RESOURCE AWARENESS FOR EFFICIENCY IN HIGH-LEVEL PROGRAMMING LANGUAGES

Andreas Zwinkau

Karlsruhe Institute of Technology
zwinkau@kit.edu

ABSTRACT

Managing hardware resources is important to write efficient software, which conserves energy, time, and money. High-level programming languages by definition abstract from the underlying machine, such that efficiency should be recouped by compiler and runtime. Our Invasive Computing project especially targets multi-application scenarios, where resources must be dynamically reallocated for optimal efficiency. Integrating hardware and software developers into the process, we developed a framework within the high-level X10 language, which enables an application to be aware of different kinds of resources and their availability. We show how applications can adapt to different architectures and dynamically exchange resources to optimize the system as a whole.

1. INTRODUCTION

Moore's Law of exponentially increasing transistor counts per chip area appears to hold for the next years. However, due to technical issues, chip producers have been unable to translate "more transistors" into "higher clock rates" for the last decade. Instead, multi-core architectures were introduced. Those increasingly parallel architectures seem to be hard to exploit, though. The speedups are diminishing [1] as the concurrency overhead grows with the parallelism.

The next trend to exploit the growing transistor count are heterogeneous architectures. Non-parallel applications need a single core with the best performance, even at the cost of complexity and transistor count. In contrast, an embarrassingly parallel application can use a lot of slow simple cores. An example of this trend is the rise of GPU computing, although they are harder to program than the common CPU. Current graphic cards are essentially massively parallel computers. While they provide homogeneous processing elements, they are used in concert with the different CPU. The importance of efficient programs is actually increasing despite the fact that our machines become increasingly powerful. The main reason is the trend to mobile devices which run on batteries. Though, for desktops and servers the energy consumption is also important, such that systems are evaluated in terms of "performance per watt" instead of raw performance.

1.1. High-Level Programming

In the 70s the C programming language was considered "high-level", because it abstracts from the hardware and makes software portable. Today most programmers consider C low-level or "portable assembly". A modern (mainstream) notion of high-level usually includes

- garbage collection, since Java showed it can be done efficiently and popularized it.
- object orientation, which is currently the most popular paradigm for structuring applications.
- type safety, although the mainstream languages all provide workarounds to break type safety if necessary.
- higher-order functions are increasingly integrated into modern languages to exploit the advantages of functional programming.
- a big standard library, which supports for example hash maps, XML processing, and regular expressions.

The X10 programming language fits this profile, since it is derived from Java and integrates additional features, such as higher-order functions. The standard library is quite small though, because the language is relatively young.

1.2. Resource Awareness

The term "resource awareness" is often used with different meanings. The question is: which resources and how are they managed? For an abstract notion of "resource", we can reduce it to a classification within the following properties:

multiplexing Can parallel use be sequentialized? Network connections are usually multiplexed, when various applications send packets simultaneously. In terms of efficiency, multiplexing is usually a bad idea, since it introduces overhead and impedes optimal exploitation. However, it also provides the pleasant abstraction of virtually infinite resources with limited hardware.

sharing Can it be used by multiple owners at the same time? For example, memory can be shared, but a CPU core cannot (disregarding hyperthreading). While sharing is very efficient, it burdens the synchronization job on the application.

division Can it be split into multiple independent resources? Memory is usually divided into address spaces, to secure processes against each other. In contrast, a CPU cannot be divided.

reuse Can it be reused afterwards? For example, a CPU can be reused, while compute time on a CPU cannot.

A resource like memory is classified differently, depending on who uses it under which circumstances. For example, threads share memory, but processes do not.

For a notion of "awareness", we classify the management of a resource in two dimensions: implicit/explicit and internal/external. For example, consider memory again. The explicit internal variant is `malloc` and `free` in C. The implicit internal variant is to use a garbage collector. The accompanying explicit external variant is `sbrk` and `mmap` on Unix. Finally, the implicit external variant is the paging and swapping of an operating system. For this paper, we do not consider the implicit external approach, since we are concerned with the language level of this problem.

