invadeX10 Documentation

Release 0.5

Andreas Zwinkau, Sebastian Buchwald, Gregor Snelting

August 16, 2013

CONTENTS

The invadeX10 framework contains the primitives and convenience functionality to enable invasive programming.

CHAPTER

ONE

CONTENTS

1.1 Quickintro

The following program fragment shows the three basic constructs invade, infect, and retreat:

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

The static method Claim.invade takes constraints and returns a claim object, which represents the allocated resources, a set of processing elements (PEs). A claim provides an infect method to distribute computations across the PEs. The argument of infect is an i-let object, which contains the code to execute together with initial data. The infect call blocks the program, until all i-let computations finish. Afterwards, the retreat method frees all resources within a claim, such that the claim is empty. Now consider the ilet variable of the example above. It could be declared as follows:

```
val ilet = (id: IncarnationID ) => {
   Console.OUT.println("Hello! ("+id.ordinal+")");
};
```

This code is executed on each PE in the claim. For example, if the claim contained four PEs, the hello message would be printed four times with id numbers 0, 1, 2, and 3.

1.2 Setting up invadeX10

1.2.1 Requirements

Obviously, you need an X10 compiler, since the framework is written in X10.

Currently, you must run the code on x86, since it uses the `rtdsc` instruction for measurements.

Also this is only about the simulation environment.

1.2.2 Get the Code

The code can be retrieved via git

```
git clone gitolite@invasic.informatik.uni-erlangen.de:invadeX10
```

1.2.3 Build

For a first test, you should run

```
make run
```

Additionally, remember the two important environment variables for the X10 runtime:

X10_NPLACES Number of places. As invadeX10 currently cannot handle multiple places, this should have no effect, as all processing elements are mapped to one place.

X10_NTHREADS Number of (unblocked) threads per place. This should be set to the total number of simulated ProcessingElements.

Effectively, the command above should better be (for bash)

```
X10_NTHREADS=81 make run
```

1.3 The Constraint System

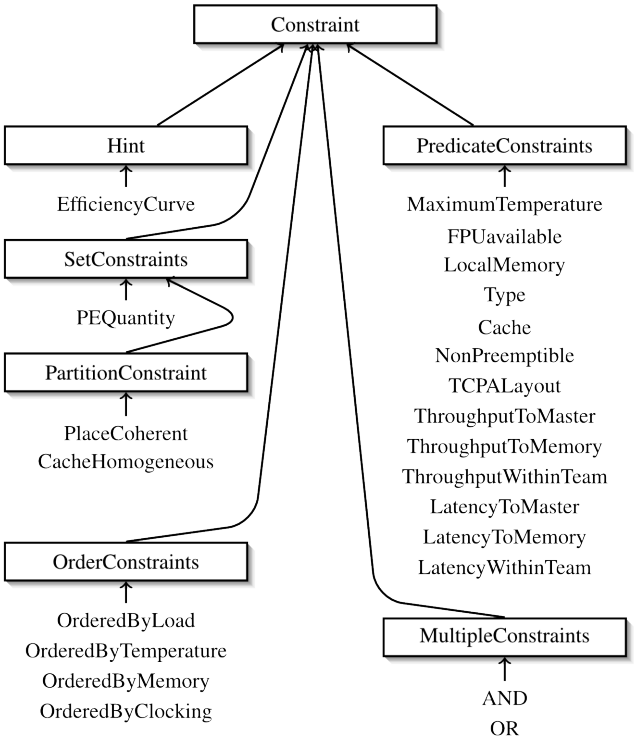

Figure 1.1: The invasive constraint hierarchy.

