

Fine-grained, language-level, resource management and measurement

Zachary Anderson
Systems Group, ETH Zürich

Barrelfish Workshop
October 20th, 2011

Goal

Goal

```
report (resource) {  
    ...  
}
```

- A “report” statement that profiles all usage of a particular resource in a block of code

Goal

```
report (resource) {  
    ...  
}
```

- A “report” statement that profiles all usage of a particular resource in a block of code

Goal

```
report (resource) {  
    ...  
}
```

- A “report” statement that profiles all usage of a particular resource in a block of code

```
require (resource (a) ) {  
    ...  
}
```

- A “require” statement that provides guarantees and limits for usage of a resource in a block of code

Goal

```
report (resource) {  
    ...  
}
```

- A “report” statement that profiles all usage of a particular resource in a block of code

```
require (resource (a) ) {  
    ...  
}
```

- A “require” statement that provides guarantees and limits for usage of a resource in a block of code

resource = CPU util, IO bandwidth, etc.

Goal

```
report (resource) {  
    ...  
}
```

- A “report” statement that profiles all usage of a particular resource in a block of code

```
require (resource (a) ) {  
    ...  
}
```

- A “require” statement that provides guarantees and limits for usage of a resource in a block of code

resource = CPU util, IO bandwidth, etc.

Why?

Why?

- Apps beginning to compose parallel libraries, hierarchically

Why?

- Apps beginning to compose parallel libraries, hierarchically
- TBB calls OpenMP calls pthreads, etc.

Why?

- Apps beginning to compose parallel libraries, hierarchically
 - TBB calls OpenMP calls pthreads, etc.
- Apps running multiple, competing parallel tasks at the same time

Why?

- Apps beginning to compose parallel libraries, hierarchically
 - TBB calls OpenMP calls pthreads, etc.
- Apps running multiple, competing parallel tasks at the same time
- Stock OS scheduler is task ambivalent

Why?

- Apps beginning to compose parallel libraries, hierarchically
 - TBB calls OpenMP calls pthreads, etc.
- Apps running multiple, competing parallel tasks at the same time
- Stock OS scheduler is task ambivalent
 - Resources go equally to threads w/o regard for higher-level goals

Outline

- Semantics
- Implementation
- Extensions
 - Resource Kind definitions
 - Policy DSL
- Preliminary Results

Semantics

```
require (resource (a) ) {  
    . . .  
}
```

Semantics

```
require (resource (a) ) {  
    . . .  
}
```

- Guarantee: Use of amount **a** of **resource**

Semantics

```
require (resource (a) ) {  
    . . .  
}
```

- Guarantee: Use of amount **a** of **resource**
- Limit: May use no more than amount **a** of **resource**

Semantics

```
require (resource (a) ) {  
    . . .  
}
```

- Guarantee: Use of amount **a** of **resource**
- Limit: May use no more than amount **a** of **resource**
- Threads block until resources are available

Semantics

```
require (resource (a) ) {  
    ...  
    require (resource (b) ) {  
        ...  
    }  
}
```

Semantics

```
require (resource (a) ) {  
    ...  
    require (resource (b) ) {  
        ...  
    }  
}
```

- Nested allocations only out of current allocation

Semantics

```
require (resource (a) ) {  
    ...  
    require (resource (b) ) {  
        ...  
    }  
}
```

- Nested allocations only out of current allocation
- i.e. **b** \leq **a**

Semantics

```
require (resource (a) ) {  
    spawn f;  
}
```

Semantics

```
require (resource (a) ) {  
    spawn f;  
}
```

- All spawned threads equally share this allocation with parent

Semantics

```
require (resource (a) ) {  
    spawn f;  
}
```

- All spawned threads equally share this allocation with parent
- (until doing their own **require**)

Semantics

```
require (resource (a) ) {  
    spawn f;  
}
```

- All spawned threads equally share this allocation with parent
- (until doing their own **require**)
 - More on this later

Semantics

```
require(resource(a)) {  
    block();  
}
```

Semantics

```
require(resource(a)) {  
    block();  
}
```

- Threads release resources when blocking

Semantics

```
require(resource(a)) {  
    block();  
}
```

- Threads release resources when blocking
- On unblocking, must continue waiting until released resources can be reacquired

Semantics

```
require (CpuUtil (0, 50%)) {  
    ...  
}
```

```
require (cores (3)) {  
    ...  
}
```

Semantics

- **resource:**

```
require (CpuUtil (0, 50%) ) {  
    . . .  
}
```

```
require (cores (3) ) {  
    . . .  
}
```

