The South African Institute of Computer Science and Information Technology

The 1997 National Research and Development Conference

Riverside Sun
Vanderbijlpark
13 & 14 November

Hosted by

Potchefstroomse Universiteit
vir Christelike Hoër Onderwys

The Department of Computer Science and Information Systems
Potchefstroom University for Christian Higher Education
Vaal Triangle Campus

PROCEEDINGS

Edited by L.M. Venter & R.R. Lombard
The South African Institute of Computer Science
and
Information Technology

Proceedings
of the
The 1997 National
Research and
Development
Conference
Towards 2000

Riverside Sun
Vanderbijlpark
13 & 14 November

Edited by
L.M. Venter
R.R. Lombard
©1997 Copyrights reside with the original authors who may be contacted directly

ISBN 1-86822-300-0

Printed and Binded by Xerox Printers, Potchefstroom

The views expressed in this book are those of the individual authors
Foreword

This book contains a collection of papers presented at a Research and Development conference of the South African Institute of Computer Scientists and Information Technologists (SAICSIT). The conference was held on 13 & 14 November 1997 at the Riverside Sun, Vanderbijlpark. Most of the organization for the conference was done by the Department of Computer Science and Information Technology of the Vaal Triangle Campus, Potchefstroom University for Christian Higher Education.

The programming committee accepted a wide selection of papers for the conference. The papers range from detailed technical research work to reports of work in progress. The papers originate mainly from Academia, but also describe work done in and for Industry. It is hoped that the papers give a true reflection of the current research scene in Computer Science and Information Technology in South Africa. Since one of the aims of the conference is Research development, the papers were not subjected to a refereeing process.

A number of people spent numerous hours helping with the organization of this conference. In this regard, we wish to thank the members of the Organizing committee, and the Programming committee who had very little time to screen the abstracts and compile the program. A special thanks goes to the secretary of the department, Mrs Helei Jooste, whose very able work was interrupted by the birth of her first child.
Organizing Committee

Conference General Chairs
Prof. J.M. Hattingh (PU for CHE)

Organizing Chair
Prof. Lucas Venter (PU for CHE)

Organizing Committee
Mrs. S. Gilliland
Mr. J.P. Jooste
Mr. R.R. Lombard
Mrs. M. Huisman

Secretariat
Mrs. H. Jooste

Program Chair
Prof A de Waal (PU for CHE)

Program Committee
Prof D. Kourie (UP)
Prof. C. Bornman (UNISA)
Prof. L.M. Venter (PU for CHE)
# Table of Contents

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreword</td>
<td>i</td>
</tr>
<tr>
<td>Organizing Committee</td>
<td>ii</td>
</tr>
<tr>
<td>List of Contributors</td>
<td>vii</td>
</tr>
<tr>
<td><em>Software Objects Change: Problems and Solution</em></td>
<td>1</td>
</tr>
<tr>
<td>S.A. Ajila</td>
<td></td>
</tr>
<tr>
<td><em>Liming-like Curve Constructions</em></td>
<td>26</td>
</tr>
<tr>
<td>M.L. Baart and R. McLeod</td>
<td></td>
</tr>
<tr>
<td><em>A Model for Evaluating Information Security</em></td>
<td>27</td>
</tr>
<tr>
<td>L. Barnard and R. von Solms</td>
<td></td>
</tr>
<tr>
<td><em>Integrating Spatial Data Management and Object Store Technology</em></td>
<td>31</td>
</tr>
<tr>
<td>S. Berman, S. Buffler and E. Voges</td>
<td></td>
</tr>
<tr>
<td><em>Metamodelling in Automated Software Engineering</em></td>
<td>32</td>
</tr>
<tr>
<td>S. Berman and R. Figueira</td>
<td></td>
</tr>
<tr>
<td><em>Using Multimedia Technology for Social Upliftment in Deprived</em></td>
<td>33</td>
</tr>
<tr>
<td>Communities of Southern Africa</td>
<td></td>
</tr>
<tr>
<td>L. Bester and E. de Preez</td>
<td></td>
</tr>
<tr>
<td><em>Extending the Client-Server Model for Web-based Execution of</em></td>
<td>36</td>
</tr>
<tr>
<td>Applications*</td>
<td></td>
</tr>
<tr>
<td>L. Botha, J.M. Bishop and N.B. Serbedzija</td>
<td></td>
</tr>
<tr>
<td><em>Access Control Needs in an Electronic Workflow Environment</em></td>
<td>45</td>
</tr>
<tr>
<td>R.A. Botha</td>
<td></td>
</tr>
<tr>
<td><em>The Use of the Internet in an Academic Environment to Commercially</em></td>
<td>51</td>
</tr>
<tr>
<td>Supply and Support Software Products</td>
<td></td>
</tr>
<tr>
<td>B. Braude and A.J. Walker</td>
<td></td>
</tr>
<tr>
<td><em>Explanation Facilities in Expert Systems Using Hypertext Technology</em></td>
<td>63</td>
</tr>
<tr>
<td>T. Breetzke and T. Thomas</td>
<td></td>
</tr>
<tr>
<td><em>Theoretical Computer Science: What is it all about, and is it of any relevance to us?</em></td>
<td>75</td>
</tr>
<tr>
<td>C. Brink</td>
<td></td>
</tr>
<tr>
<td><em>Representing Quadrics on a Computer</em></td>
<td>76</td>
</tr>
<tr>
<td>M.A. Coetzee and M.L. Baart</td>
<td></td>
</tr>
</tbody>
</table>
The Generation of Pre-Interpretations for Detecting Unsolvable Planning Problems
D.A. de Waal, M. Denecker, M. Bruynooghe and M. Thielscher

