UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
95/001,560	03/01/2011	7426720	13557.105125	8687

25226 7590 04/18/2011
MORRISON & FOERSTER LLP
755 PAGE MILL RD
PALO ALTO, CA 94304-1018

EXAMINER
STEELMAN, MARY J

ART UNIT	PAPER NUMBER
3992	

MAIL DATE	DELIVERY MODE
04/18/2011	PAPER

Please find below and/or attached an Office communication concerning this application or proceeding.

The time period for reply, if any, is set in the attached communication.

UNITED STATES PATENT AND TRADEMARK OFFICE

Commissioner for Patents
United States Patents and Trademark Office
P.O.Box 1450
Alexandria, VA 22313-1450
www.uspto.gov

DO NOT USE IN PALM PRINTER

THIRD PARTY REQUESTER'S CORRESPONDENCE ADDRESS
KING & SPALDING
1180 PEACHTREE STREET, N.E.
ATLANTA, GA 30309-3521

Date: MAILED
APR 10 2011
CENTRAL REEXAMINATION UNIT

Transmittal of Communication to Third Party Requester
Inter Partes Reexamination

REEXAMINATION CONTROL NO. : 95001560
PATENT NO. : 7426720
TECHNOLOGY CENTER : 3999
ART UNIT : 3992

Enclosed is a copy of the latest communication from the United States Patent and Trademark Office in the above identified Reexamination proceeding. 37 CFR 1.903.

Prior to the filing of a Notice of Appeal, each time the patent owner responds to this communication, the third party requester of the inter partes reexamination may once file written comments within a period of 30 days from the date of service of the patent owner's response. This 30-day time period is statutory (35 U.S.C. 314(b)(2)), and, as such, it cannot be extended. See also 37 CFR 1.947.

If an ex parte reexamination has been merged with the inter partes reexamination, no responsive submission by any ex parte third party requester is permitted.

All correspondence relating to this inter partes reexamination proceeding should be directed to the Central Reexamination Unit at the mail, FAX, or hand-carry addresses given at the end of the communication enclosed with this transmittal.

PTOL-2070(Rev.07-04)

# *OFFICE ACTION IN* INTER PARTES *REEXAMINATION*	Control No.	Patent Under Reexamination	
	95/001,560	7426720	
	Examiner	Art Unit	
	MARY STEELMAN	3992	

-- The MAILING DATE of this communication appears on the cover sheet with the correspondence address. --

Responsive to the communication(s) filed by:
Patent Owner on _____
Third Party(ies) on 1 March 2011

RESPONSE TIMES ARE SET TO EXPIRE AS FOLLOWS:

For Patent Owner's Response:
1 MONTH(S) from the mailing date of this action. 37 CFR 1.945. EXTENSIONS OF TIME ARE GOVERNED BY 37 CFR 1.956.
For Third-Party Requester's Comments on the Patent Owner Response:
30 DAYS from the date of service of any patent owner's response. 37 CFR 1.947. NO EXTENSIONS OF TIME ARE PERMITTED. 35 U.S.C. 314(b)(2).

All correspondence relating to this inter partes reexamination proceeding should be directed to the **Central Reexamination Unit** at the mail, FAX, or hand-carry addresses given at the end of this Office action.

This action is not an Action Closing Prosecution under 37 CFR 1.949, nor is it a Right of Appeal Notice under 37 CFR 1.953.

PART I. THE FOLLOWING ATTACHMENT(S) ARE PART OF THIS ACTION:

1.☐ Notice of References Cited by Examiner, PTO-892
2.☒ Information Disclosure Citation, PTO/SB/08
3.☐ _____

PART II. SUMMARY OF ACTION:

1a. ☒ Claims 1-8,10-17 and 19-22 are subject to reexamination.
1b. ☒ Claims 9 and 18 are not subject to reexamination.
2. ☐ Claims _____ have been canceled.
3. ☐ Claims _____ are confirmed. [Unamended patent claims]
4. ☐ Claims _____ are patentable. [Amended or new claims]
5. ☒ Claims 1-8,10-27,19-22 are rejected.
6. ☐ Claims _____ are objected to.
7. ☐ The drawings filed on _____ ☐ are acceptable ☐ are not acceptable.
8. ☐ The drawing correction request filed on _____ is: ☐ approved. ☐ disapproved.
9. ☐ Acknowledgment is made of the claim for priority under 35 U.S.C. 119 (a)-(d). The certified copy has: ☐ been received. ☐ not been received. ☐ been filed in Application/Control No 95001560.
10. ☐ Other _____

Inter Partes Reexamination

DETAILED ACTION

This first Office Action on the merits follows the Order granting *Inter Partes* Reexamination of USPN 7,426,720 B1 to Fresko. Claims 1-8, 10-17, and 19-22 are requested for reexamination.

Application Control Number 10/745,023 (file date 12/22/2003), issued as USPN 7,426,720 B1 (09/16/2008) to Fresko. A terminal disclosure was filed (07/10/2007) and accepted (09/20/2007) over Application control number 10/745,022, now USPN 7,293,267, where similar subject matter was claimed by the same inventor.

In order to ensure full consideration of any amendments, affidavits or declarations, or other documents as evidence of patentability, such documents must be submitted in response to this Office Action. Submissions after the next Office Action, which is intended to be an Action Closing Prosecution (ACP), will be governed by 37 CFR 1.116(b) and (d), which will be strictly enforced.

<u>Statutory Basis for Grounds of Rejections. 35 USC §102 and §103</u>

The following is a quotation of the appropriate paragraphs of 35 U.S.C. §102 that form the basis for the rejections under this section made in this Office Action:

A person shall be entitled to a patent unless -

(b) the invention was patented or described in a printed publication in this or a foreign country or in public use or on sale in this country, more than one year prior to the date of application for patent in the United States.

The following is a quotation of 35 U.S.C. §103(a) which forms the basis for all obviousness rejections set forth in this Office Action:

(a) A patent may not be obtained though the invention is not identically disclosed or described as set forth in section 102 of this title, if the differences between the subject matter sought to be patented and the prior art are such that the subject matter as a whole would have been obvious at the time the invention was made to a person having ordinary skill in the art to which said subject matter pertains. Patentability shall not be negatived by the manner in which the invention was made.

Third Party Requester's Proposed Grounds of Rejections

Ground #1

Claims 1-8, 10-17, and 19-22 are unpatentable under 35 U.S. C. § 103(a) as rendered obvious by Webb in view of Kuck and further in view of APA-Bach.

Ground #2

Claims 1-8, 10-17, and 19-22 are unpatentable under 35 U.S. C. § 102(b) as anticipated by Dike in view of Steinberg.

Ground #3

Claims 1-8, 10-17, and 19-22 are unpatentable under 35 U.S. C. § 103(a) as rendered obvious by Dike in view of Steinberg.

Ground #4

Claims 1-8, 10-17, and 19-22 are unpatentable under 35 U.S. C. § 103(a) as rendered obvious by Bryant in view of APA-Bach.

Ground #5

Claims 1-8, 10-17, and 19-22 are unpatentable under 35 U.S. C. § 103(a) as rendered obvious by Bryant in view of Traut.

Ground #6

Claims 1-8, 10-17, and 19-22 are unpatentable under 35 U.S. C. § 103(a) as rendered obvious by Srinivasan in view of APA-Bach.

Ground #7

Claims 1-8, 10-17, and 19-22 are unpatentable under 35 U.S. C. § 103(a) as rendered obvious by Sexton in view of Bugnion.

Ground #8

Claims 1-8, 10-17, and 19-22 are unpatentable under 35 U.S. C. § 103(a) as rendered obvious by Sexton in view of Johnson.

Analysis of Proposed Third Party Requester's Rejections

Webb as a primary reference

Re. Ground #1:

Claims 1-8, 10-17, and 19-22 are rejected under 35 U.S. C. § 103(a) as rendered obvious by Webb in view of Kuck and further in view of APA-Bach. Pertinent teachings found in

Request 03/01/2010, p. 36 and Exhibit 17 are incorporated by reference. This rejection is **adopted**, with additional comments added by Examiner.

Regarding independent **claims 1, 10, and 20**:

Webb provides an overview of Java (1: 22-38): "Java is an object-oriented language. Thus a Java program is formed from a set of class files having methods [claim 10, method] that represent sequences of instructions. A hierarchy of classes can be defined, with each class inheriting properties from those classes which are above it in the hierarchy. For any given class in the hierarchy, its descendants (i.e. below it) are call subclasses, whilst its ancestors (i.e. above it) are called superclasses. At run-time objects are created as instantiations of these class files [a class pre-loader], and indeed the class files themselves are effectively loaded as objects. One Java object can call a method in another Java object. In recent years Java has become very popular, and is described in many books, for example "Exploring Java" by Niemeyer and Peck, O'Reilly & Associates, 1996, USA, and "The Java Virtual Machine Specification" by Lindholm and Yellin, Addison-Wedley, 1997, USA."

