Which Problems Does a Multi-Language Virtual Machine Need to Solve in the Multicore/Manycore Era?

Stefan Marr,[1] Mattias De Wael
Software Languages Lab
Vrije Universiteit Brussel, Belgium
{stefan.marr, madewael}@vub.ac.be

Michael Haupt
Oracle Labs
Potsdam, Germany
michael.haupt@oracle.com

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

Abstract

While parallel programming for very regular problems has been used in the scientific community by non-computer-scientists successfully for a few decades now, concurrent programming and solving irregular problems remains hard. Furthermore, we shift from few expert system programmers mastering concurrency for a constrained set of problems to mainstream application developers being required to master concurrency for a wide variety of problems.

Consequently, high-level language virtual machine (VM) research faces interesting questions. What are processor design changes that have an impact on the abstractions provided by VMs to provide platform independence? How can application programmers' diverse needs be facilitated to solve concurrent programming problems?

We argue that VMs will need to be ready for a wide range of different concurrency models that allow solving concurrency problems with appropriate abstractions. Furthermore, they need to abstract from heterogeneous processor architectures, varying performance characteristics, need to account for memory access cost and inter-core communication mechanisms but should only expose the minimal useful set of notions like locality, explicit communication, and adaptable scheduling to maintain their abstracting nature.

Eventually, language designers need to be enabled to guarantee properties like encapsulation, scheduling guarantees, and immutability also when an interaction between different problem-specific concurrency abstractions is required.

1. Virtual Machines in the Manycore Era

High-level language virtual machines use highly optimized just-in-time compilers and garbage collectors to provide performance characteristics comparable to classic low-level system programming languages. Recent improvements and additions to VMs like the *Java Virtual Machine* or *Common Language Runtime* enable efficient execution of a wide range of dynamic languages. These dynamic capabilities are increasingly used to build domain-specific languages (DSLs) on top of the VMs. In turn, DSLs enable developers to tackle their problems at an even higher level of abstraction while utilizing the ecosystem around the VMs.

However, VMs have not made the step into the manycore era by supporting language designers in their utilization of concurrency and parallelism. While VMs provide low-level abstractions like threads and often come with a set of libraries for concurrent data structures or commonly used synchronization concepts, they typically do not provide mechanisms supporting more than one specific concurrent programming model out of the box [15].

We argue that such support is necessary, since the various proposed concurrent programming models all have specific applications where they shine, and use-cases which are not as well supported. The wide range of available concurrent programming models includes the following:

- *Shared memory with threads and locks* is today's standard.
- *Transactional memory* promises to handle some of the engineering challenges arising in concurrent situations.
- *Actors* and *message passing systems* employing them avoid typical low-level concurrency issues.
- *Data-flow programming* is a good fit for a number of computational problems.

Choosing a single model from this incomplete list is inappropriate for a multi-language VM. Instead of a non-existing *silver bullet* approach, we, like other researchers [4, 5], argue that language designers should be supported in building concurrency DSLs on top of VMs, to enable developers to solve their concurrency problems with appropriate abstractions.

VMs are meant to provide an abstraction layer between the actual hardware including a specific operating system, and one or multiple high-level languages. Thus, they provide portability and platform independence from any underlying system to applications that solely rely on the standardized functionality of a VM. In addition to that, VMs typically provide services like adaptive optimization and sophisticated garbage collection to leverage the underlying hardware optimally. These services require enormous amounts of engineering effort, and can thus not be replicated and adapted for each new language. This especially holds for DSLs, which make the reuse of VMs as general execution platforms for different languages and applications very attractive from an economical point of view.

A VM aiming to support DSLs in a manycore world consequently needs to provide basic abstractions which enable the implementation of a wide range of different concurrency models on top of it. Such support is necessary since implementing unsupported models on top of a VM usually brings about significant additional complexity as well as performance disadvantages.

Furthermore, the language specific implementation of a model atop a VM is typically not reusable for other languages. Examples are actor languages for the JVM, which usually compromise on

the actor properties they provide [14]. Research in software transactional memory (STM) suggests that low-level support in runtime systems is a prerequisite for acceptable performance [1]. While performance and implementation effort are important, there is a more critical requirement: The language semantics must be enforced. Otherwise, the languages' benefits will be lost, and reasoning about correctness becomes infeasible when interaction with other languages needs to be considered.

