Resource accounting and reservation in Java Virtual Machine

Inti Gonzalez-Herrera, Johann Bourcier, Olivier Barais

January 24, 2013

1 The Java platform

The Java programming language was designed to developed small application for embedded devices, but it was a long time ago. Today, Java application are running in many platforms ranging from smartphones to enterprise servers. Modern pervasive middleware is typically implemented using Java because of its safety, flexibility, and mature development environment. However, the Java virtual machine specification has not had a major revised since 1999 [50]. It was designed to execute only a single application per instances, and thus it does not provide resource accounting or per-application resource reservations in the sense most middlewares consider application. Often, current pervasive middlewares are thus unable to reserve resources for critical applications, which may cause these applications to crash or hang up when there are not enough resources. Moreover, contemporary middlewares are unable to provide resource accounting. It makes impossible to apply adaptation policies to optimize resource use.

Several researches had addressed these important issues. As result of these efforts some Java specification requests (JSR) have emerged. We consider there are seven JSRs relate to monitoring and to resource accounting and reservation: three for the Java Management eXtension API (JSRs 3, 160, 255), two for Metric Instrumentation (JSRs 138, 174) and two for resource-consumption management (JSRs 121, 284). The Java Management extension API only addresses monitoring and management: it does not define specific resource accounting or reservation strategies. JSRs 138 and 174 define monitors for the Java Virtual Machine. They are coarse grained, monitoring the number of running threads, the memory used, the number of garbage collections and so on. They monitor the entire virtual machine, not specific component so, they are useless to most middleware. Based on the Multitasking Virtual Machine [21], JSR 121 defines isolated JVM instances running in the same OS process, each instance with its own resources. Based on JRes [23], JSR 284 defines resource accounting for Java threads and processes. Together, JSRs 121 and 284 permit resource accounting and reservation for applications. However, the notion of application in JSR 121 is similar to a process at the OS level and it is incompatible with the notion of application of pervasive middlewares like OSGi [4], which relies on a notion of component.

1.1 Java Runtime Environments

A Java Runtime Environment (JRE) is composed by the java virtual machine (JVM) and the class libraries which contains a set of classes with basic functionalities. The class libraries is informally called classpath. Although both components have a well defined responsibility, they depend on each other. Usually, the classpath defines interfaces to be implemented by the virtual machine.

An exhaustive description of every Java Runtime Environment is a long task and for the purpose of this work such description is useless. The goal of this section is to give a brief introduction to some tools in the area of JVM which we consider remarkable to the object of study.

1.1.1 Java classpath implementations

There are several implementations of Java classpath. The reference implementation is Sun JDK which is shared with OpenJDK. An implementation released under the GNU/GPL license is GNU Classpath

which lack of support for Java 6 and Java 7. However, it is used for several projects like JikesRVM, SableVM, JAOS, JNode, Vmkit and so on. Apache Harmony is a JRE implementation which includes its own libraries implementation. As the classpath is a big piece of software it is not always suitable to include it into resource limited devices so, several subsets of JDK has been proposed.

1.1.2 Jikes Research Virtual Machine. JikesRVM

JikesRVM is a Java virtual machine written in Java. JikesRVM was built in the context of Jalapeño project from IBM. The main goals of the project were 1) develop a high-performance Java^{TM} server running on PowerPC multiprocessor architecture under AIX operating system, and 2) provide a flexible research platform to test new ideas [7, 6].

Overall organization

JikesRVM does not interpret the bytecode of applications. The bytecode is compiled at runtime and several compilers are available. These compilers are designed to handle the trade-off between development time, compile time and run time. A simple compiler pass is applied the first time a method is used. The generated code is not optimal but a short response time is obtained in this way. Further calls to a given method may trigger an additional compilation stage. In this case the *optimizing* compiler is used. The optimizing compiler uses three different registred-based representation of bytecode, high-level, low-level and machine-level, to optimize the generated code [7, 6].

The thread subsystem is built on top of AIX pthreads, and the locking mechanisms are implemented to avoid operating system calls. In the AIX implementation, JikesRVM maps a AIX thread to each physical processor in a structure called virtual processor. After that, many Java threads are multiplexed in a single virtual processor [7, 6].

The objects model was carefully designed to reduce the overhead of null-checks and to guarantee fast access to field objects and array elements. Other features supported are fast access to static members and fast virtual method dispatch [7, 6].

Runtime services, like exception handling, dynamic type-checking, reflection, dynamic class-loading and so on are provided through in Java implementations rather than using a low level languages. In fact, the amount of code written in other language than Java is around 6000 lines, most of them in boot loader [8].

Several memory managers are implemented, ranging from generational collectors to copying collectors. The memory manager is implemented as a separated component called MMTK. This library has been widely used as research tool in several independent projects [17, 16, 45, 18].

While building a runtime environment, access to low-level resources is needed, for instance, a memory manager cannot be built on top of type-safe language like Java because some arithmetic of pointer is needed [40]. A component called MAGIC is used to circumvent the problem. Magic library provides a set of classes to access low-level resource through a safe interface. The library is implemented in Java but with a null implementation which allows to use standard development tools. The JIT compiler injects custom code when classes from MAGIC are loaded. Code generated for each method performs the corresponding low-level operation [27].

A very important feature of JikesRVM is the bootstrap process which need a host JVM to build an image with the virtual machine. This image is loaded by a native program written in C [44].

There are more than 200 publications and at least 40 dissertations about JikesRVM or using it. The hot topics are related to compile time optimization [20, 49, 35] and memory managers optimization [17, 16]. However, others topics like heterogeneous runtime environments and low-level high-level programming are discussed as well [27, 40, 3, 51].

JikesRVM supports i32 architecture running on top of Linux, Solaris, OSX and Windows, it supports PowerPC 32 and 64 bits architecture running Linux, AIX and OSX. Support for x86_64 is being implemented. Two projects from Google Summer of Code 2012 are 1) port of the optimizing compiler to x86_64 and 2) port of the optimizing compiler to ARM/Linux. The result of this effort is unknown to the authors.

MMTK

MMTK is a key element of JikesRVM in charge of memory management. It implements canonical garbage collectors like mark-sweep, copying and incremental. In MMTK, a particular collector is represented for a plan which is a set of assemblable components [17, 16].

A policy in MMTK indicates how to handle different areas of virtual memory. For instance, a common approach is to use different allocator and collection policy for large objects or objects with long life, sometimes a given set of objects can be marked as immutable and so on. It allows a great number of combination and an easy way for testing new ideas in separate pieces like allocators and tracers. Additionally, applying specific strategies to a given set of objects is a well known technique to improve the overall performance of system [17].

Garbage collection is still rejected in some contexts because of the performance overhead it imposes. Several results which use MMTK have been published around this topic. Blackburn et al. [18] present Immix, a garbage-collection technique which uses concepts from mark-sweep and copying collector in an attempt to take the best of both approaches. Bacon et al. [11] present a real-time garbage collector, Metronome, which is capable of ensuring a minimum ratio of processor time to mutators. Shahriyar et al. [45] present an interesting study about the performance of reference counting collector and some techniques to improve its performance. Some studies about tunning a collector with different policies have been published. Blackburn et al. [16] present a comparative study between copying semi-space, mark-sweep, and reference counting in the context of JVM.