Webb discloses (3: 56-4: 2), "FIG. 1 illustrates a computer system 10 including a (micro) processor 20 which is used to run software loaded into memory 60. The software [claim 19] can be loaded into the memory by various means (not shown), for example from a removable storage device such as a floppy disc or CD ROM, or over a network such as a local area network (LAN) or telephone/modem connection, typically via a hard disk drive (also not shown). computer system [claim 1, system, claim 20, apparatus] runs an operating system (OS) 30, on top of which is provided a Java virtual machine (JVM) 40. The JVM looks like an application to the (native) OS 30, but in fact functions itself as a virtual operating system, supporting Java application 50. A Java application may include multiple threads, illustrated by threads T1 and T2 71, 72."

Webb implements (1: 22-38) a class pre-loader: "[a]t run-time objects are created as instantiations of these class files, and indeed the class files themselves are effectively loaded as objects." Web discloses (4: 26-41), a system for dynamic preloading classes with a hierarchy of class loaders, through a cloning mechanism. Webb discloses (FIG. 3, 6: 59-7: 5 & 7: 25-40) a master runtime system process to interpret and to instantiate the representation as a class definition in a memory space of the master runtime system process: "initialization of a loaded class (step 350), which represents calling the static initialization method (or methods) of the class . . . this application must be performed once and only once before the first active use of a class" which then allows for "[t]he new application class [to load the application classes into the JVM (step 410), and involves the steps shown in Fig. 3 for all the application classes." Note that initialization of an object also requires initialization of its superclasses, and so this may involve recursion up a superclass tree in a similar manner to that described for resolution. The initialization flag in a class file 145 is set as part of the initialization process, thereby ensuring that the class is not subsequently re-initialised."

Webb discloses (4: 26-41), "FIG. 2 shows the structure of JVM 40 in more detail (omitting some components which are not directly pertinent to an understanding of the present invention). The fundamental unit of a Java program is the class, and thus in order to run any application the JVM must first load the classes forming and required by that application. For this purpose the JVM includes a hierarchy of class loaders 110, which conventionally includes three particular class loaders, named Application 120, Extension 125, and Primordial 130. An application can add additional class loaders to the JVM (a class loader is itself effectively a Java program). In the preferred embodiment of the present invention, a fourth class loader is also supported, Middleware 124. Classes which are loaded by this class loader will be referred to hereinafter as middleware, whilst those loaded by Application Class loader 120 will be referred to as application."

Webb discloses (7: 25-40), "FIG. 4 illustrates a method in accordance with the present invention whereby this problem can be largely overcome. The method starts with the middleware initiating an application (e.g. a transaction), it being assumed that both the relevant part of the middleware and the application here are Java programs. In order to do this, the middleware creates an instance of the application class loader (it cannot use an existing one because the application class loader is below the middleware class loader in the class loader hierarchy as shown in FIG. 2, and so it has no reference to it). The new application class loader instance then loads the application classes into the JVM (step 410), and involves the steps shown in FIG. 3 for all the application classes. The application is then run in standard fashion (step 420), and after completion control returns to the middleware."

Webb does not disclose a runtime environment to clone the memory space as a child runtime system process responsive to a process request and to execute the child runtime system process.

However, Kuck discloses a runtime environment to clone the memory space as a child runtime system process responsive to a process request and to execute the child runtime system process (paragraph [0064]-[0065] "... initialization overhead can be reduced through the use of a pre-initialized "master" PAVM (virtual machine). Rather than initializing a new PAVM from scratch, the memory block of the master PAVM can simply be copied into the memory block of the new PAVM. Copying the template image of the master PAVM's memory block into the memory block of a new PAVM enables the new PAVM to start running in an already-initialized state...storing type information (i.e., the runtime representation of loaded classes) in a section of shared memory that can be accessed by all the PAVMs... technique can reduce the overhead for class loading, verification, and resolution incurred by each PAVM, and can be especially useful if used to share the byte code of system classes that are likely to be used by every user context").

Kuck discloses [0029] many analogous features to Webb's teachings: "Referring to FIG. 1, a network 10 includes a server 12 linked to client systems 14, 16, 18. The server 12 is a programmable data processing system suitable for implementing apparatus or performing methods in accordance with the invention. The server 12 contains a processor 16 and a memory 18. Memory 18 stores an operating system (OS) 20, a Transmission Control Protocol/Internet

Protocol (TCP/IP) stack 22 for communicating over the network 10, and machine-executable instructions 24 executed by processor 16 to perform a process 500 below. In some implementations, the server 12 can contain multiple processors, each of which can be used to execute the machine-executable instructions 24."

The Webb/Kuck combination fails to explicitly disclose the "save the copy on write limitation." The explicit, stated purpose of the disclosures is to reduce memory usage by, in part, implementing a shared memory state for the instantiation and execution of virtual machines.

APA-Bach (p. 192) discloses, "The only way for a user to create a new process in the UNIX operating system is to invoke the fork system call." APA-Bach (p. 287) discloses, "The copy-on-write bit, used in the fork system call, indicates that the kernel must create a new copy of the page when a process modifies its contents." APA-Bach (p. 289-290) discloses, "9.2.1.1 Fork in a Paging System As explained in Section 7.1, the kernel duplicates every region of the parent process during the fork system call and attaches it to the child process. Traditionally, the kernel of a swapping system makes a physical copy of the parent's address space, usually a wasteful operation, because processes often call exec soon after the fork call and immediately free the memory just copied. On the System V paging system, the kernel avoids copying the page by manipulating the region tables, page table entries, and pfdata table entries: It simply increments the region reference count of shared regions .... The page can now be referenced through both regions, which share the page until a process writes to it. The kernel then copies the page so that each region has a private version. To do this, the kernel turns on the 'copy on write' bit for every page table entry in private regions of the parent and child processes during fork. If either process writes the page, it incurs a protection fault, and in handling the fault, the kernel makes a new copy of the page for the faulting process. The physical copying of the page is thus deferred until a process really needs it."

Therefore, it would have been obvious to a person of ordinary skill in the art at the time the invention was made to modify Webb's teaching by adding a runtime environment to clone the memory space as a child runtime system process responsive to a process request and to execute the child runtime system process as taught by Kuck in order to provide avoiding the overhead and crashing the process and enabling the server to run robustly (Kuck, paragraph [0004]-[0005]). To further modify Webb/ Kuck with the teachings of APA-Bach provided a copy-on-write process cloning mechanism to instantiate the child runtime system process by copying references to the memory space of the master runtime system process into a separate memory space for the child runtime system process, and to defer copying of the memory space of the master runtime system process until the child runtime system process needs to modify the referenced memory space of the master runtime system process. In keeping with the explicit motivation to reduce memory usage and overhead impact when instantiating and executing virtual machines, it would have been obvious to one of ordinary skill in the art to combine the cloning of a master runtime process as disclosed in Webb and Kuck with the copy-on-write technology that was the focus of much of Chapter 9 of the APA-Bach reference.

Regarding the "means for" limitations of claim 20:

In support of the conclusion that the prior art elements are an equivalent, the combination of prior art elements are shown to perform an equivalent "means for", as detailed below. The means disclosed by Webb, and Kuck produce substantially the same results as the corresponding element disclosed in the specification, in substantially the same way.

Webb discloses:

- "means for obtaining a representation of at least one class..." Webb (Abstract), "a computer system includes a virtual machine supporting an object-oriented environment, in which programs to run on the virtual machine are formed from classes loaded into the virtual machine by a class loader."

-"means for interpreting and means for instantiating the representation as a class definition..." See Webb FIG. 1, Java APP 50, JVM 40; FIG. 2, Class loader, cache and storage; FIG. 3 Loading [instantiating]; col. 1: 53-67.

-"means for executing the child runtime system process..." at FIGs. 1-4.

Kuck discloses:

-"means for cloning the memory space...", [0001-0008]; [0007], "Initializing can include storing the virtual machine in a memory area, and copying a template image into the memory area"; [0029-0032]; [0042], PAVM is generated and initialized; [0064-0065].

Regarding **claim 2**, Webb discloses (5: 55-65) a cache checker to determine whether the instantiated class definition is available in a local cache associated with the master runtime system process. "The class loader cache therefore allows each class loader to check whether it has loaded a particular class ..."

Regarding **claim 3**, Webb discloses (4: 41-58) a class locator to locate the source definition if the instantiated class definition is unavailable in the local cache. "If the response from all of the class loader is negative, then the JVM walks down the hierarchy (walks the class loader hierarchy), with the Primordial class loader first attempting to locate the class, by searching in the locations specified in its class path definition."

Regarding **claim 4**, Webb does not disclose a class resolver to resolve the class definition. But Kuck discloses [0064] a class resolver to resolve the class definition. "Initializing a PAVM can be an expensive operation, as it may involve loading, verifying, and resolving several classes (e.g., Java system classes), as well as executing numerous static initializers in system classes."

Therefore, it would have been obvious to a person of ordinary skill in the art at the time the invention was made to modify Webb's teaching by adding a class resolver to resolve the class definition as taught by Kuck in order to reduce the overhead through the use of a pre-initialized master PAVM rather than initializing a new PAVM (Kuck, [0064]) .

