THÈSE D'HABILITATION
À DIRIGER LES RECHERCHES

présentée à

L'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

soutenue par

Gaël THOMAS

Le 28 septembre 2012

IMPROVING THE DESIGN AND THE PERFORMANCE OF MANAGED RUNTIME ENVIRONMENTS

JURY

Rapporteurs

Emery	BERGER	Professeur associé, Université du Massachusetts Amherst
Jan	VITEK	Professeur, Université de Purdue
Willy	ZWAENEPOEL	Professeur, École Polytechnique Fédérale de Lausanne

Examinateurs

Albert	COHEN	Directeur de Recherche, INRIA Saclay
Bertil	FOLLIOT	Professeur, UPMC Sorbonne Université
Gilles	MULLER	Directeur de Recherche, INRIA Rocquencourt
Wolfgang	SCHRÖDER-PREIKSCHAT	Professeur, Université Erlangen-Nürnberg

À Valou, Gwennou et Sachou,

REMERCIEMENTS

Mes remerciements vont d'abord à Valérie pour m'avoir laissé jouer à faire de la recherche. Merci pour ta présence joyeuse au quotidien et ta patience lors de nuits infernales à fignoler des articles. Sans ton accompagnement et sans le guilleret charivari de nos enfants, Gwenn et Sacha, le plaisir de la recherche n'aurait pas eu la même saveur.

Ensuite, je remercie Emery Berger, Jan Vitek et Willy Zwaenepoel pour avoir accepté d'être rapporteurs de mon HDR, ainsi que Albert Cohen et Wolfgang Schröder-Preikschat pour avoir accepté d'être examinateurs. Leurs travaux sont parmi les plus intéressants que je connaisse et je suis extrêmement flatté qu'ils m'aient fait l'honneur de participer à mon jury.

Je souhaite aussi remercier chaleureusement mes amis et collègues Gilles Muller et Julia Lawall qui m'ont transmis en quelques années leur fabuleuse expérience. Ils m'ont surtout montré à quel point toujours essayer de s'améliorer est amusant. Je souhaite aussi remercier chaleureusement Bertil Folliot, qui, outre me faire l'honneur de participer à mon jury, m'a donné le goût de la recherche durant ma thèse, et Marc Shapiro pour toutes ses discussions animées sur nos sujets de recherche communs.

Réussir ma recherche n'aurait pas été possible sans le travail des ingénieurs et étudiants en thèse que j'ai ou ai eu le plaisir d'encadrer - et de parfois un peu martyriser -, Nicolas Geoffray, Charles Clément, Thomas Preud'Homme, Jean-Pierre Lozi, Lokesh Gidra, Florian David, Harris Bakiras et Koutheir Attouchi. Un immense merci à vous tous pour m'avoir accompagné ces dernières années, et pour avoir eu la gentillesse de me suivre lors des périodes folles avant les remises d'articles.

Enfin, mener à bien ma recherche a aussi été rendu possible par le cadre de travail formidable dans lequel j'évolue, spécialement grâce à l'équipe Regal. Je remercie tout particulièrement Pierre Sens pour avoir créé ce cadre idyllique pour un jeune chercheur. Je tiens aussi à remercier mes amis et comparses de l'équipe, Julien Sopena, Luciana Arantes, Sébastien Monnet et Olivier Marin, pour tout ce temps passé en discussions échevelées pendant nos pauses cafés.

CONTENTS

1. INTRODUCTION 9
 1.1. Contributions 10

2. GENERAL DESIGN OF MREs 12
 2.1. Components of an MRE 12
 2.2. GC algorithms 13
 2.3. Interactions between components 15

3. MRE DESIGN WITH VMKIT 18

4. SAFETY WITH I-JVM 21

5. EFFICIENCY WITH MREs FOR MULTICORE 23
 5.1. Remote Core Locking 23
 5.2. Numa-aware GC 25

6. CONCLUSION 27

BIBLIOGRAPHY 29

1. INTRODUCTION

Support for application execution has profoundly changed over the last twenty years with the advent of *managed runtime environments* (MREs). An MRE abstracts away the hardware and the operating system by simulating a virtual instruction set and by providing a generic system programming interface. As compared to native execution, MREs brings *portability*, *safety* and *productivity*, for only a slight degradation of *performance*.

Portability. Computers are today highly heterogeneous, ranging from smartphones to desktop and servers with a high variety of operating systems. Hiding this heterogeneity is today essential with the massive distribution of code on the internet, especially during browsing where code is used pervasively to make web pages more appealing and interactive. MREs hide the hardware because they simulate a virtual instruction set. They also hide the operating system because they provide a generic system interface, which they map to the host operating system interface. Hence, MREs bring portability because they execute applications compiled once, but that can run everywhere, regardless of the operating system, hardware platform or instruction set.

Safety. Safety of code is a concern whenever code is downloaded from an untrusted source. Like portability, safety is a major issue in the context of the web. Applications from different and unknown sources are often aggregated in a single page for dynamic rendering, to add advertising or to compute statistics. This aggregation is transparent for the user, who is indeed unaware of the origin of the code. These applications are downloaded and executed on the user host, possibly opening a door for attacks [11]. Users needs therefore safety guarantees before executing these applications.

An MRE enforces safety by acting as a sandbox. The set of restrictions enforced by this sandbox may depend on the degree of trust that the user has. For example, by default, a Java application installed on the machine has more privileges than a Java applet running in a browser. At minimum, an MRE enforces memory safety by checking the memory locations accessed by each operation, thus enabling it to avoid unsafe operations such as buffer overflows or accesses to uninitialized variables. To reduce performance costs, the code can be verified at load time to avoid most of the runtime checks.

Productivity. MREs increase productivity by providing features that either prevent simple bugs or that help debugging. Most MREs provide two mechanisms for this purpose. First, an MRE offers a *garbage collector* [149] (GC), which automatically frees unused memory, thus avoiding most memory bugs. Second, an MRE verifies type safety during execution, which can help the developer to understand the cause of many bugs.

Performance. MREs originally relied on interpreters for code execution, which degraded the performance by an order of magnitude as compared to a native execution. This poor performance discouraged the adoption of MREs. The first improvement of MRE performance came with the introduction of just in time (JIT) compilers to compile hot paths on the fly into native code [25]. Since then, a large

part of the research on MREs has focused on improving the performance of JIT compilers, resulting in adaptive JIT compilers [62, 128], trace based compilers [29, 110], quasi-static compilers [61, 140] etc. Another large part of the research in MRE has been devoted to GCs, resulting for example in generational algorithms [31, 121, 133, 145], concurrent algorithms [67–69, 71, 117], or system optimizations [7, 51, 72, 85]. Other research on MREs has focused on concurrency [38, 56, 87] or language-depend mechanisms [33, 36, 60, 66]. As a result, today, executing code with an MRE gives satisfactory responsiveness for the end user even if performance remains slightly slower than native execution.

MREs are everywhere. MREs are now the norm for executing programs. In February 2012, the Tiobe programming index [166], which estimates the popularity of languages, showed that among the four most frequently mentioned languages on the web, Java (17%), C (16.6%), C# (8.5%) and C++ (7.8%), two are managed, and among the next sixteen, twelve are managed. MREs are used in all environments, ranging from web servers to desktops and embedded systems. In servers, .Net virtual machines, Java virtual machines and PhP engines are standard to execute web servers or application servers such as JSP servers, ASP servers and PHP environments. In desktops, web browsers use MREs for Javascript and Flash. MREs are also used for .Net applications on Windows and for Java applications in all operating systems. Finally, Android smartphones[1] execute applications distributed as a lightweight Java bytecode (DEX) targeting embedded device with the Dalvik virtual machine.

1.1. CONTRIBUTIONS

Over the last six years, my research has focused on improving MREs, targeting (i) the design of MREs with VMKit [182], (ii) their safety with I-JVM [183], and (iii) their performance with proposals to better exploit multicore architectures [177, 191]. This research has been carried out in collaboration with seven PhD students, two of which having already defended.

VMKit. Current MREs are monolithic. Extending them to propose new features or reusing them to execute new languages is difficult. We have proposed VMKit [182] to ease the development of new MREs and the process of experimenting with new mechanisms inside MREs. VMKit is a library that provides the basic components of MREs: a JIT compiler, a GC, and a thread manager. We have used VMKit to develop two MREs: a Java virtual machine and a .Net virtual machine. Our evaluation shows that their performance is equivalent to their monolithic counterparts, and that their language-dependant part represents only 4% of the total code, with the rest provided by VMKit. This evaluation thus shows that VMKit significantly decreases the time to develop MREs for new languages, without sacrificing performance.

I-JVM. OSGi [161] is a component framework widely used to build pervasive applications or plugin engines. OSGi executes all components in the same address space. These components are provided by different tiers that do not trust each others. The Java virtual machine is *not safe* in this context because it is unable to isolate components executed in the same address space and does not propose any mechanism to limit their individual resource consumption. We have proposed I-JVM [183], a new Java virtual machine compatible with the current Java specification that isolates components in sandboxes, but that allows the components to communicate efficiently when they trust each other. Our evaluation of I-JVM shows that it eliminates the 8 known OSGi vulnerabilities that are due to the Java Virtual Machine [143] with a performance degradation below 20%.

[1] At the end of 2011, Android smartphones represented over half (53%) of the smartphones sold in the US (https://www.npd.com/wps/portal/npd/us/news/pressreleases/pr_111213).

MREs for multicore. Most used MREs, such as the Java, .Net, Python or Javascript virtual machines, were defined for machines with only one or few cores and thus, they do not scale on modern multicore hardware [137, 191]. We have identified the lock mechanism and the GC as important bottlenecks.

To reduce the cost of locks, we have proposed a new locking mechanism called remote core locking (RCL) [177] that aims to increase the locality of critical sections and to decrease the time to obtain a lock, by dedicating a core to the execution of critical sections. A first evaluation of RCL on C applications with a 48-core machine has shown that RCL outperforms all other known lock mechanisms. Using the RCL mechanism, we get significant performance improvement on ten applications, including Berkeley DB [15], memcached [174] and applications from the SPLASH-2 [175] and the Phoenix2 [78] benchmarks. We get performance improvements of up to 2.6 times with respect to POSIX locks on Memcached, and up to 14 times with respect to lock implemented in Berkeley DB.

To reduce the cost of the GC, we have studied the performance of the GCs provided by OpenJDK 7 [163] and found that they are unable to scale when the number of cores increases, despite the use of parallel GC algorithms [191]. We have identified the non-uniform memory access (NUMA) architecture as the main problem. Current GCs do not take into account memory locality, which drastically degrades their performance. We are currently investigating how to improve the locality of GCs on NUMA architectures.

Remainder of the document. The remainder of this document is organized as follow. Section 2 presents the general design of MREs, including the main components of MREs and their interactions. Section 3 presents VMKit, Section 4 presents I-JVM and Section 5 presents MREs for multicore hardware. Finally, Section 6 concludes the document and presents future work targeting the design of the next generation of MREs that will have to adapt the application at runtime to the actual multicore hardware on which it is executed.

2. General design of MREs

An MRE is classically designed around three components: an execution engine, a thread manager and a memory manager. The execution engine is intrinsic to an MRE while the remaining components are commonly present. This section first describes the three components. Second, the section summarizes the most important algorithms used by a memory manager for GC. Third, this section shows how each GC algorithm impacts the other components, and how these components interact.

2.1. Components of an MRE

The execution engine. The execution engine executes applications by simulating the virtual instruction set. This engine has a direct impact on performance: the more efficient it is, the more efficient are the executed applications. Historically, the execution engine was implemented as an interpreter, i.e., a loop that decodes and then interprets each operation performed by the application. Interpreters are however inefficient because the interpretation of each instruction requires the reading and the decoding of the instruction in addition to its execution. This poor performance discouraged the use of MREs for mainstream languages.