Since VMs mediate between concrete hardware designs and a set of high-level language concepts, both sides influence VM design. Based on this observation, we identify two questions that need to be answered to design a VM that enables support for multiple concurrency models on today's and tomorrow's hardware platforms:

1. Which are the predominating characteristics of multicore and manycore processors that a VM needs to take into account to enable applications to deliver optimal performance?
2. Which problems do traditional implementations for concurrency models encounter that could be avoided by explicit VM support?

We will address these questions individually in the following sections. After that, we outline potential approaches to solving the identified problems.

2. Predominating Processor Characteristics

With the end of steadily increasing clock-frequencies, processor designers needed to identify new processor designs to improve performance with still steadily increasing transistor budgets. At the moment, the number of feasible but different processor designs that leverage the additional transistors seems to be larger than it was for single-core processors [12].

2.1 Brawny vs. Wimpy Cores

A single core's complexity is one design dimension, ranging from *brawny* cores to *wimpy* cores [17]. The distinction is made based on the included techniques to improve the performance of sequential code. Brawny cores include the full set of state-of-the-art technology to obtain the best sequential performance from a given program. However, technologies like out-of-order execution, register renaming, and sophisticated branch prediction consume a valuable share of the transistor budget. The trade-off between power consumption and sequential performance however depends on the individual program. Thus, wimpy cores invest the available transistor budget to replicate whole cores or arithmetic units instead of investing in techniques to improve sequential performance. Replication provides a better energy/performance ratio for parallelizable problems.

Examples for brawny cores are Intel's i7 and IBM's POWER7. Compared to those, Intel's Atom, typical ARM-based, or Tilera's TILE architecture processors classify as wimpy. Even more wimpy and special purpose are the hundreds of computational units on a typical graphics processing unit.

As argued by Hill and Marty [12] and as we already see with the Cell B.E. processor and AMD's Bulldozer/Fusion[1] architecture, hybrid architectures are able to combine the best of both worlds. This might be a good tradeoff when the typical application load has a mix of different characteristics.

From the point of view of VM design, these different design options do not have to be exposed to the user. On the contrary, a VM should abstract from these details and provide a unified and portable interface. Examples like the Hera-JVM [16] show that it is possible to target such hybrid architectures like the Cell B.E. without having to expose it through the VM ISA. Similarly, the goal of standardization efforts like OpenCL[2] is also to provide a unified set of interfaces for the different platforms.

2.2 Many-Core vs. Many-Thread

Another option in processor design is the ratio of physical to logical threads supported on a core [10]. This design dimension relates to the question which parts of a processor can be replicated to get better performance for a certain work load.

Replicating an entire core leads to manycore processors. Replicating only register files leads to many-thread processors like with Intel's hyperthreads technology or Sun's CoolThreads in their Niagara processors. This is also called simultaneous multithreading (SMT). The idea behind SMT is to increase the utilization of the computational units on a processor by interleaving different threads on the same cores when one of them stalls on a memory access. This is especially interesting for superscalar processors able to execute more than one instruction during a clock cycle.

AMD takes a middle course with the Bulldozer architecture, replicating, in addition to the register file, also parts of the arithmetic units. In this case, only integer units are replicated while floating-point units are shared. This provides additional computational resources and can improve performance of certain work loads.

The impact on memory and cache utilization patterns and scheduling opportunities requires the VM to be aware of such hardware details to draw the right conclusions. However, most of these details can be tuned independently from an application inside a VM implementation. For an application, it is important to have a useful and transparent notion of *scheduling units*, i. e., parallel activities or tasks. For some purposes it is even desirable to allow an application to adapt *scheduling* policies with the goal of scheduling related threads/activities together if they share data or benefit from SMT. Furthermore, if activities are known to thrash each others' caches or require higher memory bandwidth, they should be scheduled on independent execution units.

2.3 Memory and Communication

A third design dimension regards memory interconnects and inter-core communication mechanisms.

The *memory wall* is a well-known phenomenon that engineers compensated for with deep cache hierarchies. However, for manycore processors this is not enough. While the number of cores on a chip can be increased easily, scaling up the off-chip memory bandwidth underlies restrictions. Thus, the memory bottleneck worsens with more cores.

