OctoPOS: A Parallel Operating System for Invasive Computing*

Benjamin Oechslein+, Jens Schedel+, Jürgen Kleinöder+, Lars Bauer◇, Jörg Henkel◇
Daniel Lohmann+, Wolfgang Schröder-Preikschat+
Friedrich–Alexander University Erlangen–Nuremberg, Germany+
Karlsruhe Institute of Technology, Germany◇
{oechslein, schedel, kleinoeder, lohmann, wosch}@cs.fau.de, {lars.bauer, henkel}@kit.edu

ABSTRACT

Invasive Computing is a research program that aims at developing a new paradigm to address the hardware- and software challenges of managing and using massively-parallel MPSoCs of the years 2020 and beyond. The program encompasses twelve projects from the areas of computer architecture, system software, programming systems, algorithm engineering and applications. The core idea is to let applications manage the available computing resources on a local scope and to provide means for a dynamic and fine-grained expansion and contraction of parallelism.

In this paper we present initial thoughts on operating-system support for the invasive paradigm and discuss first experimental results with run-time support for invasive programs.

1. INTRODUCTION

Multi-core architectures are actually yesterday's news in the parallel systems community. On the horizon, however, there arise many-core architectures with 10^3 and more processors on a chip[1]. These devices will be heterogeneous in terms of on-chip processors, communication facilities and memory organization. In the future, shared and distributed memory will coexist on a single chip. At a certain level, cache coherence is no longer implemented in hardware analog to Intel's single-chip cloud computer (SCC). With future many-core systems we will have so many cores available on a chip that every single thread will be able to run on his own private core. Single-threaded cores will be the normality, multi-threaded ones the exceptional case. This calls for a radical change in the way operating systems manage processors; it also calls for programming and system paradigms suited for very fine-grained parallelism.

These issues are addressed by the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89) funded by the German Research Foundation.[2] At the time being, the centre runs 12 projects from the areas computer architecture, system software, programming systems, algorithm engineering, and applications (high-performance computing, robotics). A total of 60+ researchers explore hard- and software of future massively parallel systems (*multi-processor systems on chip*, MPSoC) for the years 2020 and beyond. The research program is structured into three phases of four years each, with the current Phase 1, launched in Q3/2010, encompassing a funding volume of about € 9 millions.

This paper presents first thoughts on operating-system support for the parallel machines envisaged and developed as part of this program and also discusses preliminary results of experiments in run-time support for invasive computing.

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

[1]http://www.itrs.net

[2]http://www.invasic.de

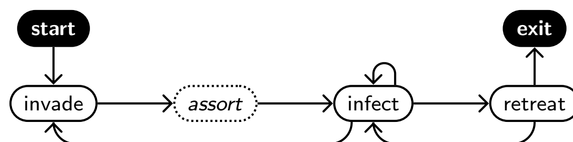

Figure 1: Structure of an invasive program

2. INVASIVE COMPUTING

One way to benefit from future MPSoCs, and bring their performance upward, is to have the running programs manage and coordinate the processing resources (i.e., processor cores, communication links, cache lines) themselves to a certain degree and in context of the local state of the underlying compute hardware. The notion of such a self-organising parallel program behaviour is invasive programming:

***Invasive Programming** denotes the capability of a program running on a parallel computer to request and temporarily claim processor, communication and memory resources in the neighbourhood of its actual computing environment, to then execute in parallel the given program using these claimed resources, and to be capable to subsequently free these resources again.* [11]

The dynamic and fine-grained *potential* parallelism of invasive programs calls for a radical change in the way the operating system manages its resources—in order to not become the bottleneck: Setup-times (for control-flow creation and distribution) have to be kept low and resources have to be managed locally wherever possible.

2.1 Programming Model

On the system level, invasive programming maps to three fundamental primitives: `invade()`, `infect()`, and `retreat()`. Fig. 1 shows theses primitives and the typical state transitions that occur during the execution of an invasive program:

First, an initial *claim* of resources has to be allocated by issuing a call to `invade()`. By *claim* we denote a set of processing elements, memory regions, and communication links the application can use to execute its parallel program. Claim requests to `invade()` are described in a declarative manner by a set of *claim properties* (e.g., the maximum available p processing elements with $p = 2^n$ that are not multiplexed) and may be fulfilled only partially.