Today, an efficient MRE provides a JIT compiler to compile at runtime the most used code sequences into native code. A JIT compiler can optimize the code [151, 152] by using fast and language-agnostic algorithms such as sparse conditional-constant propagation [129] or linear-scan register allocation [127], and can provide optimizations that specifically target the semantics of the language, such as array bounds check elimination [32, 113], lock elision [27, 34, 57], invocation dispatch using interface method tables [60] or devirtualization [63]. JIT compiled code is often as efficient as the equivalent native code compiled by an ahead of time (static) compiler. However, in an MRE, the time to JIT compile the code is spent during the execution of the application, thus, only hot paths should be compiled to ensure the amortization of the compilation time.

The thread manager. Many languages provide a concurrency abstraction. Different models are possible, for example the actor-based model [122], but they all rely on threads at the hardware level [39]. A thread is the unit of processing that can be scheduled on a core. Threads are either implemented with native threads, i.e., threads managed and scheduled by the operating system, or with MRE threads, i.e., threads managed and scheduled by the MRE.

Native threads present two advantages. First, only the operating system can exploit multicore hardware and schedule the threads on the different cores. Second, native threads simplify the implementation of the MRE since it does not have to implement a scheduler.

However, if an MRE running on a general-purpose operating system executes applications that expect specific scheduling policy, such as a real-time scheduling algorithm, the MRE has to schedule its threads itself and MRE threads are required. As the MRE is unable to directly schedule threads on the cores of a multicore hardware, the MRE has to create as many native threads as the number of cores, pin them on the cores, and then schedule its MRE threads within these native threads. Such an algorithm

greatly complicates the MRE implementation and is useless if the MRE does not need specific scheduling policies.

The memory manager. A memory manager allocates memory and offers a GC, which finds unused objects or structures, and frees them automatically. A GC helps ensure memory safety by eliminating the need for manual release of objects. The first GCs were proposed at the end of the 1950s for Lisp [136]. The complexity of early algorithms was proportional to the total number of objects, regardless of the amount of live objects, and thus represented a large portion of the total execution time of MREs. Since then, optimized algorithms [24, 31, 121, 133, 145] have been proposed. Their complexity is either proportional to the number of live objects or to a fraction of the number of allocated objects. Today, a GC only slightly degrades performance in practice [54]. As this performance degradation is counterbalanced by the increase of productivity and safety, GCs are now used in many MREs.

2.2. GC ALGORITHMS

While in principle any GC can be used within an MRE, the choice of algorithm has a substantial impact on application performance. Furthermore, we see in the next section that the choice of GC algorithm may also affect the overall design of the MRE itself, as algorithms that give the best performance require specific support from other MRE components. Figure 2.1 presents a taxonomy of GC algorithms. The remainder of this section briefly describes their properties.

<table>
<tr><th colspan="3">Algorithm</th><th>Description</th></tr>
<tr><td colspan="3">Reference counting</td><td>Counts references to an object</td></tr>
<tr><td rowspan="8">Tracing</td><td colspan="2">Conservative</td><td>Does not know where references are</td></tr>
<tr><td rowspan="2">Exact</td><td>Non-copying</td><td>Knows where references are</td></tr>
<tr><td>Copying GC</td><td>Copies objects during a collection to avoid fragmentation</td></tr>
<tr><td colspan="2">Mono-threaded</td><td>Cannot handle multiple application threads</td></tr>
<tr><td rowspan="2">Multi-threaded</td><td>Stop-the-world</td><td>Suspends the application threads during the whole collection</td></tr>
<tr><td>Concurrent</td><td>Lets the application run during a collection</td></tr>
<tr><td colspan="2">Generational</td><td>Segregates objects in generations</td></tr>
</table>

Figure 2.1: Taxonomy of GC algorithms

Definitions. A GC manipulates *objects* and *references*. An object is a structure in memory while a reference is a unique object identifier. In most MREs, a reference to an object is directly its memory address. This choice is the most efficient for object accesses because it avoids the transformation between the object reference and its address. However, a reference could be implemented in a different manner, for example, as an index in a pointer table. Although less efficient, not representing references as addresses is particularly useful to partition each object in small chunks with fixed size to avoid memory fragmentation [28].

Reference counting versus tracing. GCs are classified into two main families. In a *reference counting* algorithm, each object keeps track of the number of times it is referenced. An object's counter is incremented each time the object is referenced and decremented each time it is dereferenced. The object is immediately freed when this counter reaches zero. Reference counting algorithms are however unable to free object cycles. *Tracing algorithms* consider the memory as a graph where the nodes are the objects and the edges are determined by the reference relation: there is an edge from A to B if and only if A references B. The root set of this graph is the set of the local and global variables. An object is considered

unused if its not reachable from this root set. Tracing algorithms only collect the object graph when the memory reserved for the GC heap is exhausted. With a reference counting algorithm, objects are synchronously freed when they are no longer referenced, while with a tracing algorithm, objects are freed asynchronously during collection phases.

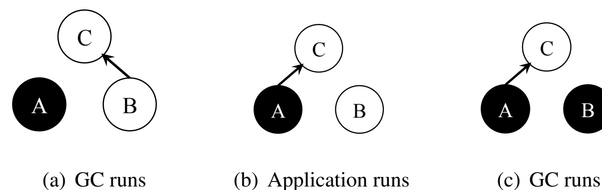
As object cycles are possible in most languages, most MREs use a tracing algorithm. In the remainder of this section, we assume an MRE that uses such an algorithm.

Copying versus non copying. A copying algorithm is able to copy the objects during a collection. The copying algorithm presents two main advantages. First, it makes it possible to eliminate memory fragmentation. Second, it makes it possible to use a very efficient algorithm for allocation that only has to update the pointer to the end of the last allocated object to allocate a new one. Copying algorithms are thus widely used in MREs.

Conservative versus exact. For languages such as C or C++ that do not provide any type information at runtime, the GC cannot determine which memory locations contain references for the object graph traversal. A conservative algorithm handles this class of languages by considering that if a value in memory has the form a valid reference, then it cannot free the referenced object [35]. Such a collector is considered conservative because it will never free a used object, but it can confuse a number with a reference and artificially consider the referenced object as alive. In contrast, an exact algorithm knows exactly whether a memory location contains a reference or a number. Exact collectors require runtime type information.

If the MRE uses the object's address directly as its reference, an *exact algorithm* is necessary to build a *copying collector*. Indeed, copying an object implies modifying its address and, as reference locations contain the object address, updating all its reference locations. If the collector cannot discriminate between addresses and numbers, updating an address would risk modifying a number.

Stop-the-world versus concurrent. Multi-threaded MREs must take into account the possibility that the application and the collector threads may concurrently access to the object graph. Indeed, if a thread of the application modifies the object graph during a collection, the GC can miss some live objects [134, 135]. For example, in the scenario presented in Figure 2.2, C will be freed even through it is still used by the application.



(a) GC runs (b) Application runs (c) GC runs

Figure 2.2: Black means scanned. The GC scans A in (a), then the application modifies the graph in (b), finally, the GC scans B but does not reach C in (c) because A is already scanned.

Collectors that supports multi-threading are classified in two categories: stop-the-world and concurrent.[1] A stop-the-world collector [149] simply suspends all the threads of the application during the

[1]Notice that, counter-intuitively, a parallel collector is not related to the presence of application threads: it is a collector that has strictly more than one collector thread [149].

whole collection to avoid concurrent accesses to the object graph. A concurrent collector lets the application threads continue to run but the application threads inform the collector of any changes in the object graph [134, 135].

Generational. Many studies show that, in practice, in a wide range of applications and languages, recently allocated objects have a greater chance of becoming unused than older ones [121, 133, 145]. A generational collector exploits this observation. When using a generational GC, object are segregated into generations, typically young and old. A generational collector collects the young generation more frequently because this generation is expected to contain most of the unused objects.

Actual algorithms used in MREs. Most efficient MREs are **generational**, **copying** and **stop-the-world**. A **generational** algorithm avoids many useless collections of the whole heap and thus improves the performance by an order of magnitude. When used in conjunction with **copying** the collector moves objects from younger space to older space as the objects age. Finally, most MREs that support multi-threading use **stop-the-world** algorithms because such algorithms do not require code instrumentation to inform the collector when the object graph is modified. They are therefore currently more efficient than concurrent algorithms. However, to build real-time MREs, stop-the-world algorithms are inadequate because the pause time is not predictable. The pause time of a **concurrent** algorithm is more predictable because it suspends the application only during the stack scanning. Hence, multi-threaded real-time MREs use such algorithms.

2.3. INTERACTIONS BETWEEN COMPONENTS

A JIT compiler is essential to achieve good performance. However, generating an optimized native code obscures some information that is essential for efficient GCs, in particular for exact GCs that have to know where the references are in memory. Moreover, concurrent, stop-the-world and generational collectors have to synchronously get information from the application to know its state. The JIT compilers inserts *callbacks* to the collector into the application code to gather this information. Figure 2.3 summarizes how components interact, Figure 2.4 summarizes the information required for each GC algorithm, and the remainder of this section describes the entailed component interactions.

Write barriers for generational and concurrent collectors. When an application modifies memory, it alters the object graph. Concurrent GCs and generational GCs must synchronously know which references in the graph are updated. To allow the use of these GC algorithms, the JIT compiler must insert write barriers: callbacks to the GC executed at each memory write of a *reference* into an *object*.

A concurrent collector needs to know which references are written by the application during a collection because such writes can cause the GC to potentially miss the referenced objects. For example, in Figure 2.2, when the reference to C is written in A, the write barrier informs the collector that C must be re-scanned.[2]

A generational GC has to find the root set of the young object graph when it collects the young generation. Because an old object can reference a young object, the collector must be able to identify old objects that act as root nodes for the young object graph. The JIT compiler thus instruments writes to track which old objects reference young objects.

[2] An alaternate approach is to instrument reads instead of writes because a language that forbids pointer arithmetic can suppose that each written reference was inevitably read from memory before [134].

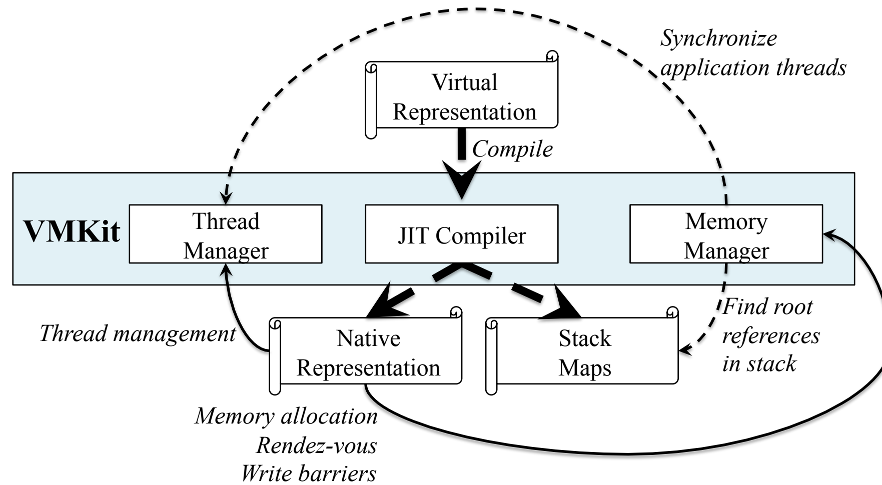


Figure 2.3: Design of an MRE.

