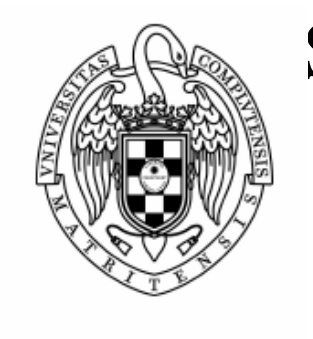


Sistemas Informáticos
Curso 2004-2005

Máquina virtual de Java Distribuida

Rafael Guijarro Crespo
Pablo Molina García
Luis Jarabo de la Llana

Dirigido por:
Prof: Katzalin Olcoz Herrero
Dpto: ACYA

Facultad de Informática
Universidad Complutense de Madrid

Índice de Contenido

0. Resumen del proyecto. 4
Summary in english. 4
Resumen en castellano. 4
1. Introducción. 5
2. Introducción a JikesRVM y dJVM. 9
2.1. JikesRVM 9
2.1.1. Visión general de la estructura de JikesRVM. 10
2.1.1.1. Componentes. 10
2.1.1.2. Estructura de paquetes. 11
2.1.2. Evaluación del rendimiento. 11
2.2 dJVM 13
2.2.1. Vista previa de la estructura de Jikes Distributed Java Virtual Machine 13
3. Estructura del código fuente. 14
3.1. Introducción. 14
3.2. Código de dJVM cluster. 14
3.2.1. Directorio de comunicaciones. 14
3.3. Código de carga de clases de la VM y dJVM. 16
3.3.1. Una visión general de la carga de clases distribuida. 16
3.3.2. Replicación de definiciones de clase en etapas. 17
3.3.3. Detalles de la implementación de la replicación. 18
3.3.4. Sumas de control para la verificación de consistencia entre JVMs. 20
3.4. Código de VM y dJVM relacionado con la transformación de código. 20
3.4.1. Invocación de métodos 20
3.4.2. Acceso a datos. 21
3.4.3. Operaciones de supervisión. 22
3.5. Código del compilador base de la VM 23
3.5.1. Invocación de métodos 23
3.6.2. Acceso a datos. 23
3.6.3. Métodos magic. 24
3.6. Código de VM y dJVM relacionado con la gestión de memoria. 24
3.6.1. Asignación de objetos. 24
3.6.2. Recolección de basura. 26
3.7. Código de VM y dJVM relacionado con planificación y otros aspectos de hilos. 29
3.7.1. Planificación 29
3.7.2. Terminación de programa. 30
3.7.3. Cerrojos. 30
3.8. Código de dJVM relativo a Utilidades. 31
4. Uso. 33
4.1. Introducción. 33
4.2. Ejecutando dJVM. 33
4.3. Ejemplo de ejecución de rvmCluster. 35
4.4. Opciones de dJVM. 36
4.4.1. Comunicación. 37
4.4.2. Asignación. 37
4.4.3. Escritura de información de depuración en ficheros locales 38
4.4.4. Transformaciones. 38

4.4.5. Acceso. 38
4.4.6. Compilación. 38
4.4.7. Inicialización. 38
4.5. Descripción de los programas de prueba y de las aplicaciones. 39
4.5.1. Probando dJVM bajo carga. 39
4.5.2. Aplicaciones. 41
4.5.3. Programas de prueba de acceso a datos. 41
4.5.4. Miscelánea de programas de pruebas. 41
5. Benchmarks. 43
5.1. Benchmark sin carga. 45
5.2.1. Benchmarks con carga y mucho uso de memoria. 47
5.2.2. Benchmarks con carga y poco uso de memoria. 49
Apéndices 52
A. Ejecutar trabajos en el cluster del DACYA. 52
A.1. Introducción. 52
A.2. Propiedades de los nodos. 52
A.3. Colas. 53
A.4. Enviando trabajos. 53
A.4.1. El script resolver.sh. 54
A.4.2. Trabajos interactivos. 55
A.5. Monitorizar el estado de los trabajos. 55
A.6. Terminación de trabajos. 56
A.7. Documentación sobre PBS. 56
B. Instalación de Jikes Distributed Java Virtual Machine. 57
B.1. Requerimientos de sistema y de utilidades. 57
B.2. Construcción de Jikes Distributed Java Virtual Machine. 57
B.2.1. Construir la estructura de directorios. 58
B.2.2. Construyendo el GNU Claspath Library. 58
B.2.3. Construyendo dJVM. 58
Bibliografía 59
Palabras clave. 60
Concesión de derechos. 61

0. Resumen del proyecto.

Summary in english.

Java is becoming increasingly important in implementing server applications. As many of these applications are multi-threaded with limited interaction between threads, a distributed JVM on a cluster platform may provide a cost-effective and high performance solution.

A promising approach, would be to base the distributed JVM on an existing JVM using dynamic translation of bytecodes into machine codes for high performance. Furthermore, organizing the distribution of objects using a specialized scheme would be better able to take advantage of Java's semantics than a general Distributed Shared Memory implementation. Jikes RVM (Research Virtual Machine) provides a flexible open testbed to prototype virtual machine technologies and experiment with a large variety of design alternatives.JikesRVM advances the art of virtual machine technologies for dynamic compilation, adaptive optimization, garbage collection, thread scheduling, and synchronization. A distinguishing characteristic of Jikes RVM is that it is implemented in the Java programming language and is self-hosted i.e., its Java code runs on itself without requiring a second virtual machine. Most other virtual machines for the Java platform are written in native code (typically, C or C++). A Java implementation provides ease of portability, and a seamless integration of virtual machine and application resources such as objects, threads, and operating-system interfaces.

DJVM (Distributed Java Virtual Machine) is a distributed Java virtual machine which extend the JikesRVM to have a distributed implementation, over nodes connected by TCP/IP. This machine allow the implementation, evaluation and analysis of distribution techniques and distributed runtime support algorithms for multi-threaded server applications. So the objetive of this project is to implement the distributed virtual machine dJVM developed by the Australian National University over the cluster placed on DACYA (computer arquitecture and automatic department) at UCM, do a deep study of its structure, check the performance and memory management.

Resumen en castellano.

El incipiente crecimiento de Java ha hecho que crezca su importancia en la implementación de aplicaciones para servidores. Puesto que muchas de estas aplicaciones están caracterizadas por ser multi-hilo y la comunicación entre dichos hilos no es excesiva, una distribución de una máquina virtual Java sobre un cluster, en principio podría ser una herramienta interesante puesto que elevaría el rendimiento sin un excesivo coste.

Una buena manera de hacerlo, sería implementar una máquina virtual distribuida que utilice traducción dinámica de bytecode a otros nodos. Además una buena distribución de los objetos mediante un sistema distribuido especializado aprovecharía mejor la semántica de Java que una versión distribuida con memoria compartida.

Jikes RVM (Jikes Research Virtual Machine) proporciona un campo de pruebas flexible para distintas tecnologías sobre máquinas virtuales y experimentar con una gran cantidad de alternativas de diseño. Adelanta tecnologías de compilación dinámica, optimización adaptativa, recolección de basura, manejo y sincronización de hilos de hilos, en máquinas virtuales. Una característica que distingue a Jikes sobre el resto de máquinas virtuales es que está completamente implementada en Java y es autosuficiente; es decir, su código Java corre bajo sí misma sin la necesidad de una segunda máquina virtual. La mayoría de las máquinas virtuales existentes están escritas en código nativo (típicamente C o C++). Una implementación en Java proporciona facilidad de portabilidad, e integración entre la máquina virtual y recursos de las distintas aplicaciones como puedan ser objetos, hilos y interfaces de sistemas operativos.

DJVM (Distributed Java Virtual Machine) es una máquina virtual Java distribuida que se sitúa sobre JikesRVM para obtener una implementación distribuida basada en nodos conectados mediante TCP/IP. Esta máquina permite la implementación, evaluación y análisis de las distintas técnicas de distribución y soporte de algoritmos para aplicaciones multihilo destinadas a servidores. Así pues el objetivo de este proyecto es la implementación del cluster desarrollado por la Universidad Nacional Australiana sobre el cluster situado en el DACYA de la UCM así como profundo estudio de su estructura, la evaluación de su rendimiento y su manejo de memoria.

1. Introducción.

El entorno de negocio de hoy en día demanda aplicaciones Web y de comercio electrónico que aceleren la entrada en nuevos mercados, ayude a encontrar nuevas formas de llegar y de retener clientes, y permita presentar rápidamente productos y servicios. Para construir y desplegar estas nuevas soluciones, necesitamos plataformas de comercio electrónico que pueda conectar y potenciar a todos los tipos de usuario mientras integra los datos corporativos, las aplicaciones mainframe, y otras aplicaciones empresariales en una solución de comercio electrónico fin-a-fin poderosa y flexible. Cualquier solución debe proporcionar el rendimiento, la escalabilidad, y la alta disponibilidad necesaria para manejar los cálculos de empresa más críticos.

Para crear dichas soluciones, muchas empresas del sector, como por ejemplo BEA Systems, IBM o iPlanet se decantan por usar Java. Por ejemplo, el servidor de aplicaciones WebLogic Server de BEA Systems utiliza tecnologías de la plataforma Java 2, Enterprise Edition (J2EE). J2EE es la plataforma estándar para desarrollar aplicaciones multicapa basadas en el lenguaje de programación Java.

Debido a que la capacidad de procesamiento que tiene que cubrir un servidor es muy grande, cada día tenemos que recurrir a servidores cada vez con mayor capacidad de procesamiento. Para ello, típicamente se han adoptado dos posturas totalmente diferentes:

- Aumentar la capacidad de procesamiento mediante supercomputadores: Consiste en usar cada vez computadores con una mayor capacidad de procesamiento. La escalabilidad de este tipo de técnicas es muy limitada, ya que implica que debemos desechar la máquina que ya tenemos, y por tanto es una solución muy cara.
- Replicación de los elementos de proceso, como en el caso de los clusters: Consiste en disponer de más elementos de proceso y hacer que diferentes tareas se ejecuten de forma paralela entre los elementos de proceso. Esta es una solución más barata que la anterior, pues la inversión en una máquina anterior no se pierde.

La tendencia actual de las grandes empresas para aumentar la capacidad de procesamiento de sus servidores sigue una mezcla de estas dos líneas; utiliza procesadores de última generación (con varios hilos de ejecución, etc…), y a la vez, utiliza una replicación de dichos elementos de proceso para obtener un mayor rendimiento mediante la ejecución paralela de trabajos. La siguiente tabla muestra como una gran empresa como Sun Microsystems Inc. sigue esta tendencia en sus servidores tanto de gama media como de gama alta:

Servidores gama media:

Tipo de Servidor	Descripción	Características
Servidor Sun Fire V490	El servidor Sun Fire V490 constituye la implementación de más bajo coste de la tecnología de Sun para centros de datos corporativos. El servidor Sun Fire V490 ocupa sólo 5 unidades de rack y alberga hasta 4 procesadores UltraSPARC IV con tecnología de subprocesos múltiples (multithreading) en chip que permiten hasta ocho procesos simultáneos y 32 GB de memoria. El sistema operativo Solaris, el sistema operativo UNIX líder del mercado, libera la potencia de estos servidores multiprocesador con subprocesos múltiples para conseguir unos niveles de rendimiento	Hasta 4 procesadores UltraSPARC IV Hasta 32 GB de memoria 2 unidades de disco FC-AL Formato compacto de 5U

	en las exigentes aplicaciones empresariales y departamentales. El servidor Sun Fire V490 es perfecto para asistencia en la toma de decisiones, infraestructura de comercio electrónico y cargas de trabajo centradas en aplicaciones.	
Servidor Sun Fire V890	El servidor Sun Fire V890 cuenta con hasta 8 procesadores UltraSPARC IV de 64 bits con tecnología de subprocesos múltiples en chip (Chip Multithreading) y ejecuta hasta 16 subprocesos de cálculo simultáneos. El servidor Sun Fire V890, con el sistema operativo Solaris, el UNIX más robusto, seguro y popular del mercado, ofrece gran capacidad de cálculo, E/S y almacenamiento en un paquete altamente integrado para una implantación rápida. El servidor Sun Fire V890 destaca en el soporte de un gran número de aplicaciones, incluyendo infraestructura de red y bases de datos, comercio electrónico y cargas de trabajo ERP.	Hasta 8 procesadores UltraSPARC IV Hasta 64 GB de memoria principal Hasta 12 unidades de disco FC-AL Sistema preparado para montaje en rack
Servidor Sun Fire E4900	El servidor Sun Fire E4900 es un servidor de centro de datos de alto rendimiento y gran disponibilidad especialmente adecuado para los entornos de centro de datos de misión crítica. Con capacidad para ampliarlo hasta a 12 procesadores UltraSPARC IV gracias a la tecnología de subprocesos múltiples (multithreading) en el chip que ejecuta hasta 24 subprocesos de cálculos informáticos simultáneos, este servidor es perfecto para grandes bases de datos departamentales, apoyo a la toma de decisiones, gestión de las relaciones con el cliente y aplicaciones técnicas de alto rendimiento que requieren la máxima disponibilidad y un rendimiento óptimo. La alta disponibilidad se obtiene gracias a la redundancia completa de hardware, a los dominios dinámicos del sistema sin fallos y a la reconfiguración dinámica de componentes clave del servidor intercambiables en caliente mientras las aplicaciones están en funcionamiento. Se protege la inversión gracias a la capacidad de mezclar y gestionar procesadores UltraSPARC IV y UltraSPARC III -- funcionando a sus velocidades nominales-- en el mismo sistema. Este servidor se suministra con los servicios remotos preparados para servicio técnico preventivo y con una completa serie de aplicaciones con Java Enterprise System preinstaladas. Este servidor está disponible en el programa Sun Customer Ready Systems (CRS) para la integración en fábrica de soluciones personalizadas y listas para el despliegue.	Hasta 12 procesadores UltraSPARC IV Cu Hasta 96 GB de memoria Hasta 2 dominios sin fallos Hasta 16 tarjetas PCI
Servidor Sun Fire E6900	El servidor Sun Fire E6900 es un servidor de gama media con gran disponibilidad con capacidad de hasta 24 procesadores UltraSPARC IV con tecnología de subprocesos múltiples (multithreading) en chip que permite ejecutar 48 subprocesos de cálculos informáticos simultáneos. Este servidor de centro de datos es perfecto para la consolidación, data warehousing, apoyo en la toma de decisiones y aplicaciones técnicas de alto rendimiento. La alta disponibilidad se obtiene gracias a la redundancia completa de hardware, a los dominios dinámicos del sistema sin fallos y a la reconfiguración dinámica de componentes clave del servidor intercambiables en caliente mientras las aplicaciones están en funcionamiento. Se protege la inversión gracias a la capacidad de mezclar y gestionar procesadores UltraSPARC IV y UltraSPARC III --funcionando a sus	Hasta 24 procesadores UltraSPARC IV Cu Hasta 192 GB de memoria Hasta 4 dominios sin fallos Hasta 32 tarjetas PCI

	velocidades nominales-- en el mismo sistema. Este servidor se suministra con los servicios remotos preparados para servicio técnico preventivo y con una completa serie de aplicaciones con Java Enterprise System preinstaladas. Este servidor está disponible en el programa Sun Customer Ready Systems (CRS) para la integración en fábrica de soluciones personalizadas y listas para el despliegue.	

Servidores de gama alta:

Tipo de Servidor	Descripción	Características
Sun Fire E20K	El servidor Sun Fire E20K es un servidor de centro de datos de alto rendimiento y gama alta con capacidad de hasta 36 procesadores UltraSPARC IV con tecnología de subprocesos múltiples (multithreading) que permite ejecutar 72 subprocesos de cálculos informáticos simultáneos. Este sólido servidor es perfecto para la migración de mainframe, consolidación y aplicaciones técnicas fundamentales de alto rendimiento. La alta disponibilidad se obtiene gracias a la redundancia completa de hardware, a los dominios dinámicos de sistema sin fallos y a la reconfiguración dinámica de componentes clave del servidor intercambiables en caliente mientras las aplicaciones están instaladas y ejecutándose para una mayor disponibilidad. Se protege la inversión gracias a la capacidad de mezclar y gestionar procesadores UltraSPARC IV y UltraSPARC III --funcionando a sus velocidades nominales-- en el mismo sistema. Este servidor se suministra con los servicios remotos preparados para servicio técnico preventivo y con una completa serie de aplicaciones con Java Enterprise System preinstaladas. También se puede actualizar en caliente al servidor Sun Fire E25K, de modo que ofrece una ruta de actualización sin fisuras sin interrumpir el servicio. Este servidor está disponible en el programa Sun Customer Ready Systems (CRS) para la integración en fábrica de soluciones personalizadas y listas para el despliegue.	Hasta 36 procesadores UltraSPARC IV de doble subproceso Hasta 288 GB de memoria por dominio Más de 120 TB de almacenamiento Hasta 9 dominios dinámicos de sistema
Sun Fire E25K	El servidor Sun Fire E25K ofrece un aumento espectacular del rendimiento del sistema y una reducción igual de espectacular del coste total de propiedad. Con los procesadores UltraSPARC IV y los nuevos sistemas PCI+ E/S intercambiables en caliente, el servidor Sun Fire E25K ofrece casi el doble de capacidad de cálculo y mayor rendimiento de E/S que el sistema Sun Fire 12K anterior, y todo ello con el mismo tamaño. Esto permite a los clientes reducir el coste de propiedad de varias formas. El cliente puede desplegar las aplicaciones existentes en un dominio/servidor más pequeño o utilizar menos procesadores, lo que reduce el coste inicial, la alimentación y refrigeración, así como los costes de soporte técnico y de mantenimiento. Esto también permite a los clientes ampliar las aplicaciones sin tener que comprar más capacidad con tanta frecuencia. Además, dada su mayor capacidad de cálculo, junto con los dominios dinámicos de sistemas (Dynamic	Hasta 72 procesadores UltraSPARC IV de subproceso doble Más de 1/2 TB de memoria en un solo dominio Más de 120 TB de almacenamiento Hasta 18 dominios dinámicos del sistema

	System Domain), estos servidores son un vehículo excelente para la consolidación de los servidores del antiguo sistema UltraSPARC II o de la competencia.	

A partir de estos datos, vemos que la principal técnica aplicada para conseguir una mayor potencia de cómputo es la replicación de unidades de cómputo elementales (en estos servidores los procesadores UltraSparc IV con multithreading).

Debido a esta escalabilidad de la potencia de cómputo y al creciente apogeo de Java como lenguaje de programación de aplicaciones, sería muy conveniente disponer de un entorno de ejecución que fuese capaz de aprovechar dicha capacidad de procesamiento.

