SAP VM Container: Using Process Attachable Virtual Machines to Provide Isolation and Scalability for Large Servers

Norbert Kuck, Harald Kuck, Edgar Lott, Christoph Rohland, Oliver Schmidt
SAP AG
Walldorf, Germany
{norbert.kuck, hq, edgar.lott, cr, oschmidt}@sap.com

The SAP VM Container is an application server framework that uses a new paradigm of *Process Attachable Virtual Machines* (PAVMs) to combine high scalability with strong isolation between user sessions. In this short paper, we present the design and implementation of the PAVM paradigm for Java virtual machines.

1 Introduction

When it comes to mission critical large enterprise server environments, robustness and scalability become main concerns for server virtual machines.

From a most general point of view, enterprise servers can be characterized as *request processing engines*, serving large numbers of (typically small) user requests that belong to user sessions. The actual request processing involves running user code (e.g. Servlets, EJBs) in a virtual machine. Request throughput is the main objective for scalability, which is traditionally achieved with thread pool based architectures.

System robustness necessarily requires strong isolation between user sessions, something that is often hard to achieve if a large number of user sessions are run within a single virtual machine.

On the other hand, operating systems provide near perfect isolation for processes - even a crashing process does not affect any other process nor does it leave behind resource leaks. Why not just provide each user session with a virtual machine and an OS process of its own ? The answer is obvious: resource consumption and OS scheduling overhead make this approach infeasible with respect to scalability. OS processes are just not designed to model entities as fine grained as user sessions.

The SAP VM Container introduces a new paradigm designed to resolve the dilemma between isolation and scalability: Using a dispatcher process and a small pool of work processes, each user session is provided with its own *Process Attachable Virtual Machine* (PAVM): a VM that can be attached to and detached from OS processes at very low cost. User requests are dispatched to work processes, which attach the appropriate PAVM and have it process the request.

2 Process Attachable VM

In order to make a VM *process attachable*, the usual affinity between a VM and the OS process it runs in needs to be removed. The PAVM paradigm can be regarded as a variation of the *Orthogonally Persistable VM* paradigm [1]. When a VM is detached from a work process, its computational state is persisted in a way that allows another work process to unpersist it at extremely low cost. The key to achieve this is to use shared memory that is accessible to all work processes to store persistent VM states.

When a user session starts, a PAVM is created with a private shared memory block, where it keeps its full computational state. The PAVM's heap and stacks are allocated directly from this *session memory*. That makes the transition to and from persistent state essentially a no-op for the PAVM's memory - the work process just has to map the *session memory* into its address space. No data is actually moved or copied.

I/O resources (file handles, sockets) have to be explicitly persisted. This is achieved by introducing an additional level of indirection - what the VM sees as a file or socket descriptor is actually just a handle that is either persistable by itself (for plain files) or employs the services of a central resource manager (for sockets and pipes). Descriptor passing provides for an efficient implementation. The resource manager notifies the dispatcher once the I/O request is completed.

Threads and monitors in user code deserve some special attention. Native OS threads cannot easily be persisted to shared memory, so PAVMs can only provide "green threads" functionality to user code. All data structures related to thread management and scheduling (including the threads' call stacks, mutexes and condition variables for Java monitors) are kept directly in *session memory*. This includes both the Java stacks and the C stacks used by the VM implementation, e.g. for the JNI implementation of dynamic method invocation. OS coroutines (see [2] for Unix) are employed to control the location of C stack memory.

Thread scheduling is coroutine-based and thus non-preemptive, which seems a drawback at first glance. But it turns out that for the special case of a *request processing engine* as described above, preemptive scheduling is actually less desirable than one might expect. In order to maximize request throughput (as opposed to fairness), a batch processing strategy is employed instead: within each PAVM, threads yield cooperatively to the thread scheduler when entering wait state (i.e. blocking on I/O or a Java monitor)[1].

The coordination between blocking I/O calls and the thread scheduler is part of the I/O redirection mecha-

[1] Very coarse preemption (implemented with timers and signals) is additionally used to prevent user sessions from monopolizing a work process by hogging the CPU.

nism described above. Mutexes and condition variables are implemented without using OS locking primitives as simple scheduler-controlled variables in *session memory*.