<table>
<tr><th colspan="2" rowspan="2">GC Type</th><th colspan="2">Safe points</th><th rowspan="2">Write barriers</th></tr>
<tr><th>Stack map</th><th>GC rendez-vous</th></tr>
<tr><td colspan="2">Conservative GC</td><td>No</td><td colspan="2" rowspan="3"></td></tr>
<tr><td rowspan="2">Exact GC</td><td>Non-copying GC</td><td rowspan="2">Yes</td></tr>
<tr><td>Copying GC</td></tr>
<tr><td colspan="2">Non multi-threaded MRE</td><td rowspan="4"></td><td>No</td><td rowspan="2">No</td></tr>
<tr><td rowspan="2">Multi-threaded MRE</td><td>stop-the-world GC</td><td rowspan="2">Yes</td></tr>
<tr><td>Concurrent GC</td><td rowspan="2">Yes</td></tr>
<tr><td colspan="2">Generational GC</td><td></td></tr>
</table>

Figure 2.4: Interactions between the MRE components

GC rendez-vous for multi-threaded MREs. A GC rendez-vous is a callback to the GC that synchronizes application threads at the beginning of a collection. GC rendez-vous are required for GCs that support multi-threaded MREs, i.e., stop-the-world and concurrent collectors.

A stop-the-world collector suspends all the application threads during a whole collection. All application threads therefore have to reach a GC rendez-vous that suspends them before a collection can begin.

A concurrent collector lets the application continue its execution during a collection by instrumenting memory writes. However, for performance reason, a concurrent collector only instruments writes to object, not to local variables. Accesses to local variables are much more frequent than accesses to objects, and instrumenting them would drastically degrade the performance. To avoid the concurrency issue illustrated in Figure 2.2 when A is a local variable, the application thread has to be initially suspended to find the references in its active function frame [109]. Application threads therefore again have to reach a GC rendez-vous that suspends the application threads at the beginning of a collection to safely let the collector find the references in its stack and in its registers. As soon as the collector has found these references, the GC rendez-vous resumes the execution of the application threads while the collector continues to run.

Stack maps for exact collectors. An exact collector, including a copying one, has to locate references in memory. For global variables and objects, the MRE can maintain itself a memory map that describes the location of the references by typing objects, For local variables, while an interpreter would store them in dedicated structures, a JIT compiler may place them on the execution stack or in registers, implying that the collector cannot foresee at which offsets in the stack and in which registers they will be placed. Therefore, the JIT compiler must generate stack maps that indicate the location of the local variables that contains references for each function frame.

Safe points. Safe points are points in the application code where the GC can safely collect memory. The JIT compiler inserts GC rendez-vous at safe points to synchronize the application threads to begin a collection, and it generates stack maps for these points to allow the collector to interpret the information on the stack.

Carefully choosing where to place safe points, and therefore GC rendez-vous and stack maps, is important. First, maintaining too many stack maps would consumes a lot of memory. Second, the test that checks if a collection is triggered in a GC rendez-vous slows down the execution and thus should not be performed too often. Third, not all points in program execution are safe for copying collectors. Indeed, the JIT compiler generates temporary values from object addresses that point inside objects, for example during array accesses. These temporary values would have to be updated by copying collector in case of an object copy, which would complicate the update process. Fourth, as application threads must reach a safe point to begin a collection, they have to reach a safe point in a finite number of step to avoid a too long synchronization. Safe points must therefore be inserted regularly in the code.

To summarize, safe points must be placed not too often but regularly, and only at points in the code where the JIT is guaranteed not to manipulate temporary values. To address this issues, most MREs insert safe points after function calls and at the beginning of the first basic block[3] of unbounded loops.

Actual instrumentation performed by the JIT compilers of efficient MREs. To summarize, as most MREs are multi-threaded and as most efficient GC algorithms are copying and generational, the JIT compilers have to generate (i) stack maps to allow the use of a copying collector, (ii) GC rendez-vous to allow the use of a stop-the-world or a concurrent collector, and (iii) write barriers to allow the use of a generational, and, possibly, concurrent, collector.

[3] A basic block is a sequence of instructions containing no jump and no label except for the first and the last instruction [151, 152].

3. MRE DESIGN WITH VMKIT

Research in MREs is currently an active topic. In just the last five years, numerous research studies have proposed to improve MREs, for example, with real-time algorithms [18, 28, 70, 116, 142], for efficient inter-operability with other languages and runtimes [49], with new concurrency abstractions [38, 87], with new memory management algorithms [31, 51, 67–69, 99, 100, 109], with new memory optimizations [42, 95, 98], with trace-based compilation [29, 50, 94, 110], with monitoring, benchmarking and debugging [3, 12, 26, 37, 47, 92, 107, 120], with runtimes and languages for multi-cores [44, 48, 148], with safe dynamic update of the code [30, 43], with mechanisms to improve isolation [11, 45, 115, 183], with languages for the Web [168], with new lock algorithms [27], etc. Most of these studies require modifications to existing MREs or the development of completely new MREs. However, modifying an existing MRE or implementing a new one is a daunting task and may require implementing and understanding complex modules such as the JIT compiler and the GC. For example, OpenJDK is eight hundred thousand lines of code, not counting the system library, which contributes 2.3 million lines of code, and the language dependent part of OpenJDK is spread out over the whole code. As a consequence implementing a new feature requires understanding, and potentially modifying, all of this code. To facilitate experimenting with new languages or new language features, it is thus essential to reduce this development burden.

To help experimentation in the domain of MREs, we propose an approach based on splitting the implementation of an MRE into two independent layers. The lower layer consists of the VMKit library, which provides basic components: the JIT compiler for a language-independent intermediate language, the GC, the thread manager and the glue between them. The upper layer, called a personality, instantiates VMKit for a particular high-level language. It defines a complete and specific MRE.

VMKit design. The key challenge in building a library for MRE development is to ensure that it does not impose any design decisions on the personalities. VMKit thus defines the core of an MRE but does not impose any object model, type system or call semantics. These are instead defined by the personality. In particular, VMKit lets a personality control how memory is managed (using a GC, without a GC, on the stack, etc.), and how methods are dispatched (direct call, indirect call, single dispatch, multi dispatch, etc).

Our implementation of VMKit relies on the state of the art third-party projects LLVM [82] for the JIT compiler, MMTk [102, 108] for the GC and the POSIX Thread library [155] for the thread manager. As these projects were not initially designed to work together, a challenge in developing VMKit has been to define the necessary glue between them to construct an efficient and language-independent substrate, without modifying these projects. Technically, the glue consists in adding the generation of stack maps, GC rendez-vous and write barriers (see Section 2) in the JIT compiler and in using them to connect the three components.

Comparison with other work. Compared to other research projects such as JikesRVM [141], OVM [112, 146], ORP [114] or VPU [119] that also help in experimenting with new features in MREs, VMKit presents two main advantages:

- VMKit is language-independent. In contrast, JikesRVM [141] focuses on Java, and .Net [162] focuses on CIL.[1] Indeed, Java and CIL impose a class-based object-oriented type system and suppose a pre-defined common root class for all objects. These MREs are therefore ill-suited to support languages that offer other programming paradigms, such as a prototype-based or a functional paradigm.
- Other attempts to propose a common substrate for different MREs (OVM [112, 146], ORP [114] or VPU [119, 189]) rely on home-made components, which have not been maintained over time. Instead, we have chosen to reuse third-party components that are already supported by large communities, LLVM [82] for the compiler, MMTk [102, 108] for the memory manager and POSIX threads [155] for the thread manager. VMKit thus naturally benefits from a continuous integration of the state-of-the-art advances of these components for only a slight amount of maintenance work that only consists of following the API changes. VMKit has been maintained over the last five years by our team. It continues to integrate the latest commits of the LLVM repository and the last releases of MMTk.

Contributions. The contributions of VMKit are threefold. First, we provide an implementation of the VMKit library. Second, we propose two personalities, a Java Virtual Machine (J3) and a CLI implementation (N3), that illustrate how to use VMKit in developing MREs. Third, we provide assessments of the usability of VMKit to build MREs, and a performance evaluation. Although the JVM and CLI have many similarities, there are enough differences between them to highlight the reusability of VMKit.

The main lessons learned from our work are:

- Different personalities can be built on top of VMKit very easily. Out of the roughly 500,000 lines of code used to implement J3 or N3,[2] only 4% (20,000 lines of code) is devoted to implementing each personality, while the remaining 96% is provided by VMKit. Once we had gained experience with the J3 implementation, which was developed in parallel with VMKit, it took one month for one person familiar with VMKit to develop N3. In other work, we have also developed a variant of the Multitasking Virtual Machine [59] based on J3, in around one month, without needing to change VMKit [183] (see Section 4).
- By providing a substrate that includes a state-of-the-art JIT compiler and GC, VMKit supports the development of MREs that have performance similar to other research or open source MREs, such as Cacao [169], Harmony [170], and Mono [164], on *e.g.*, the DaCapo [53] and the PNetMark [165] Benchmark suites. Nevertheless, optimization opportunities remain. For example, our JVM does not yet provide an adaptive compiler [33] or optimized array-bound and null-reference check elimination [113]. J3 remains therefore 1.2 to 3 times slower than the state-of-the-art MRE JikesRVM.

The work on VMKit was conducted with Nicolas Geoffray [182] during his PhD, helped by Bertil Folliot, Julia Lawall and Gilles Muller. Geoffray's PhD resulted in the development of VMKit and the two personalities (J3 and N3), which are freely available at `vmkit.llvm.orig`. VMKit is currently supported by INRIA with a two-year research engineer. VMKit is also used as a building block of several ongoing research projects: a French CIFRE PhD thesis with Orange Labs to improve isolation between components in set-top-boxes, an ANR project (Infra-JVM) to introduce load balancing between Java virtual machines in a pervasive computing context, a collaboration with the S3 team of the Purdue University to implement a personality for the R language, a research proposal with Scilab Enterprise, Arcelor Mittal, Dassault, OCamlPro and HPC Project to implement a personality for the Scilab language,

[1] CIL is the intermediate representation of the .Net framework.

[2] This number excludes the number of lines of code for the base libraries (such as rt.jar/glibj.jar and mscorlib.dll).

and an internal INRIA research project involving six INRIA teams to propose a Java virtual machine for multicores.

4. Safety with I-JVM

OSGi [161] is a Java-based component platform that is widely adopted as an execution environment for the development of extensible applications, such as Eclipse or Java Enterprise Servers. Extensibility in the OSGi platform is provided through a deployment unit called a bundle, which groups together a set of components that are loaded through a specific Java class loader [65]. The OSGi platform is popular because it provides modularity while still providing efficient communication between components using direct method calls.

Initially, the OSGi platform was designed for environments where all bundles trust each other. Nowadays, it is also being promoted for systems such as next generation Internet home gateways where third party services can be downloaded dynamically. However, the OSGi platform cannot protect a bundle against another malicious or buggy bundle for three reasons:

- First, `java.lang.Class` objects, strings and static variables are shared in the Java Virtual Machine (JVM). The corruption of any one of these entities by a malicious or buggy bundle will impact all bundles.
- Second, a thread can freeze the JVM and deny service to other bundles by exhausting memory or monopolizing the CPU.
- Third, it may be impossible to terminate a bundle that denies service, which makes a shutting down of the entire platform the only solution.

Indeed, a recent work has identified 25 different vulnerabilities in current implementations of the Java/OSGi platform that may either lead to a violation of data integrity or a freeze of the platform [143]. While 17 of these vulnerabilities are due to problems in the implementation of the OSGi framework itself and can be solved by adding suitable security checks, the remaining 8 originate in isolation issues and need to be solved at the JVM level.

One approach to providing isolation in a single JVM, through isolates (or Java processes), is provided by the Multi-tasking Virtual Machine [59] (MVM). MVM duplicates the `java.lang.Class` objects, strings and static variables between isolates. However, to ensure full isolation, MVM confines a thread to a single isolate. As a consequence, communication between two isolates must be done using an RPC-like mechanism, which requires parameter copying and may require thread synchronization [40, 58]. Since the OSGi platform uses communication between bundles heavily, using RPCs would induce a significant overhead [58], contradicting the principle of OSGi that provides efficient inter-bundle communications.