The Emerging Role of the Chief Information Officer in South Africa
B. Dekenah

A Java-Implemented Remote Respiratory Disease Diagnosis System on a High Bandwidth Network
A. Foster

Early Results of a Comparative Evaluation of ISO 9001 and ISO/IEC 15504 Assessment Methods Applied to a Software Project
C. Gee and A.J. Walker

A Neural Network Model of a Fluidised Bed
M. Hajek

The Effects of Virtual Banking on the South African Banking Industry
M.L. Hart and M. Dunley-Owen

Linear Response Surface Analysis and Some Applications
J.M. Hattingh

Model Checking Software with Symbolic Trajectory Evaluation
A. Hazelhurst

A Risk Model to Allocate Resources to Different Computerized Systems
H.A. Kruger and J.M. Hattingh

Returns on the Stock Exchange
J.W. Kruger

Cardinality Constrained 0-1 Knapsack Problems
M.F. Kruger, J.M. Hattingh and T. Steyn

An Investigation in Software Process Improvement in the Software Development of a large Electricity Utility
M. Lang and A.J. Walker

Design and Implementation of a C++ Package for Two-Dimensional Numerical Integration
D.P. Laurie, L Pluym and Ronald Cools

Algebraic Factorization of Integers Using BDE's
H. Messerschmidt and J. Robertson
Global Optimization of Routes after the Process of Recovery
M. Mphahlele and J. Roos

Using a Lattice to Enhance Adaptation Guided Retrieval in Example Based Machine Translation
G.D. Oosthuizen and S.L. Serutla

Information Systems Development and Multi Criteria Decision Making / Systems Thinking
D. Petkov, O. Petkova

The Development of a Tutoring System to Assist Students to Develop Answering Techniques
N. Pillay

Combining Rule-Based Artificial Intelligence with Geographic Information Systems to Plan the Physical Layer of Wireless Networks in Greenfield Areas
K. Prag, P. Premjeeth and K. Sandrasegaran

A Distributed Approach to the Scheduling Problem
V. Ram and P. Warren

More readings than I thought: Quantifier Interaction in Analysing the Temporal Structure of Repeated Eventualities
S. Rock

Ray Guarding Configuration of Adjacent Rectangles
I. Sanders, D. Lubinsky and M. Sears

Developing Soft Skills in Computer Students
C. Schröder, T. Thomas

Information Security Awareness, a Must for Every Organization
M. Thomson and R. von Solms

Pla Va: A Lightweight Persistent Java Virtual Machine
S. Tjasink and S. Berman

Beliefs on Resource-Bounded Agent
E. Viljoen

Object-Orientated Business Modelling and Re-engineering
M. Watzenboeck
On Indexing in Case Based Reasoning Applied to Pre-Transportation Decision Making for Hazardous Waste Handling
K.L. Wortmann, D. Petkov and E. Senior

Author Index
List of Contributors

S.A. Ajila
Department of Mathematics and Computer Science
National University of Lesotho
Roma, 180
Lesotho

L. Baart
Department of Mathematics
Vaal Triangle Campus of the PU for CHE
PO Box 1174
Vanderbijlpark, 1900

L. Barnard
Faculty of Computer Studies
Port Elizabeth Technikon
Private Bag X6011
Port Elizabeth, 6000

S. Berman
University of Cape Town
Rondebosch, 7701

L. Bester
Faculty of Computer Studies
Port Elizabeth Technikon
Private Bag X6011
Port Elizabeth, 6000

J.M. Bishop
Computer Science Department
University of Pretoria
Pretoria, 0002

L. Botha
Computer Science Department
University of Pretoria
Pretoria, 0002

R.A. Botha
Faculty of Computer Studies
Port Elizabeth Technikon
Private Bag X6011
Port Elizabeth, 6000