Constraints for invasion are structured in an extensible hierarchy, which is shown in *the invasive constraint hierarchy*. The most used constraint in practice is PEQuantity, which specifies the desired number of PEs. Predicate constraints are relatively simple, as they place a constraint on the requested PEs, like an FPU being available. The partition constraints are complex in comparison, as they specify requirements for the whole set of the requested resources, like place coherence which essentially means shared memory. The constraint hierarchy includes AND and OR combinators to construct complex constraints. This allows the programmer to provide multiple implementations for different types of processing elements. For example, the programmer might provide special code to exploit the i-Core hardware, so she requests either one i-Core or two normal SPARC cores. The agent system fulfils either of these constraints and the application adapts to the currently available resources.

The first class of constraints we identified were so-called predicate constraints, which specify a predicate for processing elements. An application might require the demanded processing elements to (1) be under a certain load, (2) be under a certain temperature, (3) have an FPU, (4) have certain amount of local memory, (5) have a scratch pad memory, (6) be of a certain type, (7) have a certain cache size, (8) be migratable, or (9) not be scheduled preemptively. Such constraints impose a simple filter operation over the set of available processing

elements within an invade operation. The second class of constraints are order constraints, which provide an ordering of processing elements according to (1) load, (2) temperature, (3) memory, or (4) speed. Using these constraints an application can communicate its preferences, whether it is IO- or CPU-bound.

The third class of constraints are set constraints, as they specify conditions for a set of processing elements as a whole. The most common constraint is the (1) quantity of processing elements to be claimed, but there are also partition constraints, (2) a certain physical layout of the PEs, (3) place coherence, which means that the PEs have shared memory, (4) type homogeneity, in terms of the instruction set architecture, (5) cache type homogeneity. Additionally, there are the two operators AND and OR to combine constraints. At last, the programmer can give nonbinding hints, which can be used to hand complex information like efficiency curves of parameters to the run-time system of the underlying MPSoC architecture. These constraints are implemented as a class hierarchy as shown in Figure above, which is available to the programmer. The constraints above form the leafs of the hierarchy tree and abstract classes, drawn as boxes, partition the tree into categories.

1.3.1 Invading Specific Processing Elements

FPUavailable

Each processing element must efficiently support floating point operations. For example this forbids soft-float emulation. Probably desirable for number crunching.

LocalMemory

Reserves an amount of tile-local memory for each processing element.

`min:Int` Minimal amount in kilobyte.

`max:Int` Maximal amount in kilobyte.

The total amount of reserved memory is the specified amount multiplied by the number of processing elements in the claim. It is *not* multiplied by the number of places or tiles, because places/tiles with more processing elements usually need more local memory.

MaximumLoad

Specify an upper bound to the load for each processing element.

`max:Int` Maximum load in percentage.

MaximumTemperature

Specify a maximum temperature for each processing element at invasion time.

`max:Int` Maximum temperature in degrees Celsius.

Note that temperature will probably increase during the infect.

Migratable

Processing elements might be virtualized and migrated to other places by the operating system for unspecified reasons.

This constraint is not supported in Phase 1 of InvasIC.

NonPreemptible

The operating system must not preempt threads on the processing elements. However, activities may still be executed concurrently. Essentially, this means switches between threads may only happen at defined points (atomic, lock, ...), but not due to an ending timeslice.

This constraint is not supported in Phase 1 of InvasIC.

ScratchPadSize

Specifies an amount of scratch pad memory for each processing element.

`min:Int` Minimal amount in Byte.

`max:Int` Maximal amount in Byte.

TCPALayout

Specifies a certain rectangular layout for all TCPA processing elements.

`minX:Int` Minimal width of TCPA layout.

`minY:Int` Minimal height of TCPA layout.

`maxX:Int` Maximal width of TCPA layout.

`maxY:Int` Maximal height of TCPA layout.

Non-TCPA processing elements are not constricted by this constraint.

ThisPlace

Specifies that processing elements must be of the current place.

Type

All processing elements must have a certain type.

`t:PEType` Type of all processing elements. Types are for example SPARC, iCore, and TCPA.

Since exactly one type is given, this implies type homogenity.

1.3.2 Requirements for the Claim as a Whole

CacheHomogeneous

All processing elements must have the same cache type.

LatencyToMaster

Specifies a lower bound for the latency of each child processing element to the master processing element.