Semantics

- **resource:**

```
require (CpuUtil (0, 50%)) {  
    ...  
}
```

- Either a basic Resource Kind

```
require (cores (3)) {  
    ...  
}
```

Semantics

- **resource:**

```
require (CpuUtil (0, 50%) ) {  
    ...  
}
```

- Either a basic Resource Kind

- Later: How these are defined

```
require (cores (3) ) {  
    ...  
}
```


Semantics

- **resource:**

```
require (CpuUtil (0, 50%) ) {  
  ...  
}
```

- Either a basic Resource Kind

- Later: How these are defined

```
require (cores (3) ) {  
  ...  
}
```

- Or a Policy

Semantics

- **resource:**

```
require (CpuUtil (0, 50%)) {  
    ...  
}
```

```
require (cores (3)) {  
    ...  
}
```

- Either a basic Resource Kind
- Later: How these are defined
- Or a Policy
- A *set* of basic resources

Semantics

- **resource:**

```
require (CpuUtil (0, 50%)) {  
    ...  
}
```

```
require (cores (3)) {  
    ...  
}
```

- Either a basic Resource Kind
- Later: How these are defined
- Or a Policy
 - A *set* of basic resources
 - Based on availability

Semantics

- **resource:**

```
require (CpuUtil (0, 50%)) {  
    ...  
}
```

```
require (cores (3)) {  
    ...  
}
```

- Either a basic Resource Kind
- Later: How these are defined
- Or a Policy
 - A *set* of basic resources
 - Based on availability
 - Later: A DSL for defining these

Options

Options

- From *when* is the **require** enforced?

Options

- From ***when*** is the **require** enforced?
 - Now? Before/after spawning a thread?

Options

- From ***when*** is the **require** enforced?
 - Now? Before/after spawning a thread?
- ***Who*** may share the allocation?

Options

- From ***when*** is the **require** enforced?
 - Now? Before/after spawning a thread?
- ***Who*** may share the allocation?
 - No one? Only child threads? Anyone?

Options

- From *when* is the **require** enforced?
 - Now? Before/after spawning a thread?
- *Who* may share the allocation?
 - No one? Only child threads? Anyone?
- When *shall* child threads make sub-allocations?

Options

- From *when* is the **require** enforced?
 - Now? Before/after spawning a thread?
- *Who* may share the allocation?
 - No one? Only child threads? Anyone?
- When *shall* child threads make sub-allocations?
 - On thread start? When calling **spawn**?

Options

- From *when* is the **require** enforced?
 - Now? Before/after spawning a thread?
- *Who* may share the allocation?
 - No one? Only child threads? Anyone?
- When *shall* child threads make sub-allocations?
 - On thread start? When calling **spawn**?
- When *may* child threads make sub-allocations?

Options

- From *when* is the **require** enforced?
 - Now? Before/after spawning a thread?
- *Who* may share the allocation?
 - No one? Only child threads? Anyone?
- When *shall* child threads make sub-allocations?
 - On thread start? When calling **spawn**?
- When *may* child threads make sub-allocations?
 - Anytime? Only when parent blocks?

When

```
require (r (OnSpawn, a) ) {  
  . . .  
}
```

```
require (r (AferSpawn, a) ) {  
  . . .  
}
```

When

```
require (r (OnSpawn, a) ) {  
  . . .  
}
```

```
require (r (AferSpawn, a) ) {  
  . . .  
}
```

When

```
require (r (OnSpawn, a) ) {  
    . . .  
}
```

- Requirement enforced immediately *before* first call to **spawn**

```
require (r (AferSpawn, a) ) {  
    . . .  
}
```


When

```
require (r (OnSpawn, a) ) {  
    . . .  
}
```

- Requirement enforced immediately *before* first call to **spawn**

```
require (r (AferSpawn, a) ) {  
    . . .  
}
```

When

```
require (r (OnSpawn, a) ) {  
    ...  
}
```

- Requirement enforced immediately *before* first call to **spawn**

```
require (r (AferSpawn, a) ) {  
    ...  
}
```

- Requirement enforced immediately *after* first call to **spawn**

Who

```
require (r (Private, a) ) {  
  . . .  
}
```

```
require (r (Shared, a) ) {  
  . . .  
}
```

Who

```
require (r (Private, a) ) {  
    ...  
}
```

- No sub-allocations are permitted. (But nested **requires** can take from any remaining non-private allocation.)

```
require (r (Shared, a) ) {  
    ...  
}
```

Who

```
require (r (Private, a) ) {  
    ...  
}
```

- No sub-allocations are permitted. (But nested **requires** can take from any remaining non-private allocation.)

```
require (r (Shared, a) ) {  
    ...  
}
```

Who

```
require (r (Private, a) ) {  
    . . .  
}
```

- No sub-allocations are permitted. (But nested **requires** can take from any remaining non-private allocation.)

```
require (r (Shared, a) ) {  
    . . .  
}
```

- Limits thread to amount **a** of resource **r**, but provides no guarantee

Child thread allocations

```
require (r (ForChild, a) ) {  
    . . .  
}
```

Child thread allocations

```
require (r (ForChild, a) ) {  
  . . .  
}
```

- **spawn**'d child threads execute effects of the **require** statement immediately after starting

Child thread allocations

```
require (r (OnBlock, a) ) {  
    . . .  
}
```

Child thread allocations

```
require (r (OnBlock, a) ) {  
    . . .  
}
```

- Blocks until parent thread blocks, then attempts executing **require**.

