Kent Academic Repository
Full text document (pdf)

Citation for published version

Marr, Stefan and Haupt, Michael and Timbermont, Stijn and Adams, Bram and D'Hondt, Theo and Costanza, Pascal and De Meuter, Wolfgang (2010) Virtual Machine Support for Many-Core Architectures: Decoupling Abstract From Concrete Concurrency Models. In: Second International Workshop on Programming Languages Approaches to Concurrency and Communication-cEntric

DOI

https://doi.org/10.4204/EPTCS.17.6

Link to record in KAR

http://kar.kent.ac.uk/63849/

Document Version

Author's Accepted Manuscript

Virtual Machine Support for Many-Core Architectures: Decoupling Abstract from Concrete Concurrency Models

Stefan Marr[1]*, Michael Haupt[2], Stijn Timbermont[1]*, Bram Adams[3]
Theo D'Hondt[1], Pascal Costanza[1], Wolfgang De Meuter[1]

[1]Programming Technology Lab
Vrije Universiteit Brussel, Belgium

[2]Hasso Plattner Institute
University of Potsdam, Germany

[3]Software Analysis and Intelligence Lab
Queen's University, Canada

The upcoming many-core architectures require software developers to exploit concurrency to utilize available computational power. Today's high-level language virtual machines (VMs), which are a cornerstone of software development, do not provide sufficient abstraction for concurrency concepts. We analyze concrete and abstract concurrency models and identify the challenges they impose for VMs. To provide sufficient concurrency support in VMs, we propose to integrate concurrency operations into VM instruction sets.

Since there will always be VMs optimized for special purposes, our goal is to develop a methodology to design instruction sets with concurrency support. Therefore, we also propose a list of trade-offs that have to be investigated to advise the design of such instruction sets.

As a first experiment, we implemented one instruction set extension for shared memory and one for non-shared memory concurrency. From our experimental results, we derived a list of requirements for a full-grown experimental environment for further research.

1 Motivation

With the arrival of many-core architectures, the variance of processors increases by another order of magnitude. This variance increases also the need for high-level language virtual machines (VMs) to abstract from variations introduced by differences among many-core architectures [19, 35, 43, 44]. We are concerned with processors having multiple cores, using non-uniform memory access architectures, and explicit mechanisms for inter-core communication.

For software developers, VMs have to provide abstractions from concrete hardware details like number of cores or memory access characteristics. In the following subsection, we categorize three groups of hardware architectures, which need to be supported by VMs, as *concrete concurrency models*. In contrast to those concrete concurrency models, we refer to the concurrency models defined by languages or libraries and used by application developers as *abstract concurrency models*. Our claim is that the currently available incarnations of abstract concurrency models in the form of languages and libraries

---

are not sufficient and need to be complemented by inherent support for multiple concurrency models by VMs.

To motivate our proposal, we analyze the challenges for VMs with regard to concrete as well as abstract concurrency models in the remainder of this section.

The remainder of this paper discusses our idea of an instruction set for concurrency and the research that has to be conducted to develop a methodology which allows to tailor such an instruction set for the needs of a specific VM and its application domain. We give a brief overview of our initial experiments and present the conclusions for a full-grown experimental environment. We also discuss the related work which contributs approaches and solutions to VMs for many-core architectures.

1.1 Challenges for VMs on Modern Processor Architectures

Since processor vendors reached an upper bound on the possible clock speed to gain more performance, the design of modern processor architectures diverges from their predecessors in central design elements with each new generation, trying to achieve better performance by introducing support for explicit concurrency.

This trend has much different consequences compared to the gradual architectural changes over the last decade. Instead of increasing the complexity of the memory hierarchy to hide latency and bandwidth issues, introducing out-of-order execution of instructions, or simply raising clock rates, changes are made which are not transparent to software anymore and require special support. As detailed in the remainder of this section, the *memory access characteristics* change, the explicit concurrency increases the need for *cache-conscious* programming, and some architecture introduce explicit inter-core communication which all needs to be support by VMs.

As already mentioned before, we refer to the concurrency models provided at the hardware level as *concrete concurrency models*. We identified three models and the challenges they imply for the implementation of VMs.

1.1.1 Single-core Processor

The most fundamental *concrete concurrency models* is a single-core system accessing memory not shared with another processor. In such a system, the only notion of concurrency is provided by the operating system (OS) offering some form of preemptive thread scheduling.

Modern single-core architectures usually use mechanisms like out-of-order execution of instructions, vector instructions, or pipeline steps which can lead to parallel execution of small code portions. However for VM implementations, these forms of parallelism do not impose additional complexity. It is not necessary to introduce a concurrent garbage collector, but a just-in-time (JIT) compiler could still benefit from these mechanisms.

However, for optimal performance, these architectures put another burden on programmers. Deep cache hierarchies have to be treated carefully for optimal performance, i.e., programmers have to be *cache-conscious*. Thus, they are responsible for reorganizing data layouts to avoid phenomena like cache thrashing and support the prefetching heuristics. JIT compilers could actively use characteristics like cache line sizes, prefetching heuristics, and branch prediction of the various hardware architectures for optimization [8,14], and interpreters could be adapted, e. g., to assist hardware branch prediction [4].

With respect to concurrency provided by the OS, a VM has to define a memory model [33] and a task model. The memory model specifies, amongst others, when a write to a shared variable by one thread can be seen by reads done by another thread. These guarantees interact in various ways with JIT compiler

optimizations, like storing temporary values in registers, and OS thread scheduling, since the guarantees need to be enforced before a thread can be rescheduled. The best performance is usually achieved if guarantees are less strong and provide opportunities for reordering to hide memory latency.