`cycles:Int` Latency in processor cycle.

The master is the processing element issuing the invade and children are the processing elements in the resulting claim.

LatencyToMemory

Specify a lower bound for the latency of each processing element to external memory.

`cycles:Int` Latency in processor cycle.

LatencyWithinTeam

Specify the minimum latency in processor cycles between processing elements in this claim.

`cycles:Int` Latency in processor cycle.

OrderedByClocking

Prefer processing elements with high clocking. Desirable for arithmetic-heavy code to improve performance.

OrderedByLoad

Prefer processing elements with low load. Desirable, if load is a performance bottleneck.

OrderedByMemory

Prefer processing elements with lots of free tile-local memory. Desirable, if tile-local memory can improve performance.

OrderedByTemperature

Prefer processing elements with low temperature at invasion time. Desirable if overheating is expected but should be avoided by switching processing elements intelligently.

PEQuantity

Specifies the minimum and maximum number of processing elements.

`min:Int` Minimal number of required processing elements.

`max:Int` Maximal number of required processing elements.

PlaceCoherent

All processing elements must to be in the same place. Effectively, the processing elements can share data without using `at`.

ThroughputToMaster

Specifies a lower bound for the bandwidth of each child processing element to the master processing element.

`bandwidth:Int` Bandwidth in Bytes/processor cycle.

The master is the processing element issuing the invade and children are the processing elements in the resulting claim.

ThroughputToMemory

Specify a lower bound for the bandwidth of each processing element to external memory.

`bandwidth:Int` Bandwidth in Bytes/processor cycle.

ThroughputWithinTeam

Specify a lower bound for the bandwidth of each processing element to the other processing elements within the resulting claim.

`bandwidth:Int` Bandwidth in Bytes/processor cycle.

In contrast to the other link constraints, this is not a predicate constraint, because this requirement constraints the claim as a whole.

TileSharing

Specifies that this claim can share a tile with others. If the ONLY_WITHIN_APPLICATION variant is chosen, then only claims of the same application are allowed to share the tile. Note that both participants must allow sharing for it to happen.

1.3.3 Hint Constraints

Some constraints are used to give hints to runtime system, but they give no guarantees to the claim holder.

AppClass

Specifies the application class for improved scheduling. Currently available classes:

- Compute intensive
- Bandwidth limited
- Low latency
- Low power
- Safety critical
- High affinity
- General purpose

PotentiallyMorePEs

Specifies that the scheduler can provide more parallelism for executing activities than previously invaded into the corresponding claim.

PotentiallyLessPEs

Specifies that the scheduler can provide less parallelism for executing activities than previously invaded into the corresponding claim.

ScalabilityHint

Specifies the scalability of the algorithm, which should be processed by the claims resources. It takes either a list of descrete points or a polynomial of degree three.

points:Array[int] Starting with zero PEs, how does the algorithm scale. 100 is the baseline of 100% or zero speedup.

alpha:int, beta:int, gamma:int represents the polynomial f(x) = alpha + beta*x + gamma*x^2. However, to provide an integer API the arguments are scaled down by a million, so for example alpha = `alpha`/1000000.

1.3.4 Composing Constraints

AND

A constraint class to combine constraints, such that each constraint must be fulfilled.

OR

A constraint class to specify alternative wishes (first-fit).

Operators

You can also use the `&&` and `||` operators to combine constraints.

1.4 Invasion

Invasion is the allocation of resources, which most prominently means processing elements (CPU cores). As seen in *Quickintro*, the invasion usually looks like this:

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

The static invade method of the *invasic.Claim* class gets a Constraint object as argument and returns a claim object. The constraints can be composed as described *above*. Internally, this will spawn a new agent and communicate with other agents to collect the claim resources. This means that this is a potentially very expensive operation in terms of run time.