Options

- These four kinds of options are orthogonal and can be combined
- e.g.: **ForChild**, **OnSpawn**
- Execute the **require** when a child thread itself calls **spawn**

Implementation Sketch

Implementation Sketch

- For each resource, each thread keeps:

Implementation Sketch

- For each resource, each thread keeps:
 - A stack mirroring the nesting of **require** statements

Implementation Sketch

- For each resource, each thread keeps:
 - A stack mirroring the nesting of **require** statements
- Globally:

Implementation Sketch

- For each resource, each thread keeps:
 - A stack mirroring the nesting of **require** statements
- Globally:
 - Keep a tree of allocations

Implementation Sketch

- For each resource, each thread keeps:
 - A stack mirroring the nesting of **require** statements
- Globally:
 - Keep a tree of allocations
 - Nodes keep some reference counts, e.g.:

Implementation Sketch

- For each resource, each thread keeps:
 - A stack mirroring the nesting of **require** statements
- Globally:
 - Keep a tree of allocations
 - Nodes keep some reference counts, e.g.:
 - Number of threads sharing an allocation

Implementation Sketch

- For each resource, each thread keeps:
 - A stack mirroring the nesting of **require** statements
- Globally:
 - Keep a tree of allocations
 - Nodes keep some reference counts, e.g.:
 - Number of threads sharing an allocation
 - Number of those that are running

Implementation Sketch

- For each resource, each thread keeps:
 - A stack mirroring the nesting of **require** statements
- Globally:
 - Keep a tree of allocations
 - Nodes keep some reference counts, e.g.:
 - Number of threads sharing an allocation
 - Number of those that are running
 - Stack entries point to nodes in the tree

Implementation Sketch

Implementation Sketch

- Need to perform operations on stacks, trees when threads:

Implementation Sketch

- Need to perform operations on stacks, trees when threads:
 - Start, **require**, block, and exit

Implementation Sketch

- Need to perform operations on stacks, trees when threads:
 - Start, **require**, block, and exit
- So, use dynamic linker to override:

Implementation Sketch

- Need to perform operations on stacks, trees when threads:
 - Start, **require**, block, and exit
- So, use dynamic linker to override:
 - pthread_create, pthread_exit, pthread_cond_wait, etc.

Implementation Sketch

- Need to perform operations on stacks, trees when threads:
 - Start, **require**, block, and exit
- So, use dynamic linker to override:
 - pthread_create, pthread_exit, pthread_cond_wait, etc.
 - sched_yield, etc.


Implementation Sketch

- C front-end
 - OCaml CIL library
 - Replaces **require** with calls to runtime library
- Runtime in C
 - 8 calls, a few data types
 - Hopefully easy to integrate into other languages

Example

r-Allocation Tree

```
f ( ) {  
    require ( r ( b ) ) {  
        . . .  
    }  
}
```

```
main ( ) {   
    require ( r ( a ) ) {  
        spawn f;  
        require ( r ( c ) ) { . . . }  
    }  
}
```

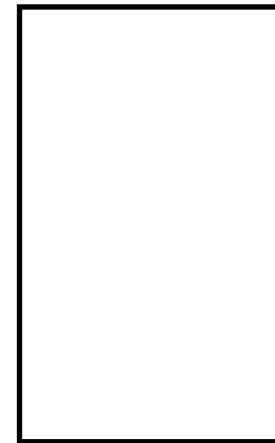
Example

r-Allocation Tree

```
f () {  
    require (r (b)) {  
        . . .  
    }  
}
```