Regarding **claim 5**, Webb discloses (9: 34-59) at least one of a local and remote file system to maintain the source definition as a class file. "... The JVM would effectively maintain a pool into which the returned class loader instance and its associated classed would be added."

Regarding **claim 6**, Webb does not disclose a process cloning mechanism to instantiate the child runtime system process by copying the memory space of the master runtime system process into a separate memory space for the child runtime system process. However, Kuck discloses [0064-0065] a process cloning mechanism to instantiate the child runtime system process by copying the memory space of the master runtime system process into a separate memory space for the child runtime system process initialization overhead can be reduced through the use of a pre-initialized "master" PAVM. Rather than initializing a new PAVM from scratch, the memory block of the master PAVM can simply be copied into the memory block of the new PAVM. Copying the template image of the master PAVM's memory block into the memory block of a new PAVM enables the new PAVM to start running in an already-initialized state.., storing type information (i. e., the runtime representation of loaded classes) in a section of shared memory that can be accessed by all the PAVMs...technique can reduce the overhead for class loading, verification, and resolution incurred by each PAVM, and can be especially useful if used to share the byte code of system classes that are likely to be used by every user context."

The feature of providing a process cloning mechanism to instantiate the child runtime system process by copying the memory space of the master runtime system process into a separate memory space for the child runtime-system process would be obvious for the reasons set forth in the rejection of claim 1.

Regarding **claim 7**, Webb fails to disclose the master runtime system process is caused to sleep relative to receiving the process request. However, Kuck discloses [0064-0066], "After a PAVM has been initialized, the PAVM can be used to process user requests from the corresponding user session. When a user request from the corresponding user session is received (504), an available process from the pool of OS processes allocated to the server can be selected to process the request (506). The PAVM of the user session that sent the request is then bound to the selected process (508)." The master runtime system process is caused to sleep relative to receiving he process request. This reduces overhead by storing information is a section of shared memory, where it sleeps until needed.

The feature of providing wherein the master runtime system process is caused to sleep relative to receiving the process request would be obvious for the reasons set forth in claim 1 above.

Regarding **claim 8,** Webb discloses (4: 27-29), the object-oriented program code is written in the Java programming language. "The fundamental unit of a Java program is the class, and thus in order to run any application the JVM must first load the classes forming and required by that application... " Analogously, Kuck discloses [0032] of Process Attachable Virtual Machines with specific reference to a Java source program.

Regarding **claims 11-17**, limitations are similar to claims 2-8 above respectively. See claim chart for mapping of limitations.

Regarding **claim 19**, Webb teaches (FIG. 1 & 3: 56-4: 12) software loaded into memory [computer readable storage medium holding code…]…system runs an operating system (OS) 30 on top of which is provided a Java virtual machine (JVM) 40…functions itself as a virtual operating system…

Regarding **claims 21 and 22**, as an example, Webb disclosed (8: 5-12) a class to be defined as final (setting management parameters) to prevent subclasses from overriding it. Kuck analogously disclosed [0020-0021], "Storing the context of a user session, as well as the heap and stack of a PAVM, in shared memory essentially makes the operation of binding or attaching the PAVM to an OS process a non-operation--the process simply has to map the appropriate section of shared memory into its address space. Data does not need to be actually moved or copied. Shared OS resources such as I/O resources can be implemented using descriptor passing between processes. For Java VMs, that only requires a few modifications to be made in a non-Java part of a class library. Alternatively, the task of managing OS resources can be delegated to a resource manager [resource controller]. Attaching PAVMs to and detaching PAVMs from processes can thus be carried out in a cost-effective manner. Consequently, a small number of processes can be used to efficiently process requests from a large number of user sessions."

Dike as a primary reference

Re. Ground #2 or Ground #3:

Claims 1-7, 10-16, and 19-22 are rejected under 35 U.S.C. 102(b) as anticipated by or, in the alternative, under 35 U.S.C. 103(a) as obvious over Dike, in view of Steinberg. See pertinent teachings found in Request a p. 36 and Exhibit 18, which are incorporated by reference. **This rejection (for Ground 2 or Ground 3) is adopted as modified**, with additional comments added by Examiner. The modification does not reject claims 8 and 17, as they are not proposed by Third Party Requester in the claim chart, Exhibit 18.

Dike fairly discloses the "copy-on-write" functionality. The Steinberg reference (p. 16, 21) is introduced to describe the copy-on-write mechanism of the Linux Fork() system call, as disclosed by Dike.

Regarding independent **claims 1, 10, and 20,** Dike discloses (§6, §4.2) one or more virtual machines (with virtual kernel / user-mode virtual machine, user-mode kernel, virtual processor and virtual memory; methods running on a system / apparatus) running on a physical machine (host processor with physical kernel memory and address space memory, using a Linux operating system /with Linux kernel/host kernel/native kernel). Dike discloses (§2.1-processor and memory management implemented virtually as a "runtime environment"; §2.2-class preloader/boot loader, initialization, memory management; § 3-applications run in the virtual machine; §5.3.3-memory management, construct and modify mm_struct objects; § 2.3, new process created, initialization) system, method and apparatus of the limitations of independent claims. Dike (§2.3) runs a "fork or clone" process to duplicate virtual machine system processes [to interpret and to instantiate the representation as a class definition in a memory space of the master runtime system process]. "Since the generic kernel arranged for the new address space to be a copy of the parent address space, and the new process has the same registers as the old one..."

Dike teaches (§ 2.1), "Each process within a virtual machine gets its own process in the host kernel." To save address space, user-mode kernel (virtual) address space and host (physical) address space share kernel data. "This converts a copy-on-write segment of the address space into a shared segment." (§ 2.3) The fork () method is used to create a new process including new address space.

To the extent that clarifying evidence is needed to support the "copy-on-write" teachings, one of ordinary skill in the art at the time of the invention would have looked to other art describing the Linux fork( ) system call and copy-on-write, including the Steinberg reference.

Specifically, it would have been obvious to one of ordinary skill in the art at the time of the invention to combine Dike's disclosure, directed to virtual machines in a Linux environment and making using of a fork( ) system call, with Steinberg's disclosure of the Linux fork( ) system call and its explicit use of the copy-on-write mechanism for supporting evidence that "copy-on-write" was known in the art.

The Steinberg reference describes (p. 21, see also Steinberg at 16 describing the Linux clone functionality) the copy-on-write mechanism of the Linux fork( ) system call, as disclosed by Dike: "The normal means by which a Linux process can create new tasks is the fork system call. This system call creates an exact copy of the calling process, which then becomes the calling process' child process. The created child process inherits copies of the parent process data space, heap and stack, which are copied on demand using a copy-on-write mechanism."

Regarding the "means for" limitations of claim 20:

In support of the conclusion that the prior art elements are an equivalent, the combination of prior art elements are shown to perform an equivalent "means for", as detailed below. The means disclosed by Dike produce substantially the same results as the corresponding element disclosed in the specification, in substantially the same way.

Dike discloses:

- "means for obtaining a representation of at least one class…" 2.1-2.2, virtual machine booting, initialization, kernel, memory

-"means for interpreting and means for instantiating the representation as a class definition…" Dike at 2.2-2.3, fork / clone process, address space, kernel booting process and initialization

-"means for executing the child runtime system process…" Dike at 2.3, process in the user space virtual machine; pid (process id) of new process; new process scheduled to run

-"means for cloning the memory space…", Dike at 2.3, generic kernel creates the task structure for the new process, data structures such as threads and stack, registers, and system call handler

Regarding **claims 2 and 11**, Dike discloses the "double caching of disk data" [a cache checker to determine whether the instantiated class definition is available in a local cache associated with the master runtime system process]. Dike discloses (§ 5.3.3), "Another area to look at is the double-caching of disk data. The host kernel and the user-mode kernel both implement buffer caches, which will contain a lot of the same data." Dike discloses (§ 2.1), "SIGIO is used to deliver the other asynchronous events that the kernel must handle, namely device interrupts. The console driver, network drivers, serial line driver, and block device driver use the Linux asynchronous I/O mechanism to notify the kernel of available data." Dike at § 2.7,"Linux implements demand loading of process code and data."

Regarding **claims 3 and 12**, Dike discloses a class locator to locate the source definition if the instantiated class definition is unavailable in the local cache. Dike at § 5.3.3., "The host kernel and the user-mode kernel both implement buffer caches." Therefore the system of Dike discloses a system which looks to a first instantiated class definition, then to a source definition if the instantiated version is unavailable.

Dike at § 5.3.3, "In addition, access to the host memory context switching mechanism would probably speed up context switches greatly. The ability to construct and modify mm_struct objects from user-space and switch an address space between them would eliminate the potential address space scan from context switches. Another area to look at is the double-caching of disk data. The host kernel and the user-mode kernel both implement buffer caches, which will contain a lot of the same data." Dike at 2.1, "SIGIO is used to deliver the other asynchronous events that the kernel must handle, namely device interrupts. The console driver, network drivers, serial line driver, and block device driver use the Linux asynchronous I/O mechanism to notify the kernel of available data." Dike at § 2.7, "Linux implements demand loading of process code and data."

