;login:

THE MAGAZINE OF USENIX & SAGE

August 2001 • Volume 26 • Number 5

Special Focus Issue:Clustering

Guest Editor: Joseph L. Kaiser

inside:

CONFERENCE REPORTS

1st Java™ VM

USENIX & SAGE

The Advanced Computing Systems Association &
The System Administrators Guild

First Java™ Virtual Machine Research and Technology Symposium (JVM '01)

Monterey, California April 23-24 2001

Keynote: Virtual Machines, Real Time

Greg Bollella, Sun Microsystems; David Hardin, aJile Systems

Summarized by V.N. Venkatakrishnan

The invited talk of the conference was the presentation on real-time virtual machines. Greg Bollella introduced the topic by presenting the scenario in embedded systems today. The presence of networking everywhere and the demand for building large, complex systems are two of the reasons for the inevitability of increasing software complexity. In this scenario, the two paradigms of system

l to r: Saul Wold, David Hardin, Greg Bollella

development – the one for business, personal, and Web computing and the other for device, scientific, and industrial computing – are moving toward a collision in this era of computing. Greg pointed out function migration from large devices to the hand held is the emerging trend. The real-time factor in hand-held devices is important because customers are used to system response in real time (e.g., a phone). Greg then presented a case example of JPL's mission data system and explained the role of real-time software in such an example.

Having motivated the listeners on the subject, Greg explained the technical aspects of a real-time system, using a car's electrical components as an example. He clarified the popular myth that a real-time system is not a fast system, but a system which includes time as an integral part of its computation.

There are several reasons why the Java language is an ideal choice for implementing such systems. As an advanced OO language, Java has a large set of libraries, a common set of APIs, an automatic memory management, and belongs to all the layers in software abstraction. A real-time JVM would thus support building various embedded system software, not just applications. But there are numerous barriers to achieving this: application-level unpredictability, hardware latencies, x86 context switch latencies, and inherent unpredictability due to various functions in the JVM such as scheduling and garbage collection.

In David Hardin's presentation, the approach taken by aJile is to implement JVM directly with simple, low-cost, low-power hardware. JVM bytecodes are native instructions and this supports real-time threads in hardware using Java thread primitives as instructions. This enables the entire system to be written in Java, with no C code or assembly required. Such an implementation has provided the fastest real-time Java performance.

Greg continued the discussion with the Java real-time specification, emphasizing such issues as scheduling, memory management, concurrency, and physical memory access. The implementation of JSR was scheduled for presentation to the expert group by April 30, and final release of the specification is in progress. The discussion culminated with a spectacular demo-presentation of piano playing robot hands controlled by two different real-time virtual machines.

After attending the talk, one was convinced that despite the various problems posed by hardware, OS, and JVM, real-time applications can be successfully built using Java.

Further information on this project can be obtained by contacting Greg at *greg.bollella@east.sun.com.*

Session: Code Generators

Summarized by V.N. Venkatakrishnan

The Java HotSpot Server Compiler

Michael Paleczny, Christopher Vick, and Cliff Click, Sun Microsystems

How can the performance of JVM improve through optimization of frequently executed application code? Michael Paleczny's talk addressed this research question through the presentation of the Java HotSpot Virtual Machine. The client version provides very fast compilation times and a small footprint with modest levels of optimization. The server version applies more aggressive optimizations to achieve improved asymptotic performance. These optimizations include class-hierarchy-aware inlining, fast-path/slow-path idioms, global value-numbering, optimistic constant propagation, optimal instruction selection, graph-coloring register allocation, and peephole optimization.

Michael described the runtime environment that both the compiler and generated code execute within, followed by the structure of the server compiler. Then he described some of the phases of compilation, discussing solutions for specific language and runtime issues. Finally, he outlined the directions for future work on the compiler which include range checks, loop unrolling, instruction scheduling, and a new inline policy.

Further information about this work can be obtained from *michael.paleczny@eng.sun.com*

Can a Shape Analysis Work at Runtime?

Jeff Bogda, Ambuj Singh, UC Santa Barbara

A shape analysis is a program analysis that can identify runtime objects that do not need to be placed in the global heap and do not require any locking. It has been shown through previous research that these two optimizations speed up some applications significantly. Since the shape analysis requires a complete call graph, it has not been implemented in the JVM.