G

T1 r-Stack



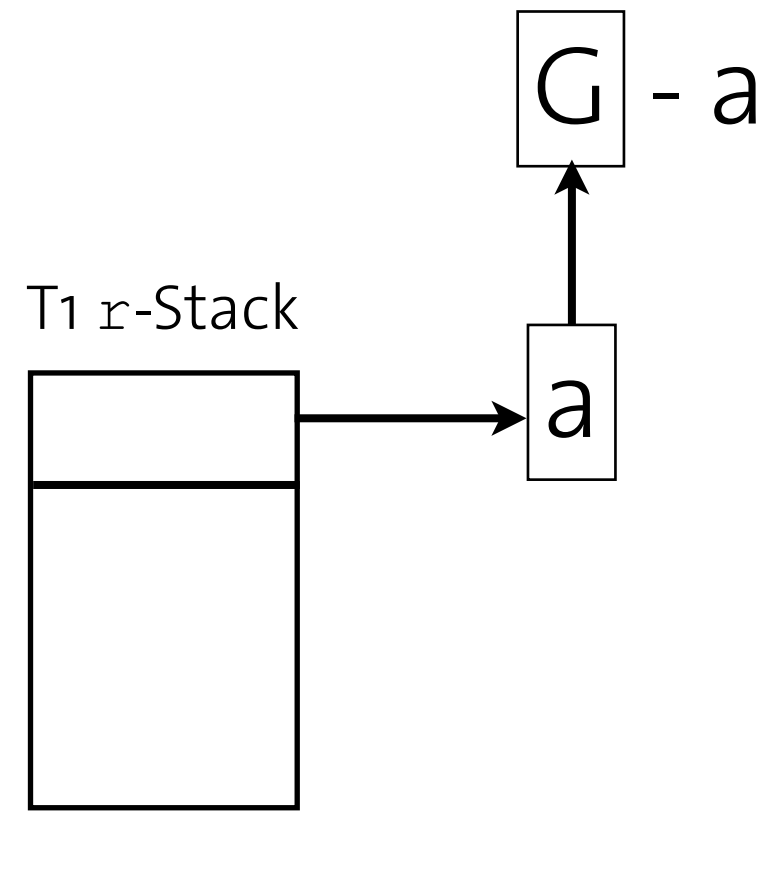
```
main () {  
    require (r (a)) {  
        spawn f;  
        require (r (c)) { . . . }  
    }  
}
```



Example

```
f () {  
    require (r (b)) {  
        ...  
    }  
}  
  
main () {  
    require (r (a)) {  
        spawn f;  
        require (r (c)) { ... }  
    }  
}
```

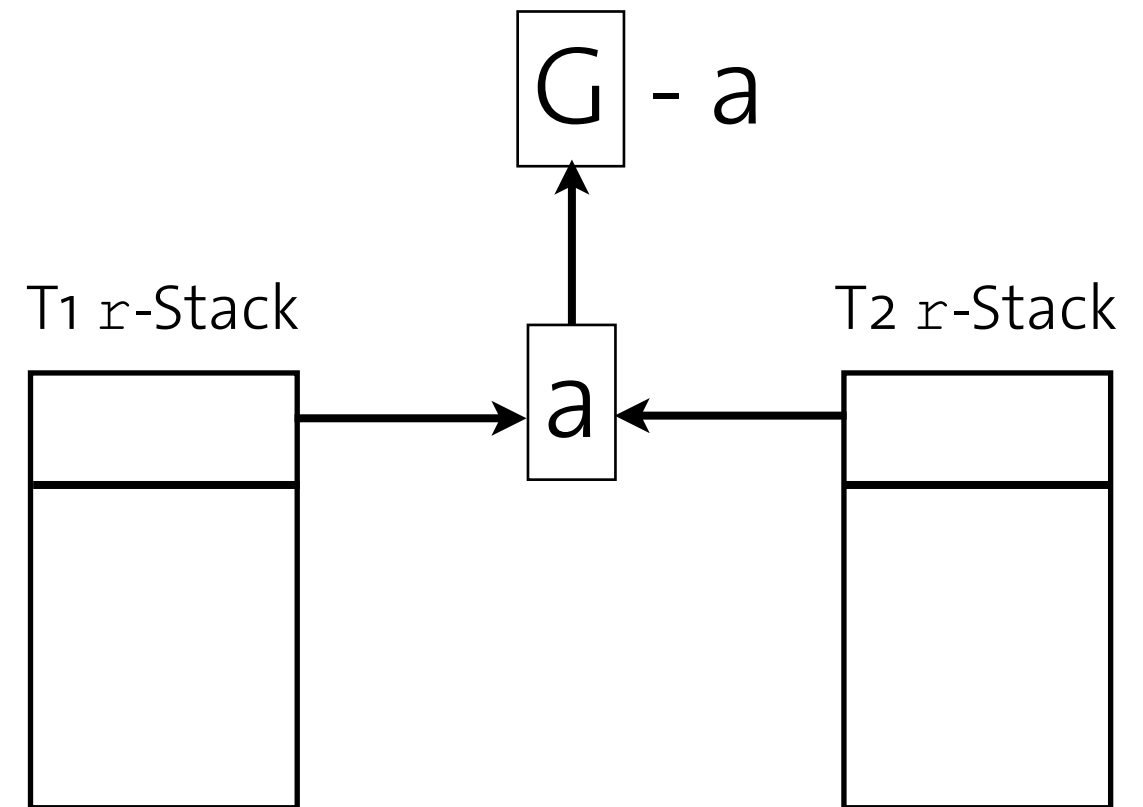
r-Allocation Tree



Example

```
f () {  
    require (r (b)) {  
        ...  
    }  
}  
  
main () {  
    require (r (a)) {  
        spawn f;  
        require (r (c)) { ... }  
    }  
}
```

