Malleable Invasive Applications

Sebastian Buchwald Manuel Mohr Andreas Zwinkau
{sebastian.buchwald,manuel.mohr,andreas.zwinkau}@kit.edu

Karlsruhe Institute of Technology

Abstract

Invasive Computing enables a resource-aware programming style, which includes adapting to external resource changes similar to malleability. We introduce *asynchronously-malleable* applications, which can adapt at any time without synchronizing the whole application, in contrast to specific synchronization points. We show how master-slave applications meet the requirements for asynchronous malleability and how Invasive Computing supports it.

1 Introduction

Invasive Computing [12] is a new paradigm for designing and programming future tiled many-core systems. It uses a hardware-software co-design approach, where we design hardware, operating/runtime system, programming language and applications from the ground up with the goal of increased efficiency. The central idea is to write applications in a cooperating and resource-aware manner, where resource needs are explicitly communicated to the system[1]. Leveraging its global view, the system decides for the best action, and order the hardware and applications to adapt. In this paper, we will explore how you can write *malleable invasive* applications using the invasive language framework. We use the X10 programming language [1, 11], for its high-level features, usability, and integrated support for distributed systems which maps well to our tiled hardware architecture.

1.1 Invasive Language

An invasive [13] application features three steps: First, a function call to `invade` claims resources specified by various constraints. We call this set of resources a *claim* and assign the resources exclusively by default. Second, an `infect` call uses a claim's resources by executing *i*-lets on them. *i*-lets are self-contained units of computation that run to completion (although the operating system can block them). Third, `retreat` gives the claim's resources back to the system. Optionally, `reinvade` can change constraints in between and allows the system to adapt a claim. A vocabulary of constraints allows the programmer to express resource needs. Most common is the amount of processing elements (PEs) and a scalability hint, so the system can estimate how different applications benefit from additional PEs. You can also request exclusive communication channels [7], memory guarantees, or provide scheduling hints.

1.2 Malleability

There has been a considerable amount of work on malleable applications in high-performance computing, roughly starting with the work by Feitelson et al. [5] who coined the word *malleable* for applications that can adapt their degree of parallelism at runtime in response to external requests.

We demonstrated [2] that our invasive language is expressive enough to write typical iterative numerical algorithms, in this case a multigrid solver, in a malleable style. Here, we use `reinvade` to modify the resource

Submission to: 8. Arbeitstagung Programmiersprachen, Dresden, Germany, 18-Mar-2015, to appear at http://ceur-ws.org

[1] Here, "system" is synonymous with operating system, but the resource management might as well be in another layer or middleware.

```
val ilet = (id:IncarnationID)=>{
  for (job in queue) {
    if (queue.checkTermination(id)) break;
    job.do(); } }
val resizeHandler = (add:List[PE], remove:List[PE])=>{
  for (pe in add) queue.addWorker(pe,ilet);
  queue.adapt();
  for (pe in remove) queue.signalTermination(pe); }
val constraints = new PEQuantity(4,10)
  && new Malleable(resizeHandler)
  && new ScalabilityHint(speedupCurve);
val claim = Claim.invade(constraints);
queue.adaptTo(claim);
claim.infect(ilet);
claim.retreat();
```

Figure 1: **Example of a asynchronously-malleable invasive application** with a master-slave structure and a global queue of jobs. First, `ilet` states the actual computation to perform. Second, `resizeHandler` defines how to handle resource changes. 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). 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.

claim at specific points during each iteration. We ran multiple instances of the malleable multigrid solver concurrently and measured an improved throughput compared to concurrent non-malleable multigrid instances.

However, algorithms exist that are even more flexible regarding reconfiguration. Flick et al. [6] 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. For distinction, we label this *asynchronously-malleable*.

We argue that applications following a master-slave pattern always have this option, if creating and destroying slaves has low cost. Additionally, we propose an asynchronous programming style for handling reconfiguration.

2 Asynchronously-Malleable Invasive Application

In an asynchronously-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 asynchronously-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 asynchronously-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 1 shows an example application. The resize handler is a function, which takes a list of PEs that were added and a list of PEs that will get removed. It does not return a value but adapts the application as a side effect. When the system calls the handler in the example, it first starts additional i-lets according to its input. Then the PEs to remove get signaled. The i-let code checks this signal and just stops executing jobs if needed.

When the application has adapted to the change order, it notifies the system of the successful adaptation by returning from the handler function. The system updates its resource state and operating system. From the application programmer's point of view, the system 1) adds additional resources to the claim, 2) executes the resize handler, and 3) removes resources from the claim as planned. This means the resize handler can assume to own the union of the claim resources before and after resizing. This is necessary, because it must terminate i-lets on PEs to remove and also must start i-lets on new PEs. Also, infect must not finish while resizing. If `infect` finishes, the main activity will continue and for example retreat the claim. Meanwhile, there must be no resize handler running, which starts additional i-lets. Thus, the claim must track how many child activities are running and the resize handler counts as one of them. So, `infect` waits for all child activities to terminate

before it returns.

We considered to let the application decide which PEs to free, but dropped the idea. The system should handle such resource decisions exclusively, otherwise we end up with duplicated code and conflicting decisions. The system has a global view of resource needs and thus is in the best position for definitive decisions. It is the application's responsibility to receive information about the system's decisions and to adapt.