After illustrating the purpose and some history of shape analysis, Jeff Bogda's talk went on with the description of his approach to build an incremental shape analysis to analyze an executing program. The analysis is done through an experimental framework to which the executing application is instrumented so that the analysis is performed at key points in the program execution. Jeff then described three approaches to performing shape analysis: immediate propagation, where the analysis is done before the method execution; delayed propagation, which delays the analysis untill an appropriate time; persistent propagation, which utilizes results from previous executions.

Jeff discussed the various trade-offs in these approaches. The experiments suggest a strategy which consults the results of the previous executions and delays the initial analysis untill the end of the first execution.

For more information on this work, the reader may visit *www.cs.ucsb.edu/~bogda* or contact Jeff at *bogda@cs.ucsb.edu*.

Saul Wold presenting Best Student Paper Award to Étienne Gagnon

SableVM: A Research Framework for the Efficient Execution of Java Bytecode

Étienne M. Gagnon, Laurie J. Hendren, McGill University

SableVM is an open-source virtual machine for Java intended as a research framework for efficient execution of Java bytecode. The framework is essentially composed of an extensible bytecode interpreter using state-of-the-art and innovative techniques. Written in the C programming language and assuming minimal system dependencies, the interpreter emphasizes high-level techniques to support efficient execution.

Sable VM introduces several innovative ideas: a bidirectional layout for object instances that groups reference fields sequentially; this allows efficient garbage collection. It also introduces a sparse interface virtual table layout that reduces the cost of interface method calls to that of normal virtual calls. Another important feature is the inclusion of a technique to improve thin locks by eliminating busy-wait in the presence of contention. In his talk, Gagnon presented SPEC benchmarks that demonstrated the efficiency of this research framework.

This paper won the best student paper award at the conference. Further details on this work can be obtained from the author (*egagnon@j-meg.com*) and at the Web site (*http://www.sablevm.org/*).

SESSION: JVM INTEGRITY

Summarized by V.N. Venkatakrishnan

Dynamic Type Checking in Jalapeño

Bowen Alpern, Anthony Cocchi, and David Grove, IBM T.J. Watson Research Center

Jalapeño is a JVM for servers. In any JVM, one must sometimes check whether a value of one type can be can be treated as a value of another type. The overhead for such dynamic type checking can be a significant factor in the running time of some Java programs. Bowen Alpern's talk presented a variety of techniques for performing these checks, each of these tailored to a particular restricted case that commonly arises in Java programs. By exploiting compile-time information to select the most applicable technique to implement each dynamic type check, the run-time overhead of dynamic type checking can be significantly reduced.

Bowen introduced the topic by going over the Java type system and the basic types. He then presented the main contributions of this research. This work suggests maintaining three data structures operationally close to every Java object. The most important of these is a display of superclass identifiers of the object's class. With this array, most dynamic type checks can be performed in four instructions. It also suggests that an equality test of the runtime type of an array and the declared type of the variable that contains it can be an important short-circuit check for object array stores. Together, these techniques result in significant performance improvements on some benchmarks.

This code that implements these techniques is not available in the public domain. The system is available for academic purposes; one may contact the author at *alpern@watson.ibm.com*. More information about the project is available at *http://www.research.ibm.com/jalapeno*.

Proof Linking: Distributed Verification of Java Classfiles in the Presence of Multiple Class Loaders

Philip W.L. Fong, Robert D. Cameron, Simon Fraser University

Computations involving bytecode verification can be expensive. To offload this burden within Java Virtual Machines (JVM), distributed verification systems may be created. This can be done using any one of a number of verification protocols, based on such techniques as proof-carrying code and signed verification by trusted authorities. Fong's research advocates the adoption of a previously proposed mobile code verification architecture, proof linking, as a standard infrastructure for performing distributed verification in the JVM. Proof linking supports various distributed verification protocols. Fong also presented an extension of this work to handle multiple class loaders.

Further details on this work can be obtained from the author at *pwfong@cs.sfu.ca*.

JVM Susceptibility to Memory Errors

Deqing Chen, University of Rochester; Alan Messer, Philippe Bernadat, and Guangrui Fu, HP Labs; Zoran Dimitrijevic, University of California, Santa Barbara; David Jeun Fung Lie, Stanford University; Durga Mannaru, Georgia Institute of Technology; Alma Riska, William and Mary College; and Dejan Milojicic, HP Labs

Deqing Chen presented a series of experiments to investigate memory error susceptibility using a JVM and four Java benchmark applications. Chen's work was woven around the fact that except for very high-end systems, little attention is being paid to high availability. This is particularly true for transient memory errors, which typically cause the entire system to fail. To bring systems closer to mainframe class availability, addressing memory errors at all levels of the system is important.