I-JVM is a Java Virtual Machine with lightweight isolates that is specifically designed to support the needs of the OSGi platform by associating each bundle with a separate isolate. The key contribution of I-JVM is to permit thread migration between isolates in order to keep the cost of inter-isolate method calls low. This approach enables complete bytecode compatibility with legacy OSGi bundles by avoiding the need to rewrite inter-bundle method calls. The main features of I-JVM are:

- Memory isolation. As shown by the MVM, making `java.lang.Class` objects, strings and static variables private to an isolate is sufficient to ensure memory isolation in a single JVM.

Therefore, in I-JVM, an isolate cannot access an object of another isolate unless a reference is given explicitly through method invocation.

- Resource accounting. I-JVM keeps track of the current isolate in which a thread is running. This allows recording, for each isolate, the CPU time spent and the amount of memory used by using the algorithm of Price at al. [104] These statistics allow an administrator or a resource monitor to detect denial of service attacks from malicious bundles.

- Termination of isolates. When an isolate terminates, methods of its classes should not be invoked anymore. The termination of an isolate starts by setting a flag. Then to prevent threads from returning to the terminating isolate, I-JVM directly modifies the execution stack. This mechanism is more efficient than instrumenting the invocation bytecodes to check if the isolate is terminated as proposed Rudys et al. [126]. When the isolate returns, an exception is raised and trapped at a lower stack level. All of the objects referenced by the terminating isolate are finally reclaimed by the GC, with the exception of objects shared with other bundles.

I-JVM has been developed by modifying J3, the Java personality of VMKit (see Section 3). We have used I-JVM to run two legacy OSGi platforms: Felix [171] of the Apache community and Equinox [172], the plugin engine of the Eclipse project [173].

Overall the results of this study are:

- I-JVM solves the 8 identified OSGi JVM-related weaknesses [143].

- I-JVM has a 16% overhead on inter-bundle calls. This is an order of magnitude lower than the cost of an RPC call between two isolates. Overall, slowdown of I-JVM is between 1% and 20% on a representative suite of macro-benchmarks.

- I-JVM requires the addition of only 650 lines of code to J3, all of them being implemented in the personality. This result shows that VMKit eases the implementation of new features in MREs because VMKit isolates the language-dependant part of the MRE.

All other projects that provide isolation, termination and resource management with software-based processes [8, 14, 16, 21, 22, 40, 59, 64] (i.e., processes isolated by using type safety) or with hardware mechanisms [41] basically split the address space between the components and prevent cross address space references. Cross address space method invocation is therefore an order of magnitude slower than direct invocation and is not directly compatible with the OSGi specification. I-JVM permits cross address spaces references, decreasing the protection between components that have to communicate when they trust each other, but I-JVM remains compatible with the OSGi specification and only slightly degrades communication performance.

The work on I-JVM was also conducted with Nicolas Geoffray during his PhD. An industrial transfer of I-JVM is currently being conducted by Koutheir Attouchi in his PhD at Orange Labs. This PhD aims at defining policies to isolate components in set-top-boxes where components are provided by untrusted tiers. I-JVM is also at the basis of the Infra-JVM project that has the goal of defining a Java virtual machine specifically adapted to the needs of pervasive computing.

5. Efficiency with MREs for multicore

The last ten years have seen the emergence of multicore architectures [73, 74, 103, 106, 139, 167]. Exploiting these architectures in MREs is required to run high-intensive computing applications such as applications servers (e.g., Tomcat, JBoss, .Net or PhP), simulators (e.g., Avora), image processing applications (e.g., Batik or Sunflow), databases (e.g., H2) or scientific libraries (e.g., Anumer.Lin, the Dambach Linear Algebra Framework, JAMA or Linear Algebra for Java). However, while there is currently substantial research on improving the performance of applications and operating systems [1, 2, 5, 6, 17, 23, 46, 125] and on proposing languages that hide the low-level details of the hardware [55, 91, 118, 153, 159, 160], optimizing MREs for multicore hardware has received less attention.

In this context, with three PhD students, I have worked on improving two mechanisms provided by MREs: the lock mechanism and the GC. For locks, we have proposed the Remote Core Locking mechanism that scales better than all other known lock mechanisms as the number of core increases. For GC, we have studied their performance on modern non-uniform memory access (NUMA) hardware where access latency is a function of the core that triggers the access and the accessed memory location.

5.1. Remote Core Locking

Over the last twenty years, a number of studies [34, 76, 84, 86, 90, 96, 97, 105, 123, 131, 147, 156] have attempted to optimize the execution of critical sections on multicore architectures, either by reducing access contention or by improving cache locality. Access contention occurs when many threads simultaneously try to enter critical sections that are protected by the same lock, causing the cache line containing the lock to bounce between cores. Cache locality becomes a problem when a critical section accesses shared data that has recently been accessed on another core, resulting in cache misses, which greatly increase the critical section's execution time. Addressing access contention and cache locality together remains a challenge. These issues imply that some applications that work well on a small number of cores do not scale to the number of cores found in today's multicore architectures.

Recently, several approaches have been proposed to execute a succession of critical sections on a single *server* core to improve cache locality [86, 97]. Such approaches also incorporate a fast transfer of control from other *client* cores to the server, to reduce access contention. Suleman *et al.* [97] propose a hardware-based solution, evaluated in simulation, that introduces new instructions to perform the transfer of control, and uses a special fast core to execute critical sections. Hendler *et al.* [86] propose a software-only solution, Flat Combining, in which the server is an ordinary client thread, and the role of server is handed off between clients periodically. This approach, however, slows down the thread playing the role of server, incurs an overhead for the management of the server role, and drastically degrades performance at low contention. Furthermore, neither Suleman *et al.*'s algorithm nor Hender *et al.*'s algorithm can accommodate threads that block within a critical section, which makes them unable to support widely used applications such as Memcached.

Remote Core Locking (RCL) is a new locking technique that aims to improve the performance of legacy multithreaded applications on multicore hardware by executing remotely, on one or several dedicated servers, critical sections that access highly contended locks. Our approach is *entirely implemented*

in software and targets commodity x86 multicore architectures. At the basis of our approach is the observation that most applications do not scale to the number of cores found in modern multicore architectures, and thus it is possible to *dedicate* the cores that do not contribute to improving the performance of the application to serving critical sections. Thus, it is not necessary to burden the application threads with the role of server, as done in Flat Combining. RCL also accommodates nested critical sections as well as blocking within critical sections, for example with variable conditions, page faults or ad-hoc synchronizations [4]. The design of RCL addresses both access contention and locality. Contention is solved by a fast transfer of control to a server, using a dedicated cache line for each client to achieve busy-wait synchronization with the server core. Locality is improved because shared data is likely to remain in the server core's cache, allowing the server to access such data without incurring cache misses. In this, RCL is similar to Flat Combining, but has a much lower overall overhead.

We have proposed a methodology along with a set of tools to facilitate the use of RCL in a legacy application. Because RCL serializes critical sections associated with locks managed by the same core, transforming all locks into RCLs on a smaller number of servers could induce false serialization. Therefore, the programmer must first decide which locks should be transformed into RCLs and on which core(s) to run the server(s). For this, we have designed a profiler to identify which locks are frequently used by the application and how much time is spent on locking. Based on this information, we propose a set of simple heuristics to help the programmer decide which locks must be transformed into RCLs. We also have designed an automatic reengineering tool based on Coccinelle [20] for C programs to transform the code of each critical section so that it can be executed as a remote procedure call on the server core: the code within the critical section must be extracted as a separate function [154] and variables referenced or updated by the critical section that are declared by the function containing the critical section code must be sent as arguments, amounting to a context, to the server core. Finally, we have developed a runtime for Linux that is compatible with POSIX threads, and that supports a mixture of RCL and POSIX locks in a single application.

RCL is well-suited to improve the performance of a legacy application in which contended locks are an obstacle to performance, since using RCL enables improving locality and resistance to contention without requiring a deep understanding of the source code. On the other hand, modifying the locking schemes to use either fine-grained locking, lock-free data structures [77, 80, 81, 88, 150], transactionnal memory [38, 52, 56, 79, 87, 89, 138] or RPC [1, 2, 6, 9, 75, 124, 144] is complex and requires an overhaul of the source code. Moreover, the first three do not improve locality.

We have evaluated the performance of RCL for C programs as compared to other locks on a custom latency microbenchmark measuring the execution time of critical sections that access a varying number of shared memory locations. Furthermore, based on the results of our profiler, we have identified Memcached [174], Berkeley DB [15] with two types of TPC-C transactions, two benchmarks in the SPLASH-2 suite [175], and three benchmarks in the Phoenix2 suite [78] as applications that could benefit from RCL. In each of these experiments, we compare RCL to the standard POSIX locks (implemented with NTPL [158]) and to the most efficient approaches for implementing locks of which we are aware: CAS spinlocks, MCS [101, 131], MCS-TP [105] and Flat Combining [86]. Comparisons are made for a same number of cores, which means that there are fewer application threads in the RCL case, since one or more cores are dedicated to RCL servers.

On an Opteron 6172 48-core machine running a 3.0.0 Linux kernel with glibc 2.13, our main results are:

- On our latency microbenchmark, under high contention, RCL is faster than all the other tested approaches. It is over 3 times faster than the second best approach, Flat Combining, and 4.4 faster than POSIX.
- On our benchmarks, we found that contexts are small, and thus the need to pass a context to the server has only a marginal impact on performance.

- On most of our benchmarks, only one lock is frequently used and therefore only one RCL server is needed. The only exception is Berkeley DB which requires two RCL servers to prevent false serialisation of critical sections.

- On Memcached, for Set requests, RCL provides a speedup of up to 2.6 times over POSIX locks, 2.0 times over MCS and 1.9 times over spinlocks.

- For TPC-C Stock Level transactions, RCL provides a speedup of up to 14 times over the original Berkeley DB locks for 40 simultaneous clients and outperforms all other locks for more than 10 clients. Overall, RCL resists better than Berkeley DB locks as the number of simultaneous clients increases.

The work on RCL [177] is conducted in collaboration with two PhD students, Jean-Pierre Lozi and Florian David, helped by Julia Lawall and Gilles Muller. Jean-Pierre Lozi is mainly in charge of the experiments of the C version of RCL and Florian David of the Java version.

5.2. NUMA-AWARE GC

One of the most attractive features of MREs is automatic GC. However, in practice, garbage collection induces a substantial proportion of the resource requirements of an MRE. Hence, GC performance plays a central role in MRE performance, and therefore in application performance as well. Today, MREs are increasingly used for application servers that run on large multicore hardware. To exploit multiple cores, a GC typically uses a parallel, stop-the-world algorithm [149]: during a collection, it stops all application threads to avoid race conditions and starts multiple GC threads in order to reduce the collection time. With parallel GC, the stop-the-world pause time, i.e., the time where the GC runs, is expected to be inversely proportional to the number of cores. We consider that a GC for which this property holds is scalable.

Most current parallel GC algorithms are designed for Symmetric Multi-Processors (SMP) with a small number of processors and a uniform memory access latency. But current multicore hardware relies on Non-Uniform Memory Access (NUMA) where the physical memory is partitioned among *nodes*, i.e., sets of cores connected to a unique memory bank. In these architectures, the time to access a memory location is a function of the address of the location and the node that triggers the access: if the memory location is directly connected to the node, the latency is short, otherwise, it is longer. Previous scalability evaluations of GCs used relatively small multiprocessors and did not target contemporary NUMA architectures [24, 28, 31, 68–71]. Therefore, we believe it is time to re-evaluate the scalability of the current GCs. We seek to answer the following questions:

- Is GC performance critical for applications on multicore hardware? Otherwise, it is not worthwhile improving it.

- Does the GC scale with the number of CPUs? In other words does the stop-the-world pause time decreases as we add more cores?