1.3. Invasive Computing

The Invasive Computing paradigm [2] suggests a resource-aware programming model, where the program can dynamically *invade* available resources, e.g., processing elements (PEs), memory and network connections. The invasion of resources can be specified by constraints. This allows for example to request a certain number of resources or to target specific hardware. After the invasion is done, the program *infects* the invaded resources by using them for a certain computation. If the resources are not needed anymore, the program *retreats* from the resources.

Invasive Computing improves efficiency through multiple approaches. First, dynamic reallocation improves the overall resource use. compared to static resource allocation. Resources are shifted between or within applications under consideration of the global state. Second, exposing resources more directly to the applications, instead of virtualizing them, removes overhead. For example, CPU cores are assigned to single applications, instead of multiplexing them. Third, the invasion of the most suited hardware can improve efficiency. For example, not all processing elements in a heterogeneous system might support floating point arithmetic in hardware. To exploit such heterogeneity, the hardware must be exposed to the programmer. Therefore, the invasive programming paradigm affects the application, the programming language, the compiler, and the operating system.

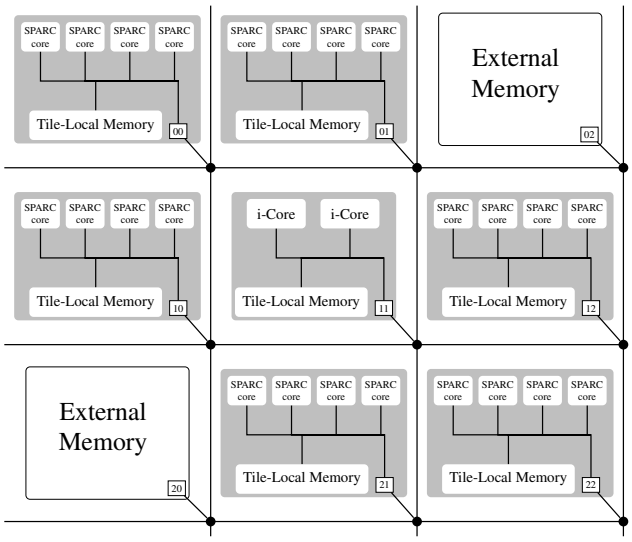

Fig. 1. Exemplary invasive architecture: Each tile is connected to the network on chip. Two tiles (02 and 20) are connectors to external memory. The other tiles each contains tile-local memory and either two i-Cores or four SPARC cores.

1.3.1. Invasive Architecture

Figure 1 shows an example of an invasive architecture [3]. It consists of two different variants of cores. The first variant are standard SPARC cores. The second variant are *i-Core*s, which are reconfigurable SPARC cores that can load accelerators to speed up certain computations [4]. The processors are grouped into tiles, which are connected by a network on chip. External memory is attached via special tiles, in addition to per-tile and per-core memory. Per-tile and external memory are visible in the global address space, in contrast to per-core memory. Each tile contains a small number of cores, so that cache coherence can be achieved with a bus snooping cache coherence protocol. However, caches in different tiles are not coherent, so the memory architecture scales well with the number of tiles.

1.3.2. Software Platform

As shown in Figure 2, an invasive application runs on the invasive runtime support system (iRTSS), which consists of per-tile instances of the operating system OctoPOS [5] and an agent system [6] for global resource management. Within the agent system, each application is represented by one agent. If an application wants to invade additional resources, it's agent checks whether and how the request can be fulfilled. In case of concurrent requests for the same resource, the corresponding agents are responsible for finding a suitable solution. Since only the competing agents are involved, this approach scales well.

On the language level, we encapsulated the resource-

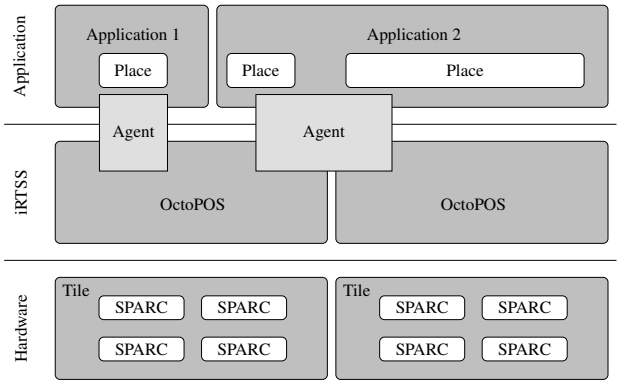

Fig. 2. The InvasIC stack: Two tiles of four SPARC cores each are executing two instances of OctoPOS. Two X10 applications are running on top; Each one has an agent for global resource management. Application 2 spans two tiles, so it has two places.

aware features in an X10 library. We chose X10 because it already provides various features that make it suitable for programming invasive architectures. For example, the partitioned global address space (PGAS), which models a cluster network, fits quite naturally to a cache incoherent multi-core architecture. The employed X10 concept is a *place*, such that each tile maps to a place. Essentially, threads within the same place have shared memory, whereas threads in different places must communicate with other means.