Second, a problem-specific *assort()* function (provided by the compiler or algorithm developer) structures the parallel problem with respect to the actually invaded resources (the claim) by assembling a *team*. By *team* we denote a set of interrelated threads of execution of an invasive-parallel program.

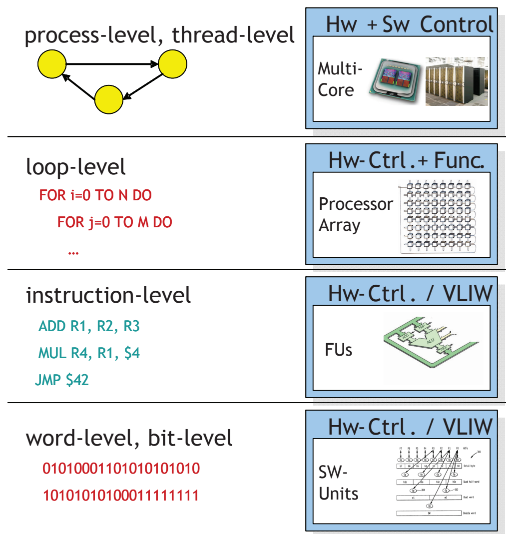

Figure 2: Levels of parallelism

Third, `infect()` is used to deploy and start the actual application code (the team) on the claim. During execution, the claim may be altered by calling `invade()` or `retreat()`, respectively to either expand or shrink the application's claim. Alternatively, when the execution finishes and the degree of parallelism has not changed, it is also feasible to dispatch a different team onto the same set of resources by issuing another call to `infect()`.

Finally, if a call to `retreat()` leaves the claim empty, there are no computing resources left for further execution of the program, hence it terminates its execution and exits.

Note that `invade()`, `infect()`, and `retreat()` are considered as the fundamental primitives of invasive programming at the *system level*. They are provided by the operating system and (to a certain degree) by the hardware that will be developed within the project. Application developers do not necessarily have to deal with these primitives. The goal is that points of potential parallelism are given as annotations or, in the ideal case, can even be deduced by the compiler. For this purpose, applications shall be developed in a high-level programming language for parallel computing, which most probably will be an extension of the X10 language [3].

2.2 Hardware Model

Over the last decade, we have seen the advent of a multitude of (typically domain-specific) massively parallel MPSoCs: Besides "traditional" multi-core CPUs we have GPUs, DSPs, or, as a recent development, network-on-chips, with the SCC as the most prominent example. These technologies exploit different levels of parallelism (Fig. 2): From traditional process- and thread-level parallelism (multi-core CPU) over loop-level parallelism (GPUs, tightly-coupled processor array (TCPA)) to instruction-level and even word-level parallelism (DSP, application-specific instruction set processor (ASIP)).

Invasive computing shall be investigated on all these levels and the MPSoCs going to be developed within the project will integrate all of them. The goal of the hardware development is to eventually merge these heterogeneous processing elements onto a single chip. Tiles comprising similar processing elements and memory will be interconnected by a common on-chip network.

Moreover, some of the hardware will provide dedicated support for invasive programs: On a TCPA employed for loop-level parallelism, for instance, regions of invasion may be managed directly by the hardware (Fig. 3), so that `invade()`, `infect()` and `retreat()` can be implemented with an overhead of just a few clock ticks. On a tile of multi-core CPUs for thread-level parallelism, the operating system has to provide for efficient abstractions in this respect.

3. SYSTEM DESIGN ISSUES

This section overviews design principles as well as concepts and abstractions of OctoPOS, the parallel operating system (POS) aimed at supporting invasive computing in a heterogeneous environment of parallel hardware beyond multi- or many-core processors.[3] As the project is still in its starting period, the following description is more a design rationale and development proposal rather than an experience report.

3.1 Overall Goals

Key aspect in the design and development of OctoPOS is to make all the capabilities of the underlying hardware available to higher (software) levels in an "unfiltrated" way—especially to application programs—and yet leave these levels gradationally hardware independent. For the "balancing act" between disclosure and concealment of hardware properties, system abstractions will be provided whose implementations result in equal-zero-overhead at run-time. Logically, these abstractions will be organised in a hierarchy of dedicated abstract machines. Physically, the boundaries between these machines need to become indistinct depending on the operating mode a given processing element (that is, real machine) shall be subjected to at user behest.

