Resource-Aware Programming and Simulation of MPSoC Architectures through Extension of X10

Frank Hannig[‡], Sascha Roloff[‡], Gregor Snelting[§], Jürgen Teich[‡], Andreas Zwinkau[§]

[‡]Hardware/Software Co-Design, Department of Computer Science,
University of Erlangen-Nuremberg, Germany

[§]Programming Paradigms Group,
Karlsruhe Institute of Technology — KIT, Germany

ABSTRACT

The efficient use of future MPSoCs with 1000 or more processor cores requires new means of *resource-aware programming* to deal with increasing imperfections such as process variation, fault rates, aging effects, and power as well as thermal problems. In this paper, we apply a new approach called *invasive computing* that enables an application programmer to spread computations to processors deliberately and on purpose at certain points of the program. Such decisions can be made depending on the degree of application parallelism and the state of the underlying resources such as utilization, load, and temperature. The introduced programming constructs for resource-aware programming are embedded into the parallel computing language X10 as developed by IBM using a library-based approach. Moreover, we show how individual heterogeneous MPSoC architectures may be modeled for subsequent functional simulation by defining compute resources such as processors themselves by lightweight threads that are executed in parallel together with the application threads by the X10 run-time system. Thus, the state changes of each hardware resource may be simulated including temperature, aging, and other useful monitor functionality to provide a first high-level programming test-bed for invasive computing.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*; D.3 [**Software**]: Programming Languages; C.1.4 [**Computer Systems Organization**]: Processor Architectures—*Parallel Architectures*

General Terms

X10, MPSoC, Simulation, Resource-aware programming

1. INTRODUCTION AND MOTIVATION

With the ever increasing number of cores that may be integrated on a single chip, difficulties arise when programming SoC devices in a resource-efficient manner. We see invasive computing [14] as a solution to this problem: With this term, we envision that applications running on a Multi-Processor System-on-a-Chip architectures (MPSoC) might map and distribute their workload themselves based on their temporal computing demands, temporal availability of resources, and other state information of the resources (e. g., temperature, faultiness, resource usage, permissions). However, in order to make this computing paradigm become a reality and to evaluate its benefits properly, the way of application development including algorithm design, language implementation and compilation tools needs to change to a large extent. The idea of allowing applications to spread their computations on resources and later free them again decentrally by themselves at run-time sounds promising. The expected benefits include an increase of speedup (with respect to statically mapped applications), fault-tolerance, and a considerable increase of resource utilization, hence computational efficiency. These efficiency numbers, however, need to be analyzed carefully and traded against the overhead caused with respect to statically mapped applications.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SCOPES'11, June 27–28, 2011, St. Goar, Germany.

Copyright © 2011 ACM 978-1-4503-0763-5/11/06...$10.00

1.1 Invasive Computing

First and most fundamentally, in [13] and [14], Teich and others introduced a novel paradigm for resource-aware computing called *invasive computing*[1] that integrates research on algorithms as well as architectures. The main idea behind *invasion* is to add to each application the ability to explore and claim resources in a certain neighborhood and to copy its configuration code, program, and possibly data to such places in a phase of invasion, and then to execute the given problem in parallel based on the available (invasible) region of processing resources. Through invasion, an application will thus be able to spread its computations for parallel execution based on the availability and the actual state of processing resources. For execution phases of reduced degree of available application parallelism, the application may itself perform a *retreat* to free occupied resources so to optimally exploit all resources and make them available for other applications.

The chart depicted in Figure 1 shows the typical state transitions that occur during the execution of an invasive program. In the beginning, an initial *claim* has to be constructed. By *claim* we denote a set of cores that the application can use for its parallel execution. Claim construction is done by issuing a call to *invade*. After that, *infect* is used to start the actual

[1]Invasive computing is not directly related to *invasive software composition* [1] that deals with software engineering.

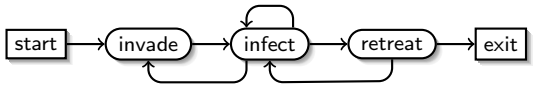

Figure 1: State chart of an invasive program.

application code on the previously allocated *claim*. The actual application code that is spread onto infected resources for subsequent parallel execution is called *i-let* (standing for *invaslet*[2]). Once the execution on all cores finishes, the number of cores inside the *claim* can be altered by calling *invade* or *retreat* to either expand or shrink the application's *claim*. In case of *retreat*, the processing elements are cleaned up from the *i-let* entities that have been setup by *infect*. Alternatively, if the degree of parallelism does not change, it is also feasible to dispatch a different program onto the same set of cores by issuing another call to *infect*. If a call to *retreat* leaves the *claim* empty, there are no computing resources left for further execution of the program, hence it terminates its execution and exits. Notably, a claim may not only contain processing resources, but also memory as well as communication resources.