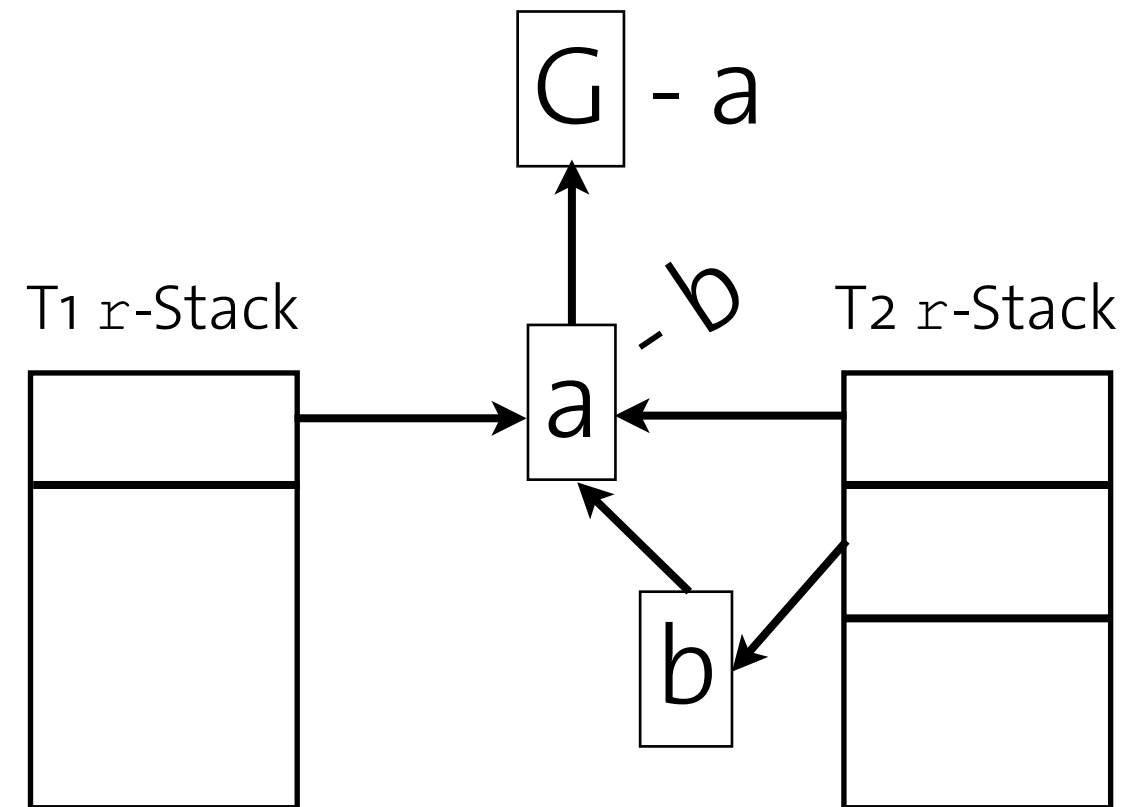
r-Allocation Tree



Example

```
f () {  
  require (r (b)) { ←  
    ...  
  }  
}  
  
main () {  
  require (r (a)) {  
    spawn f;  
    require (r (c)) { ... }  
  }  
}
```