Since it is common to compose a few constraints with AND, there is a convenience method called `invadeAND`:

```
val claim = Claim.invadeAND([
   new PEQuantity(1,8),
   new Type(PEType.RISC)
]);
```

This invasion returns a claim containing between one and eight RISC PEs.

If the agent is unable to get the minimum amount of resources to fulfill the constraints, a `NotEnoughResources` exception is thrown.

```
try {
   val claim = Claim.invade( new PEQuantity(1000000) );
} catch (e:NotEnoughResources) {
   Console.ERR.println("Could not get 1000000 PEs");
}
```

1.4.1 Introspection

A claim's contents can be inspected, for applications that can adapt the following infection. For example check the number of processing elements:

```
val count = claim.size();
```

A list of places, where the processing elements are can be obtained:

```
val places = claim.places();
```

Also the list of processing elements is directly available:

```
val pes = claim.processingElements();
```

Using this list, the attributes of the processing elements can be inspected:

```
val pe = pes.get(0);
val plce = pe.getPlace();
```

1.5 Infection

1.5.1 Simple

The most simple way for infect was in *Quickintro*:

```
val ilet = (id:IncarnationID) => { ... };
claim.infect( ilet );
```

First, an iLet is a function, which takes an incarnation id. The function might be a closure[1], which essentially means a function and data included. This function is passed/copied to the `infect` method of a claim object, which executes it once on every processing element contained.

The `infect` call always leads to one activity per processing element. However, the programmer is free to create additional activities using `async`.

1.5.2 Map-Reduce

Collecting data computed by ilets is a common task, so invadeX10 provides convenience variants of infect, which work like map-reduce:

```
val ilet = (id:IncarnationID) => { return x; };
val ret = claim.infect( ilet );
```

Here infect returns an array containing all values returned by ilets. This means the size of `ret` equals the size of `claim.processingElements()`, since each PE executes exactly one ilet. For example, if the iLet returns an `Array[int]`, then infect returns an `Array[Array[int]]`.

It is also possible to provide a reduce function:

```
val ilet = (id:IncarnationID) => { return 2; };
val reduce = (a:int, b:int) => { return a + b; };
val ret = claim.infect( ilet, reduce );
```

In this example, infect returns an `int`, which is the sum of all returned `2` values.

[1] http://beza1e1.tuxen.de/articles/closures.html

1.5.3 Manually

The programmer is free to implement infect by herself. The most naive (and inefficent) variant with similar behavior looks like this:

```
val ilet = (id:IncarnationID) => { ... };
finish {
   for (pe in claim.processingElements()) {
      val id = IncarnationID(...);
      async at (pe.place) ilet(id);
   }
}
```

The manual approach is for example appropriate, if some data must be copied to each place (not each processing element):

```
val data = ...
finish {
   for (p in claim.places()) {
      val pes = ... // PEs in claim in place p
      async at (p) {
         store(data);
         for (pe in pes) async {
            process(load_data());
         }
      }
   }
}
```

This approach also enables to provide different data per place or processing element:

```
val data = ...
finish {
   for (p in claim.places()) {
      val pes = ... // PEs in claim in place p
      val pdata = data(p);
      async at (p) {
         for (pe in pes) {
            val pedata = pdata(pe)
            async process(pedata);
         }
      }
   }
}
```

1.5.4 Distributing Data

If you have to distribute data over multiple places, most people seem to think of `DistArray` first. However, often other approaches are more appropriate. We recommend to try in this order:

1. Static fields

Static fields exist once in each place. While they are always immutable, they can contain an immutable reference to a mutable value (`Cell`, `Array`, ...).

If your algorithm for example has a varying amount of data per place, a `DistArray` does not help, while a static field of type Array can be resized.

2. GlobalRef

A `GlobalRef` becomes necessary once you cannot name or statically enumerate your variables.

3. DistArray

Finally, a `DistArray` is convenient if you work with large matrices. Note that this cannot be resized or redistributed [2] .

1.6 Retreat

1.6.1 Simple Retreat

The most basic case of retreat is free all resources of a claim. This is nearly always done, when the job is done.

```
claim.retreat();
```

The claim object is invalid afterwards, so the programmer must not call infect for example.