We do believe that invasive computing might solve many future problems of massively parallel application processing on future MPSoC platforms by providing and porting principles of *self-organization* into reconfigurable architectures, integrating 1000 and more processor cores on a single chip. The major advantages of invasive computing are that it will provide resource-awareness, a gain in computational efficiency and performance, application-level error resiliency, self-adaptive power control and management, and self-optimization of resource utilization. Another objective is to increase the lifetime or to encompass aging effects of future sub-micron technology by avoiding stressing the hardware too much. The major advantages invasive computing will be offering can be summarized as follows:

- Gain in computational efficiency and performance
- Application-level fault-detection and fault-tolerance
- Self-adaptive power control and management
- Self-optimization of resource utilization

1.2 Contributions

Based on the introduced paradigm of invasive computing, we will introduce programming constructs for resource-aware programming and show how these abstract constructs may be embedded into the existing concrete parallel computing language X10 [4] as developed by IBM using a library-based approach. Subsequently, we show how individual heterogeneous MPSoC architectures may be modeled for subsequent functional simulation by modeling compute resources such as processors themselves by light-weight parallel threads that are executed together with the application threads by the X10 run-time system. Thus, the state changes of each hardware resource may be simulated including temperature, aging, and other useful monitor functionality so to provide a first high-level programming test-bed for invasive computing. As a case study, we present fragments of embedded image computing algorithms such as parts of an invasive JPEG decoder and a resource-aware parallel histogram computation algorithm. We conclude with a summary and outline of future work.

[2]This conception goes back to the notion of a *servlet*, which is a (Java) application program snippet targeted for execution within a web server.

2. EXISTING WORK

There are only few ongoing research projects that have similarities with our ideas of invasive computing. In the CAPSULE project [10], for example, the authors describe a component-based programming paradigm combined with hardware support for processors with simultaneous multi-threading (SMT) in order to handle the parallelism in irregular programs. Here, an application is dynamically parallelized at run-time. A pure software version of CAPSULE, demonstrated on an Intel Core 2 Duo processor is presented in [3]. In the TRIPS project [11], an array of small processors is used for the flexible allocation of resources dynamically to different types of concurrency, ranging from running a single thread on a logical processor composed of many distributed cores to running many threads on separate physical cores. In the MORPHEUS project [16], heterogeneous dynamically reconfigurable SoC architectures with various types of reconfiguration granularity were developed. However, the above approaches do not touch the major problems of algorithmic design and the explicit distribution of workload across a given architecture.

One particular question addressed in the following is how the fundamental language constructs of invasive computing and resource-aware programming, as introduced in subsection 1.1, can be embedded into existing programming languages and can be applied to the modeling and functional simulation of heterogeneous MPSoCs. An instance of such a heterogeneous MPSoC is given by the tiled architecture as shown in Figure 2, including a combination of CPU tiles, application-specific tightly-coupled arrays (TCPAs), and interconnected via a network-on-chip (NoC).

Most popular languages and language extensions such as OpenMP [9] only support shared memory architectures, or in case of MPI [15], are much too low-level. A parallel programming model that fits very well to our envisioned heterogeneous tiled architectures is the concept of *partitioned global address space* (PGAS). Here, a global memory address space is logically partitioned to the given set of processors and tiles, respectively. Thus, locality within a tile can be exploited. A couple of programming language has been developed that put the PGAS model into practice: UPC [7] uses a Single Program Multiple Data (SPMD) model of computation in which the amount of parallelism is fixed at program start-up time, hence UPC is unusable for invasive computing. Fortress [12] is based on functional programming and features implicit parallelism, which does not support the invasive computing concept of explicit resource- and load-awareness. Chapel [5] and X10 [2, 4] on the other hand offer already basic language constructs, which enable a programmer to explicitly assign tasks to resources. Chapel was mainly designed with High Performance Computing (HPC) in mind, thus we selected X10 as the base for an implementation of our language for resource-aware programming.

