Resource-aware Programming in a High-level Language

Improved performance with manageable effort on clustered MPSoCs

Zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften
(Dr.-Ing.)

bei der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte
DISSERTATION

von

Dipl.-Inform. Andreas Zwinkau

Datum der mündlichen Prüfung:	2018-04-26
Referent:	Prof. Dr.-Ing. Gregor Snelting
Korreferent:	Prof. Dr.-Ing. Jürgen Teich

Abstract

Until 2001 Moore's Law and Dennard Scaling implied that execution speed doubled every 18 months due to better CPUs. Today, concurrency is the dominant way for speedups from supercomputers to mobiles. However, more recent phenomenons like Dark Silicon increasingly complicate speedups from hardware. To realize further performance gains, software has to become more aware of the hardware resources. A related phenomenon is increasingly heterogeneous hardware. Supercomputers integrate accelerators like GPUs. Mobile SoCs (for example in smartphones) integrate more and more features. Exploiting special hardware is a well-known technique to lower energy consumption, which is another important aspect that must be balanced with raw performance. For example, supercomputers are also rated by "performance per watt". Currently, low-level programmers are used to think about hardware, while the mainstream high-level programmers prefer to abstract as much of the platform as possible (for example clouds). High-level does not imply that hardware is irrelevant, just that it can be abstracted. If you write a Java application for Android, battery use might be an important aspect. Eventually, even high-level programming languages are pressured to become resource-aware to improve speed or energy consumption.

Within the transregional collaborative research center "Invasive Computing", I worked on these problems. In my dissertation, I propose a framework to make high-level applications resource-aware and thus improve performance, which for example might result in improved efficiency or speedups for the system as a whole.

One core idea is that applications do not optimize on their own. Instead, they give information to the operating system. The operating system with its global view makes resource decisions. This process we call "invasion" of resources. The job of the application is to adapt to the operating system's decision, not to

make its own. The challenge is to define the language, which applications use to communicate resource constraints and performance hints. Such a language must be expressive enough for complex information, extensible for future resource types, and convenient for the programmer.

The major contributions in this dissertation are:

- A theoretic model of resource allocation to precisely describe the essence of the resource-aware framework, to reason about the correctness of the operating system decisions with respect to the constraints of an application, and to proof my claims of efficiency and speedup in theory.
- A framework and compilation path for resource-aware programming implemented for the high-level programming language X10. We implemented applications from High Performance Computing to evaluate this approach. Speedups of 5x can be demonstrated.
- A memory consistency model for the X10 programming language as a necessary step for a formal semantics, which bridges the theoretic model to the concrete implementation.

In one sentence: **Resource-aware programming in high-level languages on MPSoCs is feasible with manageable effort and improves performance.**

Zusammenfassung

Bis 2001 bedeutete Moores und Dennards Gesetz eine Verdoppelung der Ausführungszeit alle 18 Monate durch verbesserte CPUs. Heute ist Nebenläufigkeit das dominante Mittel zur Beschleunigung von Supercomputern bis zu mobilen Geräten. Allerdings behindern neuere Phänomene wie „Dark Silicon“ zunehmend eine weitere Beschleunigung durch Hardware. Um weitere Beschleunigung zu erreichen muss sich auch die Software mehr ihrer Hardware Resourcen gewahr werden. Verbunden mit diesem Phänomen ist eine immer heterogenere Hardware. Supercomputer integrieren Beschleuniger wie GPUs. Mobile SoCs (bspw. Smartphones) integrieren immer mehr Fähigkeiten. Spezialhardware auszunutzen ist eine bekannte Methode, um den Energieverbrauch zu senken, was ein weiterer wichtiger Aspekt ist, welcher mit der reinen Geschwindigkeit abgewogen werde muss. Zum Beispiel werden Supercomputer auch nach „Performance pro Watt“ bewertet. Zur Zeit sind systemnahe low-level Programmierer es gewohnt über Hardware nachzudenken, während der gemeine high-level Programmierer es vorzieht von der Plattform möglichst zu abstrahieren (bspw. Cloud). „High-level“ bedeutet nicht, dass Hardware irrelevant ist, sondern dass sie abstrahiert werden kann. Falls Sie eine Java-Anwendung für Android entwickeln, kann der Akku ein wichtiger Aspekt sein. Irgendwann müssen aber auch Hochsprachen resourcengewahr werden, um Geschwindigkeit oder Energieverbrauch zu verbessern.

Innerhalb des Transregio „Invasive Computing“ habe ich an diesen Problemen gearbeitet. In meiner Dissertation stelle ich ein Framework vor, mit dem man Hochsprachenanwendungen resourcengewahr machen kann, um so die Leistung zu verbessern. Das könnte beispielsweise erhöhte Effizienz oder schnellerer Ausführung für das System als Ganzes bringen.

Ein Kerngedanke dabei ist, dass Anwendungen sich nicht selbst optimieren. Stattdessen geben sie alle Informationen an das Betriebssystem. Das Betriebssystem hat eine globale Sicht und trifft Entscheidungen über die Resourcen. Diesen Prozess nennen wir „Invasion“. Die Aufgabe der Anwendung ist es, sich an diese Entscheidungen anzupassen, aber nicht selbst welche zu fällen. Die Herausforderung besteht darin eine Sprache zu definieren, mit der Anwendungen Resourcenbedingungen und Leistungsinformationen kommunizieren. So eine Sprache muss ausdrucksstark genug für komplexe Informationen, erweiterbar für neue Resourcentypen, und angenehm für den Programmierer sein.

Die zentralen Beiträge dieser Dissertation sind:

- Ein theoretisches Modell der Resourcen-Verwaltung, um die Essenz des resourcengewahren Frameworks zu beschreiben, die Korrektheit der Entscheidungen des Betriebssystems bezüglich der Bedingungen einer Anwendung zu begründen und zum Beweis meiner Thesen von Effizienz und Beschleunigung in der Theorie.
- Ein Framework und eine Übersetzungspfad resourcengewahrer Programmierung für die Hochsprache X10. Zur Bewertung des Ansatzes haben wir Anwendungen aus dem High Performance Computing implementiert. Eine Beschleunigung von 5x konnte gemessen werden.
- Ein Speicherkonsistenzmodell für die X10 Programmiersprache, da dies ein notwendiger Schritt zu einer formalen Semantik ist, die das theoretische Modell und die konkrete Implementierung verknüpft.

Zusammengefasst zeige ich, dass resourcengewahre Programmierung in Hochsprachen auf zukünftigen Architekturen mit vielen Kernen mit vertretbarem Aufwand machbar ist und die Leistung verbessert.

Contents

Introduction

If you want to get eggs you can't buy at a store,
You have to do things never thought of before.

— Dr. Seuss

For decades we scaled our computer architectures according to a common formula, but this will end soon. It does not mean that our current technology will stop working, but on our path up the mountain of ever more powerful computing, there is a cliff. The following sections describe how long-term hardware trends lead to that cliff, what ideas there are to overcome it, why hardware alone cannot solve it, and my contributions to a solution.

Let us start by considering power. Digital computers require electrical power since their invention during the second world war. Power is a limited resource and thus a limiting factor for computation. If you own a smartphone or laptop, you most certainly want the battery to last longer. If you own a data or computing center, you most certainly would love to reduce your cooling and electricity bill.

The power a chip consumes is determined by two drains. We expend dynamic power $P_{dynamic}$ when transistors switch between 0 and 1 states and static power P_{static} due to leakage independent of the activity. We can observe which aspects of a chip contribute to power consumption. To reduce dynamic power consumption, we can

- use fewer transistors N, which is obvious but usually defeats the goal. We might do this temporarily for parts of the chip (clock gating).
- use smaller transistors to reduce capacitance C, which is what Moore's law [Moo65] stated in 1965 as the long term path for ever more powerful computers.
- lower the voltage V. This is a quadratic factor, so the power reduction is large in relation to the others. However, lowering the voltage also lowers transistor speed.
- lower the switch frequency f. This reduces power, but not the energy. We need more time units for the same task, as switch frequency corresponds to clock frequency.
- less switching activity A, which means to build more clever logic to avoid switching transistors.

Overall, dynamic power $P_{dynamic}$ is roughly $N \times C \times V^2 \times f \times A$ and the primary focus for managing power consumption.

In 1974, Robert Dennard observed [DGnY+74] that power density stays constant. This is known as "Dennard scaling" or Dennard's law. If you scale down the transistors by a factor of 2, you can a) reduce voltage and threshold voltage, b) increase clock frequency, and c) use the same power (assuming the same chip size). For example, if we switch from 180 nm technology to 90 nm technology, there is a factor of 2, which means 4 times as many transistors N on the same area. In lockstep, we lower voltage V and C by factor 2. For the same power, we can now also increase the switch frequency f (which roughly corresponds to clock frequency).

$$4N \times \frac{1}{2}C \times (\frac{1}{2}V)^2 \times 2f \times A \quad = \quad N \times C \times V^2 \times f \times A$$

Over five decades Dennard scaling meant the hardware speed doubled periodically and made all software exponentially faster.

1.1. Free Lunch is Over

At the beginning of the 21st century, Dennard scaling stopped. We found ourselves unable increase the clock frequency very much above 3 GHz. The main reason is that the threshold voltage V_t is now so low that the (exponentially

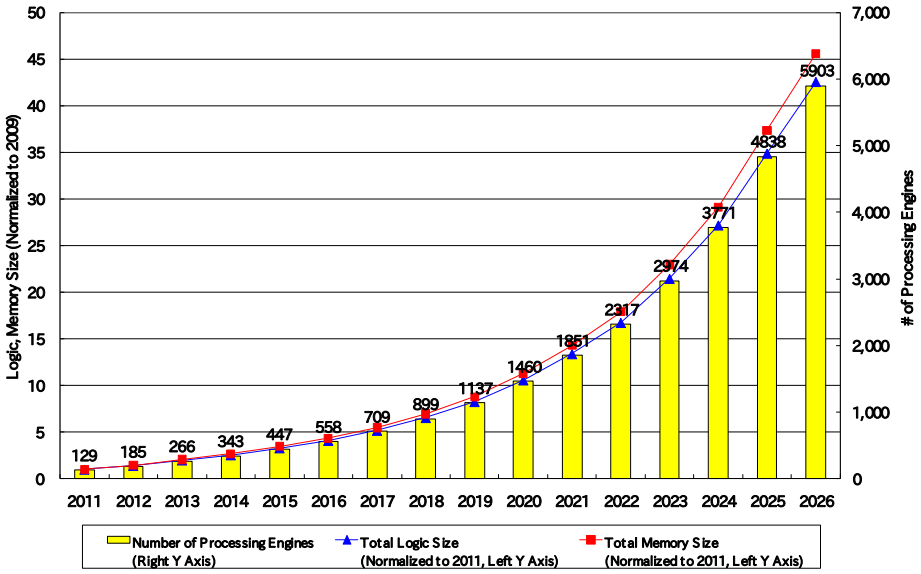


Figure SYSD5 SOC Consumer Portable Design Complexity Trends

Figure 1.1.: The International Technology Roadmap for Semiconductors (ITRS) featured this figure in 2011 [ITR11], which shows the exponential growth of processing engines. By 2019 it predicts over 1000 PEs. PE here does not directly correspond to cores as they exist in mainstream computers, but it still illustrates the trend.

growing) leakage is too large. Moore's law still keeps up, so we can put ever more transistors into the same area. Thus, we got increasing parallelism as the increase in transistors is translated to an increase in cores. The core speed does not increase much anymore, though. If this trend continues for the next years, we might see chips with a thousand cores by 2019. See fig. 1.1 for one such prediction.

In general, speedups these days come from parallelism, but not only in terms of multiple cores per chip. There are different forms of parallelism.

1. **Out-of-order execution** of instructions means that instructions can be computed in parallel, if there are no true data dependencies between them. Current Intel technology executes up to 4 parallel instructions, but

they might never increase this, as the code rarely provides opportunities for more.

2. **Vector instructions** meaning that a whole vector of values is computed in one step in a single instruction multiple data (SIMD) way. Examples would be the SSE and AVX instruction sets by Intel. The current AVX-512 provides 512 bit vectors split into 32 registers.
3. **Multicore** means multiple processing elements (cores) on a single chip within the same shared memory domain.
4. **Simultaneous multithreading** ("hyperthreading" as branded by Intel) means it looks like multicore to the application, but cores share most of their ALUs, and only one of the cores is active. This helps to hide memory latency, which can be over a 1000 cycles for one read from RAM. When a memory access blocks a CPU, the other CPU can continue.
5. **Clusters** means multiple computers in data centers.

All those tricks suffer from diminishing returns as illustrated by Amdahl's law:

$$T(n) = T(1)\left(B + \frac{1-B}{n}\right)$$

where $T(n)$ is the speedup for parallelism $n \in \mathbb{N}$, we know $T(1)$, and $B \in (0, 1]$ is the non-parallel fraction of the program. Even if $n \to \infty$, $T(n) = B\,T(1)$ never gets infinitely fast.

There are ways the hardware industry pursues to provide ever more computing power. However, the biggest gains might now be possible on the software side. Changing algorithms to enable more parallelism on all levels makes it possible to exploit more hardware parallelism.

1.2. The End of Moore's Law

Parallelism and concurrency are still hard topics. They have spawned new programming languages, tools, and abstraction layers. However, this era might be over soon as well. There are signs that even Moore's Law is nearing its end although current architectures still scale according to the plan, as shown in fig. 1.2. Intel announced a tick-tock clock in a 18 months rhythm in 2007. Each tick means the process technology shrinks the transistor size. Each tock means the microarchitecture is changed. In 2016 the clock stuttered. Instead of a tick

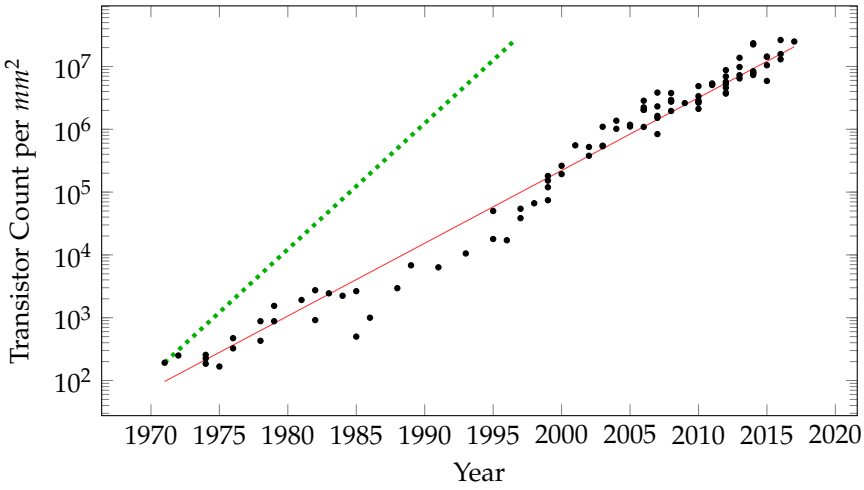


Figure 1.2.: According to data from Wikipedia, we still see exponential growth for the transistor count per area from the Intel 4004 in 1971 up to the AMD 8-core Ryzen in 2017. The red line shows the linear regression fits well. However, an increase from 192 in 1971 up to 25000000 in 2017 means it doubled every 2.4 years. The green dotted line shows a theoretical doubling every 18 months.

to 10 nm (codename "Cannonlake"), a second tock (codename "Koby Lake") updates the Skylake microarchitecture at 14 nm again. Intel slowed down the shrinking process. We can still shrink further for now, but not at the speed of "every 18 months".

The problem is that Complementary metal–oxide–semiconductor (CMOS) technology is reaching its physical limits. Silicon has a lattice spacing of 0.5 nm in its crystalline form. Various forms of leakage occur as we approach the limit and P_{static} becomes a much more significant drain than before. The transistor's gate oxide is so small that it leaks electrons. Leakage means the CPU burns energy while doing nothing and this means heat. If transistors heat up, they get slower and leak even more.

We can shrink transistors even more, but we cannot lower voltage or thin down the gate oxide accordingly. This would result in increased power density and a burned chip. Since this is not an option, the alternative is turn off

transistors which are not actively used; A phenomenon called "Dark Silicon". Predictions [EBSA+11] are 30% of a chip being dark at 20 nm and 80% at 8 nm.

For an analogy, consider the human brain. Most of the neurons in the brain are inactive [Leno3]. Yet, the brain can use up to 20% of the total body power [Bro99] while it is only 2% of the body's weight. This suggests that a body cannot support all neurons firing at a high rate. Maybe the power runs out, the brain overheats, or gets permanently damaged. Likewise, a future highly active computer will either burn or empty the battery.

There are four possible ways [EBSA+11] around this wall. First, we could refrain from building ever more cores into chips and just shrink the chips keeping their performance constant. Second, we could lower the clock frequency, so cores do not run that hot, but are slower individually. Third, we put a lot of specialized cores on the now heterogeneous chip, which are more efficient for specific tasks than generic cores. Fourth, we find a better transistor technology than the current CMOS approach. All of those are pursued by researchers and companies. If you want to improve performance incrementally the third option is the most promising.

1.3. Heterogeneity

Here, we define a "heterogeneous" CPU as a multiprocessing CPU, where the processing elements must be treated differently by the application or operating system. For example, they may differ in performance, instruction set, or monitoring. The definition is intentionally fuzzy because the concept of heterogeneity is subject to the use. If two processing elements have different cache sizes, one application might exhibit different performance, while another application achieves the same. For the latter application the processing elements are homogeneous.

Heterogeneity is an increasingly important topic for supercomputers. Since 2008 "accelerated" supercomputers are a significant part of the TOP500, as you can see in fig. 1.3. Today, this mostly means NVidia GPUs or Intel Xeon Phis. While this trend seems to have peaked in 2015 and since been in decline, even homogeneous supercomputers are not quite homogeneous anymore. Data

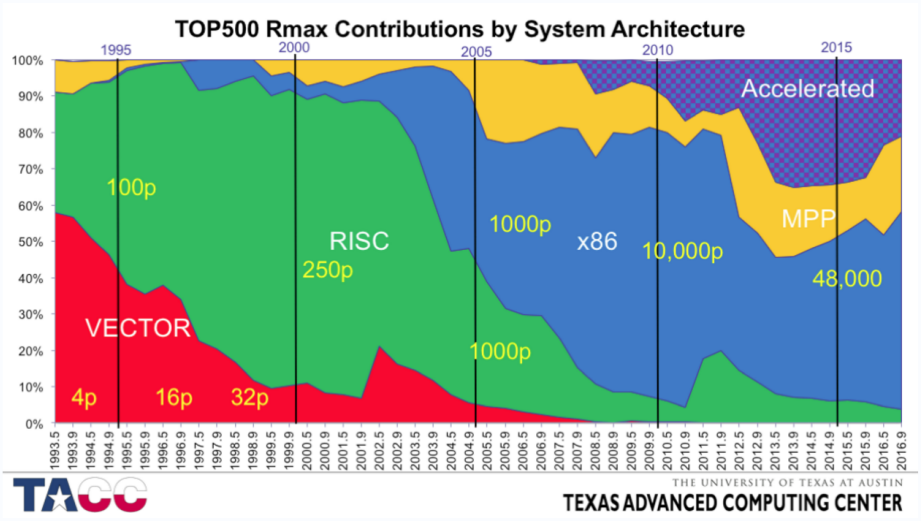


Figure 1.3.: Composition of TOP500 supercomputers over time [McC16]. The last Vector machine "Earth Simulator 2" dropped off the list in 2014. RISC machines are also nearly extinct, although the "K machine" was the fastest supercomputer in 2011. X86 dominates now. There is no clear definition of "MPP" supercomputers. Roughly, MPP means nodes are more tightly coupled than others. In this chart, MPP mostly equals IBM Blue Gene computers. Finally, the top right shows a significant part of "Accelerated", which means supercomputers enhanced with NVidia GPUs or Intel Xeon Phis (or IBM Cell in the early times).

centers build subclusters with a varying number of sockets per node and size of memory to optimize for different applications. This means even the "x86" parts in fig. 1.3 are often heterogeneous today.

For mobile computing, ARM's big.Little architecture from 2011 is a prime example of heterogeneity. It pairs high-performance processing elements (high cluster) with low-energy processing elements (low cluster). This hardware required adaptations to the Linux task scheduler under the project name "Energy Aware Scheduling". The first challenge was to integrate task scheduling, idle management, and frequency management, which were only loosely coupled before. This example demonstrates how the heterogeneity challenge is already seen in practice. Increasing heterogeneity further suggests ever more complex scheduling problems.

Also, heterogeneity not only comes from silicon differences. Today's chips use dynamic voltage and frequency scaling (DVFS). This implies that even identical processing elements not necessarily run at the same speed. For some applications this is undesirable, because it makes load balancing harder. If the load is not perfectly balanced, we observe idle resources and a suboptimal throughput. For an example, Karavadara [Kar16] investigated a way to tune voltage and frequency for reactive stream processing networks. Even without changing the applications, he could demonstrate reduced energy consumption with experiments on the Intel Single-Chip Cloud Computer (SCC) hardware [CLRB11].

Today, many ideas [BMF+16, CAB+13, ORS+04, WGH+07, SSM+07] are pursued to find a scalable design, where the number of cores scales linearly in development time, inter-dependencies, performance, etc. The closest architecture to the one this dissertation uses is probably Runnemede [CAB+13]. It aims to maximise energy efficiency for high-performance computing and targets applications like matrix multiplication. The project is an exploration of Intel to maximize efficiency and aim for exascale computing. Where our architecture only has two levels per CPU (PEs and tiles), Runnemede has three (execution engines, blocks, and units). Where we uses X10, classical Fortran and C++ code is used on Runnemede, as well as parallelized C code. Runnemede also uses custom hardware, like special instructions (`sincos`) and a special barrier network on chip. Programmability and adaptive resource management has no high priority. For example, no group of processing elements features cache coherence.

In general, we can observe that the hardware landscape is bound to become more heterogeneous in the next decades.

1.4. Clustered MPSoCs

Many people believe that cache coherence will not scale much further, but we still could increase the core count. This suggests a "clustered architecture", where groups of cores have a cache-coherent shared-memory system like current multicore computers. Between such clusters cache-coherence is not supported by the hardware. The name is inspired by "compute clusters" of the HPC domain, because the architecture feels similar: One node in a HPC cluster

is a multicore computer, but between nodes some message passing protocol must be used. The difference is that we are thinking about a single chip here instead of each chip being a single node.

This CPU architecture style originates from "tiled architectures", where the primary motivation was a scalable design without a special focus on cache coherence or shared memory. The original idea of tiled architectures is credited to the RAW processor [TLM+04]. In its first implementation it featured 4×4 tiles, each identical with a MIPS processor, FPU, small caches, and routers for the network-on-chip. Subsequently, the Tilera company was founded and in 2007 the 64-core TILE64 processor was released. Tilera's main selling point was energy-efficiency.

In 2009 Intel released the SCC. In contrast to RAW, it featured 2 cores per tile in 6×4 tiles and is usually run in a distributed fashion, where each tile runs its own operation system instance. Kalray can sell you a many core chip with 1024 processing elements clustered into tiles of 16 [Kal14]. They advertise 75 GFLOPS/W, which is state of the art.

For this work, we pursue the trend to more cores and more heterogeneity, so we assume a multiprocessor system-on-chip (MPSoC) with a clustered architecture. We also use the term "tile" for groups of cores as this is the familiar term in the hardware domain and a tile implicitly has shared and consistent memory. Section 3.2 describes more details on the hardware architecture used here.

1.5. Resource-Awareness

We observed the long term trends of computer hardware. Dennard's and Moore's law will end in the next years. This is the cliff mentioned initially. We face hardware platforms which will have many heterogeneous cores. To optimize performance we will have to optimize for energy consumption, exploit parallelism, and adapt to special hardware features.

Optimizing a single program is certainly a worthwhile task. However, today's systems rarely run only a single program. Even if it is a single program, it usually consists of more or less inter-dependent parts. For example, the probably most used program of a computer user today is the web browser. Modern browsers provide tabs to open many websites at once and to contain and isolate them. The needs of a website are very different depending on whether it shows a Youtube video, holds a conference voice call, displays static

text and images, renders a 3D WebGL scene, or plays a multiplayer game. To do each of those jobs in an efficient way, on a heterogeneous manycore system, we can schedule them to different processing elements. Some could be optimized for cryptography jobs, others for video decoding, and others for 3D rendering. For static text and images, we could use small, simple, slow processing elements, for a game, we could use big, complex, fast ones.

To clarify, we distinguish between scheduling and allocation. An "allocation" is to assign resources to applications[1]. "Scheduling" is to assign threads[2] to processing elements. Processing elements belong to applications because they are resources. Since every thread also belongs to an application, scheduling must respect allocation decisions and only assign threads to resources of the corresponding application.

Scheduling and allocating resources is the primary job of the operating system (OS). Current mainstream OSs excel in virtualizing to isolate programs. Processing elements are virtualized to threads. Memory is virtualized to isolate processes and to provide the illusion that every expressible address (usually 2^{64} today) is also accessible. Permanent storage (disk space) is virtualized to implement permissions and to change hardware transparently. Networking is virtualized to host many services on the same host. Such virtualization has its cost. For efficiency's sake, we might have to reconsider the virtualization mechanisms. For example, if the hardware provides more processing elements than there are applications, why do we need threads? Instead, we give each application its own processing element. However, conventional applications are not aware of resource management. Instead, system administration performs this job. In this work, we investigate the different paradigm of "resource-awareness", which means that applications actively consider their resource needs, communicate these needs, and adapt to the resource allocation of the system *dynamically*. We also assume that resources like processing elements are allocated *exclusively* as recent hardware trends make multiplexing unnecessary.

Naturally, changing a paradigm is impossible if you build on the status quo. Thus, we find related work in operating systems research, where there are no constraints by underlying software layers.

[1] or claims or agents or whatever entity owns resources
[2] or activities or tasks or jobs or whatever represents computation

One example is the Tesselation OS [CEH+13], which runs on mainstream Intel hardware and features a Resource Allocation Broker (RAB) service. Applications inform the RAB about high-level needs like a certain framerate and continuously communicate the current state with a heartbeat mechanism. The RAB also considers system information, e.g. cache miss rates, into account and adapts resource allocation. In Tesselation OS, the operating system makes allocation decisions, in contrast to applications. It runs on mainstream hardware (Intel i7) and does not target clustered or heterogeneous architectures, but it uses an exokernel approach, where applications get more direct access to hardware than with Linux, for example. The Application Heartbeats framework [MHS+10] uses a similar approach. They explored [MHS+11] approaches for OS decision making from heuristics over control modelling to machine learning. In both approaches the application has very little control, which means it is easier to program for, but has limited optimization potential.

In contrast, Rinnegan [PS16] leaves the decision making to the applications. They extended the Linux kernel to provide additional information about the system state. They investigate heterogenity in the sense of mainstream cores running at different clock speeds and outsourcing to the GPU. As Rinnegan does not allocate resources exclusively, some sophistication is necessary to avoid that application switch back and forth together in an infinite hunt for idle resources. Their experiments show that this decentralized approach is within 2-4% of a more informed centralized one.

Thanks to the rising interest in container technology, the Linux kernel has received a mechanism for resource management, control groups. The BarbequeRTRM framework [BMF15] exploits this and provides a resource manager and an interface for applications.

On a higher level than MPSoCs, Apache Mesos is a successful framework which provides resource management for cloud and cluster computing. The core idea of Mesos is that applications share nodes to improve data locality. The resource management happens with a two stage protocol, where applications are repeatedly offered resources, which they can then accept or ignore. There is no way for applications to guide resource management except simple boolean predicates, which act as filters. A related technology is Google's Heracles [LCG+15], which also distinguishes between low-priority best effort services and high-priority services (websearch).

Mesos is intentionally designed with a simple offer-reject mechanism to avoid complexity. The alternative would be "to provide a sufficiently expressive API to capture all frameworks' requirements, and to solve an online optimization problem for millions of tasks" [HKZ+11]. This thesis proposes such a "sufficiently expressive API". Others [KBL+11, Kob15] in Invasive Computing also work on the online optimization problem, but that is not part of this thesis.

In general, we can observe that resource-awareness provides measurable improvements. However, these techniques have not yet found their way into mainstream programming (except Mesos for cloud computing). The reason might be that there is no need, but that need will surely come if you consider the hardware trends described earlier. Another reason might be that we have not yet found the sweet spot between speedups due to special hardware and the complexity and effort to achieve it, which suggests a need for further probing. This thesis provides one more probe in this quest. We explicitly look at the language level of the resource-awareness problem, how to express and communicate resource needs and how to adapt to allocation changes.

We could have injected a resource management component into the runtime or operating system, which treats applications as black boxes [BIM08, SGS14] and thus does not require adapting the application code. This has the advantage of reusing old code, However, we choose the opposite path, because it means we avoid the limitations of backwards compatibility. Here, we explore the opportunities of opening the box (the application) and rewiring its inputs. Staying within the analogy, we even drill additional holes into the box to provide richer interaction between boxes and their environment. The interaction corresponds to the communication about resource needs and allocation changes of the resource-aware paradigm, of course. This design decision of the research project implies that legacy code is not a significant factor. Thus, the choice of programming language is less constrained.

1.6. High-Level Languages

Applications are increasingly parallel (and concurrent) today and scale to various numbers of processing elements. This means the OS must balance between the applications, which requires information about the applications like their scalability. Mainstream operating systems have no interface for such information exchange and allocation decisions. Also, applications usually do not look for a balance between heterogeneous resources, like CPU and GPU

in desktop computers, for example. There is a chicken-and-egg problem here, which manifests in a dialog like "What do you need?" – "What do you want?" This suggests that a basic research project is required to design the necessary interfaces and to investigate the advantages of such a system.

Now let us consider the programmer, who must develop software for such a platform. We must expect additional development costs. To adapt to special hardware features, each feature must be explicitly addressed by a programmer. This requires significant porting work. Optimizing for energy consumption might require additional flexibility, since the needs change over time, even over the lifetime of a single application. Finally, exploiting parallelism usually comes with concurrency, which is known to be full of pitfalls. To ease this burden, we would like to use a high-level programming language.

In general, programmer prefer to use the most high-level language possible to increase productivity and reduce errors. Originally, the C programming language was considered high-level, since it abstracted from the architecture-specific nature of assembly. However, writing architecture- or even platform-independent C is not easy. Thus, languages like Java were invented. They avoid undefined or unspecified behavior and provide memory- and type-safety. For example, the bit width of the `int` type is architecture-dependent in C, but always 32 bit in Java. Additionally, mechanisms like garbage collection are considered features of higher level languages since they abstract away the need to reason about the lifetime of objects.

The cost of abstraction is usually a decrease in performance, meaning speed or resource consumption. Of course, performance is foremost decided by the choice of algorithm. However once we are using a good algorithm, careful engineering can give further improvements. For example, Java programmers can fine-tune the garbage collector or call into C/C++ code to improve performance. Java lacks a compositional value type, which severely limits its usefulness for numeric tasks due to memory management and indirection overhead. In contrast, languages like C#, D, and X10 have a data type equivalent to C's `struct`. Garbage collection is a complicated topic. It provides memory safety at a cost. One downside of garbage collection is its indeterministic performance, which makes it a tool to avoid for soft real-time tasks like games and low latency servers. While there are real-time garbage collectors, their performance suffers. If the indeterministic performance is not an issue, garbage collection still requires more memory [HB05a] to reach competitive speed.

Invasive Computing chose X10 as its primary programming language. One reason is that X10 is designed as a "partitioned global address space" (PGAS) language, which makes it especially suitable for clustered architectures. It is also a high-level language, providing features like garbage collection, yet still targeting high performance. The reasons not to use OpenMP, MPI, UPC, Chapel, Fortress, OpenCL, or others are explained in the funding proposal [TH09]. Due to this choice, we also dismissed similar technology like GASPI, GPI-2, OpenSHMEM, Global Arrays, and OpenACC.

The described tradeoffs between high-level programming and performance are open problems. With this work we come from the high-level side of the problem and drill into performance territory. The goal is to keep the safety and the productivity, while significantly improving performance. The methodology is to build an interface for resource needs, which enables the OS to optimize allocation. We specifically target future MPSoCs to overcome the cliff at the end of Moore's law.

1.7. Dissertation Overview

Resource-aware programming in high-level languages on MPSoCs is feasible with manageable effort and improves performance. This is the thesis which this dissertation proves in the following structure.

Here, Chapter 1 describes the clustered MPSoC architecture. It also argued why optimizing sets of applications there is a relevant problem and explained the need for high-level languages. Now, we consider our specific approach of "resource-aware programming" and how it improves performance with manageable effort.

There is no clear definition of "resource-aware" and the term is used with many meanings in the literature. Thus, we call our specific approach "invasive". Chapter 2 shows a formal model of invasive resource allocation, which describes the essence of my resource-awareness without the overhead of a concrete implementation. There I prove that efficiency and utilization can only improve (theorems 1 and 2), and that makespan can improve by an unbounded factor (theorem 3).

