Dynamic X10
Resource-Aware Programming for Higher Efficiency

Manuel Mohr, Andreas Zwinkau, Sebastian Buchwald, Matthias Braun
Motivation

Static exclusive resource allocation decreases overall efficiency.
Motivation

Static exclusive resource allocation decreases overall efficiency
Motivation

Static exclusive resource allocation decreases overall efficiency.
Static exclusive resource allocation decreases overall efficiency
Allocate resources exclusively

But make the allocation dynamically changeable (i.e., at run-time)

⇒ Resizable claims
Dynamic Exclusive Resource Allocation

- Allocate resources exclusively
- But make the allocation dynamically changeable (i.e., at run-time)
  $\Rightarrow$ Resizable claims

![Diagram showing resource allocation over time with resizable claims](image-url)
Dynamic Exclusive Resource Allocation

- Allocate resources exclusively
- But make the allocation dynamically changeable (i.e., at run-time)

⇒ Resizable claims
Dynamic Exclusive Resource Allocation

Allocate resources exclusively

But make the allocation dynamically changeable (i.e., at run-time)

⇒ Resizable claims
Dynamic Exclusive Resource Allocation

- Allocate resources exclusively
- But make the allocation dynamically changeable (i.e., at run-time)

⇒ Resizable claims

![Resource Need vs. Time Graph]

- Resource Need
- Time

- Diagram showing resource allocation over time with resizable claims.
Scope of This Talk

Dynamic X10: framework for programming with resizable claims
- Integration into existing X10: what needs to change?
- Resource management for improving global efficiency
- Brief evaluation of programming effort for developers
Claim: set of cores with exclusive access
- In general: different kinds of resources
- Application’s view is restricted to resources in its claim
Claims can be resized with method `reinvade()`.

Claims can span multiple shared-memory domains ⇒ can contain multiple places.
Claims can be resized with method `reinvade()`

Claims can span multiple shared-memory domains

⇒ can contain multiple places
- Claims can be resized with method `reinvade()`.
- Claims can span multiple shared-memory domains ⇒ can contain multiple places.
- Dynamic X10: places can appear and disappear at runtime.
Natural Embedding: Traditional X10 program behaves like Dynamic X10 program running inside claim containing all cores
Alternative to resizing existing claim: creating a new claim

Useful property: isolation

- Concurrent resizing requests using `reinvade()` ⇒ bad

Method to create and destroy claims: `invade()` and `retreat()`
Multiple Claims and X10

\[
\text{infect(} \hat{=} \text{at on claim-level}
\]

Closure as argument (with deep copy semantics)

Closure runs inside the new claim

Navigation inside new claim with \texttt{at} and \texttt{async}
Multiple Claims and X10

\[\text{infect()} \hat{=} \text{at on claim-level}\]

Closure as argument (with deep copy semantics)

Closure runs inside the new claim

Navigation inside new claim with `at` and `async`
Multiple Claims and X10

\begin{align*}
\text{infect() } \triangleq \text{ at on claim-level} \\
\quad \text{ Closure as argument (with deep copy semantics)} \\
\quad \text{ Closure runs } \textit{inside} \text{ the new claim} \\
\quad \text{ Navigation inside new claim with } \texttt{at} \text{ and } \texttt{async}
\end{align*}
val claim = Claim.getCurrentClaim();
claim.reinvade(...);

at (here.next()) async {
    /* do work */
}

val newClaim = Claim.invade(...);
newClaim.infect(() => {
    /* do more work */
});
Example Usage

```scala
val claim = Claim.getCurrentClaim();
claim.reinvade(...);
at (here.next()) async { /* do work */ }
val newClaim = Claim.invade(...);
newClaim.infect(() => { /* do more work */ });
```

Question: Who changes the allocation? And how?

**The application**

- No information about other claims

⇒ Does not optimize well in multi-application scenario
Example Usage

```scala
val claim = Claim.getCurrentClaim();
claim.reinvade(...);

at (here.next()) async {
  /* do work */
}

val newClaim = Claim.invade(...);
newClaim.infect(() => {
  /* do more work */
});
```

**Question:** Who changes the allocation? And how?

**The application**
- No information about other claims
  - Does not optimize well in multi-application scenario

**The operating system**
- Not enough information about applications
  - Does not optimize well in multi-application scenario
Idea: pass application-specific information to OS
⇒ invade()/reinvade() take resource description: constraints
Modeled as class hierarchy in X10, can be combined using && and ||
Most important: PEQuantity and ScalabilityCurve

Claim.invoke(new PEQuantity(1, 3)
               && new ScalabilityCurve([130, 150, 160]));

Scalability information via offline experiments or online monitoring
Necessary X10 Adaption

1. Static fields → methods
   - Place.MAX_PLACES → Place.maxPlaces() to query current claim

2. Adapt method implementations
   - Place.places() must query current claim

3. Usage of DistArrays
DistArrays
DistArrays
DistArrays

- Create new DistArray, move data (library routine)
DistArrays

- Create new DistArray, move data (library routine)
What if we lose places? ⇒ data inaccessible
Solution 1: “Sticky claims”
- Forbid losing places
- Simple, but severely restricts reallocation $\Rightarrow$ decreased efficiency
Solution 2: Give opportunity to redistribute

- Requires programmer effort
- More flexible resource reallocation
Solution 2: Give opportunity to redistribute

- Requires programmer effort
- More flexible resource reallocation
Solution 2: Give opportunity to redistribute
- Requires programmer effort
- More flexible resource reallocation
Prototype implementation of Dynamic X10

- Multigrid: Calls \texttt{reinvade()} when resizing grid
  - Uses grid size to compute number of cores
- Prototype uses sticky claims $\Rightarrow$ no disappearing places
- \texttt{DistArray} redistribution after \texttt{reinvade()}
- $\approx 50$ lines of additional code compared to traditional X10
Conclusion

We have:
- Integrated claim concept and dynamic exclusive allocation into X10
- Adapted X10 to deal with varying number of places
- Shown that constraints can solve resource management problem
- Implemented a first version of “Dynamic X10”
- Given a preliminary evaluation of programming effort
Neither exclusive, nor static resource allocation:
- Resource virtualization: threads instead of CPUs
- Threads multiplexed onto physical cores

Not for free:
- \#threads \gg \#cores
- Cache issues
- Scheduler has no application-specific knowledge (e.g., barrier synchronization)
Dynamic X10 implemented in the scope of “Invasive Computing” project

Looks at future tiled many-core architectures (> 1000 cores)
  - Share characteristics of supercomputers

Custom operating system
  - Claims as central data structure
  - Claim-aware scheduler