X10 offers the fundamental concepts for the support of distributed, heterogeneous processor and memory architectures. In X10, the programmer can create new processes in different address spaces called *places*. A place can be simply one CPU, or a domain of processors that consists of several nodes with shared memory communication. A contemporary chip multi-processor, such as the Intel Xeon six-core architecture, would be considered one place in X10, whereas a heterogeneous hardware architecture would consist of several X10 places. In Figure 2, for example, only within one *compute*

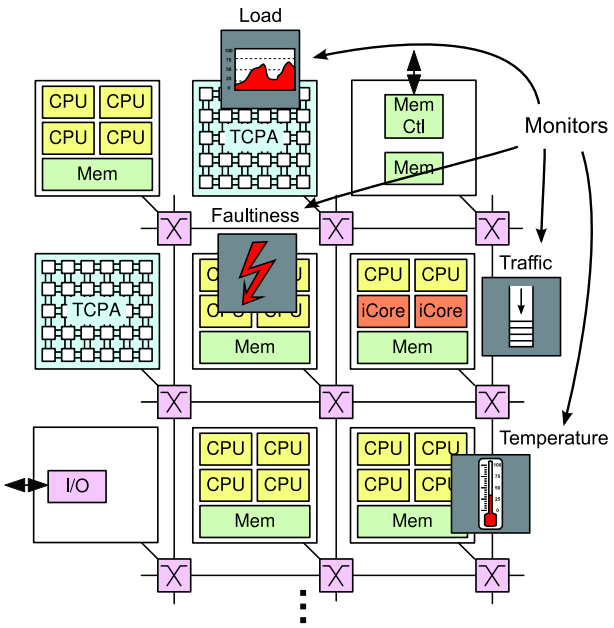

Figure 2: Generic invasive tiled architecture including several so-called loosely-coupled processor tiles (different types of RISC CPUs), tiles of tightly-coupled processor arrays (TCPAs) as well as memory and I/O tiles. Each resource may be associated one or multiple monitors for capturing, e. g., temperature, faultiness, load, or traffic.

tile, a shared memory model is considered. That is, one tile corresponds to one X10 place. X10 provides a compiler that creates not only an executable program but also a run-time system library to manage the creation of processes and the data distribution between different places. In the following sections, we briefly introduce how the concepts of invasive computing may be integrated into X10 as well as how the existing X10 run-time system can be employed in order to model and simulate the dynamic behavior of an application running on a tiled MPSoC architecture.

3. INVASIVE X10

In the following, we introduce a library-based approach to invasive programming in X10. While the run-time system restricts some possibilities, e. g., to manage worker thread pools, this approach allows to explore invasive computing and resource-aware programming in general through functional simulation.

3.1 Basic Functionality

Consider the following program fragment, which 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. While such a claim can still be infected, this would do nothing. Now consider the `ilet` variable of the example above. It could be declared as follows:

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

Here, `ilet` is assigned a function object (declared as an anonymous X10 function), which takes one argument `id`, returns nothing, and prints a greeting to standard output. We call such a function declaration an *i-let candidate*, as it can be used as an i-let, like in the example above. In contrast to C/C++, the X10 language does not provide function pointers but closure objects, which means its free variables (`Console.OUT` in our example) are bound to their values from the lexical environment. This binding is done at the assignment, which means a function object is instantiated that includes a reference to the `Console.OUT` object. Such a closure object is called *i-let instance*. During the infection, the i-let instance is copied and distributed to every processing element within the claim. By enumeration the `infect` method generates incarnation ids and calls the i-let instance function with these ids as the argument. At this point the i-let instance becomes an *i-let incarnation* until the call returns. The set of all i-let incarnations, which were generated due to a specific `infect` invocation, is called a *team*. Naturally, a team consists of as many i-let incarnations as there are processing elements in the corresponding claim.

For resource-aware programming, the application shall adapt dynamically to the available resources. Hence, the programmer specifies the resource needs by defining the `constraints` variable in our example and the system as a whole decides on the actual allocation. In our example, we require between one and eight processing elements, all in the same place, and each with a current load of at most 70 percent.

```
val constraints = new AND();
constraints.add(new PEQuantity(1,8));
constraints.add(new PlaceCoherence());
constraints.add(new MaximumLoad(0.7f));
```

All the available constraints are available as classes to the programmer. For example, the `AND` object, which is assigned to `constraints`, is a container and specifies that all constraints within must be fulfilled. In the code above, the three constraints are added to this container, so it can be used for a call to `Claim.invade`.

3.2 Invasive Command Space

We collected a batch of more than 25 pseudo-code examples of invasive programming from different areas of parallel and embedded computing, e. g., image processing, matrix computation, and signal filtering, to design the command space for the constraint system. Based on this data, the following constraints were chosen as powerful enough for all needs.