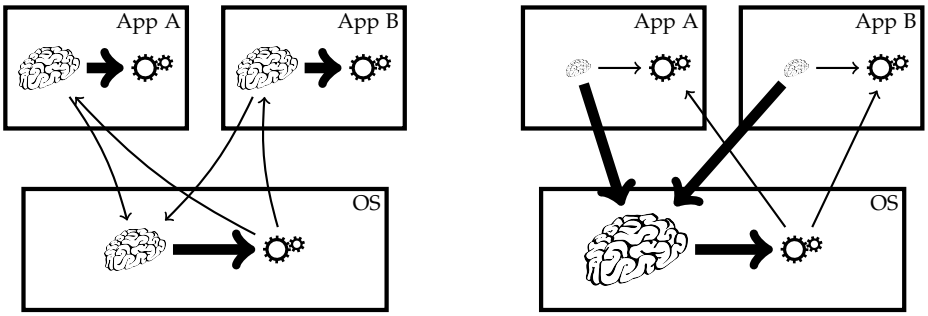


Figure 1.4.: On the left, current solutions are shown and on the right my resource-aware paradigm. The brains represent decision-making and the gears stand for the enactment of the decided policy. Note the cycle on the left: Application and OS adapt to each other in a feedback loop, which implies multiple steps to reach stability (if at all). Roughly, resource-awareness is about giving more decision power to the OS, represented by the different brain sizes. To enable this, more information must be provided to the OS, represented by thicker arrows. The advantage is that the OS decisions can be directly translated into application policy without the application thinking on it again.

For a concrete implementation, I build on the platform described in chapter 3 and built with a large team within Invasive Computing. The project yields an FPGA-based implementation of the whole system, which makes the evaluation more authoritative than a software simulation.

While a formal model and a practical implementation is available, there is still a wide gap between them. To close this gap, we require formal semantics for the implementation, especially the programming language X10 we used. Lee and Palsberg [LP10] made a first step to formalize X10 concurrency, but they did not tackle the distributed aspects (the at keyword). I took another step by defining a memory consistency model in chapter 4. This is an important step for the development of our X10 compiler, since the semantics of X10 were not specified in full detail before.

Chapter 5 describes the design and implementation of my resource-aware framework. While chapter 2 shows the essential mechanisms, constraints and hints must also be useable in practice.

With an implementation of the framework, I describe two example applications built on top in detail. Both are from the field of High Performance Computing, but exhibit different behavior. The first one, "Multigrid" in chapter 6, is a heat dissipation simulation with periodically changing resource needs. The second one, "Integrate" in chapter 7, is numeric integration with unforeseeable and dynamically changing resource needs. At this point, I prove the feasibility of my approach through the implementation and execution of realistic applications with manageable effort.

At this point, the theory and practice of resource-aware programming is laid out and we can take a step back to evaluate the full system, where multiple applications are interacting with each other. Chapter 8 investigates the speedup, which is theoretically unbounded, since the dynamicity enabled by the framework changes the game. For additional evidence, this chapter looks into the situation when application compete for resources. This proves my claim of improved performance.

This thesis concludes in chapter 9 with a short summary. The chapter also documents the open ends and conjectures for future work and ponders the impact and wider applicability of this work.

1.8. Contributions

The previous section described the linear flow of this dissertation and the major contributions were already highlighted in the abstract. Here, I present a complete list of the contributions in this dissertation.

1. A formal model of resource allocation in chapter 2 for a concise description of the concepts.

2. Proofs of improvements of efficiency (section 2.7), utilization (section 2.8), and speedup (section 2.9) based on the formal model.

3. The system is implemented and executable on a Linux host system and on custom hardware, see chapter 3.

4. A memory model for X10 in chapter 4 to fill a specification gap, which is particularly embarrassing for a language designed for concurrency.

5. An innovative language for resource-aware programming in the high-level programming language X10 in chapter 5. This new paradigm of resource awareness gives the programmer much more control about resource use and more information to the system for optimization.
6. A compilation path for the proposed language is described in section 5.11.
7. Evaluation of our implementation with High Performance Computing applications like heat simulation with a multigrid solver (chapter 6), distributed numerical integration (chapter 7).
8. How performance for a system as a whole can improve is measured in chapter 8.

Allocation Model

All models are wrong, but some are useful.
— George E. P. Box

For a clear view of invasive computing, unobstructed by technicalities, this chapter presents a basic formal model of resource allocation, which describes the partitioning of resources over time. It demonstrates the essence of the resource-aware paradigm. Intuitively the approach is as follows: Given an execution trace of an invasive program, we look at specific actions, like invade or retreat, and specify if the trace is valid. Based on the model, this chapter then provides guarantees for efficiency, utilization, and speedup improvements.

This chapter proposes a formal model for judging the validity of traces instead of defining a constructive way to derive traces. The advantage is that this model is independent of the allocation strategy of the resource management component. The allocation strategies used in practice are usually heuristics as the search space of possible allocations is huge. Allocation decisions also depend on previous decisions, which makes the trace hard to predict and could show chaotic behavior. Formalizing such strategies has little benefit, since there are many possible variations and small changes can compound to large differences. Thus, we abstract over the allocation strategy and focus on the validity of execution traces.

2.1. Resources and Claims

Let the finite set of all system resources be $\mathcal{R}$. We do not care about the actual contents. For intuition, consider it a set of processing elements (cores of a CPU). It might also include memory, bandwidth, or communication channels. For example, you could partition the memory into chunks (e.g. each page) and use that as resources. On our clustered architecture the network-on-chip can reserve channels, so you could use those as a resource. In a real-time system, we could use chunks of a timing period as a resource.

Resources are accounted for in a "claim" tuple $\kappa = \langle R, c \rangle$, where $R \subseteq \mathcal{R}$ is the set of resources in use, and c are constraints for the resources of the claim R. For now, ignore c. We define it later in section 2.3. The intuition behind the term "claim" is that applications "stake their claims" of resources like gold diggers, who have their exclusive patch of land.

Let the list of all claims in a system at some point in time be $\mathcal{K}$; For example $[\kappa_0, \kappa_1, \kappa_2]$. We will use set operations on these lists, because they work intuitively, but we use the indices of the list elements to assign them an identity. For example, $\kappa_2 \in \mathcal{K}$ is the third claim of the list. All resources are in a claim at any time, thus the invariant:

$$\mathcal{R} = \bigcup_{\langle R,c \rangle \in \mathcal{K}} R$$

Initially, there is $\kappa_0 = \langle \mathcal{R}, c_0 \rangle$ holding all resources of the system. There is no execution corresponding to this "idle claim". It serves as a background buffer of resources and is the only claim, which may be contain no resources. If other claims are empty, we prune them from $\mathcal{K}$.

Resources are also *exclusive* for each claim, thus another invariant:

$$\forall \kappa_a, \kappa_b \in \mathcal{K} : \kappa_a = \kappa_b \ \vee \ R_a \cap R_b = \emptyset \qquad \text{where } \kappa_a = \langle R_a, c_a \rangle \text{ and } \kappa_b = \langle R_b, c_b \rangle$$

The partitioning of all resources of a system to an arbitrary number of claims is a "distribution". A running invasive system determines a trace of resource distributions, which models its behavior with respect to resource allocation. Such a trace is potentially infinite and we assume a globally consistent view and order. The transition from one distribution to the next distribution in a

trace is a "redistribution" δ. Conventionally, the initial redistribution δ_0 creates a main claim κ_1 for the main application with minimal resources (e.g. one processing element).

Let $\mathbb{K}$ be the set of all possible distributions for an implicit set of resources $\mathcal{R}$. Since $\mathcal{R}$ is finite, so is the partitioning of resources. By definition, $\mathcal{K} \in \mathbb{K}$. Formally, a "redistribution" δ is a mapping from one list of of claims to another: $\delta = (\mathcal{K}, \mathcal{K}') \in \mathbb{K}^2$. A redistribution is where the implementation performs the actual work of managing resources.

What happens at a redistribution is one of the following cases. In each case, let $\delta = (\mathcal{K}, \mathcal{K}')$, $\kappa_i \in \mathcal{K} = \langle R, c \rangle$, and $\kappa'_i \in \mathcal{K}' = \langle R', c' \rangle$:

1. A new claim is created and given some initial resources R from other claims ("invade").

   $|\mathcal{K}| + 1 = |\mathcal{K}'|$ and for every κ_i there is κ'_i such that $c = c'$.

2. Resources can be moved between the resource sets of claims ("reinvade").

   $|\mathcal{K}| = |\mathcal{K}'|$ and for every κ_i there is κ'_i such that $c = c'$.

3. Like the previous case, but additionally the constraints of one claim κ^* are changed.

   $|\mathcal{K}| = |\mathcal{K}'|$ and there is exactly one $\kappa^* \in \mathcal{K}$, such that for each κ_i there is κ'_i where either $c = c'$ or $\kappa = \kappa^*$.

4. A claim κ^* is destroyed and all its resources moved to other claims ("retreat").

   $|\mathcal{K}| - 1 = |\mathcal{K}'|$ and for every κ'_i there is κ_i such that $c = c'$.

The special case of δ being the identity function is allowed. This is why in the two reinvade cases resources *can* move, but are not required.

Now we formally describe a "trace" as a starting state $\mathcal{K}_0 = [\langle \mathcal{R}, c_0 \rangle]$ and a sequence of redistributions $\delta_0, \delta_1, \ldots$. The starting state specifies all resources in the system $\mathcal{R}$. Constraints are described below, but the intention of c_0 is that any claim could take those resources.

The sequence if redistributions is implied by the redistributions: For all redistributions δ_n and δ_{n+1} of the same trace, let δ_n be $(\mathcal{K}_0, \mathcal{K}_1)$ and δ_{n+1} be $(\mathcal{K}_2, \mathcal{K}_3)$, then $\mathcal{K}_1 = \mathcal{K}_2$. The sequence implies that we could assign a time to each

redistribution from a trace. So it makes sense to say a redistribution "happens at a certain point in time". In the same sense, we can say something "happens at a certain redistribution".

2.2. Actor Claims

For each redistribution $\delta = (\mathcal{K}, \mathcal{K}')$ there is a subset $\mathbb{A}_\delta \subseteq \mathcal{K}$ that are "actor claims". Only those claims can change or disappear. Claims that appear are by definition not in $\mathcal{K}$, so there is no need to formally define them as actors. This set can be larger than the set of actually changed claims in a redistribution, thus we need a definition for identifying actor claims. There are three possible reasons for a claim to be an actor:

- The idle claim κ_0 is always an actor claim.
- Claims which are "asynchronously-malleable" are always actor claims. See chapter 7 for more details about malleability.
- There can be (at most) one actor claim induced by the program code corresponding to the trace, when an application initiates a redistribution with respect to a specific claim. It might not exist, if e.g. resource management redistributes periodically.

Often there are only two actor claims, namely κ_0 and the one which explicitly changes according to the program code involved in the redistribution.

To clarify, we distinguish between program and application. An "application" always corresponds to a single claim. A "program" consists of one or more applications and corresponds to an executable. For example, a robot can be controlled by a single program, which includes different applications for motion control, speech recognition, etc. This distinction implies that application can share code and there is no bijective mapping between code and applications.

One requirement for traces is that only actor claims change. This implies all non-actor claims stay the same. Formally,

$$\forall \delta = (\mathcal{K}, \mathcal{K}') : \forall \kappa_i \in \mathcal{K} \setminus \mathbb{A}_\delta : \quad \kappa_i = \kappa_i'.$$

Let an overline $\overline{\kappa}$ mark actor claims within claim sets with respect to a followup redistribution. For example, in $[\overline{\kappa_0}, \kappa_1, \overline{\kappa_2}]$ the first and the third claims are actor claims in the following redistribution.

The most simple invasive application acquires some resources, uses them, and finally frees them. This requires two redistributions. In the following example, there is an additional redistribution before and after, which represent a generic setup. The presence of three claims displays the role of $\mathbb{A}$. We start with an initial state $\mathcal{K}_0$, where all resources belong to the idle claim κ_0.

1. Initially
$$\mathcal{K}_0 = [\overline{\kappa_0}] = [\langle \{p0, p1, p2, p3, p4\}, c_0 \rangle]$$

2. $\delta_0 = (\mathcal{K}_0, \mathcal{K}_1)$ creates the main claim κ_1, which contains the processing element $p0$, which executes the simple invasive application.
$$\mathcal{K}_1 = [\overline{\kappa_0}, \kappa_1] = [\langle \{p1, p2, p3, p4\}, c_0 \rangle, \langle \{p0\}, c_1 \rangle]$$

3. $\delta_1 = (\mathcal{K}_1, \mathcal{K}_2)$ invades resources and thus creates another claim κ_2 with resources from κ_0. We observe that $p0$ is already claimed by non-actor κ_1 and not available.
$$\delta_1(\mathcal{K}_1) = \mathcal{K}_2 = [\overline{\kappa_0}, \kappa_1, \overline{\kappa_2}] = [\langle \{p3, p4\}, c_0 \rangle, \langle \{p0\}, c_1 \rangle, \langle \{p1, p2\}, c_2 \rangle]$$

4. $\delta_2 = (\mathcal{K}_2, \mathcal{K}_3)$ completely retreats κ_2, thus moves all its resources back to κ_0.
$$\delta_2(\mathcal{K}_2) = \mathcal{K}_3 = [\overline{\kappa_0}, \overline{\kappa_1}] = [\langle \{p1, p2, p3, p4\}, c_0 \rangle, \langle \{p0\}, c_1 \rangle]$$

5. $\delta_3 = (\mathcal{K}_3, \mathcal{K}_4)$ completely retreats κ_1, thus moves all its resources back to κ_0.
$$\delta_3(\mathcal{K}_3) = \mathcal{K}_4 = [\overline{\kappa_0}] = [\langle \{p0, p1, p2, p3, p4\}, c_0 \rangle] = \mathcal{K}_0$$

2.3. Constraints

For the programmer developing an application in a resource-aware paradigm, the challenge is to specify constraints. This is the mechanism for an application to communicate about resource needs to the global resource management.

We usually deal with subsets of $\mathcal{R}$, which naturally is in $\mathfrak{R} = 2^{\mathcal{R}}$, the powerset of $\mathcal{R}$.

The constraints c of a claim are a predicate $\mathfrak{R} \mapsto \{\bot, \top\}$, where $\mathfrak{R}$ is the resources domain. The output says whether the resources satisfy the constraints ($\top$) or does not ($\bot$). Constraints of a claim are guaranteed, thus the invariant:

$$\forall \langle R, c \rangle : c(R) = \top$$

For programmer convenience, we want c to be composable from simple building blocks. Since constraints are predicates, $\wedge$ and $\vee$ are obvious combinators.

It might be unintuitive that all constraints always apply for the full set of resources, but this is necessary for composition. For an example, let us assume you want to specify the request "two fast PEs and five slow ones". You might try $(\text{isFast} \wedge \text{count}(2)) \wedge (\text{isSlow} \wedge \text{count}(5))$, which is actually a contradiction. The set of resources cannot be fast and slow at same time. Neither can it contain exactly 2 and 5 PEs at the same time. You would have to use a predicate like "contains that many fast PEs". This is not composable, since fast and count are mixed into the same predicate.

For generality, we should extend constraints to take the full resource partitioning into account. We might want to express constraints like TileSharing (see chapter 5), which affects resources outside of its claim. Tile sharing means that for a resource p_a in a claim κ_a and another resource p_b in another claim κ_b, both resources can be on the same tile. By default TileSharing is forbidden to err on the side of isolation. To express that constraints must consider the resource sets of both claims. We omit this generalisation here, though, because it is not necessary for the following sections and would only complicate the equations.

2.4. Validity

A trace is "valid" if the three invariants of claims and the requirement for redistributions hold. To summarize, the four conditions of valid traces are:

1. $\forall \mathcal{K} : \mathcal{R} = \bigcup_{\langle R,c \rangle \in \mathcal{K}} R$
2. $\forall \kappa_a, \kappa_b \in \mathcal{K} : \kappa_a = \kappa_b \ \vee \ R_a \cap R_b = \emptyset$,
   where $\kappa_a = \langle R_a, c_a \rangle$ and $\kappa_b = \langle R_b, c_b \rangle$

3. $\forall \delta = (\mathcal{K}, \mathcal{K}') : \forall \kappa_i \in \mathcal{K} \setminus \mathbb{A}_\delta : \quad \kappa_i = \kappa_i'$

4. $\forall \langle R, c \rangle : c(R) = \top$

Let us consider a few examples. We use $n \ldots m$ as a shortcut constraint notation, which means the number of resources is in the range $[n, m)$. More precisely, $n \ldots m(R) = \top$ iff $n \leq |R| < m$ and $\bot$ otherwise.

Here is the first claim list example, which is valid by itself.

$$\left[\quad \langle \{p1, p2, p3, p4\}, 0 \ldots 999 \rangle, \quad \langle \{p0\}, 1 \ldots 3 \rangle \quad \right]$$

That example would be invalid, if $\mathcal{R} \neq \{p0, p1, p2, p3, p4\}$ though, because that would violate condition 1 above.

We can see the exclusiveness invariant condition 2 violated. In the following example, $p1$ is contained in two claims:

$$\left[\quad \langle \{p1, p2, p3, p4\}, 0 \ldots 999 \rangle, \quad \langle \{p0, p1\}, 1 \ldots 3 \rangle \quad \right]$$

To violate condition 3, we need two distributions and a redistribution, such that $\delta = (\mathcal{K}_A, \mathcal{K}_B)$. While it is allowed that an actor claim remains unchanged, a non-actor claim must not change. Thus, the addition of $p4$ to the non-actor claim of $\mathcal{K}_A$ below is a violation.

$$\mathcal{K}_A = \left[\quad \overline{\kappa_0}, \overline{\langle \{p0, p1\}, 1 \ldots 3 \rangle}, \langle \{p2, p3\}, 1 \ldots 4 \rangle \quad \right]$$
$$\mathcal{K}_B = \left[\quad \kappa_0', \langle \{p0, p1\}, 1 \ldots 3 \rangle, \langle \{p2, p3, p4\}, 1 \ldots 4 \rangle \quad \right]$$

Finally, an example which violates condition 4, as the constraints of the second claim are not fulfilled: $2 \ldots 3(\{p0\}) = \bot$.

$$\left[\quad \langle \{p1, p2, p3, p4\}, 0 \ldots 999 \rangle, \quad \langle \{p0\}, 2 \ldots 3 \rangle \quad \right]$$

2.5. Hints

The model only distinguishes valid and invalid behavior, but does not consider performance. Additional "hints" are useful to improve performance, but do not affect the validity of a trace. For example, applications can communicate information about scalability to resource management. Now, if a resource can be allocated to two applications, the scalability hint can be used as a tie breaker. The hint reveals which application would profit more from the resource, so we optimize for the throughput of the system as a whole.

For this section, we extend a claim with hints in addition to resources and constraints. So, $\kappa = \langle R, c, h \rangle \in \mathcal{K}$. Similar to the constraints, a hint h is a function $\mathfrak{R} \mapsto \mathbb{N}$. Where a constraint maps to $\bot$, a hint maps to 0. Where a constraint maps to $\top$, a hint maps to 1 or more. The higher the number the better for the application, although we are intentionally imprecise what "better" means. It might model performance, energy consumption, etc.

Now we can express the optimization goal for resource management: Maximize

$$\sum_{\langle R,c,h \rangle \in \mathcal{K}} h(R).$$

Like constraints, hints can be composed from multiple independent hints. One hint could be about parallelism and scalability and another about sharing with other applications. Since we might want to compose these different aspects, let them be in a finite set G. For composition we multiply the various hint function results.

While G describes a single configuration, we might have fundamentally different configurations for resources due to heterogenity, where the aspects must be rated very differently. There might be complex dependencies, like "with such resources it scales like this, but with other resources it scales like that". This implies we have different algorithms for each configuration and we assume we always choose the best algorithm for some given resource set. To model this, we employ the max operation over all G.

Overall, we model this with a set H of configurations. The composition structure corresponds to constraints composed in disjunctive normal form. For a claim $\kappa = \langle R, c, h \rangle$, the function h is derived from H as follows:

$$h(R) = \max_{G \in H} \left(\prod_{f \in G} f(R) \right)$$

To compose hints a commutative operation is desirable, since order should not be relevant. A declarative constraint description avoids mistakes. This motivates the use of $\prod$ in the formula. A sum would have been another option, but the product means that each factor can set the result to zero, for example if the wrong kind of resources are available.

Naturally, there might be multiple resource allocations resulting in the same optimal evaluation. However, we do not even require resource management to be optimal. In practice, resource management will use heuristics and distribute resources suboptimally, because it must do so online under time pressure.

2.5.1. Hint Example

For an example, let us assume an application with two algorithms to choose from. One is for fast processing elements, but only scales to 5 of them. The other one is for slow processing elements, but needs at least 8. Both scale linearly.

$$\begin{aligned}
f_{\text{scale}_1}(R) &= \min(15, 3|R|) \\
f_{\text{scale}_2}(R) &= |R| && \text{if } |R| \geq 8 \text{ else } 0 \\
f_{\text{fast}}(R) &= 1 && \text{if all cores in } R \text{ are fast else } 0 \\
f_{\text{slow}}(R) &= 1 && \text{if all cores in } R \text{ are slow else } 0 \\
H &= \{\{f_{\text{scale}_1}, f_{\text{fast}}\}, \{f_{\text{scale}_2}, f_{\text{slow}}\}\}
\end{aligned}$$

The fast processing elements are three times as fast as the slow ones. So 3 fast ones are as good as 9 slow ones in our optimization problem. Look at a few example evaluations. Let F_i be a fast core and S_i a slow one.

$$\begin{aligned}
h(\{F_0, F_1, S_0, S_1\}) &= \max(\{(3 \times 4) \times 0,\ 4 \times 0\}) = 0 \\
h(\{F_0, F_1, F_2\}) &= \max(\{(3 \times 3) \times 1,\ 3 \times 0\}) = 9 \\
h(\{S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8\}) &= \max(\{(3 \times 9) \times 0,\ 9 \times 1\}) = 9
\end{aligned}$$

2.5.2. Scaling Hints

Let us devise a constraint to model linear scaling up to a certain number n of resources:

$$h^n(R) = \min(n, |R|)$$

For example, $h^5(\{a, b, c\}) = 3$, but $h^2(\{a, b, c\}) = 2$. Assume 6 resources $p0, ..., p5$ overall and an initial redistribution δ_0, which creates κ_1. The constraints will always be adapted to allow a resource amount between 0 and n using the notation from above: $0 \dots n$.

Technically, we should define H for h^n and derive a function according to the formula above. However, they are both hint functions $\mathfrak{R} \mapsto \mathbb{N}$ and the function derived from $H = \{\{h^n\}\}$ would be equivalent to h^n. Thus, we use the shortcut and put h^n into claims directly.

Here is an example trace, which shows how resource management obeys hints and invariants like the actor claim criterium (condition 3 above).

δ_0 where $\overline{\kappa_1}$ gets 4 resources due to its hints h^4.

$$[\overline{\kappa_0}, \kappa_1] = [\langle \{p4, p5\}, 0 \dots 9, h^0 \rangle, \langle \{p0, p1, p2, p3\}, 1 \dots 5, h^4 \rangle]$$

δ_1 where another claim $\overline{\kappa_2}$ appears with a h^4 hint as well. However $\kappa_1 \notin \mathbb{A}$, so its resources stay the same.

$$[\overline{\kappa_0}, \overline{\kappa_1}, \overline{\kappa_2}] = [\langle \{\}, 0 \dots 9, h^0 \rangle, \langle \{p0, p1, p2, p3\}, 1 \dots 5, h^4 \rangle, \\ \langle \{p4, p5\}, 1 \dots 4, h^4 \rangle]$$

δ_2 where claim κ_1 changes its hints to h^1 and thus retreats all resources except $p1$. Let $\kappa_2 \in \mathbb{A}$, so it claims $p0$ and $p3$, but $p2$ goes to the idle claim κ_0.

$$[\kappa_0, \kappa_1, \kappa_2] = [\langle \{p2\}, 0 \dots 9, h^0 \rangle, \langle \{p1\}, 1 \dots 2, h^1 \rangle, \\ \langle \{p0, p3, p4, p5\}, 1 \dots 4, h^4 \rangle]$$

2.6. Shortcuts and Implementation of Resource Management

This chapter formalizes the behavior of resource management, but an implementation thereof must match only observable behavior. In this thesis, we do not investigate *how* resource management finds good allocations apart from functional correctness. This section ponders some variations, which can be beneficial in practice.

It would not be efficient to use c and h as black box functions in a mathematical optimization problem. For example, computing the choices for max is easy to parallelize, which is not possible with a black box approach.

Resource management should usually not consider all possible partitionings. Only resources of the actor claims $\mathbb{A}$ can be adapted, so a heuristic can ignore the rest. This sacrifices optimality, though. If the algorithm knows about the constraints and hints of non-actor claims, they can be used to reserve resources for them. Without that information an optimal solution could be impossible. It might be worthwhile to give current actor claims less resources and spare them for non-actor claims which scale better. In practice, often $\mathbb{A}$ consists of only one claim plus the idle claim, which makes idle claim the single point of resource exchange. Our implementation has a mechanism to reserve resources even in non-actor claims, which means in their next reinvade, the claims can drop reserved resources if possible.

The model defines the idle claim κ_0. An implementation might use a special non-claim resource pool, where resources can be taken from with less overhead. Such a pool can always give resources away. It does not need constraints and hints, which means there is no need to check them, which means faster allocations. Still, the current implementation uses an idle claim, because special resource pools require special cases in the code, which increases the error potential.

The model assumes the resources of a claim are fully used by the application. However, we can maintain two sets of resources per claim, the ones used by the application and unused resources. This is a way to decentralize the idle claim κ_0. Reinvasion can be triggered concurrently by multiple applications, thus we need to synchronize resource management. Especially κ_0 is under contention if it serves as the single point of resource exchange. Decentralization lowers contention. Sometimes this might even allow to perform reinvades without considering other claims at all. Less synchronization between claims

can improve performance at the cost of potentially worse heuristic decisions. This argument gains even more weight, if a distributed resource management system [KBL+11] is used, where synchronization implies communication and is even more costly.

2.7. Proof: Efficiency is not Worse

The distinctive feature of resource-aware programming is that application can reconfigure their exclusive resource use *during* run time. Conventional solutions, like those used on supercomputers today, only configure the resources at the start of a run. This implies some fragmentation of the resource allocation, so some resource will usually idle. We call this "waste" and try to give some bounds for waste in the following.

For the following proofs we assume only uniform processing elements as resources. This simplifies the notation, when we consider scalability, because then $|R|$ is the hardware parallelism. Otherwise we would have to filter for processing elements every once in a while.

In parallel computing, the formula for "speedup" is the time for serial processing divided by the time for parallel processing $S = T_1/T_p$, where p is the degree of parallelism. The quotient can be less than 1, as T_1 measures the fastest possible serial implementation. Nevertheless, the speedup initially grows from $p = 2$ onwards. This implies that adding parallelism increases the speed, but usually with each added processing element the increase is smaller. At some point speedup plateaus and decreases, when the overhead of synchronization becomes too large. To account for that we measure the "efficiency" E by calculating the speedup attained per PE:

$$E = \frac{S}{p} = \frac{\frac{T_1}{T_p}}{p} = \frac{T_1}{T_p \times p}$$

For the theoretic speedup/efficiency, we can compute the limit $p \to \infty$. However, here we deal with real applications and concrete traces, so there is no need to talk about ∞.

To define an "efficiency metric" for traces, we need to extend them with a notion of time. Thus, for each redistribution δ_n, let t_n be the time when it happens. We do not care if the unit is seconds, milliseconds, or cycles, but

assume positive integers. The start t_0 is zero. Redistributions usually change the parallelism, so we must take that into account when computing efficiency. Assume an application whose claim is created in δ_i and destroyed in δ_j. Of course, $i < j$ and there are $j - i$ intervals to consider. Let $\langle R_n, c_n \rangle$ be the claim of the application after δ_n. The time T_p equals $t_j - t_i$. We define the parallelism as the mean parallelism

$$p = \frac{\sum_{k=i}^{j-1}(|R_k| \times (t_{k+1} - t_k))}{t_j - t_i}$$

and reuse the efficiency formula above. For the non-resource-aware case, the set of resources is constant, thus $p = |R_k|$.

The formula is close to "Invasive Efficiency" (IE) [TWOSP12], which instead of building on a trace concept uses parallelism profiles. Instead of computing the sum over time intervals, IE computes the sum over all parallelism degrees, which makes it unsuitable for the proofs below, but applicable for empiric evaluations. IE additionally models overhead, which is absent in the more abstract allocation model. Another factor in IE is underutilization, which in terms of the model here means that $h(R) < h(R \cup x)$ for some $x \subset \mathcal{R}$. Intuitively, the application could run even faster with some more resources according to the hints.

Definition 1. Claim Efficiency: The efficiency E of a claim, which exists from δ_i to δ_j and contains the resources R_i to R_j respectively, is

$$E = \frac{T_1}{T_p \times p} = \frac{T_1}{(t_j - t_i) \times \frac{\sum_{k=i}^{j-1}(|R_k| \times (t_{k+1} - t_k))}{t_j - t_i}} = \frac{T_1}{\sum_{k=i}^{j-1}(|R_k| \times (t_{k+1} - t_k))}$$

A non-resource-aware application is characterized by being unable to change its resource allocation while it runs. In other words, over its lifetime from δ_i to δ_j the respective resource sets R_i to R_j are all equal. To convert a resource-aware application into a non-resource-aware equivalent, we can start by allocating the maximum number of resource we could exploit and keep them until the end. For a better comparison and without loss of generality, we insert dummy reinvades, where δ is the identity function, so both have the same number of corresponding redistributions.

Theorem 1. *Given a resource-aware application's efficiency E and its non-resource-aware equivalent's efficiency E', both aiming for minimal execution time, $E \geq E'$.*

Proof. Optimizing for minimum execution time implies $t_{k+1} - t_k = t'_{k+1} - t'_k$, as the resource-aware application would use more resources if that would speed it up. It also implies that non-resource-aware uses the maximum number of resources, thus $\forall k \in \{i, ..., j\} : |R_k| \leq |R'_k|$.

$$E = \frac{T_1}{\sum_{k=i}^{j-1}(|R_k| \times (t_{k+1} - t_k))} \geq \frac{T_1}{\sum_{k=i}^{j-1}(|R'_k| \times (t_{k+1} - t_k))} = E'$$

□

This proof builds on the assumption that nothing restricts the resource allocation of the resource-aware application. However, in a competitive scenario resources might be allocated elsewhere. Then the non-resource-aware might be restricted to a suboptimal amount of resources, which improves its efficiency assuming sublinear speedup. Efficiency is inherently a tradeoff with speedup [EZL89]. In some sense, you "pay" efficiency to attain speedup. This means that efficiency is usually not the paramount optimization goal, because then nobody would use parallel algorithms, except the few which scale linearly.

There is a major assumptions in this proof, which does not hold in reality. We assume that there is no overhead for resource allocation. You will see in chapter 6 that there can be significant overhead for resource management and also for the application to adapt to resource changes.

Now that we know that efficiency is guaranteed to not be worse for a single resource-aware application, let us consider a multi-application scenario. There the goal is not to optimize a single application, but to improve the performance of the system as a whole. Thus, we have to look at scenarios with at least two applications.

2.8. Proof: Utilization Improves

Another useful metric is utilization. We want a definition for the system as a whole. One aspect is that resources in the idle claim are not utilized, so we could calculate the percentage of non-idle-claim resources. However, we also want to reason about non-resource-aware applications, where resources are

unused although the are not in the idle claim. Let these resource be R_{idle} the "hidden idle resources". Also, let $u = |\mathcal{R}| - |R_0| - |R_{\text{idle}}|$ be the resources in use.

Definition 2. Resource Utilization: The resource utilization U of claims is the percentage of resources in use

$$U = \frac{u}{|\mathcal{R}|} = \frac{|\mathcal{R}| - |R_0| - |R_{\text{idle}}|}{|\mathcal{R}|} = 1 - \frac{|R_0| + |R_{\text{idle}}|}{|\mathcal{R}|}$$

Resource-aware applications are never under-utilized, since they would retreat from resources instead, so $R_{\text{idle}} = \emptyset$. Nevertheless, the system has a whole may be under-utilized, if not all resource can be exploited.

It might seem intuitive that utilization is higher in the resource-aware case, because non-resource-aware application can be crippled for life due to lack of resources at the start, while a resource-aware application can overcome bad starting conditions and gain more resource later. The problem is that application interact and work can be delayed. These delays can lead to a higher utilization temporarily in the non-resource-aware case as illustrated in fig. 2.1. While the intuition is indeed correct, it complicates the proof below as we must account for delayed work.

For quantification, we need a measure of work an application does in a certain time slice from t_n to t_{n+1}. For an implicit application let "work done" for the time intervals 0 to e be

$$w_e = \sum_{n=1}^{e} |R_n| \times (t_n - t_{n-1}).$$

Let "delayed work" d_n for the corresponding non-resource-aware case be the difference of work: $d_n = w_n - w'_n$. They differ in resources R used, but require the same time.

For the non-resource-aware applications the resource partitioning is arbitrary, as it depends on the system state when the applications started. For the resource-aware applications, we assume an optimal allocation with respect to their hints, because resource management can reallocate as necessary. Thus, a resource-aware application may get a sub-optimal amount of resources if other applications can make better use of them, since this is the optimization goal for resource management. So, d_n may be negative for specific applications.