1.1.3 Kaffe

Kaffe is a separate implementation of both the JVM and the library classpath [1]. It has been ported to several architectures like Alpha, x86, x86_64, ARM, MIPS and so on. The execution model is chosen at built time and it can operate like interpreter or JIT compiler. A conservative garbage collector is used and the heap is shared between application objects and virtual machine objects. Perhaps, the most interesting feature of Kaffe is its thread model. Threads in Kaffe are not implemented on top of native operating system threads as default. Instead of this, a single thread and signals are used to handle contexts switch and synchronization. The solution has the advantage of easy porting to new systems, in particular to embedded architecture without operating systems. However, this has the clear disadvantage of lack of scalability on todays multi-core architectures. Implementing threads on top of native threads is also possible but not the default option.

1.1.4 VMkit

The Virtual Machine Kit (VMkit) is a framework to support the easy development of virtual machines [31]. The main goal is to offer a platform to test new ideas. Creators of Vmkit claim that there are several facts to consider before any attempt to create a virtual machine:

- Building a virtual machine is a complex task which requires knowledge from many areas like programming languages, compiler theory, system programming, automatic memory management and so on. It implies a lot of developing time.
- Virtual machines, like JVM or CLI, use to share abstract specification like threading and automatic memory management as well as implementation details like just in time compilation.
- There are professional tools which address individual tasks like just in time compilation and automatic memory management.

Vmkit's authors state that instead of implementing all functionalities, it is a better approach to reuse existent components. The vision of VMKit follow [30, 48]:

1. Offer VMKit as a common subset of functionalities to virtual machine implementers. The subset must include:

- Threads.
- Just in time compilation.
- Automatic memory management.

2. Write code for virtual machine specification using the common substrate.

VMkit rely on three components to handle the main concerns 1) thread support is built upon Posix threads 2) automatic memory management is built using MMTK and 3) just in time compilation is implemented using LLVM. VMKit is the glue between these components.

VMKit has been successful used to build a Java Virtual Machine, a Common Language Infrastructure and a prototype R implementation. Additionally, vmkit has been used in some research projects to extend JVM [29] and to provide a partial MRE for graphic cards [52].

Using LLVM

The Low-Level Virtual Machine is a toolkit supporting compilers and tool-chain construction [39]. A key concept is a clever separation between different compilation stages which allows using LLVM in many contexts [2]. The main project in LLVM is in charge of translating LLVM IR bytecode into native code. However, this process can be performed in two ways 1) static compilations and 2) just in time compilation. LLVM IR bytecode is complete, this means that not external references are needed. We can describe roughly the steps to use LLVM:

1. Generate LLVM IR bytecode
2. Select a backend
3. Select optimization steps
4. Perform native code generation with LLVM

Although you can generate LLVM IR code directly, LLVM provides an API to perform this step using the well-known concept of basic block. By using this scheme, it is easy to create a compiler for a new language. In the same way and because the completeness of LLVM IR bytecode, writing a new backend is more manageable. CLang and Dragonegg are samples of this approach [2].

The approach in VMkit is translating the instructions for the VM into LLVM IR bytecode (e.g translating java bytecode into LLVM IR bytcode) and then apply a just in time compilation stage. The major problem with this process is related to automatic memory management. The LLVM IR code generator must indicate to LLVM the roots in the heap in order to build the stack map for garbage collector [29].

Using MMTK

We presented a brief description of MMTK as part of JikesRVM virtual machine in section 1.1.2. The problem of using MMTK in VMkit is that MMTK is implemented in Java while LLVM is written in C++. Bootstrapping is needed in order to build a virtual machine. The solution was to implement a JVM as a key component of VMKit codebase relying on a simple garbage collector. Bootstrap process has three stages:

1. Create a simple tool which resemble a JVM.
2. Use the created tool to generate the LLVM IR code for MMTK.
3. Link the code produced in step 2 with the rest of the system to generate the final virtual machine.

Note that a JVM is required even if the target virtual machine is different.

Additionally, VMKit must implement MAGIC library to satisfy MMTK dependencies. As was mentioned the LLVM IR generator must specify heap's root. The generator must specify the write barriers or read barriers to support specific collectors.

1.2 Using Java to build modular systems

In this section, we present a group of technologies, specifications and so on, from industry and research community. The technologies presented are useed to build modular systems and to support service-oriented development. Although the various approaches differ, all these technologies can take advantage of resource-aware programming. Any of such technology can be used as example to proof that current approaches to resource accounting and resource reservation.

We first present the Open Services Gateway initiative (OSGi). In section 1.2.2 we present a brief introduction to Kevoree.

1.2.1 OSGi

The OSGi [4] technology is a specification produced by the OSGi consortium. The OSGi consortium is composed of a significant number of leading companies in various fields of computing, such as IBM, Motorola, Nokia, BMW, Alcatel and EDF. The OSGi specification is composed of three parts:

- A packaging and deployment model.
- A dynamic service-oriented architecture.
- A set of common technical services.

The OSGi specification has now reached version 5. The original specification targeted embedded gateways such as domestic platforms or mobile phones. Since then, the specification has greatly evolved and can treat a much broader spectrum of applications, ranging from embedded applications, the telephony field and vehicles, to large applications running on clusters of servers, the JEE application server [34], and integrated development tools such as Eclipse.

The OSGi specification takes into account the whole software life cycle from its packaging to its uninstallation. Notably, OSGi specifies a system that manages the deployment of an application. This system can independently deploy different pieces (Bundles) of an application. OSGi deployment units define dependencies to other deployment units. When launching an application, the system verifies that all needed dependencies are present. The life cycle of an OSGi deployment unit include operations like, installing, starting, stopping, updating, and removing each deployment units independently.

The OSGi platform is based on the Java language and is consequently closely linked with this language. The OSGi is a centralized platform with a service registry. Service descriptions are implemented using a Java interface and a set of properties whose semantics are defined by the users. The invocation of a service is achieved through standard method calls of Java object approach.

The operation of the service architecture proposed by OSGi is as follows:

1. A service provider publishes its services in service directory provided by OSGi.
2. A service consumer contacts the registry and searches the services to find a provider that corresponds to its needs. The directory treats the query and returns a set of available providers corresponding to the consumer's expectations.
3. The consumer invokes naturally the service provider(s) found.

The OSGi service registry is the cornerstone of the OSGi system. It can find at any time all the available services. The role of the registry is to maintain the correspondence between the descriptions of services and the available service providers.

A description of an OSGi service is composed of:

- A Java interface that defines the functional part of the service. The interface fits the description of the service and is generally reusable among several providers of the same service. This interface defines the functional scope of the service and the syntax of the call.

- A set of properties that characterize the service provider. The properties are used to provide additional information on a particular service provider. Properties are not limited in number and their content is entirely unconstrained.

Once published in the registry, a service is discoverable by consumers. When a consumer is looking for a service it must specify:

- The desired functionality that is represented by the Java interface corresponding to the desired service.
- A query on the properties. This query takes the form of a logical expression defining a set of constraints on the properties of the service provider.

OSGi provides a dynamic service-oriented architecture. For this purpose, the OSGi platform provides a set of primitives and mechanisms that make possible the creation of applications that can be dynamically informed of the availability of services. In addition OSGi provides a primitive for the withdrawal of a service from a directory. Once a service has been withdrawn, consumers can no longer find the service in the registry. Nonetheless, consumers already bound to the service may continue to invoke it.

To address this problem, OSGi sets up a system to inform consumers of any change on a service that interests them. To do that, consumers must show their interest for certain services to the OSGi directory. Then, the platform takes care to inform them when a service corresponding to their interests becomes available or unavailable.