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)

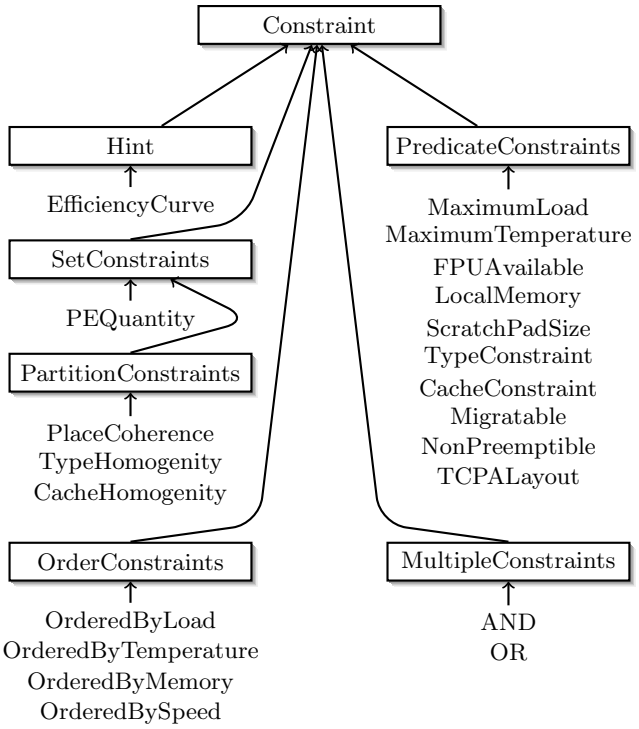

Figure 3: Constraint hierarchy.

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. By giving multiple of these constraints, the programmer can impose a secondary or tertiary ordering.

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 non-binding *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 3, 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.

Observing the constraints, one can see that only set constraints can fail, as other constraints could simply return an empty claim. Failing means that a `NotEnoughResources` exception is thrown. Since the application should handle the cases with few resources anyways, it can handle empty claims usually within the same branch. In contrast, providing a special exception for empty claims could lead to code duplication.

The combination of constraints is not trivial. For example, consider the constraint of a specific 3×3 PE layout and place coherence. Searching for both at once, it may be possible to find a place, where enough PEs in the right layout are available. If the wrong place is selected first, then a layout may not fit afterwards. If the layout is found, then a place constraint may fail or reduce the PE set afterwards. So the most complex constraint in terms of implementation logic is AND, because for an efficient selection of PEs, it must use a certain order for the constraint matching. Given a set of available processing elements, the following steps are performed:

1. Filter the set according to all predicate constraints in arbitrary order.
2. Order the PEs according to the order constraints in their specified priority.
3. Partition the PE set according to all partition constraints.
4. Select a partition that matches the quantity constraint.

3.3 Integration Aspects

Invasive computing introduces new ways to express parallelism, so we must investigate the relation to existing constructs in X10, which provides `async` and `at`. The basic concepts of X10 are the *activity*, which is a light-weight task, and the *place*, which represents shared memory.

`async <statement>` creates a new activity in the current place, which executes `<statement>`, while the current activity immediately proceeds with the next statement.

`at (<p>)<expression>` migrates the current activity to another place `<p>`, where `<expression>` is evaluated and its return value is returned by the `at` at the original place. Under the hood, the X10 compiler generates a closure object out of `<expression>` and `at` has copy semantics concerning the free variables within.

In summary, `async` is used for concurrent execution and `at` for messaging in a distributed memory system. An X10 activity is not a thread in the POSIX sense, as they are managed within the application and the run-time system provides worker threads for processing activities. This means, activity creation is very cheap compared to thread creation.

Our invasive programming language provides no primitives for messaging, apart from an initial data transfer, but an invasion also creates additional activities. However, the concept of a processing element in invasive computing corresponds to a worker thread in X10, so `async` and `infect` manage parallelism on different levels. The current run-time system hides worker threads, so actual invasive computing requires changes to the run-time system, which is outside the scope of the current implementation used for functional simulation.

3.4 Reinvade and Partial Retreat

Extended concepts of invasive computing are reinvasion, reinfection and partial retreat. Reinfection is simply to infect a previously infected claim again. The requirement from the programmers point of view is that data is not deleted when a team finishes and leaves the infected processing elements.

However, this is guaranteed, as the system is only allowed to free resources in the retreat.