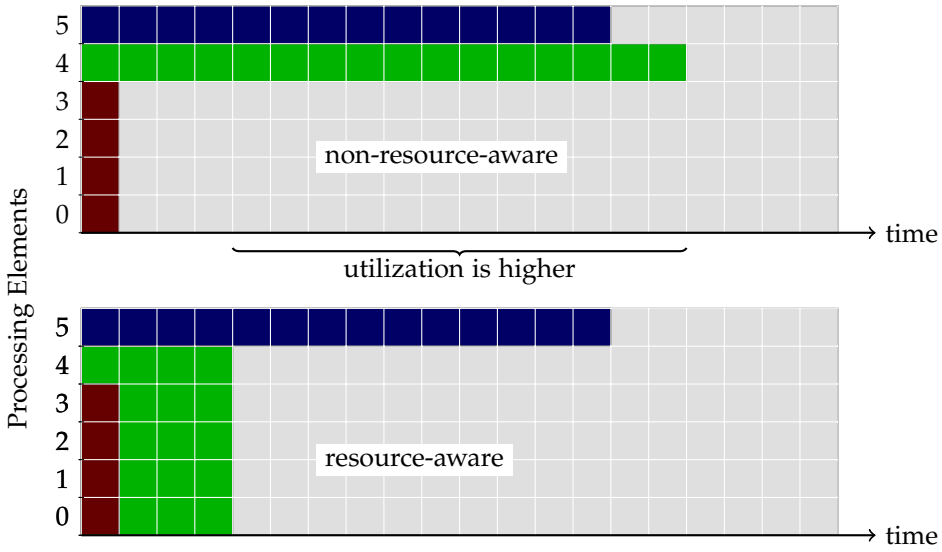


Figure 2.1.: An example, where utilization is temporarily higher in the non-resource-aware case. If the green application is resource-aware, it uses uses the resources freed after the red application terminates and finishes faster. This implies green runs longer in the non-resource-aware case and utilization is higher until it finishes.

Since resource management maximises over hints, which specify utility of resources, it also maximises work done and minimizes delayed work. In contrast, non-resource-aware allocation can be arbitrary because only the situation right at the start is considered and there is no opportunity to correct later. The implications are more delayed work.

Lemma 1. *A resource-aware system has less delayed work than a corresponding non-resource aware one:* $\sum_{n=0}^{|\mathcal{K}|} d_n \geq 0$

Proof. For simplification, we insert identity redistributions such that resource-aware and non-resource-aware always have corresponding events, so $t_i = t'_i$.

Assume at time t_i a resource x which a non-resource-aware application uses $x \in R'$, but the corresponding resource-aware application does not $x \notin R$.

If x is used by another application, then x is neutral with respect to work done.

On the other hand, if $x \in R_0$ and we assume proper resource management, then x is not useful to the resource-aware application. This implies that the non-resource-aware application uses it do get *delayed* work done. Thus, $w_i \geq w'_i$.

In both cases the resource-aware application cannot delay more work than the non-resource-aware one. □

Lemma 1 implies that the makespan for resource-aware applications is at most as high as for a corresponding non-resource-aware scenario, because the non-resource-aware one can still have delayed work, but not vice versa. This fact is the intuitive reason for higher utilization, but we still need to connect that to the formula of U.

For a proof of utilization, we can not focus on utilization at specific times U_n, because we know it can vary. Instead, we care for the utilization over the whole run time of the system until t_e, which we denote as

$$\int U = \sum_{n=0}^{e} U_n \times (t_{n+1} - t_n).$$

Theorem 2. *Let multiple resource-aware applications running in parallel have utilization U, while the non-resource-aware equivalents run with utilization U'. Then $\int U \geq \int U'$.*

Proof. To simplify the formulas, we insert identity redistributions such that the intervals $t_{n+1} - t_n$ are constant. Thus, without loss of generality, let $t_{n+1} - t_n = 1$ and

$$\int U = \sum_{n=0}^{e} U_n.$$

Because the corresponding applications perform the same work on a homogeneous set of resources, we know

$$\sum_{n=0}^{e} u_n = \sum_{n=0}^{e'} u'_n.$$

The sum $\sum u_n$ corresponds to $\int U$ by a constant factor $e|\mathcal{R}|$, because

$$\sum_{n=0}^{e} u_n = \sum_{n=0}^{e} u'_n \frac{|\mathcal{R}|}{|\mathcal{R}|} = e|\mathcal{R}| \times \sum_{n=0}^{e} \frac{u'_n}{|\mathcal{R}|} = e|\mathcal{R}| \times \int U.$$

We know $t_e \leq t'_{e'}$, because of lemma 1. Since we know $t_{n+1} - t_n = 1$, we can as well say $e \leq e'$ and conclude

$$\int U = \frac{\sum_{n=0}^{e} u_n}{e|\mathcal{R}|} = \frac{\sum_{n=0}^{e'} u'_n}{e|\mathcal{R}|} \geq \frac{\sum_{n=0}^{e'} u'_n}{e'|\mathcal{R}|} = \int U'.$$

□

Since utilization improves more work is getting done, so we should see speedups. How much of a speedup is possible?

2.9. Proof: Speedups are Unbounded

Let us assume that conventional applications specify a *fixed exact* number of resources. If they all take half of the PEs plus one, we must run the application sequentially, as illustrated in fig. 2.2. If the number of resources $n \to \infty$, the one additional resource becomes neglible, so 50% of the resources idle.

Let us assume that conventional applications specify a *range* of resources they might use. This is more flexible than a fixed number, but leads to worse results in certain cases, as illustrated in fig. 2.3. A long running conventional application could be very restricted initially. If the blocked resources become available shortly after, a conventional application is unable to exploit those resources. If we increase the run time and the number of resources, the waste can be arbitrarily big.

For a proof, we make an assumption on resource management: Redistributions are optimal with respect to speedups using information in the current redistribution. Specifically, resource management is not an oracle, which can look into the future and predict the run times of applications or when the next redistribution will happen. For real time systems, such information might be available, but we plan for the more general open case, where new applications can appear at any time.

Theorem 3. *For all $k \in \mathbb{N}$ there is a scenario such that the makespan t_m of the resource-aware claim trace M and the makespan t'_m of the non-resource-aware equivalent M' fulfill $kt_m \leq t'_m$.*

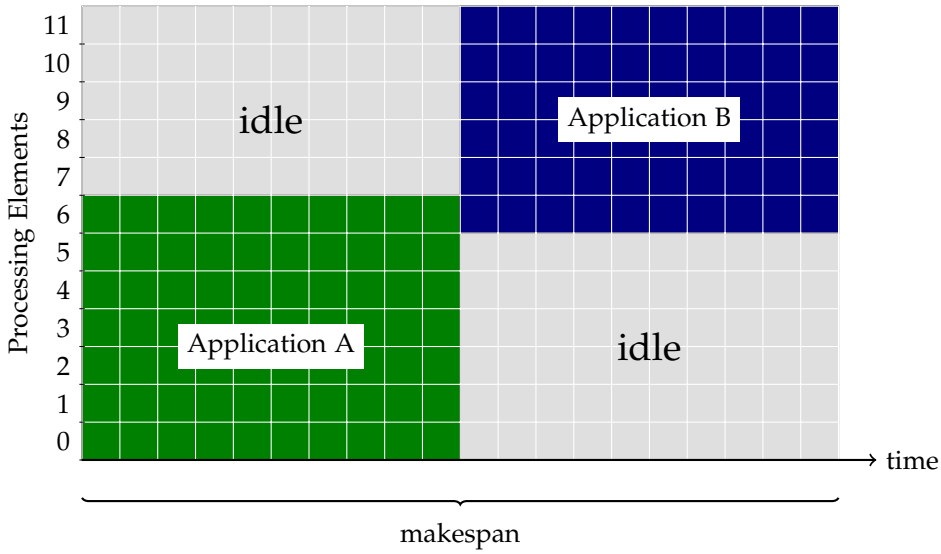


Figure 2.2.: Idling with conventional applications with an exact resource need approximates 50% of the resources in certain cases over the makespan.

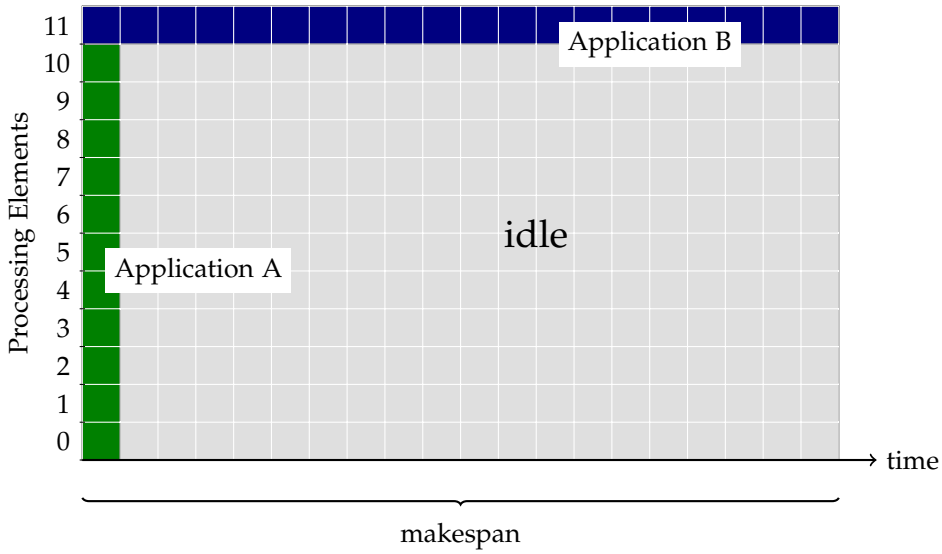


Figure 2.3.: Idling with conventional applications with an resource need range is unbounded. Application A restricts B initially to 1 PE, which is then kept for the whole run time, although A's resource become available very soon.

Proof. Given some $k \in \mathbb{N}$, we design a scenario with $n \geq 3k$ resources as follows: One claim a can use $n-1$ resources and instantly terminates. A second claim b initially takes the remaining resource, but in the resource-aware case could use all resources. Both provide scalability hints. Claim a scales linearly with a speed of $s_a(R) = |R|$, while claim b is worse with $s_b(R) = \frac{1}{2}|R|$. Let the execution time of b be $e_b(R) = \frac{k}{s_b(R)} = \frac{2k}{|R|}$.

Due to the scalability hints, resource management will give $n-1$ resources to a in δ_0 at t_0 and one resource to b in δ_1 at t_1. Then a terminates in δ_2 at t_2. Without loss of generality, let the total execution time of a be $1 = t_2 - t_0$.

In the resource-aware case, b will get all resources in δ_2. Assuming its work until then is negligible, it will finish at $t_3 = t_2 + e_b(\mathcal{R}) = 1 + 2k/n$. In contrast for the non-resource-aware case, b will only use one resource until it finishes at $t'_3 = t_2 + e_b(1) = 1 + 2k$.

$kt_m = k(1 + 2k/n) \leq k(1 + \frac{2n}{3}/n) = \frac{5}{3}k \leq 1 + 2k = t'_m$ □

The proof scenario has an interesting aspect. If we would use one less resource in total, then the non-resource-aware application could finish much faster. The allocation of claim b would fail, because a has all resources. If the program strategy would be to try again later, b gets allocated after a finishes and can use all resources. This is not a practical solution, because we usually do not know when a finishes, though.

At this point, we have a formal allocation model of resource-aware computing and its theoretic advantages, which are efficiency, utilization, and speedups for systems as a whole. Naturally, the model abstracts real world issues away. For example, we did not consider overheads, which might be significant. Nevertheless, the promise is a good foundation for further research.

The next chapters describe the implementation of these ideas and their evaluation with real programs in realistic case studies.

Implementing Invasive Computing

Predicting rain doesn't count, building arks does.
— Warren E. Buffett

To develop a resource-aware application, you need an API or a language to allocate, use, and free resources. We also realized that the runtime system of our programming language X10 required changes, because various parts are not designed with changing resources in mind. Likewise, adapting the operating systems makes sense, as global resource management is its primary job and someone has to provide the API for resource-awareness. Then it turns out that hardware could be adapted to support resource isolation and management for better efficiency. Ultimately, resource aware programming requires changes to the whole computing stack to realize its full potential. Invasive Computing [THH+11] does just that. This chapter describes the relevant parts of the stack and the platform on which this dissertation builds a resource-aware language. While the concepts can be transfered into other domains, the implementation targets the specific platform family which is described in the following sections. We will look at the relevant components in a bottom-up fashion, starting with the hardware, then the operating system, and finally the programming language.

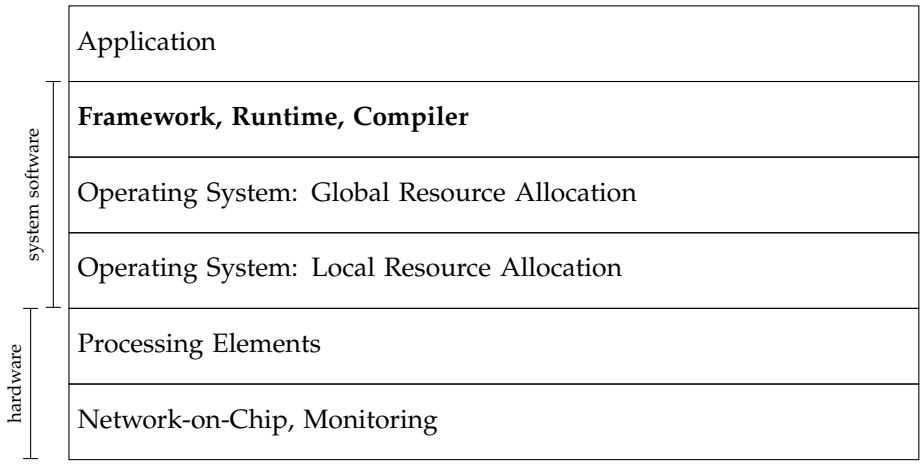


Figure 3.1.: The full Invasive Computing stack. The transregio tackled all levels. This dissertation is part of the (bold) second level.

3.1. The DFG Transregio

In 2010 the CRC/Transregio "Invasive Computing" [THH+11] was funded by the Deutsche Forschungs-Gesellschaft (DFG) and set out to investigate the design and programming of future computer architectures. With architectures of 1000 processor cores on the horizon for 2020, the transregio reconsidered the full stack. From processing elements, network-on-chip, hardware monitoring, accelerators, up to the operating system, compiler, and applications, all components were open for discussion. Figure 3.1 shows a visual overview of the stack. In addition the workflow of development was investigated with simulation and other design-time techniques.

Invasive Computing introduced the three-step operation of invade (allocating resources), infect (using resources), and retreat (freeing resources). Resources usually means processing elements, but also includes memory and communication resources. Those three steps are provided to programmers as explicit operations and enable self-adaptive and resource-aware programming.

Invasive Computing deviates from mainstream architectures in three aspects:

Reimplement Everything! We rewrote or reimplemented applications, instead of supporting legacy code. While it is certainly desirable to build on existing code bases, the dynamicity required for invasive applications would require significant changes, as you will see in the case studies in later chapters. An application which adapts at runtime to resources appearing and disappearing must be able to move its data structures around. In a first step, we required the programmer to do this explicitly. Only when solutions solidified, we moved them into the runtime as a convenience function. This approach provides a clean slate for new ideas.

Global Decisions! The job of current operating systems is usually described as a) abstracting hardware and b) isolating applications. This implies that each application optimizes resources on its own and the operating system maintains the illusion of isolation. In contrast, an invasive application delegates resource decisions to the global operating system. This does not violate the principle of separation of mechanism and policy. Global policy is composed from all applications individual policies.

Another part of invasive philosophy is that applications inform the system about their resource needs, but the system with its global view decides the actual allocation. Thus, we do not optimize single applications, but the system as a whole. This imposes an unusual challenge on API design. Developers only care about the performance of their application, but they should also provide the option for more important applications to take away resources. Our proposed solution is to provide a single method to adapt resources which may give or take resources. See chapter 5 for the concrete API.

Exclusive Resources! Current operating systems are built on the paradigm of scarce hardware resources and thus they virtualize them. Threads are virtual cores for parallelism. Operating systems use virtual addresses for isolating processes from each other and swapping memory to disk temporarily. Invasive Computing envisions a different scenario where there are more cores than we can power. Instead of virtualizing cores, we allocate them exclusively to certain applications. Likewise, we partition memory and communication channels to allocate them exclusively to applications. Isolation between application is guaranteed by the MMU hardware today, but an invasive system can delegate that to the interconnects of a clustered hardware architecture.

The general idea is that resources (like cores and memory) are explicitly requested before use and explicitly freed afterwards. By claiming resources exclusively, some common abstraction layers for virtualizing or security can be removed, which enables efficiency improvements. For example, virtual memory requires a memory management unit (MMU) in hardware. By using a memory-safe high-level programming language, the need for isolation disappears. Since we also expect more heterogeneous hardware in the future, applications have to access it directly for efficiency reasons anyways. Empiric results were shown for HPC applications like parallel sorting [SSF13] and for robotics like motion planning [KGVA16].

3.2. Invasive Hardware Architecture

To support thousands of cores on a single chip, global cache coherence via bus snooping is considered unfeasible. In contrast to bus snooping, a directory-based approach increases latency. Instead, we use a clustered architecture. The cores are grouped into tiles, such that within a tile a limited number of cores can use efficient bus-snooping for cache coherence. Between tiles we abandon cache coherence for a more flexible network on chip. In fig. 3.2 you can see an example architecture.

Although there is no cache coherence, all memory is accessible to every core. It is possible to implement "cache coherence" or rather memory consistency in software across tiles, but we consider this too error-prone for normal programming. Still, Manuel Mohr exploited this to speed up message passing [MT17] and the operating system does it internally. For X10 programmers, the type system prevents access to remote memory.

The invasive network-on-chip *i*NoC provides a mesh communication scheme between tiles and mechanisms to reserve guaranteed service (GS) connections. These GS connections can guarantee a certain latency or throughput, which enables predictability for concurrent applications. GS connections can be mixed with best effort connections, which keeps the system flexible enough to support mixtures of different applications. A resource-aware application should consider its communication needs and whether it benefits from latency or throughput guarantees. Even without guarantees, resource management can use knowledge about communication behavior to improve allocation.

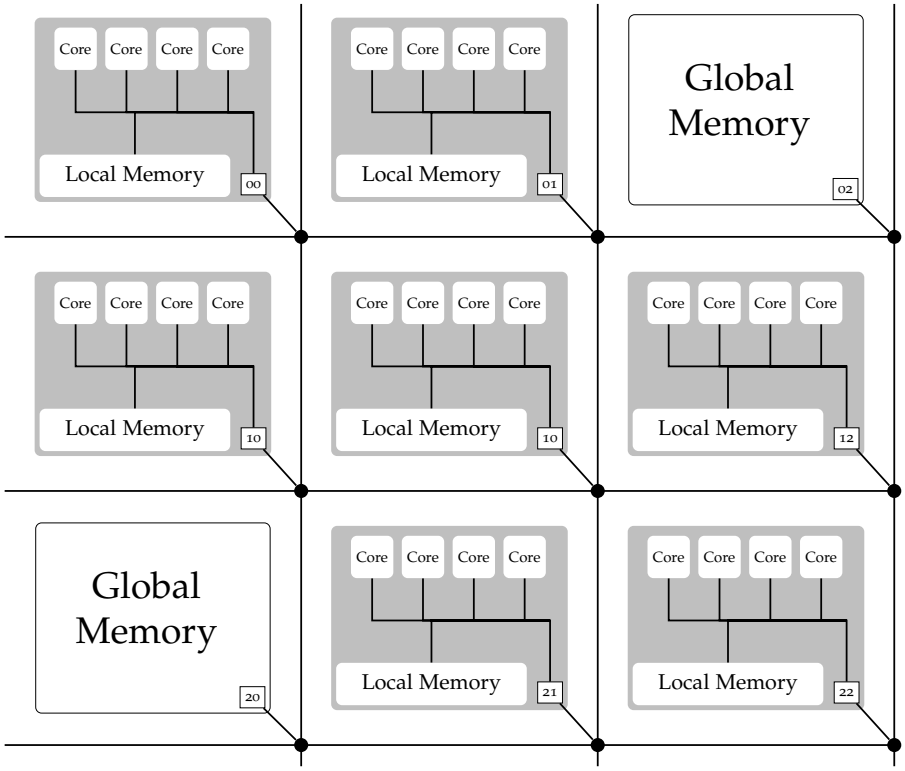


Figure 3.2.: An example of a clustered many-core architecture. The shown example has 9 tiles, arranged in a 3×3 mesh and connected by a network-on-chip. Seven of the tiles contain processor cores and two tiles provide access to external memory. Each tile has a few MB local on-chip memory. Global memory is off-chip DDR RAM.

The *i*NoC also provides a memory transfer mechanism, like a DMA controller on mainstream hardware. The operating system implements message passing with it and the X10 runtime relies on that.

There were plans to make this architecture family more heterogeneous. Unfortunately, the implementations were not solid enough to use them for the evaluations in this dissertation. Nevertheless, here is a short overview for completeness as it influenced the design of the framework.

There are special cores called *i*-Cores [HBHG11], which are extended Leon SPARC v8 cores. They include a reconfigurable fabric, which can be used to dynamically load different accelerator instructions at runtime. For example, an *i*-Core can decode H.264 video 22 times faster [HBHG11] than a Leon. A resource-aware program should be able to exploit such accelerators. This requires to request *i*-Cores and to use them correctly. As this uses instructions not part of the official SPARC v8 ISA, special compiler support is necessary.

Another accelerator are Tightly-Coupled Processing Arrays (TCPAs). They are energy-efficient [SHT+15, HLB+14] and timing-predictable [GSL+14, TGR+16]. TCPAs are a mesh of simple processors (16bit, no pipeline). Only the processing elements at the edges of the mesh have memory access. TCPAs can be used to accelerate hot loops by using symbolic tiling techniques. Like with an *i*-Core, using TCPAs requires compiler and language support. Paul et al. [PSS+14] used it in an invasive system to improve the processing rate and accuracy of a computer vision algorithm known as Harris Corner Detection.

3.3. Operating System: *i*RTSS

On top of this hardware is a custom operating system called *i*RTSS. It consists of a tile-local and a chip-global part.

3.3.1. OctoPOS

The *i*RTSS API is a superset of the API the custom operating system OctoPOS [OSK+11] provides. OctoPOS provides *i*-lets for concurrency. In contrast to the widely known pthreads abstraction, these *i*-lets have run-to-completion

```
1 finish {
2   async { foo(); }
3   l.lock(); // waits for a long time
4   bar();
5   l.unlock();
6 }
```

Figure 3.3.: Unintuitive Run-To-Completion Semantics. If you want to compute a background *i*-let foo() while waiting for a lock, you might intuitively write this code. However, if you only have one core, then your intuition can lead you astray: That "asynchronous" foo() does not execute in parallel to lock().

semantics. While they can be interrupted, it is a nonpreemptive multitasking model. This design choice makes sense on an architecture, where cores are plentiful and multiplexing single cores is not worth the complexity and overhead it introduces to the system.

Synchronisation is discouraged in OctoPOS, because everything is designed for scalability. Apart from hardware cmpxchg instructions, the operating system only provides a spinlock, which *actively* waits by continuously trying to take the lock. Unfortunately, this sometimes leads to surprising behavior like the following. If you want to perform something in parallel to a blocked *i*-let. For example, waiting for a lock does not give the core to concurrent activities like in fig. 3.3. What happens:

1. The background *i*-let foo() is queued by the scheduler, but not started.
2. The foreground *i*-let spins on the lock for a long time until it takes the lock l. Nothing productive happens apart from continuously checking the lock.
3. The foreground *i*-let computes bar() and unlocks
4. The foreground *i*-let hits the end of the finish block and waits for the child activities.
5. The background *i*-let foo() is scheduled and starts running.
6. The background *i*-let finishes and the foreground *i*-let continues behind the finish block.

The two activities compute sequentially. The background *i*-let did *not* run in parallel, while the foreground *i*-let was waiting on the lock.

Since we assumed only a single available core, parallelism is not an option, though. What would be desirable is first to try *once* to take the lock, and if that fails execute `foo()`. It is possible to code that using the `tryLock()` method, but then the code fails to parallelize if it were execute with multiple cores available. We considered options where the compiler implicitly instantiates continuations and coroutines. However, that would have required a lot of work in the compiler and so we decided this out of scope. Programmers would have to live with this pitfall.

Octopos also provide "claims", but they are *not* the claims defined in the previous chapter, so we will call them "OctoPOS claim" here. An OctoPOS claim only contains PEs of a single type on the same tile. Also, OctoPOS does only provide the mechanisms for claim management, but leaves decisions to higher levels. An OctoPOS claim is implicitly a scheduling domain, which means an *i*-let created in one OctoPOS claim will never execute on a PE in another OctoPOS claim.

While OctoPOS runs a separate instance on each tile and is generally a per-tile operating system, it also provides communication mechanisms between tiles. OctoPOS claims are used as target addresses for this communication. You can spawn an *i*-let on a remote claim, which is the minimal mechanism as it can only transfer two words (8 B) of data. Additionally, there is a mechanism to transfer blocks of data asynchronously, which spawns *i*-lets on source and target once the data is transferred. The programmer cannot control which PE will execute an *i*-let, but since they are by definition of a single type and on the same tile it should not matter.

What OctoPOS does not provide is resource management on an *inter*-tile level. For this, the second part of *i*RTSS is responsible.

3.3.2. Agent System

As described in chapter 2, one core concept is the "claim". Since a claim potentially spans multiple tiles, OctoPOS cannot manage it. Instead, for each claim, there is a corresponding "agent" which trades resources with other agents. This "agent system" provides the resource management service for the whole system. It interfaces with OctoPOS to control intra-tile resources.

Sebastian Kobbe developed the basics with DistRM [KBL+11] where he describes a decentralized system, where resources are traded in a peer to peer fashion. However, DistRM is not what is implemented in *i*RTSS, as we have no hardware with hundreds of cores available yet. Instead, *i*RTSS uses a centralized approach which is more efficient in our scenarios.

While the goal is to support any constraint and hint possible, only a few were implemented. The *i*RTSS version we use for evaluation provides the means to specify

1. the minimum and maximum number of PEs
2. if PEs should be Leon or *i*-Core
3. a scalability curve via Downey's model A and σ
4. a specific which tiles the PEs may or may not be on
5. if tiles may be shared with other claims

The API of *i*RTSS was designed with the requirement that bindings to other languages, like X10, are simple to realize. Although it is implemented in C++, it is restricted to the int type, opaque pointers and a C API and ABI. There are functions to modify the constraints request and of course invade, reinvade, and retreat mechanisms. The framework exposed to the X10 programmers maps to this lower-level C API.

3.4. The X10 Programming Language

Since we use the X10 programming language extensively and it is not a mainstream language, this is a short introduction. Here is its history according to its website[1].

> The genesis of the X10 project was the DARPA High Productivity Computing Systems (HPCS) program. As such, X10 is intended to be a programming language that achieves "Performance and Productivity at Scale." The primary hardware platforms being targeted by the language are clusters of multi-core processors linked together into a large scale system via a high-performance network. Therefore, supporting both concurrency and distribution are first

[1] `http://x10-lang.org/home/x10-history.html` as of 2015-08-04

> class concerns of the program language design. The language must also support the development and use of reusable application frameworks to increase programmer productivity; this requirement motivates the inclusion of a sophisticated generic type system, closures, and object-oriented language features. Finally, like any new language, to gain acceptance X10 must be able to smoothly interoperate with existing libraries written in other languages. This last requirement constrains both the design and the implementation of X10 in various ways

X10 started as an extended Java and later adopted a Scala-like syntax. Most prominent are the new keywords async and at. They provide "activities" and "places".

3.4.1. Activities

An "activity" is like a lightweight thread or task in other languages. It models parallelism in a scalable fashion and the programmer is encouraged to create lots of activities. The X10 runtime system maps activities to OS primitives like POSIX threads and thus limits the actual parallelism. The finish keyword enables to wait for the termination of activities in a deadlock-free way. The following snippet shows a parallel execution of foo and bar and waits for termination of both at the end.

```
1 finish {
2   async { foo(); }
3   async { bar(); }
4 }
```

Figure 3.4 shows a more realistic example. It performs numerical integration. The work load varies dynamically and is unpredictable, because you need to know the flatness/steepness of a function, as illustrated in fig. 3.5. Numerical integration is investigated in more detail in chapter 7.

On *i*RTSS, our runtime maps activities to OctoPOS *i*-lets directly. This implies activities have run-to-completion semantics. In contrast, X10 on Linux maps activities to pthreads which have different semantics. Fortunately, both is fine by the X10 language specification [SBP+14]. It requires some additional bookkeeping to implement the finish keyword: Activities must be accounted for in

```
1  import x10.lang.Math;
2  import x10.util.concurrent.AtomicFloat;
3  public class MinimalIntegrate {
4    static type T = float;
5    private val function:(T)=>T; // function to integrate
6    private val epsilon:double; // error threshold
7    private val max_depth:uint; // max recursion depth
8    private val result = new AtomicFloat(0.0f);
9    private def integrateRange(left:T, right:T, depth:uint):void {
10     val l = function(left);
11     val width = right - left;
12     if (depth > max_depth) {
13       result.addAndGet(l*width);
14       return; }
15     val r = function(right);
16     if (epsilon > Math.abs((l - r) as double)) {
17       result.addAndGet(l*width);
18       return; }
19     val center = left + width/2;
20     async { integrateRange(left, center, depth+1); }
21     async { integrateRange(center, right, depth+1); } }
22   public def this(p_function:(T)=>T, p_epsilon:T, md:uint) {
23     function = p_function;
24     epsilon = p_epsilon;
25     max_depth = md; }
26   public static def main(args: Array[String]) {
27     val fun = (x:T) => { return Math.sin(x) as T; };
28     val work = new MinimalIntegrate(fun, 0.001 as T, 5);
29     finish work.integrateRange(0,3,0);
30     val result = work.result.get().toString().substring(0,6);
31     Console.OUT.println("Integral of sin(x) between 0 and 3: "
32                         +result+" (should be ~1.9899)"); } };
```

Figure 3.4.: Minimalistic numerical integration of $\int_0^3 sin(x)dx$ in parallel X10 code.

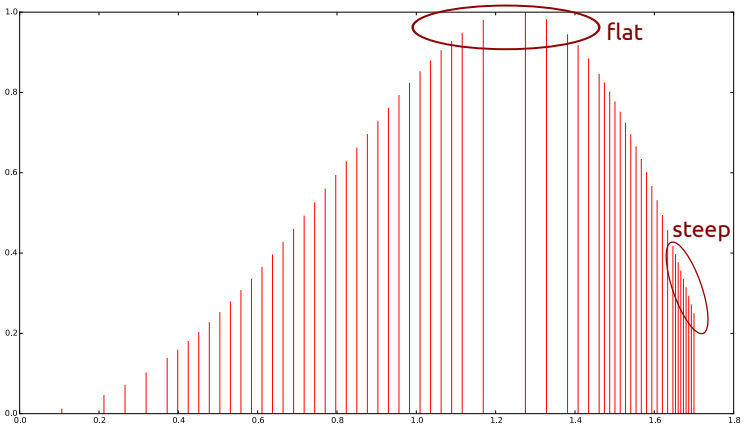


Figure 3.5.: To numerically compute an integral, like $\int_0^3 sin(x^2)dx$ here, you can evaluate a function at various points using the recursive rectangle method. Where the function is steep, we evaluate more points, since we expect more volatility. Flat regions require less evaluations, since we expect little volatility.

their lexically surrounding finish-block. At the end of a finish block, an activity blocks until all activities spawned within have terminated. The implementation inserts finish-state data structures at the corresponding places.

Conceptually, we organize activities in a tree, where leaf nodes are activities and branch nodes are finish-blocks. Since a tree contains no cycles, finish alone cannot result in deadlocks. X10 provides additional synchronization mechanisms, like transactional memory via atomic and primitive atomic types and barriers, such that programs can be deadlock-free by construction. While synchronization mechanisms like locks are also available and using those may introduce deadlocks, X10 promotes deadlock-freedom as it supports the safe mechanisms with syntax and provides unsafe ones only by library.

3.4.2. Places

A "place" represents a shared memory domain. Within a place, all activities have access to the same memory. Between places data can only be exchanged

via at, which migrates an activity and its context to another place and back. Conceptually, places form a ring topology with place ids from 0 to n. On a supercomputer system, a place usually maps to a process on a compute node.

The following snippet shows how to migrate to the next place to call the getX method and migrate the return value back to assign it to x. The keyword here always evaluates to the current place. The migration copies the object obj to the next place and calls the method on it.

```
1  val x = at (here.next()) obj.getX();
```

The integration example in fig. 3.4 only works on a single place. Let us generalize it to a "distributed" algorithm, which uses multiple places. We could simple chop the integral into n regions for n places, but the load would probably not be balanced, because steep regions require more resources. Since we cannot predict the resource usage, we have to balance dynamically. In fig. 3.6, we use a distributed queue to achieve this.

3.4.3. Distributed Data

While at provides a way to distribute (i.e. copy) data, you also need a way to refer to data which already exists. There are three options.

1. Static fields exist once in each place. They are always immutable, but they can contain an immutable reference to a mutable value (Cell, Array, ...).
2. A PlaceLocalHandle is a "variable" with one instance per place. In contrast to static fields, it can be created dynamically.
3. A GlobalRef is a "global reference" for a single value. In contrast to the previous two options, there is only one instance on one place. While the reference can be freely copied to other places, the value is not copied implicitly. To access the value, the programmer has to explicitly move the computation to its home place or explicitly copy the value to the current place.