B. Braude
Software Engineering Applications Laboratory,
Electrical Engineering
University of the Witwatersrand
Private Bag 3
Wits, 2050

T. Breetzke
Faculty of Computer Studies
Port Elizabeth Technikon
Private Bag X6011
Port Elizabeth, 6000

C. Brink
University of Cape Town
Rondebosch, 7700

M. Bruynooghe
Departement Computerwetenschappen
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee
Belgium

S. Buffler
University of Cape Town
Rondebosch, 7701

M.A. Coetzee
Department of Mathematics
PU for CHE
Private Bag X6001
Potchefstroom, 2520

R. Cools
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee
Belgium

E. de Preez
Faculty of Computer Studies
Port Elizabeth Technikon
Private Bag X6011
Port Elizabeth, 6000

D.A. De Waal
Department of Computer Science and Information Systems
PU for CHE
Private Bag X6001
Potchefstroom, 2531

B. Dekekenah
The Board of Executors

M. Denecker
Departement Computerwetenschappen
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee
Belgium

M. Dunley-Owen
Department of Information Systems
University of Cape Town
Rondebosch, 7700

R. Fiqueira
University of Cape Town
Rondebosch, 7701

A. Foster
Department of Computer Science
University of Cape Town
Rondebosch, 7701

C. Gee
Software Engineering Applications Laboratory,
Electrical Engineering
University of the Witwatersrand
Private Bag 3
Wits 2050
M. Hajek
Department of Computer Science
University of Durban Westville
Private Bag X54001
Durban, 4000

M.L. Hart
Department of Information Systems
University of Cape Town
Rondebosch, 7700

J.M. Hattingh
Department of Computer Science and Information Systems
PU for CHE
Private Bag X6001
Potchefstroom, 2520

S. Hazelhurst
Department of Computer Science
University of the Witwatersrand
Private Bag 3
Wits 2050

H.A. Kruger
Department of Computer Science and Information Systems
PU for CHE
Private Bag X9001
Potchefstroom, 2520

J.W. Kruger
University of the Witwatersrand
Private Bag 3
Wits, 2050

M.F. Kruger
PU for CHE
Private Bag X6001
Potchefstroom, 2520

M.T. Lang
Eskom Information Technology Department

D. Laurie
Department of Mathematics
Vaal Triangle Campus of the PU for CHE
PO Box 1174
Vanderbijlpark, 1900

D. Lubinsky
Department of Computer Science
University of the Witwatersrand
Private Bag 3
Wits, 2050

R. McLeod
Saltire Software Inc.
Tigard
Oregon
U.S.A.

H.J. Messerschmidt
Department of Computer Science and Informatics
University of the Orange Free State
PO Box 339
Bloemfontein, 9300

M. Mphahlele
Department of Computer Science
University of the North
Private Bag X1106
Sovenga, 0727

G.D. Oosthuizen
Department of Computer Science
University of Pretoria
Pretoria, 0002

J. Owen
University of Cape Town
Rondebosch, 7701

D. Petkov
Department of Computer Science
University of Natal
Private Bag X01
Scotsville, 3209

O. Petkova
Technikon Natal
PO Box 101112
Scotsville, 3209

N. Pillay
Department of Financial Studies
Technikon Natal, Pietermaritzburg
PO Box 101112
Scotsville, 3209

L. Pluym
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee
Belgium

K. Prag
Department of electrical Engineering
University of Durban-Westville
Private Bag X54001
Durban, 4000

P. Premjeeth
Department of electrical Engineering
University of Durban-Westville
Private Bag X54001
Durban, 4000

V. Ram
Department of Computer Science
University of Natal
Private Bag X01
Scotsville, 3209

J. Robertson
Department of Computer Science and Informatics
University of the Orange Free State
PO Box 339
Bloemfontein, 9300

S. Rock
Department of Artificial Intelligence
Edinburgh University
United Kingdom

J. Roos
Department of Computer Science
University of Pretoria
Pretoria, 0002

I. Sanders
Department of Computer Science
University of the Witwatersrand
Private Bag 3
Wits, 2050
K. Sandrasegaran  
Department of electrical Engineering  
University of Durban-Westville  
Private Bag X54001  
Durban, 4000

C. Schoder  
Faculty of Computer Studies  
Port Elizabeth Technikon  
Private Bag X6011  
Port Elizabeth, 6000

M. Sears  
Department of Mathematics  
University of the Witwatersrand  
Private Bag 3  
Wits, 2050

E. Senior  
International Center for Waste Technology  
University of Natal, Pietermaritzburg  
Private Bag X01  
Scotsville, 3209