Now, processor designers experiment with different designs for inter-core communication and core-local memory or caches to avoid the bottleneck. The interconnection mechanisms range from bus-based architectures that communicate via memory over hierarchical connect busses, to 2D-mesh interconnects with explicit inter-core communication.

Furthermore, designers experiment with different solutions for core-local memory. The Cell's *Synergistic Processing Elements* have local directly addressable memory. However, they can access main memory only asynchronously. Intel's Single-chip Cloud Computer (SCC) design is similar. It also provides a directly addressable local memory for every core. In contrast to common x86 architectures however, the main memory access in the SCC is not made coherent by any hardware mechanism and thus, requires synchronization in software.

[1] `http://www.anandtech.com/show/3863/amd-discloses-bobcat-bulldozer-architectures-at-hot-chips-2010/4`

[2] `http://www.khronos.org/opencl/`

The TILEPro64 has a two-level cache on each core, which is not directly addressable. Furthermore, memory and caches are kept coherent with hardware support. This design provides a similar programming model as on todays multicore systems. However, it has implications for performance, especially since it is a 64-core system: Implicit inter-core communication relying on cache coherence is inefficient. Instead, the TILE architecture provides explicit means for fast inter-core communication.

Thus, with raising core count, the non-uniformity of memory access will increase, and communication and synchronization costs will become dominating factors for parallel performance.

For VM design, we conclude that locality for caches, memory access, synchronization, and communication are properties that needs to be considered for memory management and scheduling. The question is how programming languages can utilize it, and whether it should be exposed to language designers.

From our perspective, this question is answered by concurrency models that explicitly or implicitly support the notion of locality, namely partitioned global address space (PGAS) [6] and non-shared memory models. Since they make locality information explicit, it can be leveraged directly. Models like the standard Java thread model, which do not exhibit such information, would require some dynamic adaptive optimizations which might not be able to deliver optimal results. Thus, we argue that a VM should expose a concept of locality to the language implementor, which in turn could make it explicit in the language or infer it.

2.4 Summary

From our point of view this results in the following requirements for a multicore/manycore VM. It should ...

- abstract from heterogeneous processor design by transparently supporting possibly different instruction sets.
- account for the differences in sequential performance of the different processing units for its scheduling decisions.
- account for the SMT characteristics in its scheduler.
- be aware of different memory access costs and restrictions, and take alternative communication mechanisms into account to improve latency and/or bandwidth usage.
- enable application developers to optimize scheduling decisions for efficiency where the standard heuristics collide with application characteristics.

3. Common Problems in Concurrency Model Implementations

This section discusses common problems that appear when concurrency models are implemented on top of VMs which do not provide explicit support for properties required by the concurrency model.

Karmani et al. [14] survey actor frameworks on the JVM and identify two properties which we consider as fundamental not only for actor-like concurrency models, but for all models which distinguish some kind of *domain of objects* and restrict the possible ways of inter-domain interaction. These properties are *encapsulation* and scheduling *fairness*.

3.1 State Encapsulation

In object-oriented languages and in languages with mutation in general, special care has to be taken for state that an actor can access and change directly. Any shared state is a violation of the actor model where the only directly mutable state is traditionally the message box used to communicate between actors and the behavior used for processing incoming messages. Thus, in languages with mutable state, this mutable state has to have a certain *ownership* relation to an actor that is used to restrict state access to its owner.

```
object semaphore {
    class SemaphoreActor() extends Actor {
        // ...
        def enter() {
            if (num < MAX) {
                // critical section
                num = num + 1; } } }

    def main(args : Array[String]) : Unit = {
        var gate = new SemaphoreActor()
        gate.start
        gate ! enter
        gate.enter } }
```

Listing 1. This semaphore is broken since Scala's actors do not enforce encapsulation. (Karmani et al. Fig. 2 [14])

See Lst. 1 for Karmani's example illustrating the problem. The used Scala actor implementation does not guarantee encapsulation, which leads to a simultaneous execution of `enter` with a race condition on the `num` variable.

The main reasons for implementing an actor-like model without adhering to its characteristics are usually the additional implementation complexity or performance impact of enforcing such properties on top of an existing VM. Furthermore, even if such guarantees are enforced on the language-level, they are often not enforced in interactions with other languages, which would be a key requirement for systems built from different concurrency DSLs that need to interact.