While X10 is a garbage collected language, garbage collection is place-local. This means PlaceLocalHandle and GlobalRef must be freed manually (or the program leaks memory). Pointers stored in there might be referenced on other places, so with only local information we cannot know if we are allowed to free it safely.

```
1  public def integrateRange(left:T, right:T):T {
2    /* initialize job queue with fair regions */
3    val stride = (right − left) / Place.places().size();
4    for (var i:uint = 0; i<parts; i++) {
5      val l = left + (i *stride);
6      val r = left + ((i+1)*stride);
7      todo().enqueue(new Job(l, r, 0)); }
8    /* parallel distributed worker activities */
9    finish for (p in Place.places()) at async {
10       while (true) {
11         val jobbox = todo().dequeue();
12         if (jobbox == null) /* all jobs finished */
13           break;
14         val job = jobbox();
15         integrateRange(job.left, job.right, job.depth); } }
16   return result().get() as T; }
17 private def integrateRange(left:T, right:T, depth:uint):void {
18   val l = function(left);
19   val width = right − left;
20   if (depth > max_depth) {
21     result().addAndGet(l*width);
22     return; }
23   val r = function(right);
24   val difference = Math.abs((l − r) as double);
25   if (difference < epsilon) {
26     result().addAndGet(l*width);
27     return; }
28   /* split in two parts for recursion */
29   val cntr = left + width/2;
30   todo().enqueue(new Job(left, cntr, depth+1));
31   todo().enqueue(new Job(cntr, right, depth+1)); }
```

Figure 3.6.: Distributed numerical integration with at.

X10 also provides distributed higher-level data structures. Such data structures are built on top of above three basic ones. For example, the distributed array DistArray is for matrices too large for a single place. It internally uses PlaceLocalHandle such that each place stores the corresponding data and adds distribution information to track the data.

This concludes the chapter on the platform this thesis builds upon, including the hardware and operating system. We have a broad impression of the mechanisms in our programming language, its philosophy, and style. The next chapter dives into technical details of X10, before chapter 5 presents the resource-aware framework developed in Invasive Computing including compiler, runtime, and language changes.

X10 Memory Consistency Model

> *The point of rigour is not to destroy all intuition; instead, it should be used to destroy bad intuition while clarifying and elevating good intuition.*
>
> — Terence Tao

A memory consistency model (in short "memory model") specifies what values parallel execution threads can observe in shared data in what order. This is important for the programmer, who reasons about the behavior of a program, and for the compiler, which must map it to the target hardware's memory model. Every implementation implicitly defines a model. Also, every programmer has a model in mind. Hopefully, the models are equivalent, but without a clear specification they usually are not.

In this chapter, we define a memory consistency model for X10. It was originally published at the X10 workshop [Zwi16]. The model here is semantically equivalent, but the presentation is revised. Furthermore, this chapter describes the requirements analysis, discusses differences to other memory models, and issues discovered with X10 semantics. The actual model is described from section 4.3 until section 4.6.

The X10 specification [SBP+14] does not contain a memory model. This is perilous, since the compilation targets are Java and C++, which have different memory models [MPA05, ISO14]. An X10 program's behavior should be the same, no matter which compiler backend is used. Thus, we must specify the behavior for the source language X10.

4.1. Intro to Memory Consistency Models

For an overview over memory models of programming languages, the popular short version is: "Sequential consistency (SC) for data race free programs". The formal version is:

> A program whose sequentially consistent executions have no data races must have *only* sequentially consistent executions.

An "execution" is the trace of a single run of a program. To understand the definition, let us look into what SC and data race mean.

4.1.1. What Is Sequential Consistency?

Memory models in general are about the order of memory "actions", which are read, write, compare-and-swap, etc. Naturally, program code defines an order within a single thread. A compiler and a programmer can understand and work with this sequence. However, if two (or more) threads run in parallel, then many interleavings become possible and programs may become indeterministic. SC is originally defined by Lamport in 1971 [Lam79]. Since then, terminology has changed. A modern definition is:

> Within each thread memory accesses follow program order and all threads immediately observe every access.

This implies a global total order of actions for a given execution.

For memory models we define the term "synchronization operations", which explicitly synchronize global memory state for the thread that executes it. We can define a simple memory model SC, which declares every memory access a synchronization operation. The problem is that this prohibits a lot of optimizations, which we want to allow compilers and CPUs to do. Thus, Java and C++ use weaker memory models. They define normal reads and

Let flag be false initially	
flag = true	flag = true

Figure 4.1.: An example of a **data race without a race condition**. Two threads set the same flag to true. There is a data race, because those actions conflict and there is no happens-before relation. There is no race condition, because the outcome is deterministic. We do not care about the order in this case as the behavior is identical.

writes as *not* sequentially consistent. Instead, they provide a limited set of synchronization operations like locks, monitors, semaphores, atomics, and other special constructs. A library cannot provide such constructs [Boe05], so they belong to the language specification.

With weaker memory models than the simple one above, one thread may observe a different order of actions than another thread. Obviously, this means additional complexity for a programmer, who tries to understand an execution in a debugger. The compromise between performance and safety is to restrict these order mismatches to data races and make the programmers responsible to avoid them.

4.1.2. What Is a Data Race?

If two memory actions are a) not synchronization operations, b) access the same data, and c) at least one writes, then we say they "conflict". A conflict is not necessarily a data race, because there may be additional synchronization operations, which guarantee an order as they define "happens-before" relations between actions. If two actions conflict and have *no* happens-before relation, then we have a "data race".

Be careful not to confuse a data race with a race condition, which means that timing or ordering of events affects a program's behavior. Some only consider it a race condition, if the behavior is faulty/incorrect. We use a weaker definition here for simplicity and to not require a notion of "correctness". Although data race and race condition often occur together, these are orthogonal concepts. Race conditions are the reason why we have so many different executions with parallel and concurrent programs. Figure 4.1 and fig. 4.2 show examples of one, but not the other.

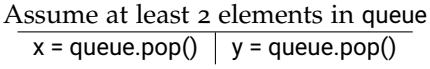


Figure 4.2.: An example of a **race condition without a data race**. Two threads try to take an element from a properly synchronized queue. Since pop is synchronized, there is no data race. There is a race condition, because it is not deterministic which thread gets which element.

4.2. Requirements for X10

Before we present the actual memory model, we analyze the requirements and derive design decisions.

The X10 compiler targets Java and C++, and there is also our inofficial assembly backend [BBMZ12]. This means whatever memory model we design, it must be possible to map it to the Java memory model (JMM) [MPA05], the C++ memory model (CMM) [ISO14], and hardware memory models.

Another aspect is that X10 targets High-Performance Computing (HPC), so performance is important.

4.2.1. Data Races Are Undefined Behavior

Programmers strive for data race free code, so CMM considers data races as undefined behavior. "There are no benign data races" in C++ [ISO14]. In contrast, Java must define the semantics of data races, otherwise a data race could be exploited to, e.g., circumvent the security manager. Via data races the current JMM fails [Loc14] to prevent an execution from reading values "out of thin air", which were never written according to the program. Must X10 care about the semantics of data races and Thin Air Reads? No, because X10 provides no isolation mechanism within the language, which would have to be secure even with data races. Neither does X10 need to use a memory model framework [SJMvP07], which supports this complexity.

If X10 defines semantics for data races, then the compiler must maintain this semantics in C++ and must not generate code with undefined behavior. Thus, the compiler must litter the code with additional synchronizing operations like memory fences, which degrades performance. This is not acceptable for HPC

```
1  def foo(y:int,n:int):void {
2    var x:int = 0;
3    while (x < y) { x += n; }
4  }
```

Figure 4.3.: We store the local variables x, y, n in registers, so there is no memory access within the loop. During the execution there is no "action" (see below) with respect to the memory model, so we consider the loop empty. Additionally, x is not used after the loop, so we do not care about its value. We cannot guarantee termination, since n might be zero. Still, the compiler can remove the loop.

programs, thus X10 cannot provide a semantics for data races. While undefined behavior is a source of agony, the advantage is a simplified memory model and a better performance.

4.2.2. Termination Can Be Assumed

With a similar argument, an X10 compiler can assume that all loops terminate. C++ [ISO14, §1.10.27] allows to remove empty loops even if they might not terminate. Thus, the compiler can remove an empty loop in X10, if it is compiled directly to a C++ loop. Therefore, X10 must use an equivalently weak semantics or the compiler must ensure not to generate empty loops by inserting dummy statements. Since the upside of stronger semantics is not clear, we assume that empty loops can be removed and thus assume termination. We see an example in fig. 4.3. This is the behavior of the current X10 version 2.5.

4.3. Actions and Executions

This sections provides a complete memory model for X10. The structure of this section mostly matches the Java memory model [MPA05] §7, with the necessary parts from §5 and §9 merged in. Where it made sense, we copied the text verbatim for better comparison, so a lot of credit goes to the authors of the JMM. However, we changed details in the adaption to X10.

In X10, an "activity" is the concept to model a thread of execution. A "place" is a shared memory domain. Activities within the same place use the same heap. Activities in different places cannot communicate via shared memory. Instead, the programmers must use the at construct to transfer an activity to another place, which implicitly copies context data.

An action a is described by a tuple $\langle t, k, v, u \rangle$, comprising:

t the activity performing the action.

k the kind of action.

Most kinds are synchronization operations: activity creation (within the spawning activity), start and end of an activity, global termination of finish block, lock, unlock, library and external actions.

Two kinds are not: read and write.

v the variable or lock involved in the action. Variables and locks on different places cannot overlap.

u an arbitrary unique identifier for the action.

As a notation for the variable or lock v of an action a, we use the notation $a.v$ in the following.

An execution E is described by a tuple $\langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V \rangle$, comprising:

P a program.

A a set of actions.

$\xrightarrow{po}$ program order, which for each activity t is a total order over all actions performed by $t \in A$.

$\xrightarrow{so}$ synchronization order, which is a total order over all synchronization actions in A. For $a_0 \xrightarrow{so} a_1$, we say a_1 is "subsequent" to a_0.

W a write-seen function, which for each read $r \in A$, gives $W(r)$, the write action seen by r in E.

V a value-written function, which for each write $w \in A$, gives $V(w)$, the value written by w in E.

An "external action" is an action that may be observable outside of an execution, and may have a result based on an environment external to the execution. For example, input and output of a program introduces external actions. An external action tuple contains an additional component, which contains the results of the external action as perceived by the activity performing the action. This may be information about the success or failure of the action, and any values read by the action. Parameters to the external action (e.g., which bytes are written to which socket) are not part of the external action tuple, since it does not concern the memory model.

A "library action" is an action from the standard library, which provides additional synchronization mechanisms as shown in section 4.6.

4.4. Synchronizes-with and Happens-before

Two additional relations are uniquely determined from an execution. Since we never reason about multiple executions at the same time, we do not annotate E with those relations explicitly.

$\xrightarrow{sw}$ "synchronizes-with", a partial order over synchronization actions determined by the total synchronization order according to the rules below.

1. An unlock action on lock l synchronizes-with all subsequent lock actions on l.
2. An action that creates an activity synchronizes-with the start action of the created activity.
3. The write of the default value to each variable synchronizes-with the first action in every activity (Conceptually, every object is created at the start of the program). The default value of non-static val fields is their initialization value.
4. The end action of an activity synchronizes-with the end of the surrounding finish block.
5. The last action of an `atomic` or `when` block synchronizes-with the first action of subsequent `atomic` blocks and `when` conditions.
6. Further actions as specified in parts of the standard library in section 4.6.

A set of synchronizes-with relations is "sufficient" if it is the minimal set such that you can take the transitive closure of those relations with program order relations, and determine all the happens-before relations in the execution. This set is unique.

$\xrightarrow{hb}$ happens-before, a partial order over actions is the transitive closure of synchronizes-with and program order.

4.5. Well-formed Executions

We only consider "well-formed executions". An execution $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V \rangle$ is well-formed if the following conditions are true:

1. Each read of a variable x sees a write to x. For all reads $r \in A$, we have $W(r) \in A$ and $W(r).v = r.v$.
2. Synchronization order is consistent with program order and mutual exclusion. Having synchronization order consistent with program order implies that the happens-before order is a valid partial order: reflexive, transitive, and antisymmetric. Having synchronization order consistent with mutual exclusion mean that on each lock, the lock and unlock actions are correctly nested.
3. The execution obeys intra-activity consistency. For each activity t, the actions performed by t in A are the same as that activity t would generate in program order in isolation, with each write w writing the value $V(w)$, given that each read r sees/returns the value $V(W(r))$. The memory model determines the values seen by each read. The program order must reflect the program order in which the actions would be performed according to the intra-activity semantics of P, as specified by the parts of the X10 specification that do not deal with the memory model.
4. The execution obeys happens-before consistency. Consider all reads $r \in A$. It is not the case that $r \xrightarrow{hb} W(r)$. Additionally, there must be no write w such that $w.v = r.v$ and $W(r) \xrightarrow{hb} w \xrightarrow{hb} r$.

Like C++ [ISO14, §1.10.27], any X10 activity will eventually "terminate, make a call to a library IO function, access or modify an atomic object, or perform a synchronization operation". This means in contrast to Java, we do not need to consider infinite executions. No "hang" action is necessary.

4.6. Constructs in the Standard Library

4.6.1. Atomics

The X10 runtime comes with atomic boolean, double, float, integer, long, and reference types in x10.util.concurrent. All method invocations of these constructs synchronize-with subsequent invocations on the same object.

4.6.2. Clock

For x10.lang.Clock the synchronizing methods are advance, advanceAll, drop, resume, resumeAll. All its method invocations synchronize-with subsequent invocations on the same clock object. In the case of advanceAll and resumeAll this affects all clocks the activity registers at.

4.6.3. Condition

All method invocations of x10.util.concurrent.Condition synchronize-with subsequent invocations on the same object.

4.6.4. Lock

The x10.util.concurrent.Lock class provides (surprise!) a lock. All its method invocations synchronize-with subsequent invocations on the same object if the operation is successful. Other synchronization primitives Monitor, SimpleIntLatch, SimpleLatch, IntLatch, Latch inherit from Lock, so they need no special treatment.

For clarification, the behavior of the trylock method is equivalent in Java, C++, and X10. The method acts like lock() if a lock is not taken already and returns a boolean whether it has taken the lock. C++ explicitly gives trylock the freedom for "spurious failure" [BA08], which means it might fail to lock even if nobody else held it at the time. Java also provides this implicitly. The JavaDoc for Lock says

x = 42	while(l.tryLock())
l.lock()	l.unlock()
	assert(x == 42)

Figure 4.4.: Undesirable use of trylock from [BA08]. The second thread waits for someone else to take the lock[1]. The assert may fail, because there is no happens-before relation with the assignment according to JMM or CMM.

> Unsuccessful locking and unlocking operations, and reentrant locking/unlocking operations, do not require any memory synchronization effects.

If trylock should be synchronizing even if it fails, it requires a slower lock() implementation with an additional fence instruction. There is a motivating example in fig. 4.4, which C++ uses to motivate spurious failure.

If the tryLock() succeeds in taking the lock, it would synchronize-with any previous locking operation. Via Lock semantics we know there cannot be a previous locking operation as we enter the loop. Thus, if we enter the loop, there is no synchronize-with relation between the threads.

If (or when) the trylock fails it has no synchronization effects. Thus, there is no synchronize-with relation to the first thread. We assume that tryLock() failing is an "unsuccessful locking operation".

Therefore, in either case there is no happens-before relation between assignment and assert. Hence, we have a data race and the value of x might or might not be 42. A compiler is free to move the assignment into the critical section, because nobody outside of the critical section can observe the assignment.

Even while there is nothing explicit about Javas tryLock being "spurious", this example demonstrates spurious behavior in Java. The JavaDoc of tryLock() says:

> Acquires the lock if it is available and returns immediately with the value true.

[1] Advice for programmer: Convert x into an AtomicInteger, remove the lock, and spin on x directly.

This is not wrong, but misleading in our example since the observed behavior looks like a spurious failure. There is no happens-before relation between assignment and assert, even if tryLock() correctly observed the lock as taken.

4.7. Differences to other languages

The two most well-known memory consistency models are Java and C++, so we compare them in more detail here. Then there is the Chapel programming language, which was designed at the same time with a similar purpose as X10.

4.7.1. Differences between X10 and Java

Compared to Java, X10 lacks four features, which simplify the memory model (and complexity of the language in general).

1. There is no Thread or Activity object in X10[2], so one cannot interrupt or join an activity. There is only the finish block, which waits for the (global) termination of all activities within.

2. X10 has no (user-defined) finalizers for objects. This also simplifies the model. See the JMM [MPA05, §16] for the details. In general, finalizers are avoided, because execution is non-deterministic.

3. X10 does not provide reflection in a modifying way. The issues of modifying final fields in Java at any time do not exist. Corresponding to Javas final fields are val fields in X10.

4. There is no security manager for isolation on the language level. To guarantee isolation like this, we would have to prohibit undefined behavior, as it includes breaking isolation.

[2] They exist hidden in the runtime, but not accessible to the programmer.

4.7.2. Differences between X10 and C++

The CMM distinguishes between acquire and release synchronization, because this is relevant to support "relaxed" operations. However, such mechanisms are not provided by the X10 standard library nor supported by the compiler in any way. For high performance applications it might become worthwhile to provide this at some point, e.g., to implement non-blocking data structures. Then we must adapt the memory model.

C++ has bitfields, where the compiler can compact fields to a certain amount of bits. This affects the memory model, since writes to bitfields are usually also writes to neighboring bitfields and might introduce data races. X10 has no such feature, which also simplifies the memory model.

4.7.3. Differences between X10 and Chapel

Chapel [Inc17] is a very similar language to X10. It was initially funded by the same DARPA program. It uses the PGAS paradigm. It provides high-level data structure like distributed arrays. It is object-oriented and type-safe.

Even the memory consistency model (§29 of the specification) is similar. It is based on sequential consistency for data-race-free programs. For other programs, the specification says "Any Chapel program with a data race is not a valid program, and an implementation cannot be relied upon to produce consistent behavior".

Unlike X10, Chapel provides weaker operations: non-sequentially consistent atomic operations, which are like C++ relaxed memory order operations. Additionally, Chapel has unordered memory operations, which require the use of a separate synchronizing variable, so the compiler inserts a fence operation. These mechanisms promise improved performance, if one is willing to pay the price (no safety, ugly code). There should be no problem to add similar mechanisms to X10 without changing fundamental parts of the memory model.

4.8. StoreStore Barrier After Constructor

Our proposed X10 memory model, and also JMM and CMM, specify no relation between references and the referenced location. This includes for example the this-pointer of an object and its fields. If you assign to a field and share the object reference with another activity, reading the field might not yield the assigned value. This is counterintuitive, if the assignment is within the constructor to an immutable `val` field. Even if the object sharing is synchronized, the field accesses are not.

In Java, the final field semantics cover this even with a data race. In C++, this is undefined behavior. The implementation advice from the JMM authors [MPA05, journal version] is a store-store-barrier at the end of a constructor. This is `NOP` on x86, which probably explains why nobody is hurt in practice. On ARMv8 the barrier is necessary, for example.

4.9. Global Address Space and the Memory Model

X10 models an asynchronous partitioned global address space (APGAS), but the memory model does not address this explicitly. It is not necessary, because an activity can only access its own part of the address space [SBP+14, sec. 2.4]:

> With remote reference, an activity can access objects at a remote place (remote objects) when the activity has moved to the remote place.

Moving an activity requires `at` and is already covered by program order. The implicit deep copy of context before and after `at` does not concern the memory model, apart from the normal read operations, which might or might not be properly synchronized.

It is possible to introduce arbitrary additional features using native code within the runtime, but also in user code. This extensibility is desirable and nobody wants to restrict that. In this case the documentation must include information about the synchronization. One example would be `Rail.asyncCopy`, which in version 2.5.4 specifies with respect to synchronization:

> The activity created to do the copying will be registered with the dynamically enclosing finish.

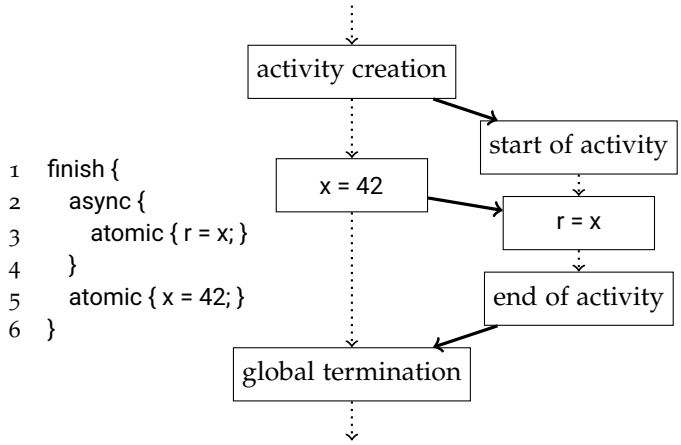


Figure 4.5.: An example execution of an activity life cycle as written in the code on the left. Each box is an action. The thick arrows are synchronize-with edges and the dotted arrows show program order. The figure demonstrates the difference between "activity creation" and "start of activity" actions.

This implies the only way to synchronize is finish at the sending place. Thus, synchronization with the receiving place requires an additional at.

4.10. Threads and Activities

The JMM talks about threads, while we have activities in X10. For the memory model this does not matter much, since both just model parallel execution. Figure 4.5 shows an example of activity creation.

The JMM explicitly mentions the problem of thread inlining [MPA05, fig. 12], which puts code of one thread into another one and removes parallelism. This must be forbidden in Java, because it introduces an additional happens-before relation and a compiler might optimize the program in an undesirable way. X10 uses lightweight activities, so the programmer is encouraged to create more of them than a Java programmer would create threads. Consequently, activity-inlining would be even more desirable than thread-inlining. The Java example [MPA05, fig. 12] relies on a data race and is thus undefined in X10. So, activity-inlining should be possible in X10.

This chapter presented a possible memory model for X10 together with a rationale for its design. While we started with the JMM, due to X10's roots in Java, the outcome is closer to the CMM. One essential assumption of this proposal is the use case of high performance computing. If X10 is changing towards Java cloud computing and orchestration[3], then Java interop might get priority over performance. In this case, a semantics for data races might become necessary and we must adapt the memory model towards Java.

During the specification process, we uncovered minor issues in the X10 standard library. The `@Volatile` annotation has unclear semantics and might be unnecessary. The `Fence` utility class is broken and you should not use it. We have reported these issues as 3547, 3548, and 3549 in the X10 bugtracker[4]. Sadly, the issues are still open over one year later.

A possible critique of the style of the JMM and this work is the distance to language semantics. Before inclusion in the language specification it would be worthwhile to express the memory model closer to the feature descriptions and language semantics. The Chapel specification [Inc17] is conveniently short and could serve as a prototype.

With a solid basis of X10 semantics, we can proceed to define a resource-aware framework on top.

[3] rumors via personal communication

[4] `https://xtenlang.atlassian.net/projects/XTENLANG`

Framework InvadeX10

> *Not being able to get the future exactly right doesn't mean you don't have to think about it.*
>
> — Peter Thiel

At the end of the first funding phase of Invasive Computing in 2014, we wrote down documentation for the framework language InvadeX10 [ZBS13]. A previous publication [Zwi12] provides an shorter but non-exhaustive introduction. Since then we added more features and changed details. This section provides an overview over the current state of the language with all its invasion parameters and convenience methods. It provides a common language between hardware and software developers, who easily misunderstand each other. Developing a practical framework for a diverse set of applications is hard to get right. From the start, we tried to use a methodology as objective as possible.

5.1. Development Methodology

The initial challenge is solve the chicken and egg problem mentioned in section 1.6: First, understand the possibility of the hardware and the needs of the software. Second, provide stubs for both domains to kickstart development.

As a first step, we inquired all related people to provide examples. Together with basic engineering principles, this lead to a first prototype language. Naturally, not all examples were fully covered, so we had to adapt them together

with its original authors to maintain the intent. When the examples were ported to the prototype, it served as common ground. For the software, we could emulate the lower levels. For the hardware, we could design the interfaces which enabled further development of the language.

A first prototype is not final and in general a language which stops evolving is dead. For further development we established a process, where a document had to be written for each change or extension to the language. The Language Change Proposal (LCP) process is modelled after well-known processes like for example RFCs (network protocols), PEPs (Python), and JSRs (Java). This process provided parallelism, because at this point feature requests were increasingly independent of each other. I served as the benevolent dictator of the language and the gatekeeper for all changes. In nearly all cases, I also implemented the framework parts.

5.2. Hello World

Now we look at the concrete language. We build on top of X10 as described in section 3.4. The concepts of invasion and retreat were already described in chapter 2; In between, there is also infect. Conceptually, there are three phases for allocating, using, and freeing resources.

As traditional in programming languages, we start with a trivial "Hello World" application that demonstrates all the necessary concepts. The invasive part is:

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

The static class method Claim.**invade** takes constraints and returns a claim object, which represents the allocated resources. Here, we implement a capability-based security concept. Since resources are exclusive to a claim, the claim serves as a handle to access those resources. Section 5.9 investigates the implications more deeply and discusses the changes to the semantics of X10.

A claim object provides an **infect** method to distribute computation across processing elements. 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. This was considered the more safe alternative. If

you desire concurrent execution, you may prefix it with async, the X10 keyword. The semantics of X10 on mainstream operating systems matches the intent accurately. Later, we realized that this is actually unintuitive when executed on the OctoPOS operating system with its run-to-completion semantics. If the current claim, which executes the infect has only one processing element, it will run to completion (until the end of the current finish block usually) before the activity to perform the infect begins. However, the programmer intuitively assumes that infection starts immediately. Since a workaround is possible and we already had a set of programs, we considered backwards compatibility more important and kept the unintuitive behavior.

Another aspect of infection is "isolation". The resources of a claim are not accessible or visible outside of a claim, except via the claim as a handle. So, **infect** also serves as the mechanism to cross isolation barriers between claims.

After the infect, the **retreat** method frees all resources within a claim, such that the claim is empty. If you would call infect again it would dutifully execute the given *i*-let on each of the claims processing elements, which are none.

Now consider the **ilet** variable of the example above. You could define it as follows:

```
1  val greeting = "Hello from ilet ";
2  val ilet = (id: IncarnationID ) => {
3     Console.OUT.println(msg + id.ordinal);
4  };
```

The **ilet** variable is assigned a closure. The closure takes one argument id, which it may use to distinguish itself from other *i*-lets executed by infect on different PEs and tiles. The whole expression is a closure, which means the compiler includes its free variable greeting and the variable is copied to every processing element. The X10 compiler and runtime system implicitly take care that all data is serialized and deserialized if transfer to different places is necessary. If the processing elements should get different data, then the programmer should instead include a handle and fetch the correct data from within the *i*-let.

This *i*-let code executes on each PE in the claim during infect. For example, if the claim contained four PEs, the hello message would be printed four times with id numbers 0, 1, 2, and 3 (though not necessarily in that order).

Now the last part is the **constraints** variable, what resources we want in the claim. For example, we could use the following to express that we desire 3 to 6 processing elements.

```
1 val constraints = new PEQuantity(3,6);
```

This is a single simple constraint. The design of the invasive command space is the largest part of the API proposed here. The next section describes its current state exhaustively, but development continues.

5.3. Invasive Command Space

We structured the constraints and hints for invade in an extensible hierarchy. The extensibility is important, because it continues to evolve as new uses for the framework are found and new hardware features turn up. Figure 5.1 shows the full hierarchy.

One of the most used constraints is PEQuantity, which specifies the number of processing elements and was shown in the previous section already. The PEQuantity predicate does not constraint individual PEs, but the set as a whole, so we classify it as a "set constraint". Additional predicates in this class are

- PlaceCoherent, which requires all PEs to be within the same place. This implies shared memory, but will limit the number of PEs depending on the hardware architecture.
- LatencyWithinTeam and ThroughputWithinTeam, which require a certain latency or throughput between all places in a claim. This configures the underlying NoC hardware to provide corresponding guarantees.
- TileSharing, which allows other claims to allocate PEs on the same tile if *both* signal their willingness. This implies that applications share some resources, like bus, tile-local memory, and caches. The idea is improve utilization at the cost of predictability and performance of individual applications.

Next to the constraint class concerning the whole set, there is a class concerning places and one concerning individual PEs[1]. The constraints always apply to each of them. The PlaceConstraints are

[1] My older publications described this as one class PredicateConstraints, which conflicts with the predicate concept in chapter 2.

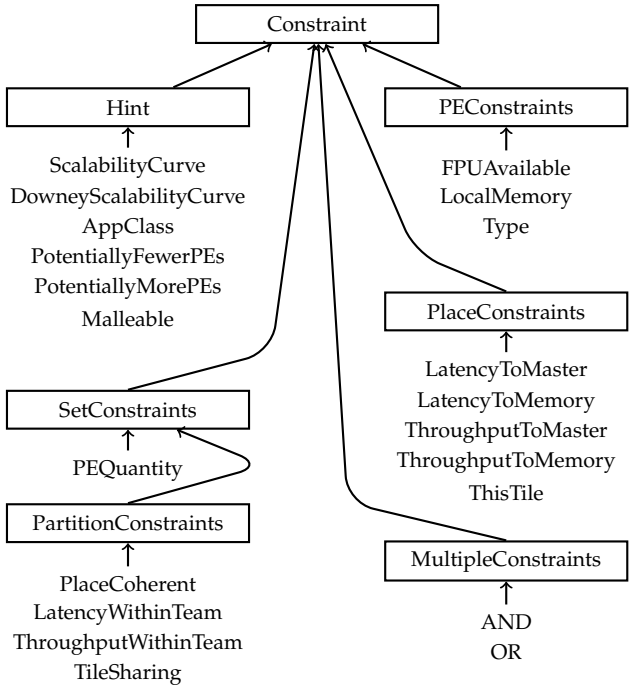


Figure 5.1.: The invasive constraint hierarchy as a class diagram. Edges represent inheritance. Boxed nodes are abstract classes and unboxed ones concrete classes. For clarity not all concrete classes have their own arrow, but the meaning should be intuitive.

- LatencyToMaster and ThroughputToMaster, which require a certain latency between all places in a claim to the place where the claim was invaded from, the master. This is useful in a master-slave architecture, where slaves primarily communicate with the master, but not with each other. This configures the underlying NoC hardware to provide corresponding guarantees.
- ThisTile requires the claim to have one place and it must be on the same tile it was invaded from. This enables underlying communication layers to exploit the shared memory and communicate cheaper. The X10 semantics still provide type and memory safety (without using explicitly unsafe mechanisms).
- TileLocalMemory requires a certain amount of tile-local memory to be reserved for the claim. In contrast to the global DDR memory, this tile-local memory is faster, but there is less of it available.

The PEConstraints apply to each PE in a claim individually:

- Type is necessary in heterogeneous environments. This constraint enforces a claim to be homogeneous. To combine heterogeneous resources into one application, the programmer uses multiple homogeneous claims with different type constraints. Our implementation knows about the type *i*-Core, Leon, and TCPA.
- FPUAvailable specifies if the PE provides hardware support for floating point arithmetic. This constraint is used exemplarily for the wide range of possible CPU extensions. For the HPC programmers in our transregio, it was a surprise that there might be CPUs without an FPU. For the hardware architecture people, it was a surprise that programmers considered it a matter of course.

Another class of constraints are hints, which are about the non-functional aspects of claims. While the allocation model in chapter 2 separates them as two parameters, the implementation merges both under the same abstract class `Constraint`. Hints are for example:

- ScalabilityCurve which describes how an application scales in relation to the number of processing elements.
- DowneyScalabilityCurve is special case of ScalabilityCurve, where the curve is describe by only two parameters A and σ [Dow97]. See more details in section 5.3.1 below.

- ApplicationClass provides a general classification for the scheduler. Each class can be configured internally as a set of weights for different parameters like temperature, power, or cache miss rate. However, on the X10 level only named classes are available. For example,
  - High Performance: Fast execution has priority, so faster cores are preferred.
  - Low Latency: Quickly starting and execution without interruption has priority, so prefer cores with a low load.
  - Low Power: Avoid cores which draw a lot of power (fast ones) and also avoid waking up idle cores.
- PotentiallyFewerPEs and PotentiallyMorePEs give more freedom to the operating system for scheduling. In the case, where two claims are on the same tile (see TileSharing above) one might have idle resources and the other might have a high load. If the hints match, the OS is allowed to cross the claim boundary when scheduling activities. This disturbes the load balancing of the applications, but can increase utilization.
- Malleable which marks the claim as "async-malleable". This is a more complex hint explained in section 5.7. Intuitively, the application promises to resize whenever resource management demands.

Now we also need two meta constraints to compose all those constraints and hints, so there is AND and OR. These are extra classes in the implementation, but due to X10 operator overloading they are usually not visible.

It is easy to come up with more ideas for constraints. For example, one could specify scratchpad size, cache size and type, scheduling, layout, monitoring possibilities and others. This just underlines the need for extensibility. The LCP process requires a motivating example application, though. So far, nobody has seen a practical need for such constraints.