Regarding **claims 4 and 13,** Dike provided a class resolver to resolve the class definition. Dike discloses the implementation of a virtual machine with broad capabilities. "A Linux virtual machine is capable of running nearly all of the applications and services available on the host architecture." Dike at Abstract. This would include the resolution of class definitions.

"In addition, access to the host memory context switching mechanism would probably speed up context switches greatly. The ability to construct and modify mm_struct objects from user-space and switch an address space between them would eliminate the potential address space scan from context switches. Another area to look at is the double-caching of disk data. The host kernel and the user-mode kernel both implement buffer caches, which will contain a lot of the same data." Dike at § 5.3.3.

"SIGIO is used to deliver the other asynchronous events that the kernel must handle, namely device interrupts. The console driver, network drivers, serial line driver, and block device driver use the Linux asynchronous I/O mechanism to notify the kernel of available data." Dike at § 2.1. "Linux implements demand loading of process code and data." Dike at § 2.7.

Regarding **claims 5 and 14**, Dike provided at least one of a local and remote file system to maintain the source definition as a class file. Dike discloses a virtual machine and a host system. The virtual machine has its own "memory management, process management, and fault support." Dike at § 2.1. Thus, class files can be stored in both locations, i.e., the host or local system and the virtual machine or remote system.

"Serial lines The serial line driver allocates a pseudo-terminal. Users wanting to connect to the virtual machine via a serial line can do so by connecting to the appropriate pseudo-terminal with a terminal program. Dike at § 1.2.

Network devices There are two network device drivers. The old network driver communicates with the host networking system through a slip device in the host. The virtual machine's side of the connection is a pseudo-terminal in the host which appears as a network device inside. There is also a newer network driver which uses an external daemon to pass Ethernet frames between virtual machines. This daemon can also attach this virtual network to the host's physical Ethernet by way of an ethertap device. With an appropriate packet forwarding policy in the daemon, the virtual Ethernet can be transparently merged with the physical Ethernet, totally isolated from it, or anything in between." Dike at § 1.2.

"SIGIO is used to deliver the other asynchronous events that the kernel must handle, namely device interrupts. The console driver, network drivers, serial line driver, and block device driver use the Linux asynchronous I/O mechanism to notify the kernel of available data." Dike at § 2.1.

"Linux implements demand loading of process code and data." Dike at § 2.7.

"In addition, access to the host memory context switching mechanism would probably speed up context switches greatly. The ability to construct and modify mm_struct objects from user-space and switch an address space between them would eliminate the potential address space scan from context switches." Dike at § 5.3.3.

Regarding **claims 6 and 15**, Dike provided a process cloning mechanism to instantiate the child runtime system process by copying the memory space of the master runtime system process into a separate memory space for the child runtime system process. Dike runs a "fork or clone" process to create a process in the host corresponding to a process in the user space virtual machine, i.e., cloning a virtual machine system process. Dike at § 2.3. "Since the generic kernel arranged for the new address space to be a copy of the parent address space, and the new process has the same registers as the old one, except for the zero return value from the system call, it is a copy of its parent." Dike at § 2.3.

"Each process within a virtual machine gets its own process in the host kernel. Even threads sharing an address space in the user-mode kernel will get different address spaces in the host. Even though each process gets its own address spaces, they must all share the kernel data. Unless something is done to prevent it, every process will get a separate, copy of the kernel data. So, what is done is that the data segment of the kernel is copied into a file, unmapped, and that file is mapped shared in its place. This converts a copy-on-write segment of the address space into a shared segment." Dike at § 2.1.

Regarding **claims 7 and 16**, Dike provided a system wherein the master runtime system process is caused to sleep relative to receiving the process request, as explicitly disclosed below.

"If a process sleeps ...the tracing thread performs the switch by stopping the old process and continuing the new one. The new process returns from the context switch that it entered when it last ran and continues whatever it was doing." Dike at § 2.5.

Regarding **claim 19**, Dike runs on a host system, and includes memory management, process management and fault support (§ 2.7) [computer readable storage medium holding code for performing...] See (§ 2.2) physical memory area, mem_init, paging_init for memory management.

Regarding **claims 21 and 22**, Dike provided an apparatus with a resource controller to set operating system level resource management parameters on the child runtime system process. Dike discloses a system wherein the access to host resources is limited: "They have no access to any host resources other than those explicitly provided to the virtual machine." Dike at § 1.1. See § 4.2 for discussion of physical machine limiting access to its resources by virtual machine processes (sandboxed processes).

Bryant as a Primary Reference

Re. Ground #4:

Claims 1-8, 10-17, and 19-22 are rejected under 35 U.S. C. § 103(a) as obvious over Bryant in view of APA-Bach. See pertinent teachings found in Request 03/01/2010, p. 37 and Exhibit 19, which are incorporated by reference. This rejection is **adopted**, with additional comments added by Examiner. Although a rejection for **claims 9 and 18** is found in Exhibit 19 Claim Chart, the claims are not requested for reexamination.

Regarding **claims 1, 10, and 20**, Bryant discloses a system (method and apparatus) for dynamic preloading of classes through, memory space cloning of a master runtime system process. Bryant notes from the outset that it is faster to connect up to the already running Java server and have the already running Java server fork a child server" than it is to execute the desired classes from scratch. Bryant, Abstract.

Thus one of ordinary skill in the art seeking to improve or accelerate the performance of a Java operation would look to combine the disclosure of Bryant with the copy-on-write technology that was very well known in the art, as evidenced by the APA-Bach text.

"The Java server invokes the Java virtual machine and preloads all potentially needed objects files during initialization of the Java virtual machine to speed up the actual execution of a particular Java application. The Java server accomplishes the execution of a particular Java application by forking itself and then having the child Java server run the Java class files in the already loaded Java virtual machine for the specific Java CGI-BIN script." Bryant, col.2, ll. 46-48. See Bryant FIG. 3, processor & memory.

Bryant discloses a class preloader to obtain a representation of at least one class from a source definition provided as object-oriented program code. The apparatus and method of Bryan works by "preload[ing] all potentially needed object files," "to speed up the actual execution of a particular Java application." Bryant, col. 2, 11. 46-48.

"First, the Java server 160 is initialized at step 161, and then waits to be called. The initialization step 161 includes starting the Java virtual machine and loading the standard class files needed for Java application execution." Bryant, FIG. 8 & col. 6, l. 66-col. 7, 1. 4.

Bryant discloses a master runtime system process to interpret and to instantiate the representation as a class definition in a memory space of the master runtime system process. The master runtime system process is merely the standard creation of a process from a set of instructions (interpret instructions and instantiate a class object at initialization and loading of the standard class files needed for Java application execution).

Bryant discloses a runtime environment to clone the memory space as a child runtime system process responsive to a process request and to execute the child runtime system process.

The execution of a child runtime process here is simply the well-known fork system call, which creates a duplication of a master runtime process [runtime environment to clone the memory space as a child runtime system process responsive to a process request and to execute the child runtime system process]: "Java server 160 forks immediately to create a child Java server 180...the pipe connection from the application program 140 is connected to both the parent Java server 160 and to the child Java server 180. The child Java server 180 receives the program name execution arguments and environmental arguments sent on the pipe connection, sets up the file descriptors, maps to the requested class and method, executes the class and method, and writes the output to a stdout; which is then returned to application program 140." Bryant, col. 5, 11. 9-14 & col.7, 11. 7-14.

To the extent that Bryant failed to explicitly disclose the "copy-on-write" method and the "fork" system call, APA Bach provided details: "The only way for a user to create a new process in the UNIX operating system is to invoke the fork system call." APA-Bach at 192. "The copy-on-write bit, used in the fork system call, indicates that the kernel must create a new copy of the page when a process modifies its contents." APA-Bach at 287.

Bach disclosed , "9.2.1.1 Fork in a Paging System As explained in Section 7.1, the kernel duplicates every region of the parent process during the fork system call and attaches it to the child process. Traditionally, the kernel of a swapping system makes a physical copy of the parent's address space, usually a wasteful operation, because processes often call exec soon after the fork call and immediately free the memory just copied. On the System V paging system, the kernel avoids copying the page by manipulating the region tables, page table entries, and pfdata table entries: It simply increments the region reference count of shared regions .... The page can now be referenced through both regions, which share the page until a process writes to it. The kernel then copies the page so that each region has a private version. To do this, the kernel turns on the 'copy on write' bit for every page table entry in private regions of the parent and child processes during fork. If either process writes the page, it incurs a protection fault, and in handling the fault, the kernel makes a new copy of the page for the faulting process. The physical copying of the page is thus deferred until a process really needs it." APA-Bach at 289-290.

Bryant in combination with APA-Bach discloses a copy-on-write process cloning mechanism to instantiate the child runtime system process by copying references to the memory space of the master runtime system process into a separate memory space for the child runtime system process, and to defer copying of the memory space of the master runtime system process until the child runtime system process needs to modify the referenced memory space of the master runtime system process.