The resolution of abstract machine boundaries is enabled through state-of-the-art program generation tools as, for example, known from software product-line engineering. As a matter of fact, the system software going to be developed thus appears as a *family of operating systems* whose individual members implement tailor-made solutions of application-specific system functions for the domain of invasive programming. Customisation of system software will be achieved to a large extent by means of an aspect-aware design [5] and aspect-oriented programming [10]. An additional aim is to employ the concept of invasion as far as possible also at system level, for example, in the course of global resource management, in order to strive for highly scalable solutions regarding the parallel/distributed implementation of selected system functions.

The overall scientific objective is to define an operating system architecture in which *contention*, *latency* and *efficiency* is controllable by higher-level user/system functions in a problem-oriented manner. Herein, the challenge is to find lower-level (central) abstractions whose implementations are either free of or minimise contention, support latency hiding, and can be used to compose higher-level (peripheral) abstractions of gradually increased performance overheads depending on the application needs. The system software comprises a number of decentralised services for hardware-aware, dynamic, time- and resource-constrained loading, unloading and protection of threads on different types of parallel processing resources. Rather than developing all required system functions from scratch, a number of these shall be application-specific extensions of a standard UNIX-like host operating system such as Linux.

Generally speaking, OctoPOS provides all system-related mechanisms to partition, virtualise, and control parallel processors as

[3]Prefix "Octo" stems from the denotation of a nature which is highly parallel in its action as well as adaptable to its particular environment: the octopus, being able to act in parallel by means of its tentacle, adapt oneself through colour change, and, due to its highly developed nervous system, attune to dynamic environmental conditions and impact.

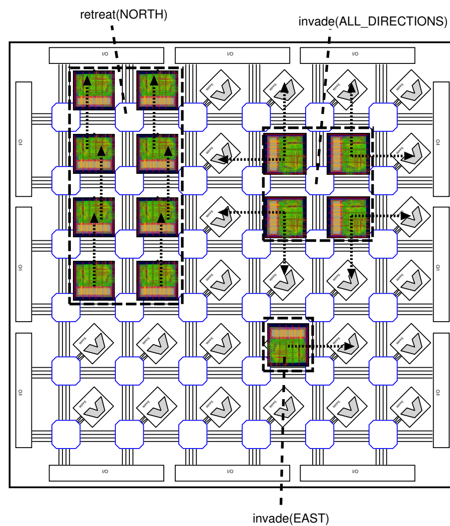

Figure 3: Three invasive programs on a TCPA

well as measures for contention prevention/avoidance, locality maintenance, and latency hiding and, in this respect, shares ideas with Corey [2], Barrelfish [9], and fos [12]. Concerning resource management against the background of invasive computing, strategic planning, allocation, and control will be handled by *software agents* [6, 1] acting on behalf of an application process. For a subset of abstractions arranged in the functional hierarchy of the system software, different implementations of the same functional interface will exist. These implementations vary in their non functional properties (e. g., with respect to contention, latency, or efficiency) and serve a fine-grained exposure to trade-offs arising from multiplexing hardware amongst competing invasive parallel processes.

3.2 Claims and Teams

For the purpose of complying with the operating system issues of invasive computing and, thus, supporting the execution of invasive-parallel programs, OctoPOS comes along with two fundamental concepts: *claim* and *team*. A claim represents a particular set of hardware resources made available to an invading application. Typically, a claim is a set of (tightly- or loosely-coupled) processing elements, but it may also describe memory or communication resources. Claims are hierarchically structured as to allow for the marshalling of homogeneous or heterogeneous clusters of processing elements. More specifically, a claim of processing elements also provides means for implementing a *place* [3], which is the concept of X10 to support a partitioned global address space. However, unlike places, claims do not only define a shared memory domain but also aim at providing a distributed-memory dimension.

In contrast, a team is the means of abstraction from a specific use of a particular claim in order to model some run-time behaviour as intended by a given application. Similar to conventional computing, where a process represents a program in execution, a team represents an invasive-parallel program in execution. Teams provide means for the clustering or arrangement of interrelated threads of execution forming an invasive-parallel program.

