# Intermediate Language Design of High-level Language Virtual Machines

## Towards Comprehensive Concurrency Support

Stefan Marr
Software Languages Lab
Vrije Universiteit Brussel, Belgium
stefan.marr@vub.ac.be

Michael Haupt
Hasso Plattner Institute
University of Potsdam, Germany
michael.haupt@hpi.uni-potsdam.de

Theo D'Hondt
Software Languages Lab
Vrije Universiteit Brussel, Belgium
tjdhondt@vub.ac.be

***Keywords*** Intermediate Language, Instruction Set, Bytecode, Design, Concurrency, Virtual Machines, Survey

## 1. Introduction

Today's major high-level language virtual machines (VMs) are becoming successful in being multi-language execution platforms, hosting a wide range of languages. With the transition from few-core to many-core processors, we argue that VMs will also have to abstract from concrete concurrency models at the hardware level, to be able to support a wide range of abstract concurrency models on a language level. To overcome the lack of sufficient abstractions for concurrency concepts in VMs, we proposed earlier to extend VM intermediate languages by special concurrency constructs [Marr et al. 2009].

As a first step towards this goal, we try to fill a gap in the current literature and survey the intermediate language design of VMs. Our goal is to identify currently used techniques and principles as well as to gain an overview over the available concurrency related features in intermediate languages. Another aspect of interest is the influence of the particular target language, for which the VM is originally intended, on the intermediate language.

## 2. Related Work on Intermediate Languages

To our knowledge, surveys on instruction set design have been conducted only in the field of hardware instruction set architectures (ISA) [Hennessy and Patterson 2007]. With respect to VMs, currently there is no survey available discussing intermediate language design.

However, several aspects thereof are discussed separately. The question whether an AST-based intermediate format has benefits over a bytecode format has been been discussed by [Kistler and Franz 1999]. Another important design decision is the machine model of the VM [Shi et al. 2008]. The optimization potential of bytecode sets with regard to branch prediction in modern CPUs has been discussed by [Proebsting 1995] and [Casey et al. 2007].

With respect to multi-language VMs, the Java Virtual Machine (JVM) [Lindholm and Yellin 1999] gained a lot attention over the last decade, but was originally designed to execute Java code only. Thus, beside the Common Language Infrastructure (CLI) [ECMA International 2006], until now, there is no major platform which was specifically designed as a multi-language execution platform. However, general comparison of the JVM and the CLI by [Gough 2001] as well as [Shiel and Bayley 2005] point out that both platforms share the basic ideas.

## 3. Survey

We surveyed 17 VMs with sufficiently detailed specifications or open source implementations. On the one hand, we covered different kinds of languages. On the other, we also include alternative implementations for the same language to gain a feeling for the influence of the target language on the VM intermediate language.

Instead of the full survey, here we give only the used criteria, which are partially inspired by the survey on hardware ISAs by [Hennessy and Patterson 2007].

***Specification or Implementation*** Only a minority of the surveyed VMs is based on a *specification*, thus it is important to draw a distinction, as the analyses based on actual *implementations* usually also regard technical subtleties.

***Abstraction Level*** The intermediate layer of a VM commonly uses either a representation which is strongly related to its target language in terms of an *abstract syntax tree* (AST), or it relies on a representation which is design with the execution machinery in mind, i. e., a *bytecode set* resembling hardware ISAs.

***Machine Model*** VMs are usually modeled similar to hardware machines. We distinguish between *stack*, *register*, *register-memory*, and *memory-to-memory* machines [Hennessy and Patterson 2007]. As the *number of registers* of a machine model, we refer to general purpose registers only.

***Representation*** For bytecode sets, we look into instruction encoding in terms of *instruction width*, whether it is a *fixed width* or *variable width* per instruction. Furthermore, the *instruction size*, *number of instructions*, and the used style to encode the instructions are of interest. Other designs are characterized by their node-types of the AST-, graph-, or S- expression-based representation.

***Instruction Categories*** Most of the common operations in the intermediate representations can be classified into one of the following categories: *Load/Store/Stack*, *Compare*, *Control Flow & Exception*, *Arithmetic and Logic*, *Creational*, and *Conversion* operations. All deviations from these categories will be reported explicitly.

***Execution Techniques*** As approaches to executing intermediate representations, we identify *switch* and *threaded* [Bell 1973] interpretation, and *dynamic* (just-in-time) as well as *static* compilation.

***Optimizations*** Regarding the intermediate representation, we are also interested in relevant design decisions achieving *optimizations*.



*VMIL'09,* October 25, 2009, Orlando, FL.

## 4. Concurrency Support

One of the conclusions we can draw from our survey is that explicit concurrency support in the intermediate language is fairly rare in today's VMs. In most VMs, even the support for native concurrency is still limited. Thus, they are not able to benefit from the concurrency provided by the underlying system. This is due to the fact that many VMs still use a global interpreter lock and provide only green threads or coroutines as means of concurrency.

In the following section, we briefly discuss the concurrency support found in the different intermediate languages.

***CLI*** Actual concurrency support in the CLI's bytecode set is limited to the `volatile` instruction causing all changes to subsequent pointer references to be made visible in memory without caching. Furthermore, the CLI relies on a flag in methods to indicate that a lock has to be acquired before the method can be executed. Locking is implemented in the standard library together with a set of more complex synchronization mechanisms and, e. g., parallel loops.

