
    - Parallel cost
    - Peak cost
- Integrated in the SACO system, Static Analyzer for Concurrent Objects

Credits

http://costa.ls.fi.upm.es

Elvira Albert
Puri Arenas
Einar Broch Johnsen
Jesús Correas
Jesús Domenech
Antonio Flores
Samir Genaim
Miguel Gómez-Zamalloa
Pablo Gordillo
Miguel Isabel
Enrique Martín-Martín
Germán Puebla
Guillermo Román
Damiano Zanardini