As already indicated, it is important to note that *state encapsulation* needs to be guaranteed for a variety of models beyond the actor model, too. It is a universal property that applies to all models with notions of restricted inter-domain interaction. Thus, it applies to all non-shared-memory models, most notably CSP (communicating sequential processes [13]), and APGAS (asynchronous PGAS [19]) languages which require that mutation is done locally to an owning place/region. An example for another model are Clojure's *agents* [11], which support unrestricted read access but require that all state update functions are executed by the agent itself in a serialized manner.

3.2 Safe Messaging

Karmani et al. further discuss the issue of messaging. Non-shared memory models require that messaging has pass-by-value semantics. Otherwise, shared state could be introduced by passing normal references to mutable objects.

An example would be similar to Lst. 1, where a normal Java collection is passed as argument to the message. In that case, the collection would be passed by reference, and both actors would have unsynchronized access to it. Again, this problem is universal and applies also to, e. g., CPS.

Traditional solutions enforce a pass-by-value semantics either by serializing the passed object graph, which comes with a considerable performance impact, or they use a type-system like Kilim [22]. However, a type-system comes with additional implementation complexity and does not necessarily provide its guarantees across language boundaries.

Note that messaging is a concept which can be emulated by other means on the application/language implementation-level. However, message passing is a fundamental concept with wide applicability for concurrent programming in shared and non-shared memory models. We believe, its fundamental nature and the hardware support for message passing are good reasons to support it directly in VMs.

```
object fairness {
    class FairActor() extends Actor {
    // ...
    def act() { loop { react {
        case (v : int) => { data = v }
        case (wait)    => {
            // busy-waiting section
            if (data > 0) println(data)
            else self ! wait
        }
        case (start)   => {
            calc ! (add, 4, 5)
            self ! wait
        }   }}}}}
```

Listing 2. Without scheduling guarantees, *busy-waiting* can starve other actors forever. (Karmani et al. Fig. 3 [14])

3.3 Scheduling Guarantees

Another issue, unrelated to state sharing, is the often implied assumption that the used scheduling mechanism provides certain guarantees.

One example discussed by Karmani et al. is the actor model's implication that no actor is starved forever. This property is required to guarantee global progress of a correct program. Without it, it would not be possible to eventually deliver all messages. Typical situations in existing implementations are that the scheduler does not guarantee any fairness. Self-messaging loops, like in the example Karmani et al. give, could starve all other actors from running, which then would prevent any global progress, since the expected message would never arrive.

Depending on the semantics the VM provides, other operations can also be problematic with respect to scheduling guarantees. Examples include computationally expensive operations in absence of preemptive scheduling, and calls to blocking native/primitive functions which hinder the execution of other actors on that thread.

An example of such an implementation approach can be found in Clojure's agents implementation. Here, the developer has to be aware of the implementation strategy, since calling blocking functions or doing long calculations will block other agents from making progress for a perhaps unacceptably long time. Clojure provides an explicit `(send-off)` function which starts a new thread for such calls, which is scheduled preemptively to enable other agents to make progress.

To avoid breaking the guarantees of a concurrency model, a VM should provide a lightweight unit of scheduling to the language developer which can provide such guarantees and has the expected performance characteristics.

3.4 Immutability

Immutability is a guarantee often only provided on the language-level. It is not only relevant to support certain concurrency semantics, but also to enable performance optimizations like replication.

The JVM for instance, provides only a weak notion of immutability. The most severe problem for the semantics of immutable objects are the reflective capabilities of the JVM. Changing a final field is allowed by the JVM specification even though the spec. states restrictions on the visibility of such changes to enable compiler optimizations like inlining.

Another example for the use of immutability is VisualWorks 7. [3] It allows to mark objects as immutable and raises an exception when mutation is attempted. However, the exception handler is free to mutate such an object. This behavior is used to map objects to persistent data storage and enable efficient consistency management.

For a concurrency-aware VM, immutability can be used for performance optimizations, either to avoid copying of shared objects, or to replicate them to improve locality of access. In either case, immutability semantics have to be guaranteed to avoid problems. Again, such a guarantee is problematic across language boundaries and thus needs to be supported on a lower level.

3.5 Model Interaction: Language-Level vs. Implementation-Level

As outlined in the introduction, we think it will be necessary to use different concurrency models in the same application to solve problems with appropriate abstractions.