Reinvasion is necessary, if an application needs additional resources at some point. While it is usually better for the system as a whole to retreat and then invade again, another process might capture resources in between. Our framework provides a safe alternative: invade a "delta" claim and then merge the two claims. Here is an example, how an original `claim` can be extended.

```
val delta = Claim.invade(constraints);
val claim2 = claim + delta;
claim2.infect(ilet2);
```

The `+` operator is overloaded to perform the union of the two PE sets. The infection afterwards, can be understood as a simultaneous infection of the original and the delta claim, where the infection id are generated with respect to the union set. The original `claim` object is available in addition to `claim2`, so it is still possible to infect just the original set of PEs.

A partial retreat is used, when an application does not need all allocated resources anymore, but must still keep a part of its claim. For this case, a claim provides a `partialRetreat` method, which takes a constraint (like invade) and retreats from resources matching this constraint. The following example retreats from the set of actually claimed resources to exactly four processing elements:

```
claim.partialRetreat(new PEQuantity(4));
```

Like the invade call, this results in an `NotEnoughResources` exception, if `claim` contains less than four PEs.

3.5 Color Space Transformation Example

The following code is part of an invasive JPEG-decoding case study. The shown algorithm computes an Irreversible Color Transform (ICT), which converts an image from the RGB to the YC_bC_r color space. In order to speed up this transformation, the application tries to invade a TCPA accelerator of size 10×10 PEs.

```
val img = Image.load(filename);
val c1 = new TCPALayout(10,10);
val c2 = new TypeConstraint(PEType.TCPA);
val c3 = new PEQuantity(1);
try {
  val claim = Claim.invadeAND([c1,c2,c3]);
  // invasion succeeded
  claim.infect((i:int)=>{
    ComponentTransform.forwardIctTCPA(i,
        img);
  });
} catch (e:NotEnoughResources) {
  // invasion failed, do it locally
  ComponentTransform.forwardIctCPU(img);
}
```

The three constraints specified above require for a claim, to be suitable, a single TCPA of size 10×10 PEs. The actual transformation code is contained in the `forwardIctTCPA` method. If the invasion succeeds, the infect uses the allocated TCPA resource. If it fails, because not enough resources are available in the system, the `catch` block uses the `forwardIctCPU` method as a fallback solution and computes the code locally and sequentially.

4. FUNCTIONAL SIMULATION

In order to enable the validation of the features of invasive computing as well as architectural variants when designing novel MPSoC architectures at an early project state, novel simulation methodologies are investigated.

In general, amongst others, one can distinguish between architectural and functional simulation. In this paper, we focus on functional simulation, which is a key feature in order to fundamentally understand the characteristics and benefits of invasive computing. The goal of the functional simulation is to enable early validation of invasive programming concepts and to allow the investigation of a broad range of different invasive hardware platforms on a functional layer. Providing such simulation facilities is important for assisting the optimization process of the underlying concepts, especially in a project state where full hardware or software implementations are not yet available.

In our scope, functional simulation of resource-aware parallel programs shall mainly provide a support for a) application programmers to write invasive application codes, but also aid b) architectural designers to evaluate resource-aware concepts when prototyping and dimensioning their future MPSoC architecture. Hence, at this level, cycle-accurate architectural simulation is not only much too slow, but also not the desired level of abstraction. Possible use cases of the following simulation concepts are thus to explore different invasion strategies, for instance, based on agents [6, 8] or to investigate the behavior of invasive applications in competitive scenarios, or to simulate resource-aware behavior such as if a processing element or other resources may become faulty, overheated, or overloaded. In this section, we present the design, the implementation, and the usage of our first realization of such a functional simulation environment for invasive applications and architectures.

4.1 Functional Simulator: Concept and Implementation

Our concept and implementation of a functional simulator uses also the introduced parallel object-oriented programming language X10. It offers many convenient features for parallel computing as explained in section 2.

Apart from the mentioned programming library for invasive programs, resource-awareness is accomplished by modeling and emulating the state of each hardware resource explicitly through a concept called *hardware threads*, see subsection 4.2 for details. Basically, each hardware resource of a modeled MPSoC architecture itself is represented by a light-weight thread so to emulate its important state information such as current temperature, load, or traffic.

In order to create a user-defined MPSoC architecture simulation model, we provide a special X10 class in which according to subsection 4.3, the customized architecture is modeled by defining the type, properties, and monitor functions of the desired resources and then creating hardware threads for each modeled resource.

Putting it all together, the following simulation framework for resource-aware programming is fully X10-based and its benefits are threefold: It is available to as well a) application-programmers for experimenting with invasive algorithmic and program behaviors, b) operating system and firmware designers for testing and comparing decisions on load balanc-

ing and resource management, as well c) MPSoC designers for exploring architectural alternatives and monitor functions and thus for evaluation of architectural design options such as number and types of tiles, tile processors, etc.