JikesRVM es una máquina virtual cuyas características principales le hacen dar un muy buen rendimiento en servidores. Las restricciones de memoria en servidores no son tan ajustadas como puedan ser en otras plataformas como un PC de usuario. Por otro lado una máquina virtual en un servidor debe satisfacer otra serie de requisitos como los presentados a continuación:

1 Explotación de procesadores de alto rendimiento: Los compiladores actuales no consiguen realizar las optimizaciones necesarias para explotar las características de los procesadores hardware actuales (jerarquía de memoria, paralelismo de instrucciones, multiprocesadores paralelos, …), que son necesarias para obtener un rendimiento comparable al de los lenguajes compilados estáticamente.
2 SMP escalabilidad: Los multiprocesadores de memoria compartida son muy populares en servidores. Muchas de las JVM actuales mapean los hilos necesarios para una aplicación java, sobre los pesados hilos del sistema operativo. Esto conduce a una pobre escalabilidad en los programas Java multihilo a medida que el número de hilos se incrementa.
3 Límite de hilos: Muchas aplicaciones Java necesitan de la creación de un nuevo hilo para cada una de las peticiones entrantes. Sin embargo debido a las restricciones del sistema operativo, muchas JVMs son incapaces de crear una gran número de hilos y por tanto tienes que limitar el número de peticiones entrantes. Esto limita mucho las aplicaciones que precisan de un gran número de usuarios.
4 Continua disponibilidad: Algunos servidores deben ser capaces de satisfacer continuas peticiones entrantes que duran largos periodos de tiempo, esto no parece ser una prioridad a la hora de diseñar máquinas virtuales.
5 Rápida respuesta: Algunos servidores tienen la rigurosa obligación de dar respuesta a una aplicación en un corto espacio de tiempo.
6 Uso de librerías: Las aplicaciones de servidores escritas en código Java están típicamente basadas en librerías existentes (beans, frameworks,…). Puesto que estas librerías están hechas para tratar casos genéricos, suelen ofrecer un pobre rendimiento.
7 Degradación de rendimiento: Es preferible que el rendimiento de un servidor se vea degradado cuando la carga de trabajo es excesiva, a que este se bloquee.

Prácticamente todos estas condiciones son satisfechas por JVM, lo que hace que esta tenga un comportamiento ideal en servidores: La explotación de procesadores de alto rendimiento es llevada a cabo por el *optimizing compiler*, los dos siguientes requisitos son satisfechos por la implementación de hilos poco pesados llevados a cabo en la implementación de la MV y el punto 4 se consigue gracias a la ayuda proporcionada por los tipos construidos en Java que evitan que el sistema se quede colgado, o disminuya su rendimiento. Es de esperar que el requisito 5 se cumpla a medida que se vayan desarrollando los nuevos algoritmos de gestión de memoria concurrentes e incrementales. El sexto requisito será satisfecho a través de las transformaciones específicas hechas a medida por el optimizing compiler, para un conjunto de librerías concretas, como por ejemplo, para el contexto de llamada de un servidor de aplicaciones. Aunque sabemos que no hay forma de garantizar la satisfacción del séptimo requisito, intentamos no perderlo de vista.

2. Introducción a JikesRVM y dJVM.

2.1. JikesRVM.

En esta sección damos una visión general de lo que es JikesRVM; la máquina virtual Java en la cual está basada nuestro proyecto.

JikesRVM es una máquina virtual desarrollada por el centro de desarrollo T.J. Watson de IBM basada en la desarrollada en el proyecto de investigación Jalapeño cuyo código fuente fue liberado por IBM con el objetivo de acelerar el desarrollo de tecnologías sobre máquinas virtuales. Sus principales características son las siguientes:

- Está completamente desarrollada en Java
- Utiliza dos compiladores y no interpretes
- Formada por una ligera estructura de hilos
- Dispone de un optimizado y agresivo compilador
- Una infraestructura de compilación flexible y de fácil adaptación.

JikesRVM esta pensada para ejecutar programas Java, típicamente utilizados en la investigación del diseño de una máquina virtual. Ésta proporciona un amplio banco de pruebas para las distintas tecnologías y la oportunidad de valorar diferentes alternativas de diseño. Puede ser utilizada en distintas plataformas (AIX™/PowerPC™, Linux®/PowerPC, OS X/PowerPC, y Linux/IA-32) y ofrece un excelente rendimiento en muchos de los benchmarks más utilizados en el panorama actual. Además JikesRVM incluye las últimas tecnologías de máquinas virtuales para compilación dinámica, optimización, recolección de basura, programación de hilos y sincronización.

La principal característica de esta máquina es que a diferencia de la gran mayoría de las máquinas virtuales existentes, que están construidas en lenguajes de programación nativos (típicamente C y C++) esta construida en un lenguaje de programación relativamente moderno como es Java. Éste proporciona facilidad de portabilidad, y una distribución uniforme de memoria para la máquina virtual. Aunque ha habido pocos casos de maquinas virtuales desarrolladas en Java, la mayoría de ellos están apoyados en una máquina virtual subyacente, lo que hace que su rendimiento se vea fuertemente disminuido. Jikes en este aspecto es única ya que es autosuficiente, se encuentra plenamente desarrollada en Java, y su código Java corre bajo si misma sin la necesidad de una segunda máquina virtual.

Con respecto a su uso, decir que fue inicialmente creada para la investigación, y ello hace que pueda ejecutar muchos pero no todos los programas Java existentes. El *classpath* que Jikes utiliza no proporciona una completa cobertura de Java; Swing y AWT no pueden ser completamente utilizados. JikesRVM tampoco soporta algunas otras características como *bytecode verification.* A pesar de estas limitaciones, puede ejecutar aplicaciones substanciales como por ejemplo Eclipse que es un interfaz de programación, de código abierto mundialmente utilizado.

¿Por qué es interesante JikesRVM para la investigación?: En el 2001 una de sus primeras versiones fue puesta a disposición de 16 universidades. Éstas encontraron en JikesRVM un excelente vehículo para la investigación de la máquina virtual. Muchas de estas universidades abandonaron sus esfuerzos en la creación de su propia MV a favor del uso de JRVM. Además desde

que en Octubre del 2001 se liberase el código fuente, han aparecido muchos trabajos de investigación, conferencias y cursos sobre el uso de dicha máquina.

Las cualidades más significativas mencionadas por estos investigadores y que hacen de JRVM entorno ideal para la investigación son:

- Está escrita en Java
- Está diseñada para la investigación, lo que hace que tenga una buena estructura modular que incluye:
  - Una estructura de optimización flexible y adaptativa.
  - Una infraestructura de optimización de compilación robusta.
  - Una serie de herramientas altamente configurables para el manejo de memoria
- Una buena fuente de desarrollo que muestra:
  - Resultados verídicos
  - Un rendimiento competitivo con los principales sistemas comerciales
- Una amplia y estable comunidad de usuarios que permite intercambiar, comparar y contrastar información acerca de Jikes.

2.1.1. Visión general de la estructura de JikesRVM.

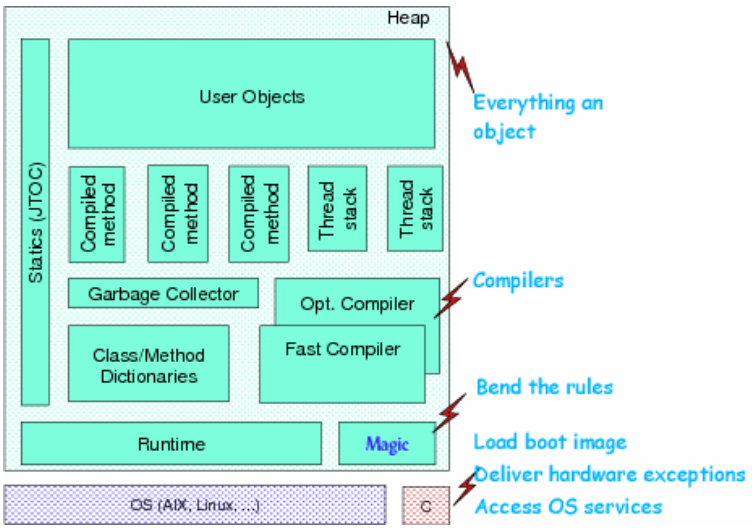
Esta sección describe muy brevemente como está estructurado el código de JikesRVM.

2.1.1.1. Componentes.

Jikes a grandes rasgos está compuesto por los siguientes componentes:

- Core Runtime (thread scheduler, class loader, library support, verifier, etc.) Este componente es el responsable del manejo de todas estructuras de bajo nivel necesarias para interacción con librerías y la ejecución de aplicaciones.
- Compilers (baseline, optimizing, JNI) este componente es el responsable de generar el código ejecutable desde el bytecode.
- Memory Managers: Este componente encargado de la asignación y recolección de objetos durante la ejecución de una aplicación.
- Adaptative optimization system: Es el responsable de perfilar el ejecutable, para ello utiliza las optimizaciones de compilación y así incrementar el rendimiento.

En la siguiente figura mostramos un esquema en el cual podemos observar como está organizada



2.1.1.2. Estructura de paquetes.

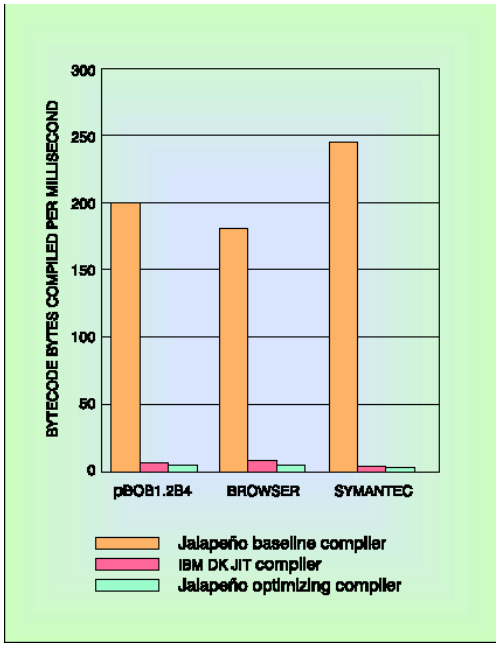
En la versión 2.2.0 que es con la que se ha trabajado, existen 9 paquetes. Cualquier clase está en uno de estos paquetes:

- com.ibm.JikesRVM: Comprende las clases para el *core runtime* exceptuando *library support*.
- com.ibm.JikesRVM.adaptative: Clases para la optimización del sistema
- com.ibm.JikesRVM.classloader: Implementación del *classloader* y las estructuras asociadas, junto con la representación de métodos y clases de la VM
- com.ibm.JikesRVM.jni: Implementacíon del JNI
- com.ibm.JikesRVM.memoryManagers.JMTK: Clases que componen los elementos de manejo de memoria JMTK(Java Memory Manager Toolkit)
- com.ibm.JikesRVM.opt: Clases asociadas con la optimización de la compilación excepto para las clases de IR-related que disponen de su propio paquete.
- com.ibm.JikesRVM.opt.ir: Clases necesarias para la interpretación intermedia (IR) de la optimización de compilación.
- com.ibm.JikesRVM.OSR Clases relacionadas con On-Stack-Replacement

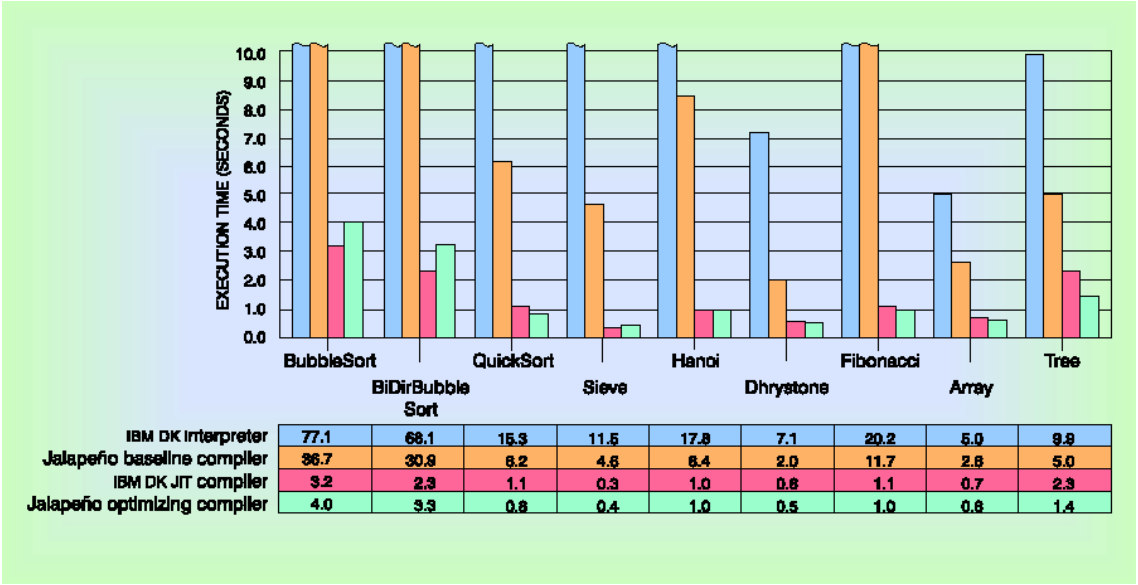
Si desea descargar Jikes, o conocer más sobre este visite la web de JikesRVM:
http://www.ibm.com/developerworks/oss/jikesrvm

2.1.2. Evaluación del rendimiento.

Una de las principales características de JikesRVM es su rapidez y gran capacidad de optimización a la hora de la compilación. En estas gráficas podemos comparar la velocidad de compilación del baseline compiler (implementado en Java y optimizado por el optimizing compiler) y el IBM DK JIT (implementado en código nativo). Como vemos el primero es del orden de 30 a 45 veces más rápido. Por su parte el optimizador es el más lento pero está casi a la altura del JIT.



En la siguiente figura podemos observar el tiempo de ejecución del código generado por los distintos compiladores para la misma aplicación (en este caso un benchmark: symantec). En ella podemos observar el efecto de la optimización en la compilación y además la escasa diferencia entre el compilador en código máquina y el baseline compiler (con y sin optimización).



	BubbleSort	BiDirBubble Sort	QuickSort	Sieve	Hanoi	Dhrystone	Fibonacci	Array	Tree
IBM DK Interpreter	77.1	68.1	15.3	11.5	17.8	7.1	20.2	5.0	9.9
Jalapeño baseline compiler	36.7	30.9	6.2	4.6	6.4	2.0	11.7	2.8	5.0
IBM DK JIT compiler	3.2	2.3	1.1	0.3	1.0	0.6	1.1	0.7	2.3
Jalapeño optimizing compiler	4.0	3.3	0.8	0.4	1.0	0.5	1.0	0.6	1.4

2.2 dJVM.

Nuestro objetivo es la adaptación de la máquina virtual dJikes al cluster **Aneto** situado en el departamento DACYA en la Facultad de físicas de la Universidad Complutense de Madrid. Una vez adaptado analizaremos su comportamiento ante diversos benchmarks, estudiaremos detenidamente su estructura y gestión de memoria, en concreto el recolector de basura. Este cluster, proporcionaría así mejoras cuando sea posible una ejecución en paralelo Un ejemplo de una aplicación en la cual se obtendrían grandes ganancias en el rendimiento sería por ejemplo un servidor web.

2.2.1. Vista previa de la estructura de Jikes Distributed Java Virtual Machine.

El árbol de directorios está en el directorio raíz *$RVM_ROOT*. En *$RVM_ROOT/rvm/src* almacenamos los archivos fuente, mientras que en *$RVM_ROOT/rvm/src/vm* almacenamos el código de Jikes junto con las extensiones de dJVM. En el subdirectorio cluster almacenamos las clases necesarias añadidas para la implementación de éste.

La siguiente estructura de directorios aproxima la estructura de paquetes:

- *$RVM_ROOT/rvm/src/vm/arch*: Contiene una serie de directorios necesarios para las dependencias arquitectónicas de Intel y PowerPc.
- *$RVM_ROOT/rvm/src/vm/classLoader*: Contiene las clases que definen los objetos necesarios para los distintos tipos utilizados. Aquí también encontramos los mecanismos de carga y transformación de tipos.
- *$RVM_ROOT/rvm/src/vm/runtime*: Contiene el core necesario para el sistema en tiempo de ejecución. Aquí tenemos incluido: métodos estáticos para algunas funciones de la VM, funciones del sistema, tablas de variables estáticas y métodos de soporte para el linkado dinámico.
- *$RVM_ROOT/rvm/src/vm/scheduler*: contiene el núcleo de proceso y las estructuras de manejo de hilos, incluido: colas de hilos, funciones para el manejo y bloqueo de hilos.
- *$RVM_ROOT/rvm/src/vm/memoryManagers*: contiene los elementos necesarios para la recolección de basura. Este algoritmo es seleccionado en el momento de la construcción.

La implementación de la versión distribuida está basada en unos cambios sobre la implementación de JikesRVM y una serie de extensiones que encontramos en *$RVM_ROOT/rvm/src/vm/cluster*. Este directorio, basado en la estructura de la máquina de IBM contiene una serie de subdirectorios adicionales, *$RVM_ROOT/rvm/src/vm/cluster/communication*, *$RVM_ROOT/rvm/src/vm/cluster/messages* y *$RVM_ROOT/rvm/src/vm/cluster/external* que contienen las extensiones necesarias para el soporte de la distribución. Su función es la siguiente:

- *$RVM_ROOT/rvm/src/vm/cluster/communication:* Contiene la infraestructura necesaria para activar y procesar la comunicación a través de mensajes de las distintas capas.
- *$RVM_ROOT/rvm/src/vm/cluster/messages:* Contiene los tipos que constituyen los tipos de mensaje y el codigo necesario para su manejo.
- *$RVM_ROOT/rvm/src/vm/cluster/external:* Proporciona una serie de métodos que constituyen una interfaz para el acceso remoto a la información.

3. Estructura del código fuente.

3.1. Introducción.

Este capitulo da una perspectiva general de la estructura del código fuente de Jikes Distributed Java Virtual Machine.

Generalmente, las modificaciones de dJVM son de dos tipos. El primero son extensiones a las clases de la Jikes Research Virtual Machine original, que normalmente son identificadas por directrices *//- if RVM_WITH_CLUSTER*. Estas aparecen en ficheros *.java* en el directorio *$rvmRoot/rvm/src/vm* y sus subdirectorios (excepto el subdirectorio cluster). El segundo son nuevas clases creadas para el entorno cluster distribuido; estas aparecen en el directorio *$rvmRoot/rvm/src/vm*/cluster y sus subdirectorios.