OctoPOS considers a team as the processing unit being subject to, for example, gang scheduling, co-scheduling, preemption, suspension, or load balancing [8]. At some higher level of abstraction, a team is additionally specified by an optional description of the dependencies between the members of the set. This way, by using the OctoPOS interface, a programmer or compiler will be able to provide an indication of the scheduling and coordinated execution of teams among themselves—for example, indications for the order of team execution to make explicit (programmed) synchronisation unnecessary, for anticipatory team release to enable pipelined execution, or for team-wise exclusive execution to mitigate memory, respectively cache contention. As this interface will be specifically designed to also ease compilers in the generation of efficient invasive-parallel machine programs, OctoPOS logically extends into the run-time system of the programming language employed.

3.3 Variability and Customization

The design and development of OctoPOS keeps in mind that the actual parameters of the primitives for invasive computing are assembled automatically, as far as possible, through static program analysis, if applicable. It is further assumed that a compiler possibly generates code patterns from respective higher-level language constructs supporting invasive-parallel programming (e. g., `async` [3]).

However, depending on precision and coverage of *a priori* knowledge available at programming or compile-time, not only the use pattern of these primitives may vary but also their semantics. This results in implementation variants not only of the same primitive but also OctoPOS functionality. There is no reason to trade off one case for another in this respect: Solutions for both are not only valid but also demanded by applications—and if one exercises a more detailed analysis of this problem domain, even more levels of abstractions will appear when reasoning about possible system-level implementations of these primitives (cf. below). Therefore, OctoPOS will be designed and developed as a *program family* [7]. Individual members of this family provide dedicated support in dependence on the properties and semantics of the three key primitives of invasive-parallel programming.

3.3.1 Claim Properties

The respective members of the OctoPOS program family are classified according to the claim attributes submitted to `invade()`. OctoPOS releases claims to an application in different shape, depending on the type and properties as specified by the requesting authority. The following properties can be given to an individual claim:

incoherent/coherent A claim is termed to be *coherent* if its elements have to reside within some physically defined proximity such as, for example, in case of a set of processing elements of some TCPA. A claim is *incoherent* otherwise.

heterogeneous/homogeneous A claim is termed to be *homogeneous* if all of its elements have to be of the same type such as, for example, given with a TCPA or Nehalem octa-core. A claim is *heterogeneous* otherwise, as exemplified by the Cell processor or an ASIP.

temporally preemptible/non preemptible A claim is termed to be *temporally preemptible* if virtualisation in time shall be possible to afford multiplexing between teams. In functional respect, revocation and reallocation of processing elements currently in use is unnoticed by execution threads. A claim is *temporally non preemptible* otherwise.

spatially preemptible/non preemptible A claim is termed to be *spatially preemptible* if virtualisation in space shall be possible to afford paging, swapping, or migration of teams. In functional respect, the replacement of areas in the physical address space of some processing element is unnoticed by execution threads. A claim is *spatially non preemptible* otherwise.

Actually, these types and properties will identify a specific level of abstraction in the *functional hierarchy* of OctoPOS. At the bottom of this hierarchy, an individual claim is coherent, homogeneous,

and non preemptible in temporal and spatial respects. This means the system software on this hierarchy level will be devoid of all functionality needed to support heterogeneity and preemption thus resulting in a slimmed down operating system interface and implementation that reduces execution overhead. The run-time behaviour of teams mapped onto such a simple claim is entirely dictated by the hardware.

Later on in Section 4 we will describe and evaluate such a slim implementation for a simple set of attributes. For applications, that are content with these attributes, it therefore can provide a fast execution platform.

3.3.2 Functional Enrichment

Passing through the hierarchy bottom-up, only a single claim property gets changed, but without hiding that very changed property. With each of the thus added properties, a *functional enrichment* of the respective level of abstraction goes hand in hand. In this sense, every single level forms a new "hardware machine" of well-defined properties in terms of the claims made possible by OctoPOS, respectively. In addition to other OctoPOS functions, each level in particular implements a variant of `invade()`, `infect()`, and `retreat()` covering exactly these attributes supported by this specific variant of OctoPOS. All of these variants are made available at the OctoPOS API in order to support execution of invasive-parallel programs as requested and in a problem-oriented manner. Ideally, the compiler makes the decision which sort of "hardware machine" (i. e., level of abstraction) executable code shall be generated by evaluating context information from static program analysis.