The copy-on-write technology was widely known in the art at the time of the purported invention of the '720 patent. The APA-Bach reference clearly discloses the utility and application of the copy-on-write method of cloning. It would have been obvious to one of ordinary skill in the art looking for a means to streamline and accelerate a Java machine would have been motivated to combine the Bryant reference with APA-Bach to minimize resource usage by copying data only when needed, i.e., employing copy-on-write.

The “means for” language of Claim 20 is reviewed with regards to 35 CRF 112 paragraph 6:

In support of the conclusion that the prior art elements are an equivalent, the combination of prior art elements are shown to perform an equivalent "means for", as detailed below. The means disclosed by Bryant produce substantially the same results as the corresponding element disclosed in the specification, in substantially the same way.

Bryant discloses:

-"means for obtaining a representation of at least one class..." Bryant, 2: 46-48; 6: 66 – 7: 4; object files are a representation of a class; See FIGs. 2, 3, & 4

-"means for interpreting and means for instantiating the representation as a class definition..." Bryant, 6: 66-7: 4, Java virtual machine, loaded class files, application execution

-"means for cloning the memory space..." Bryant 5: 9-14, fork process; FIG. 3 hardware

-"means for executing the child runtime system process..." Bryant FIG. 3; 2: 46-53, "Java server invokes the Java virtual machine and preloads objects files during initialization, forking; 4: 18-30, processor, storage device, memory, browser program is the software that interacts with the server

Regarding **claims 2-5, and 11-14**, Bryant discloses (col. 2, ll. 46-49; col. 7, ll. 1-4), "The Java server invokes the Java virtual machine and preloads [class loader] all potentially needed objects [class resolver] files during initialization [into local cache, determine whether instantiated class definition is available in local cache else load it] of the Java virtual machine to speed up the actual execution of a particular Java application." Bryant discloses (Abstract; col. 5, ll. 4-18), When an application 140 is to be executed, an object file calls the Java server process that forks itself to create a child Java server. The child Java server 180 sets up file descriptors, maps to the requested class and method, and executes the already loaded classes and methods (located in local cache of Java server 160). Within the newly created child server, a proxy object file calls a Java server daemon process to execute code that is stored in a single Java server 160. See FIG. 3, local file system and remote file system [memory 71, application program 140 executed on child Java server 180 [file system] by calling to class and method code stored in Java server 160 [file system],

Regarding **claims 6 and 15**, the process cloning mechanism of Bryant is useful to load the object file only once and then execute the fork system call to create child runtime processes. "The Java server is accessed by an object file (proxy), that is setup to access the correct Java server process. Next, when an application is to be executed, the object file calls the Java server process that forks itself and then has the child server run the already loaded classes and methods. Thus, the Java classes and methods are loaded only once when the Java virtual machine is started. With large classes and methods, it is faster to connect up to the already running Java server and have the already running Java server fork a child server to execute the correct classes and methods than it is to start and load the Java virtual machine, and execute the original classes and methods." Bryant, Abstract.

Regarding **claims 7 and 16,** Bryant discloses the master runtime system process is caused to sleep relative to receiving the process request. The listening pipe of Bryant is idle until the Java server receives a request for service call, as explained below.

"The initialization step 161 includes starting the Java virtual machine and loading the standard class files needed for Java application execution. A listening pipe is setup to receive requests for service calls and the Java server waits to receive a call in step 162." Bryant, col.7, 11.1-6. The Java server 160 forks to create a child Java server 180. The child Java server 180 sets up and executes, prior to exiting. While the child Java server 180 is executing and prior to the child Java server termination, the Java server 160 "sleeps." See 5: 4-41; FIG. 8 and 6: 66--7:12, "The Java server 160 then waits...") Bryant also discloses (7: 16-18), "The process is returned to the wait state at step 163 to wait for the next pipe connection to be established."

Regarding **claims 8 and 17**, Bryant discloses object-oriented program code is written in the Java programming language. "The process of the Java server 160...includes starting the Java virtual machine and loading the standard class files needed for Java application execution." Bryant, col.6, 1.65-col.7, 1.4.

Regarding **claim 19**, Bryant discloses a computer readable storage medium in FIG 3, storage #62.

Regarding **claims 21 and 22**, Bryant, either by itself or in combination with APA-Bach, discloses a resource controller to set operating system level resource management parameters on the child runtime system process. Bryant creates a child server by forking the parent server. This child server operates under resource management controls, executing specific tasks as instructed by the Java server. "The child Java server 180 is initialized at step 181. The child Java server 180 receives the information sent on the pipe connection created by the application program 140 at step 182. The child Java server 180 then maps, at step 183, to the specified application (i.e., class and method) identified in the information that was communicated over the pipe connection and received at step 182. The child Java server then executes the specified application (i.e. class and method) using the specified program name, execution arguments, and environment arguments present in the information received on the pipe connection at step 184 [where operating system level resource management parameters are set on the child runtime system process]." Bryant, col. 7, 11. 41-52.

Re. Ground #5:

Claims 1-8, 10-17, and 19-22 are rejected under 35 U.S. C. § 103(a) as obvious over Bryant in view of Traut. See pertinent teachings found in Request 03/01/2010, p. 37 and Exhibit 20, which are incorporated by reference. This rejection is **adopted**, with additional comments added by Examiner.

Regarding independent **claims 1, 10, and 20** (system, method, and apparatus), Bryant disclosed features that are mapped to claim limitations as noted above.

To the extent that Bryant failed to provide details on the "copy-on-write" process and the fork() method, these teachings are disclosed by Traut.

"The present invention in one implementation provides a method for increasing the efficiency of virtual machine processing. One step of the method is providing a parent virtual machine. Another step is temporarily suspending the parent virtual machine. Another step is forking the parent virtual machine to create a child virtual machine at a new location." Traut at ¶ [0012]

"Referring now to FIGS. 2 and 3, "forking" is a term used by UNIX programmers to describe the duplication of a UNIX process and its address space. Both the original process and the fork are then allowed to run as independent processes from the forking point. The implementation of forking often involves a technique called "copy on write" in which case all memory pages in both address spaces are marked "write protected". When either the original or the forked process writes to a page, a copy is made so that each process has its own copy. Pages that are not modified can continue to be shared between the two processes. This technique not only saves on memory resources, but it also makes forking much faster than otherwise possible." Traut at ¶ [0026]

"Shown in FIG. 2 is a flow diagram of a method 20 for forking a virtual machine. In step 202, a virtual machine parent is suspended. In step 204, a copy or "snapshot" is made of all of the pieces of the parent virtual machine other than the memory of the parent virtual machine. In step 206, the snapshot is moved to a new location, i.e. a location other than the location of the parent. Moving the snapshot to a new location creates a new virtual machine child." Traut at ¶ [0029]

"In step 306, it is determined whether or not the child virtual machine is accessing the memory of the parent virtual machine. If the child is accessing the parent's memory, in step 308, the child virtual machine is temporarily suspended and the piece of the parent's memory required by the child is sent from the parent to the child. If the child is not accessing the parent's memory, in step 31 O, pieces of the parent's memory that are not actively required by the child may be sent from the parent to the child. If step 308 or step 310 is completed, the method proceeds to step 312." Traut at ¶ [0031].

As discussed in the Request for Reexamination, the copy-on-write technology was widely known in the art at the time of the purported invention of the '720 patent. Therefore, it would have been obvious for one of ordinary skill in the art at the time of the invention to streamline and accelerate a Java machine by using the "copy-on-write" and fork () methods to efficiently use memory resources.

See "means for" analysis in the Bryant / APA Bach rejection above.

<u>Srinivasan as a Primary Reference</u>

Re. Ground #6:

Claims 1-8, 10-17, and 19-22 are rejected under 35 U.S. C. § 103(a) as obvious over Srinivasan in view of APA-Bach. See pertinent discussions found in Request 03/01/2010, p. 37 and Exhibit 21, which are incorporated by reference. This rejection is **adopted**, with additional comments added by Examiner.

Per independent **claims 1, 10, and 20** (system, method, apparatus), Srinivasan discloses object oriented programming and its usefulness in conjunction with Perl code. Srinivasan at 101. Srinivasan notes that, in Perl, a "class is a package." Srinivasan at 389. Srinivasan then discusses two methods for the creation of a package from source definition provided as object oriented program code. See id. at 389-390 (discussing creation of the "employee" class). In order to preload the class, Srinivasan discloses specific functional code:

"Using object package:

```
use Employee;
$emp = Employee->new ("Ada", 3 5);

$emp->set_salary(1000);
```

See id. at 390.

Srinivasan discloses an "in-memory cache of objects." Srinivasan at 178.

Srinivasan provides a discussion of the merits of Java and Perl object languages (Srinivasan at 98), and object oriented software methodology (Srinivasan at 99). "Objects of a certain type are said to belong to a class." Srinivasan at 101. Srinivasan notes that opcodes are similar in concept to machine code; while machine code is executed by hardware, opcodes (sometimes called byte-codes or p-code) are executed by a virtual machine. Srinivasan notes that Java and Perl are both examples of interpreters. Srinivasan at 323-324.