The experiments were done using the technique of fault injection. To increase detection of silent data corruption, JVM data structure checksums were examined. The results that were presented indicated that the JVM's heap area has a higher memory error susceptibility than its static data area and that up to 39% of all memory errors in the JVM and application could be detected. Such techniques will allow commodity systems to be made much more robust and less prone to transient errors.

For further information on this work, the author can be contacted by email at *lukechen@cs.rochester.edu*.

WORK-IN-PROGRESS REPORTS

Summarized by Chiasen (Charles) Chung

Implementing JNI in Java for Jalapeño

Ton Ngo, Steve Smith, IBM T.J. Watson Research Center

This talk addressed the advantages and implication of JNI implementation in Jalapeño, which is a JVM written in Java developed at the IBM T.J. Watson Research Center.

In order for the JNI functions to reuse the same internal reflection interface in Jalapeño, it is written in Java rather than in C as might be expected. This approach has two benefits: 1) changes in Jalapeño are transparent to the JNI implementation; 2) despite being a native interface, the JNI functions are portable to any platform where Jalapeño is installed.

When a native method is invoked in Jalapeño, a special static method is called to resolve the native method with the corresponding native procedure. JVM then generates the prologue and epilogue to establish the transition frames from Java to C code. The code entry from C to Java is through JNI functions defined in the specification. In Jalapeño, these are methods collected in a special Java class, and they are compiled dynamically, with special prologue and epilogue to handle the transition.

To resolve references in Jalapeño JNI, each Java object to be passed to a native code will be assigned an ID and then stored in a side stack. The native code accesses these objects based on their IDs. In a garbage collection cycle, Jalapeño JNI checks for live references in the native stack frames against the side stack.

The implementation of JNI on the PowerPC/AIX platform has been completed, while the Intel/Linux platform is still under development. The group is currently researching threading for long executions of native methods and issues concerning interaction between Java and native programs. More information can obtained at *http://www.research.ibm.com/jalapeno*.

JaRec: Record/Replay for Multi-threaded Java Programs

Mark Christiaens, Stijn Fonck, Dries Naudts, Michiel Ronsse, Koen De Bosschere, Ghent University

Debugging multi-threaded programs is difficult because thread races are hard to reenact, thus introducing non-determinism into the debugging. To solve this problem, Mark Christiaens suggested a two-phase "record/replay" technique.

JaRec is a program that records and replays the interaction sequence between threads in Java programs using two (enter and exit) monitors. Every thread has a Lamport clock which is incremented when the thread leaves or enters a monitor. During the record phase, a trace for the interaction between the threads based on this clock value is generated.

These Lamport clock values are recorded in the trace file as a timestamp. By forcing the order in which threads enter the monitors base on this timestamp, the thread execution and interaction sequence can be reproduced exactly. Synchronization is forced by waiting for a thread to report.

Both the record and replay phase in JaRec are implemented using the Java Virtual Machine Profiler Interface. The record phase is near completion and the group is currently implementing the replay phase of the system.

Kaffemik – A Distributed JVM Featuring a Single Address Space

Johan Andersson, Trinity College

Kaffemik is a scalable distributed JVM based on Kaffe VM. It is designed to run large-scale Java server applications by using clustered workstations. The goal of this project is to investigate scalability issues in a distributed JVM and to improve performance in large-scale Java applications.

Kaffemik is designed as a single JVM abstraction over the cluster by implementing a single address space architecture across all the nodes based on the global memory management protocol. On top of the common local thread operations, Kaffemik supports internode synchronization and remote-node thread creation.

Preliminary benchmark results show that Kaffemik starts local threads significantly faster than remote threads, but is much slower starting local threads compared to Kaffe. Remote threads are even more expensive due to the overhead induced by page-faults.

The current Kaffemik prototype shows that it is costly to implement distributed applications over high-speed clusters on single address space architectures. The next step in the project is to implement a two-level (global and local) memory allocator. A garbage collector for the global memory is also needed, but it is not addressed in this paper.

A Java Compiler for Many Memory Models

Sam Midkiff, IBM T.J. Watson Research Center

The Java memory model is heavily coupled into the programming language.In hopes of overcoming its various flaws, a new memory model has been proposed. Instead of fixing the memory model, this talk focused on defining the memory model as part of a property of the code being compiled.

Sam Midkiff proposes a Java compiler that accepts a ".class" file annotated with a memory-model specification. The compiler first represents the program using the Concurrent Static Single Assignment (CSSA) form. Escape analysis is applied to determine the order in which variables should be accessed according to the memory model. Next, the program represented in the CSSA graph is optimized. Finally, the compiler produces an executable that maps the program onto the underlying hardware consistency model.