The two aforementioned mechanisms are very powerful because they constitute the basis of a dynamic service-oriented architecture. Developers must act accordingly to all events that may occur and must predict the functioning of their service depending on all possible events.

1.2.2 Kevoree

Kevoree is an open-source dynamic component model. To support dynamic adaptation of systems, Kevoree is built upon the ideas of models at runtime [19]. Models@runtime pushes the idea of reflection one step further [41]. In reflection, models of running system represent a low-level abstraction of the system. That is, models are very close to the implementation model. In contrast, more abstract models like structure and behavior, are considered in Model@runtime. Models can be decouple for reasoning, modification and later automatically resynchronized with a running instance.

A main feature of Kevoree is providing support for distributed models@runtime. In this way, Kevoree is addressing the Internet of thing. To proper support distributed model@runtime, Kevoree introduces the concepts of *Node* and *Groups*. These concepts allow to model the topology which describe the network of heterogeneous nodes. The modeling is done by defining inter-node communication during synchronization of running models. Kevoree use the concept of *Channel* which allow multiple communication semantics between *Components* deployed on heterogeneous nodes. Kevoree supports multiple kinds of execution technology for nodes like Java, Android, MiniCloud, FreeBSD and Arduino [26].

The adaptation engine is a key subsystem in Kevoree. It relies on a model comparison between two Kevoree models to compute a script for a safe system reconfiguration. This is the mechanism used in Kevoree to support the link [19] between the model and the running system [26]. Execution of this script transform the system from one configuration to the new selected configuration [41]. Model comparison yields a delta-model defining changes that should be applied on the source model to obtain the target model. The delta-model can be used to generate a Kevoree script. The Kevoree Script language (KevScript) is a DSL for describing reconfiguration. Execution of a KevScript directly adapts a Kevoree system, without the need for a full Kevoree model definition [26].

1.3 Discussion

2 Resources Reservation

Computer systems must reserve resources to execute an application. The nature of these resources vary in the sense that at certain point in time a given resource can be shared among several applications or not. The classical example of shareable resource is memory. In the other hand we have CPU which cannot be shared. Resource limitation has been a very important factor in computer science for quite a long time because writing applications for an environment with unlimited resources is easier than for real computer systems. Lot of works has been done to hide the resource-limitation problem, some examples are multiplexing CPU usage and creating virtual memory subsystems. As the resource problem is hard to solve, a major concern of researchers has been to offer resource abstractions to programmers. A well-known technique addressing this issue is automatic memory management. Most time, operating systems and others runtime environments perform quite well handling resources. However, resource limit are still there and it can produce applications failures. Additionally, the performance of application is affected because resources management interfaces use to be quite general.

Every single application is subject to failure due resource limitation. This risk is increased in embedded devices and other kind of resources constrained computer's systems. For many applications a failure related to resources limitation is not a big deal. Users can be unhappy if their text editor turn slow or even fail but most time nothing really important depends on it. However, there are other applications where such failures are unacceptable.

A system is considered critical if its failure can result in financial losses, environmental damage, injuries of human beings and others [38]. A more general definition, less catastrophic, is centered in business. For a business, a software application is critical if its failure stop the proper running of business [38]. For instance, a system which handle customer's orders is critical for an online-selling business like eBay or Amazon. Other domain, where failure is not acceptable, is real-time applications. Real-time applications are those which demand an instantaneous response to external events [1]. Critical applications and real-time applications are two different sets [2]. Of course, the intersection between these two sets is big but it is important to remember the difference because it has several implications.

For simple applications, not critical and not real-time, resource limitation is still important because it can prevent normal execution. Of course, this is a matter of logic but in today's dynamic application it is a major concern. In the past, developers could define easily hardware requirements. However, dynamic applications can be deployed in resources constrained computer systems and several developers can add components at runtime so, it is not easy for developers to say when a given component will work. Resource limitation must stop being a concern just for component's developers and being a concern for middleware's developers.

Response time is the major concern in real-time applications so, CPU tends to be the main resource to ensure. For critical applications, as well as for general applications, resource's needs vary. Sometimes, it is more important to ensure network bandwidth, or memory, or disk quota and so on. It just depends on application requirements. The common factor is the resource must be available when needed, real resource not a virtualized one. Resource reservation is the only way to ensure availability. It can be as simple as marking a region of memory as owned by a single task or as complex as providing a task's scheduler for real-time systems where real-time applications have higher priority. There is a lot of related work in real-time's domain [37, 53, 34, 5]. Another interesting approach to resource management is exokernel concept [25] where the notion of resource management and resource protection is decoupled.

In Java world, resource reservation is almost inexistent. Resources are considered as a low level concept which must be handled by the runtime environment. Although most time this is a proper approach, there is an increasing interest in dealing with resources within Java applications. There are some relevant works in the domain we want to discuss. In the following sections we present some related works not just from Java domain but from operating systems and cloud computing too.

[1] There are two kinds of real-time applications, 1) soft real-time application and 2) hard real-time application. Response time is relaxed in former

[2] A computer game is real-time application because the need of real-time rendering but it is not a critical application. A datacenter is critical for most companies but it is not a real-time application

2.1 Resource Containers

Banga et al. propose in [13] a new operating system abstraction for resource management. This abstraction, called resource containers, allows a fine-grained control over definition of independent task and its resources. The intended target of their work are server systems but it is still relevant to other applications.

Banga et al. state that in current operating systems there is an incorrect association between protection domain and resource principals. The protection domain of a task is equal to the resource principal of this task. Process is the abstraction to both protection domain and resource principal in current operating systems. Reasons to use different entities for protection domain and resources principals can be, among others, security and performance. This kind of design decisions is very common in HTTP servers and time-consuming applications.

A number of examples are presented in [13] to highlight the problems. We show a summary of some examples in the following paragraphs.

In applications accessing network, the operating system kernel play a major role. In such a case, process is the correct unit for protection isolation. However, resources used by the kernel to satisfy application's request are not charged to application. This issue has been addressed by both monolithic and micro-kernel operating systems [14]. The conclusion is that in client/server architecture consumed resources by server in behalf of a client must be charged to resource principal of client. This avoid denial-of-service attack and provide QoS. Of course, this is impossible in distributed applications but it is possible in component based applications.

Some applications are split into different processes to guarantee fault isolation between component. These applications still perform a single task so, the natural protection domain is different to the desired resource principal. Observe how in this case the desired unit of resource management is bigger than a process. However, today operating systems manage resources per component in such kind of application. This scenario is common in extensible frameworks written in unsafe language where third-party extensions can be loaded. In Java world this is a common scenario too. Although Java is a type-safe language, there are other ways to produce failure due lack of isolation so, in early solutions a single instance of Java was used per application.

Other scenario is possible. A single process can perform several independent task to avoid context-switching and to reduce IPC overhead. In this case the resource management unit is smaller than a process. Resources unit in this case is the set of all resources used for the process to accomplish a single task. This setting is very common in today Java middlewares. The common solution is to provide an additional isolation unit on top of Java's classloaders.

Even in more recent operating systems where thread assume some roles of resource principal (e.g CPU usage is not longer assigned to processes but to threads) process is still resource principal for memory and files. However, the problem is not the size of resource principal. For instance, suppose we make threads the resource principal and processes the unit of protection domain, in such a case we still have the following scenario: we can multiplex a single thread between several independent tasks.

The argument is that tying resource principal to static concepts like process, thread or Java instance is wrong because it reduce the set of solutions. A flexible definition of resource principal can provide developers with a better way to deal with different concerns like security and performance.

