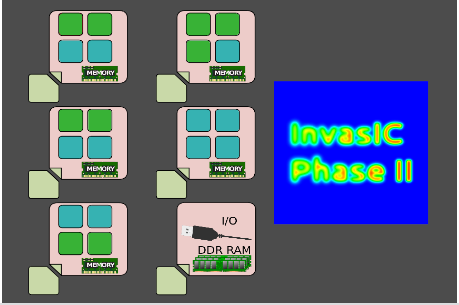
Karlsruhe Institute of Technology

RaPping and Compilation for Highly Dynamic Parallelism

Gregor Snelting

Presentation at ANDREAS-50, 26.2.16



Background

SFB Transregio **InvasIC**: FAU, KIT, TUM

- new language, soft- and hardware for highly dynamic parallelism
- 3 universities, 15 PIs, 50 doctoral researchers
- DFG funding 2010–2018 (2022?)

Background

SFB Transregio **InvasIC**: FAU, KIT, TUM

- new language, soft- and hardware for highly dynamic parallelism
- 3 universities, 15 PIs, 50 doctoral researchers
- DFG funding 2010–2018 (2022?)

Scenario: big heterogenous PGAS architectures, several kCores / Tile

Background

SFB Transregio **InvasIC**: FAU, KIT, TUM

- new language, soft- and hardware for highly dynamic parallelism
- 3 universities, 15 PIs, 50 doctoral researchers
- DFG funding 2010–2018 (2022?)

Scenario: big heterogenous PGAS architectures, several kCores / Tile
Goal: support highly dynamic parallelism through all system levels

Background

Karlsruhe Institute of Technology

SFB Transregio **InvasIC**: FAU, KIT, TUM

- new language, soft- and hardware for highly dynamic parallelism
- 3 universities, 15 PIs, 50 doctoral researchers
- DFG funding 2010–2018 (2022?)

Scenario: big heterogenous PGAS architectures, several kCores / Tile
Goal: support highly dynamic parallelism through all system levels

DFG Sonderforschungsbereich/Transregio 89
Transregional Collaborative Research Center 89

RaP to the Future

Fundamental idea: Resource-aware programming

- applications dynamically determine their resource requirements
- allocate and release resources (in particular CPUs) dynamically
- supported by language, compiler, OS, and dedicated hardware

$\Longrightarrow$ *optimize resource utilization, energy consumption, fault tolerance*

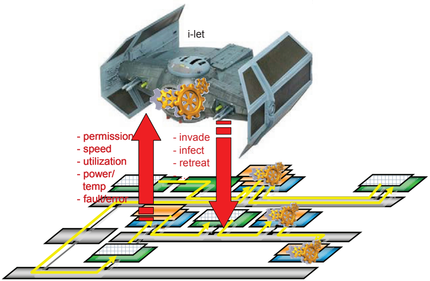
RaP to the Future

Fundamental idea: Resource-aware programming

- applications dynamically determine their resource requirements
- allocate and release resources (in particular CPUs) dynamically
- supported by language, compiler, OS, and dedicated hardware

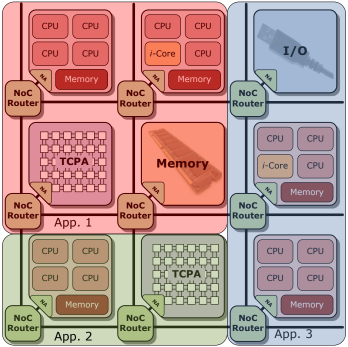
⟹ *optimize resource utilization, energy consumption, fault tolerance*

Slogan (J. Teich): “Invade, infect, combat, retreat”



Example: InvasIC Multi-Tile MPSoC

tiles: RISC multicore, memory, I/O, TCPA, ...
3 applications have invaded the example tiles