5.3.1. Performance Modelling

If a system continuously optimizes applications with respect to global state, it needs a performance model of the applications. While this work is not concerned with the system's optimization algorithm (see DistRM [KBL+11] and others [ATBS13]), the performance model must be communicated from

applications to the system. At least the application-specific parts of the model. Thus the programmer must know and parameterize the performance model of the system.

The invasive system uses a simple models for two reasons: We want to optimize applications in a scalable way *online* and to keep it easy for the programmer. This means we cannot use complex models, which for example require an UML model of the application [GBL09].

The invasive framework provides a constraint to specify scalability according to Downey's convex curves [Dow97]. To describe a "Downey curve", we need two parameters A and σ. Parameter A is the average parallelism, which intuitively is the upper bound the application scales. For example, if your application can use at most 64 cores, then $A = 64$. Parameter σ specifies the time the application does *not* run with parallelism A. Downey gives the speedup formula for $0 < \sigma \leq 1$ as

$$S(n) = \begin{cases} \frac{An}{A+\sigma/2(n-1)} = \frac{An}{A+(n-1)\sigma/2} & 1 \leq n \leq A \\ \frac{An}{\sigma(A-1/2)+n(1-\sigma/2)} = \frac{An}{\sigma A-\sigma/2+n-n\sigma/2} & A \leq n \leq 2A-1 \\ A & n \geq 2A-1 \end{cases}$$

and $\sigma \geq 1$ as

$$S(n) = \begin{cases} \frac{An(\sigma+1)}{\sigma(n+A-1)+A} & 1 \leq n \leq A + A\sigma - \sigma \\ A & n \geq A + A\sigma - \sigma \end{cases}$$

For $\sigma = 1$, the formula evaluates to the same result. Figure 5.2 illustrates the behavior of the function $S(n)$.

5.4. Constraint Graphs

The above constraint representation has limitations as already hinted at in section 2.3. If you want to mix different resource types, say Leon and *i*-Core PEs, you must invade two different claims. However, claims imply isolation, which makes using these claims unwieldy. While this can be hidden in frameworks, like ActorX10 [RPS+16], we can envision a more complex situation, which

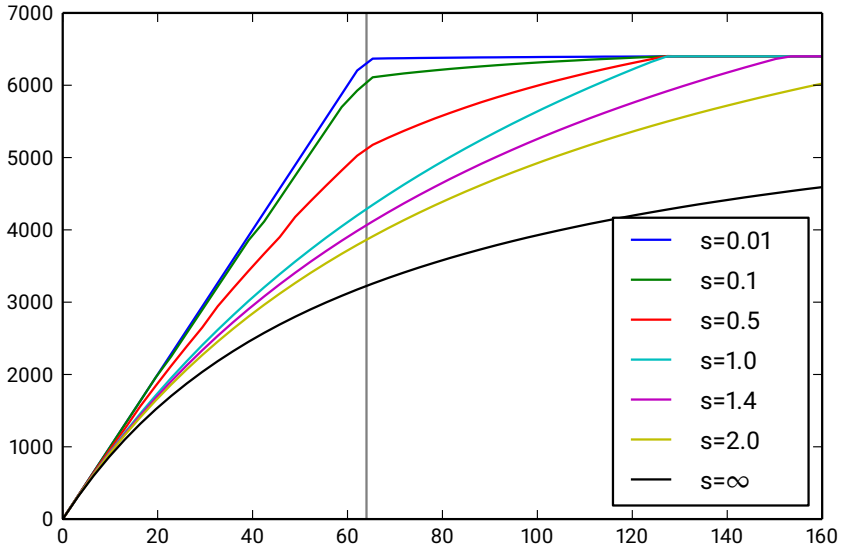


Figure 5.2.: Examples of Downey Curves with $A = 64$ with n on the x-axis and $S(n)$ on the y-axis. Each line shows a different σ. For $\sigma \simeq 0$ the curve rises linearly until A and is flat afterwards. For a higher $\sigma \leq 1$ the "knee" at $n = A$ (vertical line) is lower. For $\sigma \geq 1$, $S(n)$ reaches parallelism A only after $n = 2A$. The higher σ, the later. This graph resembles Downey's figure 2 [Dow97]. The y-axis is multiplied by 100, because our system relies on integer arithmetic.

cannot be represented at all: Have two different resources and guarantee a certain NoC latency between them. There is no way to express relationships between claims, so we need a way to express this as a single claim.

A proposed solution are "constraint graphs" [WBB+16], which represent the desired resource structure as a labeled graph. A node represents a tile and is annotated with resource types and counts. An edge represents a NoC connection and is annotated with a maximum distance and a service level. This allows for a very specific description. The tradeoff is that generic descriptions, e.g. including a scalability curve, cannot be represented. Thus, constraint graphs are not more or less powerful than the hierarchy constraints from the previous section, but useful in different situations.

This approach is applicable, if the program architecture starts with a static actor model[2]; A common technique in embedded and real time programming. For an example, consider a video decoder, which has a pipeline structure. At design time, we consider CPU speeds, memory sizes, communication bandwidth, and other factors to find a configuration which meets our requirements. Such a configuration can be specified as a constraint graph, as illustrated in fig. 5.3.

Our invadeX10 framework allows to model constraint graphs as data structures and directly use them for resource allocation. We pursue the illustration above further in fig. 5.4. We pack each constraint graph together with "quality numbers" derived from requirements, like frame latency or power consumption, into an "operating point". The invade method is overloaded to also take a list of operating points, as illustrated in fig. 5.5. Resource management will then pick one of the operating points and return a claim with resources for it. For deriving operating points from requirements, we use a design-time application analysis and runtime mapping [WGW+14] approach.

5.5. Invasion and Retreat

Invasion is the allocation of resources, which most prominently means processing elements. The static invade method of the invasic.Claim class gets a Constraint object as argument and returns a claim object. Internally, this spawns a new agent, which communicates with other agents to bargain for claim resources. This means that this is a potentially expensive operation in terms of run time.

[2] This is in no relation with "actor claims" from chapter 2. Neither is this about the π-calculus, because we do not allow the dynamic creation of actors, indicated by the "static" prefix.

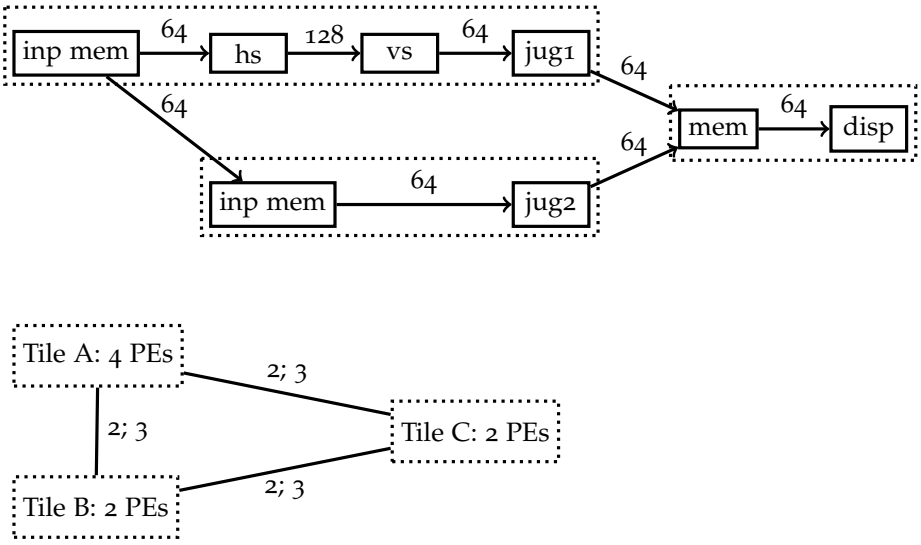


Figure 5.3.: At the top, a Picture-in-Picture pipeline commonly used in NoC research [BJM+05]. Nodes are algorithm steps and edges are communication channels with annotated bandwidth. The nodes are grouped according to their placement on tiles. The implied constraint graph is below. The bandwidth is used verbatim, the processing elements required is the number of steps in the corresponding group. The edges have a hop distance and a service level annotated, instead of explicit bandwidth requirements.

```
1  val ag = new ActorGraph();
2  val inp1 = new Actor(act_inp1);
3  ag.addActor(inp1);
4  val hs = new Actor(act_hs);
5  ag.addActor(hs);
6  ag.addChannel(inp1, hs);
7  // further actors and channels ...
8
9  val constraintGraph = new ConstraintGraph(ag);
10 val a = constraintGraph.addCluster(1, [inp1, hs, vs, jug1], new RISC());
11 val b = constraintGraph.addCluster(2, [inp2, jug2], new RISC());
12 val c = constraintGraph.addCluster(3, [mem, disp], new RISC());
13 constraintGraph.addInterconnects(a, b,
14                         2 /* hop distance */, 3 /* service level */);
15
16 val qualityNumbers = new Set[QualityNumber]();
17 qualityNumbers.add(Latency(4, "ms"));
18 qualityNumbers.add(Power(10, "MW"));
19
20 val op = new OperatingPoint(constraintGraph, qualityNumbers);
```

Figure 5.4.: Constraint graph and operating point example from fig. 5.3 in X10 code. This is integrated with the ActorX10 framework, so we model an actor graph first. Then we construct the constraint graph and define quality numbers for latency and power consumption. Finally, an OperatingPoint object is created.

```
1  //Create Binding from operating points.
2  val claim = Claim.invade(operatingPoints);
3  val binding = new Binding(claim, operatingPoints);
4
5  //Execute the task graph.
6  binding.infect();
```

Figure 5.5.: Assuming a list of operating points like in fig. 5.4, this code demonstrates invasion. The binding classes infect method inspects the claim to find out which operating point the agent system chose. Then it extracts the actor graph from the correct operating point, deploys the actors and runs them.

```
1 val claim = Claim.invade(
2     new PEQuantity(1,8) &&
3     new Type(PEType.RISC) &&
4     TileSharing.WITH_OTHER_APPLICATIONS &&
5     new DowneyScalabilityHint(5,100));
```

This invasion returns a claim containing between one and eight RISC PEs. The claim might share its tiles with other claims featuring the same hint. It scales according to a Downey curve [Dow97] with $A = 5$ and $\sigma = 1.00$, which roughly means "nearly linearly up to 5 cores". We explicitly avoid using floating point values (e.g. 1.0 instead of 100) because it must be possible to use the API on cores without floating point support. While floating point emulation in software (soft float) is possible, there is not really a need to accept that inefficiency.

If the agent is unable to get the minimum amount of resources to fulfill the constraints, a NotEnoughResources exception is thrown. This is an exceptional situation as programmers should write programs, which can run with few resources as well.

```
1 try {
2     val claim = Claim.invade( new PEQuantity(1000000) );
3 } catch (e:NotEnoughResources) {
4     Console.ERR.println("Could not get all the PEs");
5 }
```

If no exception occured, the resources are now guaranteed to the application *exclusively*. No other claim/application is able to use them and the resource set will not change unless the program explicitly allows it. However, this guarantee does not cover catastrophic failures, for example if the hardware burns. Some possibilities to hide hardware failures are possible, like redundant resources and transparent migration, our implementation currently does not provide any of this, so the promise only holds according to the abilities of the resource management.

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

```
1 val count = claim.size();
2 val places = claim.places();
3 val pes = claim.processingElements();
```

```
4 if (pes.get(0).getType() == Type.RISC) { ... }
```

This allows to program to adapt its data structures and control flow to the resource situation dynamically. For example, places with more PEs can be given more data, as higher parallelism equals more computing power.

A retreat frees all resources of a claim, when the job is done. The claim object is invalid afterwards, so the programmer must not call infect for example.

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

5.6. Explicit Reinvasion

While it is possible to retreat claims and invade anew, there is a reinvade method for adapting claims, which is more efficient. It will (constraints permitting) keep the resources already in the claim.

For many applications it only makes sense to adapt resources at certain points. For example an iterative algorithm can do it between iterations. See chapter 6 for a realistic example. For this use case reinvade can be called without an argument, which signals to the agent that at this point it may adapt resources. The agent can add or remove resources depending on the load of the system, but will always respect the constraints previously set. This call should be cheap in terms of overhead, assuming that the claim will mostly *not* change, so the application can call it often and the system gets opportunities for load balancing.

For a quick feedback, infect returns a boolean, which is true iff the claim resources changed. The application can inspect the claim (as described above) to recalibrate itself to the changed situation.

```
1 val changed = claim.reinvade(); // optimize resources
2 if (changed) adaptMyself();
```

Alternatively, you can also use reinvasion to change the constraints by giving a constraints parameter to reinvade. This call variant can be cheap, if the resources currently in the claim fulfill the new constraints. The agent may return the claim unchanged and optimize concurrently until the next reinvasion.

```
1 val claim = Claim.invade(new PEQuantity(5,10);
2 // ... after a while, we could use more PEs ...
3 val changed = claim.reinvade(new PEQuantity(5,20);
```

There is a perilous situation with respect to retreating from resources: A claim often spans across multiple tiles and claims are isolated from each other (see section 5.9 below for details). Thus, if a claim retreats from all PEs on a tile, the tile becomes inaccessible and all data there is lost. To err on the side of safety, the agent system will by default never free a tile completely. This can break the quantity constraint. For example, you have a claim with 4 PEs on 4 different tiles and you reinvade with PEQuantity(2), you will still have 4 PEs, so you do not lose the data on (at least) two tiles.

For a clean solution, we introduce a "resize handler", which are methods the application can register at a claim.

```
1 val claim = Claim.invade(constraints);
2 val resizeHandler = (add:List[PE], remove:List[PE]) => {
3   // do something
4 }
5 claim.register(resizeHandler);
```

When resources of a claim change, it happens in this order:

1. Agent decides to add or remove claim resources.
2. Agent adds resources.
3. Resource handlers are called and they all return.
4. Agent removes resources.

When the resize handler executes, it has access to the union of all resources before and after the change. This enables the handler to move data *from* tiles the claim is about to lose and *to* new tiles. See chapter 7 for a realistic case.

This mechanism is surprisingly hard to implement correctly, because it interacts with the infect method (described in detail in section 5.8). First, there is no requirement that infect cannot run multiple times in parallel, although it makes no sense. Second, the resize handler can be called when *no* infect executes. As there is no generic solution to this dilemma, the problem is left to the user: The resize handler must synchronize itself with the rest of the application.

Still, there is another problem. X10 activities form a single tree and you can use finish to wait for the termination of a whole subtree. The activity that executes the resize handler is called by the agent system, thus outside this tree. If we use finish to wait for the termination of infect, we miss activities started by resize handlers. Requiring the user to wrap all infect calls with additional synchronization is tedious. Instead, infect implements its own finish mechanism and the resize handler, knowing the claim it belongs to, registers its activities there.

This registration is subtle, because while the resize handler runs infect may return or start or not run at all. Fortunately, we are free to modify the X10 runtime. To implement finish, there is a finish state data structure, which primarily contains a semaphore to count the sub-activities. Whenever an activity or sub-finish-state is created, they register at their parent finish state. Within infect, we can store a reference to the parent finish state in the claim. The resize handler could use this reference to register its own sub-finish-state and the usual X10 runtime mechanisms would work as always. An unsolved problem is that the resize handler can only use the claim object, where it is registered at, but claim objects get copied, for example due to at.

5.7. Reinvasion from External Trigger

While section 5.3 describes all constraints, this section focuses on a special one. The Malleable hint marks a claim as "async-malleable", which means the agent system can reallocate its resources at any time, especially due to claim-external events. For example, if another claim retreats completely, an async-malleable claim can instantly take and use the resources without waiting until an explicit reinvade call. The alternative would be that some resources idle, which is not necessarily bad since the system knows about their state and could power them off. Still, it is preferable to make good use of resources and Malleable opens up more possibilities.

The term "malleable" was defined by Feitelson and Rudolph [FR96]. They worked on job schedulers for parallel supercomputers and classified jobs as shown in fig. 5.6. The intent of their work was a convergence of job scheduling on supercomputers, because schedulers used different optimization goals and techniques and often without documenting their assumptions. For a general framework for scheduling, the framework must support all four cases and more aspects further detailed in their paper.

who decides number	when it is decided	
	at submittal	during execution
user	Rigid	Evolving
system	Moldable	Malleable

Figure 5.6.: Reproduced "table 2" by Feitelson and Rudolph [FR96]. One question is *who* decides about the hardware parallelism, namely the user or the system. The second question is *when* the decision is made, namely at submission (before the application starts) or during execution.

The Invasive Computing approach cannot be classified in this schema, because user and system work together. The user sets constraints and informs the system, which then ultimately decides but constrained by the user. Are we in the column "during execution", at least? Yes, but every application which uses reinvasion is. The schema does not distinguish between applications that adapt when it is convenient (at the next reinvade call) and application that can adapt immediately (without waiting for a reinvade call). Thus, we use the term "async-malleable" for the later case.

An example, which is malleable, but not async-malleable is the multigrid application in chapter 6, a typical iterative numerical algorithm. It uses reinvade to modify the resource claim at specific points during each iteration. If we run multiple instances of the malleable multigrid solver concurrently, we can observe an improved throughput compared to concurrent non-malleable multigrid instances [BRS+13]. Still, resources may idle and wait for the next reinvade call. We want to avoid that kind of waste as well.

The Malleable constraint was the initial reason to introduce the resize handler concept and earlier work [BMZ15] registered the resize handler as parameter of Malleable. However, it turned out the resize handler is useful even without Malleable, so it became separated as described in the previous section.

How useful is this technique, if it only applies to certain applications? Any application following a master-slave pattern has the option to add or remove slaves, usually at a low cost. For example, it can be used inside a job-queue framework [Böw15] and on top of this applications like numerical integration can be implemented as described in chapter 7.

```
1  val ilet = (id:IncarnationID)=>{
2    for (job in queue) {
3      if (queue.checkTermination(id)) break;
4      job.do(); } }
5  val constraints = new PEQuantity(4,10)
6    && new Malleable()
7    && new ScalabilityHint(speedupCurve);
8  val claim = Claim.invade(constraints);
9  queue.adaptTo(claim);
10 val resizeHandler = (add:List[PE], remove:List[PE])=>{
11   for (pe in add) queue.addWorker(pe,ilet);
12   queue.adapt();
13   for (pe in remove) queue.signalTermination(pe); }
14 claim.register(resizeHandler);
15 claim.infect(ilet);
16 claim.retreat();
```

Figure 5.7.: **Example of a async-malleable invasive application** with a master-slave structure and a global queue of jobs. First, ilet states the actual computation to perform. Then constraints describes a malleable claim of 4 to 10 PEs and a speedup curve defined elsewhere. The invade call returns a resource claim and queue gets a chance to internally adapt itself to the claimed resources (e.g. for work stealing). Next, resizeHandler defines how to handle resource changes. Then infect starts an *i*-let on each PE of the claim, which execute jobs from the global queue. Finally, we retreat the claim to free the resources.

Also, you can use async-malleable for parallel search algorithms and sorting is frequent task. Flick et al. [SSF13] have enhanced multi-way merge sort with malleability. Multi-way merge sort is a popular parallel sorting approach, which contains a phase where it partitions the data into work packages that it sorts internally and independently of each other. They use a central work queue for sorting jobs and a number of worker threads that fetch jobs from the queue. Workers can be added and removed without interrupting other workers. If the work packages are small enough, the sorting process as a whole becomes malleable. Moreover, with small work packages, response to reconfiguration requests is nearly instantaneous. The authors measured comparable performance to Intel's TBB and others on an unloaded system. However, when the system is under load from other applications, malleability offered better performance.

In an async-malleable invasive application, the system can decide at any time to resize a claim. The system is only required to meet the invasion constraints. For example, requiring an async-malleable claim of 4 to 10 PEs, means the system can resize to 5 or 9 PEs at any time. In contrast to normal claims, an async-malleable claim must be adaptable even within an active infect phase. The application is responsible for starting and terminating *i*-lets with respect to added and removed PEs. Figure 5.7 shows an example use.

5.8. Infection

Infection is the phase, where resources in a claim are actually used. Since claims are implicitly an isolation mechanism, we cannot use the X10 mechanisms (async and at), but we need an extra method. Since in most cases, we want to use all the resources, the infect method implicitly distributed across all PEs, as you have seen in section 5.2.

The infect call executes one activity per processing element. The programmer is free to create additional activities using async or communicate between places within a claim using at.

The infect method is overloaded to support the map-reduce pattern for convenience. So, if the *i*-let closure type has a non-void return type, the infect method will collect the returned values.

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

Here infect returns an array containing all values returned by *i*-lets running in parallel. This means the length of the ret array equals the size of claim.processingElements(), since each PE executes exactly one *i*-let. For example, if the *i*-let returns an Array[int], then infect returns an Array[Array[int]].

It is also possible to provide a reduce function, which combines returned values into one.

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

In this example, infect returns an int, which is the sum of all returned values, which depends on the number of PEs in the claim.

5.9. Adapting X10 Semantics

In normal X10, all places are available at all times. In constrast, with Invasive Computing places appear and disappear. This requires a more dynamic X10 [BBMZ14]. It is still convenient, if Place.places() returns a sequence of all available places. Thus we need to adapt this method for more dynamicity. It constructs a sequence instead of returning a static one.

A little bit more adaptation is necessary for fields like Place.MAX_PLACES and PlaceGroup.WORLD, because they cannot be fields anymore. Instead, you must call a method Place.getMaxPlaces() now.

Even more subtle are changes to code, which implicitly assumes a constant number of places. One fundamental building block in X10 is PlaceLocalHandle, which enables the programmer to dynamically create per-place variables. This requires initialization at creation in every place and is done eagerly. When additional places appear, each place local handle must be initialized before it can be accessed. Instead of using eager initialization, we switch to a lazy model. The tradeoff is that we must first check for initialization on every access. However, every access already requires multiple indirections anyways, so the overhead is negligible.

Still, we were unable to make the adaption completely transparent for application programmers. In various cases application code must be adapted. If the programmer uses a DistArray, the data must be redistributed from removed to additional places, which is potentially a costly operation, so the programmer should have control over that. For transparency, every DistArray would have to register at its claim and add bookkeeping overhead for little gain.

The X10 core developers did similar work themselves under the project names Resilient X10 [CGH+14] and Elastic X10. Resilient X10 considers supercomputers where compute nodes sometimes fail. In the X10 language this means places can disappear and their extension throws an exception to notify the programmer about such failures. In a dual manner, Elastic X10 adds the possibility for places to appear in a cloud scenario, where administrators add additional servers. Use both approaches together, then resources in terms of places can be added and removed from a running application. This plays into the strategy

change of X10 development: Away from high performance computing towards cloud computing. The user can add and remove resources who rents more or less compute resources for his servers.

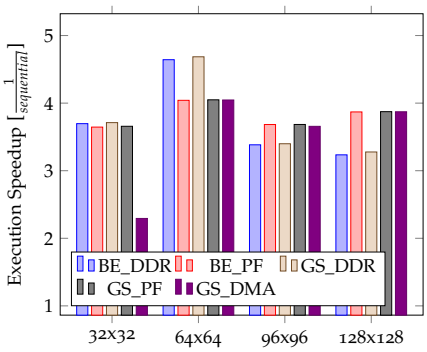
In contrast, our framework makes a more radical paradigm change still. Applications can give information to the system what and how many resources are desired. In addition, more kinds of resources than places are handled: From low level bus throughput and latency guarantees over single processing element types to memory requirements. There is also no concept of exclusive or shared resource use, which is relevant for performance.

5.10. Invading Communication Resources

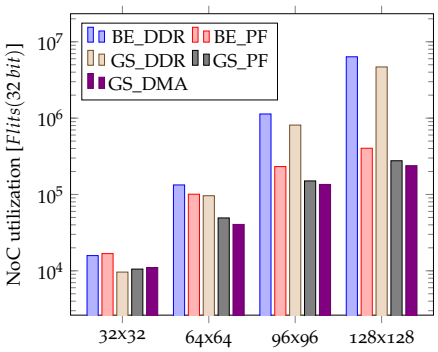
This thesis describes a *resource*-aware framework and paradigm, but mostly we consider processing elements as the sole resource type. In fact, it is only resource supported by the implementation. Still, we want to enable the management of other resources as well. This section is about communication resources; More precisely about bandwidth and latency of the network-on-chip (NoC) which connects tiles and is used for message passing between X10 places. While the framework implementation does not support it yet, we designed the interface and implemented the lower-level mechanisms. Some parts are missing the operating and resource management system layers, though.

We investigated explicitly invading connections on the NoC together with Heißwolf et al. [HZZ+14]. Compared to a bus protocol in a multicore CPU, NoCs are more complex and need more energy. Unfortunately, real power consumption could only be measured by building an ASIC, which is out of scope. For estimating the energy need, we synthesized a small system, derived toggle rates from a netlist simulation, and came up with a power consumption of 7.94 mW for an idle router (P_{idle}). While there are ways, like power gating, to reduce P_{idle}, there more interesting number is the energy used for actually communicating data. Hence, we report $P_{data} = P_{total} - P_{idle}$.

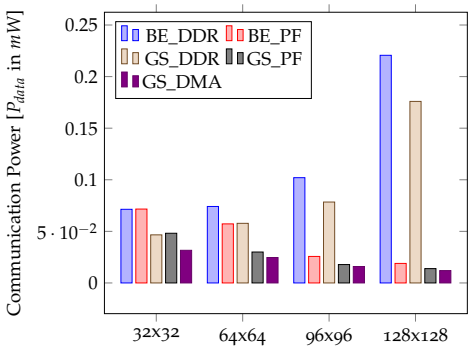
An FPGA-prototype of an architecture with four processing tiles was realized. The prototype has only one Leon3 RISC core per tile due to the limited amount of resources available on the used ML605 FPGA board. Each tile has a tile local memory of 256 kB used to store the executable and frequently used program data. The cache hierarchy per tile consists of a 512 B L1 data- and instruction



(a) Application speedup with four tiles compared to using a single one.



(b) NoC utilization correlates with matrix size.



(c) NoC power consumption correlates with utilization and reductions up to 96% are visible.

Figure 5.8.: Parallel matrix multiplication executed with different settings on a 4 tile prototype of the clustered architecture. Figure by Heißwolf et al. [HZZ+14].

cache and a 4 kB L2 cache. The prototype can be revered as heterogeneous since one of the four tiles has a DDR3 memory attached to its front-side-bus that can be accessed from the other tiles via the NoC.

As a test application, we used an integer matrix multiplication with different matrix sizes. Two versions only use "best effort" (BE) communication without invading resources explicitly. Three versions exploit the "guaranteed service" (GS) connections. For each category, there are two sub variants: One uses off-chip DDR3 memory directly (DDR) and the other prefetches to tile local memory (PF). Additionally, our hardware supports DMA transfers managed by the NoC, which makes the prefetching more efficient. Due to hardware-constraints a DMA together with BE is not possible. This leaves us with five variants to compare in fig. 5.8:

1. *BE_DDR*: Source matrices are located in the main memory, best effort communication is used.
2. *BE_PF*: Required parts of source matrices are prefetched to tile local memory by software, best effort communication is used.
3. *GS_DDR*: Source matrices are located in the main memory, GS connections are established for communication.
4. *GS_PF*: Required parts of source matrices are prefetched to tile local memory by software, GS connections are established for communication.
5. *GS_DMA*: Required parts of source matrices are prefetched to tile local memory by hardware DMA, GS connections are established for communication.

Figure 5.8(a) compares the speedup of the different variants relative to a single core variant executed on the tile that attaches the DDR-controller. The results show that prefetching has no benefit with respect to execution time for small matrix sizes due to the fact that all data fit into the L2-cache. A matrix with *64x64* elements even reaches speedups higher than four due to the fact that the overall cache size is increased compared to the single core variant. If the matrix sizes become bigger prefetching improves performance. For a matrix of *128x128* elements prefetching introduced by CAP improves performance by 26% compared to the reference *BE_DDR*. Figure 5.8(b) shows the NoC utilization caused by executing the variants of the matrix multiplication. To obtain these numbers we used the NoC link monitors available on the prototype. The results show that the amount of communication is proportional to the matrix size.

For larger matrix sizes the amount of flits can be reduced by 26% if resource allocation in the form of GS connections is used without prefetching. DMA prefetching (*GS_DMA*) can reduce the amount of communication by up to 96% compared to the reference *BE_DDR* for the largest matrix size. Finally we analyze the power consumption that is directly related to data transmissions for an ASIC implementation of the NoC. Figure 5.8(c) summarizes the results. The benefit of prefetching with respect to the power consumption grows with the size of the matrix. The *GS_DMA* variant, that applies all of the proposed CAP mechanisms, can reduce P_{data} by up to 95%. If no prefetching is used the allocation of communication resources (GS) can help to reduce the power consumption by up to 20% depending on the matrix size.

In summary, we measure up to 96% less communication and 95% less transmission power. Additionally, we see a speedup of up to 26% for the matrix multiplication benchmark. These results suggest that exploiting communication resources is in fact worthwhile and should be incorporated into a resource-aware paradigm.

5.11. Compiler Integration

The research project included building a custom compiler. At first, engineering work was necessary to port X10 to the invasive platform, which means modified SPARCv8 processors and a custom operating system. Additionally, the adaptations from section 5.9 change fundamental assumptions of the X10 runtime. While all those aspects could have been solved with the existing C++ backend of X10, we also investigated optimizations, which require deeper compiler support. We used the libFirm library [BBZ11, BBH+13] for its dependency-graph structure, optimizations, and analysis capabilities. For code generation we added code generation for SPARC. Work on register permutation [BMR15a, BMR15b] and data transfers [MT17] was enabled by using libFirm. The work on the memory model (chapter 4) is directly motivated from the compiler engineering, since the compiler has to map from the source language's memory model to the hardware's memory model, which requires the compiler to know both. This is important for a language like X10, which is explicitly designed for parallel execution.

When we started our project, X10 provided two backends. The "managed backend" translates X10 to Java code and then uses javac and the JVM. Historically, this was the first backend as X10 was built with the Polyglot frame-

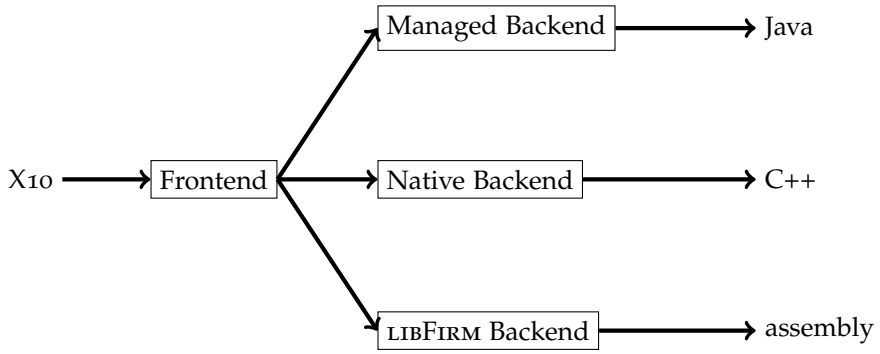


Figure 5.9.: The X10 compiler features three backends producing Java, C++, or assembly output from X10 input. Optimization happens in the frontend and in the libFirm backend. The managed frontend delays optimization to the execution phase on a JVM and the native frontend relies on the optimization of gcc compiling C++ to assembly.

work [NCM03], which enables easy modifications to Java. Later, the need for speed prompted the development of the "native backend", which translates X10 to C++ code and then uses gcc. In HPC code it is essential to provide an "array of structs" data structure, which is not possible on the JVM [3]. For our work, we added a third backend by reusing libFirm compiler toolkit. This backend generates assembly and then only uses assembler and linker from gcc. Figure 5.9 illustrates the three backends.

While the X10 frontend and the libFirm backend already existed, integrating still took years. One chunk was the support for generics in X10. Both existing output formats, Java and C++, support generics[4], but assembly does not. We chose to use the C++ method of template instantiation, instead of the Java type erasure approach, because it allows to optimize for different type parameters. For example, an array of int and an array of double should not use the same assembly code. Instead, the compiler must be enabled to do strength reduction and other optimizations for loops to improve performance. Thanks to our student Eduard Frank for implementing most of the template instantiation parts.

[3] There is the ongoing project Valhalla to add value types to Java, but it has not been successful so far.

[4] C++ templates are a superset

Another essential optimization is inlining. While inlining is essential for all optimizing compilers, it is especially significant for our backend, because there are no primitive data types in X10. Instead, even int is declared as a struct and provides native methods for arithmetic. The managed and native backends use string templates to insert primitive operations into the Java and C++ output, respectively. For assembly output, we cannot annotate assembly strings to insert, because they would have to be architecture specific. Instead, we implemented these methods as C functions in the runtime. Obviously, adding function call overhead to arithmetic operations in tight loops is disastrous for performance. The tricky aspect about this is that the C runtime and the X10 code are in different compilation units, which means they are compiled separately. What our compiler does is a limited form of link time optimization. The C runtime is fed into cparser, a C compiler with libFirm backend, which outputs the intermediate representation (IR) instead of assembly. The X10 compiler imports that IR code into the X10 compilation unit. Specific care has to be taken here to match types. For example, the C int must be matched to the X10 int. After the merge, inlining can remove the function call overhead for arithmetics. Not only arithmetics profit, though. The compiler can inline other parts of the C runtime; For example, inter-place communication, serialization, and memory management. We thank Matthias Braun for implementing the lion share of our link time optimization trick.