- If the GC does not scale, what are the bottlenecks affecting its scalability?

We have studied the scalability of GCs in the context of the OpenJDK Java virtual machine, which is one of the most widely used JVMs. OpenJDK provides multiple GCs, all of which are generational. Four main GC techniques are used with several possible combinations of young and old generation collectors. We have evaluated three combinations: (i) Parallel Scavenge with concurrent young generation collection, (ii) Concurrent Mark Sweep with parallel young generation collection [157], and (iii) Garbage First [71]. All three are the most parallel GCs provided by OpenJDK and hence are appropriate candidates for this evaluation. We ran the DaCapo Benchmark suite [53] as it is representative of real-world

Java applications. Our evaluation platform is a 48-core NUMA Magny-Cours with 8 memory nodes: on this architecture, an access to a random memory address has a seven chance out of eight of being distant and therefore costly in term of latency. Our study of GC scalability shows the following:

- DaCapo scalability test: We have evaluated the scalability of DaCapo applications regardless of the garbage collector by using a sufficiently big heap to avoid any collection. Sunflow, Lusearch and Tomcat are among the most scalable benchmarks. We focus on these three benchmarks for our tests because a scalable GC is only useful for a scalable application.
- GC effect: We have evaluated application scalability as a function of the number of cores, where each core runs both an application thread and GC thread. The experiment shows that the application time does not decrease when increasing number of cores while the previous evaluation shows that the application time, without GC, decreases. Therefore, the GC is critical to application scalability.
- GC scalability: We have evaluated GC scalability as a function of the number of GC threads, with constant number of cores and application threads, i.e., 48. Stop-the-world pause time measurements show that after 6 to 10 cores, the pause time monotonically increases. None of the three evaluated GCs scales well as core count increases.

We have also studied the factors that affect scalability and have shown that:

- Remote Scanning and Remote Copying of objects, i.e., scanning and copying of objects that live on remote nodes, are the most crucial factors affecting scalability, as up to 85% of the memory accesses are remote.
- The load is highly unbalanced between the threads that collect the memory. The current load balancing scheme does not take into account the memory topology and has a very high cost: by disabling it, we were able to improve the performance up to 33% in some cases, with no degradation in others. This result shows that a scalable NUMA-aware load balancing scheme is indeed required.

As a continuation of this study, we are currently finishing the development of a NUMA-aware GC. Our GC segregates memory between the nodes: only threads on a given node scan and copy the objects of this node. If a thread reaches an object that does not belong to its node, it sends a message to the node where the object lives and lets this node scan the object.

Compared to other GCs such as the ones of Marlow et al. [67], Ogasawara at al. [51] or Tikir et al. [85] that increase the locality of the application, we focus on increasing the locality of the GC itself. And, compared to GCs for distributed shared memory [10, 83, 130], our NUMA-aware GC supposes a cache-coherent memory where consistency is ensured by the hardware. For this reason, our GC can only improve the memory locality of the collector threads by avoiding remote accesses during the collection, but cannot change either the consistency model or intercept consistency messages.

The implementation of our NUMA-aware GC is almost done and our preliminary results show that we decrease the GC time by up to a factor of 10 on the applications of the DaCapo benchmark on the same 48-core machine.

This work [191] is conducted in collaboration with a PhD student, Lokesh Gidra, helped by Julien Sopena and Marc Shapiro.

6. Conclusion

Contemporary processor architectures are increasingly designed as heterogeneous distributed systems with complex topology. In a single chip or machine, we will find different kind of cores, from general-purpose ones to more specialized ones, for example, graphics processing units (GPUs) or programmable network interface cards (NICs). Each of these cores will have its own instruction set, frequency, memory performance and memory architecture. Some of them will target parallelism, others high sequential throughput. Communication mechanisms between the cores will also become highly heterogeneous: some cores will share their memory, which may no longer necessarily be cache-coherent, while others will explicitly communicate with messages. Communication latency between cores will also be highly heterogeneous as cores will use different communication mechanisms. Current examples of these emerging multicores range from, as the most simple, the NUMA Magny-Cour of AMD, to the Accelerated Processing Units of AMD that provides a CPU and GPU on a single core, and to the Single-Chip Cloud computer of Intel that provide a shared memory, but without cache-coherency by default.

In the future, building on the availability of these diverse platforms, hardware manufacturers will provide a large range of processors targeting different market segments and needs, from embedded devices that have to optimize their energy consumption to servers that have to provide high performance, to set-top-boxes that have to optimize home networks and video games. This proliferation of architecture raises a problem for application development. It will be impossible to specifically target one single architecture: neither to make assumptions on the memory and concurrency models provided by the machine, nor even on the number of cores or the communication latency. Moreover, for irregular high performance applications, such as servers, where the load and the needs can vary at runtime, the optimal placement of the application among the cores can change during its execution. It will therefore be essential to find a new way to develop the application once, regardless of the underlying architecture, and to adapt the application at runtime to its needs and to the hardware.

In this setting, I propose to study how an MRE can continuously adapt the application to the underlying hardware characteristics and to its behavior. Using enhanced MREs to address the problem raised by heterogeneous multicores is an attractive solution because an MRE can already hide the instruction set heterogeneity between the cores by using multiple JIT back-ends. For example, Hera-JVM [48] and Helios [1] already use this technique to hide the instruction set heterogeneity.

To carry out this study of continuous adaptation of applications by MREs, I will explore the following areas:

Ensuring the semantics of the language. As the different hardware will provide different memory models and different communication mechanisms, the MRE will have to adapt the code of the application to ensure that its intended memory model is respected. The MRE will have to implement a distributed shared memory (DSM) [132] that satisfies this expected memory model, and it will have to adapt the code of the application to simulate the DSM. Based on the expected memory model of the application, on the memory model provided by the hardware, and on the available communication mechanism, new DSM algorithms especially tailored for multicores that optimally exploit the hardware mechanisms will have to be proposed.

Non intrusive monitoring. The MRE will have to continuously monitor the application behavior in order to be able to continuously adapt the application to maximize its throughput, its memory accesses pattern etc. However, understanding an application behavior at runtime without altering its performance is an open problem. One approach relies on the hardware counters provided by current processors, for example, the cache miss counters or the false branch prediction counter. Sampling techniques are able to precisely associate these counters to instructions or functions [176], but interpreting counter values is often difficult [19]. For example, a function that causes many data cache misses may suffer either from a poor data locality or from concurrent data accesses from another cores. Preliminary strategies to understand these cache misses have recently been proposed [13, 19]. These strategies use runtime monitors that only slightly degrade the performance (by 5-10% of slowdown), but the analysis of the collected data remains expensive and thus cannot currently be performed on the fly. A trade-off is needed between the accuracy of the profiling and its performance impact on the application to define equivalent monitors for online profiling.

Best code and data placement. On complex hardware, how to place an application on cores in order to maximize its throughput and minimize its energy consumption will be challenging because of the many constraints that have to be taken into account: application behavior, memory latency, communication protocol, expected memory model of the application, frequency of communication between the different threads of the application, possible DSM algorithms, and so on. As a first attempt, I propose to use constraint solvers to periodically compute the best placement. Constraint solvers have already be successfully used in systems research to balance the load of a cluster of virtual machine monitors [111] and to find the optimal initial device configuration of the Barrelfish operating system [93]. To apply a similar technique to adapt applications on the fly to the hardware and their behavior, the main challenge will be the identification of the constraints that can actually help in optimizing the throughput and the energy consumption of the application.

The expected result of this research will be a new generation of MREs that will adapt the application at runtime to the actual multicore hardware on which it is executed. Such MREs will bring a solution to the problems raised by the next generations of multicore architectures.

Bibliography

Main bibliography

[1] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the Symposium on Operating Systems Principles, SOSP '09*, pages 221–234. ACM, 2009.

[2] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the Symposium on Operating Systems Principles, SOSP '09*, pages 29–44. ACM, 2009.

[3] Emre Kiciman and Benjamin Livshits. Ajaxscope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In *Proceedings of the Symposium on Operating Systems Principles, SOSP '07*, pages 17–30. ACM, 2007.

[4] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *Proceedings of the conference on Operating Systems Design and Implementation, OSDI '10*, pages 163–176. USENIX Association, 2010.

[5] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the conference on Operating Systems Design and Implementation, OSDI '10*, pages 1–16. USENIX Association, 2010.

[6] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: an operating system for many cores. In *Proceedings of the conference on Operating Systems Design and Implementation, OSDI '08*, pages 43–57. USENIX Association, 2008.

[7] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Cramm: virtual memory support for garbage-collected applications. In *Proceedings of the conference on Operating Systems Design and Implementation, OSDI '06*, pages 103–116. USENIX Association, 2006.

[8] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. Processes in kaffeos: isolation, resource management, and sharing in java. In *Proceedings of the conference on Operating Systems Design and Implementation, OSDI '00*, pages 23–23. USENIX Association, 2000.

[9] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the conference on Operating Systems Design and Implementation, OSDI '99*, pages 87–100. USENIX Association, 1999.

[10] Paulo Ferreira and Marc Shapiro. Garbage collection and dsm consistency. In *Proceedings of the conference on Operating Systems Design and Implementation, OSDI '94*, pages 229—-241. USENIX Association, 1994.

[11] Lon Ingram and Michael Walfish. Treehouse: Javascript sandboxes to help Web developers help themselves. In *Proceedings of the Usenix Annual Technical Conference, USENIX ATC '12*, page 12. USENIX Association, 2012.

[12] James Mickens. Rivet: browser-agnostic remote debugging for Web applications. In *Proceedings of the Usenix Annual Technical Conference, USENIX ATC '12*, page 12. USENIX Association, 2012.

[13] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. Memprof: a memory profiler for numa multicore systems. In *Proceedings of the Usenix Annual Technical Conference, USENIX ATC '12*, page 12. USENIX Association, 2012.

[14] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The jx operating system. In *Proceedings of the Usenix Annual Technical Conference, USENIX ATC '02*, pages 45–58. USENIX Association, 2002.

[15] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley db. In *Proceedings of the Usenix Annual Technical Conference, USENIX ATC '99*, pages 43–43. USENIX Association, 1999.

[16] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in java. In *Proceedings of the Usenix Annual Technical Conference, USENIX ATC '98*, pages 22–22. USENIX Association, 1998.

[17] Xiang Song, Haibo Chen, Rong Chen, Yuanxuan Wang, and Binyu Zang. A case for scaling applications to many-core with OS clustering. In *Proceedings of the EuroSys European Conference on Computer Systems, EuroSys '11*, pages 61–76. ACM, 2011.

[18] Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, and Jan Vitek. High-level programming of embedded hard real-time devices. In *Proceedings of the EuroSys European Conference on Computer Systems, EuroSys '10*, pages 69–82. ACM, 2010.

[19] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the EuroSys European Conference on Computer Systems, EuroSys '10*, pages 335–348. ACM, 2010.

[20] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the EuroSys European Conference on Computer Systems, EuroSys '08*, pages 247–260. ACM, 2008.

[21] Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. Sealing os processes to improve dependability and safety. In *Proceedings of the EuroSys European Conference on Computer Systems, EuroSys '07*, pages 341–354. ACM, 2007.

[22] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. In *Proceedings of the EuroSys European Conference on Computer Systems, EuroSys '06*, pages 177–190. ACM, 2006.

[23] David Wentzlaff, Charles Gruenwald, III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An operating system for multicore and clouds: mechanisms and implementation. In *Proceedings of the Symposium on Cloud computing, SoCC '10*, pages 3–14. ACM, 2010.

[24] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *Proceedings of the symposium on Principles Of Programming Languages, POPL '93*, pages 113–123. ACM, 1993.

[25] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the symposium on Principles Of Programming Languages, POPL '84*, pages 297–302. ACM, 1984.