The task model makes concurrency available to language developers and should allow to schedule tasks with respect to the used data, to use caches efficiently if tasks, e. g., in the form of threads, operate on shared data.

1.1.2   Multi-core Processor

The second concrete concurrency model is a shared memory approach for multi-core or hardware multi-threaded systems. To allow a clear distinction to many-core processors (see below), we will concentrate on systems with an architecture for uniform memory access (UMA)[1], i. e., multiple cores or threads connected to a single main memory system and a cache hierarchy which provides cache coherency.

These architectures have grown from single-core processors and usually share all important characteristics like deep cache hierarchies and out-of-order execution. The main difference is the additionally provided hardware concurrency and cache coherence.

The guarantees given by the memory model are even more important in this case. Here it is not only arbitrary interleaving but parallel execution which has to be taken into account. Overly strict guarantees will require that writes are followed by memory barriers to ensure that neither instruction-reordering nor the cache hierarchies are hiding changes at any given time. This will of course hurt performance since both mechanisms could be practically disabled.

By introducing cache coherency, the appropriate utilization of the available hardware mechanisms becomes more complex. One example is given by Herlihy and Shavit [14]. They discuss different lock implementations with the basic insight that a synchronizing operation like *compare-and-swap* provided by the processor might hurt performance if used inappropriately. Combined with a simple read operation which checks whether the value has changed utilizing caching, performance can be improved, since relying on cache coherence has less overhead than an operation which might need to synchronize different cores explicitly and causes memory operations which cannot be cached. This insight is not only important for the implementation of synchronization primitives provided by VMs, but also for the implementation of JIT compilers to generate efficient code.

Similar to single-core systems, task scheduling should respect data dependencies. For multi-core systems, scheduling should also be aware of the cache architecture, i. e., how cores share caches and how caches are connected to a hierarchy, to avoid cache thrashing or rather exploit caching efficiently.

1.1.3   Many-core Processors

In contrast to multi-core processors, many-core processors cannot rely on a UMA architecture anymore since the known mechanisms do not scale [27]. Instead, these processors rely on non-uniform memory access (NUMA) architectures, i. e., the cost to access a specific memory location can be different for all cores. Furthermore, some architectures will provide explicit communication facilities between cores and thus will not rely solely on shared memory for direct communication. Others will try to avoid this additional complexity. However, many-core architectures which provide shared memory and coherent caches will exhibit performance behavior which will vary with respect to data locality.

We will discuss three candidates from this category briefly.

---

[1] Often UMA systems are regarded as symmetric multiprocessing (SMP) systems, however, for this discussion, the memory architecture is the main point of interest and the actually utilizations of the cores is subordinated.

Cell BE   The Cell BE [19] is already in wide use for media systems as well as for scientific computing. One of the major characteristics of the Cell BE is its heterogeneous approach to combine a central processing element with multiple *synergistic processing elements* (SPE) to offload computational intensive tasks. The SPEs are very simple and are not part of a cache hierarchy, do not feature out-of-order execution, or even branch prediction. Each one has a local storage but cannot access main memory directly. Instead, a SPE has to request blocks of memory to be copied into its local store before it can use the data.

The interconnection of these cores is realized by a ring bus architecture. Here the physical locality is important to achieve optimal performance. The ring bus is build from four rings, where two rings can transfer data clockwise and the other two can transfer data counter clockwise. A more detailed overview of this architecture is given by Krolak[2].

TILE64   The processors produced by Tilera, e. g., the TILE64 [44] are somehow similar to the SPEs with regard to their simplistic design. However, the TILE64 is a homogenous system with only one type of cores. Each of the 64 cores has a small cache and is interconnected with neighboring cores (tiles) via a mesh network with five independent special purpose networks. Thus, to access memory, a core uses the *memory dynamic network* which transports the request to the according memory controller and returns the data. Furthermore, an *inter-cache network* allows to access the local caches of other cores. Additional inter-core communication networks allow various direct communication schemes between cores.

The challenges to implement VMs on top of such a system have been documented by Ungar and Adams [40]. The crucial obstacles they encountered where very small local caches, inefficient communication due to shared memory (as opposed to explicit core-to-core communication), and required replication of immutable objects to be cached locally since the processors cache coherency protocol allows caching of a page only on its home core. From these observations, we conclude that adequate strategies will be required to implement object heaps enforced by very small caches, as well as an appropriate way to harness the available bandwidth for inter-core communication to reach the theoretical performance maximum.

Larrabee   Intel's Larrabee [35] represents another possible homogenous design. Similar to the other two designs, the cores itself are much simpler than, for instance, the latest designs used in desktop computers. They use an in-order architecture extended by wide vector processing units since it is primarily designed as a graphics processor.

However, in contrast to the other designs, Intel has decided to go with a cache coherent system to hide some of the complexity. Each core has its own local subset of the L2 cache and accesses main memory via the coherent L2 cache using a ring network. At the moment, it seems that they will not expose this ring network explicitly and communication is only done via shared memory. Nonetheless, the performance characteristics will differ drastically from standard multi-core system especially for systems with more than 16 cores where multiple short linked rings will be used.

1.2   Challenges for Abstract Concurrency Models

Today's *abstract concurrency models* are commonly regarded as not ideal and a lot research is conducted to improve this situation with different approaches. In short, shared memory with locking is too compli-

---

[2]`http://www.ibm.com/developerworks/power/library/pa-fpfeib/`, Version: 29 Nov 2005

cated [24] and software transactional memory (STM) [36] as well as Actors [1, 15] are promising but not widely adopted.