Este capitulo esta organizado en secciones correspondientes a diferentes aspectos de la JVM. Estas secciones contienen una descripción de los ficheros fuente modificados, de los dos tipos anteriormente comentados.

3.2. Código de dJVM cluster.

Hay ciertas modificaciones de cluster en ficheros de *$rvmRoot/rvm/src/vm.*

VM.java
VM_Properties.java
VM_Configuration.java
VM_ObjectLayoutConstants.java

No obstante, la mayor parte de los códigos fuente de cluster se encuentran en el directorio *$rvmRoot/rvm/src/vm/cluster*/, y sus subdirectorios *allocator, communication, classLoader, messages, bytecode, runtime, scheduler, utility,* y *external.*

3.2.1. Directorio de comunicaciones.

En cada nodo la clase *DVM_CommunicationManager* gestiona el envío y la recepción de mensajes con los demás nodos del cluster. Esta clase es un interfaz a una instancia de la clase abstracta *DVM_ComunicationSubstrate*. Tal como la JVM distribuida esta implementada, una instancia de la subclase no-abstracta *DVM_CommunicationSubstrateSocket* es asignada a este campo durante la inicialización. Esta clase usa TCP/IP para enviar y recibir paquetes.

El substrato establece una conexión de socket (clase *DVM_Socket*) por cada otro nodo del cluster. A través de estos socket todos los mensajes son enviados y recibidos.

Para ilustrar como las clases del subdirectorio *communication* trabajan juntas, mostramos a grandes rasgos el proceso por el cual un mensaje es enviado de un nodo a otro y como el acuse de recibo es devuelto:

1. Un nuevo mensaje será creado cuando es requerido por la ejecución del programa.
2. El mensaje obtendrá un *buffer*, una instancia de la clase *DVM_MessageBuffer,* del gestor de comunicación.
3. Si es requerido un acuse de recibo, el mensaje se registrará él mismo, invocando el método estático *register ()* de *DVM_CommunicationManager*. Esto provoca que el mensaje sea insertado en un *array* y a su campo *responseID* se le asigne el valor de su índice en el *array*.
4. El mensaje creará una versión codificada de si mismo en el *buffer*. El método *DVM_CommunicationManager output ()* es invocado con el *buffer* pasado como argumento.
5. Si es requerida una respuesta, el mensaje se suspenderá usando su propio método *wait ()*, hasta que es posteriormente notificado por un mensaje de acuse de recibo.
6. En el final de la recepción, como los paquetes son recibidos por el *DVM_Socket* son pasados a su objeto *DVM_MessageReceiver*.
7. El hilo *DVM_MessageReceiver* reconstruye una copia del *buffer* original con estos paquetes, en el bucle principal de su método *run()*. Invoca entonces al método *DVM_Message decode(),* pasando el *buffer* reconstruido como argumento.
8. El método *DVM_Message decode()* leerá el primer entero de el *buffer*, el cual indicara el tipo de mensaje. Entonces invocará el método *decode()* apropiado para ese tipo.
9. El método *decode()* instanciará un nuevo mensaje, pasando el resto del *buffer* a su constructor. En este punto, hay dos posibilidades:
    a) Si se requiere *little processing*, el método *process()* del mensaje será invocado inmediatamente, tras lo cual el método *decode()* devuelve null. Un ejemplo de un tipo de mensaje en el cual ocurre esto es *DVM_MessageAccessRefGetFieldReq.*
    b) Si es requerida una cantidad significativa de proceso, el método *process()* del mensaje no será invocado, y el mensaje instanciado será devuelto por el método *decode().* Un ejemplo de tipo de mensaje en el cual ocurre esto es *DVM_MessageClassLoadReq.*
10. Si un mensaje, distinto que null, es devuelto por el método decode(), significa que el método process() no ha sido invocado. El mensaje es pasado a el método *handle()* de *DVM_CommunicationManager*. Esto provoca que un hilo *DVM_MessageHandler* invoque el método *process()* del mensaje en un momento que será determinado por el programador para ser más optimo.
11. El método *process()* del mensaje puede incluir la creación de un mensaje de acuse de recibo. El mensaje puede contener datos obtenidos del nodo recipiente.
12. El mensaje de acuse de recibo es codificado en un *buffer* y enviado de vuelta al remitente original.
13. El mensaje de acuse de recibo es reconstruido en el nodo desde el cual el mensaje de petición original fue enviado. Si el mensaje original ha sido registrado, será recuperado del *array DVM_Register* usando el *responsID* copiado del mensaje original en el otro fin.
    Alguna información, requerida por el mensaje original, será pasada a los campos apropiados del mensaje usando los métodos adecuados, después el método *notify()* del mensaje de petición original será invocado para permitir al mensaje original reanudar la ejecución.

Lo anterior ilustra el caso más complejo, pero sin embargo ejemplifica la mayoría de los intercambios de mensajes que ocurren durante la ejecución de la dJVM. Algunos mensajes no requieren acuse de recibo. Un ejemplo es *DVM_MessageControlTBInform*, que escribe un mensaje para todos los nodos del cluster en la terminal de salida del nodo maestro.

3.3. Código de carga de clases de la VM y dJVM.

El código fuente tratado a continuación se encuentra en los subdirectorios *vm/classLoader/, vm/runtime* y *vm/cluster/classLoader* de *$rvmRoot/rvm/src*.

3.3.1. Una visión general de la carga de clases distribuida.

Todas las definiciones de clases, que no han sido creadas antes del inicio de la secuencia de arranque del cluster, deben de ser construidas, cuando sean requeridas, centralmente en el nodo maestro y, desde allí, replicadas en el nodo esclavo. Esto posibilita asegurarse de que las definiciones en cada nodo son consistentes. De esta manera, por ejemplo, si el elemento 5091 de *VM_Statics* se refiere al método estático *Thread.yield()* en un nodo, entonces el elemento 5091 se debe referir también al método estatico *Thread.yield()* en cualquier otro nodo.

DVM_ClassLoader: permite instancias *ClassLoader* en cualquier nodo

Una meta del diseño de la dJVM fue no tener que imponer restricciones sobre donde una instancia *ClassLoader* puede ser creada y usada, por lo que es posible para una clase ser cargada por cualquier aplicación *ClassLoader* en cualquier nodo.

Para hacer esto, el método *ClassLoader defineClassInternal()*, a través del cual todas las definiciones de clase son creadas, ha sido cambiado y una nueva clase, *DVM_ClassLoader*, ha sido añadida. Una única instancia de *DVM_ClassLoader* es creada en el nodo maestro durante el arranque, y una referencia remota a ésta es dada a cada nodo esclavo. Estas mejoras son mostradas en las Figuras 3.1 y 3.2 mostradas más adelante.

Durante la ejecución de programas, en un nodo esclavo, el mecanismo de enlace dinámico causa que la clase sea cargada usando la aplicación *ClassLoader*. Si no han sido instanciadas las clases *ClassLoader* definidas por el usuario, éstas serán instancias del *ClassLoader* original ubicada en el nodo maestro.

No obstante, si un nuevo *ClassLoader* ha sido creado en el nodo local, invocara el *VM_ClassLoader defineClassInternal()* localmente. Los cambios, mencionados anteriormente, causan que esta llamada sea redirigida al método estático *VM_ClassLoader defineClassInternal()* del nodo maestro a través de la referencia remota a la instancia *DVM_ClassLoader*. En el nodo maestro, *defineClassInternal()* invoca a *localDefineClassInternal()* el cual es básicamente el mismo que *defineClassInternal()* en la JVM no-distribuida.

Limitaciones actuales de DVM_ClassLoader

Este mecanismo falla al trabajar con *CassLoaders* instanciados en nodos esclavos. De cualquier

manera, no interfiere con la correcta carga de clases cuando la instancia de *ClassLoader* ha sido creada en el nodo maestro.

Las aplicaciones que crean instancias de *ClassLoader* pueden no funcionar a no ser que se haga uso explicito de la asignación remota de objetos, para forzar a toda instancia de ClassLoader ser asignada en el nodo maestro.

```
32 static DVM_ClassLoader clusterClassLoader;
...
1184 public static final Class defineClassInternal(String className,
1185 InputStream is,
1186 ClassLoader classloader)
1187 throws ClassFormatError {
1188 if(VM_Class.getRemoteReplicationRequired()) {
1189 return clusterClassLoader.defineClassInternal(className,is,classloader);
1190 }
1191
1192 return localDefineClassInternal(className,is,classloader);
1193 }
...
1197 public static final Class localDefineClassInternal(String className,
1198 InputStream is,
1199 ClassLoader classloader)
1200 throws ClassFormatError {
...
1229 }
```

Figura 3.1: Modificaciones a VM ClassLoader

```
34 public class DVM_ClassLoader {
...
39 public static final Class defineClassInternal(String className,
40 InputStream is,
41 ClassLoader classloader)
42 throws ClassFormatError {
43 return VM_ClassLoader.defineClassInternal(className,is,classloader);
44 }
45 }
```

Figura 3.2: Clase DVM ClassLoader

3.3.2. Replicación de definiciones de clase en etapas.

Las definiciones de clases, requeridas para la ejecución de programas, son construidas en diferentes etapas: construcción inicial del objeto *VM_Class*, carga, transformación, resolución, instanciación e inicialización. Cada una de estas etapas, excepto para la instanciación, resulta de la adquisición de entradas de la tabla *VM_Statics* y los distintos diccionarios.

En la práctica, muchas de las etapas tienden a ser hechas de una vez, sin embargo para que la dJVM se ajuste a las especificaciones de la JVM, es necesario que los cambios incrementales a estas definiciones puedan ser replicados separadamente. Una vez que hay una definición de clase completamente inicializada en el nodo maestro, esta definición puede ser copiada en cualquier otro nodo donde puede ser requerida.

La replicación de definiciones de clase debe ocurrir antes de que cualquier aplicación de tipo *Thread* (hilo) pueda ser lanzada en un nodo esclavo. Los mecanismos de enlace dinámico causan la replicación de la definición completa de la subclase *Thread* adecuada en el nodo esclavo, antes de que su constructor o método *run* sean invocados.

Una subclase *Thread* típica contendrá también referencias a otras clases dentro de objetos *VM_Class*. Como los métodos de esas clases son requeridos para ser ejecutados, los mecanismos de enlace dinámico causarán que más definiciones de clase sean creadas centralmente en el nodo maestro y entonces replicadas localmente.

3.3.3. Detalles de la implementación de la replicación.

La carga de clases y la replicación es normalmente forzada por el método estático *initializeClassForDynamicLink*, de la clase *VM_Runtime*, mostrado en la Figura 3.3.

El método *initializeClassForDynamicLink()* no es muy diferente al de la versión no-distribuida. El código para la replicación de la información de definición de clase esta en gran parte dentro de los métodos que construyen las definiciones de clase.

La excepción es el método *loadClass()* de la subclase *ClassLoader*. Con el fin de evitar la necesidad de cambiar este API, la replicación es hecha separadamente. La instancia *ClassLoader* debe ser asignada en el nodo maestro por los problemas con *DVM_ClassLoader* descritos anteriormente.

La replicación de la definición de clase cargada es realizada en la línea 8, mediante la invocación del método *clusterReplicateLoadedClass()* de *VM_Class*. Casi toda la información de definiciones de clase es copiada por este metodo. Este metodo llama sucesivamente al metodo estático *clusterReplicateClassComponents()* de la clase *DVM_Replicator* mostrado en la Figura 3.4. Este construye y envia un mensaje ´cluster replicate definition request´ (linea 5) a el nodo maestro. El método correspondiente invocado en el nodo maestro para servir la petición es mostrado en la Figura 3.5. Cuando la respuesta llega, el buffer de respuesta es deserializado y usado para construir una copia local de la definición de clase.

Normalmente todas las definiciones de clase copiadas habrán sido transformadas ya en el nodo maestro, por lo que la invocación de la instancia del método *transform()* en la línea 15 de la Figura 3.3 no tendrá efecto normalmente. De cualquier manera, en el extraño caso donde una definición de clase, cargada, pero no transformada, ha sido copiada, la invocación de *invoke()* provocará que la definición de clase sea copiada otra vez, pero solo después de forzar la transformación de la definición de clase en el nodo maestro.

En el transcurso de la invocación del método *resolve()* (línea 17 en Figura 3.3), un mensaje *DVM_MessageClassResolveReq* es enviado. Su mensaje con acuse de recibo contiene las entradas de la tabla *VM_Static* para los métodos estáticos y los campos estáticos de la clase.

No se obtiene ninguna información del nodo maestro mediante el método *instantiate()* (línea 18 en Figura 3.3). La información añadida por *instantiate()* puede ser construida independientemente en cada nodo sin causar que las definiciones de clase se vuelvan inconsistentes.

Durante la ejecución de *initialize()* (línea 19 en la Figura 3.3), el inicializador estático de la definición de clase en el nodo maestro será remotamente invocado. Esto inicializará los campos

estáticos en el nodo maestro. Todos los campos estáticos de clases de aplicación son accedidos desde el nodo maestro, por lo tanto no son copiados al nodo esclavo.

```
1 public static void initializeClassForDynamicLink(VM_Class cls)
2       throws VM_ResolutionException {
3       try {
4               //-#if RVM_WITH_CLUSTER
5                if(!cls.isLoaded()) {
6                       cls.getClassLoader().loadClass(
7                       cls.getDescriptor().classNameFromDescriptor());
8                       cls.clusterReplicateLoadedClass();
9               }
10               //-#else
11                      cls.getClassLoader().loadClass(
12                      cls.getDescriptor().classNameFromDescriptor());
13               //-#endif
14              //-#if RVM_WITH_CLASS_TRANSFORMER
15                       cls.transform();
16              //-#endif
17              cls.resolve();
18               cls.instantiate();
19               cls.initialize();
20      } catch (ClassNotFoundException e) {
21               throw new VM_ResolutionException(
22              cls.getDescriptor(), e, cls.getClassLoader());
23       }
24 }
```

Figura 3.3: VM Runtime, método initializeClassForDynamicLink()

```
1 static DVM_MessageBuffer clusterReplicateClassComponents(int dictionaryId,
2       ClassLoader classloader, boolean forceTransform) {
3
4       DVM_MessageBuffer responseBuffer =
5                (new DVM_MessageClassReplicateDefReq(dictionaryId, classloader,
6               forceTransform)).getClassData();
7
8        VM_Atom.clusterCreateAtoms(responseBuffer);
9       VM_ClassLoader.clusterCreateTypes(responseBuffer);
10       VM_ClassLoader.clusterCreateFields(responseBuffer);
11      VM_ClassLoader.clusterCreateMethods(responseBuffer);
12      clusterCreateStringLiterals(responseBuffer);
13      clusterCreateIntLiterals(responseBuffer);
14       clusterCreateFloatLiterals(responseBuffer);
15      clusterCreateLongLiterals(responseBuffer);
16      clusterCreateDoubleLiterals(responseBuffer);
17
18      return responseBuffer;
19 }
```

Figura 3.4: DVM_Replicator, método clusterReplicateClassComponents()

```
1 static void clusterWriteClassComponents(int dictionaryId, DVM_MessageBuffer responseBuffer) {
2
3
4       VM_Atom.clusterWriteAtoms(
```

```
5       classIdSets[dictionaryId].getAtomIdSet(), responseBuffer);
6       VM_ClassLoader.clusterWriteTypes(
7       classIdSets[dictionaryId].getTypeIdSet(), responseBuffer);
8       VM_ClassLoader.clusterWriteFields(
9       classIdSets[dictionaryId].getFieldIdSet(), responseBuffer);
10      VM_ClassLoader.clusterWriteMethods(
11      classIdSets[dictionaryId].getMethodIdSet(), responseBuffer);
12      clusterWriteStringLiterals(dictionaryId, responseBuffer);
13      clusterWriteIntLiterals(dictionaryId, responseBuffer);
14      clusterWriteFloatLiterals(dictionaryId, responseBuffer);
15       clusterWriteLongLiterals(dictionaryId, responseBuffer);
16      clusterWriteDoubleLiterals(dictionaryId, responseBuffer);
17 }
```

Figura 3.5: DVM_Replicator, método clusterWriteClassComponents()

3.3.4. Sumas de control para la verificación de consistencia entre JVMs.

Al principio las JVMs en cada nodo deben de ser casi idénticas, excepto por unos pocos campos estáticos como *node identity.* Durante el arranque, esto es confirmado, cuando cada nodo esclavo calcula sumas de control (*checksums*) basadas en los valores contenidos en sus distintos diccionarios y su tabla *VM_Statics.* Estas sumas de control son enviadas al nodo maestro y cada una comparada con la equivalente suma de control en el nodo maestro. Si se encuentra alguna diferencia, un mensaje de error es impreso y la dJVM cesa su ejecución. Antes de apagarse, las claves de los diccionarios en los cuales hay sumas de control inconsistentes son escritas en ficheros en el nodo maestro y esclavo para su posterior comparación.

Después de que el arranque esté completo y los procesos de replicación de definiciones de clase bajo demanda del nodo maestro al esclavo comienzan, las sumas de control son usadas para verificar que cada definición copiada coincide con su original. Toda discrepancia en las sumas de control es tratada como un error fatal.

3.4. Código de VM y dJVM relacionado con la transformación de código.

Varias transformaciones de clase son realizadas para permitir realizar invocación, acceso remoto, y operaciones de supervisión. Las operaciones de invocación remota son la creación de métodos *proxy*, ver *DVM_ClassTransformProxy*, y la introducción de barreras software de referencias remota, ver *DVM_ClassTransformRemoteInvoke*. El acceso remoto y las operaciones de supervisión también introducen barreras software de referencia remota, y son agrupadas juntas en *DVM_ClassTransformAccess*.

3.4.1. Invocación de métodos.

Un método que es invocado en un objeto remoto es dirigido a un método proxy de ese tipo. El método *proxy* tiene la misma signatura, tipo devuelto y categoría de protección que el original, por tanto, puede ser sustituido sin violar el comportamiento de tipo o pila. Además, un *proxy* es creado solo para los métodos declarados de una clase, asegurando que el sistema de resolución dinámica de tipos de Jikes RVM resuelve al mismo tipo que el método no-*proxy*.

A cada método *proxy* le es dado el mismo nombre con el sufijo **proxy,* incluyendo los inicializadores <*init*>. Esto requiere la modificación del método *VM_Method.isObjectInitializer* para incluir el método <*init*>**proxy*.