[26] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the conference on Programming Language Design and Implementation, PLDI '10*, pages 1–12. ACM, 2010.

[27] Takuya Nakaike and Maged M. Michael. Lock elision for read-only critical sections in java. In *Proceedings of the conference on Programming Language Design and Implementation, PLDI '10*, pages 269–278. ACM, 2010.

[28] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: fragmentation-tolerant real-time garbage collection. In *Proceedings of the conference on Programming Language Design and Implementation, PLDI '10*, pages 146–159. ACM, 2010.

[29] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the conference on Programming Language Design and Implementation, PLDI '09*, pages 465–478. ACM, 2009.

[30] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: a VM-centric approach. In *Proceedings of the conference on Programming Language Design and Implementation, PLDI '09*, pages 1–12. ACM, 2009.

[31] Stephen M. Blackburn and Kathryn S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the conference on Programming Language Design and Implementation, PLDI '08*, pages 22–32. ACM, 2008.

[32] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *Proceedings of the conference on Programming Language Design and Implementation, PLDI '00*, pages 321–333. ACM, 2000.

[33] Michal Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the conference on Programming Language Design and Implementation, PLDI '00*, pages 13–26. ACM, 2000.

[34] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: featherweight synchronization for Java. In *Proceedings of the conference on Programming Language Design and Implementation, PLDI '98*, pages 258–268. ACM, 1998.

[35] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proceedings of the conference on Programming Language Design and Implementation, PLDI '91*, pages 157–164. ACM, 1991.

[36] C. Chambers and D. Ungar. Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *Proceedings of the conference on Programming Language Design and Implementation, PLDI '89*, pages 146–160. ACM, 1989.

[37] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do - a large-scale study of the use of eval in javascript applications. In *Proceedings of the European Conference on Object-Oriented Programming, ECOOP '11*, pages 52–78. Springer-Verlag, 2011.

[38] Lukasz Ziarek, Adam Welc, Ali-Reza Adl-Tabatabai, Vijay Menon, Tatiana Shpeisman, and Suresh Jagannathan. A uniform transactional execution environment for Java. In *Proceedings of the European Conference on Object-Oriented Programming, ECOOP '08*, pages 129–154. Springer-Verlag, 2008.

[39] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *Proceedings of the European Conference on Object-Oriented Programming, ECOOP '08*, pages 104–128. Springer-Verlag, 2008.

[40] Eli Tilevich and Yannis Smaragdakis. J-orchestra: Automatic java application partitioning. In *Proceedings of the European Conference on Object-Oriented Programming, ECOOP '02*, pages 178–204. Springer-Verlag, 2002.

[41] Leendert van Doorn. A secure javatm virtual machine. In *Proceedings of the USENIX Security Symposium '00*, pages 2–2. USENIX Association, 2000.

[42] Xi Yang, Stephen M. Blackburn, Daniel Frampton, Jennifer B. Sartor, and Kathryn S. McKinley. Why nothing matters: the impact of zeroing. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11*, pages 307–324. ACM, 2011.

[43] Thomas Würthinger, Danilo Ansaloni, Walter Binder, Christian Wimmer, and Hanspeter Mössenböck. Safe and atomic run-time code evolution for java and its application to dynamic aop. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11*, pages 825–844. ACM, 2011.

[44] Prasad A. Kulkarni. Jit compilation policy for modern machines. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11*, pages 773–788. ACM, 2011.

[45] Kevin J. Hoffman, Harrison Metzger, and Patrick Eugster. Ribbons: a partially shared memory programming model. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11*, pages 289–306. ACM, 2011.

[46] Tongping Liu and Emery D. Berger. Sheriff: precise detection and automatic mitigation of false sharing. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11*, pages 3–18. ACM, 2011.

[47] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. Automated construction of javascript benchmarks. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11*, pages 677–694. ACM, 2011.

[48] Ross McIlroy and Joe Sventek. Hera-jvm: a runtime system for heterogeneous multi-core architectures. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10*, pages 205–222. ACM, 2010.

[49] Michal Wegiel and Chandra Krintz. Cross-language, type-safe, and transparent object sharing for co-located managed runtimes. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10*, pages 223–240. ACM.

[50] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. Spur: a trace-based jit compiler for cil. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10*, pages 708–725. ACM, 2010.

[51] Takeshi Ogasawara. NUMA-aware memory manager with dominant-thread-based copying GC. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '09*, pages 377–390. ACM, 2009.

[52] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and implementation of transactional constructs for C/C++. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '08*, pages 195–212. ACM, 2008.

[53] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '06*, pages 169–190. ACM, 2006.

[54] Matthew Hertz and Emery D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '05*, pages 313–326. ACM, 2005.

[55] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '05*, pages 519–538. ACM, 2005.

[56] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '03*, pages 388–402. ACM, 2003.

[57] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '02*, pages 130–141. ACM, 2002.

[58] Krzysztof Palacz, Jan Vitek, Grzegorz Czajkowski, and Laurent Daynas. Incommunicado: efficient communication for isolates. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '02*, pages 262–274. ACM, 2002.

[59] Grzegorz Czajkowski and Laurent Daynés. Multitasking without comprimise: a virtual machine evolution. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '01*, pages 125–138. ACM, 2001.

[60] Bowen Alpern, Anthony Cocchi, Stephen Fink, and David Grove. Efficient implementation of Java interfaces: invokeinterface considered harmless. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '01*, pages 108–124. ACM, 2001.

[61] Mauricio Serrano, Rajesh Bordawekar, Sam Midkiff, and Manish Gupta. Quicksilver: a quasi-static compiler for java. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '00*, pages 66–82. ACM, 2000.

[62] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '00*, pages 47–65. ACM, 2000.

[63] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a java just-in-time compiler. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '00*, pages 294–310. ACM, 2000.

[64] Grzegorz Czajkowski and Thorsten von Eicken. Jres: a resource accounting interface for java. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '98*, pages 21–35. ACM, 1998.

[65] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '98*, pages 36–44. ACM, 1998.

[66] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '89*, pages 49–70. ACM, 1989.

[67] Simon Marlow and Simon Peyton Jones. Multicore garbage collection with local heaps. In *Proceedings of the International Symposium on Memory Management, ISMM '11*, pages 21–32. ACM, 2011.

[68] Cosmin E. Oancea, Alan Mycroft, and Stephen M. Watt. A new approach to parallelising tracing algorithms. In *Proceedings of the International Symposium on Memory Management, ISMM '09*, pages 10–19. ACM, 2009.

[69] Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the International Symposium on Memory Management, ISMM '08*, pages 11–20. ACM, 2008.

[70] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *Proceedings of the International Symposium on Memory Management, ISMM '07*, pages 159–172. ACM, 2007.

[71] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the International Symposium on Memory Management, ISMM '04*, pages 37–48. ACM, 2004.

[72] Ting Yang, Matthew Hertz, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Automatic heap sizing: taking real memory into account. In *Proceedings of the International Symposium on Memory Management, ISMM '04*, pages 61–72. ACM, 2004.

[73] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. The 48-core SCC processor: the programmer's view. In *Proceedings of the conference on Supercomputing, Supercomputing '10*, pages 1–11. IEEE Computer Society.

[74] Timothy G. Mattson, Rob Van der Wijngaart, and Michael Frumkin. Programming the intel 80-core network-on-a-chip terascale processor. In *Proceedings of the conference on Supercomputing, Supercomputing '08*, pages 1–11. IEEE Computer Society, 2008.

[75] Honghui Lu, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. Message passing versus distributed shared memory on networks of workstations. In *Proceedings of the conference on Supercomputing, Supercomputing '95*, page 19. ACM, 1995.

[76] Anant Agarwal and Mathews Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the International Symposium on Computer Architecture, ISCA '89*, pages 396–406. ACM, 1989.

[77] Ori Shalev and Nir Shavit. Split-ordered lists: lock-free extensible hash tables. In *Proceedings of the symposium on Principles Of Distributed Computing, PODC '03*, pages 102–111. ACM, 2003.

[78] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the symposium on High Performance Computer Architecture, HPCA '07*, pages 13–24. IEEE Computer Society, 2007.

[79] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the international conference on Distributed Computing, DISC '06*, pages 284–298. Springer-Verlag, 2006.

[80] Edya Ladan-Mozes and Nir Shavit. An optimistic approach to lock-free FIFO queues. In *Proceedings of the international conference on Distributed Computing, DISC '04*, pages 117–131. Springer-Verlag, 2004.

[81] Tim Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the international conference on Distributed Computing, DISC '01*, pages 300–314. Springer-Verlag, 2001.

[82] Chris Lattner and Vikram Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code Generation and Optimization, CGO '04*, pages 75–88. IEEE Computer Society, 2004.

[83] Paulo Ferreira and Marc Shapiro. Larchant: persistence by reachability in distributed shared memory through garbage collection. In *Proceedings of the International Conference on Distributed Computing Systems, ICDCS '96*, pages 394–401. IEEE Computer Society, 1996.

[84] Jose L. Abellán, Juan Fernández, and Manuel E. Acacio. GLocks: efficient support for highly-contended locks in many-core CMPs. In *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS '11*, pages 893–905. IEEE Computer Society, 2011.

[85] Mustafa M. Tikir and Jeffery K. Hollingsworth. NUMA-aware Java heaps for server applications. In *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS '05*, pages 108–117. IEEE Computer Society, 2005.

[86] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures, SPAA '10*, pages 355–364. ACM, 2010.

[87] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Practical weak-atomicity semantics for Java STM. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures, SPAA '08*, pages 314–325. ACM, 2008.

[88] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures, SPAA '04*, pages 206–215. ACM, 2004.

[89] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the symposium on Principles and Practices of Parallel Programming, PPoPP '05*, pages 48–60. ACM, 2005.

[90] Michael L. Scott and William N. Scherer. Scalable queue-based spin locks with timeout. In *Proceedings of the symposium on Principles and Practices of Parallel Programming, PPoPP '01*, pages 44–52. ACM, 2001.

[91] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the symposium on Principles and Practices of Parallel Programming, PPoPP '95*, pages 207–216. ACM, 1995.

[92] Rei Odaira and Toshio Nakatani. Continuous object access profiling and optimizations to overcome the memory wall and bloat. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '12*, pages 147–158. ACM, 2012.

[93] Adrian Schüpbach, Andrew Baumann, Timothy Roscoe, and Simon Peter. A declarative language approach to device configuration. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '11*, pages 119–132. ACM, 2011.

[94] Hiroshige Hayashizaki, Peng Wu, Hiroshi Inoue, Mauricio J. Serrano, and Toshio Nakatani. Improving the performance of trace-based systems by false loop filtering. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '11*, pages 405–418. ACM, 2011.

[95] Tim Harris, Saša Tomic, Adrián Cristal, and Osman Unsal. Dynamic filtering: multi-purpose architecture support for language runtime systems. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '10*, pages 39–52. ACM, 2010.

[96] F. Ryan Johnson, Radu Stoica, Anastasia Ailamaki, and Todd C. Mowry. Decoupling contention management from scheduling. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '10*, pages 117–128. ACM, 2010.

[97] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '09*, pages 253–264. ACM, 2009.

[98] Michael D. Bond and Kathryn S. McKinley. Leak pruning. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '09*, pages 277–288. ACM, 2009.

[99] Michal Wegiel and Chandra Krintz. Dynamic prediction of collection yield for managed runtimes. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '09*, pages 289–300. ACM, 2009.

[100] Michal Wegiel and Chandra Krintz. The mapping collector: virtual memory support for generational, parallel, and concurrent compaction. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '08*, pages 91–102. ACM, 2008.

[101] John M. Mellor-Crummey and Michael L. Scott. Synchronization without contention. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '91*, pages 269–278. ACM, 1991.

[102] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of the International Conference on Software Engineering, ICSE '04*, pages 137–146. IEEE Computer Society, 2004.