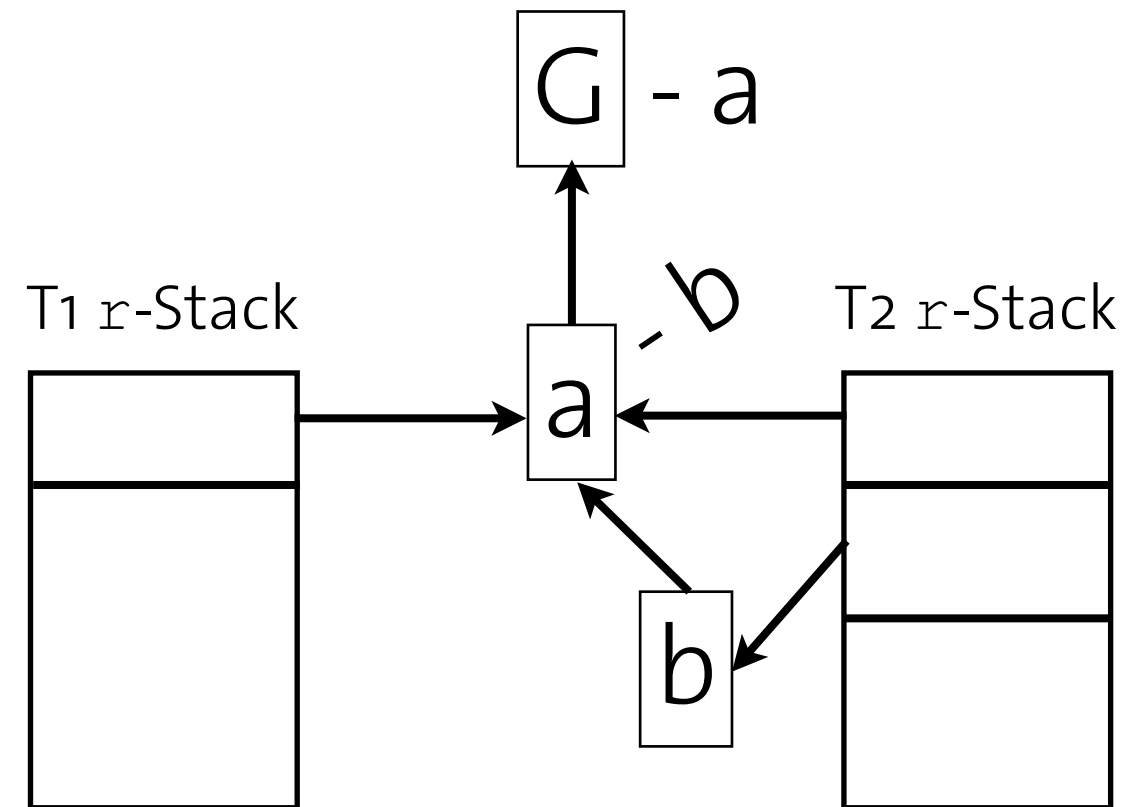
r-Allocation Tree



Example

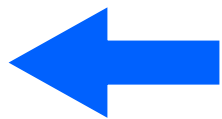
```
f () {  
  require (r (b)) {  
    ...  
  }  
}  
  
main () {  
  require (r (a)) {  
    spawn f;  
    require (r (c)) { ... }  
  }  
}
```

r-Allocation Tree

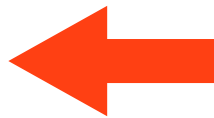


Example

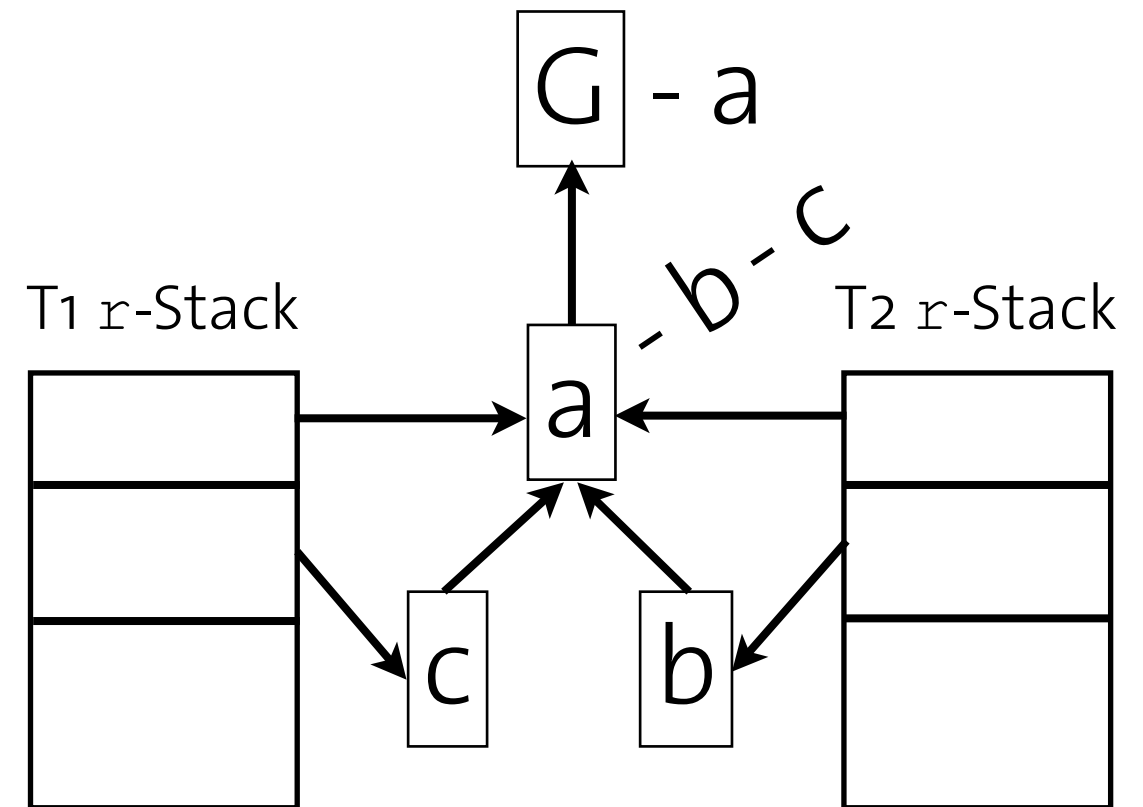
```
f () {  
  require (r (b)) {  
    ...  
  }  
}
```



```
main () {  
  require (r (a)) {  
    spawn f;  
    require (r (c)) { ... }  
  }  
}
```