El código es generado para un proxy solo si corresponde a un método concreto, por ejemplo no es una declaración de interfaz o un método abstracto. El código generado para un método *proxy* construye un mensaje y rellena el mensaje con los argumentos del método (esto puede necesitar la creación de nuevos UIDs, o el uso de un UIDs existente). Entonces envía el mensaje y espera una respuesta, ver sección 3.2.1. Después de un tiempo devuelve el valor recibido por la maquina de destino.

El fallo de referencia software es usado para determinar si un *proxy* o un método real ha de ser invocado. Cada invocación, *invokeinterface*, *invokespecial* o *invokevirtual* es sustituido por:
Procedure: replace invokevirtual (ref, class.method)

```
1:      if ref & 2 == 0 then
2:              invokevirtual (ref, class.method)
3:      else
4:              invokevirtual (ref, class.method*proxy)
5:      end if
```

donde class.method*proxy es el método *proxy* para *class.method*, y de forma parecida para *invokeinterface* y *invokespecial*. Nota: Las rutinas de transformación corrigen los mapas de excepciones y los mapas de número de línea.

Hay un conjunto de clases de núcleo que actúan solo con información interna. Estas clases son un subconjunto de las clases de soporte del núcleo de ejecución y principalmente incluyen las clases de mensaje. Estas deben ser definidas para cortocircuitar el problema recurrente de transformar todos los byte-codes de invocación. Un mecanismo de anotación para definir cual de esas clases puede o no tener *proxies* generados para ellas, no es usado ya que requeriría el destino de un byte-code de invocación para ser leído. Consecuentemente, todas las clases que son cargadas deben tener todas las clases referidas para leer; como semejante lista de clases que no tienen que ser transformadas es especificada, todas las otras clases son transformadas.

3.4.2. Acceso a datos.

Como con todos los accesos hay un plan de fallos para determinar si el dato esta disponible localmente o no. En esta versión, esto es hecho por software. Hay tres grupos básicos de acceso de datos que están modificados para incorporar el software de plan de fallos, estos son: accesos a elemento *array*, campo de instancia y campo de clase.

Esencialmente, accesos a elementos *array* y campos de instancia son lo mismo en contraste con el acceso a campo estático. Referencias locales a array y objeto son alineadas 4-byte y referencia directa a la ubicación del array u objeto. Referencias remotas a array y objeto no son alineadas 4-byte, pero son alineadas 2-byte.

Los byte-codes baload a través de saload, y bastore a través de sastore son transformados en una barrera software de referencia remota para seleccionar mecanismos de acceso local o acceso remoto. Los chequeos de límites actualmente solo son hechos en el emplazamiento que contiene el *array*. Incluyendo los cambios de los fallos de referencias software a la adaptación de baload:
Procedure: baload(ref,i)

```
1:      if ref & 2 == 0 then
2:              baload(ref,i)
```

```
3:      else
4:              invokestatic DVM RemoteAccess.remoteGetArrayByte(ref,
i)
5:      end if
```

Similarmente para los otros byte-codes de acceso a *arrays*. Y también para byte-codes *arraylength*. A nivel de código, los accesos a campo de instancia *getfield* y *putfield* son fundamentalmente de la misma forma.

Jikes RVM mantiene un array, llamado JTOC (Tabla de Contenido Java) que contiene todos los valores referenciados estáticamente en el sistema. Estos incluyen literales, campos de clase, y referencias a métodos (la manipulación de este array implica cierta escritura mágica, ver VM_Magic) y cada entrada de JTOC tiene un descriptor asociado.

Los campos de clase son fundamentalmente diferentes a los campos de instancia. Un campo de clase es global al sistema. Algunos de ellos son usados para mantener estructuras internas de soporte de ejecución para Jikes RVM. Cuando la distribución es introducida algunos datos anteriormente globales se convierten en datos específicos de una máquina, por ejemplo colas de hilos o datos de tipo. Como consecuencia debemos distinguir entre estáticos locales y estáticos globales.

Los estáticos locales se distinguen de los estáticos globales anotando las clases para las que sus estáticos son locales. Una clase que contiene estáticos locales es anotada implementando el interfaz vacío DVM_LocalOnlyStatic.

Para variables que pueden ser contenidas en el nodo maestro o esclavo, usamos uno de los bits disponibles del descriptor asociado con cada variable estática para indicar si es contenida por el nodo maestro o es contenida localmente. Cada acceso debe determinar si el campo es local o remoto y realizar un acceso similar al acceso a un campo de instancia. Este acceso a memoria añadido para determinar si es local o remoto es una fuente potencial de degradación de rendimiento, queda por ver en que medida esto es un impedimento.

3.4.3. Operaciones de supervisión.

Hay dos tipos de operaciones de supervisión en Java, operaciones explícitas llamadas por los byte-codes *monitorenter* y *monitorexit*, y operaciones implícitas sobre entradas y salidas a un método sincronizado. Para las operaciones explicitas de bloqueo es usado un mecanismo software remoto de fallos y un método de traducción de código similar al usado para el acceso a array y campos remotos. El bloqueo implícito es dirigido al emplazamiento del objeto a través del mecanismo de invocación remota, de esa manera un *proxy* no esta nunca sincronizado. En ambos casos el código de JikesRVM dirige las operaciones de bloqueo localmente a *VM_Lock.lock* y *VM_Lock.unlock*, ver sección 3.7.

Para métodos sincronizados las operaciones de bloqueo y desbloqueo son realizadas en la clase asociada al tipo del objeto. Esto requiere resolución en tiempo de ejecución, consecuentemente, un mecanismo similar al mecanismo de invocación es usado para determinar el tipo. Para los métodos abstractos las peticiones son dirigidas a *VM_Class*, ver sección 3.7.

3.5. Código del compilador base de la VM.

El *baseline compiler* es un compilador sencillo, que traduce eficazmente byte-codes Java a código nativo. El proceso de traducción es rápido, pero no produce códigos eficientes. Su código fuente esta en el directorio *$rvmRoot/rvm/src/vm/arch/intel/compilers/baseline*.

El plan de fallos es inicialmente implementado usando un plan de fallos software. Este es implementado en el *baseline compiler* modificando ciertos mapeos de Java byte-codes a código nativo. En particular: accesos a *arrays*, instancias y campos de clase; invocación de código; comprobación de tipo. Para soportar estos mecanismos también se introducen ciertos métodos *magic*.

3.5.1. Invocación de métodos.

La invocación de métodos se divide en dos categorías, métodos de instancia y métodos de clase. El modelo de invocación dirige las llamadas de los métodos de instancia a la máquina que tiene el objeto, y las invocaciones a métodos de clase a la máquina local. Consecuentemente, la invocación a métodos estáticos (*invokestatic*) permanece inalterada, mientras que las invocaciones de instancia deben reubicar objetos remotos.

Los métodos de instancia se dividen en invocaciones a un método que es conocido en tiempo de compilación, y aquellas que deben ser resueltas en tiempo real. La transformación de invocaciones a métodos descrita en la Seccion 3.4 cambia el mecanismo software de fallos al nivel byte-code. Por lo tanto, para invocaciones que no requieren determinación de tipo en tiempo de ejecución no es necesario ningún cambio del código (por ejemplo *invokespecial* permanece inalterado ya que son invocaciones de tipos estáticamente).

Las invocaciones dependientes de tipo en tiempo de ejecución, como *invokevirtual* e *invokeinterface*, deben determinar el objeto que esta siendo llamado. Para tipos contenidos localmente esto es obtenido de la cabecera del objeto, y para tipos contenidos remotamente esto es mantenido en los datos de la recopilación de referencias remotas. Por lo tanto, es usada una técnica de fallos de dirección para determinar y obtener el tipo.

3.6.2. Acceso a datos.

El mecanismo dinámico de carga y enlace también demora la resolución de información de tipo para *arrays*, campos de instancia y campos de clase. Así si el desplazamiento del campo no puede ser determinado en tiempo de compilación una pequeña cantidad de código es insertada para forzar la resolución durante la ejecución. Este necesita la misma información de resolución de tipo en tiempo de ejecución tanto para objetos contenidos remotamente como para invocación remota.

3.6.3. Métodos magic.

JikesRVM incluye un mecanismo para generar un pequeño número de métodos especiales, llamados métodos magic. Estos métodos no son compilados usando el mecanismo común de compilación pero son cortocircuitados al compilar el byte-code de invocación, lo cual produce una secuencia predefinida de instrucciones nativas (o instrucciones IR en el compilador de optimización). El conjunto de métodos magic es declarado en *VM_Magic* e identificado por las cadenas de caracteres en *VM_MagicNames*. El conjunto de métodos magic ha sido extendido en la dJVM.

3.6. Código de VM y dJVM relacionado con la gestión de memoria.

La gestión de memoria es manejada por *Java Memory Management Toolkit* (JMTk). Las clases del interfaz con JMTk se encuentran en *…/src/vm/cluster/memoryManagers/*.

3.6.1. Asignación de objetos.

Instancias de tipos primitivos, tipos *array* y la mayor parte de las clases son siempre asignadas localmente. La excepción son las clases que implementan el interfaz *Runnable* o *DVM_ForceRemoteAllocation*. Los cambios necesarios, que permiten la asignación remota de instancias de estos tipos son mostrados en las Figuras 3.6, 3.7 y 3.8.

En el método *resolvedNewScalar()* de *VM_Runtime*, mostrado en la figura 3.6, una instancia de una clase es asignada invocando el método *allocateScalar()*de la clase *DVM_Allocator* en la línea 15. Este método es mostrado en la Figura 3.7. El nodo en el cual el objeto es asignado es devuelto por el método *allocateNode()* de *policy*, una instancia de la subclase *DVM_AllocatorPolicy*, en la línea 5. Si policy es de los tipos 'round robin', 'random', o 'node' y el tipo a ser asignado es 'runable' o 'force remote allocation', entonces el nodo devuelto debe de ser remoto, en cuyo caso el método *allocateScalar()* del asignador local de ese nodo es invocado remotamente (línea 7). Un asignador local es una instancia de la clase *DVM_LocalAllocator*.

La asignación, mediante el método *allocateScalar()* es mostrada en la Figura 3.8. En la línea 4 es obtenida una definición completamente inicializada de la clase, si es necesario forzando la construcción de esa definición en el nodo *maestro* y copiando la definición en el nodo local. De esa definición se obtiene un *type information block(tib).* Ese *tib* es entonces usado para asignar un objeto mediante el método *allocateScalar()* de *VM_Interface* de la misma forma que ocurriría en una JVM no distribuida.

Los diferentes tipos de políticas de asignación son:

1. null – Es interpretado como asignar siempre localmente.
2. *DVM_AllocatorPolicyRounRobin*.
3. *DVM_AllocatorPolicyRandom.*
4. *DVM_AllocatorPolicyNode*- dirige todas las asignaciones a un nodo en particular.

Las políticas de asignación no-local solo asignarán tipos ‘runable’ y ‘force remote allocation’ de forma no-local. La política de asignación por defecto es ‘round robin’, pero puede ser cambiada con parámetros de la línea de comandos.

En tiempo de ejecución puede establecerse una nueva política de asignación mediante una llamada a DVM_Allocator.setPolicy(policy). Tan pronto como la política de asignación es establecida, es usada. Hay ciertas restricciones en el uso de políticas de asignación en tiempo de ejecución:

• El objeto asignador debe ser un objeto local a la maquina, ya que una llamada remota puede requerir alguna asignación.

• El método allocateNode(VM_Class klass) debe ser compilado antes de usarlo, ya que la compilación requiere asignación.

La falta de alguna de estas condiciones causará un bucle que solo terminara cuando todo el espacio de pila sea consumido.

```
1 public static Object resolvedNewScalar(int size, Object[] tib, boolean hasFinalizer, int allocator)
2                               throws OutOfMemoryError {
3
4
5
6
7 ...
8
9         // Allocate the object and initialize its header
10         //-#if RVM_WITH_CLUSTER
11        Object newObj = null;
12        VM_Type type = (VM_Type)(VM_Magic.getObjectAtOffset(tib, 0));
13
14        if (type.isClassType()) {
15                  newObj = DVM_Allocator.allocateScalar(size, type.asClass(),allocator, hasFinalizer);
16
17                 if (newObj != null) {
18                           return newObj;
19                 }
20         }
21        newObj = VM_Interface.allocateScalar(size, tib, allocator);
22        //-#else
23        Object newObj = VM_Interface.allocateScalar(size, tib, allocator);
24        //-#endif
25
26         if (hasFinalizer) VM_Interface.addFinalizer(newObj);
27                  return newObj;
28 }
```

Figura 3.6: *VM Runtime*, metodo *resolvedNewScalar()*

```
1 public static Object allocateScalar(int size, VM_Class klass, int allocator, boolean hasFinalizer)
2                               throws VM_PragmaInline {
3
4          if (policy != null) {
5                  int node = policy.allocateNode(klass);
6                  if (node != -1 && node != nodeId) {
```

```
7                       Object obj = interfaces[node].allocateScalar(size,
8                       klass.getDictionaryId(), allocator, hasFinalizer);
9                       return obj;
10              }
11      }
12      return null;
13 }
```

Figura 3.7: *DVM Allocator*, método *allocateScalar()*

```
1 public Object allocateScalar(int size, int typeId, int allocator, boolean hasFinalizer)
2               throws VM_PragmaInline {
3
4       VM_Type type = VM_ClassLoader.clusterGetInitializedType(typeId);
5       Object[] tib = (Object[])VM_Statics.getSlotContentsAsObject(type.getTibSlot());
6
7        Object obj = VM_Interface.allocateScalar(size, tib, allocator);
8        if(hasFinalizer) { VM_Interface.addFinalizer(obj); }
9        return obj;
10 }
```

Figura 3.8: *DVM LocalAllocator*, metodo *allocateScalar()*

3.6.2. Recolección de basura.

La dJVM puede ser construida tanto sin recolección de basura (opción noGC) como con una adaptación del recolector de basura JMTk *Mark Sweep*. Este recolector de basura no es todavía completamente distribuido. El espacio ocupado por los objetos, que son directa o indirectamente referenciados desde otros nodos, no puede ser liberado. Solo el espacio ocupado por objetos que son referenciados solamente dentro del mismo nodo pueden ser liberados.

El código de este recolector de basura se encuentra en *../src/vm/memoryManagers/JMTk/plan/distMarkSweep/*. Las dos clases *Plan* y *Header* han sido adaptadas de clases con el mismo nombre en el directorio *markSweep/* del mismo nivel.

La clase *Header* en este directorio es prácticamente idéntica a la de la versión no distribuida. Los cambios a la clase *Plan* son mostrados en la Figura 3.9. Las diferencias con la clase *Plan* no distribuida, que consisten en evitar que objetos que son referenciados desde otros nodos sean recolectados, están en las líneas 4 y 11.

```
1 public static final VM_Address traceObject(VM_Address obj) {
2       VM_Address addr = VM_Interface.refToAddress(obj);
3        // don't trace across node boundaries
4       if (! addr.isRemote()) {
5               if (addr.LE(HEAP_END)) {
6                       if (addr.GE(MS_START))
7                               return msCollector.traceObject(obj);
8                       else if (addr.GE(IMMORTAL_START))
9                               return Immortal.traceObject(obj);
10                      }
11              } // else this is not a heap pointer
12              return obj;
13      }
```

Figura 3.9: clase JMTk *Plan*, método *traceObject()*

Hemos intentado modificar el colector JMTk Mark Sweep de la versión distribuida de tal forma que pudiese seguir las referencias remotas a otros objetos. Para ello, hemos intentado acceder al colector de basura local de cada nodo incluyendo métodos en las clases siguientes:

```
/** Añadido por Rafael Guijarro Crespo, Luis Jarabo de la Llana
*   y Pablo Molina García
*/
public boolean isRemote(VM_Address obj){
        try {
                //VM.sysWriteln("Comprobando dentro de isRemote si la direccion es remota...");
        VM_Address addr = VM_Interface.refToAddress(obj);
        //VM_Address addr = VM_Magic.objectAsAddress(obj);
        boolean isRemote = addr.isRemote();
                if(isRemote) {
        //  VM.sysWriteln("isRemote:La direccion es remota.");
        }
                else {
        //  VM.sysWriteln("isRemote:La direccion no es remota.");
        };
        return isRemote;
        }
        catch(NullPointerException e) {
                //VM.sysWriteln("Error en isRemote");
        //VM.sysWriteln(e.printStackTrace());
        return true;
        }
}

public Plan getPlan(){
        return (Plan) VM_Interface.getPlan();
}
```

Figura 3.9.1: Métodos añadidos en DVM_LocalAllocator.java

Después de añadir dichas modificaciones en la clase DVM_LocalAllocator (así como una serie de modificaciones en el método traceObject anterior que no detallaremos) para que se pudiese acceder al Plan de cada nodo, y así poder trazar la referencia del objeto, vimos que los interfaces que nos dan acceso a los nodos, una vez inicializados y construidas las estructuras necesarias en cada nodo, eran liberados, de forma que no podían ser accedidos en la fase de recolección de basura.