N.B. Serbedzija  
GMD FIRST  
Rudower Chaussee 5  
D-12489 Berlin  
Germany

S.L. Serutla  
Department of Computer Science  
The University of Pretoria  
Pretoria, 0002

T. Steyn  
PU for CHE  
Private Bag X8001  
Potchefstroom, 2520

M. Thielscher  
Fachgebiet Intellektik, Fachgebiet Informatik  
Technische Hochschule Darmstadt  
Alexanderstrasse 10  
D-64283 Darmstadt  
Germany

T. Thomas  
Faculty of Computer Studies  
Port Elizabeth Technikon  
Private Bag X6011  
Port Elizabeth, 6000

M. Thomasson  
Faculty of Computer Studies  
Port Elizabeth Technikon  
Private Bag X6011  
Port Elizabeth, 6000

S. Tjasink  
University of Cape Town  
Rondebosch, 7700

E. Viljoen  
Department of Computer Science and  
Information Systems  
University of South Africa  
PO Box 392  
Pretoria, 0001

E. Voges  
University of Cape Town  
Rondebosch, 7701
PLaVa: A Lightweight Persistent Java Virtual Machine

S. Tjasink and Dr S. Berman
University of Cape Town

October 1, 1997

1 Introduction

This paper describes the implementation and tuning of an orthogonally persistent Java Virtual Machine (JVM) for small computers. It is based on a JVM that was targeted at a digital satellite television decoder with 1 megabyte each of ROM and RAM.

Orthogonally persistent languages allow all program values to persist on disk, regardless of their type; and enable program code to have the same form regardless of the longevity of the data on which it operates. They improve productivity because there is no need for programmers to learn a separate database language or to manage data transfer and translation when using persistent objects.

Java has received a lot of attention since its release by Sun Microsystems in 1995. Many features of Java make it well suited to orthogonal persistence: source code is compiled to a platform-independent bytecode; there is implicit heap management, strong typing and useful programming constructs like threads, exceptions and classes. Java is also appropriate for embedded applications because of its safety and security features.

Several researchers have now begun working on persistent Java implementations. This paper demonstrates the feasibility of accessing a persistent Java store from very small computers which obtain applets over the net.

The paper starts with an overview of persistent systems, followed by an explanation of how the Java language itself has been affected by the addition of persistence. The next section discusses how the support for persistence has been added to the virtual machine. That is followed by a section discussing related work and we conclude with long-term goals.

2 Overview of persistent systems

Orthogonally persistent languages allow values to persist by automatically saving them to a persistent store and enable the program code to have the same form regardless of the longevity of the data on which it operates. [AM95] gives an outline of orthogonally persistent systems and provides a tutorial for those commencing research or study in the field. It summarises the benefits of persistence over conventional database applications and covers the basics of persistence, existing achievements and current research issues.
The principles of persistence are given here. They must all be adhered to when designing an orthogonally persistent system. This section is just a quick introduction to the concepts involved before our implementation is discussed in detail.

The Principle of Persistence Independence The form of a program is the same whether it is manipulating persistent or transient data.

The Principle of Data Type Orthogonality Data of all types should be allowed to persist. This complies with the language design principle of type completeness.

The Principle of Persistence Identification There should be a well defined system for identifying which data is persistent and it should be orthogonal to the type system. This principle is often implemented using identification by reachability. In this case, at least one root of reachability is needed, from which other reachable objects can be found.

When an attempt is made to use an object that is referenced from another object, it must be loaded into memory from the persistent store before it can be used. There are usually two forms of addresses that are used. These are:

Persistent Identifier (PID) that is used to refer to persistent objects in the store. It is usually the index of the object in the persistent store. These identifiers cannot be used directly by an executing program and must be converted to local addresses (see below).

Local Address (LA) which is a pointer into physical memory. These are used to refer to objects that have been loaded into memory for use by the program.

If an LA is encountered then execution should not be interrupted. If a PID is encountered then steps have to be taken to load the required object (called an object fault, similar to a page fault in operating system parlance). First, we check to see if the referenced object has already been loaded (referred to as a residency check). If not, we need to load it. If it has already been loaded because it was used by another part of the program then of course we do not have to load it (this is called a false object fault).

When an object is loaded, the mapping between its PID and its LA is added to a table to facilitate future residency checks. The PID that was referenced is overwritten with the LA of the loaded version of the object (called swizzling). This is done so that next time the reference is encountered, it will be in LA form and execution can proceed without being interrupted.