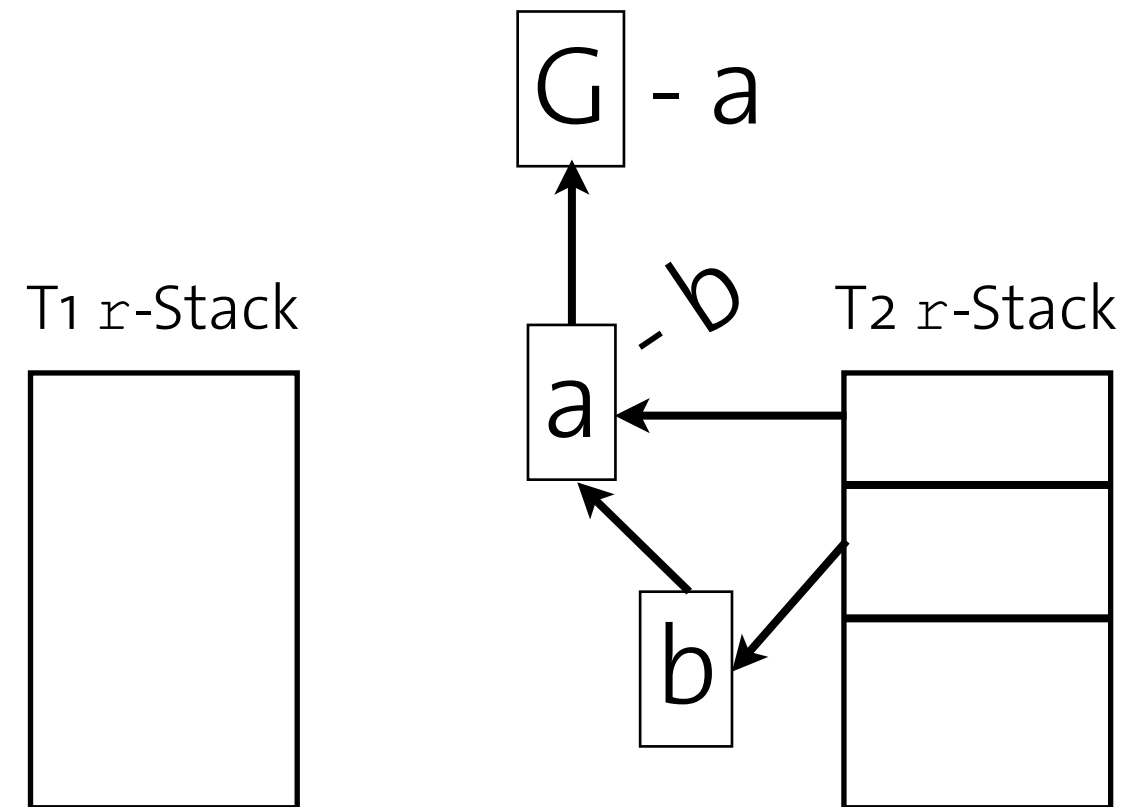
r-Allocation Tree



Example

```
f () {  
  require (r (b)) {  
    ...  
  }  
}  
  
main () {  
  require (r (a)) {  
    spawn f;  
    require (r (c)) { ... }  
  }  
}
```

r-Allocation Tree



Resource Kinds

Resource Kinds

- Define a structure, register with API call:

Resource Kinds

- Define a structure, register with API call:
 - Name, number of devices, names of devices, per-device max allocation

Resource Kinds

- Define a structure, register with API call:
 - Name, number of devices, names of devices, per-device max allocation
 - Operations (all optional):

Resource Kinds

- Define a structure, register with API call:
 - Name, number of devices, names of devices, per-device max allocation
 - Operations (all optional):
 - Get current usage

Resource Kinds

- Define a structure, register with API call:
 - Name, number of devices, names of devices, per-device max allocation
- Operations (all optional):
 - Get current usage
 - Calculate usage rate

Resource Kinds

- Define a structure, register with API call:
 - Name, number of devices, names of devices, per-device max allocation
- Operations (all optional):
 - Get current usage
 - Calculate usage rate
 - OS function for throttling usage

Resource Kinds

- Define a structure, register with API call:
 - Name, number of devices, names of devices, per-device max allocation
- Operations (all optional):
 - Get current usage
 - Calculate usage rate
 - OS function for throttling usage
 - OS function for pinning to device