A third feature we added to our compiler was support for exceptions. We use the zero-cost approach [HMP97], which has no overhead in the absence of errors, but is slower when an exception is actually thrown. This matches with the intuition that exceptions are exceptional and rare. For regular function calling, exceptions have no overhead and can be ignored. For each try block, the catch and finally blocks are registered in a special section and referenced by the activation record (also known as "call frame" on the call stack). When the application throws an exception a generic handler walks the call stack looking for a matching handler and executing finally blocks during the walk. Thanks to our students Julian Oppermann and Jonas Haag [Haa16] for implementing most of the exception support.

The three described aspects were distinct blocks of work, we had to do for a working compiler. Of course, much more had to be implemented, like SPARC code generation, the message passing runtime, integrating a garbage collector, or porting the standard library. The core team includes (ordered by number of commits to x10i) Matthias Braun, myself, Manuel Mohr, and

Sebastian Buchwald. While this was engineering work, some of our scientific work [MT17, BLU16, MBZ+15, BMR15a, BMR15b, Buc15] is directly inspired from this.

Resource-awareness is inherently architecture specific in the details. While it allows high-level programming, the ability to optimize for certain platform features is essential. Likewise, compilation specifically targets a platform. Thus, a framework for resource-aware computing naturally belongs into the runtime of a compiler.

5.12. Framework Offsprings

Our framework has already spawned a few offsprings, which port the ideas of Invasive Computing into more mainstream environments. This section reviews and contrasts them.

The High Performance Computing community has published some works in the same general direction and our ideas fall on fertile soil. While we focus on many-core architectures, HPC targets supercomputers. Still, there are many similarities, like the lack of consistent shared memory and the use of message passing, which makes the programming of such systems quite similar.

Adaptive MPI [HLK04, HZKK06] uses processor virtualization to achieve malleability. Computations are divided into a large number of *virtual* MPI processes, which are mapped to the physical resources. A runtime system can optimize this mapping using object migration. This technique allows for adaptive overlapping of communication and computation, automatic load balancing, flexibility of running on arbitrary number of processors, and checkpoint/restart support. In contrast, Invasive Computing assigns resources exclusively and programs cannot rely on resource virtualization to hide underutilization. Features like load balancing or checkpointing must be implemented on top.

Leopold et al. [LS06] present a case study of making an existing MPI-2 application malleable. The studied application is an iterative numerical simulation using a master-slave design. New slaves become usable to the master starting with the next iteration. They did not handle (but investigate) removal of slaves due to limitations in MPI. In another paper [LSB06], they used a global water prognosis application and parallelized it with MPI and OpenMP. They also

```
1  int main() {
2    Claim claim;
3    int sum = 0;
4
5    claim.invade(PEQuantity(1,3));
6    std::cout << claim.toString() << std::endl;
7
8    // Parallel for-loop with given resources
9    #pragma omp for reduce reduction(+:sum)
10   for (int i=0; i<100000; i++)
11     sum += 1;
12
13   claim.retreat();
14   std::cout << claim.toString() << std::endl;
15 }
```

Figure 5.10.: iOMP example code from listing 2 in [GHM+12]. The API for invade and retreat is similar to the X10 API. There is no infect, as the conventional OpenMP (pragma omp) mechanisms are used instead.

made it malleable, but provide no performance evaluation thereof, because "the performance gain from malleability is difficult to quantify, as it depends on the cluster load."

Maghraoui et al. [EMDSV07, EMDSV09] present a general approach for extending existing iterative MPI applications with malleability features. The authors broaden the definition of malleable to also vary the number of application processes, so they should rely not only on dynamic load balancing via resource virtualization. Implementation-wise, they extended the middleware library PCM (Process Checkpointing and Migration) with functions regarding malleability. In particular, they support splitting and merging processes via collective operations.

The work so far was conducted outside of Invasive Computing. Within the transregio, we also investigated ways to bring resource-aware programming to mainstream platforms.

5.12.1. Invasive OpenMP

OpenMP is a widely used API for shared-memory parallel programming in C/C++. It is integrated into many compilers, like the GCC. One project of Invasive Computing ported the concepts to OpenMP and made them available to a wider audience of HPC programmers. Implemented in iOMP [GHM+12], it provides a subset of the invasive command language in C++. Figure 5.10 shows example code. Only the quantity of processing elements can be requested and claims provide no isolation. Still, `PEQuantity` is modelled in a hierarchy, so it would be easy to extend applications with more constraints.

In a test scenario similar to chapters 6 and 8, where multiple applications were started at different times, the overall throughput improved while overhead was insignificant assuming enough computational load. The conclusion is that "Overall machine throughput can be drastically improved by dynamically distributing the cores to concurrent applications".

5.12.2. Invasive MPI

Comprés at al. [CMHGB16] ported the resource management of Invasive Computing to MPI. In MPI with version 2.0 dynamic processes were added to the standard, although it is not widely used, because of performance and limitations. The adaptation pattern of the proposed MPI extensions is shown in fig. 5.11.

This work does not support that applications specify constraints and hints for their resources yet, so an essential part of resource-aware programming is still missing. Still, the paper concludes that while the work to modify an existing application into a resource-aware variant is "not trivial, the benefits to system wide performance as well as the performance of the application itself may turn out to be worthwhile."

5.12.3. Communicating Thread Pools

In the world of Java, thread pools are a well-known tool for dynamic load balancing. As we can transfer the ideas of Invasive Computing into mainstream concepts, Tobias Weiberg explored this in his bachelor thesis [Wei14]. He made drop-in replacements for `java.util.concurrent.ThreadPoolExecutor`. In a scenario

```
1  void mpi_adapt(void) {
2    MPI_Probe_adapt(...);
3    if (current_operation == ADAPT_TRUE) {
4      MPI_Comm_adapt_begin(...);
5      // Adapt to changes
6      MPI_Comm_adapt_commit(...);
7    }
8  }
```

Figure 5.11.: MPI_Adapt from algorithm 1 in [CMHGB16]. The Probe_adapt checks if adaptation is necessary and is well optimized, so it can be called often with negligible overhead. The adapt_begin call contains the most work, as it starts new processes and performs most of the management. Finally, adapt_commit is a barrier synchronization and some minor bookkeeping before the computation can continue.

where multiple thread pools run on the same multicore system, dividing the cores into disjoint sets for each thread pool, we can measure speedups due to less contention and context switching. The speedups vary between 0 and 50%. This means it very much depends on the circumstances if this is worthwhile. However the overhead is neglible, so apart from the additional complexity behind the scenes, there is no downside for communicating thread pools.

Now, the framework is documented and we can evaluate its advantages. The next chapters present different case studies with realistic applications.

Case Study: Invasive Multigrid

We are dwarfs astride the shoulders of giants,
In the pursuit of that which we show our reliance.
For each step a thousand mistakes are made,
We learn from each with knowledge's aide.

— /u/justafeather

6.1. The Multigrid Application

A program we use in various benchmarks [BRS+13, BBMZ14] is a time-dependent heat dissipation simulation. A laser engraves symbols on a metal plate, which heats up in the process. The basic idea is to have a 2D array of temperature points of the plate. In each simulation step they adjust the temperature depending on their neighbor points. We get a dynamic behavior during computation of a single time-step due to the multigrid method we use. Multigrid is a well-known technique to speed up convergence. From time to time you transform your array into a smaller, more coarse array and back. Naturally, for smaller arrays you need less resources, which means the load for the system is periodically lower. One such dip is called a v-cycle and is illustrated in fig. 6.1.

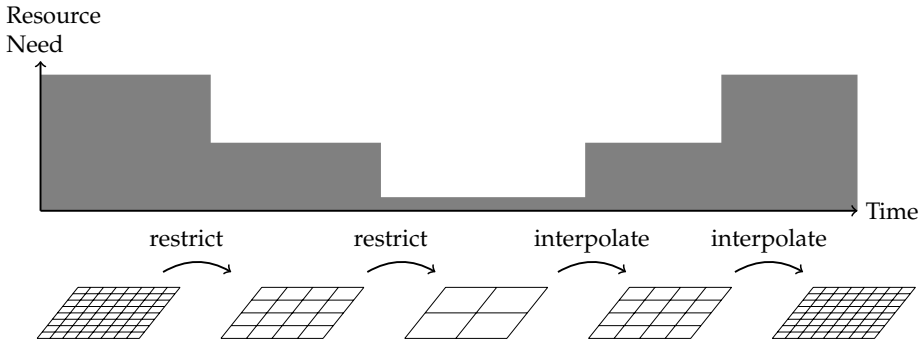


Figure 6.1.: An illustration of the v-cycle in the multigrid method. From a fine grid, the data is restricted to a more coarse representation and then interpolated back into the finer data. The more coarse, the less compute resources are necessary.

Most of the implementation was done by Martin Schreiber. The description in section 6.1.1 was primarily written by him [BRS+13]. We talk about a specific implementation and the multigrid method is the dominating aspect, so we refer to this application as "Multigrid". It consists of nearly 3000 lines of X10 code (including comments etc). My contribution was to implement the resource-aware framework and together we developed the data redistribution parts of the application.

Using our invasive framework, we can free resources during a v-cycle, so *other* application can use them more productively. In this chapter we focus on the application itself and chapter 8 investigates a multi-application scenario.

6.1.1. Problem formulation and discretization

We continue with a short description of the mathematical formulation of our problem as well as its discretization and refer to [BHM00] and [TOS01] for detailed information on multigrid solvers.

The heat distribution over time in an isotropic material is given by the following equation:

$$\frac{dT(x,y,t)}{dt} = \alpha \Delta T(x,y,t) + E(x,y,t), \quad (x,y) \in \Omega.$$

This equation describes the temperature distribution T, an external energy input E (a laser hitting the material) on our domain $\Omega = [0;1]^2$, and the thermal diffusivity coefficient α. We set the boundary conditions to 0 implementing homogeneous Dirichlet boundary conditions $T(x,y,t) = 0, (x,y) \in d\Omega$.

For spacial discretization we store the temperature values at $N \times N$ grid-points for each discrete grid-point (i,j) to a 2D array. The Laplace operator is discretized by the stencil approximating the 2nd partial derivatives

$$\Delta \approx \frac{1}{h^2} \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

with the mesh-width h. The stencil is applied by computing the discrete convolution using this stencil as the convolution kernel.

For discretization of the time-stepping, we approximate the temperature distribution with 1st order forward differences and use an implicit update scheme

$$\frac{T(x,y,t+\Delta t) - T(x,y,t)}{\Delta t} = \Delta T(x,y,t+\Delta t) + E(x,y,t).$$

This leads to the system of linear equations $\mathbf{A}\vec{x} = \vec{b}$ representing the implicitly discretized heat-equation. The external energy is handled by appropriate updates of $\vec{b}$. The approximated solution for $T(x,y,t)$ is then given in $\vec{x}$ after computation of an approximated solution of this system of equations.

Since solving such a systems of linear equations by using direct solvers — e. g. Gauss-elimination— would destroy the sparsity pattern of the matrix, we employ an iterative solver to compute an approximate solution.

The Jacobi-Solver is one of such iterative solvers which we employed here to solve our system of equations: The matrix A is formally decomposed into $\mathbf{A} = L + D + R$ with L the lower diagonal components, D the diagonal components and R the upper diagonal components. One solver-iteration is executed by

$$\vec{x}^{(i+1)} = D^{-1}(\vec{b} + (D - A)\vec{x}^{(i)})$$

with $\vec{x}^{(i)}$ storing the approximated solution of $\vec{x}$ after the i-th iteration.

With this iterative solver, we avoid storing the matrix A explicitly and can use sparse matrix data structures instead. For example, the diagonal matrix D is just a vector. We use the Euclidean norm on the residual $\vec{r}^{(i)} = \mathbf{A} \cdot \vec{x}^{(i)} - \vec{b}$ for the stopping criteria at the end of each V-cycle.

6.1.2. Geometric Multigrid Solver

Applying the iterative Jacobi solver for the heat equation directly would lead to elimination of high frequencies in error only while leaving low frequencies in error almost unchanged. Multigrid solvers were developed to account for elimination of lower frequencies by applying the solver on coarser grids as well. The scheme of a multigrid V-cycle running on different resolutions of the original problem is given in fig. 6.1. We use the error-correction scheme of restricting the residual instead of the solution, as it is the case for full-approximation scheme. This reduces our 2D problem size on each level by a factor of 4 which accounts for our *dynamic behavior* when solving for the next time-step. E. g., if we solve a heat equation with a size of 128×128, this leads to (at most) 7 levels for the up- and down-cycle as $2^7 = 128$.

6.1.3. Parallelization

Three data arrays have to be stored for each multigrid level: The approximated solution $\vec{x}$ of the current level, its right side $\vec{b}$ and the residual $\vec{r}$ which is prolonged to the finer level. We used the slicing method for the domain decomposition, distributing our domain to computation units by splitting it along the 2nd array dimension.

For our implementation in X10, all data arrays are stored in distributed arrays with appropriate invasive extensions.

6.1.4. Invasive Parallel Multigrid

The parallel invasive multigrid program with a V-cycle is given in pseudo-code in fig. 6.2. Extensions for invasion are in line 11 and lines 18-21.

Obviously, multigrid levels with a problem size below the number of cores available in the system would not lead to any benefits from the cores unable to run computations on the level.

Basic variables:

N	problem size
x	solution
b	right hand side
r	residual
e	approximated error

Multigrid:

Nr	problem size for restricted level
rr	restricted residual
er	restricted approx. error

Claims:

nc	new claim after reinvade
nc2	updated claim after V-cycle for lower levels

```
1  vcycle(N, x, b):
2          r = computeResidual(N, x, b)
3          while |r| > threshold:
4                  vcycleIteration(N, x, b)
5                  r = computeResidual(N, x, b)
6  vcycleIteration(N, x, b):
7          smoother(N, x, b) # pre−smooth
8          if (N > 3):
9                  r = residual(N, x, b)
10                 nc = reinvade(N, claim)
11                 Nr = N/2 # restricted level
12                 rr = restrict(N, r) # restrict residual to new claim
13                 er = (Nr, 0) # setup error with 0 values
14
15                 nc2 = vcycleIteration(Nr, er, rr) # v−cycle recursion
16
17                 if (nc != nc2): # redistribute
18                         nc = nc2
19                         x.redistribute(Nr, nc)
20                         b.redistribute(Nr, nc)
21                 e = prolongate(Nr, er) # prolongate error
22                 x = x + e # apply correction
23         smoother(N, x, b) # post−smooth
24         return nc # possibly modified claim
```

Figure 6.2.: Invasive parallel multigrid in pseudo code

Running computations on levels with lower problem size, our multigrid also requests fewer compute resources. Once the resource manager redistributes these resources, two crucial aspects have to be considered for distributed memory systems present on our target platforms: data locality and data migration. While the responsibility of data locality is given to the resource manager which is described in the next section, the data migration has to be handled by the application itself. In case of a redistribution of resources, possible data migration has to be done to decrease the latency of accessing data stored at other places.

6.2. Communication Reduction on Data Redistribution

The tricky part of Multigrid is the data redistribution if reinvasion changes the number of resources. A good balance between the places is essential, for all the parallel computations to finish at roughly the same time. Otherwise, processing elements idle, computation potential is wasted, and throughput suffers. This brings up the question how to redistribute data. More specifically, X10 provides DistArray as the primary data structure and for convenience, our invasive runtime should come with redistribution methods. It turns out that the layout of places makes a big difference. Description and figures in this section is based on a joint publication [BBMZ14] together with Sebastian Buchwald, Manuel Mohr, and Matthias Braun.

If the number of places changes, data saved in existing DistArrays may need to be redistributed to balance the load. In case a new place appears, data needs to be moved to this place in order to exploit its additional processing power. Correspondingly, before a place is removed its data must be distributed to other places. As this process is generic and mechanical, it makes sense to offer it as part of a library instead of replicating the code in each application. Communication is potentially expensive, in the following we will present ways to reduce the amount of communication necessary.

We will focus on new places appearing, but the dual case of disappearing places can be handled similarly, where the direction of all communication operations is reversed. Figure 6.3 shows a simple example of a DistArray with a BlockDist distribution, initially distributed over 4 places. If a new place appears, the position of the new place relative to the existing places is not fixed yet. It might make sense to choose the mapping from place ids to physical nodes in a cluster so that neighboring nodes also have consecutive place ids. However, in general,

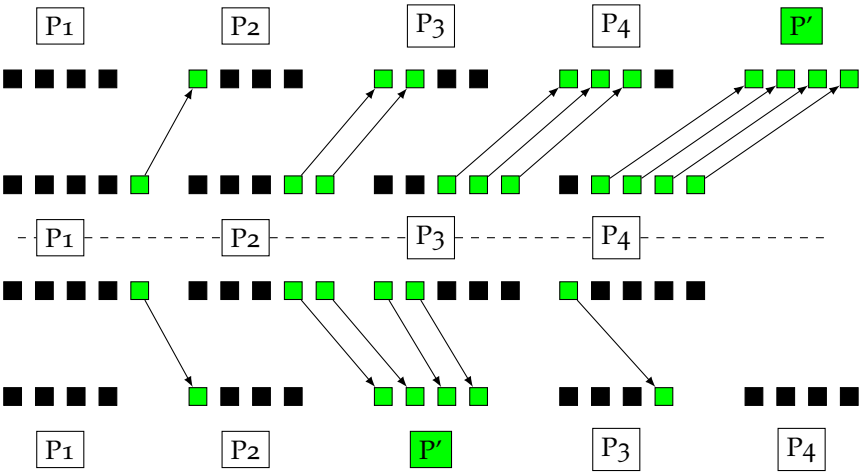


Figure 6.3.: The initial situation with four places P_1 to P_4 is in the middle and each place contains five elements of a DistArray (black boxes). If a new place P' is added to the end, a migration to the top is necessary. If a new place P' is added in the center, a migration to the bottom is necessary. Intuitively, as you see less moved (green) boxes at the bottom, this strategy requires less communication.

the new place can be assigned an arbitrary id in the allowed range. As we will show in the following, it does make a difference in terms of communication overhead which place id is chosen.

We will assume that all communication operations have equal cost, independent of source and destination place, and we will only focus on the number of DistArray elements that have to be transferred. Let p be the number of places before a new place is added and m the number of DistArray elements per place. For ease of presentation, let us assume that m is also divisible by $p+1$, so all array elements mp can also be evenly distributed over $p+1$ places. The amount of data that has to be moved per place is $d := m - \frac{mp}{p+1}$. If the new place is added at the end of the place list, as shown in the upper half of fig. 6.3, the first place has to copy one element, the second two elements, etc. Thus, the total communication costs are

$$c = d \cdot \Sigma_{i=1}^{p} i = d\frac{p(p+1)}{2} = d\frac{p^2+p}{2}.$$

As the number of places p approaches infinity, c can be approximated by the term $dp^2/2$.

However, if the new place is added in the center as shown in the lower half of fig. 6.3, the new place receives half its data from the left $\frac{p}{2}$ places and half its data from the right $\frac{p}{2}$ places (assuming p is even). Following the same reasoning as above, we find that

$$\frac{c'}{2} = d \cdot \Sigma_{i=1}^{\frac{p}{2}} i = d\frac{\frac{p}{2}(\frac{p}{2}+1)}{2}.$$

Hence, $c' = d\frac{p^2+2p}{4}$, which approaches $\frac{dp^2}{4}$ as p approaches infinity. This shows that asymptotically, inserting the new place in the center can save up to 50% of communication operations.

Throughout the argument we assumed that each place should have the same number of elements, but if we have a varying number of PEs per place, then load balancing should distribute the load unevenly. This generalization is straightforward, as you can then calculate with PEs elements instead of places. Likewise, if the system is heterogeneous, PEs may need a different number of elements, but the basic idea is the same. However, in both cases it can be more difficult to compute the "center". A fitting definition of center is that the compute power should be equal on both sides. Calculating the compute power in practice can be hard though, so it might be necessary to use heuristics.

Another assumption we made is that the number of communication operations is the metric to optimize for. Depending on the interconnect between places, only some communication can be parallelized. For example, communication between place 1 and 2 can probably be done in parallel with communication between place 3 and 4. However, communication between place 2 and 5 might not be parallelized with communication between place 3 and 5, because place 5 is a bottleneck. In this case, the new place P' will always be the bottleneck and our strategy to insert it in the center will at least not be worse.

6.3. Multigrid Overhead

We have shown the application itself and pondered the particularly tricky aspect of data migration. Now we investigate, how the application actually behaves when implemented and executed.

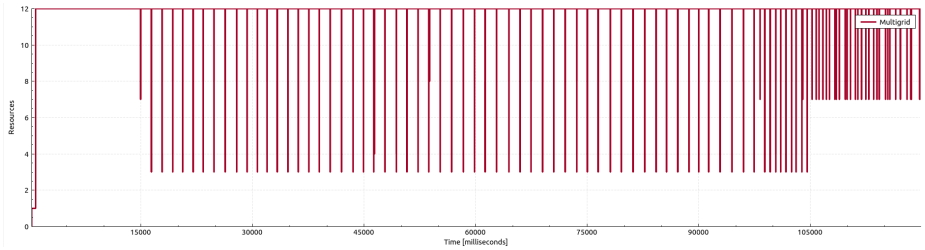


Figure 6.4.: A measurement of Multigrid with 100 iterations on guest layer and visualized with the invadeVIEW tool. You can see the v-cycles as quick dips. For the first iterations the laser is not active, so no coarsening is necessary and no dibs are observed. In the end, the laser takes a second pass over the plate, and the v-cycles get a more chaotic.

A multigrid application actually has nearly constant resource use and is nearly embarrassingly parallel. The v-cycle diagram in fig. 6.1 suggests valleys of low resource use. In practice, the valleys are quick dips, as you can see in fig. 6.4 because the coarsening implies that there is much less to compute. Effectively, a single multigrid is a worst-case scenario, because the application cannot gain anything, while data redistribution adds overhead. The advantage of using our invasive framework is that the application can adapt in a competitive scenario and chapter 8 looks into that. Here we focus only on a single application scenario.

We run evaluations on an invasive guest-layer platform with 3 compute tiles of 4 PEs each. We could configure a higher parallelism than 3×4, but this matches the synthesized FPGA configuration below. It also matches in finer details, for example the amount of tile-local memory. Host OS is an Ubuntu 16.04 AMD64. The hardware is an Intel i7-3770 CPU at 3.4 GHz with 16 GB RAM and 4 cores (8 with hyperthreading). We measure using the Temci benchmarking tool [Bec16], which relies on perf-stat. We use x10i in revision 3118333 and with it *i*RTSS 2017-03-22.

To measure the overhead of Invasive Computing, we run Multigrid in two modes: With and without reinvasion. Without reinvasion it is unable to change its resources, so no data redistribution happens and we call this the "non-invasive" case. Multigrid is configured to run for 100 iterations, which is roughly one pass of the laser across the grid.

We measure the wall-clock time and the cycle counts across 40 runs. Then Temci computes the mean and the standard error mean (SEM). For the invasive Multigrid, we measure a mean wall-clock time of 4.0 s (SEM is 0.005 ms) and 30.0 Gcycles (SEM is 0.04 Gcycles). For the non-invasive Multigrid, we measure a mean wall-clock time of 3.7 s (SEM is 0.007 ms) and 28.5 Gcycles (SEM is 0.022 Gcycles). This means an overhead of 5% in cycles and 8% in wall-clock time. The p-values are practically zero.

When we look at the multigrid phases, we observe that one v-cycle takes 31 ms and the reinvade and redistribute phase take 3 ms of that, which is the 8% difference we measure in wall clock time.

Although reinvade and redistribute are not major factors, they are the ones directly concerned with resource-awareness, so we investigate them further. We run the application ten times and measure each reinvasion and redistribution time. Each v-cycle requires multiple reinvade and redistribution steps, so the numbers do not add up to the 3 ms above. We observe a median reinvasion time of 171 µs with a median absolute deviation (MAD) of 50 µs. In contrast, redistribution times are 519 µs with an MAD of 82 µs. Thus, redistribution is usually the more time consuming phase which takes roughly three times as long. However, we also see maximum times of 14 063 µs for reinvade and 102 602 µs for redistribute, which means either could dominate the time under specific circumstances. For example, the garbage collector runs three times during the 100 vcycles at indeterministic times. These maxima vary a lot between runs, but the median and MAD are stable.

It is quite possible that this timing variation and cache issues come from the fact that we simulate a manycore system on a multicore hardware, so multiplexing and cache thrashing would be no surprise. We also have cycle-accurate hardware synthesis on an FPGA. Unfortunately, running this evaluation there is less reliable, since most tries to run it fail. Sometimes the configuration of the FPGA fails. At times, we have to turn it off and on again with a USB-controlled power switch. If the software runs, it can crash due to invalid instructions and memory accesses, panic when OctoPOS detect broken invariants, or hangs due to a deadlock or infinite loop. The software problems nearly exclusively happen with the invasive applications, which are identical except that the do not call `reinvade` all the time. This suggests that the errors are in the agent system. These problems could easily introduce bias into the measurements,

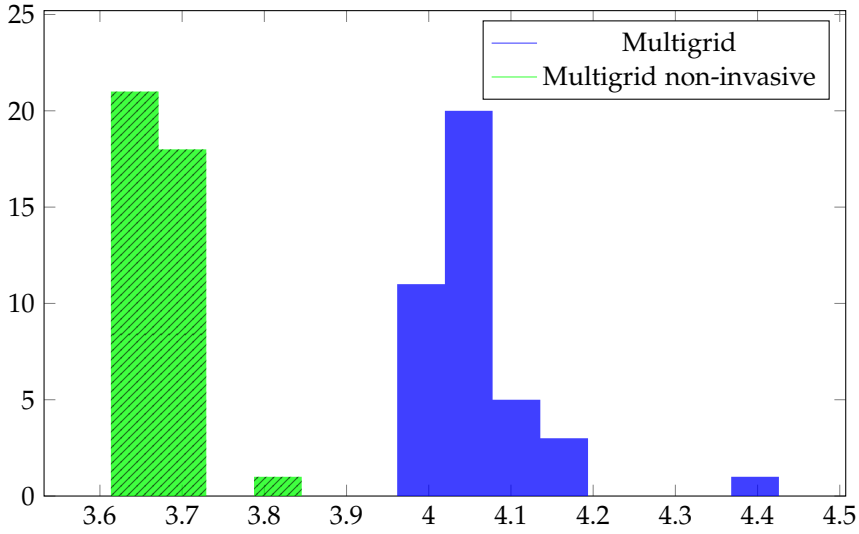


Figure 6.5.: Distribution of wall clock times for invasive and non-invasive (striped) multigrid applications.

but the numbers are still interesting. The tooling to investigate deeper is very limited [FHB14], since the synthesized hardware lacks performance monitoring to measure cache miss rates, for example.

We use the same X10 source code, compiler and *i*RTSS versions as before, but generate SPARC code for the synthesized Leon cores. The hardware platform is the same as the guest-layer: 3 compute tiles with 4 Leon PEs running at 25 MHz each. Our FPGA platform is CHIPit by Synopsis which contains six Xilinx Virtex-5 LX330T FPGAs equivalent to 12 million ASIC gates.

We have 10 successful runs for each variant (and in the invasive case additionally 23 unsuccessful runs with a crash, panic, or hang) and recorded the wall time as reported by *i*RTSS via `clock`. For the invasive variants we measure 306 s and 252 s with deviations of less then 1 s each. This corresponds to an overhead of 21%.

Case Study: Invasive Numeric Integration

Don't worry about people stealing an idea. If it's original, you will have to ram it down their throats.

— Howard H. Aiken

7.1. Numerical Integration

Numerical integration is an important basic technique in high performance computing. We have already seen a small example in fig. 3.5 in section 3.4.1. In realistic applications, functions might take hours to compute and the results might be very precise, for example requiring a hundred decimal places.

Here we are concerned with the load balancing aspects, so we use a simple function and the precision of `double`. Our integration algorithm is the recursive rectangle method, which requires a recursion depth as termination condition in steep regions. It is also adaptive, which requires a parameter ϵ as early termination condition for flat regions. Figure 7.1 shows an illustration.

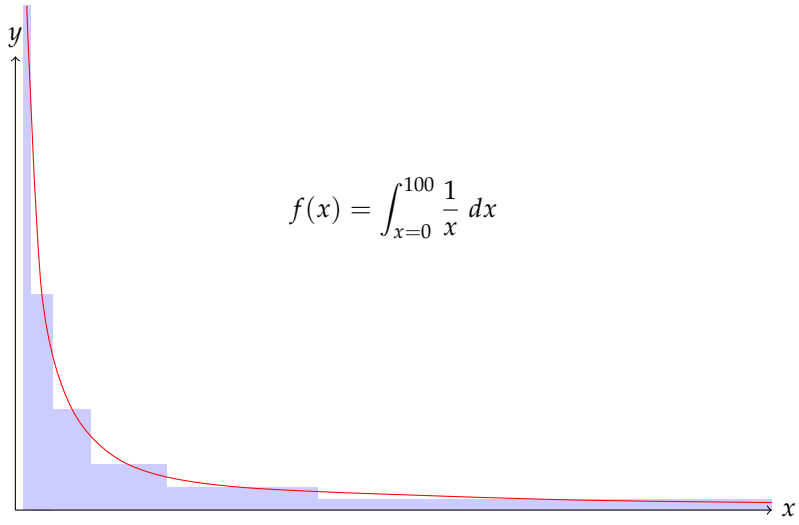


Figure 7.1.: Our adaptive integration algorithm visualized. On the flat region on the right, a wide box is approximation enough. On the steep region on the left, tight steps are necessary. As $1/x \to \infty$ for $x \to 0$, the recursion depth termination is hit there.

7.1.1. Job-Queue Framework

As you have seen in the code in fig. 3.4, numerical integration can be expressed on top of a job-queue abstraction. Norman Böwing developed a generic framework in his master thesis [Böw15]. It features a two tiered work-stealing algorithm, which balances jobs between processing elements within a place and between places across the whole system.

If you consider a function like $1/x$, work balancing is essential. Whichever place evaluates the range close to zero, will create infinitely many jobs. In contrast, the other places will sooner or later run out of jobs due to flatness, so they must steal jobs from the first place again.

The job-queue framework uses reinvasion internally, which means there is no code overhead for the programmer assuming he uses a job-queue framework anyways. Figure 7.2 shows what the programmer has to specify. The application is only 100 lines of X10 code. The distribute queue is 2000 lines of X10 code, but generic and provided by the invasive standard library. Reinvasion is considered

```
1  private val parts = 5; // split factor
2  public def integrateRange(left:T, right:T) {
3    /* initialize starting jobs */
4    val stride = (right − left) / parts;
5    val initialWork = new ArrayList[Job]();
6    for (var i:uint = 0; i<parts; i++) {
7      val l = left + (i *stride);
8      val r = left + ((i+1)*stride);
9      initialWork.add(new Job(l, r, 0));
10   }
11
12   /* initialize job queue framework */
13   val constraints = new PEQuantity(1,12);
14   val builder = new JobQueueFrameworkBuilder[Job, T](constraints);
15   queue = builder
16     .setWorkList(initialWork)
17     .setAutomaticReinvades(true,100)
18     .build();
19
20   /* start computation */
21   val total = queue.start((job:Job) => {
22     return integrateRange(job.left, job.right, job.depth);
23   }, (i:T, j:T) => i + j);
24   return total;
25 }
```

Figure 7.2.: Extending the examples figs. 3.4 and 3.6, this is how numerical integration can be programmed within our invasive framework. The initial jobs are computed in the same way, but need to be packed as Job objects. We configure the job queue framework with custom constraints, the initial job list, and we also let it perform automatic reinvades every 100 ms. The actual computation calls the practically identical integrateRange from the previous examples.

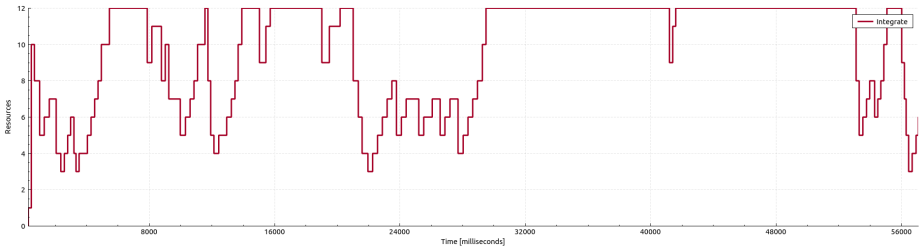


Figure 7.3.: A measurement of Integrate with recursion depth 10 and $\epsilon = 0.01$ integrating $f(x) = 1/x$ on guest layer and visualized with the invadeVIEW tool. It shows capricious unpredictable dynamicity.

periodically. The framework collects load information from all workers and requests more or less resources accordingly. Figure 7.3 plots how this plays out in practice.

To trigger reinvasion periodically works ok. Still, it introduces idle time, from the point where another application retreats from resources until the period ends and the job queue framework uses the resources. This suggests an approach where a retreat of one application triggers reinvasion of another application.

7.2. Integration Overhead

To measure the overhead, we run the application in two modes like the multi-grid in section 6.3. Integration was configured to integrate $f(x) = 1/x$ from 0 to 100 with a recursion depth of 10 and $\epsilon = 0.01$. The non-invasive integration uses all resource for the whole time.