Picking up the thought on a minimal incremental change in claim properties implemented by OctoPOS, the level right above the "hardware machine" offering coherent claims solely adds incoherent claims. The next higher level adds heterogeneous claims, which is extended by temporally preemptible claims, which, in turn, extend to spatially preemptible claims. There are also *incidental attributes* that can be given to individual claims at `invade()`-time. An example is pinning of data such that, after an already executed `retreat()`, the next `infect()` applied to the same claim comes upon the data left by the previous computation phase.

At the very top, claims may also be subject to *preclaiming* and *reclaiming*. In the former case, claims are, at the latest, already known at program load time. OctoPOS will account for the resource requirements of the respective programs in advance and feeds corresponding (concurrent) processes to long-term scheduling. The latter case, reclaiming, concerns temporally or spatially non preemptible claims. These sorts of claims generally place severe restrictions on autonomous global resource management and self-organisation of (massively) parallel systems. In order to be able to freely manage the computing system in spite of non preemptible claims, OctoPOS will provide for optional functions to signal requests for claim release to the user level. Similar to interrupt requests of a real hardware machine, these signals may be disabled by higher (user) levels to temporarily or even permanently ignore release requests of the virtual "hardware machine" OctoPOS. However, once enabled, OctoPOS will be able to accomplish improved claim management on behalf and by assistance of the concurrent processes if claim virtualisation is not an option. Note that these reclaim signals will be mapped to exceptions by the X10 run-time system such that developers of invasive-parallel programs may employ high-level language exception handling concepts for convenient resource-aware programming.

3.4 Concurrency Playground

An architectural decision of OctoPOS that has a serious influence on contention and latency refers, for instance, to the question of how coordinated execution of a critical section by several execution threads is ensured. One option is to enforce sequential execution of such sections by falling back on one of the many implementation variants of the concept of mutual exclusion. In other words, consistency of shared data is ensured at the point in time the critical section is entered. As a consequence, execution threads are potentially blocked. This case relates to blocking synchronisation. Another option is to allow for concurrent execution, but not without falling back to a sophisticated procedure to ensure consistency of shared data at the point in time when the critical section is left. As a consequence, execution threads are never blocked—but they may suffer from starvation, depending on the *progress guarantee* of that procedure given to the system or individual execution threads. This case relates to non-blocking synchronisation.

Again another option is to render these explicit synchronisation measures unnecessary at all by exploiting *a priori* knowledge. This works with offline as well as online scheduling of execution threads (i. e., teams in terms of OctoPOS). However, while such an approach helps to let application-level programs forget critical sections, it does not necessarily relieve an execution platform from taking care of explicit (blocking/non-blocking) synchronisation interiorly. Although (top-down) *a priori* knowledge regarding team ordering, for example, may result in a deterministic schedule of the respective execution threads, unpredictable (bottom-up or middle-out) events nonetheless result in non deterministic system behaviour (i. e., thread executions at system level). All of these aspects cross-cut design and implementation of system software (such as OctoPOS), and they have a decisive impact on an operating system architecture.

Which of these different synchronisation concepts—or many implementation variants of these concepts—performs, respectively scales best for invasive computing in general depends on the use case as defined by a given application, on the operating mode of the computing machine, on its load, and of course on the actual hardware properties. In the specific situation of OctoPOS, this problem also depends on the execution pattern of an invasive-parallel program, on the number and structure of claims and teams, and on the data dependencies of the teams.

4. FIRST RESULTS

As a starting point the first prototype of OctoPOS is implemented on a traditional MPSoC system. The system implements a minimal set of claim properties (see Section 3.3.1): The computing cores are homogeneous and coherent, as they are all of the same type and access the same piece of memory with uniform access costs. The system itself does not implement preemption neither spatial nor temporal.

4.1 Hardware Setup

The basic layout of the MPSoC architecture used for evaluation consists of six SPARC LEON 3 processor cores and a DDR 2 memory module connected via a shared AMBA AHB-Bus.[4] The components are implemented in VHDL and are taken from the Gaisler *GRLIB IP Library* [4]. The individual SPARC LEON 3 cores are configured to use two-set 16 KB instruction and data caches and support the complete SPARC V8 instruction set including multiply and divide instructions.