Example (CpuUtil)

Example (CpuUtil)

- Name: CpuUtil

Example (CpuUtil)

- Name: `CpuUtil`
- Number of Devices: `NUM_CPUS`

Example (CpuUtil)

- Name: `CpuUtil`
- Number of Devices: `NUM_CPUS`
- Names of Devices: `[0, ..., n]`

Example (CpuUtil)

- Name: CpuUtil
- Number of Devices: NUM_CPUS
- Names of Devices: $[0, \dots, n]$
- Max Allocation: $[1.0, \dots, 1.0]$

Example (CpuUtil)

Example (CpuUtil)

- Operations:

Example (CpuUtil)

- Operations:
 - Getter: CPU time on behalf of thread

Example (CpuUtil)

- Operations:
 - Getter: CPU time on behalf of thread
 - Calculator: $(\text{getter_2} - \text{getter_1}) / \text{Time}$

Example (CpuUtil)

- Operations:
 - Getter: CPU time on behalf of thread
 - Calculator: $(\text{getter_2} - \text{getter_1}) / \text{Time}$
 - Throttler: None

Example (CpuUtil)

- Operations:
 - Getter: CPU time on behalf of thread
 - Calculator: $(\text{getter_2} - \text{getter_1}) / \text{Time}$
 - Throttler: None
 - Pinner: Linux cpuset cgroup

Resource Kinds

Resource Kinds

- Now we can write:

Resource Kinds

- Now we can write:
 - `require (CpuUtil (core, util))`

Resource Kinds

- Now we can write:
 - `require (CpuUtil (core, util))`
- All platform specific stuff is in the Resource Kind definitions

Resource Kinds

- Now we can write:
 - `require (CpuUtil (core, util))`
- All platform specific stuff is in the Resource Kind definitions
- Hopefully this makes porting easy

Policies

Policies

- Query availability, **require** a set of resources

Policies

- Query availability, **require** a set of resources
 - e.g. n cores, don't care which

Policies

- Query availability, **require** a set of resources
 - e.g. n cores, don't care which
- Allocated and released all-together

Policies

- Query availability, **require** a set of resources
 - e.g. n cores, don't care which
- Allocated and released all-together
- Defined by DSL embedded in C

Policies

- Query availability, **require** a set of resources
 - e.g. n cores, don't care which
- Allocated and released all-together
- Defined by DSL embedded in C
 - No writing globals, or calling arbitrary functions

Policies

- Query availability, **require** a set of resources
 - e.g. n cores, don't care which
- Allocated and released all-together
- Defined by DSL embedded in C
 - No writing globals, or calling arbitrary functions
 - But a few special functions are made available

Policy DSL Example

```
policy cores(int n) {  
    int i, found = 0;  
    int d = num_devs("CpuUtil");  
  
    for (i = 0 ... d) {  
        if (found == n) break;  
        if (available("CpuUtil", i) == 1.0) {  
            require("CpuUtil", i).value = 1.0;  
            found++;  
        }  
    }  
    if (found < n) return PolicyFailure;  
    return PolicySuccess;  
}
```

Preliminary Results

- QR decomposition of big matrices:
 - Intel TBB -> Intel MKL -> GNU OpenMP
 - “deltaX” matrix on 4-core Intel box
 - Using only the “cores” policy

Preliminary Results

- QR decomposition of big matrices:
 - Intel TBB -> Intel MKL -> GNU OpenMP
 - “deltaX” matrix on 4-core Intel box
 - Using only the “cores” policy

Method	L2 Miss Rate	Context Switches	CPU Migration	Changes	Time(s)
Default	3.2%	2.6×10^5	4891	0	96.8
Lithe <small>[Pan, et al. PLDI'10]</small>	2.3%	4.6×10^4	27	Custom TBB, OMP	89.7
Us	3.0%	4.0×10^4	41	4	85.9

Continuing Work

Continuing Work

- Try out on more apps

Continuing Work

- Try out on more apps
- Write a library of useful policies

Continuing Work

- Try out on more apps
- Write a library of useful policies
- Think of a clever name

Continuing Work

- Try out on more apps
- Write a library of useful policies
- Think of a clever name
- more resources => rerun policy function?

Continuing Work

- Try out on more apps
- Write a library of useful policies
- Think of a clever name
- more resources => rerun policy function?
- Port to Barrelfish

Continuing Work

- Try out on more apps
- Write a library of useful policies
- Think of a clever name
- more resources => rerun policy function?
- Port to Barrelfish
 - Seek cooperation between OS and runtime

Continuing Work

- Try out on more apps
- Write a library of useful policies
- Think of a clever name
- more resources => rerun policy function?
- Port to Barrelfish
 - Seek cooperation between OS and runtime
 - Single machine -> Cluster?

questions