Thus, we expect that ongoing efforts in building languages, to handle the inherent concurrency of many-core systems, will likely lead to domain-specific languages and will require support by the underlying VMs. In this regard, VMs like the *Java Virtual Machine* (JVM) and the *Common Language Infrastructure* (CLI) are becoming more important as common execution platforms for multiple languages, since not only the implementation of JIT compilers and efficient garbage collectors is a tremendous effort, but the ability to reuse the existing infrastructure surrounding a VM is an economical concern.

Realistically, there will not be one single model for expressing concurrency. Thus, we argue that a VM has to provide support for a wide range of concurrency models at its core. Very likely, developers will have to deal with several models; e. g., in relation with legacy code requiring proper support. Furthermore, support for a wide range of models eases the work of language designers to implement new ideas or domain-specific solutions. VM developers can also benefit from richer concurrency semantics, as it would enable efficient implementations of different abstract concurrency models on top of the concrete models.

To illustrate our argument, we will discuss the example of the actor model [15].

The JVM and CLI are both widely used and host all kinds of different programming models. Functional as well as imperative languages and in the recent past they started to provide support for dynamic languages, too. However, if it comes to concurrency, both support only a shared-memory model with threads and locks.

The implementation of an actor-based concurrency model, like it is found for instance in Erlang [41], on top of these VMs has turned out to be a tough problem. Karmani et al. [20] surveyed different language and library implementations of actor models on top of the JVM. They observed that only few of them actually implement a model which preserves properties like isolation so that actors never share any state in terms of references to a common object graph. The few ones which do, usually rely on inefficient mechanisms like serializing the object graph which is then send as a copy. A VM could provide support for much more efficient zero-copying strategies and enforce the desired properties of the actor model at the same time.

1.3 Conclusions

The presented concrete concurrency models represent actual hardware architectures which differ widely. The important characteristics are their cache hierarchies, memory access architectures, the provided form of concurrency, and means for communication between cores.

Theses characteristics influence not only various implementation details all over the VM but affect the optimal design of memory, task, and communication model for each of the different concrete concurrency models. For example, the challenge for VMs on many-core architectures is not solely the utilization of available hardware concurrency but also to use the provided memory and communication facilities appropriately. Thus, VMs' concurrency abstraction layers must enable efficient implementations on top of the different concrete concurrency models.

To achieve that, we argue that VMs should provide explicit and comprehensive support for concurrency. Explicit support for the various different abstract concurrency models would allow direct mappings from congruent models which will allow an efficient utilization of the available facilities and would ease the task to find a suitable mapping for the remaining, not directly supported concepts.

For instance, the discussed actor model offers opportunities for an efficient mapping onto many-core architectures. Since cache coherence is an issues in these architectures, it would be possible to use

shared-memory only for immutable global state. The state of single actors could be stored in distinct parts of the memory, so that false sharing is avoided and the small local caches can reach peak efficiency. In a standard JVM, it would be rather hard to reconstruct the necessary semantics for such a mapping from the bytecode, but a semantically enriched instruction set could would allow a JIT compiler to apply such optimizations.

2   VM Instruction Sets with Concurrency Support

Our proposal to achieve a concurrency abstraction layer is to extend the VM instruction set by concurrency operations. Such an instruction set will decouple the concurrency models on the different levels of implementation in such a way that they can be varied independently. Fig. 1 visualizes this idea by showing three different abstract concurrency models mapped to an instruction set with explicit concurrency support implemented on top of three different concrete concurrency models.
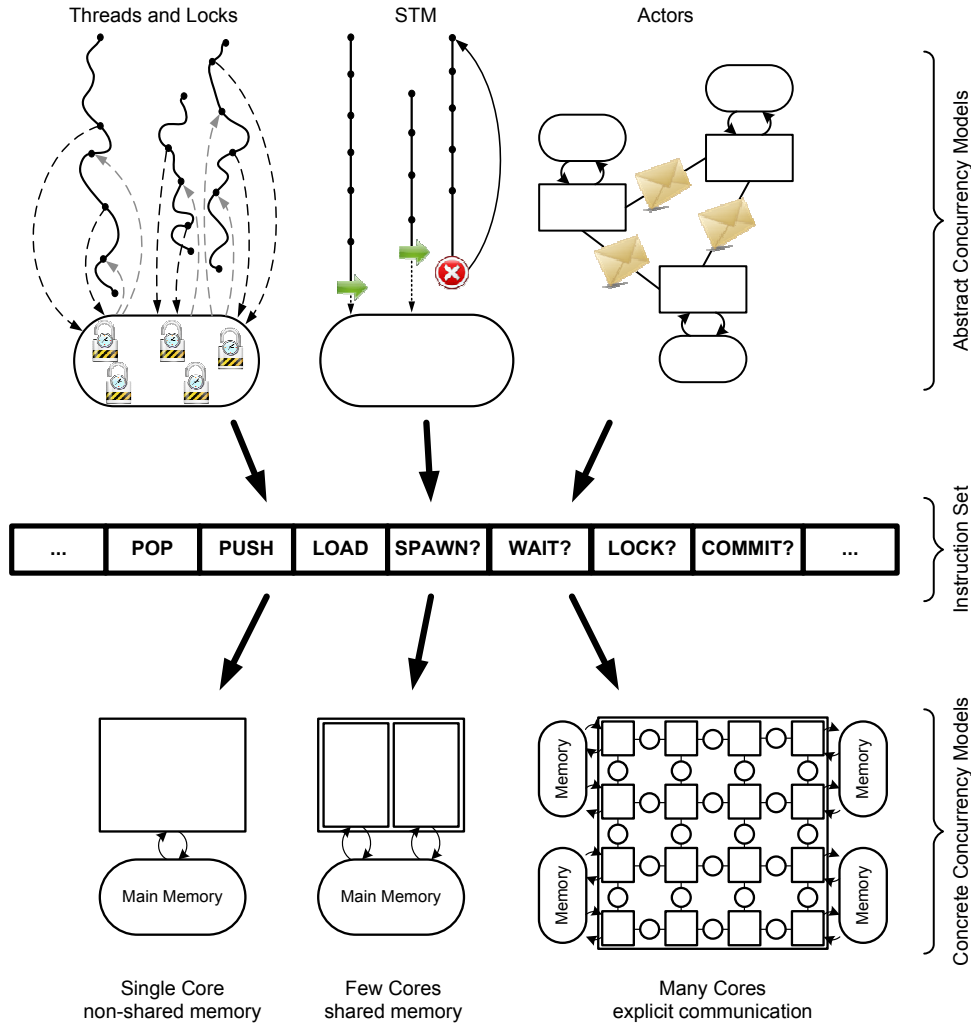