Thread scheduling continues for a PAVM until all its threads have entered wait state, indicating that the user request is either completed or waiting for I/O. In either case, the PAVM can be detached from the work process. The dispatcher will reattach it to some work process when the next user request comes in resp. when the I/O request is completed.

3 Benefits

The PAVM paradigm essentially combines the advantages of threads and processes, while omitting many of their respective drawbacks. The process pool ensures good CPU usage for SMP systems. Green threading within user sessions provides for lightweight scheduling and gives full control over batch/fairness behavior. Contention is limited to user session scope, even for system activities like garbage collection. Full isolation and memory protection is granted to user sessions (a work process only maps the *session memory* for a single PAVM at a time). Even a crashed work process can easily be restarted, only invalidating its current user session. Standard debugging and profiling tools can easily be used on a per-session basis.

4 Limitations

The PAVM paradigm relies on assumptions that are specific to the *request processing engine* model outlined above. It is not suited for a general purpose Java VM. Even for some typical Java server scenarios (e.g. non-distributable Servlets), strong session isolation is not compatible with application semantics.

JNI code can only be run within a PAVM if conforming to a fairly complex set of rules regarding memory usage and I/O.

5 Optimizations and related work

The JNI restriction can be lifted to allow for arbitrary JNI user code using the redirection techniques introduced by Czajkowski et al. [3].

The considerable overhead for class loading, verification and resolution for each PAVM can be reduced by keeping type information (i.e. the runtime representation of all loaded classes) in shared *system memory* common to all PAVMs. This technique is expected to be introduced with Sun's Hotspot 1.4.1 VM [4] and is applicable to PAVM as well.

Even with optimized class loading, PAVM initialization is still fairly expensive (if only for the numerous static initializers in system classes). This can be avoided by initializing each PAVM's *session memory* from a suitable "template" image, which is easily obtained by copying the *session memory* of a freshly initialized "master" PAVM at system startup.

This approach can be taken even further. Following the work of Dillenberger et al. [5], PAVMs could be implemented to be *serially reusable* after the corresponding user session ends. This is particularly effective for very short user sessions or stateless user requests.

An API for Java isolation is currently being defined by JSR 121 [6]. We expect that the PAVM paradim can assist in the implementation of JSR 121 for server environments.

6 Conclusion and work in progress

We have presented a short overview of the PAVM paradigm within the SAP VM Container, its benefits and restrictions and outlined possible optimizations with respect to related works.

The PAVM paradigm is currently being implemented for Java VMs to be part of the SAP Web Application Server, where the SAP VM Container technology has been used quite successfully for more than 10 years as the foundation for SAP's R/3 system, an application server running a comprehensive suite of business applications by means of a SAP specific programming language and virtual machine named ABAP (*Advanced Business Application Programming*).

Sun's CVM codebase from J2ME is used as the starting point for SAP's PAVM implementation. A working prototype running Servlets and JSPs demonstrates feasibility (5/2002), but does not yet allow any performance measurements and still lacks the possible optimizations outlined above.

References

[1] M. Jordan, M. Atkinson: *Orthogonal Persistence for the Java Platform: Draft specification.* http://www.sun.com/research/forest/index.html, October 1999

[2] The Open Group: *ucontext manual page*. In: Single Unix Specification, Version 2. http://www.opengroup.org/onlinepubs/007908799/xsh/ucontext.h.html

[3] G. Czajkowski, L. Daynès, M. Wolczko: *Automated and Portable Native Code Isolation.* Sun Microsystems Laboratories Technical Report 01-96, April 2001

[4] G. Hamilton: *The Java 2 Platform, Standard Edition (J2SE) 1.4 Release, and Beyond*. Java One presentation, April 2002

[5] D. Dillenberger et al: *Building a Java virtual machine for server applications: The Jvm on OS/390.* IBM Systems Journal Vol 39 No 1, 2000

[6] JSR 121 expert group: *JSR 121: Application Isolation API Specification.* http://www.jcp.org/jsr/detail/121.jsp