A resource container is the new abstraction proposed in [13] to deal with resource principal for tasks. It is an entity which contains all the system resources used by a particular independent task [13]. Resource containers have attributes like scheduling parameters, resource limits and so on.

As was mentioned, in classical operating systems there is a fixed binding between processes/threads and resource principal. Resource consumption of process/thread is charged to the associate process. However, resource containers allow having dynamic binding between threads and resource principal and this under the control of application. In this way several scenarios are possible, for instance:

- Threads of different processes sharing resource container.
- Multiplexing a thread between several resource containers.

Resource usage is charged to the correct container and allocation algorithms can manage consumption in principals using different policies. Containers is just a mechanism to provide resource management to application developer and so, it can be used with several policies. Resource containers form a hierarchy. Resource usage of a child is constrained by attributes of parent container. This allow easy controlling complete subsystems by defining special policies.

To support containers, new operations are defined [13]. We briefly present the operations because a detailed explanation is not relevant to this work.

- Creating a new container.
- Set a container's parent.
- Container release.
- Sharing containers between processes.
- Access container attributes.
- Access container usage information.
- Binding a thread to a container.
- Binding a socket or a file to a container.

Note that in most middlewares implemented in Java we have a scenario were resource principal is bigger than unit of isolation. The resource principal is the whole Java instance[3] and the unit of isolation is classloader. Independent activities share the resource principal.

2.2 Real-time programming in Java

Real-time is a term used to describe the desired behavior of some applications. In such applications, there are real-world time requirements. For instances, in a computer game the desired frame rate can be established in 30 frames per second so, the renderer must render a frame in 2 seconds to avoid animation freezing. Time failure occurs if this requirement is not met. The previous example is considered a soft real-time application because a time failure is not considered catastrophic. In the other hand, hard real-time applications have strict time requirements. Special hardware and software is needed to obtain response in very short time.

Designers of real-time applications need deterministic time response to articulate a system with desired properties. Designers of execution environments, like real-time operating systems and so on, devote huge effort to reduce nondeterministic performance effects because by lowering nondeterminism the number of runnable RT applications increase.

The nature of Java language and JVM specification introduce nondeterminism in time-response. Three main sources of nondeterminism are 1) dynamic class loading, 2) just in time compilation and 3) garbage collection as implementation of automatic memory management. Dynamic class loading affects because a class representation is loaded first time the application use the class. Delay in application's execution and time failure can take place if the application load a class while it is responding to a real-time event. Time failure can be considerable even for soft RT application because loading a class may trigger additional dynamic class loading in a recursive way. This unlike effect is worst because most JVM implementations use just in time compilation to speed up execution. Common techniques like adaptive compilation, where hot spot bytecode is recompiled to optimize, may slow down RT application's execution with undesirable effects on time response. Finally, garbage collector play a major role in nondeterminism of Java application. Most garbage-collection technique perform a stop the world step at some point. Time spend collecting depends on many factors like rate of living objects, size of the heap and so on. Anyway, it is impossible to say when a collection will be trigged and how long it will take.

[3] This is not the case if you use something like Multitasking Virtual Machine

The Real-time Specification for Java was created to address some of the limitations of Java that prevent its use as RT execution environments. The RTSJ addresses several problematic areas, including scheduling, memory management, threading, synchronization, time, clocks, and asynchronous event handling.

RTSJ is related to resource reservation because one of its main goal is to guarantee deterministic CPU time devoted to RT applications. Other features of RTSJ like threading system, synchronization, timer and event handling are not directly related to resource reservation so, we will skip it in the section.

Memory management is the major source of nondeterministic response time in Java application. The overhead imposes by garbage collection is high so, only applications with larger scale and loose timing requirements can afford to rely on GC technology. RTSJ addresses this problem with to different approaches. The former is providing two new memory regions to applications and the latter is providing garbage collector with deterministic behavior. It is important to note the concern of these approaches is CPU reservation and have nothing to do with memory reservation.

RTSJ provides two new memory regions to applications in addition to the standard Java heap. New memory areas are an immortal space and scoped spaces. Immortal space is shared between threads and not collection is done over objects in this space. Allocated objects in immortal space live until application's termination so, immortal space is a limited resource. Scope spaces are created with fixed size and destroyed by application at programmer will. All objects in a given scoped area are released at the same time. RTSJ define rules to control the way objects in different areas interact each others. Additional rules define the way objects in scoped space are finalized and when a memory area can be reused as scoped area. All these rules limit the usability of the approach because it enforce changes in Java programming model. Using immortal and scope memory areas is only recommended when not GC pauses are acceptable at all.

The second approach is useful when short GC pauses are acceptable. Core idea is using a collector with the following properties:

Property 1. No single GC pause exceeds some maximum upper bound.

Property 2. GC will consume no more than some percentage of any given time window by controlling the number of pauses during that window.

This means that RTSJ ensures a minimum percentage of CPU time to the mutator over any time interval. For instance, if the JVM implementing RTSJ was configured to ensure 80 % then in 60 seconds at least 48 seconds will be devoted to mutator.

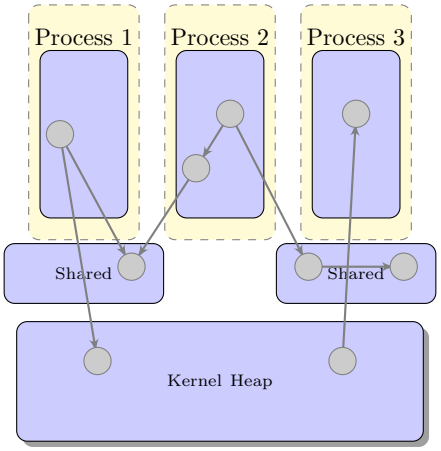
Metronome GC is a method developed as part of WebSphere Real Time [12]. Collection is scheduled at allocation time in most garbage collection methods but this result in nondeterministic long GC pauses. Instead of his, the Metronome GC uses a time-based method of scheduling, which interleaves the collector and the application on a fixed schedule. The approach consists in dividing time in quanta of 500 microseconds. A quanta is devoted to a single activity, the mutator or GC. The execution environment enforce a minimum amount of quanta to mutator. This ensure that Metronome GC conform the property 2. Metronome performs collection during many short GC pauses, in order to do that the work is divided into several parts which require using write barrier and others technique to avoid costly operation like copying [12]. Metronome's implementation conforms property 1 but it has some performance penalties due write barriers.

It is important to note that developer can specify the desired percentage for application execution. However, using a number too high can produce out of memory exception although the rate of living object remains low. The reason is that GC need time to do its job.

Main advantage of using this approach instead of immortal and scoped areas is related to programming model. Any program written in Java can be executed with Metronome GC. Developers need to learn nothing. It is really a big deal considering the way classpath is implemented. The core Java library is implemented with the assumption of GC existence so, lot of objects with short live are created.

RTSJ uses other techniques not related to GC to guarantee deterministic response time. These techniques deal with the problem of just in time compilation. Although it result an interesting topic, it is not related to the current work.

Figure 1: Memory layout in KaffeOS with three processes and some allocated objects



2.3 KaffeOS and MVM

Most common concern related to resource reservation in Java is memory reservation. The reason to this is the memory footprint imposed by automatic memory management. Natural approach to deal with this problem is partitioning memory into different areas and assigning an area per task. We present two related works which follow the approach.