This work explores the development that supports programmable memory models. Relative efficiency of different memory models running on a common hardware can be investigated. More information can be obtained from *http://www.research.ibm.com/people/m/midkiff/.*

State Capture and Resource Control for Java: The Design and Implementation of the Aroma Virtual Machine

Niranjan Suri, University of West Florida

Aroma VM is a research VM designed to address some of the limitations of current Java VMs. The capabilities for Aroma were motivated by the needs to mobilize agent systems and distributed systems.

Aroma provides two key capabilities: the ability to capture the execution state (of either the complete VM or individual threads) and the ability to control the resources used by Java programs running within the VM. The state capture capabilities are useful for load-balancing and survivable systems. The resource-control capabilities are useful for protecting against denial of service attacks, accounting for resource usage, and as a foundation for quality of service. Aroma currently provides both rate and quantity controls for CPU, disk, and network resources.

There is no Just-in-Time compiler for Aroma currently, but there are plans to integrate freely available JIT compilers (such as OpenJIT) in the future. More information on Aroma VM can be obtained from *http://nomads.coginst.uwf.edu/.*

OpenJIT2: The Design and Implementation of Application Framework for JIT Compilers

Fuyuhiko Maruyama, Satoshi Matsuoka, Hirotaka Ogawa, Naoya Maruyama, Tokyo Institute of Technology; Kouya Shimura, Fujitsu Laboratories

OpenJIT2 is a JIT compiler for Java written in Java that is based on "open compilers" construction technique. It not only serves as a JIT compiler but also as an application framework for JIT compilers. This framework allows multiple coexisting JITs to compile different parts of a program.

In the OpenJIT system, each instantiated compiler is a set of Java objects that compiles at least one method. The selection of methods to be compiled is determined through an interface that is based on method attributes. If the attribute does not specify a particular compiler (a set of compilet objects) to be used, the default baseline compiler will be selected.

Both baseline compiler and compilets are constructed using the OpenJIT2 framework and class library. Without the limitations of OpenJIT1's relatively simple internal structure, OpenJIT2 uses complex compiler modules to carry out analysis, program transformation, and optimization during compilation. The preliminary result shows that the baseline compiler will have reasonable compilation speed as an optimizing compiler compared with IBM's jitc and Jalapeño's optimizing compiler.

The first version of OpenJIT2 is expected to be completed by the second quarter of 2001. Once OpenJIT2 is complete, a more comprehensive runtime performance will be evaluated.

Session: Threading

Summarized by Okehee Goh

An Executable Formal Java Virtual Machine Thread Model

J. Strother Moore and George M. Porter, University of Texas at Austin

This presentation describes a research project in which formal methods are applied to Java Virtual Machine (JVM). "Formal methods" is the idea of using mathematics to model and prove things about computing systems. Certain aspects of the JVM are modeled, including classes, objects, dynamic method resolution, and threads. A benefit of modeling software in a mathematical notation is that theorems can be proved about the model. These proofs can be checked mechanically via a theorem prover. This paper discusses several such theorems about the JVM and byte-code programs for it. The theorems were proven with the ACL2 theorem prover.

ACL2 (A Computational Logic for Application Common Lisp) is a theorem prover for a functional programming language based on Common LISP. The JVM is modeled in ACL2 by defining a simulator for it. The state of the JVM consists of three components, including a collection of threads, a heap, and a class table. The semantics of each bytecode is represented as a function that transforms the state.

There are certain differences between this model and the JVM. For example, the model does not support bounded arithmetic or exceptions. Many such features were omitted to make it easier to explore alternative modeling and proof techniques. There is ample evidence from other ACL2 case studies that such features can be added without unduly complicating the analysis.

Complicated features of JVM bytecode programs, such as thread synchronization, can be analyzed using this mathematical model. Eventually, it should be possible to prove properties about the JVM itself, such as that the bytecode verifier is correct. Because the JVM is a very good abstraction of Java, models such as this will eventually permit mechanically checked correctness proofs about Java software.

More details about ACL2 are available at *http://www.cs.utexas.edu/users/moore/acl2* The case studies using ACL2 are at *http://www.cs.utexas.edu/users/moore/publications.*

TRaDe: A Topological Approach to On-the-Fly Race Detection in Java Programs

Mark Christiaens and Koen De Bosschere, ELIS, Ghent University, Belgium

The worst type of bug occurring in multi-threaded programs is a data race, which occurs when multiple threads execute while they modify a common variable in an unordered fashion. Normally it is hard to find a data race because they are non-deterministic and non-local.