***Dis VM*** The concurrency mechanisms of the Dis VM[1] have been inspired by CSP [Hoare 1978]. As a result, the bytecode set includes instructions to spawn new threads, create typed channels, and to send and receive values in a synchronizing way. Instructions to select a ready channel from a list of sending or receiving channels in a blocking or non-blocking way are included, too.

***Erlang*** Erlang's support for the Actor model [Agha 1986] is realized by five instructions. The `send` instruction implements asynchronous message sends to a specified process. Instructions to wait for messages arriving at the incoming message queue exist, including variations with timeouts. Furthermore, removal of a message from the queue is made explicit. In contrast to the Dis VM, Erlang does not include an explicit instruction to spawn new processes.

***JVM*** The JVM specification defines two concurrency-related instructions: `monitorenter` and `monitorexit`. Basic locking functionality is thus included in the bytecode set. Furthermore, similar to the CLI, methods have a flag indicating whether a lock has to be acquired prior to execution. Other than that, the JVM defines the semantics of the memory model in the presence of threads. Thus, it gives additional information about the semantics for *use*, *assign*, *load*, *store*, *lock*, and *unlock* instructions. Java also incorporates various higher-level concurrency support in its standard library and it is planned to add a fork/join framework.

***Mozart*** The main concept in Mozart/Oz[2] to express concurrency are so-called *data flow variables*. However, the bytecode set contains `LOCKTHREAD` to explicitly acquire a lock. Interestingly, no unlock instruction can be produced by the Oz compiler. The `TASKLOCK` instruction supported by the runtime is not part of the official bytecode set, but is classified as an optimization which can be generated by the runtime system. Mozart's distribution mechanisms are realized without introducing additional bytecodes.

## 5. Conclusion and Future Work

Support for concurrency in VMs is still limited, and the few VMs offering it support mechanisms as diverse as the languages.

Mozart, the CLI, and the JVM provide mostly implicit support, using structural hints and memory model definitions. Although the JVM provides lock and unlock instructions, it is the Java standard library that provides functionality relevant to concurrency.

The Dis VM and Erlang demonstrate how concurrency models can be explicitly supported by the intermediate language. Similarly, hardware ISAs provide support for shared-memory concurrency: they offer a number of low-level operations to allow the efficient implementation of high-level constructs like locks or lock-free data structures by, e. g., operating systems.

We believe in multi-language VMs, since the effort to implement an efficient language runtime is tremendous. The challenge here is the anticipation of a wide range of languages and the design of sufficient support for a large set of different concurrency mechanisms. Furthermore, efficiency in terms of runtime performance is an important feature to attract language developers instead of them implementing a dedicated runtime for their language.

Thus, we think that it will be necessary to provide support for low-level as well as high-level concurrency concepts in the intermediate language. Language developers can potentially use low-level concepts like atomic operations on values, *compare and swap*, or *load linked and store conditional* to implement higher level concepts, which might not even be envisioned today, efficiently. On the other hand, high-level operations like message send or even support for transactional memory might allow runtime optimizations by the compiler for a large set of languages using these mechanisms.

We plan to use both insights to investigate how the currently limited concurrency support in VM intermediate languages can be improved. We will explore the opportunities and benefits of introducing low-level as well as high-level constructs into the intermediate language with the goal to provide support for a wide range of different concurrency models on top of it, and thus, advance the notion of multi-language VMs.

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

[2] `http://www.mozart-oz.org`, Mozart 1.4

## Acknowledgments

Stefan Marr is supported by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.

## References

Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

James R. Bell. Threaded code. *Commun. ACM*, 16(6):370–372, 1973. ISSN 0001-0782.

Kevin Casey, M. Anton Ertl, and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *ACM Trans. Program. Lang. Syst.*, 29(6):37, 2007. ISSN 0164-0925.

ECMA International. *Standard ECMA-335 - Common Language Infrastructure (CLI)*. 4 edition, June 2006.

K. John Gough. Stacking them up: a comparison of virtual machines. *Aust. Comput. Sci. Commun.*, 23(4):55–61, 2001.

John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2007.

Charles Antony Richard Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. ISSN 0001-0782.

Thomas Kistler and Michael Franz. A tree-based alternative to java bytecodes. *IJPP*, 27(1):21–33, 1999.

Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Longman, Amsterdam, 2 edition, April 1999.

Stefan Marr, Michael Haupt, Stijn Timbermont, Bram Adams, Theo D'Hondt, Pascal Costanza, and Wolfgang De Meuter. Virtual machine support for many-core architectures: Decoupling abstract from concrete concurrency models. In *Proc. PLACES'09*, EPTCS, York, UK, March 2009.

Todd A. Proebsting. Optimizing an ansi c interpreter with superoperators. In *Proc. POPL '95*, pages 322–332, New York, NY, USA, 1995. ACM.

Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim.*, 4 (4):1–36, 2008. ISSN 1544-3566.

Sam Shiel and Ian Bayley. A translation-facilitated comparison between the common language runtime and the java virtual machine. *ENTCS*, 141(1):35–52, 2005. ISSN 1571-0661. Proc. Bytecode'05.