Programming with incoherent caches is challenging. One obviously correct programming model is the partitioning of available memory, assigning a part of it to each tile. This PGAS model is mapped to X10 by representing each tile as a place. The exchange of information between tiles only happens on **at** expressions, so the runtime system can ensure a correct sequence of cache flushing and synchronization. Partitioning the address space inhibits sharing a memory cell between multiple tiles, although the architecture would allow this. However, it is possible [7] to take advantage of multiple readers to alleviate the costs of copying data between partitions.

1.4. Outline

In Section 2 we present the invasive programming framework for X10 and show examples of resource awareness in Section 3. In Section 4 we contrast to other publication on resource aware programming.

2. THE INVASIVE FRAMEWORK

For a new paradigm like Invasive Computing a common language is necessary, so people do not talk at crossed purposes. The invasive framework fulfills a core role in unifying the language and meaning.

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

Fig. 3. The basic idea of invasive programming: Invade allocates resources under specific constraints in competition with other applications; Infect uses those resources by letting iLets (similar to kernels in embedded computing) run; Retreat frees allocated resources.

2.1. Development Process

Since the language of a paradigm influences hardware and software design, we used a process where everybody was integrated and encouraged to participate. From hardware developers to application programmers we collected pseudo code snippets to capture their understanding of resource awareness, invasion, processing element, and other terminology. Based on those code examples, we developed a first version of the framework in X10. Using this concrete code, our partners ported their pseudo code examples into X10, which can already be executed by simulating an invasive platform [8, 9, 10].

At this point the invasive language was stable enough to modularize further updates. A document-based process, inspired by PEPs [11], JSRs [12], and RFCs [13], was introduced to document and design further enhancements.

2.2. Basic Invasive Application

The basic idea of invasive programming is demonstrated in Figure 3. The concept of allocating, using and freeing resources is known from memory allocation. Invasive computing generalizes the concept to invade, infect and retreat under specific constraints. So far the framework supports the following reusable resources:

Processing element Not multiplexed, shared, or divisible.

Tile-local memory Not multiplexed; Shared within the tile; Divisible between and within applications.

Inter-tile network connections Multiplexed, but not shared or divisible by default. Applications can invade connections to a certain extend, such that the network divides the bandwidth accordingly.

Constraints for invasion are structured in an extensible hierarchy, which is shown in Figure 4. 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

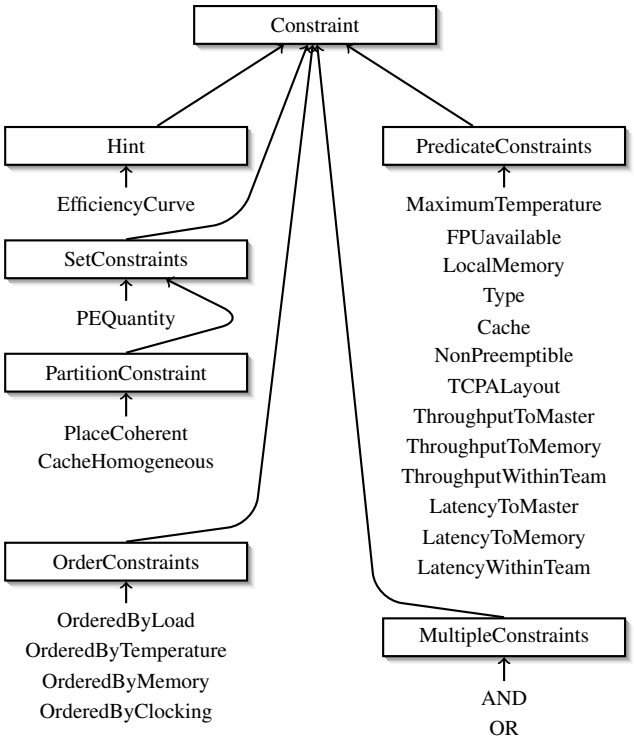

Fig. 4. The constraint hierarchy for invasion. Boxes represent abstract categories and specific constraints inherit from them.