4.2 Hardware Threads

The simulation of the interplay between invasive program behavior on the one hand and the resulting state of the underlying processing resources such as their temperature, load, or faultiness in dependence of its state of invasion on the other hand, is one of the key features of our functional simulator.

In order to implement such a functionality, we introduce a concept called *hardware threads* for modeling monitorable resource states. In particular, for each resource in a user-defined architecture, a light-weight thread is created, which emulates its state through one or more so-called *monitor functions* that explicitly change its dynamic state of parameters such as temperature, load, and aging. We will present a case study of a very simplistic temperature monitor later.

In terms of X10, each hardware thread itself is realized by an X10 activity similar to each application i-let. In the resulting X10-based simulation, each hardware thread runs in parallel with the application. The i-lets and the hardware threads are basically indistinguishable in the resulting X10 simulation run. Once, the applications are started, each call to the invasive programming library is directed to the simulator in which invasion strategies how to obtain and compose claims to satisfy the constraints and the assignment of i-lets to resources in dependence of the state of each available resource are implemented. Hence, the simulator class emulates currently the run-time system of the invasive MPSoC.

The nice thing about this high-level functional simulation approach is that the X10 programming environment needs not to be changed and that the interplay between invasive program behavior and MPSoC customization may be explored together as desired.

4.3 MPSoC Architecture Modeling

For the functional simulation of the interplay of an invasive application and a customized invasive MPSoC, it is necessary to describe its structure and configuration in terms of topology, number of tiles, number and types of resources within each tile as well as monitors available so to enable resource-aware programming.

For modeling heterogeneous architectures, we assume and restrict ourselves to tiled architectures according to Figure 2, that is, networks of tiles. In general, each tile might be composed again by networks of tiles. However, for simplicity and according to our current implementation, we restrict ourselves first to non-hierarchical tiles each of which may contain several *processing elements* of potentially different types and properties, see also Figure 2 as an example. Now, in order to simulate the dynamic behavior of invasion and resource-awareness for a specific architecture, the dynamic state of the resources needs to be simulated, too. For this purpose, the initialization and customization of an individual MPSoC and its properties is done in the initialization of a special simulator class created for this purpose.

The following X10 code listing gives an example of how to construct a simulator that is able to simulate invasive programs and resource-awareness for an MPSoC composed of just a single tile that consists of four equal processing elements, see, e. g., the lower right tile in Figure 2.

```
// create a new architecture
val arch = new Architecture();
// create a new tile within this
// architecture
val tile = arch.createTile();

// create four PEs within the tile
for (i=0; i<4; i++){
  val pe = tile.createPE();
  // specify the properties of the PE
  pe.peType = PEType.RISC;
  pe.cacheType = CacheType.
      FourWayAssociative;
  pe.localMem = 2048; // KiB
  pe.scratchPadMem = 128; // KiB
  pe.clockFrequency = 1500; // MHz
  pe.isMigratable = false;
  pe.isPreemptible = false;
  pe.hasFPU = true;
}
```

As shown in the above listing, one can specify a number of static properties for each processing element, which can later be used in predicate constraints within each invade command. Furthermore, in order to be able to simulate the dynamic state of each processing element such as its load and temperature, each resource may be assigned one or more individual user-defined monitor-functions that is accessible via constraints. In the simulation framework, different monitor-functions may be assigned to each hardware thread that corresponds to a processing element in a one-to-one manner.

In the following code listing, a very simple monitor function simulating the temperature evolution of a processing element is shown. In case the PE is infected, its temperature is increased every 200 ms until a threshold is reached. Elsewise, if the PE is in idle mode, the temperature is decreased until an idle temperature is reached. In order to simulate the increase of temperature, the method must be called periodically during simulation. This is achieved by periodic activation of each monitor function.

```
val temp = temp_idle;
while(pe.isActive) {
  if(pe.isInfected) {
    if(temp < temp_threshold)
      temp = temp + delta_temp;
  } else {
    if(temp > temp_idle)
      temp = temp - delta_temp;
  }
  System.sleep(200); // ms
}
```

Finally, in our simulation framework each modeled tile is mapped to an X10 place. Using the above construction, one can describe very customized architectures and define customized architecture properties and resource monitor functions very easily.

In our current implementation, only non-hierarchical tiles and PEs can be specified within an architecture, but no

user-defined communication topologies as well as hierarchically nested architectures. These foreseen capabilities will be added to the simulator implementation at a later stage.