Demonstrator implemented via FPGAs

Fundamental InvasIC Ingredients

- Dynamic X10 + Compiler: see below
- constraint system: attribute hierarchy for resource selection
- OctoPos OS: dedicated scheduling, load balancing
- Agent system: priorizes / allocates dynamic resource requests
- CiC: hardware support on local tile for app code dispatch / monitoring
- NoC: network on chip
- i-core: adaptive hardware supporting specific accelerators
- TCPAs: support (numerical) loop parallelization

Fundamental InvasIC Ingredients

- Dynamic X10 + Compiler: see below
- constraint system: attribute hierarchy for resource selection
- OctoPos OS: dedicated scheduling, load balancing
- Agent system: priorizes / allocates dynamic resource requests
- CiC: hardware support on local tile for app code dispatch / monitoring
- NoC: network on chip
- i-core: adaptive hardware supporting specific accelerators
- TCPAs: support (numerical) loop parallelization

Further aspects:

- applications: HPC, robotics
- dark silicon: dynamic selection of non-overheated cores etc.
- predictability: soft realtime and security
- hardware sensors: temperature, speed, fault status etc

...

PP Group @ InvasIC

People: Gregor Snelting, (Matthias Braun),
Sebastian Buchwald, Manuel Mohr, Andreas Zwinkau

PP Group @ InvasIC

People: Gregor Snelting, (Matthias Braun),
Sebastian Buchwald, Manuel Mohr, Andreas Zwinkau

Contributions:

- framework design (based on X10)
- fundamental RaP examples
- compiler, library, RTS
- code generation for MPSoCs
- optimizations for InvasIC OS / hardware
- interfaces to simulator, OS, TCPA subcompiler [FAU], ...
- application support

Basic RaP Patterns & Constraints

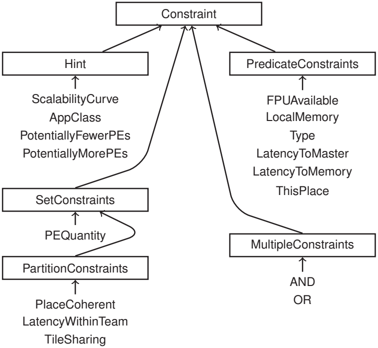
fundamental “invade/infect/retreat” pattern (X10) uses closure for app code:

```
val ilet = (id:IncarnationID) => {
    do_something(id);
};
claim = Claim.invade(constraints)
claim.infect(ilet)
claim.retreat()
```

Basic RaP Patterns & Constraints

fundamental “invade/infect/retreat” pattern (X10) uses closure for app code:

```
val ilet = (id:IncarnationID) => {
    do_something(id);
};
claim = Claim.invade(constraints)
claim.infect(ilet)
claim.retreat()
```



Constraint

Hint

ScalabilityCurve
AppClass
PotentiallyFewerPEs
PotentiallyMorePEs

PredicateConstraints

FPUAvailable
LocalMemory
Type
LatencyToMaster
LatencyToMemory
ThisPlace

SetConstraints

PEQuantity

PartitionConstraints

PlaceCoherent
LatencyWithinTeam
TileSharing

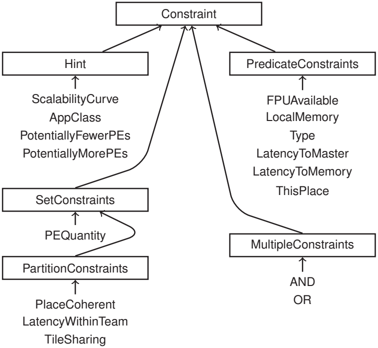
MultipleConstraints

AND
OR

Basic RaP Patterns & Constraints

fundamental “invade/infect/retreat” pattern (X10) uses closure for app code:

```
val ilet = (id:IncarnationID) => {
    do_something(id);
};
claim = Claim.invade(constraints)
claim.infect(ilet)
claim.retreat()
```