The system is implemented on a Xilinx Virtex 5 XC5VLX110T FPGA. The processors and the interconnecting bus run at 80 MHz, the DDR2 memory at 190 MHz.

[4] We use this rather unusual hardware platform as it forms the basis for hardware development within SFB/TR 89 "Invasive Computing".

As memory access is done through a common bus by all cores, the evaluation system is a typical symmetric multiprocessing system with a uniform memory architecture. Code and data are located inside the same physical memory, so the system acts as a classical von-Neumann architecture.

4.2 Thread Model and Control

We chose not to implement a traditional threading scheme within our OctoPOS prototype, to most efficiently implement the core functionality of invasion. The teams used to model the single computing phases of the application are not implemented as fully fledged independent threads. Instead, they are modelled as functions. `infect()` dispatches one such team to the application's claim, which, in our implementation, simply comprises the set of cores the application currently uses. Once the execution of a team member finishes, the executing core is ready to accept members of another team.

The individual team members have run-to-completion semantics. So when they finish, the overhead to switch to a member of the next team is comparable to an indirect function call, as no context information of the preceding one has to be preserved.

One remaining issue is the synchronisation of the `invade()`, `retreat()` and `infect()` phases. As can be seen in Fig. 1, an `infect()` phase can always be followed by either an `invade()`, `retreat()` or another `infect()` phase. However, by calling `infect()`, a parallel portion of the program is started and then executes on all cores belonging to the claim simultaneously. When modifying the claim through `invade()` or `retreat()`, or dispatching a new team to the cores with `infect()`, the previously running program on all cores must be finished. To achieve this globally, some kind of barrier is needed to ensure the current computing phase is finished before the claim is modified or another computing phase starts. In our current implementation explicit calls to a barrier implementation are used inside the application code to achieve this synchronisation of the respective computing phases.

4.3 Basic Invasive Constructs

The core of the system software comprises the implementation of the basic invasive constructs `invade()`, `infect()` and `retreat()`. These provide the means to manage applications with a varying degree of parallelism.

Invade and retreat.

Both primitives have a similar interface: They take an integer parameter denoting the number of cores to be allocated or deallocated and return a boolean value indicating if the operation was successfully executed. Using `invade()` and `retreat()`, an application can adjust the number of cores it uses at run-time to match its varying degree of parallelism. This frees unused cores during less parallel periods which then can be used by other applications. The necessary bookkeeping is done by means of a free list that contains *core descriptors* for all currently unused cores in the system. The cores used by a specific application (i.e., its *claim*) are collected within a *claim descriptor*. Allocation of a core through `invade()` is now simply done by copying a reference to the core descriptor from the free list to the calling application's claim descriptor. Deallocation through `retreat()` is done accordingly.

Both operations can fail if there aren't enough cores available either in the free list in the case of `invade()` or in the claim descriptor in the case of `retreat()`.

Infect.

This primitive takes as argument a team describing the parallel application and distributes it to the cores comprising the application's claim. The dispatching functionality mainly consists of two distinct parts:

Δc	1	2	3	4	5
invade	31	42	52	62	70
retreat	33	43	52	62	71

Δc: Number of de-/allocated cores

k	1	2	3	4	5	6
infect	27	51	65	98	133	184

k: Number of infected cores

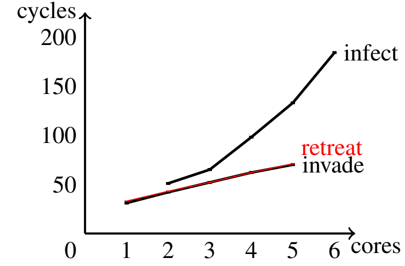

Figure 4: Execution times for invade, infect and retreat given in CPU cycles of 12.5 *ns*

1. Infection on behalf of the application by calling `infect()`. The producer part that feeds the team members into the dispatching algorithm.

2. By means of infection placed code running on all the cores of the system. The consumer part, as it receives and subsequently executes team members.

The individual team members in our implementation are realised as functions, hence the code portion of a team can be flexibly referenced by a set of function pointers. The data part can either be represented as function parameters, embedded into the function code or be addressed by other means, e.g. processor id.

