NGen-VM: New Generation Execution Environments

João Lourenço[1], Nuno Preguiça[1], Ricardo Dias[1], João Nuno Silva[2], João Garcia[2], Luís Veiga[2]
(1) FCT - New University of Lisbon (2) INESC-ID / Technical University of Lisbon, Portugal
{joao.lourenco,nuno.preguica,rjfd}@di.fct.unl.pt {joao.n.silva, joao.c.garcia, luis.veiga}@inesc-id.pt

ABSTRACT

This document describes a work-in-progress development of NGen-VM, a distributed infrastructure that manages execution environments with run-time and programming language support targeting applications developed in the Java programming language, deployed over clusters of many-core computers. For each running application or suite of related applications, a dedicated single-system image will be provided, regardless of the concurrent threads running on a single machine (on several cores) or scattered on different computers.

Such system images rely on a single model for concurrency management (Transactional Shared Memory Model), in order fill the gap between the hardware infrastructure of clusters of many-core nodes and the application runtime that is independent from that hardware infrastructure.

Interactions between threads in the same tasks will be supported by a Transactional Memory framework that provides the programming language with Atomic and Isolated code regions. Interactions between thread on different machines will also use the Transactional Memory model, but now resorting to a Distributed Shared Memory abstraction.

1. INTRODUCTION

One can estimate that computers will soon include dozens of processors [5] in the same chip (many-core processors) and computing centers will soon provide clusters of such computers with a large number of processors as the standard execution environments. Until recently programmers focused mainly in single-threaded program development. With the widespread infrastructural support already available and the forthcoming new many-core based architectures, software developers must exploit multi-threading as a way to adequately use the multiple cores at disposal. But multi-threading and high processor usage comes at the expense of a dramatic increase in the complexity of the applications, as parallelization brings along a whole new set of problems, such as multiple concurrent control flows, deadlocks, livelocks, priority inversion, convoying, etc.

Software targeting clustering environments frequently rely in the message-passing model (MP) and middlewares for data exchange and control flow synchronization. MP can be used on a wide range of problems since it exploits both task parallelism and data parallelism although exhibiting some dependency on algorithms targeting specifically distributed architectures. The MP model is adequate for data parallelization relying in algorithms based in a Partitioned Global Address Space (PGAS). Being very sound for cluster environments, the MP model is also applicable to shared memory systems, although for these systems a shared memory model (and not MP), relying in algorithms based in a Shared Global Address Space (SGAS), is the common option.

For clusters of many-core computers, none of these models, per si, completely fulfills the application development requirements. Algorithms usually rely in a PGAS or a SGAS, but not both simultaneously. Developing applications mixing algorithms from both classes and using both models, SM for exchanging data within the same node and MP between different nodes, is cumbersome and error prone. A more convenient approach would be independent from the physical architecture and use a single model.

Theoretically, MP could be used to support coordination and data exchange between different components, independently of their location in the same of different nodes. This approach, although acceptable for medium- and coarse-grain parallelism does not apply to fine-grain parallelism, thus nor for may-core computing environments. On the other hand, the shared memory model (SM), which benefits from algorithms developed specifically for the SM model, adapts very well to coarse, medium and fine-grain parallelism, but usually does not support distributed environments efficiently.

A third approach, higher in the abstraction level, resorts to the transactional model (which has been used in the databases world for many years and provides well-known properties and features to deal with concurrency) which is applied to the management of central memory [3, 4] and leaves the contention and management independent from the implementation. Besides being well know by a large number of programmers, the transactional model is independent from the implementation, widening the scope of the runtime support services and their optimizations [2].

All these aspects of the current scenario indicate that interesting solutions can derive from choosing hybrid approaches. This has been attempted before, yet with more limited goals, in JVM-clustering middleware such as Terracotta, designed to allow seamless execution of Java applications on clusters, by offering a SM model where threads manipulate a SGAS, and system-level approaches such as Mosix. Nonetheless, these show a number of shortcomings such as: i) absence of global scheduling and task migration and STM support in Terracotta, and ii) absence of global addressing and need for explicit programmatic inter-process communication in Mosix, and utility-computing infrastructures such as Amazon EC2.