Partial Retreat

This is equivalent to reinvade, so look at *Resource-aware Programming*.

1.7 Resource-aware Programming

1.7.1 Adapting to Internal Requirement Changes

While it is possible to retreat claims and invade anew. You should use the `reinvade` method for adapting claims.

Update Claim

Without an argument, reinvade gives the application's agent the chance to optimize the claim.

```
val claim = Claim.invade(constraints);
claim.infect(foo);
val changed = claim.reinvade(); // optimize resources
claim.infect(bar);
```

The claim's resources might have changed, so for example a list of processing elements is not up to date anymore. The `reinvade` method returns `false` if nothing has changed and `true` otherwise.

Change Claim Constraints

Alternatively, reinvasion can also be used to change the constraints.

```
val claim = Claim.invade(constraints);
claim.infect(foo);
val changed = claim.reinvade(other_constraints);
claim.infect(bar);
```

[2] Although there are ideas to extend DistArray for the invasive architecture

This behaves similiar like the call without arguments, but the claim now is differently constrainted.

Note that both methods are relatively cheap to call. They will not trigger inter-place communication for example. However, this means that changing constraints can fail, if the currently hold resources are not enough to fulfill the changed constraints. In this case a `NotEnoughResources` exception is thrown.

Partial Retreat

There is also the `partialRetreat` method, which is likely to disappear in future versions.

1.7.2 Adapting to External Resource Changes

Currently, not actually possible. However, LCP001[3] tries to remedy this. There is only an implicit reaction to external changes due to optimizations by the agent on `reinvade` calls.

1.8 Changing invadeX10

1.8.1 Language Change Proposals

The Language Change Proposal (LCP) process[4] has been introduced to support and document the ongoing changes in the invasive language. This manual should not duplicate the information, so only summaries are integrated here and for details you can look up specific LCPs.

LCPs, which are active (complete and implemented) can also be found in the `docs` directory.

1.8.2 Hacking the Code

If you change code in invadeX10, you should adhere to the guidelines in the `HACKING` file:

```
Coding Guidelines
-----------------

 * Use Tabs not Spaces for indentation
 * Avoid floating point numbers in the API
 * "make run" must succeed
 * Code Comments
   * Do not commit dead or commented-out code
   * // comments for TODO, FIXME, etc. which should disappear
   * /**/ comments for documentation, which should stay
 * FIXME annotations are used in experimental branches, but not in master or X10-something
 * The master branch should only contain:
   * single commit features
   * single commit fixes
   * merges without conflicts
 * branches must not live across versions (vX.X tags)
 * Merges of active LCPs must include a copy of the wiki page in docs/

Contributions
-------------

If you have patches for invadeX10, you can send them to Andreas Zwinkau <zwinkau@kit.edu>.
However, even better, setup a git repo, where he can pull from and track your further changes.
```

[3] https://invasic.informatik.uni-erlangen.de/intern/dokuwiki/subprojects/a1/lcp001
[4] https://invasic.informatik.uni-erlangen.de/intern/dokuwiki/subprojects/a1/lcp

Adding a New Application

If you want to create a new application which uses this framework, look at the example in `src/applications` first. Then you start by duplicating on of the examples.

Add your application to `src/testbenches/TestBench.x10`, then it is build and run on `make run`.

The Simulator

The usual way to develop with invadeX10 is to use the simulator, so you do not need an FPGA to synthesize the invasive architecture. The simulator (in `src/simulator`) is implemented in (nearly) pure X10. However, this is only a functional simulator[5], so you should not consider the timing to be in any correlation to a run on a real invasive architecture. It emulates

- the invasive architecture (multi-tile, heterogeneous, cache-incoherent global memory,...)
- the iRTSS (OctoPOS and agent system)
- the invasive X10 runtime system

[5] http://en.wikipedia.org/wiki/High_level_emulation

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

INDEX