TRaDe models the ordering of instructions performed by threads through the use of vector clocks. To detect data races, an access history for every object is constructed. When a new read or write operation occurs, it is compared to the previous operations to uncover data-race conditions. However, because the size of each vector clock is proportional to the number of threads, the memory and time consumption is very costly. One way to minimize this cost is to reduce the number of objects for which an access history must be maintained. Objects are distinguished into two types: local objects accessible to one thread and global objects accessible to several threads. Because “global objects” have the potential to be involved in a race, access to those objects must be checked, and the JVM instructions that can change the topology of the object interconnection graph must be observed.

Relative to the benchmark created by using an implemented TRaDe method in the Sun JVM1.2.1, TRaDe is 1.62 times faster than existing commercial products with comparable memory requirements.

The overhead of data-race detection is still large when compared to normal execution. The authors plan to reduce this gap, applying static analysis techniques such as “escape analysis.”

SESSION: JVM POTPOURRI

Summarized by Johan Andersson

The HotSpot Serviceability Agent: An Out-of-Process High-Level Debugger for a Java Virtual Machine

Kenneth Russell, Lars Bak, Sun Microsystems

This talk demonstrated a really useful Java debugging tool, built with the HotSpot Serviceability Agent (SA). This is a set of APIs for the Java programming language, developed to help developers recover to a high-level state from a HotSpot JVM or core file, to make it possible to examine high-level abstract data types. When examining a JVM with a traditional C/C++ debugger, all this high-level information is gone, since these debuggers only deal with raw bits.

The SA can attach a remote process or a core file, read remote-process memory, and symbols lookup in remote processes. In principle, the Solaris version of SA launches a native debugger called dbx to actually interface with a remote process. It then loads a core file or attaches to a running HotSpot JVM process. This allows transparent examination of either live processes or core files, which makes it suitable to debug the JVM itself or Java applications. In order to examine the high-level data types in Java, the APIs in the SA mirrors the C++ structures found in the HotSpot JVM.

Kenneth Russell demonstrated the features found in the SA’s APIs, which seemed to be very useful. It was very easy to traverse the heap and the stack, get histograms of allocated objects, and look up symbols.

In the future, the SA APIs, which are currently available for Solaris and Windows, will be ported to Linux. Russell said the APIs haven’t been included in the JDK yet, but they are working on making this technology available for end users. The SA sources are currently available to licensees in the HotSpot source bundles.

More Efficient Network Class Loading through Bundling

David Hovemeyer, William Pugh, University of Maryland

David Hovemeyer presented bundling, a technique for transferring files over a network. Files that tend to be needed in the same program execution and that are loaded close together are placed together into groups called bundles. Hovemeyer presented an algorithm to divide a collection of files into bundles based on profiles for file-loading behavior. The main motivation for bundling is to improve the performance of network class loading in Java, by transferring as few bytes as possible to make best use of available bandwidth. This is very useful in areas of wireless computing, where bandwidth is a scarce resource.

Before Hovemeyer introduced the bundling algorithm, he discussed the alternatives. The first alternative involves downloading individual files: no unneeded files are transferred, but for each file that is, the cost is high in terms of network latency. The other alternatives are to use monolithic archives such as JAR, thus risking transfer of unwanted files, or to use individual-class loading with on-the-fly compression, which can be time-consuming.

Hovemeyer and Pugh's bundling approach is a hybrid of the above alternatives, combining the advantages of each. The collection of files making up the application is divided into bundles, which are then compressed. The basic idea is to avoid files that are not used and to transfer files to match the order of request by the client. The problem is to divide the collection of files into bundles. To solve this, Hovemeyer talked about establishing class-loading profiles, which can be determined by using training sets of applications to record the order and time at which each class was loaded during execution. The bundling algorithm then uses this information to group the files into bundles, according to the average use in the class-loading profiles.

The experimental results indicated that bundling is a good compromise between on-demand loading and monolithic archives. The results also showed that bundling is no worse than the JAR format, when used on an application not included in the training set.

The bundling algorithm is described in detail in the paper. Links to related research done at the University of Maryland can be found at *http://www.cs.umd.edu/~pugh/java/*.