[103] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the Design Automation Conference, DAC '07*, pages 746–749. ACM, 2007.

[104] David W. Price, Algis Rudys, and Dan S. Wallach. Garbage collector memory accounting in language-based systems. In *Proceedings of the symposium on Security and Privacy, SP '03*, pages 263–274. IEEE Computer Society, 2003.

[105] Bijun He, William N. Scherer, and Michael L. Scott. Preemption adaptivity in time-published queue-based spin locks. In *Proceedings of the international conference on High Performance Computing, HiPC '05*, pages 7–18. Springer-Verlag, 2005.

[106] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram R. Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabric Pailet, Shailendra Jain, Tiju Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas Apel, Kersten Henriss, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek De, Rob F. Van der Wijngaart, and Timothy G. Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proceedings of the International Solid-State Circuits Conference, ISSCC '10*, pages 108–109. IEEE Computer Society.

[107] Rei Odaira, Kazunori Ogata, Kiyokuni Kawachiya, Tamiya Onodera, and Toshio Nakatani. Efficient runtime tracking of allocation sites in java. In *Proceedings of the international conference on Virtual Execution Environments, VEE '10*, pages 109–120. ACM, 2010.

[108] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. Demystifying magic: high-level low-level programming. In *Proceedings of the international conference on Virtual Execution Environments, VEE '09*, pages 81–90. ACM, 2009.

[109] Gabriel Kliot, Erez Petrank, and Bjarne Steensgaard. A lock-free, concurrent, and incremental stack scanning for garbage collectors. In *Proceedings of the international conference on Virtual Execution Environments, VEE '09*, pages 11–20. ACM, 2009.

[110] Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for web 3.0: trace compilation for the next generation web applications. In *Proceedings of the international conference on Virtual Execution Environments, VEE '09*, pages 71–80. ACM, 2009.

[111] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy: a consolidation manager for clusters. In *Proceedings of the international conference on Virtual Execution Environments, VEE '09*, pages 41–50. ACM, 2009.

[112] K. Palacz, J. Baker, C. Flack, C. Grothoff, H. Yamauchi, and J. Vitek. Engineering a customizable intermediate representation. In *Proceedings of the workshop on Interpreters, Virtual Machines and Emulators, IVME '03*, pages 67–76. ACM, 2003.

[113] Mikel Luján, John R. Gurd, T. L. Freeman, and José Miguel. Elimination of Java array bounds checks in the presence of indirection. In *Proceedings of the conference on Java Grande, JavaGrande '02*, pages 76–85. ACM, 2002.

[114] Michal Cierniak, Brian T. Lewis, and James M. Stichnoth. Open runtime platform: flexibility with performance using interfaces. In *Proceedings of the conference on Java Grande, JavaGrande '02*, pages 156–164. ACM, 2002.

[115] Isabella Thomm, Michael Stilkerich, Rüdiger Kapitza, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Automated application of fault tolerance mechanisms in a component-based system. In *Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES '11*, pages 87–95. ACM, 2011.

[116] Isabella Thomm, Michael Stilkerich, Christian Wawersich, and Wolfgang Schröder-Preikschat. Keso: an open-source multi-jvm for deeply embedded systems. In *Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES '10*, pages 109–119. ACM, 2010.

[117] Christine H. Flood, David Detlefs, Nir Shavit, and Xiaolan Zhang. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the Java^{TM} Virtual Machine Research and Technology Symposium, JVM '01*, pages 21–30. USENIX Association, 2001.

[118] Pekka O. Jääskeläinen, Carlos S. de La Lama, Pablo Huerta, and Jarmo Takala. OpenCL-based design methodology for application-specific processors. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation, SAMOS '10*, pages 223–230. IEEE Computer Society, 2010.

[119] Ian Piumarta. The virtual processor: fast, architecture-neutral dynamic code generation. In *Proceedings of the Virtual Machine Research and Technology Symposium '04*, pages 97–110. USENIX Association, 2004.

[120] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. Jsmeter: comparing the behavior of javascript benchmarks with real web applications. In *Proceedings of the USENIX conference on Web application development, WebApps '10*, page 12. USENIX Association, 2010.

[121] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the software engineering symposium on practical Software Development Environments, SDE '84*, pages 157–167. ACM, 1984.

[122] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI '73*, pages 235–245. Morgan Kaufmann, 1973.

[123] Srinivas Sridharan, Brett Keck, Richard Murphy, Surendar Chandra, and Peter Kogge. Thread migration to improve synchronization performance. In *Proceedings of the workshop on Operating System Interference in High Performance Applications, OSIHPA '06*, 2006.

[124] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. Data-oriented transaction execution. *Proceedings of the VLDB Endowment, PVLDB '10*, 3:928–939, 2010.

[125] David Wentzlaff and Anant Agarwal. Factored Operating Systems (FOS): the case for a scalable operating system for multicores. *SIGOPS Operating System Review (OSR)*, 43(2):76–85, 2009.

[126] Algis Rudys and Dan S. Wallach. Termination in language-based systems. *ACM Transactions on Information and System Security*, 5(2):138–168, 2002.

[127] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21:895–913, 1999.

[128] Urs Hölzle and David Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):355–400, 1996.

[129] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, 1991.

[130] Y. Charlie Hu, Weimin Yu, Alan Cox, Dan Wallach, and Willy Zwaenepoel. Run-time support for distributed sharing in safe languages. *ACM Transactions on Computer Systems (TOCS)*, 21(1):1–35, 2003.

[131] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9:21–65, 1991.

[132] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[133] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[134] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 21(11):966–975, 1978.

[135] Guy L. Steele, Jr. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, 1975.

[136] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3:184–195, 1960.

[137] Kuo-Yi Chen, J. Morris Chang, and Ting-Wei Hou. Multithreading in java: Performance and scalability on multicore systems. *IEEE Transactions on Computers*, 60(11):1521–1534, 2011.

[138] Tim Harris, Adrian Cristal, Osman S. Unsal, Eduard Ayguade, Fabrizio Gagliardi, Burton Smith, and Mateo Valero. Transactional memory: An overview. *IEEE Micro*, 27(3):8–29, 2007.

[139] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[140] Gilles Muller and Ulrik Pagh Schultz. Harissa: A hybrid approach to java execution. *IEEE Software*, 16(2):44–51, 1999.

[141] Bowen Alpern, Steve Augart, Stephen M. Blackburn, Maria A. Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen J. Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark F. Mergen, J. Eliot B. Moss, Ton Anh Ngo, Vivek Sarkar, and Martin Trapp. The Jikes research virtual machine project: building an open-source research community. *IBM System Journal*, 44:399–418, 2005.

[142] Michael Stilkerich, Isabella Thomm, Christian Wawersich, and Schröder-Preikschat. Tailor-made jvms for statically configured embedded systems. *Concurrency - Practice & Experience (CP&E)*, 24(8):789–812, 2012.

[143] Pierre Parrend and Stéphane Frenot. Security benchmarks of OSGi platforms: toward hardened OSGi. *Software - Practice & Experience (SP&E)*, 39(5):471–499, 2009.

[144] Eliseu M. Jr. Chaves, Prakash Das, Thomas J LeBlanc, Brian D. Marsh, and Michael L. Scott. Kernel-kernel communication in a shared-memory multiprocessor. *Concurrency - Practice & Experience (CP&E)*, 5(3):171–191, 1993.

[145] A. W. Appel. Simple generational garbage collection and fast allocation. *Software - Practice & Experience (SP&E)*, 19(2):171–183, 1989.

[146] Krzysztof Palacz, Jason Baker, Chapman Flack, Christian Grothoff, Hiroshi Yamauchi, and Jan Vitek. Engineering a common intermediate representation for the ovm framework. *Science of Computer Programming*, 57(3):357–378, 2005.

[147] Leonid B. Boguslavsky, Karim Harzallah, Alexander Y. Kreinin, Kenneth C. Sevcik, and Alexander Vainshtein. Optimal strategies for spinning and blocking. *Journal of Parallel and Distributed Computing (JPDC)*, 21(2):246–254, 1994.

[148] Mark Summerfield. *Programming in Go: Creating Applications for the 21st Century*. 1st edition, 2012.

[149] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC, 1st edition, 2011.

[150] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[151] Andrew W. Appel and Palsberg Jens. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2003.

[152] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2006.

[153] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann, 2001.

[154] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[155] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming - a POSIX standard for better multiprocessing*. O'Reilly, 1996.

[156] Travis S. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, Department of Computer Science, university of Washington, 2003.

[157] Memory management in the Java hotspot™ virtual machine. Technical report, Sun Microsystems, 2006.

[158] Ulrich Drepper and Ingo Molnar. Native POSIX thread library for Linux. Technical report, RedHat, 2003.

[159] CUDA C programming guide. Technical report, NVIDIA Corporation, 2012.

[160] Google. *The Go programming language specification*, 2012. http://golang.org/ref/spec.

[161] The OSGi Alliance. OSGi service platform core specification, release 4.2. http://www.osgi.org/Release4/Download, 2009.

[162] ECMA International. ECMA-335 common language infrastructure (CLI), 4th edition, 2006.

[163] OpenJDK7. http://jdk7.java.net/, 2012.

[164] The mono project. http://www.mono-project.org, 2012.

[165] DotGNU portable.NET. http://dotgnu.org/pnet.html, 2012.

[166] Tiobe software: tiobe index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, Feb. 2012.

[167] TILE-Gx processor family. http://www.tilera.com/products/processors/TILE-Gx_Family, 2012.

[168] Dart, a new platform for structured web apps. http://www.dartlang.org/, 2012.

[169] Cacao home page. http://www.cacaojvm.org/, 2012.

[170] Apache Harmony home page (project retired since nov 16, 2011). http://harmony.apache.org/, 2011.

[171] Apache Felix home page. http://felix.apache.org/, 2012.

[172] Eclipse Equinox home page. http://www.eclipse.org/equinox/, 2012.

[173] Eclipse home page. http://www.eclipse.org/, 2012.

[174] Memcached home page. http://memcached.org, 2012.

[175] Modified SPLASH-2. http://www.capsl.udel.edu/splash, 2012.

[176] Oprofile home page. http://oprofile.sourceforge.net/, 2012.

PERSONNAL BIBILIOGRAPHY

International Conferences (12)

[177] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote Core Locking: migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the Usenix Annual Technical Conference, USENIX ATC '12*, pages 65–76, Boston, MA, USA, 2012. USENIX Association.

[178] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: ten years later. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '11*, pages 305–318, Newport Beach, CA, USA, 2011. ACM.

[179] Thomas Preud'Homme, Julien Sopena, Gaël Thomas, and Bertil Folliot. BatchQueue: fast and memory-thrifty core to core communication. In *Proceedings of the international Symposium on Computer Architecture and High Performance Computing, SBAC-PAD '10*, pages 215–222, Petrópolis, Brazil, 2010. IEEE Computer Society.

[180] Luciana Arantes, Pierre Sens, Gaël Thomas, Denis Conan, and Leon Lim. Partition participant detector with dynamic paths in mobile networks. In *Proceedings of the international symposium on Network Computing and Applications, NCA '10*, pages 224 – 228, Cambridge, MA, USA, 2010. IEEE Computer Society.

[181] Sergey Legtchenko, Sébastien Monnet, and Gaël Thomas. Blue banana: resilience to avatar mobility in distributed MMOGs. In *Proceedings of the international conference on Dependable Systems and Networks, DSN '10*, pages 171–180, Chicago, IL, USA, 2010. IEEE Computer Society.

[182] Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller, and Bertil Folliot. VMKit: a substrate for managed runtime environments. In *Proceedings of the international conference on Virtual Execution Environments, VEE '10*, pages 51–62, Pittsburgh, PA, USA, 2010. ACM.