Tras varios estudios y tentativas sobre otro posible método para acceder al colector de basura local de cada nodo, descubrimos la forma de saber en que nodo se encuentra el objeto, de forma que no nos hacía falta ir recorriendo todos los nodos para ver en cual de ellos el objeto no era remoto para allí invocar al método traceObject de Plan (que son exactamente las modificaciones que antes no hemos mencionado). Una vez que sabíamos el nodo en el cual estaba el objeto, intentamos acceder directamente al Plan a través de la clase VM_Interface, pero el problema surgía porque el plan solamente estaba inicializado en el nodo 0, que junto con el comentario que incluíamos que nos decía en que nodo se estaba llevando a cabo la recolección de basura, nos ha llevado a la conclusión de que solamente hay un colector de basura situado en el nodo maestro, y cuando encuentra una referencia a un objeto situado en otro nodo, evita trazar la referencia en el otro nodo, pues no existe un colector de basura en dicho nodo, y por tanto no se puede trazar la referencia. La última versión de código correspondiente al método traceObject que hemos usado se puede ver en la figura 3.9.2:

```
/** Añadido por Rafael Guijarro, Luis Jarabo y Pablo Molina */
import com.ibm.JikesRVM.DVM_CommunicationManager;
import com.ibm.JikesRVM.DVM_LocalAllocator;
import com.ibm.JikesRVM.DVM_Allocator;
import com.ibm.JikesRVM.DVM_UIDManager;
import com.ibm.JikesRVM.VM_Type;
import com.ibm.JikesRVM.VM_TypeDictionary;
import com.ibm.JikesRVM.VM_Scheduler;
import com.ibm.JikesRVM.VM_Processor;
……..
public static final VM_Address traceObject(VM_Address obj) {
        VM_Address addr = VM_Interface.refToAddress(obj);
        int nodo= DVM_UIDManager.find(obj).getNode();
        //int nodoActual= DVM_CommunicationManager.getNodeId();

        //if (nodoActual!=0){
                //VM.sysWriteln("Plan.traceObject: Estamos en el nodo:  "+nodoActual);
                //VM.sysWriteln("Plan.traceObject: El objeto esta en el nodo:  "+nodo);
        //}
        // don't trace across node boundaries
        if (nodo == DVM_CommunicationManager.getNodeId()) {
                //VM.sysWriteln("Plan.traceObject: Objeto no remoto:  "+nodo);
        if (addr.LE(HEAP_END)) {
                        if (addr.GE(MS_START)){
                //VM.sysWriteln(obj.toString());
                                return msCollector.traceObject(obj);
                        }
                        else if (addr.GE(IMMORTAL_START))
                                return Immortal.traceObject(obj);
        }
        } // else this is not a heap pointer
        else {
                //VM.sysWriteln("Plan.traceObject: Estamos en el nodo:
                "+DVM_CommunicationManager.getNodeId());
                //VM.sysWriteln("Plan.traceObject: El objeto esta en el nodo:  "+nodo);
                //VM_Interface.showPlans();
                //VM.sysWriteln("Total de procesadores: "+VM_Scheduler.processors.length);
                VM_Processor p = VM_Scheduler.processors[nodo];
                Plan plan = VM_Interface.getPlanFromProcessor(p);
                //VM.sysWriteln("Comprobando el plan");
                if (p == null){
                        VM.sysWriteln("El plan del nodo "+nodo+" es null");
                }
                else {
                        VM.sysWriteln("Trazando el objeto en el nodo:  "+nodo);
                        return plan.traceObject(obj);
                }
        }
        return obj;
}; // Fin del método traceObject(obj)
```

Figura 3.9.2: Última versión del método traceObject() usada durante las pruebas

3.7. Código de VM y dJVM relacionado con planificación y otros aspectos de hilos.

3.7.1. Planificación.

El sistema de planificación provee de la infraestructura para la gestión del procesador y los hilos en JikesRVM. Las fuentes se encuentran en los directorios:

$Base/Jikes/rvmRoot/rvm/src/vm/scheduler
$Base/Jikes/rvmRoot/rvm/src/vm/cluster/scheduler

Hay dos intereses principales en el sistema de planificación, el arranque y los hilos remotos.

Arrancar JikesRVM construye las estructuras de datos de tiempo de ejecución necesarias para soportar programas multi-hilo, ver *VM_Scheduler.java*. Una vez estas estructuras de soporte están en su lugar la conmutación de hilos es permitida y Jikes RVM funciona con normalidad.

Un único hilo principal, *MainThread*, es creado y programado para la ejecución en JikesRVM antes de habilitar la conmutación de hilos. En la vesión de JikesRVM distribuida un único hilo principal es iniciado solo en el nodo maestro del cluster. Los mecanismos de comunicaciones dependen de la ejecución multi-hilo para soportar la distribución. De ese modo el cluster solo puede ser iniciado después de que la ejecución multi-hilo sea habilitada.

Un hilo puede recorrer las máquinas que participan en la JikesRVM distribuida. Su pila puede estar distribuida a través de las máquinas. Un hilo que se esta ejecutando en una máquina tiene un recurso de hilo local, su identidad es el hilo que fue creado originalmente, ver *java.lang.Thread* y *VM_Thread.Java.* Una identidad constante es mantenida por un hilo por dos razones:

1. Uso de recursos-una cadena de llamadas de un hilo que vuelve a visitar una máquina debe usar el mismo recurso local de hilo
2. Cierre – un hilo es autorizado a adquirir un cerrojo más de una vez antes de liberarlo.

La manipulación de identidad de hilo es manejada por los mensajes enviados entre máquinas, ver sección 3.2.

La figura 3.10 muestra el flujo de información y los puntos de decisión en la ejecución remota de un método. Las líneas finas continuas indican el paso de información a través de llamadas a métodos, las líneas discontinuas indican paso de información a otro hilo. Un hilo de aplicación debe ser ejecutado con la misma identidad de hilo, de esta manera, la referencia del hilo y el tipo deben de ser comunicados. En la recepción, es establecido un mapeado entre identidades de hilo locales y globales, si todavía no existe uno. Para hacer esto el tipo del hilo global debe de ser sabido, si no lo es, entonces el tipo debe ser cargado. Antes de que el hilo local ejecute el método debe desempaquetar los argumentos, y todos los tipos que no conoce deben de ser cargados; entonces el hilo local ejecuta el método. El método retorna normalmente y un resultado es enviado, o una excepción es enviada.

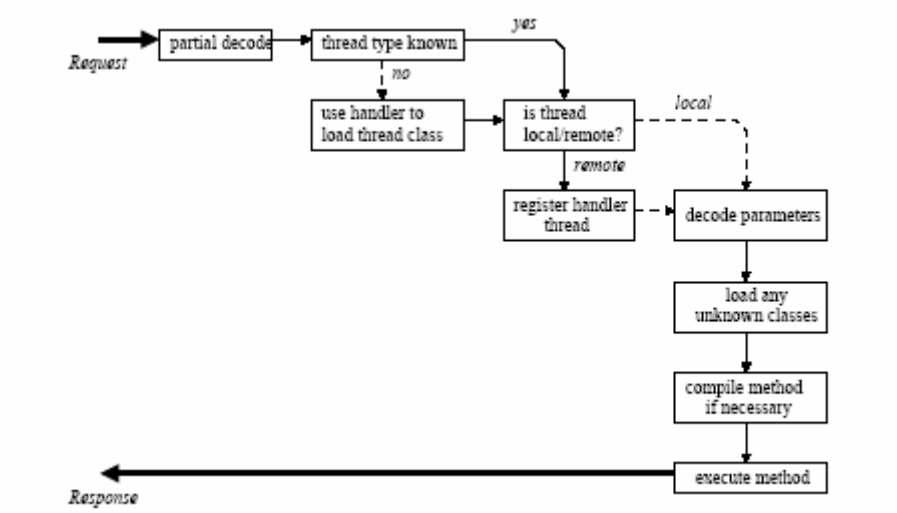


Figura 3.10 Procesamiento de una petición de ejecución.

Un hilo que fue creado en la maquina A puede hacer una llamada a método remoto a la máquina B. La máquina B maneja esta petición usando un manejador local de mensajes, este hilo manejador asume la identidad del hilo que lo llama. Si una serie de llamadas atraviesa las máquinas y vuelve a visitar la máquina B, entonces el hilo manejador ya ha sido asignado y el hilo asignado es interrumpido pasándole la petición. La asignación de hilos es manejada registrando un hilo antes de la ejecución y sacándolo del registro después de la ejecución, ver *DVM_ThreadResource.java.*

3.7.2. Terminación de programa.

La clase *DVM_Termination* es una implementación de una forma básica de *Task Balancing Algorithm* (algoritmo de balance de tareas). Esto asegura que el nodo maestro será capaz de decir cuando todos los hilos y todos los nodos han finalizado ejecución. En este punto un *DVM_messageControlInformCloseReq* es enviado a cada nodo esclavo. Cuando este es recibido, el nodo esclavo termina sus hilos *DVM_MessageReceiver*, envía un mensaje de acuse de recibo y entonces invoca *VM.sysExit().* Después que todos los mensajes de acuse de recibo son recibidos en el nodo maestro, sus hilos *DVM_MessageReceiver* son similarmente terminados y *VM.sysExit()* es invocado.

3.7.3. Cerrojos.

Jikes RVM encapsula el código de cierre en la clase *VM_Lock.java*. En una VM distribuida el cierre esta dividido en partes remotas y locales. Cuando una primitiva de cerrojo es llamada, debe ser determinado si el destino de la operación de cerrojo es mantenido local o remotamente. Si es local entonces es manejado como antes, de otro modo tiene lugar una llamada a un cerrojo remoto que será satisfecha o rechazada en esa máquina, ver *DVM_RemoteAccess.java* y *DVM_LocalLock.java*.

Similarmente a la ejecución de un método remoto un hilo manejador es usado para adquisición y liberación de cerrojos. Adquirir un cerrojo requiere que el hilo esté registrado, además, ese hilo local debe permanecer asociado con el global hasta que no haya cerrojos ni invocaciones pendientes. De ese modo, el hilo local permanece registrado después de que el cerrojo es obtenido, y solo deja de estar registrado cuando el cerrojo es liberado.

3.8. Código de dJVM relativo a Utilidades.

Los códigos fuente se encuentran en los directorios:
$Base/Jikes/rvmRoot/rvm/src/vm/utility
$Base/Jikes/rvmRoot/rvm/src/vm/cluster/utility.

El contenido de los dos directorios de utilidades provee funciones de soporte para JikesRVM y dJVM respectivamente.

El archivo *VM_CommandLineArgs.java* es usado para procesar las opciones de línea de comandos específicas de VM, por ejemplo los argumentos anteriores a la clase que contiene el método principal a ser ejecutado. Esta clase está modificada para incluir las opciones de línea de comandos descritas en la Sección 4.2.

Los archivos en *$Base/Jikes/rvmRoot/rvm/src/vm/cluster/utility* proveen de algunas clases auxiliares. Estas incluyen:

- *DVM_Checksum.java*
- *DVM_Logger.java*
- *DVM_Util.java*
- *DVM_Splay.template, DVM_SplayNode.template y DVM_SplayIterator.template* que proveen un template de arbol extendido y clases de utilidad.

La clase DVM_Checksum usa el algoritmo Adler32 para calcular sumas de control. Es usado para confirmar la consistencia de los diccionarios, tablas estáticas y definiciones de clases entre todos los nodos como es descrito en la Subsección 3.3.4.

La clase DVM_Util contiene métodos estáticos, relacionados todos ellos con arrays de tipos primitivos. Los siguientes métodos son usados para dar formato de cadena de caracteres al contenido de los *arrays*, para poder ser impresos: *intArrayToString*(), *longArrayToString*() y *byteArrayToString*().

Los siguientes métodos de DVM_Util usan arrays para crear y manipular conjuntos de enteros: *addToArraySet(), quadrupleSizedIntArray(), createArraySet(), arraySetSize() y arrayWithNoEmptyElements().* Estos métodos son usados a fin de que la sobrecarga requerida por el uso de clases como /*java/util/HashSet* pueda ser evitada durante los procedimientos de arranque.

La clase *DVM_Logger* es un dispositivo de logging, que permite determinar la prolijidad de *logging output* por su campo estático *debugLevel.* Los niveles de depuración en orden descendente de prolijidad son: *DEBUG, INFO, WARN, ERROR y SILENT*. Cada método toma un gran número de argumentos por lo que no será necesario construir mensajes a partir de cadenas de caracteres concatenadas. Esto evitará la creación de nuevos objetos, que podrían degradar el rendimiento de la JVM distribuida, debido al hecho de que no están todavía disponibles mecanismos efectivos de recolección de basura a través del cluster. Una plantilla de árbol extendido es definida en *$Base/Jikes/rvmRoot/rvm/src/vm/cluster/utility*. Las funciones de plantilla son parecidas a las plantillas usadas para la clase *VM_Dictionary,* donde un preprocesador es usado para sustituir ocurrencias predefinidas de cadena de caracteres para nombres de tipos. Esto provee una forma de polimorfismo que permite que sean generados nuevos tipos que son semánticamente idénticos, pero almacenan y manipulan diferentes tipos de datos incluyendo tipos primitivos.

Los tipos generados son definidos en *Makefile*. El archivo *DVM_UIDManager.java* usa un

mapeo *forward and reverse* de identificadores globales UID y referencias locales *Object*. El mapeo de UID a *Object* es generado expandiendo las plantillas:

- *DVM_Splay.template*- la base del árbol extendido.
- *DVM_SplayNode.template*- un nodo en un árbol extendido.
- *DVM_SplayIterator.template*- un iterador que puede ser usado para el árbol extendido

El mapeo invertido es definido en *$Base/Jikes/rvmRoot/rvm/src/vm/cluster/external.*

4. Uso.

4.1. Introducción.

Este capítulo describe como ejecutar dJVM con sus distintas opciones de línea de comandos. Muchas de las opciones de la versión no distribuida pueden ser usadas también en la versión distribuida, sin embargo otras no pueden ser usadas o simplemente no tienen efecto. Por ejemplo, las opciones relacionadas con optimizaciones del compilador no pueden ser usadas debido a que las modificaciones necesarias para su correcto funcionamiento con dJVM no han sido hechas. A continuación se muestran con más profundidad cuales de las opciones están permitidas.

Al final del capítulo se describen una serie de aplicaciones y programas de prueba que han sido incluidas en esta distribución.

4.2. Ejecutando dJVM.

Asumiendo que el entorno ha sido correctamente configurado como se describe en la sección 2.2, dJVM puede ser ejecutado simplemente poniendo el comando *rvmCluster* en lugar de rvm. Por ejemplo, para ejecutar la aplicación *TestRunner* usando dJVM, escriba:

rvmCluster TestRunner

El script también necesita un fichero de configuración que contiene la lista de direcciones IP de los hosts que componen el cluster dJVM. Cada host debe tener además instalado dJVM y el entorno del shell debe ser configurado de forma correcta como se describe en la sección 2.2. En cada host debe también existir un directorio con un nombre que concuerde con el directorio de trabajo actual, que es, el que resulte de quitar el *${HOME}* del principio en ambos.

```
# Fichero de configuración 1 de rvmCluster
192.168.1.4     pila03
192.168.1.3     pila02
192.168.1.2     pila01

# Fichero de configuración 2 de rvmCluster
192.168.1.4
192.168.1.3
192.168.1.2
```

Figura 4.1. Ficheros de configuración

Si no existe dicho fichero de configuración, *rvmCluster* simplemente invoca a *rvm* pasándole las opciones introducidas en la línea de comandos.

Dos ficheros de configuración distintos se muestran en la figura 4.1. Cada uno define un cluster pequeño que contienen los tres mismos hosts. El primero incluye campos de comentarios y campos de registros, mientras que el segundo no.

Las identidades de los nodos son asignadas en el orden en que son listadas en el fichero, empezando por el 0. El nodo 0, o nodo maestro, es el primer host (192.168.1.4). Esta dirección debe ser desde la que se ejecuta el comando. Los dos nodos esclavos, el nodo 1 y el nodo 2 son respectivamente los nodos con direcciones 192.168.1.3 y 192.168.1.2. Si en el fichero de

configuración aparece dos veces la misma dirección entonces se lanzarán dos procesos en ese host.

Por defecto, el fichero de configuración se llama *DJVMHosts* y deberá estar en el directorio de trabajo actual.

El comando mostrado arriba, si se ejecuta con el fichero de configuración descrito arriba, generará tres líneas de comando, cada una de las cuales será ejecutada en su nodo designado. La primera será invocada localmente en el nodo maestro para iniciar el proceso JVM maestro:

rvm -X:dvm:nodeid=0 -X:dvm:nodecount=3
-X:dvm:listen=192.168.1.4 -X:dvm:portpipe=ListeningPort.0 TestRunner

Las otras dos líneas de comandos serán cada una ejecutadas remotamente en los respectivos nodos esclavos:

rvm -X:dvm:nodeid=1 -X:dvm:listen=192.168.1.3
-X:dvm:contact=192.168.1.4:35073

rvm -X:dvm:nodeid=2 -X:dvm:listen=192.168.1.2
-X:dvm:contact=192.168.1.4:35073

-X:dvm:nodeid=0 : Identificación del nodo. Debe ser 0 en el nodo maestro.
-X:dvm:nodecount=0 : Tamaño del cluster.
-X:dvm:listen=192.168.1.4: La dirección IP del host en el que se esta ejecutando. Esta opción se usa en todos los nodos.
-X:dvm:portpipe=ListeningPort.0: El nombre del fichero FIFO en el que esta escrito el numero del puerto de escucha. El numero de puerto en este fichero se le pasa a todos los procesos esclavos usando *-X:dvm:contact* como se describe más abajo.
TestRunner: Nombre de la aplicación. Cualquier argumento de la clase que se le pase a rvmCluster debe ir detrás del nombre de la clase.
-X:dvm:nodeid=1 : Identificación del nodo. Debe ser 1,2,... para los nodos esclavos.
-X:dvm:listen=192.168.1.3: La dirección IP del host en el que se esta ejecutando. Esta opción se usa en todos los nodos.
-X:dvm:contact=192.168.1.5:35073: El puerto de escucha creado por el proceso maestro. Este argumento es el mismo para todos los procesos hijos.

La línea de comandos que genera el proceso maestro es generada por *rvmCluster* y luego ejecutada como un proceso en background. El proceso maestro entonces establece un puerto de escucha que se escribe en el fichero FIFO especificado por la opción *-ListenPortFile=<filename>*(por defecto con nombre ListeningPort.0). El script genera entonces las líneas de comando que generarán los procesos esclavos usando ese número de puerto de escucha. Los procesos esclavos son invocados remotamente usando *rsh* o *ssh*, dependiendo del valor de la variable de entorno RSH, o de si el flag -UseSsh ha sido usado. *rvmCluster* entonces espera entonces a que termine el proceso maestro.

Cada proceso esclavo establece su propio puerto de escucha usando el número de puerto TCP/IP especificado. La dirección de cada puerto entonces se manda al nodo maestro y, a través de éste, a los demás nodos esclavos durante el proceso de arranque.

dJVM puede ser arrancado explícitamente insertando los comandos mostrados

anteriormente en los diferentes hosts. El proceso maestro debe ser arrancado antes de que se arranquen los procesos esclavos, para que el puerto de escucha del nodo maestro pueda ser obtenido del fichero FIFO. Esto se haría de la siguiente forma:

```
# rvm -X:dvm:nodeid=0 -X:dvm:nodecount=3 -X:dvm:listen=192.168.1.4
 -X:dvm:portpipe=ListeningPort.0     TestRunner
# cat ListeningPort.O
35073
```

El número de puerto 35073 puede así ser usado para construir manualmente los nodos esclavos en línea de comandos como se ha descrito anteriormente.