However, as this section argued so far, interaction across language/concurrency model boundaries can be problematic. Encapsulation properties, scheduler guarantees, and immutability semantics can be specific to a model and should be preserved even across boundaries.

Typical examples are the interaction with existing code and libraries, which have not been adapted to be compatible with a specific model. Another example is the implementation of a new concurrency model, i. e., a specific language.

Imagine building a new language on top of the JVM with support for immutable objects. When that language tries to utilize the rich ecosystem and its libraries, the immutability guaranteed for a specific object should be preserved on the language-level. Ideally, existing libraries are usable without adaptations. However, when a library attempts to mutate an immutable object, one approach would be to raise an exception instead of silently disregarding the language guarantees.

However, at the same time, implementing that particular language with Java must be possible. This implies that it needs to be possible to distinguish between a language-level immutability and implementation-level immutability. Otherwise it becomes problematic to, for instance, build circular immutable data structures.

3.6 Summary

From our point of view, these problems result in the following requirements. VMs should provide ...

- configurable enforcement of encapsulation guarantees between domains of objects.
- message-passing with configurable semantics for arguments to allow optimal implementations (e. g. by transferring object ownership, or using copy-on-write).
- configurable scheduling guarantees separated by domains of objects.
- the fundamental concept of immutability, since it is relevant for cross-domain interaction, but also for various compiler optimizations.
- the explicit notions of an implementation level and reflective operations.

4. Possible Solution Approaches

The literature in the field of supporting heterogenous architectures and handling scheduling issues is already providing solutions for most of the problems discussed here.

As mentioned in Sec. 2.1, the Hera-JVM [16] shows that it is practical to abstract from instruction set and memory model differences. Shelepov et al. [20] present a solution to schedule tasks on heterogenous architectures; and schedulers like SLAW [9] are able to take locality into account. Approaches like Lithe [18] give

[3] VisualWorks 7: `http://www.cincomsmalltalk.com/CincomSmalltalkWiki/VisualWorks+7+White+Paper`

more control over the scheduling to the user and hierarchical place trees are a framework to enable the user to model data locality to facilitate scheduling and data movement [25]. Work done by Ungar and Adams regards caching characteristics and performance implications to make an object-oriented system feasible on architectures like the TILE64 [23].

Literature on supporting multiple concurrency models on top of the same VM is considerably more scarce. Most notable is the *language virtualization* done by Chafi et al [5]. Compared to a VM-based approach like ours, they advocate a language-based solution that avoids potentially unsafe operations in the first place; i. e., they use either type systems or frameworks for language composition. Neither of these seems to be appropriate when supporting a large-scale ecosystem. In our approach, the VM provides these safety mechanisms and is independent of the semantics of any particular tool chain or sub-ecosystem.

One mechanism we are aware of that provides an enforcement for encapsulation guarantees for a specific concurrency model is the language symbiosis of AmbientTalk [24]. It enforces actor semantics when interacting with Java code. To support such properties on the VM-level, solutions like *AppDomains* of the *Common Language Infrastructure* [7], Java's `SecurityManager` and `ClassLoader` could be a start. However, these mechanisms are either heavyweight or too inflexible for the configurability required for different concurrency models and their interactions.

The VMs we are aware of that provide message-passing primitives are the DisVM[4] and Erlang's BEAM [2]. The semantics of messages and how objects are passed need to be considered carefully to enable different semantics and performance optimizations. Extensible mechanisms, as used in distributed systems, are a starting point [8].

One notion of immutable objects is supported by VMs like VisualWorks 7. While Java does not go beyond offering idioms, there has been discussion to introduce support for object life-cycle allowing *larval* mutable objects which can later mature into immutable ones.[5] Another option would be to treat immutable objects as owned by a special domain which does not allow any mutation. Thus, mechanisms to provide different semantics for concurrency models could be general enough to support the notion of generally immutable objects, too.

Explicit support for reflective operations and switching between different semantic layers was proposed with 3-Lisp and its reflective towers [21]. However, to control language-level reflection capabilities, a finer degree of control, as it is possible with Mirrors [3], is desirable.

5. Conclusion

VMs need to change in order to cope with the new requirements of the multicore and manycore era. They need to provide language designers with richer concurrency mechanisms to enable them to provide concurrency DSLs that allow application developers to attack their problems on a higher level of abstraction.

