Invasive Malleable Applications

Sebastian Buchwald, Manuel Mohr, Andreas Zwinkau

Karlsruhe Institute of Technology
TCRC 89 "Invasive Computing"
ATPS 2015
No faster CPUs, only more of them

WANTED A FASTER CPU

GOT 4 INSTEAD
More cores means slower cores

Figure 1. Multicore scaling leads to large amounts of dark silicon. Across two process generations, there is a spectrum of trade-offs between frequency and core count; these include increasing core count by 2× but leaving frequency constant (top), and increasing frequency by 2× but leaving core count constant (bottom). Any of these trade-off points will have large amounts of dark silicon.

Battery power is more important these days
Challenge:

Use more cores
more efficiently
Invasive Computing means Tiled Manycore Architectures
Invasive Computing means completely rewritten stack

<table>
<thead>
<tr>
<th>Custom Hardware</th>
<th>iNoC, CiC, i-Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom Operating System</td>
<td>iRTSS, OctoPOS</td>
</tr>
<tr>
<td>Custom Programming Language</td>
<td>invadeX10</td>
</tr>
<tr>
<td>Applications ported/rewritten</td>
<td>HPC, Robotics, ...</td>
</tr>
</tbody>
</table>
The invasive framework gives access to more OS functionality.
Invasive Computing is about invade-infect-retreat
val ilet = (id:IncarnationID)=>{
    Console.OUT.println("Hello World");
}

val constraints = new PEQuantity(4,10)
    && new ScalabilityHint(speedupCurve);

val claim = Claim.invade(constraints);
claim.infect(ilet);
claim.retreat();
Invasive Applications are Malleable

D. G. Feitelson, L. Rudolph; Towards convergence in job schedulers for parallel supercomputers, IPPS 1996

<table>
<thead>
<tr>
<th>User decides (on submission)</th>
<th>rigid</th>
</tr>
</thead>
<tbody>
<tr>
<td>System decides (at runtime)</td>
<td>evolving</td>
</tr>
<tr>
<td></td>
<td>malleable</td>
</tr>
</tbody>
</table>
Example:

Heat Dissipation with Multigrid Approach

Multigrid lets resource idle
Invasive Apps exchange cores
Asynchronously Malleable: „System decides any time at runtime“

Works nicely for Master-Slave Applications
Asynchronously Malleable is more efficient
Integration is async-malleable

\[ f(x) = \sin(x^2) \]
Sorting is async-malleable

Patrick Flick, Peter Sanders, Jochen Speck
Malleable Sorting
IPDPS 2013
val  **ilet**  = (id:IncarnationID)=>{
  for (job in queue) {
    if (queue.checkTermination(id)) break;
    job.do();
  }
}

val resizeHandler = (add:List[PE], remove:List[PE])=>{
  for (pe in add) queue.addWorker(pe,ilet);
  queue.adapt();
  for (pe in remove) queue.signalTermination(pe);
}

val  **constraints**  = new PEQuantity(4,10)
    && new AsyncMalleable(resizeHandler)
    && new ScalabilityHint(speedupCurve);

val claim = Claim.**invade**(constraints);
queue.adaptTo(claim);
claim.**infect**(ilet);
claim.**retreat**();
val ilet = (id:IncarnationID) => {
    for (job in queue) {
        if (queue.checkTermination(id)) break;
        job.do();
    }
}

val resizeHandler = (add:List[PE], remove:List[PE]) => {
    for (pe in add) queue.addWorker(pe, ilet);
    queue.adapt();
    for (pe in remove) queue.signalTermination(pe);
}

val constraints = new PEQuantity(4,10) && new AsyncMalleable(resizeHandler)
    && new ScalabilityHint(speedupCurve);

val claim = Claim.invade(constraints);
queue.adaptTo(claim);
claim.infect(ilet);
claim.retreat();
val ilet = (id:IncarnationID) => {
  for (job in queue) {
    if (queue.checkTermination(id)) break;
    job.do();
  }
}

val resizeHandler = (add:List[PE], remove:List[PE]) => {
  for (pe in add) queue.addWorker(pe, ilet);
  queue.adapt();
  for (pe in remove) queue.signalTermination(pe);
}

val constraints = new PEQuantity(4,10) && new Malleable(resizeHandler) && new ScalabilityHint(speedupCurve);

val claim = Claim.invade(constraints);
queue.adaptTo(claim);
claim.infect(ilet);
claim.retrait();
Invasive Malleable Applications

The Invasive Framework gives access to more OS functionality

```
val ilet = id:IncarnationID cita
  for (job in queue) {
    if (queue.checkTermination(id)) break;
      job.do();
  }

val resizeHandler = (add: List[PE], remove: List[PE]) =>
  for (pe in add) queue.addWorker(pe, ilet);

val constraints = new MaxQuantity(4, 10)

val claim = Claim.invoke(constraints);
```

Asynchronously Malleable is more efficient

More cores means slower cores

Multigrid lets resource idle

Not idle!
Appendix
Case Study: **Multigrid**

M. Schreiber, A. Zwinkau, et al. *Invasive computing in HPC with X10*  
X10 Workshop 2013
val ilet = (id:IncarnationID)=>{
    for (job in queue) {
        if (queue.checkTermination(id)) break;
        job.do();
    }
}

val resizeHandler = (add:List[PE], remove:List[PE])=>{
    for (pe in add) queue.addWorker(pe,ilet);
    queue.adapt();
    for (pe in remove) queue.signalTermination(pe);
}

val constraints = new PEQuantity(4,10)
    && new Malleable(resizeHandler)
    && new ScalabilityHint(speedupCurve);

val claim = Claim.invade(constraints);
queue.adaptTo(claim);
claim.infect(ilet);
claim.retreat();