Storage of data into the persistent store also has to be automated. When an application requests a checkpoint, all loaded persistent objects that have been changed need to be written back to the store. Also, any new objects that have been created that are now reachable from a persistent root need to be promoted to persistence. The same applies (recursively) to all new objects that are reachable via these.

3 Language design issues

3.1 The Java language and persistence

Java is a modern, object-oriented programming language [AG96, GJS96]. It is loosely based on C++ [Str91] but does not include the commonly confusing features and constructs that have made...
• PStore openStore(String storeName): This method is used to open an existing persistent store of the given name. If the store is not found then a StoreNotFoundException is thrown. An instance of the PStore class representing the store is returned.

When a store has been created or opened, objects can be added to the store to be used as persistent roots. Roots are identified by a name, which can be used to retrieve them later. These root objects can also be removed as roots, which means that the object and any objects reachable from it will no longer persist unless they are reachable via another root. Methods that perform these functions are provided; they can be called on any instance of the PStore class that has been obtained via CreateStore or OpenStore.

• void addRoot(String name, Object ob): Adds the specified object as a new persistent root. It is given the specified name so that it can be retrieved at a later stage.

• Object getRoot(String name): Returns the specified root if a root of that name exists. It will need to be typecast to its actual type.

• void removeRoot(String Name): Removes the named root from the store's list of roots.

In order to convert a standard Java program to a persistent version of the program, only a few changes need to be made:

1. Open a persistent store (or create a new store if one does not already exist).
2. Retrieve the root object for the program that you wish to run.
3. Insert calls to stabilise the store (discussed in the following section) at relevant points in the program in order to update the store if needed. This is not essential as the store is stabilised implicitly when the program execution ends.

A short example showing how the store is opened and the objects accessed follows. In practice, the calls for opening the store and accessing roots throw exceptions if errors occur during execution and these exceptions should be caught to deal with the problems.

```java
/* Open the store called "cddb.store" */
PStore p = PStore.openStore("cddb.store");

/* Ask for the root called "CDB1"... must be cast to its correct type */
CDBthing mydb = (CDBthing) p.getRoot("CDB1");

/* Run the code in the object you've accessed. All other persistent objects that are referenced in the code will automatically be faulted in during execution. */
mydb.run();

/* Stabilise to write all objects back to the store */
p.stabiliseAll();
```
The program that is started in the third line of code above is a standard Java program that maintains a small database of information about compact discs including title, artist and the names of songs. All that the program needs to do is to construct a collection (e.g. a tree, array or linked list) containing the database and have a reference to this collection from the object mydb that is loaded above. Because the tree is accessible from mydb and mydb is a registered root of the store, all data in the collection will automatically be saved during a stabilise.

### 3.3 Transactions

At present only a global stabilise method, stabiliseAll, is provided in the PStore class. This can be used to write all loaded persistent objects back to the store (and to promote all the newly reachable objects).

In the PJama system [AJDS96] developed by the Glasgow group, an internal as well as an external transaction API is specified. The external API is provided by the TransactionShell class and its specialisations. These should satisfy most application programmers, but programmers who wish to define new transactional models by defining their own specialisations of TransactionShell can use the internal API to do so.

### 4 Persistence mechanism design

This section presents in some detail the design and implementation of the PLaVa persistent virtual machine. First, a very brief overview of relevant aspects of the JVM architecture is given. This is followed by a discussion of how swizzling has been added to the machine and then the actual object faulting. After this, update tracking and writing back to the store are dealt with. The way in which Java heap garbage collection interacts with the persistence mechanisms is then explained and finally we discuss the matter of native methods in persistent code.

#### 4.1 The Java Virtual Machine

As mentioned in the section on language issues, changes were made to a standard JVM in order to support persistence. The JVM is a stack-based machine that executes the Java bytecode. This means that it has an operand stack and most of the bytecode instructions take their operands from this stack and push their results onto it. Each method also has a number of local variables that it uses.

#### 4.2 When to swizzle

Swizzling is done whenever a reference to a persistent object is loaded onto the operand stack from an object. If the field that is being loaded onto the stack is a reference and is in PID form then its corresponding LA is loaded onto the stack. This will involve faulting the object in if it is not already resident. In both cases the PID reference that is being used is overwritten at its source with the LA so that all successive uses will be able to use the LA directly.

This scheme is essentially a lazy, direct swizzling scheme using object faulting. It turns out to be similar to swizzling on discovery as described in [WD92]. In general compiled languages discussed in [WD92], actually discovering the original location of a reference so that it can be swizzled can be
problematic. In Java this is simplified by the fact that discoveries can be isolated to the execution of two bytecode instructions.