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.

2.3. Hardware Support

To be aware of hardware resources, it is necessary for the hardware to provide information. For example, temperature monitors can be used to choose cooler processing elements to reduce the need for cooling or avoid hardware failure.

An important question is which parts to implement in hardware or software for each resource. For example, the bus throughput and latency guarantees are completely handled and enforced in hardware. The software only configures the bus controllers. In contrast, distributing applications across tiles is completely managed by the agent system in software, because the distributed algorithm is too complex for hardware. However, assigning threads across the cores within a tile is supported by hardware [14].

```
val constraints = new AND();
constraints.add(new FPUavailable());
var claim:Claim = null;
if (currentPE.hasFPU()) {
  constraints.add(new PEQuantity(0,3));
  claim = Claim.invade(constraints);
  claim.andCurrentPE().infect(ilet);
} else {
  constraints.add(new PEQuantity(1,4));
  claim = Claim.invade(constraints);
  claim.infect(ilet); }
```

Fig. 5. Explicit internal resource awareness: If the current PE has an FPU, we can use it for the infection and only invade up to three PEs, otherwise one more PE has to be invaded.

```
for (image in videostream) {
  val claim = Claim.invade(constraints);
  claim.infect(videofilter);
  claim.retreat(); }
```

Fig. 6. Invasive video filter: For each image in the video stream an extra invade is issued. Depending on the system load, the claim size might be different on each call.

3. EXAMPLES OF RESOURCE-AWARE PROGRAMMING

We now show examples for explicit internal, explicit external, and implicit internal resource management. We are not concerned of implicit external resources, because this is not tackled on the language level.

3.1. Consider Invasion Overhead

Resource allocation in Invasive Computing is done with an invade Call. However, this is an expensive operation, since it involves a distributed consensus algorithm. Hence, it makes sense to avoid this cost when possible. While the system can provide shortcuts, some resource information must be exploited by the application itself in an explicit internal way.

For example, one could remove the floating point unit (FPU) from some cores in a system, since they use relatively large amounts of silicon and energy. Many programs are fine with only fixed point arithmetic, but we can imagine a program, which requires floating point occasionally. For example in Figure 5, let the code in ilet require floating point arithmetic, but the surrounding code does not. Depending on a type of the current PE, different constraints are specified.

```
val claim = Claim.invade(constraints);
val master = here;
val activePEs = new List[Boolean](claim.size(), true);
val ilet = (id:IncarnationID) => {
  while (moreChunks() && !id.mustTerminate()) {
    val work_package = getChunk();
    localSort(work_package); }
  at (master)
    atomic { activePEs(id.ordinal()) = false; } }
val handler = (removed:List[PE], added:List[PE])=>{
  if (!claim.active()) return;
  claim.addPEs(added); // no op, if empty
  for (pe in removed) // signal termination
    pe.setTerminateFlag();
  for (pe in removed) // wait for termination
    when (!activePEs(pe.ordinal())) ; }
claim.setResizeHandler(handler);
claim.infect(ilet);
```

Fig. 7. Excerpt of Resizable Parallel Sort: Individual workers (ilet) repeatedly get a chunk to sort locally. The loop exits, if there are no more chunks or the terminate flag is set. When they finish, they notify the master accordingly. The master registers a handler function in the claim, which is called, whenever a change to the claim is issued by the external agent system. There is nothing to do, if the claim is currently infected, since the master is the only one currently running. The handler function adds and removes PEs according to the agent systems decision and waits for their the termination of the removed workers. This leaves time for the workers to finish their current work package, then the mustTerminate call will return true and the loop finishes.

3.2. Reinvasion for Adaptation

Every invasion is an example of implicit internal resource management, but to exploit that for a system's efficiency an application must use invade and retreat often. This means resources are given back to the system and reinvaded when necessary. Others can use them meanwhile. Of course, the granularity for this pattern depends on the time the invasion takes. An example for this pattern is shown in Figure 6. The resource awareness is application internal, but implicitly left to the agent system.