New hardware characteristics require control over scheduling, memory usage, communication, and locality concerns.

On the language level, the support for different concurrency models needs to allow language designs to vary the given guarantees with respect to encapsulation, scheduling, and properties like immutability. Furthermore, these guarantees need to be enforceable in interactions across models, languages, and reflective operations.

This will increase the value of VMs as the center of ecosystems with large libraries of reusable artifacts and tools, and will hopefully lead to natural solutions for the different concurrency problems.

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

[5] `http://blogs.oracle.com/jrose/entry/larval_objects_in_the_vm`

References

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proc. of PLDI '06*, pages 26–37. ACM, 2006.

[2] J. Armstrong. A history of erlang. In *Proc. of HOPL III*, pages 6–1–6–26. ACM, 2007. ISBN 978-1-59593-766-X.

[3] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proc. of OOPSLA '04*, pages 331–344. ACM, 2004.

[4] B. Catanzaro, A. Fox, K. Keutzer, D. Patterson, B.-Y. Su, M. Snir, K. Olukotun, P. Hanrahan, and H. Chafi. Ubiquitous parallel computing from berkeley, illinois, and stanford. *IEEE Micro*, 30(2):41–55, 2010.

[5] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In *Proc. of OOPSLA '10*, pages 835–847. ACM, 2010.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proc. of OOPSLA '05*, pages 519–538. ACM, 2005.

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

[8] S. Gopal, W. Tansey, G. C. Kannan, and E. Tilevich. Dexter — an extensible framework for declarative parameter passing in distributed object systems. In *Proc. of the ACM/IFIP/USENIX 9th International Middleware Conference*, pages 144–163. Springer-Verlag, 2008. ISBN 978-3-540-89855-9.

[9] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. Slaw: A scalable locality-aware adaptive work-stealing scheduler for multi-core systems. *SIGPLAN Not.*, 45:341–342, January 2010.

[10] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. Weiser. Many-core vs. many-thread machines: Stay away from the valley. *IEEE Comp. Arch. Letters*, 99(2), 5555.

[11] S. Halloway. *Programming Clojure*. Pragmatic Programmers. Pragmatic Bookshelf, 1 edition, 2009. ISBN 1934356336.

[12] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41 (7):33–38, 2008.

[13] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8): 666–677, 1978.

[14] R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the jvm platform: A comparative analysis. In *Proc. of PPPJ '09*, pages 11–20. ACM, 2009.

[15] S. Marr, M. Haupt, and T. D'Hondt. Intermediate language design of high-level language virtual machines: Towards comprehensive concurrency support. In *Proc. of the 3rd VMIL Workshop*, pages 3:1–3:2. ACM, October 2009. (abstract).

[16] R. McIlroy and J. Sventek. Hera-jvm: a runtime system for heterogeneous multi-core architectures. In *Proc. of OOPSLA '10*, pages 205–222. ACM, 2010.

[17] T. N. Mudge and U. Hölzle. Challenges and opportunities for extremely energy-efficient processors. *IEEE Micro*, 30(4):20–24, 2010.

[18] H. Pan, B. Hindman, and K. Asanović. Lithe: Enabling efficient composition of parallel libraries. In *Proc. of HotPar '09*. USENIX Association, 2009.

[19] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu. The asynchronous partitioned global address space model. Technical report, June 2010.

[20] D. Shelepov, J. C. S. Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. Hass: A scheduler for heterogeneous multicore systems. *SIGOPS Oper. Syst. Rev.*, 43:66–75, April 2009.

[21] B. C. Smith. Reflection and semantics in lisp. In *Proc. of POPL '84*, pages 23–35. ACM, 1984.

[22] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. In *Proc. of ECOOP '08*, pages 104–128, 2008.

[23] D. Ungar and S. S. Adams. Hosting an object heap on manycore hardware: An exploration. In *Proc. of DLS '09*, pages 99–110. ACM, 2009.

[24] T. Van Cutsem, S. Mostinckx, and W. De Meuter. Linguistic symbiosis between event loop actors and threads. *Computer Languages, Systems & Structures*, 35 (1):80 – 98, 2009.

[25] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *Proc. of the 22nd LCPC Workshop*, volume 5898 of *LNCS*, pages 172–187. Springer, 2009.