These function pointers are the main item passed around inside the system to identify a certain team member: All cores in the system execute a loop querying a memory location for new team members in the form of function pointers. As soon as a new one is found, the function pointed to is executed and after its execution finishes, the core goes back to querying the memory location for its next team member.

Dispatching of a team to a set of cores is done by writing the team's set of function pointers to the memory locations queried by the cores. This design enables the programmer to use different functions for every core.

4.4 Setup-times of invade, infect and retreat

All measurements are taken on the platform described in Section 4.1 by means of toggling an external output pin to mark the areas of interest in the implementation and an oscilloscope. All measured data is averaged over thousand iterations.

The measurements shown in Fig. 4 represent the execution time of `invade()` and `retreat()` for the allocation/deallocation of Δc cores and the execution time for `infect()` of k cores. There are no test results with $\Delta c = 6$ for `invade()`, as the test cases start with a one element claim and increases it by Δc cores, meaning Δc cannot surpass five on our six-core machine. Same holds for `retreat()`, as the resulting claim always has at least one element left. In general, these numbers represent a best case scenario, as there is no real algorithm running in between the calls to the invasive primitives, so the system software can use the cache almost exclusively.

As can be seen in Fig. 4, both `invade()` and `retreat()` have strictly linear execution time behaviour depending only on the num-

ber of cores to be invaded or released. This is comprehensible, when looking at the implementation described in Section 4.3, that needs Δc copy operations for Δc function pointers.

The same should hold for `infect()`, as k function pointers have to be written for k cores. However, the results in Fig. 4 look different. This can be explained, when looking closer at the implementation details. Though `infect()` writes k function pointers to the memory locations of the claim's cores, on the receiving side there are k cores querying these memory locations. So for the function pointers to arrive at the destination core, the data has to travel over the bus connecting processors and memory. As this is a shared resource between all cores, it becomes a bottleneck, when infecting an increasing number of cores. This explains the non linear behaviour of `infect()` for a rising number of affected cores. Invade, however, only runs on a single core without the involvement of other cores and so has no scalability issue, as memory accesses mostly operate on the core's cache.

The numbers for `infect()` denote only the execution time on the infecting core. However, one has also to take into account the time it actually takes until the program starts running on the infected core. Therefore, we also measured the time passing from the infecting core writing the function pointer into memory to the moment the function actually starts executing on the infected core. This takes exactly 64 cycles within our implementation.

4.5 Discussion of the Results

Based on the constaints imposed by the testbed it is clear that it will not be possible to simply transfer our OctoPOS protoype to a 1000-core machine. However, it still can serve as a building block when supporting invasion for large scale MPSoCs comprising multiple tiles of processing elements. In this case inner-tile invasion is supported by means of the *minimal subset of OctoPOS functions* evaluated, while cross-tile (global) invasion will need *minimal OctoPOS extensions* (Section 3.3.2) which are going to be developed next in the project.

At the time being, even application and hardware development of the project is still at the beginning and, thus, does not support cross-tile invasion. At this stage and, generally, as long as the application does not need to cross tile borders, any additional operating system functionality entails unnecessary overhead—and contradicts the idea of an operating system family.

Another goal of our minimal implementation is to determine the lowest possible overhead introduced through the use of `invade()`, `infect()` and `retreat()`. This gives us a reference point against which we can measure future implementations gradually supporting more attributes (see Section 3.3.2)

This overhead of invasion determines at which point it pays off to parallelise the program by using invasion. When looking at systems with 1000 cores or more it is obvious that applications have to exploit as much parallelism as possible to efficiently use these architectures. The overhead generated by parallelisation is, in our implementation, determined by the execution time of `invade()`, `infect()` and `retreat()`. Therefore, if this overhead is as small as possible, it pays off to even parallelise loops comprising only a few instructions.