We defend that the three aforementioned models can be blended to achieve a two-fold goal: i) improve developer's productivity by providing an intuitive programming model (combining SM and TM with a SGAS), ii) enhance performance by transparently employing MP-related mechanisms when appropriate and advantageous.

2. ARCHITECTURE

The base architecture of NGen-VM comprises an *Infrastructure*, a variable number of *Single-System Images* [1], and *Applications* running on top of the latter.

Infrastructure. A dynamic aggregation of heterogenous computational resources to create a virtual cluster with each node executing the system-level NGen enabling middleware. Basic services offered by the infrastructure include heterogeneity support, energy-aware scheduling, entry/exit/enrollment protocols, inter-system confinement, security/trust. Other services include data compression, copy-on-write, lazy allocation, delayed zeroing, class and method GC, object swapping, and exploiting byte-code redundancy.

Single-system Image. An allocated subset of the infrastructure based on required resources with unified interface that can be mapped to any specific fraction of the infrastructure at a given time. It offers direct support for large-scale computing providing a single address/reference space within a system image, system-wide scheduling and resource management policies, concurrency control using transactional memory, and parallelization support.

Applications. These run on top of individual system-images where code is deployed and data storaged. Tasks are supported as first-level entities within VMs, may be explicit (such as threads, atomic and isolated blocks) or implicit via speculative execution supported by lower layers.

3. TRANSACTION HANDLING

By relying on the transactional model, NGen-VM aims at a large set of automatic (system and compiler) optimizations: optimistic concurrency control for transactions, speculative execution of fine and medium-grain tasks via futures (with adequate support for side-effects), speculative execution upon committing a transaction, partial data and code replication for increased availability.

We propose to divide transactions according to: i) their scope, local or distributed, and ii) their nature, read-only or read-write. Local transactions will be managed according to the nature of the TM framework to be selected. To reduce the number of aborts, the transactional framework may reschedule the transactions in an order from the real one, as long as the transactional properties are preserved.

Read-only may be rescheduled with more liberal policies, and this may be considered to reduce the number of aborted transactions. The impact of identifying the read-only distributed transactions will be even much higher. The identification of read-only transactions may be user-based, fully automatic, or user-assisted.

We propose to go for the latter and aim to provide the user with a notation for expressing transactions known to be read-only, helping generating optimized code and better exploiting the runtime properties, essentially trusting the user annotations (if they exist) to generate optimized code for the transaction but still validating that the used did not misinform the system about the nature of the transaction. For transactions without annotation, unless known otherwise, we plan to assume the transaction will be read-only and start it as such. On the first write operation we will try to promote the transaction to a read-write transactions and, if we can't, restart the transaction as a read-write transaction.

4. LANGUAGE- AND VM-INTEGRATION

Integration with the Java programming language and the architecture of the Java Virtual Machine is achieved through three different approaches trading program transparency with VM portability, described next.

Library-based. In this approach, creation of distributed memory transactions is done in two ways: created by the user or, dynamically created in runtime by detecting an access to a distributed shared-data. User-defined distributed memory transactions are a simple approach but this puts the burden of deciding the type of transaction to be used in each case, on the user. Dynamic distributed memory transactions can be generated by the runtime when, inside a local transaction, is made an access to distributed shared-data, and then the transaction is aborted and restarted as a distributed memory transaction.

Source-to-source compiler. This approach adds programming language support and allows to do rapid prototyping on extending languages with new constructs (keywords or field attributes) that are evaluated at compile time. Polyglot [6] is an extensible compiler framework for Java which allows to develop source-to-source compilers to extend the Java programming language with new features. The code generated by the Polyglot source-to-source compiler is pure Java code which can be compiled by any Java compliant compiler. With compiler support is possible to add transactions directly into the programming language and the generated code will make calls to the developed runtime library.

Native VM support. By supporting transactional memory with distributed shared memory directly on a JVM, programs could run on top of this modified JVM, and program data could be shared transparently at different nodes and, use the transactional model to safely modify private and shared-data state. This can be achieved by extending Jikes RVM, a JVM for researchers totally implemented in Java, by implementing support of transactional memory with distributed shared memory.