In order to create multiple threads of control, Srinivasan invokes the fork( ) system call to create "process-level parallelism." Srinivasan at 193-194. This creates a new process, called the child process. " [instantiate the representation as a class definition in a memory space of the master runtime system process] The newly created child process meanwhile has a copy of its parent's environment [parent runtime system process / master runtime system process] and shares all open file descriptors." See id. You can use multiple interpreters to enforce completely isolated namespaces. Each interpreter has its own "main" package and its own tree of loaded packages." Srinivasan at 323.

"Perl... supports fork, the way to get process-level parallelism. The server process acts as a full-time receptionist...spawns a child process... The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors...When the child is done...it simply exits. Each process is therefore dedicated to its own task and doesn't interfere with the other. (See code segment example of a forking server.) The fork call results in two identical processes-the parent and child-starting from the statement following the fork. The parent gets a positive return value, the process ID ($pid) of the child process. Both processes check this return value and execute their own logic; the main process goes back to 'accept'... Srinivasan at 194-195.

To the extent that Srinivasan failed to provide explicit details related to the 'copy-on-write' method, APA-Bach provided such teachings. "The only way for a user to create a new process in the UNIX operating system is to invoke the fork system call." APA-Bach at 192.

Srinivasan in view of APA-Bach provided the use of fork in Unix/Linux, which includes a copy-on-write process cloning mechanism to instantiate the child runtime system process by copying references to the memory space of the master runtime system process into a separate memory space for the child runtime system process, and to defer copying of the memory space of the master runtime system process until the child runtime system process needs to modify the referenced memory space of the master runtime system process.

"The copy-on-write bit, used in the fork system call, indicates that the kernel must create a new copy of the page when a process modifies its contents." APA-Bach at 287. "9.2.1.1 Fork in a Paging System As explained in Section 7.1, the kernel duplicates every region of the parent process during the fork system call and attaches it to the child process. Traditionally, the kernel of a swapping system makes a physical copy of the parent's address space, usually a wasteful operation, because processes often call exec soon after the fork call and immediately free the memory just copied. On the System V paging system, the kernel avoids copying the page by manipulating the region tables, page table entries, and pfdata table entries: It simply increments the region reference count of shared regions .... The page can now be referenced through both regions, which share the page until a process writes to it. The kernel then copies the page so that each region has a private version. To do this, the kernel turns on the 'copy on write' bit for every page table entry in private regions of the parent and child processes during fork. If either process writes the page, it incurs a protection fault, and in handling the fault, the kernel makes a new copy of the page for the faulting process. The physical copying of the page is thus deferred until a process really needs it." APA-Bach at 289-90.

Therefore it would have been obvious, to one of ordinary skill in the art at the time of the invention, to modify the teaching of Srinivasan, with the explicit teachings of APA-Bach to include a description of the copy-on-write bit. The fork( ) system call disclosed by Srinivasan was commonly used in conjunction with the copy-on-write mechanism to further streamline the impact on system memory.

The "means for" language of Claim 20 is reviewed with regards to 35 CRF 112 paragraph 6:

In support of the conclusion that the prior art elements are an equivalent, the combination of prior art elements are shown to perform an equivalent "means for", as detailed below. The means disclosed by Srinivasan produce substantially the same results as the corresponding element disclosed in the specification, in substantially the same way.

Srinivasan discloses:

-"means for obtaining a representation of at least one class..." p. 178, in-memory cache of objects; p. 98-101, Java code packages and classes;

-"means for interpreting and means for instantiating the representation as a class definition..." p. 323, interpreters; p. 321, running system

- "means for cloning the memory space...", pp. 193-194, multiple threads of control, fork() system call, child process has copy of parent's environment

Regarding **claims 2 and 11**, as an example of Srinivasan's disclosure of "a cache checker to determine whether the instantiated class definition is available in a local cache associated with the master runtime system process": "Adaptor File" function, which "converts [a] query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification." Srinivasan at 179. "This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." Srinivasan at 178.

Regarding **claims 3 and 12**, Srinivasan discloses an Adaptor File function that searches a file and matches objects against a query [a class locator to locate the source definition if the instantiated class definition is unavailable in the local cache]. The file adaptor has an attribute called all instances, a hash table of all objects given to its store method (and indexed by their _id), as shown in Figure 11-2. Srinivasan at 179-180.

Regarding **claims 4 and 13**, Srinivasan discloses “run-time binding” [class resolver] of class methods. Srinivasan at 107-108. As another example: “Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more attributes are added to the object implementation. The schema is said to have evolved. The framework has to be able to reconcile [resolver] old data with newer object implementations. Srinivasan at 179-180.

Regarding **claims 5 and 14**, as an example of a local and remote file system to maintain the source definition as a class file, Srinivasan discloses, for example, that "[i]n a typical client/server system, the server has the "real" objects. But the system is written in such a way that a client can remotely invoke a method of the object, with familiar OO syntax. For example, if a client program wants to invoke a method on a remote bank account, it should be able to say something like this:..." Srinivasan at 134.

Regarding **claims 6 and 15**, as an example of process cloning mechanism to instantiate the child runtime system process by copying the memory space of the master runtime system process into a separate memory space for the child runtime system process, Srinivasan discloses, for example creating multiple interpreters. Srinivasan at 323. Srinivasan invokes the fork( ) system call to create "process-level parallelism." This creates a new process, called the child process. "The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors." Srinivasan at 193-195.

Regarding **claims 7 and 16**, Srinivasan discloses that after forking / creating the child process, "the main process goes back to 'accept'…" (sleep) Srinivasan at 195.

Regarding **claims 8 and 17**, Srinivasan discloses Java object-0riented programming language. Srinivasan at 98 and 323-324.

Regarding **claim 19**, Srinivasan discloses an in-memory cache [computer readable storage medium holding code for performing the method] of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." Srinivasan at 178.

Regarding **claims 21 and 22**, Srinivasan discloses a server process that manages resources. Srinivasan at 195. As an example, the fork() method and garbage collection method are used to manage resources. Srinivasan at 112 and 179-180.

Sexton as a Primary Reference

Re. Ground #7:

Claims 1-8, 10-17, and 19-22 are rejected under 35 U.S. C. § 103(a) as obvious over Sexton in view of Bugnion. See pertinent discussions in Request 03/01/2010, p. 37 and Exhibit 22, which are incorporated by reference. This rejection is **adopted**, with additional comments added by Examiner.

Regarding **claims 1, 10, and 20**, Sexton discloses "the use of a shared state area [that] allows the various VM instantiations to share class definitions and other resources" [dynamic preloading of classes through memory space cloning of a master runtime system process]. Sexton, col. 5, ll. 53-57. "Each VM instance has read-only access to the data that has been loaded into the shared state area, and therefore the VM instances do not contend with each other for access rights to that data. According to one embodiment, the shared state area is used to store loaded Java classes." Sexton, col. 8, ll. 45-48. "[t]he non-session-specific data for the class, including the methods, method table and fields, are not duplicated in the session memory for each VM instance. Rather, all VM instances share read-only access to a single instantiation of the class, thus significantly reducing the memory requirements of VM instances (the user-session memory requirements)." Sexton, col. 8, ll. 55-61.

"A virtual machine is software that acts as an interface between a computer program that has been compiled into instructions understood by the virtual machine and the microprocessor (or "hardware platform") that actually performs the program's instructions [method]. Once a virtual machine has been provided for a platform, any program compiled for that virtual machine can run on that platform." Sexton, col. 2, 11. 35-42. See "system" and "apparatus" at Sexton, FIG. 1.

Sexton provided a class preloader, e.g., the shared "state information," to obtain a representation of at least one class from a source definition provided as object-oriented program code: "...state sharing tends to reduce the resource overhead required to concurrently service the requests..." Sexton, col. 3, 11. 51-63. "The database instance memory 220 is a shared memory area for storing data that is shared concurrently by more than one process... used to store the read-only data and instructions (e.g. bytecodes of JAVA classes) that are executed by the server processes 213 and 217. The database instance memory 220 is typically allocated and initialized at boot time of the database system 200, before clients connect to the database system 200." Sexton, col. 6, 11. 59-67.

"When a database session is created, an area of the database memory 202 is allocated to store information for the database session...session memories...are memories used to store static data, i.e., data associated with a user that is preserved for the duration of a series of calls, especially between calls issued by a client during a single database session. JAVA class variables are one example of such static data." Sexton, col. 7, 11. 1-11.

Sexton provided a master runtime system process to interpret and to instantiate the representation as a class definition in a memory space of the master runtime system process. "Techniques are provided for instantiating separate Java virtual machines for each a session established by a server. "The separate VM instances can be created and run, for example, in separate units of execution that are managed by the operating system of the platform on which the server is executing [master runtime system]. For example, the separate VM instances may be executed either as separate processes, or using separate system threads. Because the units of execution used to run the separate VM instances are provided by the operating system, the operating system is able to ensure that the appropriate degree of insulation exists between the VM instances." Sexton, col. 5, 11. 29-44.