We measured the wall-clock time and the cycle counts across 40 runs. Then we compute the mean and the standard error mean (SEM). For the invasive Integrate, we measured a mean wall-clock time of 51 s (SEM is 0.7 ms) and 990 Gcycles (SEM is 11 Gcycles). For the non-invasive integrate, we measured a mean wall-clock time of 45 s (SEM is 0.5 ms) and 980 Gcycles (SEM is 4 Gcycles). This means an overhead of 1% in cycles and 13% slowdown in wall-clock time. The p-value for wall-clock time is practically zero, but for cycles the t-test computes $p = 0.36$.

Looking deeper, we see 67% more branch misses and 48% more cache-references with 15% more instructions in the invasive version. The task-clock and cpu-clock measurements show nearly no difference and a p-value over 30%. This suggests that invasive integration does roughly the same work, which seems reasonable since no data redistribution is necessary and the computation is the same. The higher wall clock time means that the job queue framework likes to free resources, which is desirable in general, but unnecessary here. Of course, more resources can be requested, but it requires additional work stealing afterwards.

We can also run the benchmark on the FPGA platform with the same caveats as before. We have 10 successful runs for each variant. The results are wall clock times of 89 s (MAD 6 s) invasive and 38 s (MAD 2 s) non-invasive. This means a slowdown of 134% which is excessive and requires looking deeper.

One theory is that the function evaluation is so cheap, that we primarily measure the overhead. Thus we vary the application such that function evaluation takes much longer: We insert a 1 s "busy sleep", which continuously polls the time from the operating system. To counter the increase in run time, we also lower recursion depth from 10 to 5 and increase ϵ from 0.01 to 0.1. We measure at least 5 runs for each.

Now we measure a wall clock time on the guest-layer of 177 s (MAD 15 s) invasive and 186 s (MAD 4 s) non-invasive, which is a speedup of 5%. Since they perform the same computation work and invasive has additional overhead, a speedup can only come from load balancing. Also, the deviation suggests that there is not actually a difference between them. On the FPGA, we measure 178 s (MAD 18 s) invasive and 231 s (MAD 25 s) non-invasive. Which is a speedup of 25% and the load balancing can be the only reason, as well. Considering the deviation, the balancing seems to be consistent here, which is credible because OctoPOS is simpler and more predictable than Linux.

In general, these varied measurement supports the theory that using the function $\frac{1}{x}$ is too cheap and we primarily measure communication, synchronization, and other overhead in the first case. The fact that the invasive variant (at least) as fast as the non-invasive one suggests that the overhead is actually negligible given enough compute work, which matches the experience with iOMP [GHM+12].

It is interesting that the FPGA times are very close to the guest-layer ones, although the guest-layer runs at hundred times the clock frequency. Also, the guest-layer features a clever Intel i7, in contrast to a simple Leon, which could mean another order of magnitude speedup. The advantage of the FPGA is that there are more cores available, hence hardware parallelism, although this only explains a factor of 3 here.

Multi-Application Evaluation

Seek a test that lets reality judge between you.
— Eliezer Yudkowsky

After looking at our two example applications Multigrid (Chapter 6) and Integrate (Chapter 7) in detail, we now investigate their combination. Figure 8.1 shows the wrapper code, which executes both applications simultanously.

Running multiple Multigrid applications in parallel is boring, as they quickly partition the resources equally and then stop trading. The reason is that most of the time Multigrid requires the maximum amount of resources and the quick dips never align in practice. For this reason we need an application like Integrate, which has lower resource needs for longer periods of time.

8.1. Utilization

Once two invasive applications run on the same system, they have to trade resources, because they are assigned exclusively. Figures 8.2 and 8.3 show such a competitive scenario over time with a Multigrid and an Integrate application. In the invasive case, it illustrates that PEs are traded back and forth. Also, most of the time a 100% utilization is achieved and only temporarily some PEs are idle. Without the ability to adapt if more resources become available, overall utilization and performance suffers.

```
1  static def runSeparate(fun:(Claim)=>void) {
2    val claim = Claim.invade(new PEQuantity(1)
3                && TileSharing.WITH_OTHER_APPLICATIONS);
4    claim.infect((id:IncarnationID)=>{
5      fun(claim);
6    });
7    claim.retreat();
8  }
9  public static def main(args:Array[String]) {
10   finish {
11     async runSeparate((claim:Claim)=>{
12       /* run numerical integration */
13       val depth = 10;
14       val epsilon = 0.01;
15       val work = new Integrate3(Integrate3.getFunction(),
16                                 epsilon, depth, false, true);
17       work.integrateRange(0f, 100f);
18     });
19     async runSeparate((claim:Claim)=>{
20       /* run multigrid */
21       val input = new multigrid.ImageInput();
22       val publish = new multigrid.NoOutput();
23       MainMultigrid.default_main(input, publish, 50, true);
24     });
25   }
26 }
```

Figure 8.1.: The runSeperate method executes the provided function in a newly created claim with a single PE. The main function uses it to run the two applications. This is done in parallel via X10's async.

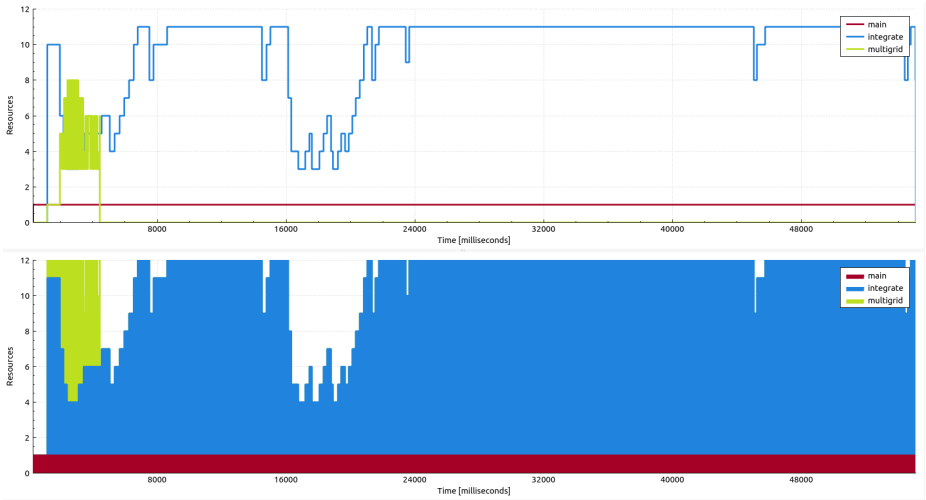


Figure 8.2.: Resource use during a competitive scenario over time. Multigrid in green and Integrate in blue. The red application is the main application, which only started the other two. The upper diagram shows absolute resources use. The lower diagram is stacked, so it shows the total utilization.

We cannot calculate the utilization according to definition 2. Unfortunately, there is no good way to measure the hidden idle resources R_{idle}, so we ignore that factor. We need to calculate utilization over time: $\int U$ (see section 2.8). We can observe all redistributions δ_i and record the number of used resources $p_i = |R| - |R_0|$ and the wall clock time t_i. Now we compute utilization as

$$\sum_i p_i \times (t_i - t_{i-1}).$$

Using this metric, we measure a utilization of 93% (MAD 2.8%) for the invasive variant and 10% (MAD 0.2%) for the non-invasive one. This matches the graphs in figs. 8.2 and 8.3. Since integration is not able to scale arbitrarily, 100% utilization is impossible in this scenario.

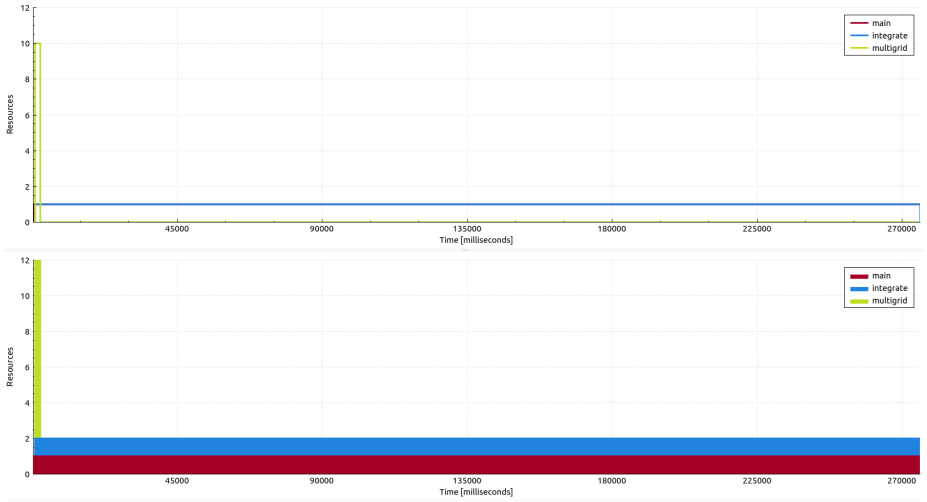


Figure 8.3.: Resource use during a non-invasive competitive scenario in contrast to fig. 8.2. We see resource use like the theory illustration in fig. 2.3. The blue integration application is unable to use the idle resources after the green multigrid application has terminated and continues to use only few PEs.

8.2. Arbitrary Speedup

Section 2.7 showed the speedups to be unbounded in theory. What about real programs, though? What is the benefit if we evaluate a competitive scenario and can we show arbitrary speedups with real programs there? We use the Integrate application as the flexible one and Multigrid as initial resource hog.

Once again, we compare two variants, the invasive and the non-invasive one. In both variants, the same two application, Multigrid and Integrate, are running with the same parameters and data. More concretely, Integrate computes $\int_{x=0}^{100} 1/x$ to a maximum depth of 10, but may terminate early if $\epsilon \leq 0.01$. Multigrid computes 50 iterations for the problem size 127, a maximum residual norm of 0.00001, a recursion depth of 9, a time step size of 0.001, and a heat coefficient of 0.001. The benchmarking platform is the same as in chapters 6 and 7. For each variant we measured the real wall clock time in 20 runs.

Figure 8.4 compares the means: 48 sec and 275 sec with insignificant deviations. The large difference of the means and thus the 5× speedup is obvious.

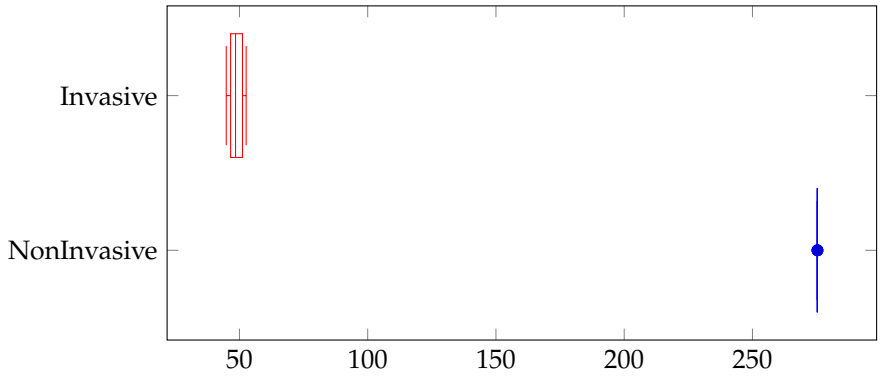


Figure 8.4.: Comparison of the means in the competitive scenario in a whisker plot. The means are 48 sec (deviation is 5.1 σ per mean) for invasive and 275 sec (0.013) non-invasive. Clearly, Invasive is more than 5 times faster than NonInvasive, so computing a p-value is unnecessary.

We can also run the benchmark on the FPGA platform with the same caveats as before. We have 7 successful runs for each variant. The results are 22 s invasive and 29 s non-invasive with insignificant deviations. The speedup is not as good, but still obvious.

We can change the difference arbitrarily by adapting the number of iterations of Multigrid or the depth of Integrate. Thus we conclude, the theoretic speedups from section 2.7 can be observed with real programs as well.

Conclusions

It's still magic even if you know how it's done.
— Terry Pratchett, A Hat Full of Sky

Resource-aware programming in high-level languages on MPSoCs is feasible with manageable effort and improves performance.

9.1. Summary

The introduction described recent trends in computer architectures, extrapolated the trends, and identified the need for resource-awareness and high-level languages. This thesis presented a framework for resource-aware programming on a theoretical level (chapter 2), a useable implementation (chapter 5), and first steps for a bridging between them with a memory consistency model (chapter 4). Within Invasive Computing (chapter 3), we evaluated the implementation on two single application case studies (chapter 6, chapter 7) and also in a competitive multi-application scenario (chapter 8).

The core idea is that applications provide information about their behavior to a resource management system, which decides with global information. Application then must adapt to these decisions. The theory promises improvements for efficiency, utilization, and speed (theorems 1 to 3). In practice, those promises were not always realized. Sometimes overhead overcomes the efficiency improvements. Utilization can suffer if resource management heuristics

go wrong. Speed improvements depend on the emergent application behaviors. Still, chapter 8 shows improvements for utilization and speedup in practice with complex real-world applications.

9.2. Mistakes in Hindsight

The framework presented here is certainly not perfect. Aspects like extensibility due an open hierarchy of constraints have aged well. Additional constraints and hints are easy to add, although interactions with others must be kept in mind. The framework is used by other people, so backwards compatibility is a factor and some issues remain. This section describes what should have been done differently in hindsight.

9.2.1. Reinvade from Inside

We designed the framework with more focus on invasion than on reinvasion. Invasion must be done from *outside* of a claim, because the claim does not exist before. It turned out that reinvasion is the more important mechanism, because it enables the dynamicity. Reinvade is usually performed more often, since invade can by definition only be performed once per claim.

As we gained experience writing invasive applications, we repeatedly observed that reinvade was performed from *within* the claim, as shown in fig. 9.1. Inside the claim is the location where the data is which we need to define constraints and hints. Combined with the idea that claims are an isolation mechanism, it turned out that modifying resources is more intuitive from the inside.

In hindsight, it would have been a simpler and safer design, if we would only provide a single method which executes this invade with a single PE. It would take a single parameter: the closure to execute inside the new claim. The closure itself would take the new claim as a parameter as fig. 9.2 illustrates. Apart from being the common use, there are two advantages. First, the retreat can be implicit at the return of the closure. This is safer as the programmer cannot forget about it. Second, it decouples patterns like map-reduce. Currently, we overload infect to support this pattern in an integrated way. Distributed map-reduce is a hard problem in itself, so it is limiting to integrate it into our framework. We could provide such a method for convenience of course, but it should be promoted as the default way to invade.

```
1  val claim = Claim.invade(new PEQuantity(1)
2                  && TileSharing.WITH_OTHER_APPLICATIONS);
3  claim.infect((id:IncarnationID)=>{
4    /* application code ... */
5    claim.reinvade(other_constraints);
6    /* application code ... */
7  });
8  claim.retreat();
```

Figure 9.1.: Reinvasion from inside the claim: After creating a new claim with only a single PE, all the actual resource-aware dynamicity happens within the claim during the infect within the *i*-let.

9.2.2. X10 Language

X10 is a well-designed language for HPC and backing by IBM promised success in 2010. Nevertheless, it failed at its goal to replace Fortran and C++. Today, development on X10 itself has practically stopped. While version 2.6.1 was released in 2017, there are nearly no changes to the codebase anymore as fig. 9.3 shows.

One promise of X10 was dependent types, but they are only implemented in a very limited form. The X10 dependent type checker only implements the assignment of integer and boolean values and checking them for equality. The only practical use seems to be that array dimensions are known at the type level and the compiler guarantees consistency for array operations. Our initial hope to use the type system for resource-awareness checks was disappointed.

```
1  Claim.create((c:Claim) => {
2    /* application code ... */
3    claim.reinvade(other_constraints);
4    /* application code ... */
5    /* implicit retreat */
6  });
```

Figure 9.2.: Alternative for fig. 9.1, where invade and retreat are implicit.

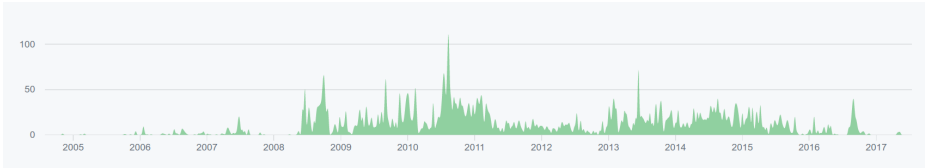


Figure 9.3.: Contributions to the official X10 repository as seen by Github. There was action from 2009 to 2016, but in 2017 only a small blip can be observed.

X10 comes with garbage collection (GC) to provide memory and type safety in an easy to use manner. It relies on GC nearly as much as Java and has no particular support for explicit memory management. The only advantage are value types, which allows for "arrays of structs" where Java would have to use an array of references and thus indirection. The problem with garbage collection [HB05b] is that it requires more memory than explicit memory management. It turned out that the systems we are able to synthesize on our FPGA platform are relatively memory limited compared to desktop computers.

9.3. Future Work

Now let me describe the unfinished business from this thesis, which would be worthwhile to proceed.

9.3.1. Make it Usable

The goal of science is invention, discovering new ideas never thought before. Innovation makes inventions useable. While we have an implementation that we used to develop and run real programs, it is still a research prototype. Turning that into usable tools requires a lot more (primarily engineering) work.

For example, Tobias Weiberg porting some aspects of resource-aware programming to Java ThreadPools [Wei14] was one such effort. Due to the popularity of containers, Linux has isolation mechanism like control groups for resource management now. One could imagine a system daemon, which partitions

resources for applications. In a similar fashion the core idea can be ported to server orchestration, supercomputer cluster management or the dual use of CPUs and GPUs, for example.

9.3.2. Decoupled Performance Modelling

Performance modelling is a wide field, which reaches from simple formulas with one parameter (e.g. Amdahl's Law) over the analysis of complex structures (e.g. annotated UML graphs) to an analysis of code. In the case of constraints, we are restricted to a simple model, because it has to be communicated to resource management, which analyses it at runtime. The model we currently use always felt too simple, but even if you introduce only a few additional parameters it quickly explodes in complexity. Various people stated a need for a better model but with very different ideas.

The one aspect that should be improved is to decouple the three factors platform, application, and data. To calibrate any performance model you have to profile the application on some platform with some data and measure its behavior. Often people use different data to get more generic model parameters. Sometimes people use different platforms (hardware or operating system). However, if you use data with very different characteristics or a different hardware or a different operating system, then people have to measure again. Otherwise the models predictions are not trustworthy. In contrast, a decoupled model does not need this full calibration process. For example, if we change the hardware, we would only need to determine the hardware parameters and reuse the others.

An initial idea would be to start with Amdahl's Law, a simple formula. Then we extend for various phenomena we observe in the real world. For example, algorithms may contain mutually exclusive regions, where time grows linearly with the parallelism. Cache sizes affect the speed and are sometimes the reason for superlinear speedups [HM90], when total cache sizes grows with the number of cores if they all have their own caches. Communication patterns can be modelled [HZQ+13], because we should know if a parallel algorithm sends messages to single neighbors or uses all-to-all mechanisms. More detailed notes on this topic are in appendix B, but none of that has been validated with real soft- and hardware.

9.3.3. Energy-Awareness

Energy consumption is still a growing concern. The hard aspect is that not only the CPU must be considered, but also memory, cooling, disks, network, and other components that draw power. For supercomputers, Sarood et al. [SLK+13] show that it is not straightforward optimize performance with a given power budget. So did Rountree et al. [RAdS+12] with a focus on frequency scaling. For mobile computers or smartphones, we also observe a trend to heterogenity. Most prominently, ARM's big.LITTLE architecture which mixes slow and fast cores on the same system-on-chip. Currently, resource management is done very naively: Either the slow or the fast cores are active, but not both at the same time or some of both classes.

Currently, our resource manager focuses on performance and utilization, but energy consumption adds another conflicting goal. The first challenge is to develop the hints about energy consumption an application can give. Then the second challenge is how the resource manager should weigh performance against energy use. The boundaries for energy consumption are probably not properties of the application but of the system: The smartphone user decides when a low-power mode is used. The administrator of a supercomputer specifies how much power should be consumed.

Acknowledgments

At first, I want to thank my doctoral advisor Gregor Snelting. He is a great supervisor, a great professor, and a great scientist. He provided me with a lot of freedom and supported me in political skirmish. I do not know any professor where I would prefer to work.

Also thanks to my second advisor Jürgen Teich, whose relentless drive made Invasive Computing possible. It follows that this work was partially supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Invasive Computing" (SFB/TR 89).

Also essential for this thesis were my colleagues: Andreas Fried, Andreas Lochbihler, Denis Lohner, Joachim Breitner, Jürgen Graf, Manuel Mohr, Martin Hecker, Martin Mohr, Matthias Braun, Maximilian Wagner, Sebastian Buchwald, Sebastian Ullrich, and Simon Bischof. Together we played, hacked, built, discussed, learned, and teached. Some directly contributed by proof-reading this thesis. Others contributed indirectly.

Another essential part of our chair is our secretary Brigitte Sehan, who managed travel and other bureaucracy for me.

I had more colleagues within Invasive Computing, who built stuff above and below me. They pushed me and I pushed them. In many discussions, we beat the framework into its current shape. Explicitly, I want to thank Martin Schreiber, Jochen Speck, Stephanie Friederich, Benjamin Oechslein, Jens Schedel, Christoph Erhardt.

I had the honor to supervise quite a few great students. Some of their work also contributed to this thesis. There is Johannes Bechberger. His tool Temci helped with the benchmarking and was used for some of the plots. Norman Böwing made the invasive job-queue framework used here.

I also want to thank the people who led me into the compiler field during my diploma thesis: Christoph Mallon, Michael Beck, and Rubino Geiß.

My final thanks go to my lovely wife Anne-Kristin, who grounded me, motivated me, and all around supported my through the whole PhD process.

♡

Appendix

CHIPit Measurements

Starting at 2017-09-06, I measured the following executions with commands like

```
octo_run.sh chipit 2017_06_28−2x2_chipit_rev2 1 $HOME/Multigrid.leonexe 1 withIoTile
```

on the CHIPit platform. The X10 main function prints a timestamp right at the beginning and the end. The timestamp value is provided by OctoPOS. The start times are very consistently at 6 s, so they are not mentioned in the evaluation.

A “hang” means the application locked up somehow. All cores are idle. All activities and *i*-lets are blocked.

A “crash” means the application terminated with an error.

A “panic” means *i*RTSS terminated with an error.

Sometimes CHIPit fails to even load the code. These errors are not recorded here, because they are indifferent to the code.

A.1. Multigrid

6023340000 — crash
6019877000 — hang
6055054000 — hang
6026941000 — hang
6064552000 — 305961451000
6052687000 — panic
6015753000 — 305948422000
5960559000 — hang
6022943000 — panic
6019227000 — hang
6120120000 — hang
6029979000 — hang
6024037000 — hang
6070010000 — 306002214000
6055339000 — hang
6055339000 — 305170982000
6021193000 — 305336807000
6014605000 — hang
6034719000 — panic
6230502000 — hang
6228665000 — hang
6056186000 — 305266846000
6122405000 — hang
6122405000 — 305531268000
6022033000 — panic
6022033000 — 306334512000
6055879000 — hang
6110203000 — hang
6215225000 — panic
6051823000 — hang
6016830000 — 304740104000
5932718000 — panic
6050126000 — 305819156000

A.2. MultigridNonInvasive

6016471000 — 252090548000
6130638000 — 251220463000
6026459000 — 251929601000
6050097000 — 252156316000
6019885000 — 252697562000
6051959000 — 252203716000
6023501000 — 251701900000
6020088000 — 252446842000
6053210000 — 251322617000
5942510000 — 251561023000

A.3. Integrate3

6098791000 — 89657730000
6050744000 — 90776434000
6199396000 — 70646868000
6015231000 — 92005066000
6047291000 — 76840486000
6285467000 — hang
6029948000 — 82472362000
6014959000 — 64751480000
6011776000 — 88619940000
6050268000 — 87017554000
6049786000 — 96044045000

A.4. Integrate3NonInvasive

6015786000 — 38868304000
6040247000 — 35546533000
6012950000 — 39748864000
6007883000 — 35907027000
6012478000 — 37153610000
6015613000 — 37964265000
6126903000 — 38451009000
6045066000 — 36843126000
6022314000 — 42140522000
6062065000 — 40238203000

A.5. MultigridVsIntegrate

5971906000 — 218715085000
6258059000 — 215956450000
6005036000 — hang
5979320000 — 221509589000
6071173000 — hang
5976634000 — 213470888000
5975091000 — panic
6004954000 — panic
5974314000 — 220158034000
5977047000 — 217284978000
5869514000 — 216931957000

A.6. MultigridVsIntegrateNonInvasive

6010570000 — 289084569000
6080787000 — 289150678000
6077190000 — 289140597000
6083952000 — 289158340000
6005996000 — 289073573000
6066288000 — 289118611000
5876878000 — 288958683000

Decoupled Performance Modelling

This chapter contains some leftover notes on a better performance model, which tries to decouple hardware and application factors. I could not bring myself to delete it from this thesis, because these thoughts would probably get lost then. However, I did not find the time pursue this further and validate the model, so is not yet a worthy contribution at this point. To resolve this predicament, the appendix felt like a suitable way out.

We need a good performance model for parallel applications, e.g. for resource management [KBL+11]. There are only simplistic models [Dow97] available, so here I try to make a less simple but more realistic model.

One important pragmatic goal is to separate program and hardware parameters. Measuring parameters is time consuming. For current performance models, the process must be repeated on every little hardware or software change. A desired improvement is to only update the hardware parameters, when the hardware changes and keeping the software parameters the same. Vice versa when the software changes. One step further would be to separate (static) program and (dynamic) data parameters.

Amdahl's Law is common knowledge after fifty years. It provides an argument that speedup through parallelization is bounded due to sequential portions in an algorithm The law gives a time prediction of

$$T(n) = T(1)\left(B + \frac{1-B}{n}\right) \tag{B.1}$$

where $n \in \mathbb{N}$ is the degree of parallelism, $B \in [0,1]$ is the sequential fraction of the algorithm, and $T(1)$ the time of single threaded execution.

B.1. Composable Parallel Regions

One approximation of Amdahl's Law is the simple algorithm pattern of one parallel region. In reality, we can observe multiple parallel regions with different characteristics. Those regions might be in sequence (algorithms with multiple phases) or nested into each other (recursive parallelism).

With multiple region we require additional notation.

n_r = degree of parallelism of a region r
s_r = sequential fraction of a region
p_r = fraction of a region wrt. its parent region
t_r = time of a region r wrt. n_r
T_r = time of a region r with $n_r = 1$

The original law in new notation, with x being the name of former nameless region:

$$t_x = T_x \left(s_x + \frac{p_x}{n_x} \right) \qquad \text{with } p_x = 1 - s_x$$

B.1.1. Parallel Regions in Sequence

For multiple parallel regions in sequence, we can extend the law as

$$t_x = T_x s_x + \sum_i \frac{T_i p_i}{n_i} \tag{B.2}$$

where i sums over all child regions of x. There is no p_x or n_x necessary in this equation.

The s_x factor is computed from p_i, since the fractions must always add up to 1.

$$s_x = 1 - \sum_i p_i \tag{B.3}$$

B.1.2. Nested Parallel Regions

Nesting regions seems trivial. Just substitute T_r for t_r in parent equation and substitute accordingly. Naturally, a nested region might contain a sequential part. In fact, a nested region without further nested regions contains only a sequential part. In other words $s_r = 1$, so the $\sum$ of the equation is 0.

Doing the substitution, we see a pattern:

$$\begin{aligned}
t_x &= T_x s_x + \sum_i \frac{t_i p_i}{n_i} \\
&= T_x s_x + \sum_i \frac{\left(T_i s_i + \sum_j \frac{t_j p_j}{n_j}\right) p_i}{n_i} \\
&= T_x s_x + \sum_i \frac{T_i s_i p_i + p_i \sum_j \frac{t_j p_j}{n_j}}{n_i} \\
&= T_x s_x + \sum_i \left(\frac{T_i s_i p_i}{n_i} + \frac{p_i}{n_i} \sum_j \frac{t_j p_j}{n_j} \right) \\
&= T_x s_x + \sum_i \left(\frac{T_i s_i p_i}{n_i} \right) + \sum_i \sum_j \left(\frac{t_j p_i p_j}{n_i n_j} \right) \\
&= T_x s_x + \sum_i \frac{T_i s_i p_i}{n_i} + \sum_i \sum_j \frac{T_j s_j p_i p_j}{n_i n_j} + \sum_i \sum_j \sum_k \frac{t_j p_i p_j p_k}{n_i n_j n_k}
\end{aligned}$$

The sum expands with each t_x substitution. For a completely expanded version we need two more definitions, for the expanding products:

$$\begin{aligned}
u(r) &= \text{parent region of region } r \\
P_r &= p_r \times p_{u(r)} \times p_{u(u(r))} \times ... \\
N_r &= n_r \times n_{u(r)} \times n_{u(u(r))} \times ...
\end{aligned}$$

Now, the formula for nested and sequenced regions is

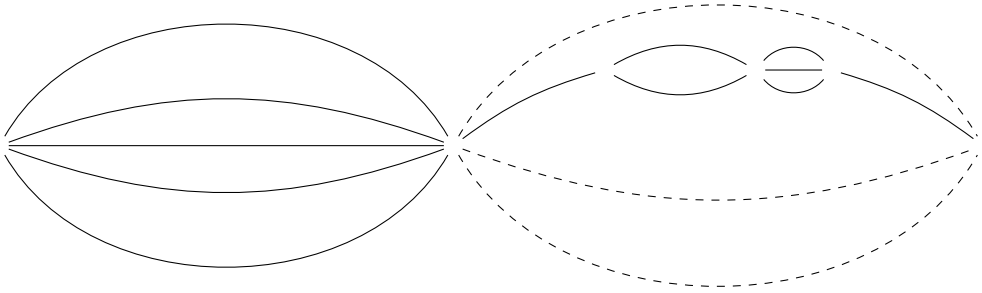
$$t_x = T_x s_x + \sum_i \frac{T_i s_i P_i}{N_i} \tag{B.4}$$

where i sums over all regions except root 0. Wlog, we define the root region to have no sequential part and simplify the formula thus:

$$t_0 = \sum_i \frac{T_i s_i P_i}{N_i} \tag{B.5}$$

B.1.3. Example

Consider a setup, with two regions in sequence (a and b), where b has another two nested regions in sequence (c and d).



Furthermore specify

r	T_r	p_r
a	6	0.4
b	7	0.5
c	8	0.5
d	9	0.2

We can compute the actual factors like this:

$$\begin{aligned}
t_0 &= \sum_i \frac{T_i s_i P_i}{N_i} \\
&= \frac{T_a s_a P_a}{N_a} + \frac{T_b s_b P_b}{N_b} + \frac{T_c s_c P_c}{N_c} + \frac{T_d s_d P_d}{N_d} \\
&= \frac{T_a s_a p_a}{n_a} + \frac{T_b s_b p_b}{n_b} + \frac{T_c s_c p_b p_c}{n_b n_c} + \frac{T_d s_d p_b p_d}{n_b n_d} \\
&= \frac{T_a p_a}{n_a} + \frac{T_b s_b p_b}{n_b} + \frac{T_c p_b p_c}{n_b n_c} + \frac{T_d p_b p_d}{n_b n_d} \\
&= \frac{6 \times 0.4}{n_a} + \frac{7 \times 0.3 \times 0.5}{n_b} + \frac{8 \times 0.5 \times 0.5}{n_b n_c} + \frac{9 \times 0.5 \times 0.2}{n_b n_d} \\
&= \frac{2.4}{n_a} + \frac{1.25}{n_b} + \frac{2}{n_b n_c} + \frac{0.9}{n_b n_d}
\end{aligned}$$

B.2. Non-linear Speedups

This simple law is a very crude approximation. It assumes linear speedup within parallel regions. In reality we can observe non-algorithmic influences, which yield sublinear speedup, e.g. due to hardware contention or runtime limitations. Also, superlinear speedups can be observed, e.g. due to cache effects. To model this environment influence, we include a function $e : \mathbb{N} \mapsto \mathbb{R}^+$, such that

$$e(n) = \begin{cases} > n & \text{for superlinear effects} \\ n & \text{for linear effects} \\ < n & \text{for sublinear effects} \end{cases}$$

where n is also the degree of parallelism.

Thus instead of n_i, we now use $e_i(n_i)$. We can change the meaning of N_i accordingly and avoid changing the formula.

$$N_r = e_r(n_r) \times e_r(n_{u(r)}) \times e_r(n_{u(u(r))}) \times ...$$

The actual challenge in reality: How to obtain e? A disappointing but pragmatic answer would be by measurement. More useful would be a generic environment model, from which e can be derived without measurements.

B.2.1. Mutual Exclusion

One very common method to coordinate parallel work is mutual exclusion. We do not care about the actual mechanism (OS lock, spinlock, monitor, transactional memory, etc), but assume a constant time per thread T^e. Increasing the number of threads n, naturally the time spent for mutual exclusion increases, since it adds up: $n_r * T_r^e$. This models a sublinear effect, which can be statically derived from an algorithm.

There is a conservative assumption here that we do not parallelize mutual exclusion with actual work. Hence, in practice the algorithm might be faster.