Constraint

Hint

ScalabilityCurve
AppClass
PotentiallyFewerPEs
PotentiallyMorePEs

PredicateConstraints

FPUAvailable
LocalMemory
Type
LatencyToMaster
LatencyToMemory
ThisPlace

SetConstraints

PEQuantity

PartitionConstraints

PlaceCoherent
LatencyWithinTeam
TileSharing

MultipleConstraints

AND
OR

“`reinvade()`” allows OS to adapt resources
“`reinvade(nC)`” allows app to reallocate

```
claim.infect(ilet)
// optimize resource allocation:
val changed1 = claim.reinvade()
claim.infect(ilet)
// respecify resource needs
val changed2 = claim.reinvade(otherConstraints)
```

Example: Invasive Quicksort

Sequential X10 code:

```
public static def sort(data: Array[Int](1)) =
  qsort(data, 0, data.size-1);

private static def qsort(
  data: Array[Int](1),
  left: Int,
  right: Int)
{
  if (data.size>1) {
    val p = partition(data, left, right);
    val i = p.first;
    val j = p.second;
    qsort(data,  left,   i);
    qsort(data,  j,      right);
  }
}
```

Goal: Parallelize Recursion

Pedagogical engineering on this part:

```
qsort(data, left, j    );
qsort(data, i,    right);
```

Requirements:

- Invasive
- Multi-core
- Multi-tile
- Dynamically Resource-aware

Disclaimer:

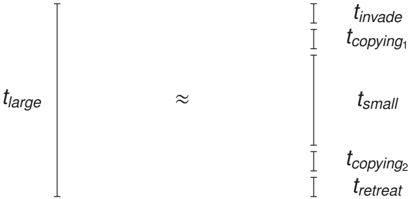
- *Pedagogical* example
  e.g. does not parallelize `partition` step

Handling the two parts

- After decision to parallelize: two parallel program paths
  1. `qsort()`
  2. `invade()` + (copying) + `qsort()` + (copying) + `retreat()`
- Important: *large* and *small* part, not left and right
- Hope: $t_{large} \approx t_{invade} + t_{copying_1} + t_{small} + t_{copying_2} + t_{retreat}$

⇒ Always handle large part locally

⇒ try to handle small part on new PE



t_{large} $\approx$

t_{invade}

$t_{copying_1}$

t_{small}

$t_{copying_2}$

$t_{retreat}$

RaP 1: Invasive Single-Tile

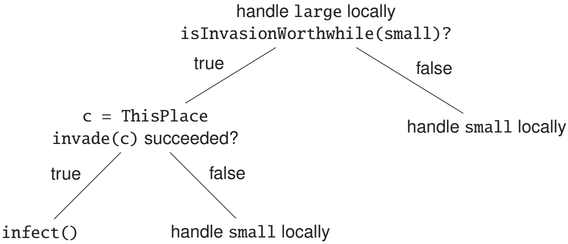
small part runs on new core in local tile (`X10`: “`ThisPlace`”)

RaP 1: Invasive Single-Tile

small part runs on new core in local tile (X10: “`ThisPlace`”)

“`isInvasionWorthwhile()`”: user defined; checks new thread for small part; based on problem size, processor speed, invade latencies, ...

Decision tree:



handle `large` locally
`isInvasionWorthwhile(small)?`

true

false

`c = ThisPlace`
`invade(c)` succeeded?

handle `small` locally

true

false

`infect()`

handle `small` locally

RaP 1: X10 Code

```
finish {
  if (large.size > 1) { async qsort(large); }
  if (isInvasionWorthwhile(small)) {
    val constraints = new AND(); // constraint system
    constraints.add(new PEQuantity(0, 1));
    constraints.add(new ThisPlace());
    constraints.add(new ScalabilityHint([100, 190]));
    val claim = Claim.invade(constraints);
    if (claim.size() == 1) {
      val ilet  = (id: IncarnationID) => qsort(small);
      claim.infect(ilet);
    } else { qsort(small); }
    claim.retreat();
  } else if (small.size > 1) {
    qsort(small);
  }
}
```