4.4 Case Study: Temperature-Aware Load Balancing

The following code fragment is part of the implementation of an invasive parallel histogram calculation algorithm for image frames. The algorithm obtains as input an array of integers describing pixel intensities and basically counts the number of occurrences of each integer number and saving this value in an output (histogram) array. Obviously, this algorithm can be easily parallelized by just tiling the input array into equal stripes and by mapping these individual stripes to different processing elements which compute the histogram on their parts of the array so to generate a partial result. At the end, these partial results have to be only summed up so to obtain the desired histogram.

In the following invasive code fragment, the above algorithm is implemented to perform a batch processing of histogram calculations for a sequence of frames until a user-defined termination criterion is reached. The workload caused by an image frame of size 1024×1024 is represented here by an array of equal size and initialized (just for simplicity) by random integer values from 0 to 255. The histogram calculation is then distributed over two out of four processing elements of type RISC, which shall be located in the same tile according to Figure 2 so to share the same local pixel memory. The workload is partitioned into two parts and distributed to the processing elements using *distributed arrays* of X10 (code not shown). Inside each histogram i-let, the distributed array access is restricted to those values which are mapped to the proper processing element. In order to collect the created partial results from the processing elements, *remote arrays* are used. Before execution, a remote array of the size of the global result array is created on the home place on each processing element. Each i-let then calculates its partial result in its remote array. After execution, the values from the remote arrays are collected in a serial manner and added finally to the global result array.

```
// start the simulator initializing all
// hardware threads and their monitors
Simulator.init();

// create a (random) workload
val rnd = new Random();
val workload = new Array[Int]((0..1023)
    *(0..1023)), (Point) => rnd.nextInt
    (255));

// specify the constraints
val c = new AND();
c.add(new TypeConstraint(PEType.RISC));
c.add(new PEQuantity(2));
c.add(new MaximumTemperature(80));
c.add(new PlaceCoherence());

// main loop processing histogram batch
// requests until user termination
while (!terminated) {
  // invade
  val claim = Claim.invade(c);

  // distribute the workload to the PEs
  ...

  // prepare a result array on each PE
  ...

  // code for the histogram (i-let)
  val code = (id:IncarnationID) => {
    for([i,j] in workloadPart)
      partialResult(workloadPart(i,j))++;
  };

  // infect
  claim.infect(code);
  // collect results from the PEs
  ...

  // retreat
  claim.retreat();
}

// shutdown the simulator
Simulator.exit();
```

Please note that the above invasive program for histogram computation is resource-aware in the sense that claims are built by the run-time system so as to guarantee all specified constraints. In the above example, a claim shall contain exactly two RISC-type processors the temperature of which must not exceed 80° Celsius at invasion time.

Our implementation of the simulator class allows also to record the execution profile of an invasive X10 program: In Figure 4, an automatically generated Gantt chart is shown for the invasive histogram application. One can observe that the second histogram calculation run is performed on different processing elements than the first iteration, because the temperature of previously assigned processors has obviously exceeded the specified limit of 80° Celsius. Hence, migrating strategies as part of the underlying run-time system may be easily incorporated and simulated for correctness of behavior and also quantitatively. As such, even if the time axis does not present real machine cycles of the user-specified platform emulated by hardware threads but rather the wall clock time of the platform where the X10 simulation has been run, not only different invasion strategies may be explored and visualized, but also different architectural platform settings and novel monitor concepts may be tested. Also, the (relative) times required for invasion, infection, and retreat from resources, obviously determining the overhead of invasive vs. non-invasive statically allocated programs, may be put into perspective.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced a language implementation of invasive computing using the X10 programming language as well as a first framework to compile and functionally simulate resource-aware invasive programs on a dedicated MPSoC platform, using the concept of hardware threads and monitor functions. In summary, our framework of X10-based functional simulation of invasive programs shall be used as an early test-bed for resource-aware programming of individual MPSoC architectures at a stage much earlier then their design. Hence, we expect a true and threefold benefit for as well a) application-programmers as b) operating