- **getfield**: This instruction is used for loading the contents of fields of all types from class instances onto the operand stack. In the case where the field is a reference, the field is swizzled. Reference fields are identified by the field's signature. Its signature starts with 'L' in the case of a reference to a class instance or with a '[' in the case of a reference to an array.

- **aaload**: This instruction is used for loading a reference onto the operand stack from an array of references. The corresponding array element is swizzled in this case.

Java bytecode instructions that use references always take these references from the operand stack. Thus the system of making sure that only LAs appear on the operand stack will ensure that Java bytecodes only ever encounter LAs. Because all parameters that are passed to methods are taken from the operand stack, all references that are passed (including the target object) will be LAs. This automatically satisfies target object residency and formal parameter residency as discussed in [MH94, Hos96]. Procedure result residency will also automatically be enforced as the return instruction that returns references from methods takes the return value from the current method's operand stack.

### 4.3 Mechanics of object faulting

When an object is faulted in, its PID to LA mapping is added to the resident persistent object table (RPOT) so that future residency checks will find it. The table is a hash table addressable by PID. It does not need to be addressable by LA as well, because reverse mappings are not needed in PLaVa. An object's PID is stored in a field in the object header and when it is needed it can be read from there without having to query the RPOT.

A macro is used to determine whether or not a reference is a PID, currently all PIDs are negative integers. This is used to determine whether or not the reference needs to be swizzled. When an object is faulted in, the object's PID is stored in a field in that object's header as described above. The space used by the PID is less than the space that would be required by an index to make the RPOT addressable by LA as well as by PID.

Swizzling can also be turned off. In this case, PIDs are never overwritten with LAs and a residency check needs to be done every time one is used. This will slow normal program execution because of the extra residency checks that need to be done, but references will not have to be deswizzled when objects are written back to the store. In the PLaVa system deswizzling is a quick operation so this is not much of a performance gain.

### 4.4 Recording updates

All objects that are changed need to be marked as such so that the store can be updated when required by a stabilise.

The execution of bytecodes that write to fields of objects or elements of arrays was changed so that operands could reflect their altered status. If an object is changed then its PERSIST.Updated flag is set. If a reference field in an object or an element in an array of references is changed then its PERSIST.RefsUpdated flag is set. These flags are stored in the object header with a number of other flags.
Figure 2: Object faulting stage 1. A reference field in PID form is dereferenced. The PIDLAM (called RPOT in PLaVa) is queried to see if the object corresponding to that PID has already been loaded. In this case it has not.

Figure 3: Object faulting stage 2. The object corresponding to the PID is loaded from the store into the heap. Its PID and its LA (its address in the heap) are inserted into the PIDLAM.

Figure 4: Object faulting stage 3. The reference field that was used to reference the object is overwritten with the loaded object's LA (swizzling).
The two separate flags are used to make the process of updating changed objects more efficient. If an object is marked as having had one of its reference fields updated then when it is written back to the store, the objects that it references must be checked to see if they have also been updated. In the case where none of an object's reference fields have been updated its fields do not have to be checked and it can simply be written to the store. Instructions affected are:

- **putfield**: This instruction needs to flag the changed class instance as updated. It is marked with the `PERSIST.RefsUpdated` flag if it is a reference field that is updated (determined by looking at the field's signature). In the implementation of `putfield`, the signature had to be checked to see if fields were 32 or 64 bits side, so this is not an extra expense.

- **zastore**: (includes `iastore`, `lastore`, `fastore`, `dastore`, `bastore`, `castore` and `sastore`) These instructions need to flag the changed array as updated (with the `PERSIST.RefsUpdated` flag in the case of `aastore`).

### 4.5 Object updating

During a stabilise, all objects that are marked as updated need to be written back to the store. Only those objects that have been marked as having a reference field updated need have their fields scanned for references. When a reference field is found, the object that it references is updated as well (this will include promotion to persistence if it is currently transient).

If the reference field is an LA then it must be overwritten with its PID so that it can be written back to the store correctly. This unswizzling just requires reading the PID from the object header since every persistent object stores its own PID in its object header. When all of an object's reference fields (or elements in the case of an array) have been stabilised then it is written to the store.

Each object that has an entry in the RPOT is considered and the following criteria are used for determining the objects that need to be written to the store:

- An object is not written back if it is already persistent and is not flagged as updated.

- An object is updated in the store if it is persistent and flagged as updated. If it is flagged as having had its reference fields updated then each of the reference fields (or elements if it's an array) is checked as well, with these update criteria being applied to the referenced objects. The reference fields have to be checked because new objects might now be reachable and these new objects will have to be promoted to persistence.

- An object is promoted to persistence if it is not already persistent, regardless of whether or not it is marked as updated. This ensures that all new objects that become reachable from persistent objects also become persistent.