5. SUPPORT FOR LEGACY CODE

Legacy-code support in NGen-VM addresses three scenarios:

I/O in transactional contexts. In most contexts I/O is not revertible (e.g., output to the terminal) and, thus, cannot be freely executed inside transactions. In some other context I/O is theoretically revertible, such as I/O to the filesystem. We intend to adapt already existing prototypes to the new distributed TM framework.

Support for legacy source-code. Existing source-code, e.g., from scientific libraries, must be analyzed for its transactional properties, adapted and transformed to fit into the newly defined transactional model.

Support for legacy binary-code. Even in the cases where the user does not have the source code available, it may be necessary to consider integrating such code in the application. For this, support for legacy binary-code must be introduced in order to not violate the transactional model and assumptions when executing this legacy-code.

6. CONCLUSION

This document presents a new approach to design scalable and efficient distributed execution environments for high-level languages such as Java. It combines system-level management mechanisms (scheduling, migration, energy-awareness, single-system image) for increased scalability and reliability, with programming-level constructs (software transactional memory, speculative execution, byte-code enhancement) for programming correctness and adequacy to emerging multi-core architectures.

7. REFERENCES

[1] R. Buyya, T. Cortes, J. Toni, and H. Jin. Single system image. *Int. J. High Perform. Comput. Appl.*, 15(2):124–135, 2001.

[2] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2):5, 2007.

[3] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93*, USA, 1993. ACM.

[4] M.and Luchangco V. Herlihy, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03*, USA, 2003. ACM.

[5] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the intel 80-core network-on-a-chip terascale processor. In *ACM/IEEE Supercomputing'08*, USA, 2008. IEEE Press.

[6] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *Int'l Conf. on Compiler Construction(ETAPS 2003)*, 2003.

NGen-VM: New Generation Execution Environments

João Lourenço Nuno Preguiça Ricardo Dias
{joao.lourenco, nuno.preguica, rjfd}@di.fct.unl.pt
CITI — Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

João Nuno SIlva João Garcia Luís Veiga
{luis.veiga, joao.n.silva, joao.c.garcia}@inesc-id.pt
INESC-ID — Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

inesc id lisboa
Distributed Systems Group

The problem – *Architectures* 1

Trend for multi- and many-core processors
- Technological limitations
- Computing power per energy unit
- High demand of parallel programming

New computing clusters architecture
- Before: single or dual-processor nodes
- Soon: many-core nodes with dozens of processors

The problem – *Programming models* 2

Parallel programming models
- Support for either SGAS or PGAS
- Support for either fine-grain or coarse-grain parallelism
- Examples: Open-MP and MPI

Message passing
- Is location independent
- Support for control and data parallelism
- Communication costs demand coarser grain
- Low-level concurrency control

Shared memory
- Adequate for fine-grain parallelism
- Is location dependent
- Low-level concurrency control

Transactional memory
- Higher-level concurrency control abstractions
- Implementation independent
- Has place for many compiler optimizations

NGen-VM – *Model* 3

Transactional Shared Global Address Space
- Supported over native shared memory
  in tightly-coupled (many-core) nodes
- Supported over distributed shared memory (DSM)
  in loosely-coupled (cluster) nodes
- *DSM is aware of the transactional model*

NGen-VM – *Architecture* 4

Infrastructure
- A variable number of Single-System Images
- Each node is executing the NGen-VM middleware
- Basic services include:
  heterogeneity support, energy-aware scheduling,
  enrollment protocols, security
- Other services supported:
  data compression, copy-on-write, lazy allocation,
  delayed zeroing, object swapping,
  class and method garbage collection,
  exploiting byte-code redundancy

NGen-VM Single-System Image
- Allocated subset of the infrastructure
- Confines the application execution environment
- Provides a Single Address Global Space for each SSI,
  system wide scheduling,
  system wide resource management,
  transactional memory based concurrency control,

Applications
- Solving user problems

NGEN-VM – *Language and VM Integration* 5

Library based
- For rapid prototyping

Source-to-source compiler
- For performance optimization

Native VM support
- Support for I/O in transactional contexts
- Support for legacy source code
- Support for legacy binaries