Despite of its name, KaffeOS is Java virtual machine based on Kaffe which provide the concept of process at virtual machine level [10]. KaffeOS offers main abstractions like data protection, safe termination, process forking and inter-process communication. KaffeOS isolates the data of each process by providing a per-application user heap where pointing to a different user heap is forbidden. Figure 1 shows a possible configuration, further details can be found in [10]. Process forking is implemented as a system call. In KaffeOS there are three classes of heap 1) user heap where process's allocations take place, 2) shared heap with fixed size used by many processes and 3) a global kernel heap used to allocate special data structures. Specific rules are enforced through write barriers to ensure protection, safe termination and per-application garbage collection [10].

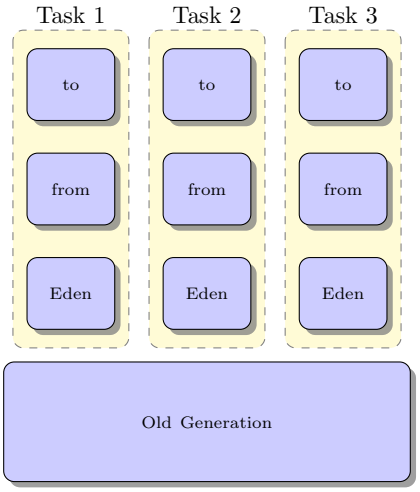
The main goal of KaffeOS is offering isolation between tasks inside JVM. However, we consider it is not the best choice because it enforces a wrong programming model to deal with components. Using the process abstraction inside Java is just skipping chances that any MRE may offer.

Czajkowski et al. present the multitasking virtual machine in [21]. The goal is to provide many Java virtual machine instances within a single operating system process. It decreases memory consumption by sharing common data structures like cache of code, constant pools and so on. Additionally, starting new applications is fast because the initialization process is not needed. A main design decision is to forbid direct object sharing between tasks. The implications of this are 1) the Java programming model is not restricted and 2) the usability in middleware contexts is limited because communications through RMI is required.

In MVM the garbage collector has been modified to guarantee separation of resources between tasks. The collector uses a generational approach where the heap for old generation is shared among tasks. The new generation is represented with three spaces: 1) eden space , 2) from-space and 3) to-space. All three with equal size. New objects are allocated from the eden space and the from-space contains objects which survive some collections but are still young. The role of to-space is the traditional in semi-space collector. Like in any generational collector when an object become old it goes to old space. Figure 2 shows the scheme.

The disadvantage of using MVM to build middleware is the high cost of RMI. By isolating Java instances the communications between components running in different instances is slowed.

Figure 2: MVM with three tasks



2.4 JAMUS

RAJE is an extension to the Java 2 platform implemented in the context of RASC project. The project's aim is supporting high-level resource-aware environments and applications. To accomplish its goals, RAJE provides software components with means to express their hardware/software requirements and to use the information at any stage of components life-cycle [32].

The Java Accommodation of Mobile Untrusted Software (JAMUS) is a research platform dedicated to support the deployment of "untrusted" software components such as application programs and applets. Components can specify their requirements regarding resource utilization in both qualitative and quantitative terms. Qualitative requirements are, for instance, access right to sockets. Quantitative requirements are resource quota. The source of components could vary from remote Internet sites to components received as Email attachments. As a consequence, emphasis is put in JAMUS on providing a safe and guaranteed runtime environment for components, as well as guaranteed QoS as far as resource availability is concerned. By providing monitoring facilities RAJE become a useful platform to support adaptive systems, security-oriented systems and QoS-oriented systems.

JAMUS follow a contractual approach to resource control. At deploy time, software component must specify explicitly what resources it will need at runtime, and in what conditions. As was mentioned, access conditions can be specified in both a qualitative way and a quantitative way. Signing the contract means the candidate component requests a specific service from the JAMUS platform. This contract states the component will use no other resource than those mentioned explicitly. The platform promises to provide this component with all resources it requires but it reserves the right to punish any offensive component. Based on these contracts, JAMUS provides some quality of service regarding resource availability. It also provides components with a relatively safe runtime environment, since no component can access or monopolize resources to the detriment of other components.

JAMUS implements a resource broker. It role is to guarantee the availability of resources for deployed components. The broker receives a description of the available resources at startup. JAMUS provides a series of interfaces and classes that enable the specification of resource access conditions as "resource utilization profiles" [47]. The broker builds a structure which represent its perception of resource availability.

A component that applies for being deployed on the JAMUS platform must first pass an admission control examination. The requirements of this component must be expressed as a set of resource utilization profiles. These requirements are examined by the broker to decide if the component can be admitted on the platform. Admission is based on a simple rule: a component is deployable if the resources it requires are available on the platform in sufficient quality and quantity. Resources a component requires

are reserved for its sole usage until it reaches completion. The resource reservation is performed by updating the broker's perception about resources availability [47].

After passing successfully the admission control test, a component start running on platform. However, once a component has been accepted, it is considered as non-trustworthy. An offensive component may attempt to access resources it did not explicitly ask for. To prevent such problems, every component runs under the control of a dedicated component monitor, whose implementation relies on RAJE. When a monitor observes that the component is acting against the contract, it can punish the component. Sanctioning a component is done by forbidden the resource, or by killing the component.

2.5 Dynamic resource reservation

Simão et al. present in [46] the Adaptative and Resource-Aware Java Virtual Machine, ARA-JVM, a research in progress to deal with resource management for cloud computing using high-level virtual machine. The goal is to build a cloud computing platform with special concern in resource consumption. The solution involves modifying a JVM to deal with nodes, task migration between nodes, transparent communication between nodes and so on. From the perspective of resource-aware platform, the approach is interesting because resource reservation for application is done dynamically. An additional advantage of this approach is that it is totally transparent to application's developer.

ARA-JVM is built upon several runtime instances. Each instance cooperates to the sharing of resources. To implement resource sharing, a global mechanism is needed to make both simple and complex adaptations. A MRE with enhanced services is required at every node. Upon that, a mechanism to aggregate individual VMs is used. It gives running applications support for a single system image [46].

Applications are monitored to obtain precise information about resource consumption and this information is used to build a per application profile. When an application needs more resources, the platform select an application with *low* profile which *donates* part of its resources [46]. It is important to note that moving resource can be explicit or implicit. For instances, by reducing the heap of certain application the memory released can be used by another application. In this case the resource was moved in an implicit way. Of course, the key point of this approach is obtaining a good per application profile.

The current implementation is based on JikesRVM. It is not clear the progress of the whole project but at least resource monitoring has been implemented in the form of JSR 284 - The Resource Management API. In the same way, they use an external language to provide policies to the virtual machine at startup time. Performed experiments are related to heap size handling. The authors implemented policies to keep memory consumption at low level [46].

The research is still a work in progress but there is an obvious lack in the definition of application. Another major problem is lack of good experimenting scenarios. However, the dynamic handling of resources by discovering patterns of usage is an approach highly valuable.

2.6 Discussion

3 Resource accounting in Java

In the context of today middleware systems, resources limitation is an important concern because it is easy to develop offensive applications which affect QoS. Resource-aware programming is about providing application developers with a way to change its behavior at runtime to take care of resource limit violations. The key to support resource-aware programming is resources accounting. Knowledge about the amount of resources consumed by an application can be used by the application to control itself or by an external agent to control offending applications.