Currently, when an object needs to have its reference fields checked, all fields need to be examined in order to see which of them are reference fields. This requires examining each field's signature. Checking reference fields could be made more efficient by clustering them to either the beginning or the end of the object's field storage memory. A bit map could also be maintained to track updates to fields or just to provide quick differentiation between reference fields and other fields. The bit map update-tracking method will require extra processing during actual execution although it will require fewer checks during a stabilise.
Null PID references are represented exactly the same as local null values (i.e. 0). No distinction needs to be made between the two as null values are invalid for dereferencing in both cases.

When objects are updated in the store, their swizzled fields are unswizzled back to PID form in which they are required to be when put into the store because the object is copied directly from the heap into the store. This means that these fields will have to be reswizzled when they are used again in the course of program execution. These are likely to cause false object faults on reswizzling because the objects will still be resident (unless they become victims of garbage collection before they are referenced again). The previous values of all reference fields could be stored in order to restore them to their LA values after the write-back to the store, but this would involve extra memory use during the stabilise operation. This has been added as an option that can be enabled or disabled for testing purposes.

4.6 Heap garbage collection

As has been mentioned previously, memory allocation and de-allocation in Java is implicit and the JVM uses a garbage collector to reclaim the memory that is occupied by objects that are no longer useful to the program that is running.

The standard (non-persistent) JVM on which the persistent JVM is based uses a simple mark-and-sweep garbage collection system for disposing of unusable objects. All objects that are reachable are marked. This involves checking all objects that are accessible from local variables and the operand stacks and recursively checking all those objects' fields. When all reachable objects have been marked, those that are unmarked are disposed of.

The victim-selection process that the garbage collector uses is currently the same as in the non-persistent virtual machine. The garbage collector uses reachability and PIDs will not be recognised as valid memory addresses, so objects that are referenced by PID will not be considered reachable. PID references to memory-resident objects will only appear if a stabilise has occurred since the
reference was last used. Naturally, reswizzling on stabilise (as discussed above) would remedy this. It may be better not to do this however, because the stabilise will have ensured that the latest version of that object was written to disk and hence the valuable memory that it was occupying can be used for another purpose. We will investigate the effects of this more fully.

When a persistent object is discarded from memory it is stabilised and removed from the RPOT. If a non-persistent object is discarded then it is not written to the store since if it is not reachable from any objects in memory then it is not possible for it ever to become reachable from any persistent object.

At the moment only unreachable objects are thrown away. This can be changed so that reachable persistent objects can be written back to the store and removed from memory in order to free more heap if needed. However, this would require dealing with all LAs that reference discarded objects.

4.7 Persistence and native methods

4.7.1 The problem with native methods

Because the Java bytecode is platform-independent and interpreted by the JVM, some other mechanism has to be provided for access to platform-specific services. These services include file I/O, any textual or graphical output and network access. To provide these services, Java allows native methods to be run. These are methods that have been written in C and compiled on for the architecture on which the JVM is running. The Java programmer calls these in the same way as calls to native methods in the same way as normal Java methods. It is the responsibility of the JVM to check a method call to see if it is a Java method or a native method and handle the invocation accordingly.

Because native methods are compiled to platform-dependent machine code and are not executed by the JVM, the normal mechanisms for discovering and swizzling PIDs and tracking updates that have been discussed cannot be applied. Programmers who have to use native methods must handle these aspects themselves. The Java Native Interface (JNI) provides a number of standard method calls that can be made from native methods to perform certain tasks. We have added a few calls to the JNI to allow PLaVa native methods in programs to handle persistence.

If these calls are not used then unreliable program behaviour will result. Because of this, the use of native methods in PLaVa programs is strongly discouraged and they should only be used where absolutely necessary.

4.7.2 Swizzling

Although all references in the method's parameters will be in LA form since they were taken from the operand stack, references to other objects from these objects may still be in PID form. Obviously, references should be in LA form before they can be used. To swizzle all references in objects that are passed to native methods would not be sufficient since any arbitrary chain of references could be followed by the native code.

The following calls must be used in native code before any reference is used (excluding of course those in the method's actual parameters). The calls will check a reference and swizzle it if it is in PID form, returning the LA so that it can be used.

- void SwizzleObjectField(JNIEnv*, jobject, jfield) This swizzles the requested field in the requested object and returns its LA. If the field is not a reference field then it returns
NULL without having taken any action.

- void SwizzleArrayElement(JNIEnv*, jreferenceArray, jint) This swizzles the requested element of the requested array and returns its LA. If the array is not an array of references then it returns NULL without having taken any action.