Deterministic Execution of Java's Primitive Bytecode Operations

Fridtjof Siebert, University of Karlsruhe; Andy Walter, Forschungszentrum Informatik (FZI)

Siebert started his talk by presenting the problems with real-time Java and gave a brief definition of Java real-time. To provide Java with real-time support, all operations must be carried out in constant time, or at least the upper bounds for the execution times of Java bytecode operations must be known. Essentially, the worst-case execution time for object allocations, dynamic calls, class initialization, type checking, and monitors must be determined.

The talk presented a JVM called Jamaica, which implements a deterministic JVM and a hard real-time garbage collector (GC). First, Siebert discussed the typical mark-and-sweep GC, followed by a presentation on how garbage collection and memory allocation are implemented in Jamaica to guarantee a hard upper bound for an allocation. To avoid memory fragmentation, compacting or moving garbage collection techniques are usually employed. However, Jamaica takes a new turn on this issue in order to avoid fragmentation altogether. The heap is divided into small, fixed-sized blocks (32 bytes). An object, depending on the size, is assembled as a linear list of possibly non-contiguous objects. With this model there is no need to defragment memory and move objects. When a block is allocated, the GC scans a certain number of blocks. This approach can guarantee that the system does not run out of memory, as well as guaranteeing an upper bound for the garbage collection work for the allocation of one block of memory.

The rest of the talk focused on how to obtain deterministic bytecode execution. Most bytecode operations can be implemented directly as a short sequence of machine instructions that executes in constant time. These operations include access to local variables and the Java stack, arithmetic instructions, comparisons, and branches. Siebert briefly discussed this but focused more on the bytecodes where deterministic implementation is not straightforward: for example, class initialization, type checking, and method invocation. The details of this can be found in the paper.

Finally, Jamaica's performance was compared to Sun's JDK implementation using SPECjvm98. The results suggested that performance comparable with Sun's non-deterministic implementations can be reached, by tuning the compiler, for example, and by direct generation of machine code instead of using C as the current intermediate representation.

For more information, contact the authors or visit *http://www.aicas.com*.

SESSION: GARBAGE COLLECTION

Summarized by Hughes Hilton

Mostly Accurate Stack Scanning

Katherine Barabash, Niv Buchbinder, Tamar Domani, Elliot K. Kolodner, Yoav Ossia, Shlomit S. Pinter, Ron Sivan, and Victor Umansky, IBM Haifa Research Laboratory; Janice Shepherd, IBM T.J. Watson Research Laboratory

A garbage collector must scan registers and the stacks in order to find objects which can be collected. Typically, there are three types of garbage collector: conservative, type-accurate, or conservative with respect to roots. All three have advantages and disadvantages.

A conservative collector is very simple to implement and has a low performance penalty. However, it must retain some garbage because it is not absolutely positive about what is garbage and what is not. This uncertainty also prohibits object relocation, which means that the stack cannot be compacted, degrading performance over time.

A type-accurate collector is much more complex to implement and is very expensive in terms of performance. However, all object-references are known with certainty and therefore all garbage is collected. Objects can also be moved so that memory may be compacted.

Type-accuracy also adds the factor that threads may be stopped only where type maps exist. Creating maps at every instruction can be very voluminous (although maps may be compressed somewhat). Certain algorithms, such as polling and patching, allow for better performance but are still comparatively expensive.

Lastly, a conservative approach with respect to roots scans the stack conservatively, but uses object type information to scan objects accurately. This is a compromise of the other two types of garbage collectors and works well. It allows object relocation and is used widely in Java Virtual Machines. However, compaction and some other GC algorithms are still difficult with this method of scanning.

The contribution of this paper is to propose another type of stack scanning: mostly accurate with respect to roots. In this method, the stack is only scanned accurately where it is easy to do so (most stack frames) and scanned conservatively otherwise. Therefore most objects can be relocated (allowing compaction), and the performance hit is minimal. Also, threads can be stopped anywhere. Further information about projects of IBM's Haifa research group is available at

http://www.haifa.il.ibm.com/projects/systems/Runtime_Subsystems.html.

Hot-Swapping Between a Mark&Sweep and a Mark&Compact Garbage Collector in a Generational Environment

Tony Printezis, University of Glasgow

Two algorithms for generational garbage collection that are often implemented in JVMs are Mark&Sweep and Mark&Compact. The main difference between the two is that Mark&Compact compacts the remaining objects to consolidate free space after garbage collection. These two algorithms are being considered when they are applied to the old generation of the system; they share the same algorithm for young garbage collections (that is, copying).