To integrate this effect into the formula, we need to split the exclusive parts s_r^e from the rest of the sequential part s_r. Hence, the formula becomes

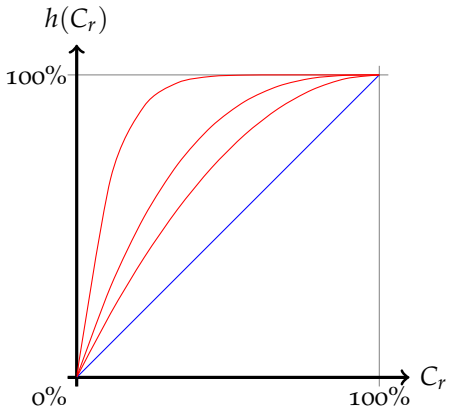
$$t_0 = \sum_i \left(\frac{T_i}{N_i} s_i P_i + n_i T_i^e s_i^e P_i \right) \tag{B.6}$$

B.2.2. Caching

Caching in general has a big effect on algorithms (even if not primarily memory latency bound). Superlinear effects happen because increasing the number of processors increases the total cache amount as well. The insight that increasing cache size speeds up is not very helpful. In general, modifying the cache changes speedup. Caches today have multiple layers, varying sizes, varying retention strategies, different cache coherence protocols, and more variables. Nevertheless, we only care about speedup due to parallelism, which keeps most of those aspects constant. We only consider the size of the cache in relation to the data and the data per processor decreases with more parallelism. Also, we consider processor-local memory in general, so a scratchpad qualifies as (software-managed) cache here. From data size d and the algorithm, we can deduce how cache-dependent it is. Using the cache size per core c, we can compute the relative cache size C:

$$C_r = \min\left(1, \frac{N_r c}{d_r}\right) = \min(d_r, N_r c)/d_r$$

The cache hit rate does not directly correlate with cache size. Instead even small caches might cover a lot of misses, but 100% is only slowly approximated. However, when the cache is as big as the data, a 100% hitrate can be achieved. The following plot illustrates the approximation, which plots the hitrate $h_r(C_r)$ against the relative cache size C_r.



$h(C) = 1 - (1 - C)^2$
$h(C) = 1 - (1 - C)^3$
$h(C) = 1 - (1 - C)^9$

As this cache related speedup is by definition superlinear (up to 100%), this effect creates a superlinear influence to the sequential speedup. We only consider the sequential part of regions, since the parallel part just consists of multiple nested sequential regions.

We also need to know the speed difference between cache hit and miss. For example, $m^c = 10$ means that a miss takes 10 times as long as a hit. Now the relative time is

$$s^c(C) = (1 - h(C)) + \frac{h(C)}{m^c} = 1 - \frac{m^c - 1}{m^c} h(C)$$

Now substitute this for the previous term

$$T_r \implies T_r \times \left(1 - \frac{m^c - 1}{m^c} h(C)\right)$$

The modified formula for nested regions becomes:

$$t_x = \sum_i \frac{T_i}{N_i} s_i P_i s_i^c (\min(d_i, N_i c)/d_i) \tag{B.7}$$

$$= \sum_i \frac{T_i}{N_i} s_i P_i \left(1 - \frac{m^c - 1}{m^c} h_i(\min(d_i, N_i c)/d_i)\right) \tag{B.8}$$

The minimum can usually be removed depending on your goal. If you are interested in $N_r \to \infty$, then consider $\min(d_r, N_r c)/d_r) = 1$. However, for the computation for finite numbers of N_r, we can usually assume that the data will never fit completely into cache, so $\min(d_r, N_r c)/d_r = N_r c/d_r$.

B.2.3. Communication

As a start, we can consider communication as a phase within a region, where the time changes according to the parallelism. How it changes depends on the communication pattern of the algorithm. In addition to s_r, we also define m_r (messaging), and let s_r shrink so $1 = s_r + m_r + \sum_{u(i)=r} p_i$. The time is given by $T_r^c(n_r)$.

$$t_0 = \sum_i \left(\frac{T_i s_i P_i}{N_i} + T_i^c(n_i) m_i P_i\right) \tag{B.9}$$

The $T_r^c(n_r)$ factor follows communication patterns. If communication is constant (only send data to one neighbor), then $T_r^c(n_r) \in \mathcal{O}(1)$. For everybody-to-everybody communication $T_r^c(n_r) \in \mathcal{O}(n_r)$ for complete connection graph. In the case of bus communication, where messaging must be sequentialized, $T_r^c(n_r) \in \mathcal{O}(n_r^2)$.

B.2.4. All of the Above

Combining the formulas for mutual exclusion and caching into one, yields this monster:

$$t_0 = \sum_i \left(\frac{T_i}{N_i} s_i P_i s_i^c (\min(d_r, N_r c)/d_r) + T_i^c(n_i) m_i P_i + n_i T_i^e s_i^e P_i\right) \tag{B.10}$$

with $1 = s_i + m_i + s_i^e + \sum_{u(i)=r} p_i$.

For simplicity we ignore the cache effects within mutually excluded parts and messaging. Also, ignore the mutual exclusion within messaging.

Limitations Limitations of this model:

- Only symmetric parallelism, which assumes that threads within a parallel region are all doing the same with respect to the model. There cannot be two parallel threads, where one spawns a 3-parallel region and the other a 2-parallel region. Thus would require something like "maximum time of several distinct parallel regions", which complicates the formula enormously. This prevents modelling pipeline applications and multiple applications at once.

- Strict phases. The different kinds of operation (mutual exclusion, sequential computation, messaging) cannot be performed in parallel to each other.

- Homogeneous cores. Our simple model assumes all parallel cores equal so only the number matters. However, in real systems it matters which cores are used. There is hyper-threading within a core, cores sharing a NUMA domain or not, cores sharing a cache or not, cores being on the same socket/tile/processor/rack/continent or not. Acknowledging this in the model transforms it from a single-dimensional optimization problem to a high-multi-dimensional one.

B.2.5. Example

We reuse the example from above, but refine the parameters. From the hardware we set the cache size per core c =1KB and a cache miss takes $m^c = 10$ times as long as a hit. Let $h(C) = 1 - (1 - C)^9$. As we are interested in finite results, we simplify: $\min(d_r, N_r c)/d_r \Rightarrow N_r c/d_r$.

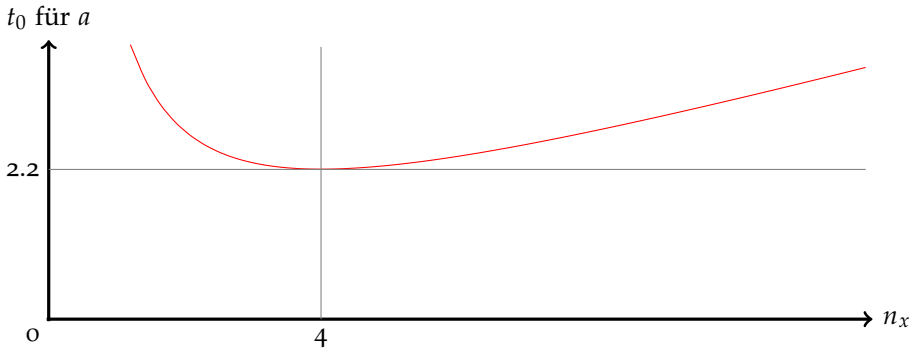
			cache effects		messaging	mut. excl.	
r	p_r	T_r	d_r	s_r	m_i	T_i^e	s_i^e
a	0.4	21	1MB	0.7	0.1	3	0.2
b	0.6	17	10MB	0.2	0.2	4	0.1
c	0.3	42	100MB	0.9	0.1	–	0.0
d	0.2	19	1MB	0.4	0.2	–	0.0

Let $T_r^c(n_r) = n_r$ for all regions except $T_d^c(n_d) = 1$.

First, we just look at region a, because we can optimize it independently from the others.

$$
\begin{aligned}
t_0 &= \sum_i \left(\frac{T_i}{N_i} s_i P_i s_i^c (N_i c / d_r) + T_i^c(n_i) m_i P_i + n_i T_i^e s_i^e P_i \right) \\
&= \frac{T_a}{N_a} s_a P_a s_a^c (N_a c / d_r) + T_a^c(n_a) m_a P_a + n_a T_a^e s_a^e P_a + \dots \\
&= \frac{21}{n_a} \times 0.7 \times 0.4 \times s_a^c(n_a \times 1KB/1MB) + T_a^c(n_a) \times 0.1 \times 0.4 \\
&\qquad\qquad + n_a \times 3 \times 0.2 \times 0.4 + \dots \\
&= \frac{4.48}{n_a} \times \left(1 - \frac{m^c - 1}{m^c} (1 - (1 - 0.001 n_a)^9) \right) + 0.04 \times T_a^c(n_a) + 0.24 n_a + \dots \\
&= \frac{4.48}{n_a} \times \left(1 - 0.9 (1 - (1 - 0.001 n_a)^9) \right) + 0.04 \times T_a^c(n_a) + 0.24 n_a + \dots \\
&= \frac{4.48}{n_a} \times \left(1 - 0.9 + 0.9 (1 - 0.001 n_a)^9 \right) + 0.04 n_a + 0.24 n_a + \dots \\
&= \frac{0.448}{n_a} + \frac{4.032}{n_a} (1 - 0.001 n_a)^9 + 0.28 n_a + \dots
\end{aligned}
$$

This formula has a local minimum at $n_a = 4$. Mostly the messaging determines the upper bound.



Since the times are just summed up, we can optimize the regions independently of each other. Now for region b without subregions c and d.

$$
\begin{aligned}
t_0 &= \ldots \frac{T_b}{N_b} s_b P_b s_b^c (N_r c / d_r) + T_b^c(n_b) m_b P_b + n_b T_b^e s_b^e P_b + \ldots \\
&= \ldots \frac{17}{n_b} \times 0.2 \times 0.6 \times s_b^c (N_c \times 1KB/10MB) + T_b^c(n_b) \times 0.2 \times 0.6 \\
&\qquad + n_b \times 4 \times 0.1 \times 0.6 + \ldots \\
&= \ldots \frac{2.04}{n_b} \times \left(1 - \frac{m^c - 1}{m^c}(1 - (1 - 0.0001 n_b)^9)\right) + 0.12 \times T_b^c(n_b) + 0.24 n_b + \ldots \\
&= \ldots \frac{2.04}{n_b} \times \left(0.1 + 0.9(1 - 0.0001 n_b)^9\right) + 0.12 n_b + 0.24 n_b + \ldots \\
&= \ldots \frac{0.204}{n_b} + \frac{1.836}{n_b}(1 - 0.0001 n_b)^9 + 0.36 n_b + \ldots
\end{aligned}
$$

Local minimum at $n_b = 2$.

$$
\begin{aligned}
t_0 &= \ldots \frac{T_c}{N_c} s_c P_c s_c^c(N_r c/d_r) + T_c^c(n_c) m_c P_c + n_c T_c^e s_c^e P_c + \ldots \\
&= \ldots \frac{42}{n_b \times n_c} \times 0.9 \times 0.18 \times s_c^c(N_c \times 1KB/100MB) \\
&\qquad + T_c^c(n_c) \times 0.1 \times 0.18 + n_c \times 0 \times 0.0 \times 0.18 + \ldots \\
&= \ldots \frac{6.804}{n_c} \times \left(1 - \frac{m^c - 1}{m^c}(1 - (1 - 0.0001N_c)^9)\right) + 0.018 \times T_c^c(n_c) + \ldots \\
&= \ldots \frac{6.804}{n_c} \times \left(0.1 + 0.9(1 - 0.00001N_c)^9\right) + 0.018 n_c + \ldots \\
&= \ldots \frac{0.6804}{n_c} + \frac{6.1236}{n_c}(1 - 0.00001N_c)^9 + 0.018 n_c + \ldots
\end{aligned}
$$

Local minimum at $N_c = n_b \times n_c = 20$. With $n_b = 2$, we conclude $n_c = 10$.

$$
\begin{aligned}
t_0 &= \ldots \frac{T_d}{N_d} s_d P_d s_d^c(N_r c/d_r) + T_d^d(n_d) m_d P_d + n_d T_d^e s_d^e P_d \\
&= \ldots \frac{19}{n_b \times n_d} \times 0.4 \times 0.12 \times s_d^c(N_c \times 1KB/1MB) \\
&\qquad + T_d^c(n_d) \times 0.2 \times 0.12 + n_d \times 0 \times 0.0 \times 0.12 \\
&= \ldots \frac{0.912}{n_d} \times \left(1 - \frac{m^c - 1}{m^c}(1 - (1 - 0.001N_c)^9)\right) + 0.024 \times T_d^d(n_d) \\
&= \ldots \frac{0.912}{n_d} \times \left(0.1 + 0.9(1 - 0.00001N_c)^9\right) + 0.024 \times 1 \\
&= \ldots \frac{0.0912}{n_d} + \frac{0.82}{n_d}(1 - 0.001N_c)^9 + 0.024
\end{aligned}
$$

There is no local minimum for d. It goes to 0.024 and scales infinitely.

B.3. Lessons for InvasIC

What constraints can be infered from this theoretical model? What example algorithms can or cannot be modelled like this?

The hardware properties need no constraints, as the agent system can query this info from the operating system once and use it statically.

We could use a constraint to describe synchronization time. A static analysis (aka the compiler) might be able to infer this.

For the cache effects, we need the data size and the hitrate function.

For communication, we need the pattern (all-to-all,neighbor-only,etc) and the relative amount of time the algorithm uses. This must be combined with hardware knowledge (hypergrid,bus,etc) to estimate $T_r^c(n_r)$.

List of Figures

Bibliography

[ATBS13] I. Anagnostopoulos, V. Tsoutsouras, A. Bartzas, and D. Soudris. Distributed run-time resource management for malleable applications on many-core platforms. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, May 2013.

[BA08] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. *SIGPLAN Not.*, 43(6):68–78, June 2008.

[BBH+13] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. Simple and efficient construction of Static Single Assignment form. In Ranjit Jhala and Koen Bosschere, editors, *Compiler Construction*, volume 7791 of *Lecture Notes in Computer Science*, pages 102–122. Springer Berlin Heidelberg, 2013.

[BBMZ12] Matthias Braun, Sebastian Buchwald, Manuel Mohr, and Andreas Zwinkau. An X10 compiler for invasive architectures. Technical Report 9, Karlsruhe Institute of Technology, 2012.

[BBMZ14] Matthias Braun, Sebastian Buchwald, Manuel Mohr, and Andreas Zwinkau. Dynamic X10: Resource-aware programming for higher efficiency. Technical Report 8, Karlsruhe Institute of Technology, 2014. X10 '14.

[BBZ11] Matthias Braun, Sebastian Buchwald, and Andreas Zwinkau. Firm—a graph-based intermediate representation. Technical Report 35, Karlsruhe Institute of Technology, 2011.

[Bec16] Johannes Bechberger. Besser benchmarken, April 2016.

[BHM00] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial*. Society for Industrial Mathematics, 2000.

[BIM08] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 318–329. IEEE Computer Society, 2008.

[BJM+05] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. *Parallel and Distributed Systems, IEEE Transactions on*, 2005.

[BLU16] Sebastian Buchwald, Denis Lohner, and Sebastian Ullrich. Verified construction of Static Single Assignment form. In Manuel Hermenegildo, editor, *25th International Conference on Compiler Construction*, CC 2016, pages 67–76. ACM, 2016.

[BMF15] Patrick Bellasi, Giuseppe Massari, and William Fornaciari. Effective runtime resource management using linux control groups with the BarbequeRTRM framework. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(2):39, 2015.

[BMF+16] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. OpenPiton: An open source manycore research framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 217–232, New York, NY, USA, 2016. ACM.

[BMR15a] Sebastian Buchwald, Manuel Mohr, and Ignaz Rutter. Optimal shuffle code with permutation instructions, April 2015. Long version of WADS 2015 paper with same title.

[BMR15b] Sebastian Buchwald, Manuel Mohr, and Ignaz Rutter. Optimal shuffle code with permutation instructions. In Frank Dehne, Jörg-Rüdiger Sack, and Ulrike Stege, editors, *Algorithms and Data Structures*, volume 9214 of *Lecture Notes in Computer Science*, pages 528–541. Springer International Publishing, 2015.

[BMZ15] Sebastian Buchwald, Manuel Mohr, and Andreas Zwinkau. Malleable invasive applications. In *Proceedings of the 8th Working Conference on Programming Languages (ATPS'15)*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2015.

[Boe05] Hans-J. Boehm. Threads cannot be implemented as a library. *SIGPLAN Not.*, 40(6):261–268, June 2005.

[Böw15] Norman Christopher Böwing. Invasives verteiltes job queue framework, December 2015.

[Bro99] Guy Brown. *The Energy of Life*. Free Press, 1999.

[BRS+13] Hans-Joachim Bungartz, Christoph Riesinger, Martin Schreiber, Gregor Snelting, and Andreas Zwinkau. Invasive Computing in HPC with X10. In *Proceedings of the third ACM SIGPLAN X10 Workshop*, X10 '13, pages 12–19, New York, NY, USA, 2013. ACM.

[Buc15] Sebastian Buchwald. Optgen: A generator for local optimizations. In Björn Franke, editor, *Compiler Construction*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2015.

[CAB+13] Nicholas P. Carter, Aditya Agrawal, Shekhar Borkar, Romain Cledat, Howard David, Dave Dunning, Joshua Fryman, Ivan Ganev, Roger A. Golliver, Rob Knauerhase, Richard Lethin, Benoit Meister, Asit K. Mishra, Wilfred R. Pinfold, Justin Teller, Josep Torrellas, Nicolas Vasilache, Ganesh Venkatesh, and Jianping Xu. Runnemede: An architecture for ubiquitous high-performance computing. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, pages 198–209, Washington, DC, USA, 2013. IEEE Computer Society.

[CEH+13] J.A. Colmenares, G. Eads, S. Hofmeyr, S. Bird, M. Moreto, D. Chou, B. Gluzman, E. Roman, D.B. Bartolini, N. Mor, K. Asanovic, and J.D. Kubiatowicz. Tessellation: Refactoring the OS around explicit resource containers with continuous adaptation. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pages 1–10, May 2013.

[CGG+14] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1):5–28, 2014.

[CGH+14] David Cunningham, David Grove, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Hiroki Murata, Vijay Saraswat, Mikio Takeuchi, and Olivier Tardieu. Resilient X10: Efficient failure-aware programming. *SIGPLAN Not.*, 49(8):67–80, February 2014.

[CLRB11] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. Evaluation and improvements of programming models for the Intel SCC many-core processor. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 525–532, July 2011.

[CMHGB16] Isaías Comprés, Ao Mo-Hellenbrand, Michael Gerndt, and Hans-Joachim Bungartz. Infrastructure and API extensions for elastic execution of MPI applications. In *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016, pages 82–97, New York, NY, USA, 2016. ACM.

[DGnY+74] Robert H. Dennard, Fritz H. Gaensslen, Hwa nien Yu, V. Leo Rideout, Ernest Bassous, Andre, and R. Leblanc. Design of ion-implanted MOSFETs with very small physical dimensions. *IEEE J. Solid-State Circuits*, page 256, 1974.

[Dow97] Allen B Downey. A model for speedup of parallel programs. Technical report, University of California, Berkeley, CA, USA, 1997.

[EBSA+11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[EMDSV07] K. El Maghraoui, T.J. Desell, B.K. Szymanski, and C.A. Varela. Dynamic malleability in iterative MPI applications. In *Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 591–598, May 2007.

[EMDSV09] K. El Maghraoui, Travis J. Desell, Boleslaw K. Szymanski, and Carlos A. Varela. Malleable iterative MPI applications. *Concurr. Comput. : Pract. Exper.*, 21(3):393–413, March 2009.

[EZL89] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, Mar 1989.

[FHB14] Stephanie Friederich, Jan Heisswolf, and Jürgen Becker. Hardware/software debugging of large scale many-core architectures. In *Proceedings of the 27th Symposium on Integrated Circuits and Systems Design*, SBCCI '14, pages 45:1–45:7, New York, NY, USA, 2014. ACM.

[FR96] Dror G. Feitelson and Larry Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer Berlin Heidelberg, 1996.

[GBL09] Vahid Garousi, Lionel C. Briand, and Yvan Labiche. A UML-based quantitative framework for early prediction of resource usage and load in distributed real-time systems. *Software & Systems Modeling*, 8(2):275–302, 2009.

[GHM+12] M. Gerndt, A. Hollmann, M. Meyer, M. Schreiber, and J. Weidendorfer. Invasive Computing with iOMP. In *Specification and Design Languages (FDL), 2012 Forum on*, pages 225–231, Sept 2012.

[GSL+14] D. Gangadharan, É. Sousa, V. Lari, F. Hannig, and J. Teich. Application-driven reconfiguration of shared resources for timing predictability of MPSoC platforms. In *2014 48th Asilomar Conference on Signals, Systems and Computers*, pages 398–403, Nov 2014.

[Haa16] Jonas Haag. Exception support in graph-based intermediate representation, April 2016.

[HB05a] Matthew Hertz and Emery D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 313–326, New York, NY, USA, 2005. ACM.

[HB05b] Matthew Hertz and Emery D Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *ACM SIGPLAN Notices*, volume 40, pages 313–326. ACM, 2005.

[HBHG11] 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.

[HKZ+11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[HLB+14] Frank Hannig, Vahid Lari, Srinivas Boppu, Alexandru Tanase, and Oliver Reiche. Invasive tightly-coupled processor arrays: A domain-specific architecture/compiler co-design approach. *ACM Trans. Embed. Comput. Syst.*, 13(4s):133:1–133:29, April 2014.

[HLK04] Chao Huang, Orion Lawlor, and L.V. Kalé. Adaptive MPI. In *Languages and Compilers for Parallel Computing*, volume 2958 of *Lecture Notes in Computer Science*, pages 306–322. Springer Berlin Heidelberg, 2004.

[HM90] D. P. Helmbold and C. E. McDowell. Modeling speedup (n) greater than n. *IEEE Trans. Parallel Distrib. Syst.*, 1(2):250–256, April 1990.

[HMP97] Markus Hof, Hanspeter Mössenböck, and Peter Pirkelbauer. Zero-overhead exception handling using metaprogramming. In *SOFSEM'97: Theory and Practice of Informatics*, pages 423–431. Springer, 1997.

[Höl10] Urs Hölzle. Brawny cores still beat wimpy cores, most of the time. *IEEE Micro*, 30(4), 2010.

[HZKK06] Chao Huang, Gengbin Zheng, Laxmikant Kalé, and Sameer Kumar. Performance evaluation of adaptive MPI. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 12–21, 2006.

[HZQ+13] Tian Huang, Yongxin Zhu, Meikang Qiu, Xiaojing Yin, and Xu Wang. Extending Amdahl's law and Gustafson's law by evaluating interconnections on multi-core processors. *The Journal of Supercomputing*, 66(1):305–319, 2013.

[HZZ+14] Jan Heisswolf, Aurang Zaib, Andreas Zwinkau, Sebastian Kobbe, Andreas Weichslgartner, Jürgen Teich, Jörg Henkel, Gregor Snelting, Andreas Herkersdorf, and Jürgen Becker. CAP: Communication aware programming. In *Design Automation Conference (DAC), 2014 51th ACM / EDAC / IEEE*, 2014.

[Inc17] Cray Inc. *Chapel Language Specification vo.983*, April 2017.

[ISO14] Programming language C++. Standard, International Organization for Standardization, Geneva, CH, November 2014.

[ITR11] International technology roadmap for semiconductors, 2011.

[Kal14] Kalray. MPPA® MANYCORE is a family of programmable many-core processors, 2014.

[Kar16] Nilesh Karavadara. *RA-LPEL: A Resource-Aware Light-Weight Parallel Execution Layer for Reactive Stream Processing Networks on The SCC Many-core Tiled Architecture*. PhD thesis, University of Hertfordshire, 2016.

[KBL+11] Sebastian Kobbe, Lars Bauer, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Jörg Henkel. DistRM: Distributed resource management for on-chip many-core systems. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis*, CODES+ISSS '11, pages 119–128, New York, NY, USA, 2011. ACM.

[KGVA16] Manfred Kröhnert, Raphael Grimm, Nikolaus Vahrenkamp, and Tamim Asfour. Resource-aware motion planning. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pages 32–39. IEEE, 2016.

[Kob15] Sebastian Kobbe. *Scalable and Distributed Resource Management for Many-Core Systems*. Dissertation, Chair for Embedded Systems (CES), Department of Computer Science, Karlsruhe Institute of Technology (KIT), Germany, 2015.

[Krö17] Manfred Kröhnert. *A Contribution to Resource-Aware Architectures for Humanoid Robots*, volume 1. KIT Scientific Publishing, 2017.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[LCG+15] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 450–462. ACM, 2015.

[Len03] Peter Lennie. The cost of cortical computation. *Current Biology*, 13(6):493–497, mar 2003.

[Loc14] Andreas Lochbihler. Making the Java memory model safe. *ACM Transactions on Programming Languages and Systems*, 35(4):12:1–12:65, 2014.

[LP10] Jonathan K. Lee and Jens Palsberg. Featherweight X10: A core calculus for async-finish parallelism. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 25–36, New York, NY, USA, 2010. ACM.

[LS06] Claudia Leopold and Michael Süß. Observations on MPI-2 support for hybrid master/slave applications in dynamic and heterogeneous environments. In Bernd Mohr, Jesper Larsson Träff, Joachim Worringen, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192 of *Lecture Notes in Computer Science*, pages 285–292. Springer Berlin Heidelberg, 2006.

[LSB06] Claudia Leopold, Michael Süß, and Jens Breitbart. Programming for malleability with hybrid MPI-2 and OpenMP: Experiences with a simulation program for global water prognosis. In *Proceedings of the European Conference on Modelling and Simulation*, pages 665–670, 2006.

[MBZ+15] Manuel Mohr, Sebastian Buchwald, Andreas Zwinkau, Christoph Erhardt, Benjamin Oechslein, Jens Schedel, and Daniel Lohmann. Cutting out the middleman: OS-level support for X10 activities. In *Proceedings of the fifth ACM SIGPLAN X10 Workshop*, X10 '15, pages 13–18, New York, NY, USA, 2015. ACM.

[McC16] John D. McCalpin. Memory bandwidth and system balance in HPC systems. Supercomputing, 2016.

[MHS+10] Martina Maggio, Henry Hoffmann, Marco D Santambrogio, Anant Agarwal, and Alberto Leva. Controlling software applications via resource allocation within the heartbeats framework. In *Decision and Control (CDC), 2010 49th IEEE Conference on*, pages 3736–3741. IEEE, 2010.

[MHS+11] Martina Maggio, Henry Hoffmann, Marco D Santambrogio, Anant Agarwal, and Alberto Leva. Decision making in autonomic computing systems: comparison of approaches and techniques. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 201–204. ACM, 2011.

[Moo65] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

[MT17] Manuel Mohr and Carsten Tradowsky. Pegasus: Efficient data transfers for PGAS languages on non-cache-coherent many-cores. In *Proceedings of Design, Automation and Test in Europe Conference Exhibition*, DATE. IEEE, 2017.

[NCM03] Nathaniel Nystrom, Michael R Clarkson, and Andrew C Myers. Polyglot: An extensible compiler framework for Java. In *International Conference on Compiler Construction*, pages 138–152. Springer, 2003.

[ORS+04] J. Oliver, R. Rao, P. Sultana, J. Crandall, E. Czernikowski, L. W. Jones, D. Franklin, V. Akella, and F. T. Chong. Synchroscalar: a multiple clock domain, power-aware, tile-based embedded processor. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, pages 150–161, June 2004.

[OSK+11] 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 Ross McIlroy, Joe Sventek, Tim Harris, and Timothy Roscoe,

editors, *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA)*, volume USB Proceedings of *Sixth International ACM/EuroSys European Conference on Computer Systems (EuroSys)*, pages 9–14. EuroSys, April 2011.

[PS16] Sankaralingam Panneerselvam and Michael Swift. Rinnegan: Efficient resource use in heterogeneous architectures. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 373–386, New York, NY, USA, 2016. ACM.

[PSKA14] Johny Paul, Walter Stechele, Manfred Kröhnert, and Tamim Asfour. Resource-aware programming for robotic vision. *CoRR*, abs/1405.2908, 2014.

[PSS+14] Johny Paul, Walter Stechele, Éricles Sousa, Vahid Lari, Frank Hannig, Jürgen Teich, Manfred Kröhnert, and Tamim Asfour. Self-adaptive harris corner detector on heterogeneous many-core processor. In *Design and Architectures for Signal and Image Processing (DASIP), 2014 Conference on*, pages 1–8. IEEE, 2014.

[RAdS+12] Barry Rountree, Dong H Ahn, Bronis R de Supinski, David K Lowenthal, and Martin Schulz. Beyond DVFS: A first look at performance under a hardware-enforced power bound. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 947–953. IEEE, 2012.

[RPS+16] Sascha Roloff, Alexander Pöppl, Tobias Schwarzer, Stefan Wildermann, Michael Bader, Michael Glaß, Frank Hannig, and Jürgen Teich. ActorX10: An actor library for X10. In *Proceedings of the 6th ACM SIGPLAN Workshop on X10*, X10 2016, pages 24–29, New York, NY, USA, 2016. ACM.

[SBdADP13] Juliana M. N. Silva, Cristina Boeres, Lúcia Maria de A. Drummond, and Artur Alves Pessoa. Memory aware load balance strategy on a parallel branch-and-bound application. *CoRR*, abs/1302.5679, 2013.

[SBP+14] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification. Technical report, IBM, February 2014.

[SGS14] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. Adaptive, efficient, parallel execution of parallel programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 169–180, New York, NY, USA, 2014. ACM.

[SHT+15] Éricles Sousa, Frank Hannig, Jürgen Teich, Qingqing Chen, and Ulf Schlichtmann. Runtime adaptation of application execution under thermal and power constraints in massively parallel processor arrays. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '15, pages 121–124, New York, NY, USA, 2015. ACM.

[SJMvP07] Vijay A. Saraswat, Radha Jagadeesan, Maged Michael, and Christoph von Praun. A theory of memory models. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 161–172, New York, NY, USA, 2007. ACM.

[SLK+13] Osman Sarood, Akhil Langer, Laxmikant Kalé, Barry Rountree, and Bronis De Supinski. Optimizing power allocation to CPU and memory subsystems in overprovisioned HPC systems. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013.

[SSF13] Jochen Speck, Peter Sanders, and Patrick Flick. Malleable sorting. In *International Symposium on Parallel and Distributed Processing*. IEEE Computer Society, May 2013.

[SSM+07] Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J. Eggers. The WaveScalar architecture. *ACM Trans. Comput. Syst.*, 25(2):4:1–4:54, May 2007.

[TGR+16] J. Teich, M. Glaß, S. Roloff, W. Schröder-Preikschat, G. Snelting, A. Weichslgartner, and S. Wildermann. Language and compilation of parallel programs for *-predictable MPSoC execution using invasive computing. In *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, pages 313–320, Sept 2016.

[TH09] Jürgen Teich and Sebastian Harl, editors. *Invasive Computing*. Funding Proposal. DFG Transregional Collaborative Research Centre 89, 2009.

[THH+11] Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting. Invasive Computing: An overview. In Michael Hübner and Jürgen Becker, editors, *Multiprocessor System-on-Chip – Hardware Design and Tool Integration*, pages 241–268. Springer, Berlin, Heidelberg, 2011.

[TLM+04] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 2–, Washington, DC, USA, 2004. IEEE Computer Society.

[TOS01] Ulrich Trottenberg, Cornelis Oosterlee, and Anton Schüller. *Multigrid*. Academic Press, 2001.

[TWOSP12] J. Teich, A. Weichslgartner, B. Oechslein, and W. Schröder-Preikschat. Invasive Computing - concepts and overheads. In *Proceeding of the 2012 Forum on Specification and Design Languages*, pages 217–224, Sept 2012.

[Val11] Leslie G. Valiant. A bridging model for multi-core computing. *Journal of Computer and System Sciences*, 77(1):154 – 166, 2011. Celebrating Karp's Kyoto Prize.

[WBB+16] Stefan Wildermann, Michael Bader, Lars Bauer, Marvin Damschen, Dirk Gabriel, Michael Gerndt, Michael Glaß, Jörg Henkel, Johny Paul, Alexander Pöppl, Sascha Roloff, Tobias Schwarzer, Gregor Snelting, Walter Stechele, Jürgen Teich, Andreas Weichslgartner, and Andreas Zwinkau. Invasive Computing for timing-predictable stream processing on MPSoCs. *it – Information Technology*, 58(6):267–280, 2016.

[Wei14] Tobias Weiberg. Kommunizierende thread pools, March 2014.

[WGH+07] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the Tile processor. *IEEE Micro*, 27(5):15–31, September 2007.

[WGW+14] Andreas Weichslgartner, Deepak Gangadharan, Stefan Wildermann, Michael Glaß, and Jürgen Teich. DAARM: Design-time application analysis and run-time mapping for predictable execution in many-core systems. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2014 International Conference on*, pages 1–10. IEEE, 2014.

[ZBS13] Andreas Zwinkau, Sebastian Buchwald, and Gregor Snelting. InvadeX10 documentation v0.5. Technical Report 7, Karlsruhe Institute of Technology, 2013.

[Zwi12] Andreas Zwinkau. Resource awareness for efficiency in high-level programming languages. Technical Report 12, Karlsruhe Institute of Technology, 2012.

[Zwi16] Andreas Zwinkau. An X10 memory model. In *Proceedings of the sixth ACM SIGPLAN X10 Workshop*, X10 '16, June 2016.

Index