The return value of a native method will be put onto the operand stack so this must also be converted to an LA so as not to introduce PIDs onto the operand stack.

### 4.7.3 Recording updates

Since changes can be made to objects from within native methods, we have added a couple of calls to the JNI to deal with this.

- void MarkUpdated(JNIEnv*, jobject) This marks the specified object as having been updated.

- void MarkRefsUpdated(JNIEnv*, jobject) This marks the specified object as having had one or more of its references updated.

These calls must be called appropriately whenever an object is updated in a native method. If they are not called then there will be no record that the object has been updated and its changes will not be written into the database, resulting in inconsistency.

### 5 Related work

The object faulting architecture discussed in this paper above is used in a number of existing systems. Early systems were PS-ALGOL [ACC82, BC85] and LOOM [KK83] which was a persistent implementation of Smalltalk [GR83].

Another approach is to use a page faulting system, such as in [Wil91, WK92, VD92]. In this approach, objects are loaded into memory a page at a time. When a reference to an object in a non-resident page is dereferenced, that is made resident (thus bringing all the objects in it into memory). This system can utilise the built-in page fault mechanism of operating systems.

Swizzling can be done either eagerly or lazily. Eager swizzling swizzles all required objects (or a subset thereof) in advance whereas lazy swizzling swizzles objects only when they are needed. Swizzling can also be either direct or indirect. Direct swizzling brings the referenced object into memory when the reference is swizzled. Indirect swizzling only brings objects into memory when they are needed. In this latter case, the swizzling process is more complex, with the first step being to point the reference to a proxy object. When the object is used, the second step is done and this involves replacing the proxy object with the actual object that is loaded from the store. A study comparing the performance of various swizzling techniques in the context of the Mneme store [Mos90] is given in [Mos91, HM93] and in the context of the E programming language [Ric90] in [WD92].

Other persistent implementations of Java include PJama [AJDS96, Jor96] which is being developed by the Glasgow group in association with Sun Microsystems. The PJama system's architecture is similar to that of PLaVa in its basic design. The JavaSPIN approach [KMRW96] combines Java with the existing SPIN (Support for Persistence, Interoperability and Naming) framework. This
enables JavaSPIN to interact seamlessly with other languages that have SPIN extensions. There are also attempts to interface Java with relational databases such as [dST96, Gru96]. These systems should provide a Java interface to legacy data but it is unlikely that they will be able to attain a high degree of orthogonality.

References to most of the work that has been done in the field of persistent languages can be found in [AM95].

6 Conclusion

6.1 Summary

We have extended an existing JVM to support basic persistence. With the addition of a few lines of code to a standard Java program, it can be converted into a persistent application that will run on the PLaVa machine. The rest of the program need not be changed because the object faulting and update tracking are automatically done by the PLaVa virtual machine.

The virtual machine implements persistence by detecting which references that are being dereferenced are in PID form and converting them to LA form so that they can be used. This conversion involves automatically loading the object from the store. When an object is changed, a flag in its header is set to record that it has been updated. Updated objects are all written back to the store during a stabilise operation. All new objects that have become reachable from existing persistent objects are added to the store as well.

Decisions that were taken during the design were aimed towards keeping memory usage low to facilitate use on a small machine. Further investigation into these matters will be carried out as explained in the following section.

6.2 Future work

Now that we have demonstrated the feasibility of using the PLaVa system on a small machine, our emphasis turns to performance measurements. We'll test the effectiveness of various configurations of the virtual machine with regards to speed and space efficiency and tradeoffs between them. This involves selecting and deselecting alternative strategies for residency checking, object faulting, caching and pointer swizzling.

We wish to use an efficient, universal store with our virtual machine; stores that should be available in the near future include PJSL [PAD97] which is being developed by the Glasgow group for their PJama system and Stephen Blackburn's store which is based on his PSI interface model [BS97, Bla97]. PJSL is likely to be the most efficient as its design is based on experiences with the previous version of the PJama store and it is being supported by Sun Microsystems.

Other outstanding work includes the development of a remote store that will allow the PLaVa system to run on a small machine without a local disk. This machine would obtain all its data from a remote persistent store via a network. The internal mechanism of the persistent virtual machine that accesses the store will not need to be changed. The function calls that interact with the store will have to be modified to contact the remote store instead of accessing a local store.

We hope to implement the remote store interface using Java's remote method invocation (RMI) package, with the remote store being a Java object on a remote machine. We will investigate the efficiency of this approach and consider alternatives if the performance is not acceptable.
References


[GR83] A. Goldberg and D. Robson. Smalltalk-80: The Language and its Implementation. Addison-Wesley, 1983.


Siacsit ’97 265