Traditionally, resources accounting has been performed at operating system level because this is the normal environment to run applications. Resources accounting is easy to implement inside the operating system. The reason to this is the classical view of operating system as a software layer to abstract, protect and multiplex resources. Moving application's execution to a higher layer pop up a problem related to resources accounting because MREs do not have direct access to real resources. Additionally, MREs offer higher abstraction like type-safe systems. Although type-safe systems do not remove the necessity to application's isolation, it offers chances to lightweight components communications. For instances, it offers the chance to avoid using complex interprocess communication mechanism. However, as some objects can be shared among applications it become hardest to account for some resources.

Java virtual machine specification was built upon the idea that an operating system process is the resource container for a single java application so, you need as many JVM instances as java applications. However, JVM platform has been widely accepted like a tool to middlewares building and the notion of Java instance per Java application was rejected. Main reasons to this are related to language features. The problem with the new approach is the introduction of many QoS risks and even security risks. Isolation is a major concern in Java middlewares but addressing isolation at resources usage level is still a challenge.

Resources accounting at virtual machine level is highly dependent on virtual machine implementation. For instances, thread's model affects the way CPU usage is accounting. In a virtual machine like Kaffe you can apply sampling method [42] but it is a better and even cheaper solution to perform direct accounting. The same problem emerge in memory accounting, memory-management policy you use will affect the accounting subsystem. Even more, the target operating system affect the resources accounting subsystem. For instance, in Unix family the **/proc** pseudo-filesystem provide information about resource usage but the exact location and the amount of information vary among implementations. This particular issue is really important for applications' developers because not all accounting systems offer the same accuracy.

There is not ubiquitous Java virtual machine implementation. Instead, there are many implementations with different aims and providers. However, there is not official specification for resources accounting in Java. This mean developers can rely on specific solutions because lack of portability. Even more, different implementations at virtual machine level can provide disparate accuracy and services. For developers this mean rewriting resources accounting concern for different Java API.

It is not an easy task ensure the quality of accounting. There are three agents running inside any MRE, 1) the mutator, 2) the garbage collector and 3) the supervisor which spend time jitting code and so on. The way this can interfere in accounting depends on implementation details. Some questions need to be answer, for instances:

- Is the garbage collector using a stop the world technique?
- Is the virtual machine just in time compiling the code or interpreting?
- How is implemented the threads' model?

However, the way developers use the virtual machine is even more important because considering that a single JVM instance can handle several user applications has a huge impact on resources accounting. For instance, if an object is shared for two components (a client which created the object and a server which need a reference) it is hard to define which component must carry on with the accounting for such object. It is even hardest to charge a component with the time spend by the garbage collector to release

this object. Additionally, we must charge some application with spent time during garbage collection but the question is to know which application. We can generalize by noting that there are two main reason which make accounting a hard problem, the former is that implementation details can distort the accounting, the latter is that there is not formal definition of resource container or task in Java middleware. Developers started using JVM as platform for running middlewares but JVM specification did not evolve to support resources accounting in the new context.

In this chapter we present some methods to perform resources accounting in Java. There are two main approaches to resources accounting in Java 1) at user level and 2) at virtual machine level. Both approaches will be discussed in the following sections.

3.1 Portable solution by bytecode rewriting

A well known technique to resource accounting in Java is rewriting the bytecode of applications. This process can be done in three different moments.

1. At compilation time.
2. In a post-compilation stage.
3. At runtime.

By rewriting the bytecode at runtime, while the application is loading classes, an additional benefit is obtained because third party code could be used. In this approach at least one new Classloader must be defined in order to instrument the code. The Classloader injects code to estimate resources usage. The function of the injected code is to increment counters of resource usage. As it is impossible to know the exact execution path of a given method the solution must rely on control-flow graph. The classloader injects code at begin (or end) of every basic block in order to increment the counters. For some methods it is possible to infer statically the number of instructions to execute or the amount of memory to use but in general this kind of statical analysis is not available due to the complexity of algorithms. The range of operations that can be inserted vary from a simple counter update to calls to complex APIs. However, the concept is the same.

A function to calculate n-th Fibonacci number is shown in listing 1. The associated control-flow graph and basic blocks is shown in figure 3. For every basic block the instructions to be injected are shown. A similar process is used to account the memory consumption.

Advantage of this approach is portability because the entire solution can be implemented in Java as part of a middleware. In place of the simple statement we are using a more complex code can be injected in order to implement an observer pattern. The observer can execute actions to stop the abnormal resource's consumption [23].

Disadvantage of bytecode rewriting is related to the high performance penalty imposed to applications. The performance penalty emerge in two dimensions, the former is CPU consumption because of the new instructions to execute and the latter is memory usage. Czajkowski et al. showed in [23] that the overhead can be higher than 15% if the amount of allocated objects is high but this is only for memory usage accounting. Binder at al. in [15] found an overhead around 25% for CPU usage accounting. According to Hulaas et al. in [36] an overhead of 40% emerged when an approach based on bytecode rewriting is applied to SPEC JVM98.

Listing 1: Finding the n-th Fibonacci number

```
int Fibonacci(n)
        if (n < 2) return 1
        fi = 0
        fj = 1
        index = 2
        while (index <= n)
                tmp = fi + fj
```

```
                fi = fj
                fj = tmp
                index ++
        return tmp
```

A technique to profile Java Application is presented in [24]. The basic approach is using bytecode rewriting to monitor resources usage. This work states that bytecode instrumentation impose high overhead over running systems so, a better way to handle monitoring is needed. In this approach, instrumentation's code can be inserted or removed at runtime. The approach is named dynamic bytecode instrumentation. Data collected in this way are presented in calling context tree format [9]. The mechanism cannot be implemented without virtual machine support because semantic of Java virtual machine does not allow it. The author modified Hotspot Java VM to perform bytecode rewriting under demand, acting like a server. Most work inside JVM involved tunning the well-known mechanism of swapping between jitted code and bytecode interpretation.

The solution is composed by two components, the former is a JVM which acts like a server and the latter is a client which instrument the code. A brief explanation is given below:

1. Client sends information about what is the root method to instrument.
2. Server obtains call subgraph and send this information to client.
3. Client rewrites bytecode in these methods and send back bytecode to server.
4. Server swaps the current running bytecode by the instrumented one.

Advantages of this approach are 1) lower overhead due dynamic bytecode rewriting and 2) dynamic discovering of call subgraph presenting results as resources consumed by a particular functional task.

Although the original intend is profiling Java application one can think in using the mechanism to monitor resources usage. A lightweight monitoring technique can be used and under certain conditions dynamic bytecode instrumentation can be triggered for a suspicious task to obtain a fine-grained measure. It is easy to rollback the modification at any point to obtain performance advantages.

Bytecode rewriting is useful to account resources in any entity we consider an independent task. We are not restricted by classloaders. We can even consider a thread is multiplexed between many independent task. By knowing the API used to multiplex the thread we can generate the proper instrumentation code.