La sintaxis completa de la línea de comandos puede obtenerse usando rvmCluster –h, aunque es la siguiente:

```
Usage: rvmCluster [-UseSsh] [-DJVMHostsFile=<fileName>] [-
ListenPortFile=<filenmae>}\
[[JikesRVM arg] ...]\
<application class> [[application arg] ...]; or

rvmCluster (-h|-help|--help)

Note: it is assumed that the first argument not beginning
with a '-' is the application class.
```

En muchos casos es posible omitir los argumentos *–UseSsh*, *-DJVMHostsFile=* y *–ListenPortFile=*, ya que solo son usados en el script y no se le pasan a *rvm*. Al igual que en la invocación de otras máquinas virtuales java, los argumentos de la máquina virtual siempre preceden al nombre de la clase, mientras que los argumentos de la aplicación van detrás del nombre de la clase.

La opción *-DJVMHostsFile=<filename>* puede ser usada para sobrescribir la configuración por defecto.

4.3. Ejemplo de ejecución de rvmCluster.

La forma más simple de uso se describe en la figura 4.2, en nuestro caso realizada en *aneto*, cuyas características se describirán en el apéndice A.

El fichero *hostsfile* aquí especifica un proceso maestro (en *pila04*), y tres procesos esclavos (en *pila03*, *pila02* y *pila01*). Este fichero ha sido obtenido a partir del script *resolver.sh* que se detallará más adelante, y usando un sistema de reserva de nodos y encolado de trabajos instalado en aneto que también se detallará más adelante. Para este ejemplo, hemos usado la aplicación *WorkerFarmTimedIntegrate* que integra la función identidad en el intervalo [0,1] usando un paso de integración de 0.0001.

```
#[jikes2@aneto]~ cd $RVM_ROOT/rvm/src/examples/cluster/appn
#[jikes2@aneto]~ jikes WorkerFarmTimedIntegrate.java
#[jikes2@aneto]~ cd $HOME/proyecto
#[jikes2@aneto]~ qsub –I –l nodes=4:ppn=1
#[jikes2@pila04]~ cd proyecto
```

```
#[jikes2@pila04]~ ./resolver.sh >hostsfile
#[jikes2@pila04]~ cat hostsfile
192.168.1.5
192.168.1.4
192.168.1.3
192.168.1.2
#[jikes2@pila04]~ cd $RVM_ROOT/rvm/src/examples/cluster/appn
#[jikes2@pila04]~ rmvCluster –DJVMHostsFile=$HOME/proyecto/hostsfile
WorkerFarmTimedIntegrate 0 1 .0001
```

Figura 4.2. Ejemplo de ejecución via rvmCluster

Un ejemplo de la salida producida por este comando se muestra en la figura 4.3. El proceso maestro crea 4 hilos que son ubicados en los hosts indicados por el fichero *hostsfile*, usando la política por defecto que es round-robin e invoca su método start(). Las líneas finales demuestran el uso de memoria en el hilo maestro antes y después de la recolección de basura.

```
...
Starting worker threads
Creating worker thread 0
WorkerThread(0): 2
Started worker thread 0
Creating worker thread 1
WorkerThread(1): 3
Started worker thread 1
Creating worker thread 2
WorkerThread(2): 0
Started worker thread 2
Creating worker thread 3
WorkerThread(3): 1
Started worker thread 3
Calc: from 0.5 to 0.75 = 0.156312
Calc: from 0.0 to 0.25 = 0.0312625
Calc: from 0.25 to 0.5 = 0.0937875
Calc: from 0.75 to 1.0 = 0.218837
Results = 0.5002
used pages = 5890 (23.00 Mb)= (ms) 5858 (22.88 Mb) + (imm) 32 (0.12 Mb)
used pages = 2747 (10.73 Mb)= (ms) 2715 (10.60 Mb) + (imm) 32 (0.12 Mb)
```

Figura 4.3. Salida producida por WorkerFarmTimedIntegrate

En los otros nodos, el la línea de comandos *rvm* ejecutada será:

```
rvm -X:dvm:nodeid=1 -X:dvm:listen=192.168.1.4 -X:dvm:contact=192.168.1.5:35073
rvm -X:dvm:nodeid=2 -X:dvm:listen=192.168.1.3 -X:dvm:contact=192.168.1.5:35073
rvm -X:dvm:nodeid=3 -X:dvm:listen=192.168.1.2 -X:dvm:contact=192.168.1.5:35073
```

4.4. Opciones de dJVM.

Las siguientes opciones descritas en esta sección, son pasadas directamente al ejecutable de JikesRVM o indirectamente a través de los scripts *rvm* y *rvmCluster*.

4.4.1. Comunicación.

Cada máquina virtual que compone el cluster usa la misma imagen inicial. El rol de cada máquina virtual, tanto maestro como esclavo, es determinado por la línea de comandos que se le pasa a cada máquina virtual. Estos argumentos son:

- *-X:dvm:listen:<mi_direccion_ip>* : Especifica qué dirección IP tiene que usar la máquina que ejecuta la máquina virtual.
- *-X:dvm:portpipe=<nombre_puerto_escucha>* : Usado solamente en el nodo maestro. Especifica el nombre del fichero FIFO creado por el proceso maestro para grabar el puerto de escucha del nodo maestro, y que se requiere para invocar a los procesos esclavos.
- *-X:dvm:contact=<direccion_ip_maestro>:<puerto>* : Usado solamente en los nodos esclavos. Especifica la dirección IP del nodo maestro y su puerto de escucha.
- *-X:dvm:nodecount=<n>* : Usado solamente en el nodo maestro. Especifica el número total de nodos en el sistema. Esta información se le envía a todos los nodos esclavos durante el arranque.
- *-X:dvm:nodeid=<id>* : Especifica el número de identificación del nodo, de forma que id=0 para el nodo maestro, y $1 \leq id \leq n$ para los procesos esclavos.
- *-X:dvm:substrate=<metodo_comunicacion>* : Especifica el tipo de capa de comunicación usada. Actualmente solo esta disponible *socket*, y por tanto este es el valor por defecto.

Nótese además, que todas las direcciones IP están especificadas en el formato x.x.x.x y no por el nombre de la máquina.

4.4.2. Asignación.

El sistema actualmente permite diferentes asignaciones en tiempo de ejecución. Inicialmente, todos los objetos son asignados localmente, y, cuando las conexiones del cluster están establecidas, se habilita la asignación remota. Por defecto, la política de asignación es round-robin, pero puede ser sobrescrita con la opción:

- *-X:dvm:allocation=<política_asignacion>*

donde las posibles opciones son:

- *round-robin* : Usa el asignador round robin. Cada asignación sucesiva de una clase que implemente *java.lang.Runnable* se realiza en el siguiente nodo con identidad más alta, o en el nodo 0, si el último nodo ha sido alcanzado ya.
- *random* : Usa el asignador aleatorio. Cada asignación de una clase que implemente *java.lang.Runnable* es realizada en un nodo cuya identidad es determinada de forma aleatoria.
- *local* : Usa el asignador local. Todas las asignaciones de objetos se realizan de forma local.

4.4.3. Escritura de información de depuración en ficheros locales.

La capacidad de escritura de la salida de depuración en ficheros locales puede ser controlada con el siguiente parámetro:

- *-X:dvm:localwrite=[o[(+|-)]][,[m[(+|-)]]]*

Por defecto los mensajes no se registran, mientras que otras salidas de depuración si que escriben. Los signos '+' o '-' indican si una opción esta activada o desactivada, aunque el '+' puede ser omitido.

4.4.4. Transformaciones.

Las acciones de creación del proxy y de invocación de transformadores de clases del proxy puede ser monitorizada, si el sistema ha sido construido con los diagnósticos habilitados. Estos flags son:

- *-X:dvm:trace:proxy:generation=<bool>* : Traza la generación de los métodos del proxy.
- *-X:dvm:trace:proxy:ejecution=<bool>* : Traza la transformación de operaciones de invocación para ejecuciones condicionales a través del proxy o para invocaciones sin proxy.

Estas operaciones de transformación no pueden ser deshabilitadas en línea de comandos.

4.4.5. Acceso.

El acceso a datos remotos puede ser trazado en dJVM solo si el sistema ha sido compilado con los diagnósticos habilitados. Para esto se proporcionan los siguientes flags:

- *-X:dvm:trace:access=<bool>* : Traza el acceso a instancias y campos de clases, peticiones de bloqueos, elementos de arrays y longitudes de arrays.
- *-X:dvm:trace:uid=<bool>* : Traza la creación y eliminación de entradas de mapeado UID.

4.4.6. Compilación.

El código generado para la operación de forma distribuida puede ser también monitorizado utilizando la siguiente opción de línea de comandos:

- *-X:dvm:trace:compile=<bool>*

4.4.7. Inicialización.

Un determinado número de operaciones son realizadas durante el arranque de la máquina virtual. Entre ellas esta la carga de clases esenciales para la comunicación y las operaciones del cluster. También el mapeado de información de tipo local al tipo de información en el nodo maestro. Esto puede ser monitorizado usando la siguiente opción:

- *-X:dvm:dump:initial:uid=<bool>* : Vuelca la lista completa de tipos y de entradas del diccionario usadas hasta el arranque del cluster (NOTA: Esta lista puede tener una gran extensión).

4.5. Descripción de los programas de prueba y de las aplicaciones.

Esta sección contiene una descripción de los programas de prueba creados para dJVM con el propósito de probar diferentes aspectos de JVM que involucren la ejecución distribuida. Se dividen en dos categorías: Programas para probar la funcionalidad de diferentes mecanismos, y programas de aplicación diseñados para probar el sistema en su conjunto y para proporcionar algunos resultados de prueba de rendimiento.

Otros programas de prueba, que sirvieron para un determinado propósito durante el proceso de desarrollo, pero que no se podrían ejecutar sin una modificación sustancial no han sido incluidos en esta estructura de directorios. Además algunos aspectos de testeo pueden estar duplicados en distintos programas.

Para más información consultar los ficheros README adjuntos o usar la opción en línea de comandos –h o –help.

Los programas de prueba se encuentran en la ruta:

$RVM_ROOT/rvm/src/examples/cluster

Para compilar los programas de prueba es necesario primero construir dJVM, ya que determinadas partes de los programas de prueba usan clases específicas de JikesRVM. Todos los programas de prueba pueden ser construidos usando el compilador de IBM para Java, Jikes, con solo ir a directorio donde están los programas fuente y escribir:

jikes <fichero> [fichero…]

Puede que sea necesario indicar la ubicación del CLASSPATH con la opción –classpath=<ruta> en el caso de que la variable de entorno no haya sido correctamente definida como se indica en la instalación en el apéndice B.

4.5.1. Probando dJVM bajo carga.

El programa *ClusterTestRunner* se puede encontrar en *$RVM_ROOT/rvm/src/examples/cluster /loadtest.*

Un gran número de hilos *productores* ‘producen’ cuando incrementan una serie de campos de objetos *conducto*. Un solo hilo *consumidor* ‘consume’ de estos objetos conducto.

La ejecución termina cuando el campo contador de los hilos consumidores es mayor o igual que el valor esperado, y todos los campos contador en los objetos conducto son iguales a cero. Si el campo contador de los consumidores no encaja con el valor esperado, el programa de prueba ha fallado.

Para ejecutar este programa, se requiere un gran número de cadenas de invocaciones remotas de métodos, abarcando hasta cinco nodos separados. Hay también una gran cantidad de contenciones por el bloqueo de los objetos conducto que hacen tanto los objetos productores como los consumidores.

ClusterTestRunner puede ser ejecutado con el siguiente comando:

rvmCluster ClusterTestRunner –multipliers=,3,2,2,4,8

Por conveniencia todos los argumentos han sido unidos, como "sub-argumentos", en un solo argumento más largo. Si omitimos alguno, se usan los que tiene definidos el programa por defecto. Estos sub-argumentos son:

1. El número de invocadores primarios, cada uno de los cuales creará un determinado número de invocadores secundarios. Por defecto se usa uno por cada nodo.
2. El número de invocadores secundarios por cada invocador primario. Cada uno de ellos creará a su vez un determinado número de invocadores terciarios.
3. El número de invocadores terciarios por cada nodo secundario, cada uno de los cuales creara un número de hilos productores y arrancará sus métodos start( ).
4. El número de lanzadores de productores creados por cada invocador terciario.
5. El número de hilos productores creados por cada lanzador de productores.
6. En cuantos objetos conductos del número total de objetos conducto (uno por nodo) producirá cada productor. Sin esta limitación, la cantidad de trabajo debería de verse incrementada en un factor de n^2, mientras que el número de nodos se incrementa en un factor de n.

```
jikes2@pila08:~/proyecto/DJikesRVM/rvmRoot/rvm/src/examples/cluster/loadtest$ rvmCluster -
DJVMHostsFile=$HOME/proyecto/hostsfle ClusterTestRunner
-multipliers=8,2,2,2,2,5

    starting slaves 1 2 3 4 5 6 7
    Multiplier values : -multipliers=8,2,2,2,2,5

    Test Program will create:
            first tier invokers      : 8
            second tier invokers     : 16
            third tier invokers      : 32
            producer starter threads : 64
            producer threads         : 128

    Each producer thread will produce 9 times into 5 conduits
    randomly selected from a total of 8 conduits

    This will be consumed by a single consumer thread.
    The expected total to be both produced and consumed is : 5760

    Consumed count is 5760
    vm: exit 143
```

Figura 4.4. Salida producida por ClusterTestRunner

Aquí se muestra la salida de la ejecución del programa. La última línea muestra que se ha devuelto el código 143. En el JVM del nodo maestro, ha sido establecido a este valor cuando se ha encontrado que el contador de consumidos encajaba con lo esperado. Esto es necesario debido a que

en un determinado punto, cuando la cantidad de salida de depuración es muy grande, puede ser muy difícil o incluso imposible encontrar si el programa ha terminado correctamente o no.

4.5.2. Aplicaciones.

El subdirectorio *$RVM_ROOT/rvm/src/examples/cluster/appn* contiene una serie de aplicaciones simples.

- *HelloWorld.java* : Es un programa que crea hilos y hace una serie de manipulaciones con objetos y sincronización entre hilos. Habilita la opción de trazado de dJVM, de forma que una ejecución correcta muestra el número de líneas trazadas (precedidas de *HelloWorldThread. *():*), seguidas de la cadena *Hello World Done*.

- *WorkerFarmIntegrate.java* : Es un programa que calcula una simple integración numérica de la función identidad entre los intervalos determinados por los dos primeros argumentos que son de tipo double float. El tercer argumento, también de tipo double float, es el paso de integración. El cómputo se inicia en el hilo principal, que crea *n* hilos esclavos para procesar parte de la integral, donde *n* es el número de procesos dJVM. Esto representa un uso muy simple de dJVM.
  La versión actual también imprime el uso de memoria antes y después de que se lleve a cabo la invocación manual del colector de basura.

- *WorkerFarmTimedIntegrate.java* : Es una extensión del programa anterior. Éste toma dos parámetros adicionales: *r* (número de veces que se llevará a cabo la integración) y *p* (número de nodos esclavos creados, que puede diferir del número de procesos dJVM). Usando la clase *Timer* del archivo *Timer.java* muestra una estadística detallada del tiempo de varias partes del proceso, que pueden ser usadas para una evaluación del rendimiento de dJVM, por ejemplo, para deducir la sobrecarga de la llamada a un método remoto.

4.5.3. Programas de prueba de acceso a datos.

Hay varios programas de prueba de acceso a datos remotos en el directorio *$RVM_ROOT/rvm/src/examples/cluster/access* :

- *TestStatic.java* : Prueba la lectura y escritura de diferentes tipos de variables estáticas desde el nodo maestro a nodos esclavos. Los resultados se comparan con los valores predeterminados.
- *TestField.java* : Prueba la lectura y escritura de diferentes campos de tipos tanto en el nodo maestro como en nodos esclavos. Se apoya en el uso de variables estáticas como una vía alternativa de comunicación, de forma que solo es válido si la prueba de variables estáticas es válida.
- *TestArray.java* : Es esencialmente igual que el *TestField.java* solo que este opera con arrays.

4.5.4. Miscelánea de programas de pruebas.

- *Hello World* : El “Hello World” trivial puede ser encontrado en el directorio

$RVM_ROOT/rvm/src/examples/cluster/helloworld . Este programa también puede ser ejecutado usando rvmCluster, pero puesto que solo tiene un hilo de ejecución, solo se ejecutará en el nodo maestro. Este programa no debe ser confundido con el Hello World de la sección 4.5.2.