Figure 1: A VM instruction set as abstraction layer between abstract and concrete concurrency.

Expressing concurrency in the instruction set instead of using libraries has two major advantages. First, it will be possible to compile concurrency-related language constructs directly to these instructions, avoiding dependencies between languages and libraries on top of the VM. Second, this choice leads to a larger optimization potential at the VM level, e. g., for JIT compilation, which benefits from the instruction set's precise semantics.

Since there will not be a single instruction set matching all possible requirements, we will work on one instruction set representing a very generic set of requirements, and investigate the design tradeoffs to derive design advice for more concrete requirements as well. Thus, we plan to devise a methodology to develop VM instruction sets with inherent concurrency support, enabling VM designers to build a concurrency abstraction layer optimized for their particular requirements.

The methodology will describe how to decouple abstract and concrete concurrency. Language designers will be provided with a strategy to map abstract concurrency models to instruction sets, and VM implementers will be enabled to implement instruction sets efficiently on top of concrete concurrency models. The methodology will not only guide such undertakings, but will also give an impression on the effort necessary for their realization. Below, we discuss our approach in more detail.

2.1 Approach to Synthesize the Instruction Set

To devise a broadly applicable methodology, we decided to adopt a step-wise approach to designing a general instruction set and discovering the important design tradeoffs. The three currently most important concurrency models are significantly different in how they represent and realize concurrency: shared-memory with locking, STM, and actors. For each of these, we will survey different incarnations in languages or libraries to find each model's set of primitives relevant for an instruction set.

Potential candidates for examination are, to name just a few, Java [11], C# [18], Smalltalk [10], Cilk/JCilk [2, 7], and frameworks like Fork/Join for Java [23]. Furthermore, constructs like monitors and semaphores are considered as well [39]. In the field of STM, we currently consider the work of Shavit and Touitou [36], Ziarek et al. [47], Saha et al. [32], and Marathe et al. [25]. In the world of actor models the work of Hewitt et al. [15] and Agha [1] as well as the languages Erlang [41], Scala [13], and Kilim [38] are considered starting points.

2.2 Ideas to Combine Abstract Concurrency Models

One of the major research challenges will be to find appropriate combinations of the different abstract concurrency models. The idea is not to build an instruction set which is a simple enumeration of primitives for the different models, but instead an elaborated combination thereof. Thus, the interaction between different models has to be completely understood and defined, too.

Our ideas for model combinations are based on the following work. Volos et al. [42] and Blundell et al. [3] have described possible solutions for combining locking based code with STM. A combination of locking based code and actors is described by Van Cutsem et al. [6]. STM has many similarities with common transaction processing systems; thus, we will investigate the application of transaction processing monitors [12] as used in distributed settings to use STM in conjunction with actors.

2.3 Tradeoffs to be Investigated

For the methodology, the discussion of the following design tradeoffs will be an important part.

Model Combination: Different solutions to combine concurrency models on the instruction set level will be considered, and their benefits and drawbacks investigated. This will reveal critical details like incompatibilities and the possible degree of concurrency.

Model Mapping: Strategies to map the concurrency models preserved in the instruction set onto concrete concurrency models. Here the differences in the memory models, cache hierarchies, and communication mechanisms have to be considered.

Condensed vs. Bloated Instruction Set: Only few instructions should be added to avoid exceeding the limited number of instructions in a typical bytecode set. However, additional semantics in the instruction set could reduce the complexity of implementing an abstract concurrency model on top of it. It can also be beneficial for an efficient mapping to a concrete concurrency model. Since language and VM implementations should be reasonably manageable, these conflicting interests have to be investigated.

Bytecode vs. High-level Representation: Currently, bytecode sets are the most common representation for VM instruction sets. With respect to communication centric many-core architectures, we will investigate the potential of abstract syntax tree-like high-level representations of interpretable code in terms of reducing the implementation effort for new instructions and JIT compilers. These investigations will be based on the work of Kistler and Franz [22].

Instruction Set vs. Standard Library: A strategy, to decide which concepts are valuable in the instruction set itself, e. g., by facilitating JIT compiler optimizations, and which are less common or less fundamental and should be provided only in the standard library for a given application domain, is necessary, too.

3   Initial Experiments

For our first basic experiments we used SOM++[3], a very simple VM, implementing a Smalltalk-like language. This VM is designed to be used for teaching and to prototype ideas rapidly.

Originally, it has a very small instruction set (16 instructions) and features a straightforward bytecode interpreter. Its overall design favors simplicity over performance and utilizes C++ to provide an object-oriented implementation. This results in a VM implementation which emphasizes conceptual clarity. Thus, experiments usually require a minimal effort. The downside of this approach is, that SOM++ is considered unoptimized with regard to performance. Hence, experiments on SOM++ are useful to show the general impact of different implementation strategies, for instance for garbage collection, but only provide a rough estimate about performance and interaction effects between subsystems.

In the context of our first experiments, this is not an issue. The goal was to gain an impression of the general impact of introducing concurrency related instructions into the bytecode set of a virtual machine. In our experiments, we chose to focus on shared memory and non-shared memory concurrency in the first place.

The foundation for these experiments is the SOM++ bytecode set. As mentioned before, it consists of 16 instructions. It is purely stack-based and design with simplicity as the main goal in mind. Thus, the bytecodes are encoded as bytes with the values from 0 to 15. Even though it would be possible to encode arguments—e. g., indexes for local variables or symbols—within the remaining bits, they are provided as an additional byte each. Thus, bytecode instruction length varies in the range from 1 to 3. The bytecode set is outlined in Tab. 1.

---

[3]http://hpi.uni-potsdam.de/swa/projects/som/

`DUP`	duplicate top element
`PUSH_*`	push locals, arguments, fields, blocks, constants, and globals onto stack
`POP`	remove top element
`POP_*`	pop top element to locals, arguments, and field variables
`SEND sig`	send a message identified by `sig` to the top element
`SUPER_SEND`	send a message to the top element, use implementation of the parent class
`RETURN_LOCAL`	return from the current block of execution to its outer context
`RETURN_NON_LOCAL`	leave the currently executed method from an inner block
`HALT`	leave the interpreter loop

Table 1: SOM++ bytecode set

In the following sections, we briefly describe the two experiments, to illustrate potential concurrency related instructions in VM bytecode sets.

3.1 Shared Memory Concurrency

Our very first experiment was to add basic instructions for shared-memory concurrency to the SOM++ bytecode set. We designed the extension similar to the existing instructions. Simplicity was not the foremost concern here, but we have chosen to add only the five basic instructions outlined in Tab. 2. They operate on the top element of the execution stack. For SPAWN, the top element has to be a block which is then executed in a new thread. As a result, SPAWN pushes a new thread object onto the stack. The other four operate on an arbitrary object on the top of the stack. The stack itself is not affected.

We relied on an existing implementation of shared-memory concurrency using the *Pthreads* library. Thus, the largest part of the work was refactoring the existing implementation from primitives, i. e., native functions for the Smalltalk thread library to bytecode instructions. Subsequently, the SOM++ compiler was adapted to emit the new bytecodes on special messages.

`SPAWN`	spawn a new thread with the given block on top of the stack
`LOCK`	lock the lock of the top element
`UNLOCK`	unlock the lock of the top element
`WAIT`	wait on a notification on the top element
`NOTIFY`	notify all threads waiting on the top element

Table 2: Additional instructions for shared memory concurrency

In the context of SOM++, the question arose whether it is beneficial to have these instructions in the instruction set instead of implementing them as primitives. In the course of this project, bytecode instructions are actually the only option, which however brought about considerable overhead in implementing the required extensions in the compiler.

However, SOM++ is not the type of virtual machine we like to target with such extensions. Instead, these kinds of instructions are meant for multi-language virtual machines. Here, the purpose of an instruction set shifts from being a runtime representation of a program to being a full-fledged assembly language for all kinds of language implementations. Thus, a richer instruction set allows to move implementation effort from the language-level, which has to be redone for each language, to the platform-level where all language implementations can benefit from it without additional effort.

For future experiments we will consider additional shared-memory operations to increase the flexibility and expressiveness of the instruction set. At the moment, we think that several low-level operations known from hardware instruction set architectures could be useful additions to allow language designers for instance to use lock-free synchronization mechanisms or data structures at the heard of their languages.

Examples for such operations are atomic updates like XADD and *compare-and-swap* (CMPXCHG) from the IA-32 instruction set architecture [17], as well as operations like *load-and-reserve/store-conditional* which are included in the PowerPC instruction set architecture [9] in form of lwarx and stwcx.

3.2   Non-shared Memory Concurrency

The second experiment we conducted was inspired by the work of Schippers et al. [34] describing an actor-based machine model. The aim of this experiment was to adapt SOM++ to implement concurrency by actors which do not share memory, but use explicit message passing for communication. This kind of machine model is typically found in distributed object systems [31].

In this model, actors are containers for objects. It is derived from the notion of *vats* introduced in the E language (and its predecessors) [28] where actors are not "active objects", but containers for a number of regular objects. The contained objects are shielded from undesired concurrent modifications, since each actor only has a single thread of control. Messages between actors are exchanged using an incoming message queue per actor. Objects can reference objects located in another actor by means of remote references. Usual message sends between objects can be synchronous or asynchronous, independent from whether the message is sent locally or over a remote reference.

Inside an actor, coroutines are allowed to support a simple means of concurrency. This is useful since synchronous message sends over remote references do not block the sending actor, but can yield control to another coroutine until the return message is received.

To support this machine model, the instruction set had to be adapted as outlined in Tab. 3. The basic instructions stay the same except for SEND. For message sends to objects over remote pointers, SEND was adapted. It forwards the message sent to the actor owning the object and yields the coroutine to wait for the result value. The result value is later returned by the RETURN_REMOTE bytecode. Usual asynchronous message sends are realized by the SEND_ASYNC bytecode and coroutines can explicitly yield control using the YIELD bytecode.

SEND	sends of remote references yield coroutine and wait for return value
RETURN_REMOTE	sends the return value to the waiting coroutine
SEND_ASYNC	send a message asynchronously to an object, the message queue of the actor owning the receiving object is used
YIELD	yields control flow, possibly to another coroutine

Table 3: Additional instructions for non-shared memory concurrency

3.3   Choosing a Research Platform

From our experiments, we conclude four requirements for a full-grown experimental environment fit to demonstrate the advantages of an instruction set supporting a wide range of concurrency models:

- The VM has to be portable to platforms like TILE64 [44] or Cell BE [19] to be able to evaluate the benefits in mapping from an extended instruction set to different concrete concurrency models.

- Implementations of considered abstract concurrency models which use a compilation to the VM instruction set as implementation strategy should be available.

- The VM instruction set should provide space (i. e., unused bytecode instructions) for experiments.

- The VM should provide an easy to adapt JIT compiler to experiment with optimizations.

Based on these requirements, we compiled a list of more than fifty VMs comparing mainly open source implementations for various languages like Erlang, JavaScript, Python, and Scheme. Here we present only a small subset of this comparison to discuss the reasoning for choosing our research platform.

Tab. 4 lists for each VM the characteristics of interest to choose our research platform. The column *language* contains the target language implemented by the VM, *ACM* reflects the abstract concurrency model. The availability of threads, STM, and actors implementations are represented by *IS* for instruction set support, *Lib* for available libraries or language implementations, or *"-"* if implementations are not available but described in literature. Furthermore, we consider whether a JIT compiler is available and a port to a many-core system would be feasible. PyPy's thread support is marked with a "-", since it relies on a global interpreter lock and thus does not allow true parallelism. This DisVM was included even so we only have access to its specification.

Name	Language	ACM	Threads	STM	Actors	JIT	Port	Size (SLoC)	Impl. Lang.
SOM++	Smalltalk	T/L[4]	IS		IS		x	6k	C++
Lua	Lua		Lib		Lib		x	13k	C
LuaJIT[5]	Lua		Lib		Lib	x	x	20k	C
RVM[6]	Smalltalk	T/L	Lib	-	-		x	28k	C++
CacaoVM[7]	Java	T/L	Lib	Lib	Lib	x	x	121k	C++
Mozart[8]	Oz	Data-flow						159k	C++
Erlang	Erlang	Actors			IS	x	x	247k	C
PyPy	Python	T/L	-	Lib	Lib	x		318k	RPython
Maxine[9]	Java	T/L	IS	Lib	Lib	x		361k	Java
HotSpot	Java	T/L	IS	Lib	Lib	x		540k	C++
JikesRVM	Java	T/L	IS	Lib	Lib	x		978k	Java
DisVM[10]	Limbo	CSP[11]						spec.	

Table 4: Overview of potential research platforms

Starting with SOM++, we have to conclude from our experience, that its idealized architecture and its

---

[4]T/L: threads and locks

[5]http://luajit.org/

[6]A Squeak VM developed at IBM Research [40] for the TILE64

[7]http://www.cacaovm.org/

[8]http://www.mozart-oz.org

[9]http://research.sun.com/projects/maxine/

[10]http://doc.cat-v.org/inferno/4th_edition/dis_VM_specification

[11]CSP: Communicating sequential processes [16]

simple implementation allows for fast prototyping of ideas, but on the other hand might conceal problems associated with our approach especially with regard to performance.

Lua is also small, but has been implemented with a clearer performance objective. Furthermore, an implementation with a JIT compiler exists which is small enough to be ported to a many-core architecture without requiring overly large effort. Thus, we will consider it as a vehicle to validate our research in the context of embedded VMs.

The RVM is already tailored to the TILE64 processor. Since it utilizes the many-core architecture, its special inter-core communication facilities, and has a moderate complexity, we will use it for our first experiments, applying our idea in the setting of many-core systems.

CacaoVM seems to be the smallest and most widely ported open source JVM with a JIT compiler. Compared to other JVMs in the table, a port of the CacaoVM to a many-core system should be more feasible, especially since it already has been ported to the Cell BE [37].

However, it might become necessary to consider VMs like HotSpot, JikesRVM and Maxine when it comes to the validation of performance properties. At the moment, it is still not clear whether we will need a JIT compiler with production-level performance to rule out performance characteristics not introduced by our approach but other modifications done in the development.

Erlang, Mozart, and the DisVM have been included for consideration since they implement other abstract concurrency models than the usual shared-memory model with threads and locks. Interpreted Erlang got already official support for the TILE64 and will allow to conduct partial experiments. However, due to its nature of a VM for a functional language and the complexity of its JIT compiler, we will not chose it as our main research platform. Mozart implements an abstract concurrency model based on data-flow variables. Due to its complexity and focus on distributed environments it does not seem to be a feasible platform for our research. The DisVM is an interesting design of a VM where the abstract concurrency model is inspired by CSP. Unfortunately we do not have access to the implementation and thus, an evaluation as a research platform was not possible.

4   Related Work

Support for concurrency in VM instruction sets is currently limited. The Erlang VM's BEAM instruction set[12] is a notable exception, providing dedicated support for its efficient light-weight process implementation. It includes instructions for asynchronous message sends, reading from the process' mailbox, waiting and timeouts. It is an example of how one particular model can be supported at the core of the VM. Another example is the DisVM. It provides instructions to create channels between non-shared memory threads as well as to receive and send messages synchronously. Still, we argue that this concurrency support is not sufficient, since each VM only provides support for a single abstract concurrency model. By contrast, today's VMs have to support many different programming models to justify the investments in sophisticated and efficient JIT compilers and garbage collectors. Thus, they have to provide the basic means for a wide range of concurrency models in the same way as they act as execution platforms for different languages.