In general, it pays off to parallelise loops when the parallelised run-time of the loop $\frac{T}{p}$ for p cores including the overhead of invasion is smaller than the sequential run-time T: $\frac{T}{p} + \text{Overhead} < T$

The break even point for parallelisation can thus be computed using $T = \frac{\text{Overhead} \cdot p}{p-1}$

With the numbers obtained during our evaluation (Fig. 4), we can compute the overhead for different scenarios. Going from one to six cores takes $\text{invade}(5) + \text{infect}(6) + 64 = 318$ CPU cycles and then subsequently returning to a sequential execution takes $\text{retreat}(5) + \text{infect}(1) + 64 = 162$ CPU cycles. Hence, the overhead for this scenario is 480 CPU cycles. So it pays off to parallelise loops, as soon as their serial execution time surpasses $T = 480 \cdot 6/6 - 1 = 576$ CPU cycles.

To put these numbers into perspective, we did some measurements on a 16-core x86 machine with four sockets populated with Intel Xeon E7340 quadcore chips running at 2.40 Ghz and using Linux 2.6.32.21 as an operating system. We measured the time it takes to wake up six threads pinned to different processors using pthread mutexes. This took approximately 120000 CPU cycles. So for the parallelisation of a loop to be profitable the loop body would have to take at least $T = 120000 \cdot 6/6 - 1 = 144000$ CPU cycles. This shows that current traditional (general-purpose) operating systems set a rather high barrier for exploiting micro-parallelism. With a minimal (special-purpose) operating system however, it becomes feasible to accelerate even rather short loops through parallelisation.

5. SUMMARY

In this paper, we introduce OctoPOS, an operating system for invasive computing. Its focus lies on optimally using the underlying hardware depending on the properties of the application. With an operating system specifically tailored to the needs of the application it is possible to keep overheads low. This is particularly important in massively parallel systems, as it helps to avoid bottlenecks in the operating system implementation itself.

First results show that by using a very slim implementation it is possible to exploit even so-called micro-parallelism. This shows, that applications with low demand on the operating system indeed can profit from such an implementation.

6. REFERENCES

[1] AL FARUQUE, M. A., JAHN, J., EBI, T., AND HENKEL, J. Runtime thermal management using software agents for multi/many-core architectures. *IEEE Design & Test 27*, 6 (Nov./Dec. 2010), 58–68.

[2] BOYD-WICKIZIER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, M. F., MORRIS, R., PESTEREV, A., STEIN, L., ND MING WU, DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An operating system for many cores. In *OSDI '08* (Berkeley, CA, USA, 2008), USENIX Association, pp. 43–57.

[3] CHARLES, P., GROTHOFF, C., SARASWAT, V. A., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SHAKAR, V. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA 2005* (2005), R. E. Johnson and R. P. Gabriel, Eds., ACM, pp. 519–538.

[4] GAISLER, J. GRLIB IP Library User's Manual (Version 1.1. 0). *Gaisler Research* (2010).

[5] LOHMANN, D., HOFER, W., SCHRÖDER-PREIKSCHAT, W., STREICHER, J., AND SPINCZYK, O. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *USENIX'09* (Berkeley, CA, USA, 2009), USENIX Association, pp. 215–228.

[6] NWANA, H. S. Software agents: An overview. *Knowledge Engineering Review 11*, 2 (Sept. 1996), 1–40.

[7] PARNAS, D. L. On the design and development of program families. *IEEE Transactions on Software Engineering 2*, 1 (Jan. 1976), 1–9.

[8] SCHRÖDER-PREIKSCHAT, W. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, Upper Saddle River, NJ, USA, 1994.

[9] SCHÜPBACH, A., PETER, S., BAUMANN, A., ROSCOE, T., BARHAM, P., HARRIS, T., AND ISAACS, R. Embracing diversity in the Barrelfish manycore operating system. In *MMCS'08* (2008), K. Schwan, D. D. Silva, and M. Milenkovic, Eds., ACM Digital Library.

[10] SPINCZYK, O., AND LOHMANN, D. The design and implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software 20*, 7 (2007), 636–651.

[11] TEICH, J., HENKEL, J., HERKERSDORF, A., SCHMITT-LANDSIEDEL, D., SCHRÖDER-PREIKSCHAT, W., AND SNELTING, G. Invasive computing: An overview. In *Multiprocessor System-on-Chip – Hardware Design and Tool Integration*, M. Hübner and J. Becker, Eds. Springer, Berlin, Heidelberg, 2011, pp. 241–268.

[12] WENTZLAFF, D., AND AGARWAL, A. Factored operating systems (fos): The case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review 43*, 2 (Apr. 2009), 76–85.