3.2 Solutions at virtual machine level

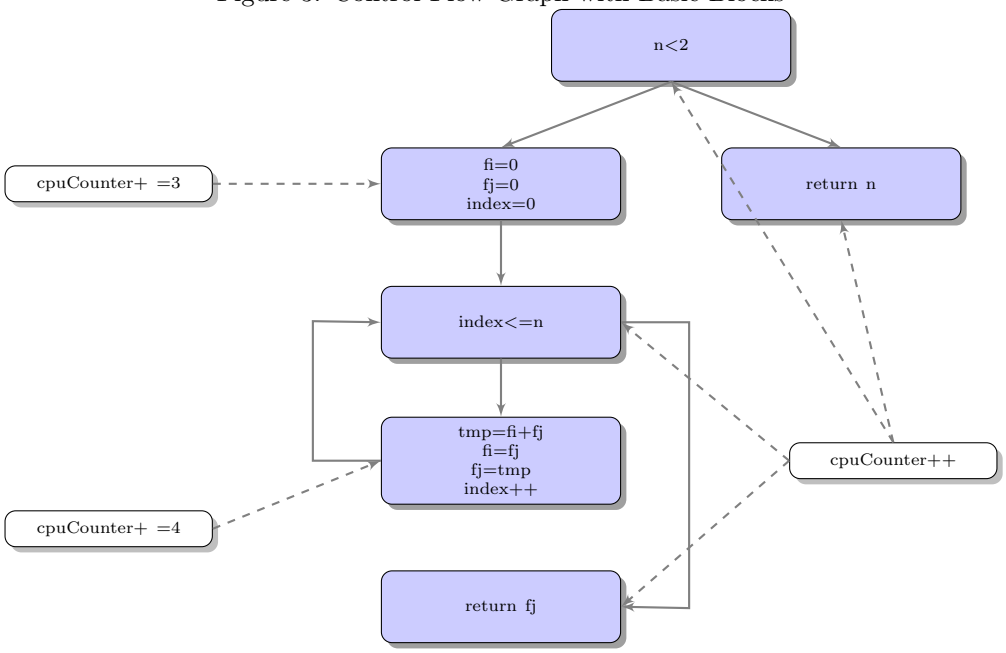
A different approach is making changes at virtual machine level to account for resources usage. The exact nature of this method depend on virtual machine specification and implementation. For instance, some virtual machines have a specific way to deal with threads so, resources accounting is more or less hardest to implement. In some published results the programming model is not preserve and this fact changes the way resources are handled. As there are many considerations, it is hard to generalize. Instead of that we prefer to present related works which highlight trends in this field.

Resource accounting in KaffeOS

In KaffeOS two resources are monitored, the former is memory consumption and the latter is CPU usage. Resources accounting is implemented inside the Java virtual machine as part of the processes handling policy [10].

Memory accounting is easy to implement because every process has its own user heap so, the associates allocator and collector can account memory in every allocation and collection cycle. In fact, the amount of memory used is the sum of objects' size in user heap. The memory used in shared heaps is charged to all processes pointing to the heap. Finally, the kernel has been carefully implemented to keep the number

Figure 3: Control-Flow Graph with Basic Blocks



of kernel objects low and many of them are allocates inside the user heap, for instance, the process data structure is allocate inside process's heap [10].

As section 1.1.3 explains, Kaffe threading model is implemented in user space so, CPU accounting is straightforward. Accounting is done per-process and both the user time and kernel time is measured. This is easy to implement because the separation between user heaps and kernel heaps and their collectors allows accounting CPU in kernel mode. The accuracy of CPU accounting is increased by minimizing the time spend in no-preemptible sections [10].

Modifying the garbage collector

A different approach to memory accounting is modifying the garbage collector. Price et al. [43] presents a solution where objects are no charged to its allocator but to any task having a reference. In their approach a single heap is used for all task and objects sharing semantic is respected. The main modification is in the tracing algorithm. For every task a set of roots is created. This set contains static fields of classes and stack maps of thread. A tracing with this set of roots is performed and memory consumption for the task is the sum of reachable objects. After repeating this procedure for every task, a final tracing is executed to find uncountable references. Due the nature of the modification, the task's order matter so, a shared object could be charged to different tasks. To avoid this problem the authors introduce uncountable references which stop the tracing algorithm and are charged to the allocator task.

As the tracing stage varies between collector this methodology must be adapted. A particular challenge which need further modification are generational collectors. The technique is no applicable to conservative collectors and reference counting collectors.

Finally, the overhead imposed for this technique is around 2 % for experiments executed. However, we consider that more realistic benchmark can be applied. The main advantage of the approach is that Java programming model can be applied. A limitation is using classloaders as definition of task.

Multitasking Virtual Machine

Memory accounting is easy because only the objects in old space need to be dynamically accounted. The memory consumption of any task is the sum of size of eden space, from-space and to-space because this

amount of memory is reserved to the task. The size of every objects allocated for the task which reside in old space is accounting too. Accounting for consumption in old space is performed in two moments, 1) when a collection in new space move an object from young generation to old generation and 2) when MVM collects old generation.

The overhead of this solution to memory accounting is negligible. Of course this is a direct consequence of design decision in memory layout. However, it is important to note that each isolate in MVM is a single JVM instances. We will have the same problem of lack of resource accounting if we execute a middleware per isolate. Another major problem in MVM is its lack of support for CPU usage accounting.

Resource accounting to support resource-aware programming

RAJE is an extension to the Java 2 platform implemented in the context of RASC project. The project's aim is to provide software components with means to express their hardware/software requirements and to use the information at any stage of components life-cycle [32]. The solution considers an application is mapped to a classloader. A resource register is associated with each application and some classes like Memory, CPU, Thread and so on are defined to represent resources. The standard API for Java threads was extended and an application can obtain the amount of resources used by calling some methods. Two models of resources monitoring are used, 1) synchronous monitoring and 2) asynchronous monitoring. The former allows a fine grain resource accounting but it is to expensive and it is not available for all resources. The latter allows accounting resources like CPU usage but it is less accurate.

RAJE was implemented on top of Kaffe for Linux operating system. Some changes were necessary to easy capture CPU usage. Threads subsystem was ported to use native threads so, access to **/proc** pseudo file-system allows obtaining statistics. Memory accounting is done per-thread by intercepting allocations and collections at virtual machine level.

Although the solution is not portable it offers to applications the ability of observe the resource consumption because every resource object could have any number of attached listeners. Resource objects expose a locking interface with the following semantic: locked resources cannot be used. This mechanism is enough to implement several resource's management policy.

3.3 Resource-aware programming interface

Any resource accounting mechanism must provide an API to allow the platform as well as te application to monitor resource usage. In early solutions [23, 22], the center idea was using observer pattern [28]. In this case, the resource principal assumes the role of *subject* and monitors act as *observers*. Modern solutions rely on Java Management Extensions. In additions to common extensions to control coarse-grained resources like threads, GC cycles and total memory consumption; you can add your own agents to monitor resource consumption to wherever level you want.

In the other side of resource-aware programming, the side related to providing mechanism to adapt the application behavior, there are few options. The most common is provided for the Java runtime itself and it consist in stopping offensive threads or lowering its priority. There are other options which allow using capabilities [33] to revoke permissions. Although there are other solutions [32], more general adaptation policies are less common and remains a challenges.

3.4 Discussion

References

[1] The kaffe virtual machine.

[2] The llvm compiler infrastructure.

[3] Edward Aftandilian and Samuel Z. Guyer. Gc assertions: using the garbage collector to check heap properties. In *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, MSPC '08, pages 36–40, New York, NY, USA, 2008. ACM.

[4] OSGi Alliance. Osgi service platform, core specification release 5.0, June 2012.

[5] Alejandro Alonso, Maria-Paola Bianconi, Nicolas Francois, Giovanni Cortese, and Erik Yu. Flexible java real-time profile for business-critical systems. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, JTRES '06, pages 135–143, New York, NY, USA, 2006. ACM.

[6] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeño virtual machine. *IBM Syst. J.*, 39(1):211–238, January 2000.

[7] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing jalapeño in java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, pages 314–324, New York, NY, USA, 1999. ACM.