In the broader field of instruction set design, there are ongoing efforts to extend the capability of the JVM to act as a platform for different programming languages by introducing the INVOKEDYNAMIC instruction[13]. More general work on improving instruction sets with semantic extensions [21, 30] has

---

[12]http://erlangdotnet.net/2007/09/inside-beam-erlang-virtual-machine.html
[13]http://jcp.org/en/jsr/detail?id=292

been done for the hardware level, but the concepts for, e. g., compiler adaption can be applied to VMs as well.

For the Cell BE, VM applicability has been evaluated. Besides porting and designing JVMs for this platform [29,37], some optimizations have been considered to utilize available computation power [5,45].

Distributing a VM over several computational elements bears additional challenges. Some of them have been addressed for VMs distributed on cluster setups; e. g., class loading, strategies for distributed method invocation, data access on the VM level [48], or thread migration [46].

5   Summary and Future Work

We proposed to decouple abstract and concrete concurrency models to be able to cope with the variability of upcoming many-core architectures and their different memory access architectures. We argue that this step is necessary to be able to provide support for several kinds of languages and their abstract concurrency models on top of a VM. Furthermore, the benefits of a semantically rich concurrency abstraction layer will allow more efficient VM implementations on the various different hardware platforms.

The goal of our ongoing research is to design a comprehensive methodology to design VM instruction sets combining several concurrency models to provide this abstraction. The methodology will address the various different design tradeoffs. Our preliminary prototype enabled us to refine our initial requirements for an experimental environment and provided us with the necessary insights to be able to proceed with our research on a suitable platform.

The next step of our work is to investigate the design principles for intermediate languages and the state of the art in concurrency support. Preliminary results on this work have been presented at the workshop on *Virtual Machines and Intermediate Languages* 2009 [26].

With the insights of design tradeoffs for the languages, i. e., the instruction sets themselves, we plan to investigate which low-level primitives for shared memory concurrency should be included. Later, the integration with non-shared memory models in the same language will be tackled and thus, we will do the step to real multi-model concurrency support for VMs.

References

[1] Gul Agha (1986): *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press.

[2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall & Yuli Zhou (1995): *Cilk: An Efficient Multithreaded Runtime System*. *SIGPLAN Not.* 30(8), pp. 207–216.

[3] Colin Blundell, E Christopher Lewis & Milo M. K. Martin (2006): *Unrestricted Transactional Memory: Supporting I/O and System Calls within Transactions*. Technical Report TR-CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA. Available at `http://www.seas.upenn.edu/~milom/papers/tr06_unrestricted_transactions.pdf`.

[4] Kevin Casey, M. Anton Ertl & David Gregg (2007): *Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters*. *ACM Trans. Program. Lang. Syst.* 29(6), p. 37.

[5] Chen-Yong Cher & Michael Gschwind (2008): *Cell GC: Using the Cell Synergistic Processor as a Garbage Collection Coprocessor*. In: *Proc. VEE '08*. ACM, New York, NY, USA, pp. 141–150.

[6] Tom Van Cutsem, Stijn Mostinckx & Wolfgang De Meuter (2007): *Linguistic Symbiosis between Actors and Threads*. In: *Proc. ICDL '07*. ACM, New York, NY, USA, pp. 222–248.

[7] John S. Danaher, I.-Ting Angelina Lee & Charles E. Leiserson (2006): *Programming with exceptions in JCilk*. *Sci. Comput. Program.* 63(2), pp. 147–171.

[8] M. Anton Ertl & Andreas Krall (1992): *Instruction Scheduling for Complex Pipelines*, LNCS 641, pp. 207–218. Springer.

[9] freescale semiconductor (2005): *Programming Environments Manual for 32-Bit Implementations of the PowerPCTM Architecture*. Specification MPCFPE32B. Available at `http://www.freescale.com/files/product/doc/MPCFPE32B.pdf`.

[10] Yaoqing Gao & Chung Kwong Yuen (1993): *A survey of implementations of concurrent, parallel and distributed Smalltalk*. SIGPLAN Not. 28(9), pp. 29–35.

[11] James Gosling, Bill Joy, Guy Steele & Gilad Bracha (2005): *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition.

[12] Jim Gray & Andreas Reuter (1992): *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[13] Philipp Haller & Martin Odersky (2007): *Actors That Unify Threads and Events*. Coordination Models and Languages .

[14] Maurice Herlihy & Nir Shavit (2008): *The Art of Multiprocessor Programming*. Morgan Kaufmann.

[15] Carl Hewitt, Peter Bishop & Richard Steiger (1973): *A Universal Modular ACTOR Formalism for Artificial Intelligence*. In: *IJCAI'73: Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 235–245.

[16] Charles Antony Richard Hoare (1978): *Communicating Sequential Processes*. Commun. ACM 21(8), pp. 666–677.

[17] Intel Corporation (2009): *Intel 64 and IA-32 Architectures Software Developer's Manual*. Specification. `http://www.intel.com/products/processor/manuals/index.htm`.

[18] ECMA International (2006): *Standard ECMA-334 - C# Language Specification*. 4 edition.

[19] Charles R. Johns & Daniel A. Brokenshire (2007): *Introduction to the Cell Broadband Engine Architecture*. IBM Journal of Research and Development 51(5), pp. 503–519.

[20] Rajesh K. Karmani, Amin Shali & Gul Agha (2009): *Actor Frameworks for the JVM Platform: A Comparative Analysis*. In: *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. ACM, New York, NY, USA, pp. 11–20.

[21] Uwe Kastens, Dinh Khoi Le, Adrian Slowik & Michael Thies (2004): *Feedback Driven Instruction-Set Extension*. SIGPLAN Not. 39(7), pp. 126–135.