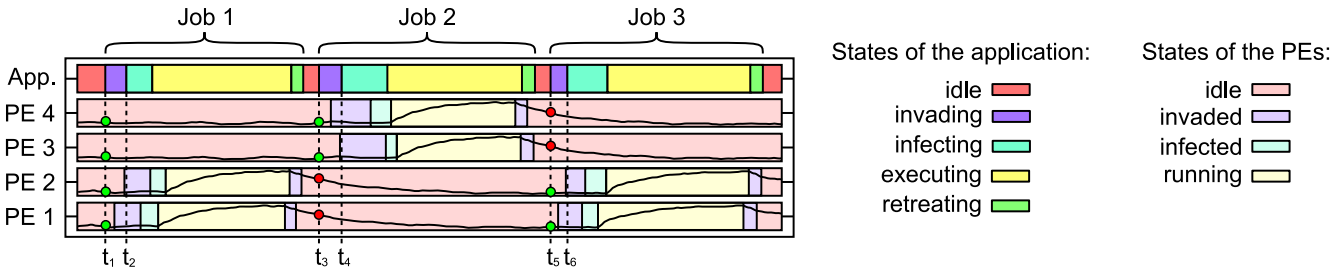

Figure 4: Gantt chart of the execution of the resource-aware parallel histogram batch processing application claiming 2 PEs for each job processed on a 4 PE tile according to the lower right tile in Figure 2. The decision which PEs are claimed for the execution of each job is done inside the corresponding invasion intervals $[t_1, t_2]$, $[t_3, t_4]$, and $[t_5, t_6]$. It can be seen that according to the max. temperature constraint of 80°C, e. g., reached by PE1 and PE2 in interval $[t_3, t_4]$, the individual resource invasion for each job may lead to different PE assignments.

system and run-time system developers and c) architecture designers of next generations MPSoC platforms.

In the future, we would like to extend this framework to also include the modeling and the simulation of other resources such as the invasion of memories and communication resources (e. g., NoCs) and to provide a proper timing model such that design space explorations of invasive applications and architectures shall become possible.

6. ACKNOWLEDGMENTS

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

7. REFERENCES

[1] U. Aßmann. *Invasive Software Composition*. Springer, 2003.

[2] G. Bikshandi, J. Castanos, S. Kodali, V. Nandivada, I. Peshansky, V. Saraswat, S. Sur, P. Varma, and T. Wen. Efficient, Portable Implementation of Asynchronous Multi-Place Programs. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 271–282, Raleigh, NC, USA, 2009. ACM.

[3] O. Certner, Z. Li, P. Palatin, O. Temam, F. Arzel, and N. Drach. A Practical Approach for Reconciling High and Predictable Performance in Non-Regular Parallel Programs. In *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 740–745, Munich, Germany, Mar. 2008.

[4] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielsstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538. ACM, 2005.

[5] S. J. Deitz, D. Callahan, B. L. Chamberlain, and L. Snyder. Global-view abstractions for user-defined reductions and scans. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, New York, NY, USA, 2006. ACM.

[6] T. Ebi, M. Al Faruque, and J. Henkel. TAPE: Thermal-Aware Agent-Based Power Economy for Multi/Many-Core Architectures. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, San Jose, CA, USA, Nov. 2009.

[7] T. El-Ghazawi and L. Smith. UPC: Unified Parallel C. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, Tampa, FL, USA, 2006. ACM.

[8] J. Jahn, M. Al Faruque, and J. Henkel. CARAT: Context-Aware Runtime Adaptive Task Migration for Multi Core Architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 515–520, Grenoble, France, March 2011.

[9] OpenMP Architecture Review Board. OpenMP Application Program Interface – Version 3.0, 2008. `http://www.openmp.org/mp-documents/spec30.pdf`.

[10] P. Palatin, Y. Lhuillier, and O. Temam. CAPSULE: Hardware-Assisted Parallel Execution of Component-Based Programs. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 247–258, Orlando, FL, USA, Dec. 2006. IEEE Computer Society.

[11] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. Keckler, and D. Burger. Distributed Microarchitectural Protocols in the TRIPS Prototype Processor. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 480–491, Orlando, FL, USA, Dec. 2006. IEEE Computer Society.

[12] Sun Microsystems Inc. The Fortress Language Specification, 2005. `http://research.sun.com/projects/plrg/fortress0618.pdf`.

[13] J. Teich. Invasive Algorithms and Architectures. *it - Information Technology*, 50(5):300–310, 2008.

[14] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting. *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, chapter 11, Invasive Computing: An Overview, pages 241–268. Springer, 2011.

[15] The MPI Forum. MPI-2.0 Specification. `http://www.mpi-forum.org/docs/`.

[16] F. Thoma, M. Kühnle, P. Bonnot, E. Panainte, K. Bertels, S. Goller, A. Schneider, S. Guyetant, E. Schüler, K. Müller-Glaser, and J. Becker. MORPHEUS: Heterogeneous Reconfigurable Computing. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 409–414, Aug. 2007.