- *Task Balancing* : Son programas para probar el algoritmo de balance de tareas, que asegura la terminación de todas la JVMs. Se pueden encontrar en el directorio:
  $RVM_ROOT/rvm/src/examples/cluster/taskbalancing

5. Benchmarks.

Para comprobar el rendimiento de esta aplicación hemos utilizado un benchmark típico de servidores de aplicaciones Java, cuya trabajo consiste básicamente en crear hilos en el nodo principal y mandarlos ejecutar; el servidor permanece a la escucha, y las distintas peticiones las ejecuta mediante un hilo en uno de los nodos exclamos. Utilizamos una política de asignación de tipo "Round-Robin" por lo que la distribución de hilos en los distintos nodos será uniforme. Esta aplicación esta compuesta por tres clases que son las siguientes:

ThreadAllocationTester.java: Clase principal a partir de la cual se crean los hilos primarios. A esta le podemos pasar una serie de parámetros los cuales nos determinaran el número de hilos primarios y secundarios que se crearan. La aplicación en principio crea un nº de hilos que depende directamente del número de nodos que hayamos reservado. Esta ha sido modificada para que se creen siempre el mismo número de hilos primarios y secundarios y así poder evaluar como afecta la cantidad de nodos al rendimiento.

```
import com.ibm.JikesRVM.VM;
import com.ibm.JikesRVM.VM_Magic;
import com.ibm.JikesRVM.DVM_Allocator;
import com.ibm.JikesRVM.DVM_AllocatorPolicy;
import com.ibm.JikesRVM.DVM_Out;
import com.ibm.JikesRVM.DVM_Cluster;

import com.ibm.JikesRVM.DVM_ClassMonitor;
import com.ibm.JikesRVM.DVM_Terminator;

public class ThreadAllocationTester {

  static PrimaryThread[]   threadSet;
  static int primaryThreadNo;
  static int secondaryThreadNo = 16;
  static boolean quiet = false;
  static boolean execute = true;

  public static void main(String[] args) {
    int ii, len;

    DVM_Terminator.assertClusterExecutionHasNotSucceeded();

    setParameters(args);

    if(!execute) return;

    if(!quiet) printInfoMessage();

    VM.sysWrite("Test program started. Find detailed output in local files\n");
    runTest();
    VM.sysWrite("Test program finished\n");
  }//Fin del método main

  public static void runTest () {
    int ii;

    DVM_Terminator.assertClusterExecutionHasNotSucceeded();
    DVM_Out._log("ThreadAllocationTester starting");


    threadSet = new PrimaryThread[primaryThreadNo];
    for(ii = 0; ii < threadSet.length; ii++) {
```

```
      DVM_Out._log("ThreadAllocationTester creating Primary Thread(id)", ii);

      threadSet[ii] = new PrimaryThread(ii, secondaryThreadNo);
    }



       DVM_Out._log("ThreadAllocationTester starting threads");
    for(ii = 0; ii < threadSet.length; ii++) {
      threadSet[ii].start();
      DVM_Out._log("ThreadAllocationTester thread started (id)", ii);
    }

    DVM_Out._log("ThreadAllocationTester joining threads");
    for(ii = 0; ii < threadSet.length; ii++) {
      try {
            threadSet[ii].join();
        DVM_Out._log("ThreadAllocationTester thread joined (id)", ii);
     } catch (InterruptedException ie) { }
    }

    DVM_Terminator.setClusterExecutionReturnCode(DVM_Terminator.TEST_SUCCEEDED);

  }//Fin del método runTest

  static void setParameters(String[] args) {
    String multipliers = null;
    int delimIdx;
    String intSubArg;
    String argKey;

    primaryThreadNo = 100;
    //primaryThreadNo = DVM_Cluster.getNodeCount() * 4;

    for(int ii = 0; ii < args.length; ii++) {
      String arg = args[ii];
      if(arg.startsWith("-multipliers=")) {
        argKey = "-multipliers=";
        multipliers = arg.substring(argKey.length());
        delimIdx = argKey.length();

        intSubArg = getSubArgument(arg, delimIdx);
        if(!intSubArg.equals("")) {
          primaryThreadNo = Integer.parseInt(intSubArg);
        }

        delimIdx += (intSubArg.length()+1);
        intSubArg = getSubArgument(arg, delimIdx);
        if(!intSubArg.equals("")) {
          secondaryThreadNo = Integer.parseInt(intSubArg);
        }
      } else if(arg.equals("-quiet")) {
        quiet = true;
      } else if(arg.equals("-no-execute")) {
        execute = false;
      } else if(arg.equals("-h") || arg.equals("-H") || arg.equals("-H")
                || arg.equals("-?") || arg.equals("-help") ||
                arg.equals("--help")) {
        showUsage();
        execute = false;
        return;
      } else {
        showUsage();
        execute = false;
        return;
      }
    }//Fin del for
  }//Fin de setParameters
}//Fin de ThreadAllocationTester
```

PrimaryThread.java: Esta clase compone un hilo el cual a su vez manda ejecutar una serie de hilos secundarios que serán los que lleven la carga de trabajo.

```
import com.ibm.JikesRVM.DVM_Out;
import com.ibm.JikesRVM.VM_Magic;

public class PrimaryThread extends Thread {
  int id;
  int noOfSecondaryThreads;

  public PrimaryThread(int anId) {
    id = anId;
    DVM_Out._log("PrimaryThread (id)",id);
  }

  public PrimaryThread(int anId, int aNoOfSecondaryThreads) {
    id = anId;
    noOfSecondaryThreads = aNoOfSecondaryThreads;
    DVM_Out._log("PrimaryThread (id,sec thr count)", id, noOfSecondaryThreads);
  }

  public void run() {
    for(int ii = 0 ; ii < noOfSecondaryThreads; ii++) {

      DVM_Out._log("starting sec thread (primaryId, id)", id, ii);
      SecondaryThread secThr = new SecondaryThread(ii, id);
      secThr.start();
    }
  }//Fin del run
}//Fin de PrimaryThread
```

SecundaryThread.java: Esta clase compone un hilo, el cual será el que lleve la carga de trabajo que se ejecutará en uno de los nodos (bien maestro, bien exclavo).

```
import com.ibm.JikesRVM.DVM_Out;
import java.lang.*;

public class SecondaryThread extends Thread {
  public int id;
  public int primaryId;

 public SecondaryThread(int anId, int aPrimaryId) {
    id = anId;
    primaryId = aPrimaryId;
    //DVM_Out._log("SecondaryThread created(primaryId,id)", primaryId, id);
  }

    public void run() {
       DVM_Out._log("SecondaryThread running(primaryId,id)", primaryId, id);
       //for (int l=0; l<1; l++){
       //     this.multiplicaMatriz();      Carga de trabajo
       //}

    }//Fin del mÃ©todo Run
}
```

5.1. Benchmark sin carga.

En primer lugar ejecutamos la aplicación sin ningún tipo de carga de trabajo; ello haría en teoría que cuantos más nodos haya peor rendimiento tengamos puesto que el tiempo de ejecución es prácticamente nulo, y por tanto al ejecutarla en la máquina distribuida el rendimiento de esta se vea degradado por el tiempo invertido en comunicaciones, y demás. Esta creará como hemos dicho en

todos los casos 100 hilos primarios y 1600 secundarios.

Los resultados obtenidos son los siguientes:

Resultados obtenidos cuando reservamos 8 nodos:
Tiempo Real: 98.500 segundos
Tiempo Usuario: 66.334 segundos
Tiempo sistema: 32.167 segundos

Resultados obtenidos cuando reservamos 7 nodos:
Tiempo Real: 43.293 segundos
Tiempo Usuario: 33.244 segundos
Tiempo sistema: 10.069 segundos

Resultados obtenidos cuando reservamos 6 nodos:
Tiempo Real: 42.783 segundos
Tiempo Usuario: 33.724 segundos
Tiempo sistema: 9. 068 segundos

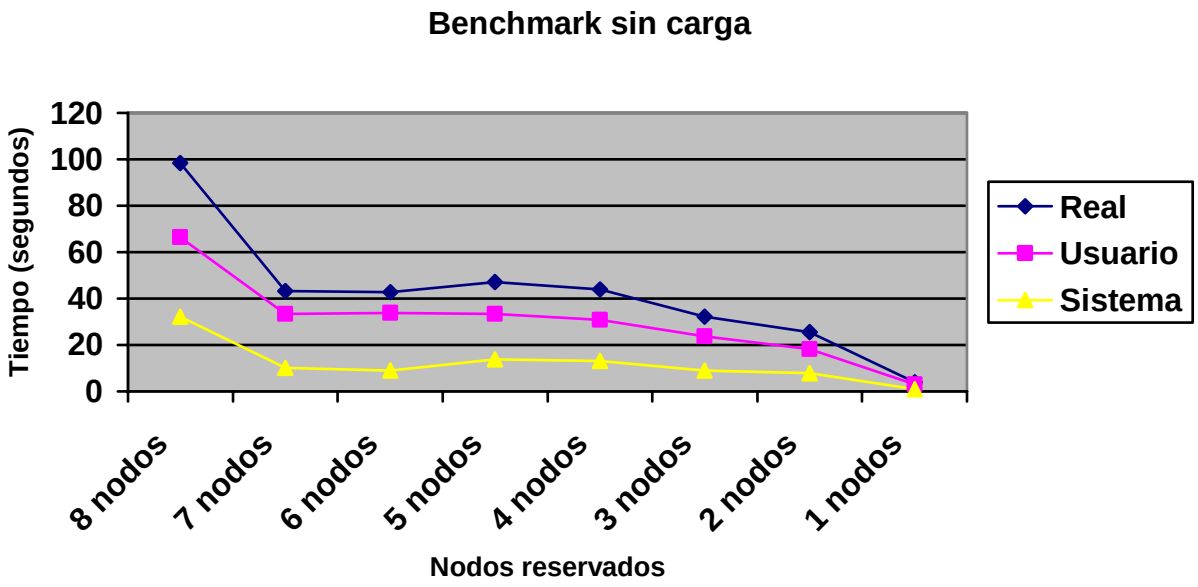
Resultados obtenidos cuando reservamos 5 nodos:
Tiempo Real: 47.130 segundos
Tiempo Usuario: 33.436 segundos
Tiempo sistema: 13.703 segundos

Resultados obtenidos cuando reservamos 4 nodos:
Tiempo Real: 43.951 segundos
Tiempo Usuario: 30.815 segundos
Tiempo sistema: 13.124 segundos

Resultados obtenidos cuando reservamos 3 nodos:
Tiempo Real: 32.336 segundos
Tiempo Usuario: 23.729 segundos
Tiempo sistema: 8.494 segundos

Resultados obtenidos cuando reservamos 2 nodos:
Tiempo Real: 25.433 segundos
Tiempo Usuario: 18. 198 segundos
Tiempo sistema: 7.805 segundos

Resultados obtenidos cuando reservamos 1 nodos:
Tiempo Real: 3.828 segundos
Tiempo Usuario: 2.987 segundos
Tiempo sistema: 0.828 segundos



Como se puede observar, cuantos menos nodos tenemos, menos tardamos en ejecutar la aplicación. Esto parece razonable puesto que si los hilos no tienen ningún tipo de carga de trabajo, el tiempo se invierte en mandar los hilos a ejecutar a nodos esclavos, con lo que el peso principal se lo llevan las comunicaciones, lo que hace que el rendimiento de la máquina se vea decrementado.

5.2.1. Benchmarks con carga y mucho uso de memoria.

Ahora insertamos carga de trabajo en los hilos secundarios. En concreto en cada uno de los hilos secundarios introducimos una multiplicación de matrices de 15*15. Esto en teoría haría que el rendimiento de la aplicación se vea mejorado cuantos mas nodos tengamos, puesto que así lograríamos una ejecución en paralelo de los distintos hilos, en vez de una ejecución secuencial.

```
public int filas1 = 15;
public int filas2 = 15;
public int columnas1 = 15;
public int columnas2 = 15;
public double[][] matrix1 = new double[filas1][columnas1];
public double[][] matrix2 = new double[filas2][columnas2];
public void multiplicaMatriz(){
      // Inicializacion de matrices
      for (int i = 0; i < matrix1.length; i++){
           for (int j = 0; j < matrix1[0].length; j++){
            matrix1[i][j]= java.lang.Math.random();
         }
      }
      for (int i = 0; i < matrix2.length; i++){
           for (int j = 0; j < matrix2[0].length; j++){
            matrix2[i][j]= java.lang.Math.random();
         }
      }
      //Multiplicacion de las matrices
      for (int i = 0; i < matrix1.length; i++){
           for (int j = 0; j < matrix2[0].length; j++){
                for(int k = 0; k < matrix1[0].length; k++){
                    matrix3[i][j] += matrix1[i][k] * matrix2[k][j];
```

```
            }
         }
      }
  }//Fin de multiplicaMatriz
```

Los resultados obtenidos son los siguientes:

Resultados obtenidos cuando reservamos 8 nodos:
Tiempo Real: 230.085 segundos
Tiempo Usuario: 193.617 segundos
Tiempo sistema: 63.174 segundos

Resultados obtenidos cuando reservamos 7 nodos:
Tiempo Real: 225.898 segundos
Tiempo Usuario: 180.871 segundos
Tiempo sistema: 37.266 segundos

Resultados obtenidos cuando reservamos 6 nodos:
Tiempo Real: 224.832 segundos
Tiempo Usuario: 179.385 segundos
Tiempo sistema: 38.543 segundos

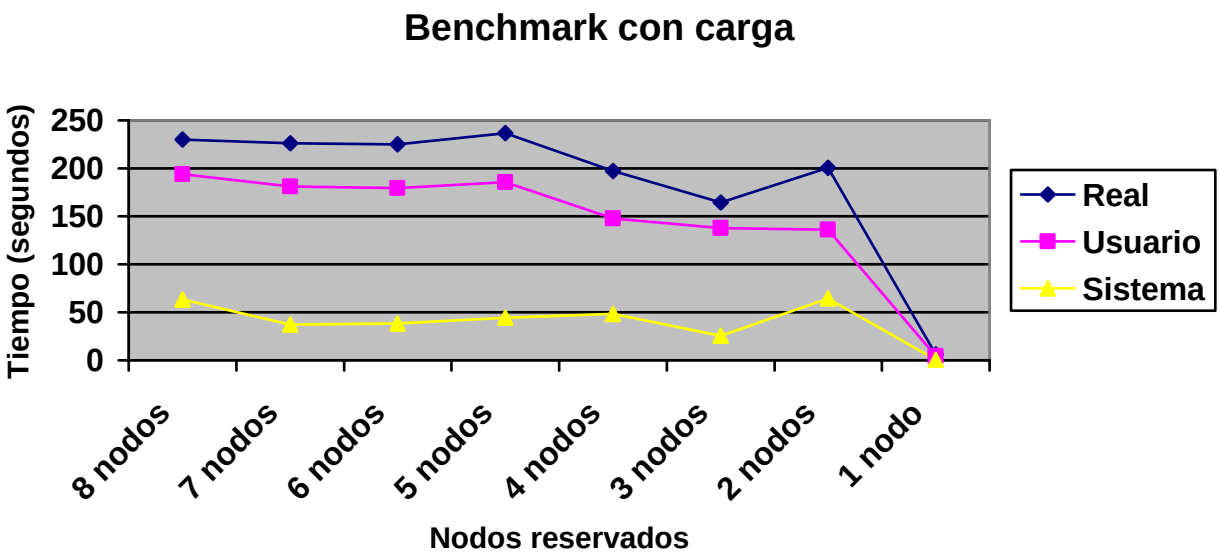
Resultados obtenidos cuando reservamos 5 nodos:
Tiempo Real: 236.533segundos
Tiempo Usuario: 185.344 segundos
Tiempo sistema: 44.218 segundos

Resultados obtenidos cuando reservamos 4 nodos:
Tiempo Real: 197. 312 segundos
Tiempo Usuario: 147.983 segundos
Tiempo sistema: 48.598 segundos

Resultados obtenidos cuando reservamos 3 nodos:
Tiempo Real: 164.605 segundos
Tiempo Usuario: 137,920 segundos
Tiempo sistema: 25.807 segundos

Resultados obtenidos cuando reservamos 2 nodos:
Tiempo Real: 200.606 segundos
Tiempo Usuario: 136.305 segundos
Tiempo sistema: 64.295 segundos

Resultados obtenidos cuando reservamos 1 nodos:
Tiempo Real: 6.101 segundos
Tiempo Usuario: 4.447 segundos
Tiempo sistema: 0.831 segundos



En este caso, no obtenemos el resultado esperado porque es necesaria una recolección de basura al tener tanto uso de memoria. Esto, sumado a el hecho de que solamente haya un colector de basura en el nodo maestro (en los nodos esclavos no se recolecta basura), hace que se vea tan degradado el rendimiento.

5.2.2. Benchmarks con carga y poco uso de memoria.

Para esta nueva prueba también vamos a tener carga de trabajo en los hilos secundarios. En concreto en cada uno de los hilos secundarios introducimos 10000000 multiplicaciones de números enteros de forma que ahora no tenemos que almacenar matrices tan grandes en memoria, y por tanto es de esperar que esta vez no se lanze el colector de basura. Además hemos reducido el número de hilos primarios a 16, y el de secundarios a 64. La nueva clase SecondaryThread queda como sigue:

```
import com.ibm.JikesRVM.DVM_Out;
import java.lang.*;

public class SecondaryThread extends Thread {
  public int id;
  public int primaryId;


  public SecondaryThread(int anId, int aPrimaryId) {
      id = anId;
      primaryId = aPrimaryId;
      //DVM_Out._log("SecondaryThread created(primaryId,id)", primaryId, id);
  }

  public void run() {
      DVM_Out._log("SecondaryThread running(primaryId,id)", primaryId, id);
      for (int l=0; l<10000000; l++){
            int valor = 5*5;
      }
```

```
  }//Fin del metodo Run
}
```

Los resultados obtenidos son los siguientes:

Resultados obtenidos cuando reservamos 8 nodos:
Tiempo Real: 6.739 segundos
Tiempo Usuario: 5.526 segundos
Tiempo sistema: 1.225 segundos

Resultados obtenidos cuando reservamos 7 nodos:
Tiempo Real: 6.754 segundos
Tiempo Usuario: 5.685 segundos
Tiempo sistema: 1.057 segundos

Resultados obtenidos cuando reservamos 6 nodos:
Tiempo Real: 7.367 segundos
Tiempo Usuario: 6.127 segundos
Tiempo sistema: 1.249 segundos

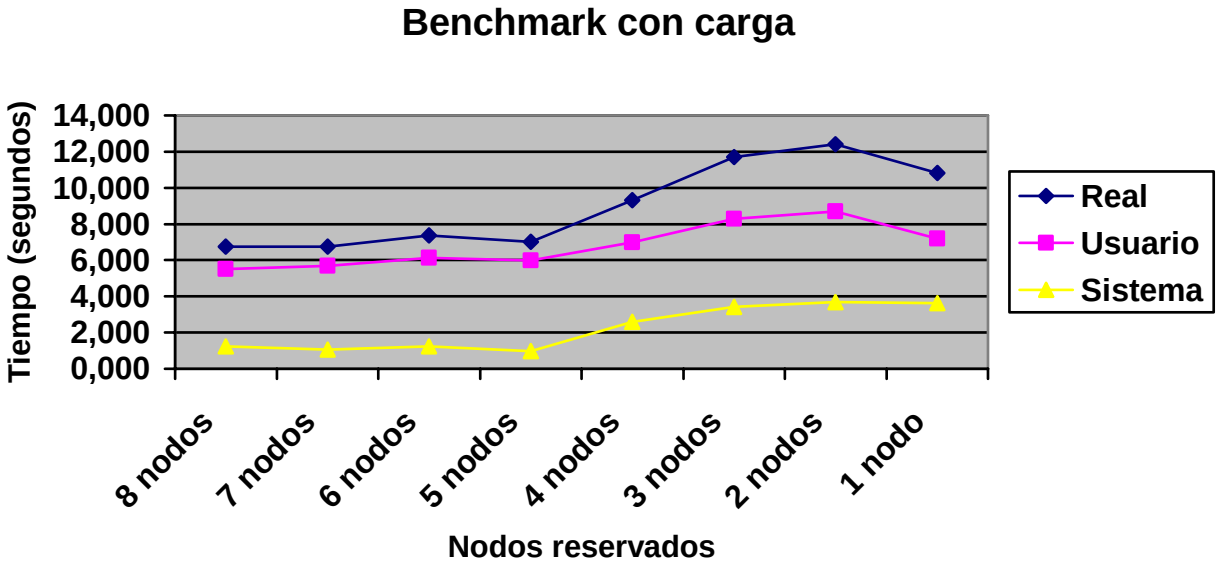
Resultados obtenidos cuando reservamos 5 nodos:
Tiempo Real: 7.011 segundos
Tiempo Usuario: 5.980 segundos
Tiempo sistema: 0.986 segundos

Resultados obtenidos cuando reservamos 4 nodos:
Tiempo Real: 9.312 segundos
Tiempo Usuario: 6.983 segundos
Tiempo sistema: 2.598 segundos

Resultados obtenidos cuando reservamos 3 nodos:
Tiempo Real: 11.692 segundos
Tiempo Usuario: 8.277 segundos
Tiempo sistema: 3.425 segundos

Resultados obtenidos cuando reservamos 2 nodos:
Tiempo Real: 12,397 segundos
Tiempo Usuario: 8.705 segundos
Tiempo sistema: 3.686 segundos

Resultados obtenidos cuando reservamos 1 nodos:
Tiempo Real: 10.811 segundos
Tiempo Usuario: 7.180 segundos
Tiempo sistema: 3.631 segundos



Ahora podemos observar como sí que obtenemos un mayor rendimiento a partir de 4 nodos. No obtenemos mayor rendimiento antes, pues la potencia de cómputo que conseguimos no compensa el coste de la comunicación para la versión distribuida.