[22] Thomas Kistler & Michael Franz (1999): *A Tree-Based Alternative to Java Byte-Codes*. International Journal of Parallel Programming 27(1), pp. 21–33.

[23] Doug Lea (2000): *A Java Fork/Join Framework*. In: *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*. ACM, New York, NY, USA, pp. 36–43.

[24] Edward A. Lee (2006): *The Problem with Threads*. Computer 39(5), pp. 33–42.

[25] Virendra J. Marathe, William N. Scherer & Michael L. Scott (2004): *Design Tradeoffs in Modern Software Transactional Memory Systems*. In: *Proc. LCR '04*. ACM, New York, NY, USA, pp. 1–7.

[26] Stefan Marr, Michael Haupt & Theo D'Hondt (2009): *Intermediate Language Design of High-level Language Virtual Machines: Towards Comprehensive Concurrency Support*. In: *Proceedings of the 3rd Workshop on Virtual Machines and Intermediate Languages*. ACM. (extended abstract).

[27] Michael R. Marty (2008): *Cache Coherence Techniques for Multicore Processors*. Ph.D. thesis, Madison, WI, USA. Adviser-Hill, Mark D.

[28] Mark S. Miller, E. Dean Tribble & Jonathan Shapiro (2005): *Concurrency Among Strangers: Programming in E as Plan Coordination*. In: R. De Nicola & D. Sangiorgi, editors: *Symposium on Trustworthy Global Computing, Lecture Notes in Computer Science* 3705. Springer, pp. 195–229.

[29] Andreas Gal Albert Noll & Michael Franz (2008): *CellVM: A Homogeneous Virtual Machine Runtime System for a Heterogeneous Single-Chip Multiprocessor*. In: *Workshop on Cell Systems and Applications*. Beijing, China.

[30] A. Peymandoust, L. Pozzi, P. Ienne & G. De Micheli (2003): *Automatic instruction set extension and utiliza-*

tion for embedded processors. In: *Proc. ASAP '03*. pp. 108–118.

[31] José M. Piquer (1996): *Indirect Distributed Garbage Collection: Handling Object Migration*. ACM Transactions on Programming Languages and Systems (TOPLAS) 18(5), pp. 615–647.

[32] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh & Benjamin Hertzberg (2006): *McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime*. In: *Proc. PPoPP '06*. ACM.

[33] Vijay A. Saraswat, Radha Jagadeesan, Maged Michael & Christoph von Praun (2007): *A Theory of Memory Models*. In: *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, New York, NY, USA, pp. 161–172.

[34] Hans Schippers, Tom Van Cutsem, Stefan Marr, Michael Haupt & Robert Hirschfeld (2009): *Towards an Actor-based Concurrent Machine Model*. In: *Proceedings of the fourth workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*. ACM.

[35] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan & Pat Hanrahan (2008): *Larrabee: A Many-Core x86 Architecture for Visual Computing*. ACM Trans. Graph. 27(3), pp. 1–15.

[36] Nir Shavit & Dan Touitou (1995): *Software Transactional Memory*. In: *Proc. PODC '95*. ACM.

[37] Georg Sorst (2007): *Java on Cell B.E.* Diploma thesis, Fachhochschule Aachen. Available at `http://vergiss-blackjack.de/diploma-thesis_georg-sorst_java-on-cell.pdf`.

[38] Sriram Srinivasan & Alan Mycroft (2008): *Kilim: Isolation-Typed Actors for Java*. In: *Proc. ECOOP 2008*.

[39] John A. Trono & William E. Taylor (2000): *Further comments on "A Correct and Unrestrictive Implementation of General Semaphores"*. SIGOPS Oper. Syst. Rev. 34(3), pp. 5–10.

[40] David Ungar & Sam Adams (2009): *Hosting an Object Heap on Manycore Hardware: An Exploration*. In: *DLS 2009*.

[41] Robert Virding, Claes Wikstrom & Mike Williams (1996): *Concurrent Programming in Erlang*. Prentice Hall PTR, 2 edition.

[42] Haris Volos, Neelam Goyal & Michael M. Swift (2008): *Pathological Interaction of Locks with Transactional Memory*. Technical Report CS-TR-2008-1631, University of Wisconsin–Madison. Available at `http://www.cs.wisc.edu/multifacet/papers/transact08_txlock.pdf`.

[43] Xu Wang, Ge Gan, Joseph Manzano, Dongrui Fan & Shuxu Guo (2008): *A Quantitative Study of the On-Chip Network and Memory Hierarchy Design for Many-Core Processor*. In: *Proc. ICPADS '08*. IEEE Computer Society.

[44] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III & Anant Agarwal (2007): *On-Chip Interconnection Architecture of the Tile Processor*. IEEE Micro 27(5), pp. 15–31.

[45] Kevin Williams, Albert Noll, Andreas Gal & David Gregg (2008): *Optimization Strategies for a Java Virtual Machine Interpreter on the Cell Broadband Engine*. In: *Proc. CF '08*. ACM, pp. 189–198.

[46] Wenzhang Zhu, Cho-Li Wang & Francis C. M. Lau (2002): *JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support*. cluster 00, p. 381.

[47] Lukasz Ziarek, Adam Welc, Ali-Reza Adl-Tabatabai, Vijay Menon, Tatiana Shpeisman & Suresh Jagannathan (2008): *A Uniform Transactional Execution Environment for Java*. Proc. ECOOP 2008 , pp. 129–154.

[48] John Zigman & Ramesh Sankaranarayana (2003): *Designing a distributed JVM on a cluster*. In: *Proceedings of the 17th European Simulation Multiconference*. Nottingham, United Kingdom.