Our example contains a subtle circular dependency. A claim object requires constraints which in turn require the resize handler as an argument. This ensures via the type system that the programmer provides a resize handler whenever he declares malleability. However, the resize handler requires the claim object to start more i-lets. In Figure 1, we break this cycle via the global queue, which stores the claim it gets via `adaptTo` and provides the `addWorker` method.

The system shall handle errors in the resize handler in a safe manner. The handler might fail to adapt to the decided changes, e.g. by not terminating i-lets on PEs to remove. If the system ignores this "sit-in", it could give the PEs to another application, which finds itself unable to use them. Alternatively, the system could kill the blocking i-lets, but this might result in a deadlock of the whole application. So, the only sensible reaction is to kill the whole application, which failed to adapt to the decided changes. For another source of error, the (X10) resize handler might fail by throwing an exception. The system is not implemented in X10, so handling the exception is not an option. Like the first case, the application state is unknown at this point, so recovery is impossible and the system must kill the application. These error cases show the crucial role of the resize handler. Reproducing and debugging errors is hard, because a call to the resize handler depends on the behavior of other applications and is effectively non-deterministic.

Our system does not support virtualization (preemption, address spaces, etc) and inter-tile migration of i-lets so far. Thus, the system cannot remove an application from a tile during `reinvade`. The application would lose tile-local data without a chance to copy it elsewhere. So, the system ensures that at least one processing element per tile remains. However, the resize handler provides a migration chance, so asynchronously-malleable claims can lose tiles.

3 Related Work

Adaptive MPI [8, 9] uses processor virtualization to achieve malleability. Computations are divided into a large number of virtual MPI processes, which are mapped to the physical resources via a runtime system with object migration support. In contrast, we assign resources exclusively by default and programs cannot rely on resource virtualization to hide underutilization.

Leopold et al. [10] present a case study of making an existing MPI application malleable. The studied application is an iterative numerical simulation using a master-slave design. The authors extend this design with a server process that runs concurrently to the master and keeps track of newly started started slave processes during runtime; removing slaves is not handled. New slaves become usable to the master starting with the next iteration.

Maghraoui et al. [3, 4] 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 and handle migration via saving and restoring the current state.

In contrast to this work, we do not constrain ourselves to iterative applications, as shown by the asynchronously-malleable sorting example, nor do we have to support legacy MPI applications. Furthermore, we look at malleable applications in the context of many-core architectures on a single chip whereas prior work focused distributed high-performance computing.

4 Future Work

Our design proposal is improvable. For example, much of the application boilerplate or even the whole queue implementation should be generalized into the framework.

The termination signal in our example is setting a flag across a distributed system. It might pay off to use special (hardware or OS) mechanisms to speed this up. Furthermore, should the resize handler wait for the termination to succeed? If a PE keeps executing code, while the system assigns it to another application, this might or might not pose a problem.

Acknowledgments

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Invasive Computing" (SFB/TR 89). Special thanks go to Jochen Speck, Sebastian Kobbe, and Martin Schreiber for extensive discussions about this. We also thank the anonymous reviewers for their suggestions.

References

[1] M. Braun, S. Buchwald, M. Mohr, and A. Zwinkau. Dynamic X10: Resource-Aware Programming for Higher Efficiency. Technical Report 8, Karlsruhe Institute of Technology, 2014. X10 '14.

[2] H.-J. Bungartz, C. Riesinger, M. Schreiber, G. Snelting, and A. Zwinkau. Invasive computing in HPC with X10. In *Proceedings of the third ACM SIGPLAN X10 Workshop*, X10 '13, pages 12–19. ACM, 2013.

[3] K. El Maghraoui, T. Desell, B. Szymanski, and C. Varela. Dynamic Malleability in Iterative MPI Applications. In *Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 591–598, May 2007.

[4] K. El Maghraoui, T. J. Desell, B. K. Szymanski, and C. A. Varela. Malleable Iterative MPI Applications. volume 21, pages 393–413, Mar. 2009.

[5] D. G. Feitelson and L. Rudolph. Towards convergence in job schedulers for parallel supercomputers. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS '96, pages 1–26, 1996.

[6] P. Flick, P. Sanders, and J. Speck. Malleable Sorting. *IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 418–426, May 2013.

[7] J. Heisswolf, A. Zaib, A. Zwinkau, S. Kobbe, A. Weichslgartner, J. Teich, J. Henkel, G. Snelting, A. Herkersdorf, and J. Becker. CAP: Communication Aware Programming. In *Design Automation Conference (DAC), 2014 51th ACM / EDAC / IEEE*, 2014.

[8] C. Huang, O. Lawlor, and L. Kal. Adaptive MPI. In *Languages and Compilers for Parallel Computing*, volume 2958 of *Lecture Notes in Computer Science*, pages 306–322. Springer Berlin Heidelberg, 2004.

[9] C. Huang, G. Zheng, L. Kalé, and S. 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.

[10] C. Leopold, M. Süß, and J. Breitbart. Programming for malleability with hybrid MPI-2 and OpenMP: Experiences with a simulation program for global water prognosis. *Proceedings of the European Conference on Modelling and Simulation*, pages 665–670, 2006.

[11] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 Language Specification. Technical report, IBM, February 2014.

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

[13] A. Zwinkau, S. Buchwald, and G. Snelting. InvadeX10 Documentation v0.5. Technical Report 7, Karlsruhe Institute of Technology, 2013.