The Mark&Sweep algorithm is slightly faster than Mark&Compact, in most cases, because Mark&Sweep provides 200-300% faster collection for old objects, although old objects are usually not garbage collected as often as young objects. However, memory fragmentation can occur in a Mark&Sweep system, which can affect long-term performance.

The Mark&Compact algorithm is 10-20% faster in collecting the younger generation of objects because it provides faster allocation of objects to old space (which occurs during young garbage collection). Young garbage collection can occur up to 1000 times more often than old garbage collection, and it must also be taken into account that Mark&Compact defragments memory.

The performance difference between these two types of generational collection is fairly minimal and depends on the behavior of the application involved. However, what if a garbage collector could hot swap between the two types and get the best of both worlds? That was the question that Tony Printezis asked, and the subject of his paper.

The requirements set forth by Printezis for a hot-swapping garbage collector are fairly rigid. It must swap back and forth in constant time, incur a minimal performance penalty from swapping, be time flexible, and make minimal changes to the Mark&Sweep and Mark&Compact algorithms.

In order to develop the switching algorithm, Printezis had to use a fake byte array class to make a free chunk of memory look like garbage to the Mark&Compact collector, while still looking like a free chunk to the Mark&Sweep collector. He used a simple heuristic for when to swap. Mark&Sweep was used mostly for old garbage collections, but if linear allocation of objects from the young generation to the old generation failed a lot, one pass was made with Mark&Compact to defragment the memory.

In benchmarks, the hot-swapping algorithm fared well. It was the fastest of the three garbage collectors in two of the six benchmarks, and those benchmarks it did not win were very close. Also, the fact that the algorithm prevents memory fragmentation must be taken into account when considering the results. In the future, Printezis wants to develop more complex swapping heuristics, but preliminary results look very promising.

Parallel Garbage Collection for Shared Memory Multiprocessors

Christine H. Flood, David Detlefs, Sun Microsystems Laboratories; Nir Shavit, Tel-Aviv University; Xiolan Zhang, Harvard University

Since Java is being used increasingly with shared-memory multiprocessor systems, it makes sense that those systems should employ garbage collection algorithms that can take advantage of multiple processors to increase performance. This paper describes how Christine Flood and her fellow researchers parallelized two sequential, stop-the-world garbage collection algorithms: a two-space copying algorithm (semispaces) and a

Mark&Sweep algorithm with sliding compaction (Mark&Compact).

Load balancing is a big problem for parallel garbage collection. The key to load balancing is correctly and efficiently partitioning the task of tracing the object graph. This task does not lend itself to static partitioning, which is too expensive. Another solution might be over-partitioning by making more chunks than needed and having each processor get a chunk and come back for more. The problem with this algorithm is that the size of the problem is not necessarily known. The solution is a work-stealing algorithm. In work stealing, threads that have work copy some of it to auxiliary queues, where it is available to be stolen by other threads that do not have work to do.

In parallelizing the semispaces algorithm, Flood and her team used work-stealing queues to represent the set of objects to be scanned, rather than Cheney's copy and scan pointers (used traditionally). To avoid contention when many threads were allocating objects into space at the same time, they had each thread allocate relatively large regions called local allocation buffers (LABs).

Mark&Compact consists of four phases that must be parallelized: marking, forward-pointer installation (sweeping), reference redirection, and compaction. The researchers did the mark phase in parallel using work-stealing queues. They handled the forward-pointer installation by over-partitioning the heap. They implemented the reference redirection phase by treating the scanning of the young generation as a single task and reusing the previous partitioning done in the forward-pointer installation phase for the old generation. Finally, they parallelized the compaction phase by using larger-grained region partitioning.

In benchmarks it was found that with the teams' algorithms, the more processors working, the greater the advantage in garbage collection. With eight processors, there was as much as a 5.5x performance gain. The team concluded that parallel garbage collection must be used to avoid bottlenecks in large, multi-threaded applications. The contents of this paper and other works appear on Sun's site at: *http://www.sun.com/research/jtech/.*

SESSION: SMALL DEVICES

Summarized by Chiasen (Charles) Chung

Automatic Persistent Memory Management for the Spotless Java Virtual Machine on the Palm Connected Organizer

Daniel Schneider, Bernd Mathiske, Matthias Ernst, and Matthew Seidl, Sun Microsystems, Inc.

PalmOS does not support automatic multi-tasking capabilities. To achieve that, programmers have to implement low-level event callbacks using the OS database API to suspend and reload their applications. The talk proposes an alternative approach to allow transparent multi-tasking support for Java programs running on Spotless VM, a predecessor of KVM.