[8] Bowen Alpern, Maria Butrico, Anthony Cocchi, Julian Dolby, Stephen J. Fink, David Grove, and Ton Ngo. Experiences porting the jikes rvm to linux/ia32. In *Proceedings of the 2nd Java™ Virtual Machine Research and Technology Symposium*, pages 51–64, Berkeley, CA, USA, 2002. USENIX Association.

[9] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Not.*, 32(5):85–96, May 1997.

[10] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. Processes in KaffeOS: isolation, resource management, and sharing in java. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 23–23, Berkeley, CA, USA, 2000. USENIX Association.

[11] David F. Bacon, Perry Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for java. *SIGPLAN Not.*, 38(7):81–92, June 2003.

[12] David F. Bacon, Perry Cheng, and V.T. Rajan. The metronome: A simpler approach to garbage collection in real-time systems. In *In Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops*, pages 466–478, 2003.

[13] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pages 45–58, Berkeley, CA, USA, 1999. USENIX Association.

[14] Yungang Bao, Mingyu Chen, Yuan Ruan, Li Liu, Jianping Fan, Qingbo Yuan, Bo Song, and Jianwei Xu. Hmtt: a platform independent full-system memory trace monitoring system. *SIGMETRICS Perform. Eval. Rev.*, 36(1):229–240, June 2008.

[15] Walter Binder, Jane G. Hulaas, and Alex Villazon. Portable resource control in java. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 139–155, New York, NY, USA, 2001. ACM.

[16] Stephen M Blackburn. Myths and realities: The performance impact of garbage collection. In *In Proceedings of the ACM Conference on Measurement and Modeling Computer Systems*, pages 25–36. ACM Press, 2004.

[17] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? high performance garbage collection in java with mmtk. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 137–146, Washington, DC, USA, 2004. IEEE Computer Society.

[18] Stephen M. Blackburn and Kathryn S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. *SIGPLAN Not.*, 43(6):22–32, June 2008.

[19] G. Blair, N. Bencomo, and R.B. France. Models@ run.time. *Computer*, 42(10):22 –27, oct. 2009.

[20] Rhodes H. F. Brown and R. Nigel Horspool. Local redundant polymorphism query elimination. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 78–88, New York, NY, USA, 2010. ACM.

[21] Grzegorz Czajkowski and Laurent DaynÃ©s. Multitasking without comprimise: a virtual machine evolution. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 125–138, New York, NY, USA, 2001. ACM.

[22] Grzegorz Czajkowski, Stephen Hahn, Glenn Skinner, Pete Soper, and CiarÃ¡n Bryce. A resource management interface for the java™ platform. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2003.

[23] Grzegorz Czajkowski and Thorsten von Eicken. JRes: a resource accounting interface for java. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 21–35, New York, NY, USA, 1998. ACM.

[24] Mikhail Dmitriev. Profiling java applications using code hotswapping and dynamic call graph revelation. *SIGSOFT Softw. Eng. Notes*, 29(1):139–150, January 2004.

[25] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[26] Francois Fouquet, Brice Morin, Franck Fleurey, Olivier Barais, Noel Plouzeau, and Jean-Marc Jezequel. A dynamic component model for cyber physical systems. In *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, CBSE '12, pages 135–144, New York, NY, USA, 2012. ACM.

[27] 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 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 81–90, New York, NY, USA, 2009. ACM.

[28] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[29] N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frenot, and B. Folliot. I-JVM: a java virtual machine for component isolation in OSGi. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pages 544 –553, July 2009.

[30] Nicolas Geoffray, Gaël Thomas, Charles Clément, and Bertil Folliot. A lazy developer approach: building a jvm with third party software. In *Proceedings of the 6th international symposium on Principles and practice of programming in Java*, PPPJ '08, pages 73–82, New York, NY, USA, 2008. ACM.

[31] Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller, and Bertil Folliot. Vmkit: a substrate for managed runtime environments. *SIGPLAN Not.*, 45(7):51–62, March 2010.

[32] Frédéric Guidec and Nicolas Le Sommer. Towards Resource Consumption Accounting and Control in Java: a Practical Experience. In *Proceedings of the Workshop on Resource Management for Safe Language*, page 6 pages, Màlaga, Spain, June 2002.

[33] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in java. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '98, pages 22–22, Berkeley, CA, USA, 1998. USENIX Association.

[34] M. Teresa Higuera-Toledano. About 15 years of real-time java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 34–43, New York, NY, USA, 2012. ACM.

[35] Kenneth Hoste, Andy Georges, and Lieven Eeckhout. Automated just-in-time compiler tuning. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 62–72, New York, NY, USA, 2010. ACM.

[36] Jarle Hulaas and Walter Binder. Program transformations for portable cpu accounting and control in java. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '04, pages 169–177, New York, NY, USA, 2004. ACM.

[37] Christoph M. Kirsch, Marco A. A. Sanvido, and Thomas A. Henzinger. A programmable microkernel for real-time systems. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 35–45, New York, NY, USA, 2005. ACM.

[38] John C. Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 547–550, New York, NY, USA, 2002. ACM.

[39] 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: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[40] Ralf Mitschke, Andreas Sewe, and Mira Mezini. Magic for the masses: safer high-level low-level programming through customizable static analyses. In *Proceedings of the 1st workshop on Modularity in systems software*, MISS '11, pages 13–17, New York, NY, USA, 2011. ACM.

[41] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jezequel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 122–132, Washington, DC, USA, 2009. IEEE Computer Society.

[42] Priya Nagpurkar, Hussam Mousa, Chandra Krintz, and Timothy Sherwood. Efficient remote profiling for resource-constrained devices. *ACM Trans. Archit. Code Optim.*, 3(1):35–66, March 2006.

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

[44] Ian Rogers, Jisheng Zhao, and Ian Watson. Boot image layout for jikes rvm. In *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*, July 2008.

[45] Rifat Shahriyar, Stephen M. Blackburn, and Daniel Frampton. Down for the count? getting reference counting back in the ring. In *Proceedings of the 2012 international symposium on Memory Management*, ISMM '12, pages 73–84, New York, NY, USA, 2012. ACM.

[46] Jose Simão and Luis Veiga. Vm economics for java cloud computing: An adaptive and resource-aware java runtime with quality-of-execution. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, CCGRID '12, pages 723–728, Washington, DC, USA, 2012. IEEE Computer Society.

[47] Nicolas Le Sommer and FrÃ©dÃ©ric Guidec. JAMUS: Java Accommodation of Mobile Untrusted Software. 2002.

[48] Gaël Thomas, Nicolas Geoffray, Charles Clément, and Bertil Folliot. Designing highly flexible virtual machines: the jnjvm experience. *Softw. Pract. Exper.*, 38(15):1643–1675, December 2008.

[49] Kai Tian, Yunlian Jiang, Eddy Z. Zhang, and Xipeng Shen. An input-centric paradigm for program dynamic optimizations. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 125–139, New York, NY, USA, 2010. ACM.

[50] Frank Yellin Tim Lindholm. *The Java^{TM} Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., 1999.

[51] Guoqing Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. Leakchaser: helping programmers narrow down causes of memory leaks. *SIGPLAN Not.*, 46(6):270–282, June 2011.

[52] Wojciech Zaremba, Yuan Lin, and Vinod Grover. Jabee: framework for object-oriented java bytecode compilation and execution on graphics processor units. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 74–83, New York, NY, USA, 2012. ACM.

[53] Khawar M. Zuberi, Padmanabhan Pillai, and Kang G. Shin. Emeralds: a small-memory real-time microkernel. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, SOSP '99, pages 277–299, New York, NY, USA, 1999. ACM.