"... the Java virtual machine itself takes the form of a set of global variables accessible to all threads, where there is only one copy of each global variable...in one embodiment of the invention, an entire Java VM instance is spawned for every session made through the server...each Java VM instance is spawned by instantiating a VM data structure in session memory. During execution, the state of a VM instance is modified by performing transformations on the VM data structure associated with the VM instance, and/or modifying the data contained therein... the routines access session-specific variables that are stored within the VM data structure... [clone the memory space as a child runtime system process responsive to a process request and to execute the child runtime] Consequently, the contention for resources that otherwise occurs between threads associated with different sessions is significantly reduced, because those threads are associated with different VM instances." Sexton, col. 7, 1. 61 - col. 8, 1. 18.

It is noted that Sexton disclosed methods for a plurality of VMs to access certain "shared state" data such that data and methods are not duplicated in the session memory for each VM instance,

"In addition, techniques are provided for reducing startup costs and incremental memory requirements of the Java virtual machines instantiated by the server. For example, the use of a shared state area allows the various VM instantiations to share class definitions and other resources. In addition, while it is actively processing a call, each VM instance has two components, a session-duration component and a call-duration component. Only the data that must persist in the VM between calls is stored in the session-duration component [infers persistence via a copy on write technique]. Data that need not persist between calls is stored in the call-duration component, which is instantiated at the start of a call, and discarded at the termination of the call." Sexton, col. 5, 11. 53-65. Sexton failed to explicitly discuss "copy-on-write."

To the extent that Sexton failed to explicitly disclose "copy on write" techniques, Bugnion is relied upon for teaching "copy-on-write mappings to reduce copying and to allow for memory sharing." Bugnion, col. 15, 1. 66 - col. 16, 1. 1.

"The VMM layer also maintains copy-on-write disks that allow virtual machines to transparently share main memory resources and disk storage resources... VMM layer may also comprise a virtual memory resource interface to allow processes running on multiple virtual machines to share memory." Bugnion, col. 6, ll. 29-36.

"Disco's copy-on-write disks allow virtual machines to share both main memory and disk storage resources... allows Disco to support a system-wide cache of disk blocks in memory that can be transparently shared between all the virtual machines." Bugnion, col. 14, ll. 55-64.

"Attempts to modify a shared page will result in a copy-on-write fault handled internally by the monitor. Using this mechanism, multiple virtual machines accessing a shared disk end up sharing machine memory. The copy-on-write semantics means that the virtual machine is unaware of the sharing with the exception that disk requests can finish nearly instantly... ...Effectively we get the memory sharing patterns expected of a single shared memory multiprocessor operating system even though the system runs multiple independent operating systems."

"To preserve the isolation of the virtual machines, disk writes must be kept private to the virtual machine that issues them. Disco logs the modified sectors so that the copy-on-write disk is never actually modified. For persistent disks, these modified sectors would be logged in a separate disk partition managed by Disco. To simplify our implementation, we only applied the concept of copy-on-write disks to non-persistent disks and kept the modified sectors in main memory whenever possible." Bugnion, col. 14, l. 66 - col. 15, l. 35.

Sexton in view of Bugnion provided a copy-on-write process cloning mechanism to instantiate the child runtime system process by copying references to the memory space of the master runtime system process into a separate memory space for the child runtime system process, and to defer copying of the memory space of the master runtime system process until the child runtime system process needs to modify the referenced memory space of the master runtime system process. Therefore it would have been obvious, given the goal of reducing session memory by sharing data between multiple Virtual Machines, by one of ordinary skill in the art at the time of the invention, to modify the teachings of Sexton with the Bugnion prior art by only copying necessary files, using the well-known copy on write technology, thereby placing the artisan in possession of the invention.

The "means for" language of Claim 20 is reviewed with regards to 35 CRF 112 paragraph 6:

In support of the conclusion that the prior art elements are an equivalent, the combination of prior art elements are shown to perform an equivalent "means for", as detailed below. The means disclosed by Sexton produces substantially the same results as the corresponding element disclosed in the specification, in substantially the same way.

Sexton discloses:

-"means for obtaining a representation of at least one class..." FIGs. 1, 2, & 3; 2: 36-57, virtual machine software, microprocessor / hardware platform, instruction set, registers, stack, heap; 9: 44-10: 22, hardware overview as shown in FIG. 1.

-"means for interpreting and means for instantiating the representation as a class definition…" 2: 50-52, Java virtual machine can interpret the bytecode; 3: 15, Java virtual machine may spawn [instantiate]; 5: 54-57, memory requirements of the Java virtual machines instantiated by the server, shared state area, shared class definitions and other resources; 6: 29, virtual machine interpreter

- "means for cloning the memory space…", 3: 21-23, server may cause Java virtual machine to spawn [clone] a second thread for executing the second Java program; 8: 1-5, Java VM instance is spawned by instantiating a VM data structure in session memory

-"means for executing the child runtime system process…" 4: 35-36, memory manager; 4: 58-61, separate VM instances can be created and run [executing] in separate units of execution that are managed by the operating system of the platform on which the server is executing; 9: 1-8, a call processed by system thread using VM instance, shared access to shared state area/memory; 9: 26-31, a dispatcher arranges for requests to be executed

Regarding **claims 2 and 11**, Sexton discloses system checks for whether a Virtual Machine instance has been established already and, if not, instantiates one in session memory.

Bugnion further teaches:

"… a global buffer cache that is transparently shared among the virtual machines …" Bugnion, col. 6, 11. 25-29; col. 7, 11. 43-45. "The machines use a directory to maintain cache coherency [use a cache checker to determine whether instantiated class definition is available in local cache], providing to the software the view of a shared-memory multiprocessor with non-uniform memory access times."

Bugnion, col. 8, 11.31-34.

Regarding **claims 3 and 12**, Sexton discloses the process of checking for an instantiation and, if "no VM instances bas been established.., a VM instance for the session is instantiated in session memory." Sexton, col. 6, 11. 2-8. Sexton further suggests, "...each Java VM instance is spawned by instantiating a VM data structure in session memory. During execution, the state of a VM instance is modified by performing transformations on the VM data structure associated with the VM instance [class locator to locate the source definition if the instantiated class definition is unavailable in the local cache], and/or modifying the data contained therein. Specifically, the VM data structure that is instantiated for a particular session is passed as an input parameter to the server routines that are called during that session..." Sexton, col. 7, 1. 61 - col. 8, 1. 18.

Regarding **claims 4 and 13**, Sexton discloses "...user sessions share the state information required by the virtual machine. Such state information includes, for example, the bytecode for all of the system classes... the bytecode being executed for first user in a first thread has access to information and resources that are shared with the bytecode being executed by a second user in a second thread... [a class resolver resolves the class definitions]" Sexton, col. 3, 11. 51-63.

"The database instance memory 220 is a shared memory area for storing data that is shared concurrently by more than one process...The database instance memory 220 is typically allocated and initialized at boot time [resolved] of the database system 200..." Sexton, col. 6, 11. 59-67.

Regarding **claims 5 and 14**, Sexton discloses, "Various forms of computer readable media may be involved... a magnetic disk of a remote computer... main memory 106, from which processor 104 retrieves and executes the instructions. The instructions received by main memory 106 may optionally be stored on storage device 110 either before or after execution by processor 104." [local and remote file system to maintain the source definition as a class file] Sexton, col. 10, 11. 45-60.

Regarding **claims 6 and 15**, Sexton discloses:

"In addition, techniques are provided for reducing startup costs and incremental memory requirements of the Java virtual machines instantiated by the server [cloning the memory space of the master runtime system process into a separate memory space for the child runtime system] ...the use of a shared state area allows the various VM instantiations to share class definitions and other resources... Only the data that must persist in the VM between calls is stored in the session-duration component." Sexton, col. 5, 11. 53-65.

"... according to one embodiment, a data structure, referred to herein as a "java_active_class", is instantiated in session space [child runtime process] to store session-specific values (e.g. static variables) of a corresponding shared Java class. The non-session-specific data for the class, including the methods, method table and fields, are not duplicated in the session memory for each VM instance... " Sexton, col. 8, 11.40-64.

Bugnion provides additional details:

"The VMM layer also maintains copy-on-write disks that allow [cloned] virtual machines to transparently share main memory resources and disk storage resources...The VMM layer may also comprise a virtual memory resource interface to allow processes running on multiple virtual machines to share memory." Bugnion, col. 6, 11. 29-36.

"...share disk and memory resources among virtual machines. Disco's copy-on-write disks allow virtual machines to share both main memory and disk storage resources...allows Disco to support a system-wide cache of disk blocks in memory that can be transparently shared between all the virtual machines." Bugnion, col. 14, 11. 55-64. See plurality of cloned virtual machines at FIG. 4.

"... Attempts to modify a shared page will result in a copy-on-write fault handled internally by the monitor... disk writes must be kept private to the virtual machine that issues them..." Bugnion, col. 14, l. 66 - col. 15, l. 35. "The virtual subnet and networking interfaces of Disco also use copy-on-write mappings to reduce copying and to allow for memory sharing." Bugnion, col. 15, l. 66 - col. 16, l. 1.