[183] Nicolas Geoffray, Gaël Thomas, Gilles Muller, Pierre Parrend, Stéphane Frénot, and Bertil Folliot. I-JVM: a Java virtual machine for component isolation in OSGi. In *Proceedings of the international conference on Dependable Systems and Networks, DSN '09*, pages 544–553, Estoril, Portugal, 2009. IEEE Computer Society.

[184] Nicolas Geoffray, Gaël Thomas, Charles Clément, and Bertil Folliot. A lazy developer approach: building a JVM with third party software. In *Proceedings of the international symposium on Principles and Practice of Programming in Java, PPPJ '08*, pages 73–82, Modena, Italy, 2008. ACM.

[185] Colombe Herault, Gaël Thomas, and Philippe Lalanda. A distributed service-oriented mediation tool. In *Proceedings of the international Conference on Services Computing, SCC '07*, pages 403–409, Salt Lake City, UT, USA, 2007. IEEE Computer Society.

[186] Nicolas Geoffray, Gaël Thomas, and Bertil Folliot. Transparent and dynamic code offloading for Java Application. In *Proceedings of the international conference on Distributed Objects and Applications, DOA '06*, pages 1790–1806, Montpellier, France, 2006. LNCS.

[187] Frédéric Ogel, Gaël Thomas, and Bertil Folliot. Support efficient dynamic aspects through reflection and dynamic compilation. In *Proceedings of the Symposium on Applied Computing, SAC '05*, pages 1351–1356, Santa Fe, NM, USA, 2005. ACM.

[188] Assia Hachichi, Gaël Thomas, Cyril Martin, Simon Patarin, and Bertil Folliot. A generic language for dynamic adaptation. In *Proceedings of the European conference on Parallel processing, EuroPar '05*, pages 40–49, Lisboa, Portugal, 2005. LNCS.

International Journals (2)

[189] Gaël Thomas, Nicolas Geoffray, Charles Clément, and Bertil Folliot. Designing highly flexible virtual machines: the JnJVM experience. *Software - Practice & Experience (SP&E)*, 38(15):1643–1675, 2008.

[190] Gaël Thomas, Frédéric Ogel, Antoine Galland, Bertil Folliot, and Ian Piumarta. Building a flexible Java runtime upon a flexible compiler. In Jean-Jacques Vandewalle David Simplot-Ryl and Gilles Grimaud, editors, *Special Issue on 'System & Networking for Smart Objects' of IASTED International Journal on Computers and Applications*, volume 27, pages 28–47. ACTA Press, 2005.

International Workshops (7)

[191] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. Assessing the scalability of garbage collectors on many cores. In *Proceedings of the SOSP Workshop on Programming Languages and Operating Systems, PLOS '11*, pages 1–5, Cascais, Portugal, 2011. ACM.

[192] Nicolas Palix, Julia Lawall, Gaël Thomas, and Gilles Muller. How often do experts make mistakes? In *Proceedings of the workshop on Aspects, Components, and Patterns for Infrastructure Software, ACP4IS '10*, pages 9–16, Rennes and Saint Malo, France, 2010.

[193] Nicolas Geoffray, Gaël Thomas, Charles Clément, and Bertil Folliot. Towards a new isolation abstraction for OSGi. In *Proceedings of the workshop on Isolation and Integration in Embedded Systems, IIES '08*, pages 41–45, Glasgow, Scotland, UK, 2008.

[194] Nicolas Geoffray, Gaël Thomas, and Bertil Folliot. Live and heterogeneous migration of execution environments. In *Proceedings of the international workshop on Pervasive Systems, PerSys '06*, pages 1254–1263, Montpellier, France, 2006.

[195] Colombe Hérault, Gaël Thomas, and Philippe Lalanda. Mediation and enterprise service bus – A position paper. In *Proceedings of the international workshop on Mediation in Semantic Web Services, Mediate '05*, pages 1–13, Amsterdam, Netherlands, 2005.

[196] Frédéric Ogel, Bertil Folliot, and Gaël Thomas. A step toward ubiquitous computing: an efficient flexible micro-ORB. In *Proceedings of the 2004 ACM SIGOPS European Workshop*, pages 176–181, Leuven, Belgium, 2004. ACM.

[197] Frédéric Ogel, Gaël Thomas, Bertil Folliot, and Ian Piumarta. Application-level concurrency management. In *Proceedings of the NATO workshop on Concurrent Information Processing and Computing, CIPC '03*, pages 1 – 13, Sinaia, Romania, 2003.

French Journals (1)

[198] Frédéric Ogel, Gaël Thomas, Antoine Galland, and Bertil Folliot. MVV : une plate-forme à composants dynamiquement reconfigurables – La machine virtuelle virtuelle. In Jean-Louis Giavitto, editor, *Technique et Science Informatiques (TSI)*, volume 23, pages 1269–1299. Hermes, 2004.

French Conferences and Workshops (8)

[199] Thomas Preud'Homme, Julien Sopena, Gaël Thomas, and Bertil Folliot. BatchQueue : file producteur / consommateur optimisée pour les multi-cœurs. In *Proceedings of the Conférence Française en Systèmes d'Exploitation, CFSE '11*, pages 1–12, Saint-Malo, France, 2011.

[200] Nicolas Geoffray, Gaël Thomas, Gilles Muller, Pierre Parrend, Stéphane Frénot, and Bertil Folliot. I-JVM: une machine virtuelle Java pour l'isolation de composants dans OSGi. In *Proceedings of the Conférence Française en Systèmes d'Exploitation, CFSE '09*, pages 1–12, Toulouse, France, 2009.

[201] Didier Donsez and Gaël Thomas. Propagation d'événements entre passerelles OSGi. In *Proceedings of the 2006 Atelier de travail OSGi™*, pages 1–5, Paris, France, 2006.

[202] Nicolas Geoffray, Gaël Thomas, and Bertil Folliot. Distribution transparente et dynamique de code pour applications Java. In *Proceedings of the Conférence Française en Systèmes d'Exploitation, CFSE '06*, pages 85–96, Perpignan, France, 2006.

[203] Assia Hachichi, Cyril Martin, Gaël Thomas, Simon Patarin, and Bertil Folliot. Reconfigurations dynamiques de services dans un intergiciel à composants CORBA CCM. In *Proceedings of the conférence francophone sur le Déploiement et la (Re)configuration de logiciels, DECOR '04*, pages 159–170, Grenoble, France, 2004.

[204] Gaël Thomas, Bertil Folliot, and Frédéric Ogel. Jnjvm : une plateforme Java adaptable pour applications actives. In *Proceedings of the Conférence Française en Systèmes d'Exploitation, CFSE '03*, pages 1 – 12, La Colle sur Loup, France, 2003.

[205] Gaël Thomas, Bertil Folliot, and Ian Piumarta. Les Documents actifs basés sur une machine virtuelle. In *Proceedings of the 2002 Atelier journées des Jeunes Chercheurs en Systèmes, chapitre français de l'ACM-SIGOPS*, pages 441–447, Hammamet, Tunisie, 2002.

[206] Bertil Folliot and Gaël Thomas. Protocole de membership hautement extensible : conception est expérimentations. In *Proceedings of the Conférence Française en Systèmes d'Exploitation, CFSE '01*, pages 25–36, 2001.

Others (4)

[207] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. Assessing the scalability of garbage collectors on many cores. *Best papers from PLOS'11, SIGOPS Operating System Review (OSR)*, 45(3):15–19, 2011.

[208] Jean-Pierre Lozi, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote Core Locking: Migrating critical section execution to improve the performance of multithreaded applications. Work in progress in the Symposium on Operating Systems Principles, SOSP '11, 2011.

[209] Jean-Pierre Lozi, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote Core Locking (RCL): migration of critical section execution to improve performance. Poster at the EuroSys European Conference on Computer Systems, EuroSys '11, 2011.

[210] Nicolas Geoffray, Gaël Thomas, Charles Clément, Bertil Folliot, and Gilles Muller. VMKit: a substrate for virtual machines. Poster at the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '09, 2009.

Book chapters (5)

[211] Olivier Marin, Sébastien Monnet, and Gaël Thomas. Distibuted Systems: Design and Algorithms. Number 4, chapter Peer-to-Peer storage, pages 59–80. John Wiley & Sons, Ltd., 2011.

[212] Sébastien Monnet and Gaël Thomas. Distibuted Systems: Design and Algorithms. Number 5, chapter Large-Scale peer-to-peer game applications, pages 81–103. John Wiley & Sons, Ltd., 2011.

[213] Bertil Folliot and Gaël Thomas. *Techniques de l'Ingénieur*, chapter Virtualisation logicielle : de la machine réelle à la machine virtuelle abstraite, pages 1–15. Hermes, 2009.

[214] Emmanuel Saint-James and Gaël Thomas. *Systèmes répartis en action : de l'embarqué aux systèmes à large échelle*, chapter Applications pair-à-pair de partage de données, pages 223–256. Number 11. Hermes, 2008.

[215] Frédéric Ogel, Gaël Thomas, Ian Piumarta, Antoine Galland, Bertil Folliot, and Carine Baillarguet. Towards active applications: the virtual virtual machine approach. In Mitica Craus, Dan Gâlea, and Alexandru Valachi, editors, *New Trends in Computer Science and Engineering*, pages 1 – 21. A92 Publishing House, polirom press edition, 2003.

PhD thesis (1)

[216] Gaël Thomas. *Applications actives : construction dynamique d'environnements d'exécution flexibles homogène*. PhD thesis, Université Pierre et Marie Curie, Paris, France, 2005.

Abstract. With the advent of the Web and the need to protect users against malicious applications, Managed Runtime Environments (MREs), such as Java or .Net virtual machines, have become the norm to execute programs. Over the last years, my research contributions have targeted three aspects of MREs: their design, their safety, and their performance on multicore hardware. My first contribution is VMKit, a library that eases the development of new efficient MREs by hiding their complexity in a set of reusable components. My second contribution is I-JVM, a Java virtual machine that eliminates the eight known vulnerabilities that a component of the OSGi framework was able to exploit. My third contribution targets the improvement of the performance of MREs on multicore hardware, focusing on the efficiency of locks and garbage collectors: with a new locking mechanism that outperforms all other known locking mechanisms when the number of cores increases, and with a study of the bottlenecks incurred by garbage collectors on multicore hardware. My research has been carried out in collaboration with seven PhD students, two of which having already defended. Building on these contributions, in a future work, I propose to explore the design of the next generation of MREs that will have to adapt the application at runtime to the actual multicore hardware on which it is executed.

Résumé. Avec l'avènement du Web et du besoin de protéger les utilisateurs contre des logiciels malicieux, les machines virtuelles langage, comme les machines virtuelles Java et .Net, sont devenues la norme pour exécuter des programmes. Ces dernières années, je me suis principalement intéressé à trois aspects des machines virtuelles : leur design, leur sûreté de fonctionnement, et leur performance sur les architectures multicœur. Ma première contribution est VMKit, une librairie qui facilite le développement de nouvelles machines virtuelles performantes en cachant leur complexité dans un ensemble de composants réutilisables. Ma seconde contribution est I-JVM, une machine virtuelle Java qui élimine les 8 vulnérabilités connues qu'un composant de la plateforme OSGi était capable d'exploiter. Ma troisième contribution vise à améliorer les performances des machines virtuelles sur les architectures multicœur en se focalisant sur les verrous et les ramasse-miettes : avec un mécanisme de verrouillage qui surpasse tous les autres mécanismes connus lorsque le nombre de cœurs augmente, et avec avec une étude des goulots d'étranglement des ramasse-miettes sur les architecture multicœur. Ces travaux ont été menés avec sept étudiants en thèse, deux d'entre eux ayant déjà soutenu. Basé sur ses contributions, dans mes futurs travaux, je propose d'explorer le design de la prochaine génération de machines virtuelles qui va devoir adapter l'application à la volée pour l'architecture multicœur réelle sur laquelle elle s'exécute.