3.3. Hints and Dynamic Reallocation

Many applications are flexible in their use of processing elements. Some can even adapt during the computation. Usually, such applications are embarrassingly parallel and can divide the work into small work packages without interdependencies. For example, one phase of parallel sorting [15] fits this profile, as the data is split into many chunks which are sorted

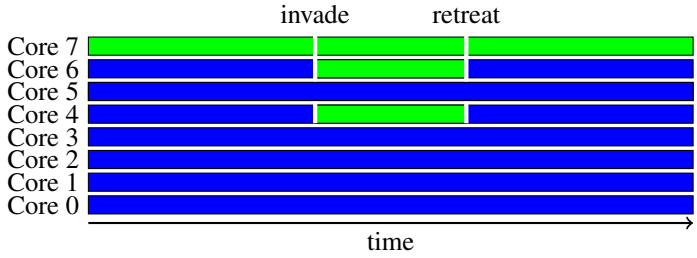

Fig. 8. Another job partially stalls the resizable sort in between, as the agent system decides that the five other cores are enough for sorting.

locally first, as shown in Figure 7. The agent system might decide that another application requires some of the resources the sorting is currently using. The application is notified that it has to retreat from certain resources, so it has the chance to reorganize data. Afterwards the free resources can be used by the other application. When it retreats and the sorting is not finished, the agent system can notify the application again that additional resources are provided.

Since we have no complete invasive architecture so far, we can only show data from a synthetic functional simulation in Figure 8. Some preliminary tests [16] suggest that partially stalling the sorting provides better performance than multiplexing the cores via preemptive threading, probably due to overhead of context switching and cache flushing.

4. RELATED WORK

Moreau and Queinnec [17] developed a resource-aware programming framework for Java. It is used to manage resources like compute time, disk space and network throughput. The intention is execute untrusted code with finer grained controls than the Java SecurityManager provides. The user programs are not aware of the resource management, but a supervising manager process is. In contrast, invasive applications are aware of the framework. Their conclusion mentions the need for an abstract policy language, where they probably had something like our constraints in mind. Similarly, Janos [18] provides resource control over the execution of untrusted Java bytecode, where the resources are memory pages, CPU cycle rates, and network throughput. While the application must be specifically written for this environment, they are not resource-aware themselves.

The Sumatra framework [19] focuses on mobile programs, which can migrate control threads between address spaces, similar to the X10 **at** mechanism. In contrast, mobility is not the focus of Invasive Computing, since we target an MPSoC where address space separation exists only on the language level. Also in contrast to Invasive Computing, Sumatra builds on top of Java, so it cannot be aware of the

instruction set as a resource.

Kowalik et al [20] improved a mesh routing protocol by integrating a resource-aware cost heuristic. By passively monitoring signal strength, interference, and bandwidth, it does not impose any overhead for the network. This demonstrates the gain through resource awareness on the driver level.

Taha [21] uses resource-aware programming to name a class of programming languages, which (1) provide abstraction mechanisms like higher-order functions and objects within clear semantics, (2) support staged compilation, and (3) can statically check resource usage like memory consumption. Taha's primary example is a program, which generates an fast fourier transformation circuit from the recurrence equation. Such static analysis of resources is future work for the invasive framework.

The term "resource-aware programming" is used for very different approaches, ranging from grid-computing to hardware design. However, all those approaches concentrate on homogeneous platforms (usually Java), which excludes the support for architecture-specific resources. In contrast, invasive computing supports heterogeneous architectures.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we described resource aware programming in the context of Invasive Computing. We showed how programs written in a high-level language like X10 can be resource aware and how this yields efficiency gains. Although, we cannot test this empirically yet, simulation and preliminary tests suggest that Invasive Computing is worthwhile.

6. ACKNOWLEDGMENTS

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

7. REFERENCES

[1] Herb Sutter, "Welcome to the Jungle," 2011.

[2] Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting, "Invasive computing: An overview," in *Multiprocessor System-on-Chip – Hardware Design and Tool Integration*, pp. 241–268. Springer, Berlin, Heidelberg, 2011.