To restrict open memory access, the OS provided a simple database API. The API not only accesses a small subset of RAM for the application program but is also costly. Thus, the database API is bypassed by calling an undocumented system call to disable memory protection. The byte-code interpreter in the persistent Spotless VM still resides in the dynamic memory, but all the Java data (including the byte-codes and thread data) are stored in the static memory.

A program is first started by creating a new Store in the resource database tag of the type "appl." When the program is suspended, the VM automatically saves the current state of the application by closing the persistent Store in a controlled manner. To resume the suspended Spotless VM, it will be retrieved from the Store database. Next the VM will restore each heap record. Since the OS can move Store records in the heap segments, VM needs to update the pointers. After all the pointers have been updated, each module of the VM restores their state from the content in the Store header field before execution of the application continues. When a program finally terminates, the VM will remove the Store data from the database.

A program often needs external states or data that are not under the control of the program runtime system. Spotless VM supports persistence in these states through the implementation of an interface "External." External data have to synchronize with the internal data when the program is suspended or resumed. To achieve this, Spotless uses a protocol adopted from the Tycoon-2 system.

Disabling write protection creates a new dimension of safety issues for PalmOS. It is arguable whether a well-implemented VM will not cross its boundary, but hardware restriction is suggested. More information on Spotless Java Virtual Machine is available at *http://www.research.sun.com/spotless/.*

Energy Behavior of Java Applications from the Memory Perspective

N. Vijaykrishnan, M. Kandemir, S. Kim, S. Tomar, A. Sivasubramaniam, and M. J. Irwin, Pennsylvania State University

With mobile and wireless computing gaining popular ground, battery lifespan has become a growing concern. N. Vijaykrishnan's presentation addressed the energy behavior of the memory system during the execution of Java programs. It has been observed that memory systems consume a large fraction of the overall memory energy. Load/store are the instructions that access the most memory, consuming more than 50% of the total energy in both interpreted and JIT-compiled programs. As data themselves, byte-codes need to be fetched from memory, and so interpreters are more memory-intensive than JIT-compiled code.

ExactVM (EVM) is the JVM from Sun Labs Virtual Machine for Research used in experiments. The experiment is based on the seven applications from the SPEC JVM98 benchmark suite, with emphasis on "javac" and "db." Beside the actual execution of Java applications, EVM utilizes memory heavily in three areas: class-loading, dynamic method compilation, and garbage collection. Other than the frequency of memory accesses, energy consumption is also dependent on frequency of cache misses since off-chip memory accesses are more expensive than on-chip accesses; thus data locality is an issue. It was found in their experiment that energy consumption is inversely proportional to the cache size

To improve energy consumption, it was recommended that class files should be reused across different applications and that heap allocators and garbage collectors be energy aware. Although energy consumed by dynamic compilation in JIT mode is quite significant, a well-designed compiler will produce native code that actually reduces energy consumption. More information on the talk can be found at *http://www.cse.psu.edu/~mdl/*.

On the Software Virtual Machine for the Real Hardware Stack Machine

Takashi Aoki, Takeshi Eto, Fujitsu Laboratories Ltd.

This talk focused on using picoJava-II as a software virtual machine running on a real hardware stack machine. picoJava-II is a Java chip developed at Fujitsu. Unlike traditional JVM, which uses a straightforward memory area as a Java stack, picoJava-II takes advantage of the hardware cache for the stack to improve the bytecode execution performance. Sun's PersonalJava 3.02 is ported onto picoJava-II, which is running on REALOS.

picoJava-II has a different engine architecture from traditional JVMs. Numerous modifications have to be made in order to port PersonalJava onto the direct bytecode execution engine of picoJava-II. picoJava-II has a 64-word stack cache to improve bytecode execution performance. Since there is no coherency between the stack and the data cache, the former has to be flushed frequently before accessing the stack frame. Another issue is that the stack grows in the opposite direction (downward), requiring additional computation to resolve the start of the next frame. JavaCodeCompact (JCC) is a tool available on PersonalJava to improve class-loading performance and reduce code size. The internal data structure of JCC has to be modified before the hardware can accept it.

The testing indicates that the Java microprocessor is significantly better than the conventional C interpreter. It is also competitive with JIT-compiled code. However, there are a number of open problems encountered in the research. First, the lack of coherency between stack and data caches complicates software design. Next, the JNI implementation can be more efficient if the C compiler of picoJava-II follows the calling convention of the Java method. Lastly, the presence of aggregate stacks for solving the stack cache incoherency problem complicates system programming.

Saul Wold & Étienne Gagnon