Apéndices

A. Ejecutar trabajos en el cluster del DACYA.

A.1. Introducción.

Para proporcionar un acceso libre de conflictos a los nodos de cómputo (pila01-pila08) del DACYA, la asignación de recursos y la planificación de tareas se lleva a cabo usando PBS (Portable Batch System) en su versión 2.3.16. Para evitar la interferencia con los trabajos PBS, todos los accesos a los nodos de cómputo, incluidas las shells interactivas, deben arrancarse a través de PBS. No hay necesidad de acceder vía rlogin o telnet a cada nodo individual del Dacya, y todos los comandos rsh y rcp que accedan a los nodos serán enviados como trabajos de PBS. Los trabajos no enviados vía PBS pueden ser eliminados sin ningún tipo de aviso.

A.2. Propiedades de los nodos.

El servidor del DACYA, **aneto** tiene a menudo una gran variedad de actividades llevándose a cabo en él, que son independientes de PBS, como pueden ser login shells, compilaciones, editores, transferencias de ficheros, etc. Para estos trabajos que necesitan máximo rendimiento con la mínima interferencia, es importante distinguir entre el nodo servidor y los nodos de cómputo a la hora de enviar un trabajo de PBS. El servidor también difiere de los nodos de cómputo en otras cosas, como el tamaño de la memoria y el tamaño del disco. Las propiedades de los nodos del DACYA, se listan a continuación:

Nombre del Nodo	***Tipo de Nodo***	***Tipo de Procesador***	***Configuración de la CPU***	***Tamaño de memoria (MB)***	***Kernel***	***Sistema Operativo / Distribución***	***Tamaño de disco (GB)***
aneto	servidor	P4 3.0 Ghz	Hyperthreading	2048	2.6.7-SMP	Debian	1500
Pila01	computo	P4 3.2 Ghz	Hyperthreading	2048	2.6.6-SMP	Debian	40
Pila02	computo	P4 3.2 Ghz	Hyperthreading	2048	2.6.6-SMP	Debian	40
Pila03	computo	P4 3.2 Ghz	Hyperthreading	2048	2.6.6-SMP	Debian	40
Pila04	computo	P4 3.2 Ghz	Hyperthreading	2048	2.6.6-SMP	Debian	40
Pila05	computo	P4 3.2 Ghz	Hyperthreading	2048	2.6.6-SMP	Debian	40
Pila06	computo	P4 3.2 Ghz	Hyperthreading	2048	2.6.6-SMP	Debian	40
Pila07	computo	P4 3.2 Ghz	Hyperthreading	2048	2.6.6-SMP	Debian	40
Pila08	computo	P4 3.2 Ghz	Hyperthreading	2048	2.6.6-SMP	Debian	40

Desde la versión 2.3 de PBS, se sabe que una CPU con Hyperthreading puede ejecutar dos procesos en paralelo. Estos procesos no tienen porque pertenecer al mismo trabajo, ya que PBS maneja CPUs (CPUs virtuales, que no es realmente un modo de operación dual), no hosts. Los nodos de cómputo del DACYA se tratan como nodos de un cluster cuyas CPUs se asignan de forma exclusiva a cada trabajo.

Ejemplos:

Como ilustran los siguientes ejemplos, las propiedades de los nodos se especifican mediante la opción "*-l nodes*" cuando se envía un trabajo con el comando *qsub*.

Una petición de dos nodos de computo podría ser la siguiente:

qsub -l nodes=2

Si deseamos pedir cuatro nodos, dos de ellos con dos CPUs por nodo y los otros dos con una CPU por nodo, podemos usar el siguiente comando:

qsub -l nodes=2:ppn=2+2:ppn=1

Aquí el uso de *ppn=...* es crucial. Esta petición asiganrá 2 nodos de computo duales primero (cada CPU dual debe estar listada 2 veces en el fichero *$PBS_NODEFILE*), seguido de dos nodos con CPUs duales usando solamente una de las CPUs. Cuando arranquemos el trabajo, se ejecutará un proceso por CPU asignada.

Si solamente usamos una CPU en un nodo con CPUs duales, podemos terminar compartiendo la memoria y el ancho de banda con otro proceso. Para evitar eso, debemos usar *ppn=n* de la forma indicada antes.

Si queremos pedir pila01 y pila02:

qsub -l nodes=pila01+pila02

Si PBS no puede satisfacer la petición debido a un conflicto o a que no dispone de suficientes nodos, qsub devolverá un mensaje indicando: *Job exceeds queue resource limits*.

A.3. Colas.

Además de los anteriores mecanismos asociados al planificador por defecto de PBS, existe una elaborada estructura de colas. Afortunadamente, los usuarios no deben especificar una cola particular cuando se envía un trabajo, de hecho, PBS esta configurado para evitar esto. En su lugar, todos los trabajos son enviados por defecto a una cola de enrutado que examina sus peticiones de recursos y los envía a la cola de ejecución adecuada.

A.4. Enviando trabajos.

Todos los trabajos son enviados a PBS vía el comando *qsub*, o a través de *xpbs* que es una interfaz gráfica de PBS. Aquí nos vamos a centrar en *qsub*. Su sintaxis básica es:

qsub [opciones] [script]

script contiene el comando ejecutable, y tal vez directivas de PBS, que comprende el trabajo. Por defecto, PBS ejecuta el script invocando un login shell de usuario, típicamente uno de: sh, bash, ksh, csh o tcsh. Si el script se omite, qsub leerá los comandos y directivas de la entrada estandard.

qsub proporciona numerosas opciones. Para nuestro propósito, las más importantes son:

-l nodes=	Especifica el número de nodos y sus propiedades
-l cput=	Especifica el máximo tiempo de reloj limite para el trabajo.
-I	Arranca un trabajo interactivo

La opcion "*-l nodes*" ha sido descrita en el apartado A.2. Si se omite esta opción, el nodo especificado por defecto es "*1:compute*".

La opcion "*-l cputime*" indica el tiempo máximo para el trabajo. Este valor se usa con "*-l nodes*" para determinar en que cola de ejecución se pondrá el trabajo. Si el intervalo de tiempo no se especifica, el valor por defecto será entre 5-10 minutos, dependiendo del número de nodos pedidos. Si el trabajo excede el límite de tiempo, PBS lo matará.

Para facilitar la correcta planificación de trabajos, el tiempo requerido de CPU debería estar razonablemente bien ajustado al tiempo necesario real, con un pequeño margen para evitar terminaciones accidentales. Estimaciones de tiempo muy diferentes de las reales, pueden provocar que los trabajos se sitúen en colas inapropiadas, con menor prioridad, y permitiendo a otros trabajos consumir el tiempo de la CPU, dejando a otros más prioritarios fuera. El intervalo de tiempo puede ser especificado de diferentes formas. Las siguientes son equivalentes:

-l cput=4800	Segundos
-l cput=80:00	Minutos y segundos
-l cput=1:20:00	Horas, minutos y segundos

La opción *-I* se describirá mas adelante en la sección A.4.2. PBS proporciona solamente asignación de tareas y planificación de servicios; no es un sistema específico para el lanzamiento de programas en paralelo. En su lugar, se apoya en otros tipos de scripts o programas para actuar como interfaz de aplicaciones paralelas.

A.4.1. El script resolver.sh.

El código del script que nos permite determinar las direcciones de los nodos asignados por PBS nodes es el siguiente:

```
#!/bin/sh

LINEAS=`wc -l $PBS_NODEFILE | cut -d" " -f1`
L=1


while [ "$L" -le "$LINEAS" ] ; do

HOST=`head -n$L $PBS_NODEFILE | tail -n1`
IP=`grep $HOST /etc/hosts | awk '{printf $1}'`
echo -e $IP

let L=L+1
done
```

Este script básicamente lo que hace es acceder a la variable de entorno *$PBS_NODEFILE* que nos dice los nodos asignados por PBS, y busca en el fichero de hosts (*$HOST*) para determinar la dirección IP.

A.4.2. Trabajos interactivos.

La opción "*-I*" se usa para iniciar trabajos interactivos, lo que significa que el proceso shell será conectado a la sesión de terminal desde la que ha sido invocado qsub. Esto permite al usuario enviar comandos de forma interactiva en los nodos asignados. *qsub -I* debería ser usando en vez de otros comandos como rlogin, telnet, rsh o rcp. Por ejemplo, para obtener un login shell en pila02 durante 30 minutos, podemos usar:

qsub -I -l nodes=pila02 -l cput=30:00

A.5. Monitorizar el estado de los trabajos.

PBS proporciona varias herramientas para monitorizar el estado de los trabajos en las colas. La más simple de todas es el comando inuse que proporciona una lista de los nodos actualmente en uso.

El comando pbsnodes también puede ser usado para comprobar el estado de los nodos.

- Para listar el estado y las propiedades de cada nodo:
  pbsnodes -a
- Para saber que nodos están caídos o fuera de línea:
  pbsnodes -l

La información sobre trabajos y colas nos la proporciona el comando qstat.

- Para listar todas las colas y sus limites de ejecución:
  qstat -q
- Para conocer el estado de todas las colas:
  qstat -Q
- Para listar todos los trabajos y sus estados:
  qstat -a
- Para listar el estado de todos tus trabajos:
  qstat -a -u $USER
- Para listar todos los trabajos con una breve descripción de sus estados:
  qstat -a -s
- Para listar todos los trabajos que se están ejecutando:
  qstat -r
- Para listar todos los trabajos que no se están ejecutando:
  qstat -i
- Para obtener información detallada sobre todos los trabajos:
  qstat -f
- Para obtener información detallada sobre el trabajo X (X es un número):
  qstat -f X
- Para obtener la lista de los nodos asignados al trabajo X (X es un número):
  qstat -n X

A.6. Terminación de trabajos.

Los trabajos pueden ser eliminados de las colas (y terminados si se están ejecutando) usando el comando qdel:

qdel job_id

Nota 1: La terminación de trabajos es un proceso que se lleva a cabo en varios pasos que pueden tardar varios minutos en completarse. Si se lanza más de un qdel para un trabajo ejecutándose, la limpieza del nodo puede abortarse, dejando procesos hijos activos huérfanos. Como regla, es aconsejable esperar 5 minutos antes de asumir que la petición de eliminación ha fallado.
Nota 2: Cuando un trabajo excede su tiempo límite, puede que PBS tarde unos pocos minutos en darse cuenta, y unos pocos minutos más para que complete la terminación y limpieza del proceso. Durante este proceso, no intente eliminarlo manualmente con qdel. Espere hasta que el proceso haya sobrepasado unos 10 minutos su tiempo límite antes de concluir que algo ha fallado.

Las mismas precauciones deben ser usadas con el comando *qsig*, que debe ser usada solamente en circunstancias extraordinarias.

A.7. Documentación sobre PBS.

PBS tiene más características además de las aquí descritas. PBS tiene disponible un manual muy completo que en nuestro caso puede ser encontrado en /usr/local/man en el servidor del DACYA.

También hay disponibles varios manuales como los siguientes:

v2.0_ers.pdf	External Reference Specification(ERS): Información detallada que incluye las características, especificaciones, APIs, comandos de usuario y de administrador, y protocolos de red.
v2.0_UserCmds.pdf	Capítulo 5 del ERS, que describe los comandos de usuario. Esencialmente un subconjunto de las páginas del manual de nivel usuario.
v2.3_admin.pdf	Administrator's Guide: Instrucciones de compilación, instalación, configuración y manejo de PBS.

B. Instalación de Jikes Distributed Java Virtual Machine.

En este capítulo explicamos los pasos necesarios para llevar a cabo la instalación de Jikes dJVM.

B.1. Requerimientos de sistema y de utilidades.

Jikes Distributed Virtual Machine requiere que la computadora que actúa como host tenga corriendo bajo Linux 2.4 el conjunto de instrucciones i386. El sistema ha sido probado satisfactoriamente con los Kernel 2.4.9 y 2.4.20.

En el directorio $java, debemos tener instalada una de las siguientes versiones de Java:

- **SUN JDK v1.3.1 08**, disponible en http://java.sun.com/j2se/1.3/download.html y instalado en /usr/java/jdk1.3.1 08.
- **Blackdown JDK v1.3.1-02b**, disponible en ftp://mirror.aarnet.edu.au/pub/java-linux/JDK-1.3.1/i386/FCS-02b y instalado en /usr/local/j2sdk1.3.1.

(*)Nota: Se ha probado en algunas otras versiones como la JDK 1.4.2 pero daba problemas.

Además de esto hacen falta las siguientes tres herramientas:

- **Kshell**: versión 5.2 o posterior que normalmente viene instalado en /bin/ksh, y disponible en casi todas las versiones de RedHat (y demás) distribuciones de Linux.
- Compilador de Java **Jikes**: versión 1.18 o posterior, que lo podemos descargar de http://oss.software.ibm.com/developerworks/project/showfiles.php?group_id=10, y debemos de tener instalado en /usr/bin/jikes, compilador de Java escrito en C totalmente independiente de las librerías Java utilizado para compilar el código JikesRVM.
  (*)Nota: Solamente podemos garantizar que funciona con la versión 1.18
- **Unzip**: versión 5.50 o superior

B.2. Construcción de Jikes Distributed Java Virtual Machine.

Deben ser declaradas las siguientes variables de entorno:

RVM	*$HOME/jikes2/proyecto*
RVM_ROOT	*$RVM/rvmRoot*
RVM_BUILD	*$RVM/rvmBuild*
RVM_TARGET_CONFIG	*$RVM_ROOT/rvm/config/i686-pc-linux-gnu*
RVM_HOST_CONFIG	*$RVM_ROOT/rvm/config/i686-pc-linux-gnu*
RVM_HOST_JAVA_HOME	*$java*
PATH	*$RVM_ROOT/rvm/bin: $RVM_HOST_JAVA_HOME/bin:$PATH*
CLASSPATH	*.:$RVM_BUILD/RVM.classes: $RVM_BUILD/RVM.classes/rvmrt.jar: $RVM_BUILD/RVM.classes/jksvm.jar*
RSH	*either ssh or rsh*

Estas variables de entorno son necesarias tanto para configurar la máquina como para compilar y ejecutar aplicaciones. Para instalar DJVM asegurarse de que el directorio $RVM existe y se puede escribir en el.

Obtener y descomprimir usando "unzip" el archivo "DJVM.tar" y seguir los pasos mostrados a continuación.

B.2.1. Construir la estructura de directorios.

El árbol de directorios es construido utilizando el script DJVMinstall. Este script crea el árbol estándar de Jikes RVM no distribuido en $RVM_ROOT y le añade lo siguiente:

- Nuevos directorios específicos para la versión distribuida y transforma algunas de las clases de Jikes RVM
- La librería de arranque extraida desde jlibraries.tar.gz
- Las fuentes de las librerías de arranque extraídas desde jlibsource.tar.gz

Una vez que el árbol es creado, se le aplica el patch jikesrvm-cluster-extns-1.0.1.tar,gz. Este patch incluye los cambios necesarios sobre las clases que se han tenido que adaptar, o que han tenido que ser creadas que son aplicados sobre los directorios existentes.

B.2.2. Construyendo el GNU Claspath Library.

El archivo GNU classpath (…/support/lib/classpath.jar) puede ser construido durante el curso normal de la construcción de dJVM si el archivo jar no es encontrado. Por conveniencia y para evitar la innecesaria invocación de wget hemos adaptado el script jBuildClassPathJar que pasa a ser DJVMClassPathJar. Este debe ser arrancado desde el mismo directorio inmediatamente después de DJVMinstall.

(*)Nota: Es posible que durante el lanzamiento de este script aparezcan algunos "warning". Sin embargo la librería será construida si nada extraño sucede.

B.2.3. Construyendo dJVM.

Una vez que estos pasos se han completado, dJVM puede ser construida de la misma forma que la versión no distribuida:

1. Ejecutar el script jconfigure dandole como parámetro uno de los archivos de configuración disponibles en …/rvm/config/build/. Los archivos disponibles son: ClusterBaseBaseMarkSweep y ClusterBaseBaseNoGC.
2. Cambiar de directorio a $RVM_BUILD.
3. Ejecutar el script ./jbuild. Esto puede llevar más o menos tiempo dependiendo de la máquina utilizada. (Aprox 3 minutos en un AMD 1667 MHz)

El proceso de construcción ha sido alterado para la regeneración de la librería rvmrt.jar. Esto ocurre durante la fase de copia ejecutada en el script jbuild.copy invocado desde jbuild.

Bibliografía

- The Jikes™ Reserarch Virtual Machina User´s Guide : Disponible en versión html en la dirección: http://www.ibm.com/developerworks/oss/jikesrvm/userguide/HTML/userguide.html
- The Jikes Distributed Java Virtual Machina Manual: Disponible en la formato PostScript en la dirección: http://cap.anu.edu.au/cap/projects/dJVM .
- Documentación sobre el cluster del DACYA disponible en http://aneto.dacya.ucm.es/documentacion/index.html aunque con acceso restringido.
- Información sobre Jalapeño: Disponible en http://researchweb.watson.ibm.com/jalapeno .
- Información adicional sobre JikesRVM: Disponible en http://jikesrvm.sourceforge.net .

Palabras clave.

Máquina virtual

Cluster

Java

JikesRVM

dRVM

Recolector de Basura

Benchmark

Aneto

Jalapeño

Servidor

Concesión de derechos.

Los abajo firmantes autorizan a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Luis Jarabo de la Llana Pablo Molina García Rafael Guijarro Crespo

En Madrid, a día 1 de Julio del año 2005.