RaP 2: Invasive Multi-Tile

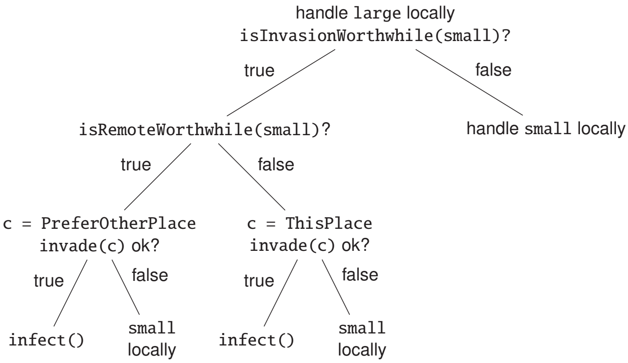
small part runs on new tile

RaP 2: Invasive Multi-Tile

small part runs on new tile

"`isRemoteWorthwhile()`" checks new tile for small part; based on problem size, tile attributes, network latencies, ...

Decision tree:



RaP 2: X10 Code

```
finish {
  if (large.size > 1) { async qsort(large); }
  if (isInvasionWorthwhile(small)) {
    val constraints = new AND();
    constraints.add(new PEQuantity(0, 1));
    if (isRemoteWorthwhile(small)) {
      constraints.add(new PreferOtherPlace());
      constraints.add(new ScalabilityHint([100, 160]);
    } else {
      constraints.add(new ThisPlace());
      constraints.add(new ScalabilityHint([100, 190]);
    }
    val claim = Claim.invade(constraints);
    if (claim.size() == 1) {
      val ilet = (id: IncarnationID) => qsort(small);
      claim.infect(ilet);
    } else {
      qsort(small);
    }
    claim.retreat();
  } else if (small.size > 1) {
    qsort(small);
  }
} // finish
```

RaP 3: Fine Tuning

Other sensible constraints in this example:
`ThroughputToMaster`, `ThroughputToMemory`, `ThroughputWithinTeam`, `LatencyToMaster`, `LatencyToMemory`, `LatencyWithinTeam`, . . .

RaP 3: Fine Tuning

Other sensible constraints in this example:
`ThroughputToMaster`, `ThroughputToMemory`, `ThroughputWithinTeam`, `LatencyToMaster`, `LatencyToMemory`, `LatencyWithinTeam`, ...

General Principle:

- Application dynamically crafts its constraints based on *locally* available data, e.g. the input size
- The *larger* the input size, the *weaker* the constraints
  - Three “levels” in quicksort example
- Application decides when it is certainly *not* sensible to parallelize
- Application supplies scalability hints for `invade()` call
- OS (“Agent system”) decides which application actually gets PEs

The Compiler

working compiler was built with only a few PJ!

- front end: adapted from IBM X10 compiler
- back end: PP@KIT, based on libFIRM ($\rightsquigarrow$ Goos, Hack et al.)
  “FIRM is like llvm, only better”

The Compiler

working compiler was built with only a few PJ!

- front end: adapted from IBM X10 compiler
- back end: PP@KIT, based on libFIRM (⇝ Goos, Hack et al.)
  "FIRM is like llvm, only better"
- generates code for SPARC (Leon 5)
- "invade" etc implemented as OctoPos calls
- compiler tested and integrated into demonstrator (FPGA system)

The Compiler

Karlsruhe Institute of Technology

working compiler was built with only a few PJ!

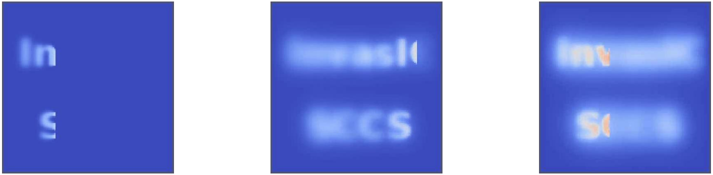
- front end: adapted from IBM X10 compiler
- back end: PP@KIT, based on libFIRM (⇝ Goos, Hack et al.)
  "FIRM is like llvm, only better"
- generates code for SPARC (Leon 5)
- "invade" etc implemented as OctoPos calls
- compiler tested and integrated into demonstrator (FPGA system)

Current work:

- specialised optimizations for InvasIC
  inference of app characteristics for better load balancing,
  infer minimal execution time for "`isInvasionWorthwhile()`",
  infer efficient i-core configurations (accelerators, permutation instructions, ...),
  efficient copying of data to other places,
  ...
- generate code for NoC, i-Core

Case Study: Multigrid Software

- Invasive multigrid: solver for part.diff.equations from TUM+KIT
- discretization with dynamically varying grid resolutions
- application: simulation of laser engraving on metal plate
  $\Longrightarrow$ heat equation (Laplace operator) must be discretized and solved
- grid levels: $N \times N$, $N/2 \times N/2$, $N/4 \times N/4$, ...
- result at timesteps 50/200/450:



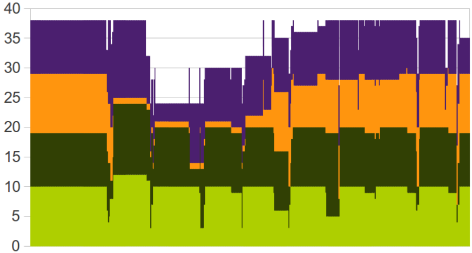
X10 Code + Dynamic Loads

```
vcycle(N, x, b):
r = computeResidual(N, x, b)
while |r| > threshold:
vcycleIteration(N, x, b)
r = computeResidual(N, x, b)


vcycleIteration(N, x, b):
smoother(N, x, b)       # pre-smooth
r = residual(N, x, b)   # residual

nc = reinvade(N, claim) # reinvade claim

Nr = N/2
# restricted level
rr = restrict(N, r)
# restrict residual to new claim
er = (Nr, 0)
# setup error with 0 values

nc2 = vcycleIteration(Nr, er, rr)
# vcycle

# redistribute
if (nc != nc2):
nc = nc2
x.redistribute(Nr, nc)
b.redistribute(Nr, nc)

e = prolongate(Nr, er)
# prolongate error
x = x + e
# apply correction

smoother(N, x, b)       # post-smooth

return nc  # possibly modified claim
```

X10 Code + Dynamic Loads

```
vcycle(N, x, b):
r = computeResidual(N, x, b)
while |r| > threshold:
vcycleIteration(N, x, b)
r = computeResidual(N, x, b)


vcycleIteration(N, x, b):
smoother(N, x, b)       # pre-smooth
r = residual(N, x, b)   # residual

nc = reinvade(N, claim) # reinvade claim

Nr = N/2
# restricted level
rr = restrict(N, r)
# restrict residual to new claim
er = (Nr, 0)
# setup error with 0 values

nc2 = vcycleIteration(Nr, er, rr)
# vcycle

# redistribute
if (nc != nc2):
nc = nc2
x.redistribute(Nr, nc)
b.redistribute(Nr, nc)

e = prolongate(Nr, er)
# prolongate error
x = x + e
# apply correction

smoother(N, x, b)       # post-smooth

return nc  # possibly modified claim
```

Four multigrid applications;
combat mode on 38 PEs:



Conclusion

- **InvasIC**: hard- + software support for resource-aware parallel programming
- Dynamic RaPping based on local estimations:
  1. check whether additional resources required,
  2. *invade* resources (= obtain a “claim” of PEs),
  3. *infect* them (= load app code + data to new PEs + execute),
  4. *combat* (= OS balances dynamic resource requests),
  5. *retreat* (= free invaded claims)

Conclusion

Karlsruhe Institute of Technology

- **InvasIC**: hard- + software support for resource-aware parallel programming
- Dynamic RaPping based on local estimations:
  1. check whether additional resources required,
  2. *invade* resources (= obtain a “claim” of PEs),
  3. *infect* them (= load app code + data to new PEs + execute),
  4. *combat* (= OS balances dynamic resource requests),
  5. *retreat* (= free invaded claims)
- PP@KIT: language + compiler + applications
- next step: validate resource utilization / energy consumption / fault tolerance in combat setting on demonstrator system

Thank You!