[3] Ravi Kumar Pujari, Thomas Wild, Andreas Herkersdorf, Benjamin Vogel, and Jörg Henkel, "Hardware assisted thread assignment for RISC based MPSoCs in invasive computing," in *Proceedings of the 13th International Symposium on Integrated Circuits (ISIC)*, Dec. 2011.

[4] Jörg Henkel, Lars Bauer, Michael Hübner, and Artjom Grudnitsky, "i-Core: A run-time adaptive processor for embedded multi-core systems," in *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 2011)*, July 2011, invited paper.

[5] Benjamin Oechslein, Jens Schedel, Jürgen Kleinöder, Lars Bauer, Jörg Henkel, Daniel Lohmann, and Wolfgang Schröder-Preikschat, "OctoPOS: A parallel operating system for invasive computing," in *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA)*. EuroSys, Apr. 2011, vol. USB Proceedings of *Sixth International ACM/EuroSys European Conference on Computer Systems (EuroSys)*, pp. 9–14.

[6] Sebastian Kobbe, Lars Bauer, Jörg Henkel, Daniel Lohman, and Wolfgang Schröder-Preikschat, "DistRM: Distributed resource management for on-chip many-core systems," in *Proceedings of the IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct. 2011, pp. 119–128.

[7] Matthias Braun, Sebastian Buchwald, Manuel Mohr, and Andreas Zwinkau, "An x10 compiler for invasive architectures," Tech. Rep. 9, Karlsruhe Institute of Technology, 2012.

[8] Frank Hannig, Sascha Roloff, Gregor Snelting, Jürgen Teich, and Andreas Zwinkau, "Resource-aware programming and simulation of MPSoC architectures through extension of X10," in *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. June 2011, pp. 48–55, ACM Press.

[9] Sascha Roloff, Frank Hannig, and Jürgen Teich, "Approximate time functional simulation of resource-aware programming concepts for heterogeneous MPSoCs," in *Proceedings of the 17th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2012, pp. 187–192.

[10] Sascha Roloff, Frank Hannig, and Jürgen Teich, "Fast architecture evaluation of heterogeneous MPSoCs by host-compiled simulation," in *Proceedings of the 15th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. May 2012, ACM Press.

[11] Barry Warsaw, Jeremy Hylton, and David Goodger, "PEP1: PEP Purpose and Guidelines," 2000.

[12] Oracle America Inc., "JCP2: The Java Community Process," 2011.

[13] S. Bradner, "RFC2026: The Internet Standards Process – Revision 3," 1996.

[14] Ravi Kumar Pujari, Thomas Wild, Andreas Herkersdorf, Benjamin Vogel, and Jörg Henkel, "Hardware assisted thread assignment for RISC based MPSoCs in invasive computing," in *Proceedings of the 13th International Symposium on Integrated Circuits (ISIC)*, Dec. 2011.

[15] Mirko Rahn, Peter Sanders, and Johannes Singler, "Scalable Distributed-Memory External Sorting," in *26th IEEE International Conference on Data Engineering (ICDE)*, 2010, pp. 685–688.

[16] Patrick Flick, "Paralleles Sortieren als malleable Job," bachelor thesis, Karlsruhe Institute of Technology, 2011.

[17] Luc Moreau and Christian Queinnec, "Resource aware programming," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, pp. 441–476, May 2005.

[18] Patrick Tullmann, Mike Hibler, and Jay Lepreau, "Janos: A java-oriented OS for active network nodes," in *IEEE Journal on Selected Areas in Communications*, 2001, pp. 501–510.

[19] Anurag Acharya, M. Ranganathan, and Joel Saltz, "Sumatra: A language for resource-aware mobile programs mobile object systems towards the programmable internet," vol. 1222 of *Lecture Notes in Computer Science*, chapter 10, pp. 111–130. Springer Berlin / Heidelberg, Berlin, Heidelberg, 1997.

[20] K. Kowalik, B. Keegan, and M. Davis, "Making OLSR aware of resources," in *Wireless Communications, Networking and Mobile Computing, 2007. WiCom 2007. International Conference on*. 2007, pp. 1488–1493, IEEE.

[21] Walid Taha, "Resource-Aware programming embedded software and systems," vol. 3605 of *Lecture Notes in Computer Science*, chapter 6, pp. 38–43. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2005.