Regarding **claims 7 and 16**, Sexton discloses "...various VM instantiations... actively processing a call [master runtime system process is caused to sleep relative to receiving the process request], where the server awaits additional service requests as arranged by a dispatcher. Sexton, col. 9, ll. 24-30. "The server application 120 process is suspended at step 128 until output data is received from the specified application program140." Sexton, col. 6, ll. 18-20.

"After the server process completes its processing of a call from one client, the server process is free to be assigned to respond to the call of another client." [master runtime system process is caused to sleep relative to receiving the process request] Sexton, col. 7, ll. 48-51.

Regarding **claims 8 and 17**, Sexton discloses Java programming language. Sexton, col. 2, ll. 43-65.

Regarding **claim 19**, Sexton discloses computer readable medium to hold the code for executing the method. Sexton, col. 10, ll. 23-60.

Regarding **claims 21 and 22,** Sexton discloses a resource controller. Sexton, col. 6, ll. 59-67, "...managed by the operating system of the platform on which the server is executing... operating system is able to ensure that the appropriate degree of insulation exists between the

VM instances." Also see Sexton, col. 5, ll. 34-35; col. 12. ll. 57-58, "responding to a call...by scheduling, for execution... [resource controller]

Analogously Bugnion discloses resource management: "The unique virtual machine monitor of the present invention virtualizes all the resources of the machine...The monitor manages all the resources... allows multiple copies of potentially different operating systems to coexist on the multiprocessor...The virtual machine monitor schedules the virtual resources (processor and memory) or the virtual machines on the physical resources of the scalable multiprocessor." Bugnion, col. 4, 11. 25-38.

"Although the system looks like a cluster of loosely-coupled machines, the virtual machine monitor uses global policies to manage all the resources of the machine..." Bugnion, col. 4, 11. 51-67.

"The virtual machine monitor (VMM) layer executes directly on the hardware layer and comprises a resource manager that manages the physical resources of the multiprocessor, a processor manager that manages the computer processors, and a hardware emulator that creates and manages a plurality of virtual machines. The operating systems execute on the plurality of virtual machines and transparently share the plurality of computer processors and physical resources through the VMM layer. In a preferred embodiment, the VMM layer further comprises a virtual network device providing communication between the operating systems executing on the virtual machines, and allowing for transparent sharing optimizations between a sender operating system and a receiver operating system. In addition, the resource manager maintains a global buffer cache that is transparently shared among the virtual machines using read-only mappings in portions of an address space of the virtual machines. The VMM layer also maintains copy-on-write disks that allow virtual machines to transparently share main memory resources and disk storage resources, and performs dynamic page migration/replication that hides distributed characteristics of the physical memory resources from the operating systems. The VMM layer may also comprise a virtual memory resource interface to allow processes running on multiple virtual machines to share memory." Bugnion, col. 6, 11. 6-36.

Re. Ground #8:

Claims 1-8, 10-17, and 19-22 are rejected under 35 U.S. C. § 103(a) as obvious over Sexton in view of Johnson. Pertinent discussions are found in Request 03/01/2010, p. 37 and Exhibit 23, and are incorporated by reference. This rejection is **adopted**, with additional comments added by Examiner.

Claim limitations that map to Sexton's teachings are addressed above.

Analogously Johnson discloses:

"When the Factory class creates Persistent Container object 222, the Factory class simultaneously creates Persistent ClassLoader object 244. Each Persistent Container object 222 has its own Persistent ClassLoader object 244 to load class data, in the form of a Class

Encapsulator object containing the class data, into the Persistent Container object in which it is contained." Johnson, col. 13, 11. 36-42.

"The JVM is preferably implemented to include and work with the Persistent Container objects, Persistent ClassLoader objects, Class Encapsulator objects, Object Encapsulator objects and Persistent Handles to create, store and interact with persistent Java objects. [representation of at least one class from a source definition provided as object-oriented program code]..." Johnson, col. 14, 11. 44-57.

"When method findClass( ) is called on Persistent ClassLoader object 244, findClass( ) searches the list of classes contained with Persistent ClassLoader object 244 to determine whether a Class Encapsulator object containing the definition for a specified Java class is already loaded into the Persistent Container object. If a Class Encapsulator object is already loaded into Persistent Container object 222, method findClass simply returns the reference of the class." Johnson, col. 19, 11. 9-20.

Regarding the "copy-on-write process cloning", Johnson more explicitly discloses:

"The preferred embodiment introduces copy on write storage which provides access to static variables. Static variable as defined by a class file are stored at a particular address in SAS copy on write storage. When an instance of a class is created, any static variables defined for that class are created using the definitions stored in the class file in SAS. The present invention allows static variables to be shared among instances of a class running in a particular JVM. However, different JVMs do not access the same static variables. Each JVM has its own copies of any static variables defined for a class of which it has an instance." Johnson, col. 18, ll. 34-44.

Sexton in view of Johnson provides a method for dynamic preloading of classes through memory space cloning of a master runtime system process. Sexton is directed to "reducing startup costs and incremental memory requirements" associated with the instantiation of Java virtual machines; Sexton calls for a "the use of a shared state area [that] allows the various VM instantiations to share class definitions and other resources." Sexton, col. 5, ll. 53-57. And Johnson discloses that when a process needs a persistent object, the persistent object is copied from persistent storage and into real memory." Johnson, col. 3, ll. 4-6.

Given the goal of reducing session memory by sharing data between multiple Virtual Machines, one of ordinary skill in the art at the time of the invention could take the teachings of Sexton in combination with the Johnson prior art and be in possession of the invention. Here, given the goal of the reduction of overhead of Sexton, it would be obvious to one of ordinary skill in the art to combine Sexton with the well-known copy on write technology, thereby placing the artisan in possession of the invention.

Regarding the "means for" limitations, see Sexton / Bugnion rejection above.

Regarding **claims 2 and 11**, Sexton discloses functionality acting as a cache checker. Further, Johnson discloses: "... The page table entry list corresponding to that key number n in the lookaside buffer 218 is then searched to determine if requested data is in the page cache 212. If the data is in page cache...If the data is not in the page cache... loads that page into the page cache 212..." Johnson, col. 11, 11. 52-67.

"The intelligent reference objects 229 are preferably created and cached... IRO Managers create and cache...for each thread of execution in an application. In particular, each time an address translation is performed, an IRO is created and cached by the IRO Manager corresponding to the current thread of execution... quickly search the cache for an existing IRO that encapsulates the needed address translation. If such an IRO is not found in the IRO Manager cache, the IRO Manager interacts with the virtual address translator 210 to translate the SAS address. An IRO that encapsulates the new address translation is then created by the IRO Manager 127 and put into its cache under the current frame. Finally, the address of the IRO is returned to the JVM." Johnson, col. 12, 1. 55 - col. 13, 1. 6.

Regarding **claims 3 and 12**, Sexton teaches an obvious disclosure of a class locator to locate the source definition if the instantiated class definition is unavailable in the local cache. Further Johnson discloses:

"If the data is not in the page cache, the pager 214 retrieves the page of data in which the requested data is located [locate the source definition] from the backing store 404 and loads that page into the page cache 212..." Johnson, col. 11, ll. 52-67.

"quickly search the cache for an existing IRO that encapsulates the needed address translation. If such an IRO is not found in the IRO Manager cache, the IRO Manager interacts with the virtual address translator 210 to translate the SAS address. An IRO that encapsulates the new address translation is then created by the IRO Manager 127 and put into its cache under the current frame. Finally, the address of the IRO is returned to the JVM." Johnson, col. 12, l. 55 - col. 13, l. 6.

Regarding **claims 4 and 13**, Sexton teaches an obvious disclosure of a class resolver. Further to resolve a class, Johnson discloses:

The virtual address translator 210 uses a hash table 216, page table entry list, lookaside buffer 218 to determine if requested data is in the page cache 212. If the data is in page cache, a 32 bit address corresponding to the location of the data in the page cache is returned. If the data is not in the page cache, the pager 214 retrieves the page of data in which the requested data is located from the backing store 404 and loads that page into the page cache 212. The 32 bit native address of the data in the page cache 212 can then be returned." Johnson, col. 11, ll. 52-67. See also Johnson, col. 12, l. 5 - col. 13, l. 6.

Regarding **claims 5 and 14**, Sexton provides an obvious disclosure of a local and remote file system to maintain the source definition as a class file.

Further Johnson discloses:

"...two main types of data storage, transient data storage such as DRAM, and persistent storage such as hard disk drives, optical drives and such...

Most commodity computer systems today...use a system called two-level store (TLS). TLS systems use a file system for storing data on permanent storage and a virtual memory system for running application processes..." Johnson, col. 6, 11. 29-47.

"The SLS system maps all of the data storage mediums, generically referred to as backing store, into a single large address space. The backing store can include any type of local storage medium, such as magnetic and optical disk drives, and can also include the storage mediums of multiple computer systems connected by large networks. In the SLS model each bite of data contained within this large backing store area is addressed using its own unique, context independent virtual address. This makes the entire storage system function as a single "virtual memory" with a context independent addressing scheme." Johnson, col. 7 11. 48-60.

Regarding **claims 6 and 15**, Sexton provides an obvious disclosure of a process cloning mechanism.

Further Johnson discloses: