Effective Implementation of Object-Oriented Programming Languages
Dagstuhl-Seminar-Report: 00451
05.11.2000-10.11.2000

Organized by
Uwe Aßmann, Linköping University
Laurie Hendren, McGill University
Barbara Ryder, Rutgers University (on sabbatical leave 2000-2001 at IBM T.J. Watson Research Center)
Frank Tip, IBM T.J. Watson Research Center

Editor: Barbara G. Ryder, Uwe Aßmann

Contents

1 Introduction

Motivation

Object-oriented languages are becoming increasingly popular for the development of software systems of all kinds, ranging from small web-based applications to large server-side applications. Modern object-oriented languages such as Java contain features such as exception handling, dynamic binding, extensive control of visibility, and threads. Although these features add to a language's expressive power and provide many benefits from a software engineering point of view, they also make it more difficult to implement a language efficiently.

The purpose of this seminar is to bring together experts in the areas of object-oriented language design, analysis and translation with experts in run-time systems, and to investigate problems and techniques related to effective implementations of object-oriented languages. It is becoming increasingly apparent that all of these areas can contribute to the ultimate goal of obtaining efficient object-oriented implementations. For example, by choosing the right language features, the job of the compiler and run-time system can be made easier. Moreover, static compiler analyses can benefit from profiling information as a means for obtaining feedback on predictable program behavior, and for selecting profitable transformations.

Program

The seminar was organized around the general themes of garbage collection, call graph construction and JVMs for Java. There were invited tutorials on garbage collection and call graph construction algorithms as well as invited presentations by three Java compiler groups: IBM's Jalapeno compiler, Sun's Hotspot compiler and Microsoft's Marmot compiler. There were 24 additional 30 minute presentations by seminar participants which discussed topics in optimization, analysis, profiling, language design, object persistence and industrial applications. Each presentation session finished with at least a 15 minute question session; often discussions continued into the coffee break or lunch. One night was reserved for a demo session in which participants showed their software tools to other attendees. Many attendees were graduate students; all of them presented their research. Although the bad weather forced cancellation of the traditional Wedsnesday hike, participants did have a wine tasting dinner in Trier and an IBM-sponsored banquet as part of the seminar.

Goals

An important seminar goal was to gather industrial compiler groups and academic researchers together for substantive research discussions; this goal was achieved as exchanges over topics such as on-stack replacement, utility of SSA-like program representations, effectiveness of different garbage collection methodologies took place both in the seminar sessions and especially during evening informal gatherings. Another goal was to engender more communication between colleagues in the functional programming community and those in the object-oriented community; we were less successful in this effort as several of our functional programming colleagues declined our invitation. Finally, conversations at the seminar made it clear that many researchers, industrial and academic, feel that compiler infrastructure is a critical need of the community to enable better research; colleagues in attendance were sympathetic to providing research infrastructure for joint research between academic and industrial research folks.

Research Areas Uncovered

Many research questions focussed on Java and possible extensions to the programming language. It is clear that genericity and assertions are the awaited features that the community wants in Java, for example. Much discussion centered around how to provide those features of object-oriented languages that provide flexibility and dynamicism, while maintaining optimized execution performance. Dynamic class loading in Java was cited as a particularly difficult feature to accommodate.

The next big research accomplishment predicted by seminar participants is the development and use of practical interprocedural analyses and the optimizations they facilitate. For example, there was much discussion of how to perform effective "inlining"; no consensus was reached, but it was agreed that more empirical investigation is needed. In addition, participants agreed that thread management is an area needing work.

Acknowledgements

We would like to thank the Scientific Director for supporting us in organizing this seminar. We also want to thank the efficient staff at the Dagstuhl Office in Saarbrücken and at the Schloss Dagstuhl office, for all their help with our seminar. IBM Research supported the Dagstuhl centre and the seminar with a very generous contribution of DM 10,000, including a nice dinner on Thursday evening. Merci beaucoup! We also want to thank all of our participants, those who presented talks and those who participated in the invigorating discussions. The seminar was a success primarily because of the enthusiasm of the particpants. We also want to thank everyone for sending electronic copies of their talks for inclusion on our seminar webpage.

2 Programme

2.1 Garbage Collection

Single Address-Space Garbage Collection (Invited Tutorial)

Hans Boehm, HP Labs

We give an overview of basic garbage collection techniques, of the pointer-finding techniques that enable them, and of some of the refinements that are either in common use, or frequently discussed in the research community.

We point out that garbage collector performance characteristics are fundamentally different from systems based on explicit deallocation. We conclude with a brief overview of some recent research in the area of garbage collection, and present a few measurements of garbage collector performance and multiprocessor scalability algorithms.

Making the Trains Run on Time

Alex Garthwaite, Sun Microsystems

One approach to limiting pause times in the collection of older generations is to use an incremental collection technique. Collectors based on the train algorithm are one example: at each collection, a portion of the older generation is collected. Train-based collectors have advantages over other incremental collectors because they allow objects to be relocated. Unfortunately, there are several problems that make train-based collectors fail to live up to their promise:

- their reliance on remembered sets and the fact that these remembered sets may grow to represent the size of the heap
- the fact that objects in data structure may have to be copied $O(d^2)$ times before the collector may tell if they are garbage (where d is the depth of the data structure)
- the tension between collecting too aggressively (to keep up with allocation in the generation and to deal with the need to reorder data structures to identify garbage) and collecting at a slow pace (to allow objects a chance to die and to constrain pause times)

We will present techniques that allow us to overcome these limitations though:

- better remembered-set representation
- separation of issues such as the size of oversized objects versus the size of normal cars
- the isolation of popular objects through dynamic feedback
- the placement of objects in large (1GB) heaps to reduce the amount of floating garbage

The result is that it is possible to constrain pause times, achieve good throughput, and still support incremental, copying garbage collection.

Thread-Specific Heaps

Bjarne Steensgaard, Microsoft Research

In typical multi-threaded programs, many objects never escape the thread that created them. We propose allocating such objects in thread-specific heaps. A key property of such thread-specific heaps is absence of pointers into the heap from objects in other heaps. Objects that cannot be allocated in thread-specific heaps may be allocated in a shared heap.

We make a number of observations about thread-specific heaps:

1. Thread-specific heaps can be garbage-collected independent of other thread-specific heaps, and in particular can be garbage-collected concurrently with such heaps.

2. Having thread-specific heaps without cross-pointers is likely to increase locality during garbage collection.

3. Having smaller heaps that can be garbage-collected independently may decrease maximum memory usage for semi-space copying garbage collectors.

We report on our experience with thread-specific heaps in Marmot, an ahead-of-time Java-to-native code compiler.

On the Effectiveness of GC in Java

Ron Shaham, Tel Aviv University

We study the effectiveness of garbage collection (GC) algorithms by measuring the time difference between the actual collection time of an object and the potential earliest collection time for that object. Our ultimate goal is to use this study in order to develop static analysis techniques that can be used together with GC to allow earlier reclamation of objects. The results may also be used to pinpoint application source code that could be rewritten in a way that would allow more timely GC.

Specifically, we compare the objects reachable from the root set to the ones that are actually used again. The idea is that GC could reclaim unused objects even if they are reachable from the root set. Thus, our experiments indicate a kind of upper bound on storage savings that could be achieved. We also try to characterize these objects in order to understand the potential benefits of various static analysis algorithms.

The Java Virtual Machine (JVM) was instrumented to measure objects that are reachable, but not used again, and to characterize these objects. Experimental results are shown for the SPECjvm98 benchmark suite. The potential memory savings for these benchmarks range from 23% to 74%. We also built a tool to analyze the instrumentation results. Our tools can suggest the parts of the code where there is a potential for memory savings. These hints could be used by an automatic tool to perform profile-guided static analysis or by a programmer to rewrite code in order to save memory. Manual code rewriting based on the hints of our tool for SPECjvm benchmarks yielded up to 42% of actual memory savings.

Using Garbage Collection to Implement Cache-Conscious Data Layouts

Trishul Chilimbi, Microsoft Research

This talk will discuss a wide variety of techniques for producing cache-conscious object layouts in languages such as Java and Cecil. These techniques, which include class/object splitting and object co-location, leverage the garbage collector and are completely automatic, requiring no programmer assistance. The techniques are shown to improve the performance of several medium-sized Java benchmarks by 18-28% overall.

Packaging Java in Applications for Embedded Systems Running Big Things On Small Devices

Aldo Eisma, Object Technology International, Inc.

Traditionally, most embedded systems used to be programmed in languages like C and assembler because they were simple and static and had relatively long life cycles. With the advent of pervasive computing, bringing us all sorts of new dynamic, networked and complex embedded devices, there is a strong interest for programming these devices in the Java programming language. The limitations and special characteristics of embedded devices require a new approach for the Java VM, class libraries and supporting development tools. One of the main components in the embedded Java development environment is the packager. The packager collects Java class files and resources and builds a run-time image that can be stored in ROM, minimizing startup time and decreasing RAM requirements on the device. The images contain pre-linked optimized Java class files from which redundant and unused elements have been removed. Various techniques have been researched by IBM Watson and OTI, and have been implemented in IBM's VisualAge Micro Edition SmartLinker and VM components. An introduction to the issues and an overview of the techniques used is given in this presentation.

2.2 Compilers and Java Virtual Machines

A Marmot Overview

Todd Knoblock, Microsoft Research

Marmot is an optimizing compiler, runtime and libraries for Java. It includes a bytecode-to-native compiler. Our focus is on techniques that would allow 'industrial strength' applications to be programmed in Java without a significant performance penalty over traditional languages.

In this talk, the architecture of the Marmot internals is presented. It will be described how it was necessary to modify standard techniques and optimizations for use in Marmot. Also, other, less standard, techniques will be discussed that have been tried.

HotSpot Server Compiler: Experiences and Lessons

Chris Vick, Sun Microsystems

Runtime compilation has long been thought to be solely the domain of super fast template based compilers. In implementing the HotSpot Server Compiler, we chose to disregard much of the common wisdom of runtime compilation in order to explore what a full optimizing compiler could achieve in a runtime system. In doing so, we discovered that there were many myths in the common wisdom about runtime compilation. We also discovered that embedding an optimizing compiler in a runtime system can be very effective, and also very challenging. This presentation is a summary of some of the lessons that we learned, and some of the approaches that we took to the challenges that we encountered.

Adaptive Optimization in the Jalapeño JVM

Michael Hind, IBM T.J. Watson Research Center

Future high-performance virtual machines will improve performance through sophisticated online feedback-directed optimizations. This paper presents the architecture of the Jalapeno adaptive optimization system, a system to support leading-edge virtual machine technology and enable ongoing research on online feedback-directed optimizations. We describe the extensible system architecture, based on a federation of threads with asynchronous communication. We present an implementation of the general architecture that supports adaptive multi-level optimization based purely on statistical sampling. We empirically demonstrate that this profiling technique has low overhead and can improve startup and steady-state performance, even without the presence of online feedback-directed optimizations. The paper also describes and evaluates an online feedback-directed inlining optimization based on statistical edge sampling. The system is written completely in Java, applying the described techniques not only to application code and standard libraries, but also to the virtual machine itself.

The Sable VM

Etienne Gagnon, McGill University

This talk presents the architecture and goals of the SableVM open source Java virtual machine. SableVM is an bytecode interpreter, with objective of being both portable (C code, minimal assembly/machine specific code) and efficient.

A Single Intermediate Language That Supports Multiple Implementations of Exceptions

Norman Ramsey, Harvard University

For a compiler writer, generating good machine code for a variety of platforms is hard work. One might try to reuse a retargetable code generator from another compiler, but code generators are complex and difficult to use, and they limit one's choice of implementation language. One might try to use C as a portable assembly language, but C limits the compiler writer's flexibility and the performance of the resulting code. The wide use of C, despite these drawbacks, argues for a portable assembly language.

`C--` is a new language designed expressly as a portable assembly language. It comes with a run-time interface (specified in C) that a garbage collector, exception dispatcher, or debugger can use to inspect and modify the state of a suspended `C--` computation. This talk focuses on the mechanisms that enable `C--` to express four of the best known techniques for implementing exceptions, all within a single, uniform framework. Our approach clarifies the design space of exception-handling techniques, and it should allow a single `C--` optimizer to handle a variety of implementation techniques. Our ultimate goal is to allow a source-language compiler the freedom to choose its exception-handling policy, while encapsulating the architecture-dependent mechanisms and their optimization in an implementation of `C--` that can be used by compilers for many source languages.

Compiling Lazy Functional Languages for the Java Virtual Machine

David Wakeling, School of Engineering and Computer Science, University of Exeter

For some time now, we have been interested in the efficient implementation of lazy functional languages on very small computers, such as those found in consumer electronics devices. So far, our implementations have assumed that next-generation products will be controlled by previous-generation RISC processors. But Sun's Java processors, with their compact instruction encoding, are an attractive alternative. This talk considers whether these processors, which are designed to run only Java programs, could successfully run lazy functional programs instead.

Alternatives to the Java Virtual Machine - Mobile Code Representations Aiding Code Generation

Michael Franz, UC Irvine

We introduce SafeTSA, a type-safe mobile code representation based on static single assignment form. We are developing SafeTSA as a replacement technology to the Java Virtual Machine (JVM), over which it has several advantages. (1) SafeTSA is better suited as input to an optimizing code generator and allows CSE to be performed at the code producer's site. (2) SafeTSA uses "type separation" to achieve provable type safety, which reduces the code verification effort at the code consumer's site considerably. (3) SafeTSA can transport the results of type and range-check elimination in a tamper-proof manner, and lastly (4) SafeTSA is usually considerably more compact than JVM code.

2.3 Programming Language Design

Genericity in the Java Programming Language: Key Issues

Gilad Bracha, Sun Microsystems

This talk will focus on the more controversial issues involved in adding genericity to the Java programming language. These are primarily support abstracting over primitive types and run-time support for generic types. The implications for compatibility and performance will be emphasized.

Bit-Level Object-Oriented Programming (BLOOP)

David F. Bacon, IBM T.J. Watson Research Center

Object-oriented programming languages have always distinguished between "primitives" and "user-defined" data types, and in the case of languages like C++ and Java, the primitives are not even treated as objects, further fragmenting the programming model. This distinction is aesthetically unappealing, creates continuing pressure to expand the set of primitive types in the language, and requires an external specification of the semantics of primitive types.

BLOOP provides a uniform reference-based object model down to the bit level, and allows primitive types to be defined within the language. While the semantics is uniform, implementations need not be, and complicated types can be represented very efficiently and often stored in registers.

The result is a language that allows programmers to expand the set of primitive types in a simple and natural way, and provides many efficiency benefits.

Extending Java with Language Constructs for High-Performance Computation

Kees van Reeuwijk, Delft University of Technology

Although Java was not designed for high-performance computations, it turns out that with a few extensions to the language, it is also useful in that area. The most important addition is the notion of multi-dimensional array. We also added support for the construction of specialized array representations, such as block arrays, sparse arrays, symmetric arrays, specific sorts of generic types, overloading of the subscript operator, and tuples.

Reflection Using Hierarchical Iterators

Hanspeter Moessenboeck, University of Linz

Reflection deals with obtaining information about executing programs. This includes static information such as the structure of types, fields and methods as well as runtime information such as the contents of the activation stack and the heap.

We show a simple technique for accessing this metainformation in a uniform way. The general idea is to organize the metainformation as a nested sequence of elements that can be traversed using iterators (so-called *riders*). Instead of keeping metaobjects in main memory we store them in a reference file and let the iterators parse the file. In this way, reflection imposes only a negligible memory overhead. Users don't have to pay for this feature if they don't use it.

In contrast to some other reflection models iterators not only allow us to explore the static program structure but also the runtime data. For example, an iterator can traverse the activation frames on the stack, zoom into a particular frame, traverse the variables in this frame, zoom into a structured variable and so on until the desired information is found.

Our reflection model has been implemented for the Oberon system and has been used for a number of system programming tasks including a general IO module, an exception handling mechanism and a database binding.

Multiparadigm Extensions to Java

Timothy A. Budd, Department of Computer Science, Oregon State University

In 1995 my students and I developed Leda, a multiparadigm language based on the Pascal model. Leda allowed programmers to create abstractions in an object-oriented, functional, or logic programming style. More recently, we have been interested in recreating this work but this time using Java as the language basis. The objective is to add as few new operations as possible and to make these operations seem as close to Java as possible so that they seem to fit naturally in to the language. To date we have proposed facilities for breaking apart composed objects - sometimes called unboxing - functions as first class values, for pass-by-name parameters, and for relational (or logic) programming.

Active Object System (AOS)

Jürg Gutknecht, ETH Zürich

Active objects in our terminology are objects together with an integrated thread of control. Based on the metaphor of active objects and on the Oberon programming language we have implemented an extremely compact light-weight kernel called AOS, running on Intel SMP (symmetric multiprocessor) platforms. The runtime model used is a dynamic population of communicating and collaborating 'agents' (active objects). They come in a great variety of forms and granularities comprising, for example, device drivers, network controllers, interaction commanders, and remote terminals (Telnet and VNC). Active components can easily be plugged-in and plugged-out, which makes AOS ideally suitable to support a wide spectrum of applications including servers and 'wearable devices'. Worth to be mentioned from the implementation perspective are system-managed conditions (replacing the traditional signals and contributing to the scalability of the system) and a carefully thought-through modular ordering that statically guarantees absence of deadlocks up to a very high degree.

The new ASF+SDF Meta-Environment

Mark van den Brand, CWI

ASF+SDF is an algebraic specification language for specifying the syntax and semantics of programming languages. Its main application area is language prototyping, domain specific languages, and software renovation. ASF+SDF is supported by an integrated programming environment: the ASF+SDF Meta-Environment. In the first generation of this environment the emphasis was on incremental and lazy generation techniques with respect to scanning, parsing, and rewriting. We are about to release the second generation of this environment. The emphasis now is on advanced rewriting techniques with respect to ASF+SDF, the use of a scannerless parsing techniques, and reuse of components to develop efficient stand-alone environments. One of the components is a compiler for translating ASF+SDF specifications into C code. This C code is based on a library which ensures maximal sharing of terms and provides automatic garbage collection. This enables us to generate very efficient code that uses a minimal amount of memory when being executed. This compiler is completely specified in ASF+SDF and is bootstrapped.

2.4 Callgraph Construction, Analysis, and Optimization

An Overview of Popular Call-Graph Construction Algorithms

Jens Palsberg, Purdue University

We present four popular call-graph construction algorithms: RA (Reachability Analysis), CHA (Class Hierarchy Analysis), RTA (Rapid Type Analysis), and 0-CFA. Using a set-based framework, it is easy to see how they relate, and to see that they are instances of the same algorithmic idea.

Scalable Propagation-Based Call Graph Construction Algorithms

Frank Tip, IBM T.J. Watson Center

Propagation-based call graph construction algorithms have been studied intensively in the 1990s, and differ primarily in the number of sets that are used to approximate run-time values of expressions. In practice, algorithms such as RTA that use a single set for the whole program scale well. The scalability of algorithms such as 0-CFA that use one set per expression remains doubtful.

In this paper, we investigate the design space between RTA and 0-CFA. We have implemented various novel algorithms in the context of JAX, an application extractor for Java, and shown that they all scale to a 325,000-line program. A key property of these algorithms is that they do not analyze values on the run-time stack, which makes them efficient and easy to implement. Surprisingly, for detecting unreachable methods, the inexpensive RTA algorithm does almost as well as the seemingly more powerful algorithms. However, for determining call sites with a single target, one of our new algorithms obtains the current best tradeoff between speed and precision.

Practical Method Call Resolution for Java

Laurie Hendren, McGill University

This paper addresses the problem of resolving virtual method and interface calls in Java bytecode. The main focus is on a new practical technique that can be used to analyze large applications. Our fundamental design goal was to develop a technique that can be solved with only one iteration, and thus scales linearly with the size of the program, while at the same time providing more accurate results than two popular existing linear techniques, class hierarchy analysis and rapid type analysis.

We present two variations of our new technique, variable-type analysis and a coarser-grain version called declared-type analysis. Both of these analyses are inexpensive, easy to implement, and our experimental results show that they scale linearly in the size of the program.

We have implemented our new analyses using the Soot framework, and we report on empirical results for seven benchmarks. We have used our techniques to build accurate call graphs for complete applications (including libraries) and we show that compared to a conservative call graph built using class hierarchy analysis, our new variable-type analysis can remove a significant number of nodes (methods) and call edges. Further, our results show that we can improve upon the compression obtained using rapid type analysis.

We also provide dynamic measurements of monomorphic call sites, focusing on the benchmark code excluding libraries. We demonstrate that when considering only the benchmark code, both rapid type analysis and our new declared-type analysis do not add much precision over class hierarchy analysis. However, our finer-grained variable-type analysis does resolve significantly more call sites, particularly for programs with more complex uses of objects.

A Study of Devirtualization Techniques for a Java Just-In-Time Compiler

Kazuaki Ishizaki, IBM Tokyo

Many devirtualization techniques have been proposed to reduce the runtime overhead of dynamic method calls for various object-oriented languages, however, most of them are less effective or cannot be applied for Java in a straightforward manner. This is partly because Java is a statically-typed language and thus transforming a dynamic call to a static one does not make a tangible performance gain (owing to the low overhead of accessing the method table) unless it is inlined, and partly because the dynamic class loading feature of Java prohibits the whole program analysis and optimizations from being applied. We propose a new technique called direct devirtualization with the code patching mechanism. For a given dynamic call site, our compiler first determines whether the call can be devirtualized, by analyzing the current class hierarchy. When the call is devirtualizable and the target method is suitably sized, the compiler generates the inlined code of the method, together with the backup code of making the dynamic call. Only the inlined code is actually executed until our assumption about the devirtualization becomes invalidated, at which time the compiler performs code patching to make the backup code executed subsequently. Since the new technique prevents some code motions across the merge point between the inlined code and the backup code, we have furthermore implemented recently-known analysis techniques, such as type analysis and preexistence analysis, which allow the backup code to be completely eliminated. We made various experiments using 16 real programs to understand the effectiveness and characteristics of the devirtualization techniques in our Java Just-In-Time (JIT) compiler. In summary, we reduced the number of dynamic calls by the average of 40.2%, and we improved the execution performance with the geometric mean of 17%.

A Framework for Reducing the Cost of Instrumented Code

Matthew Arnold and Barbara G. Ryder, Rutgers University

Instrumenting code to collect profiling information can cause substantial execution overhead. This overhead makes instrumentation difficult to perform at runtime, often preventing many known offline feedback-directed optimizations from being used in online systems. We present a general framework for instrumenting code that uses fine grained sampling to allow previously expensive instrumentation to be performed accuratly with low overhead. Our framework does not rely on any hardware or operating system support and is fully tunable; the sample rate can be adjusted at any time to match the type of instrumentation being performed. By reducing the overhead of instrumentation, our framework eliminates one of the biggest obstacles to performing feedback-directed optimizations at runtime. We present experimental results validating the overhead and accuracy of our technique.

On the Predictability of Java Byte Codes

Karel Driesen, McGill University

Java byte codes are platform-independent. That means that any characterization of Java applications at the byte code execution level will reveal characteristics that any Java Virtual Machine will have to deal with, no matter whether this JVM is a Just-In-Time native code optimizing compiler running on a state-of-the-art high-performance workstation, or a byte code interpreter running in a watch. We believe that predictability profiles are particularly well-suited to capture and visualize program behavior, at a variable level of detail, as required by a systems architect interested in control flow, data flow, or automatic memory management.

We present predictability profiles for 6 SPECJVM98 programs, for three byte code sub traces, Invoke (polymorphic call target prediction) Load (load effective address prediction) and New (new effective type prediction). For example, for Invoke byte codes, we measured the prediction rate achieved by invoke target predictors within every 20000 byte codes of the first 2 million byte codes executed using an unlimited, fully accurate BTB, and of two-level predictors of path lengths 1,2,4,8, and 16. Prediction profiles for all these predictors are generally close together, but usually a BTB performs best in variable program phases.

A Framework for Deep Analysis of Java Programs

Manish Gupta, IBM T.J. Watson Research Center

We describe a new framework for extensive interprocedural analysis of Java programs. Our framework is able to handle dynamic features of Java like dynamic class loading and runtime binding of methods, and yet, it avoids the high cost associated with runtime compilation. The first part of our talk presents quasi-static compilation, which reduces the cost of dynamic compilation, while maintaining compliance with the dynamic features of Java. The quasi-static compiler generates persistent code images ahead of time, and during a production run, performs validation checks and adapts those images to the new execution context while preserving global optimizations and maintaining binary compatibility. A preliminary implementation of this approach in the Jalapeno VM has led to performance improvements ranging from 9% to 91% for the SPECjvm98 benchmarks with size 100, and 54% to 356% for the (shorter running) SPECjvm98 benchmarks with size 10. The second part of the talk describes an extension of the extant analysis framework (presented recently by Sreedhar et al.) in the context of the quasi-static compiler, which enables global analyses and optimizations in the presence of dynamic class loading. Our approach enables transformations like unguarded devirtualization and/or inlining of virtual methods in cases that are beyond the scope of previously known techniques.

Interprocedural Value Flow Analysis

Erik Ruf, Microsoft Research

We describe an approach for interprocedural value flow optimizations for type safe object oriented programs. A generic analysis based on flow-insensitive method summaries achieves efficient context sensitive propagation of arbitrary attributes describing runtime values. This talk describes the basic approach, its use in a variety of applications, and some improvements presently under investigation.

Implementing Java Exceptions Efficiently

Nigel Horspool, U. Victoria

Throwing and catching an exception in Java is remarkably expensive. Implementers have apparently taken the view that exceptions are rare events and that efficiency is unimportant. We take the view that exceptions should be a standard programming pattern and that efficiency does matter. We first examine where all the time is taken. Then, based on these experimental observations, we recommend strategies for making significant reductions in execution time. Our recommendations do not require any changes in the class file format. We further propose modest changes to the Java language and to the class library that would ease the burden on the implementer of the exception handling mechanism.

Acknowledgement: Mike Zastre has contributed to this work.

Program Optimizations Enabling/Enabled by Object Persistence

Anthony Hosking, Purdue University

Techniques for aggressive optimization of programs often rely on analyses and transformations that cut across the natural abstraction boundaries that allow for separate compilation of the source programming language. These atomic compilation units typically correspond to natural encapsulation boundaries, such as procedures in procedural languages, or classes in class-based object-oriented languages like Java. Such aggressive analyses and optimizations then take into account the particular combination of units that make up a given application program, and specialize the code from each unit to that particular combination. For statically-linked languages such as C and Modula-3, where the units that make up a given program are known statically, aggressive (and possibly expensive) analysis and optimization can be performed off-line. Unfortunately, execution environments for more dynamic languages such as Java link code into the application as it executes, precluding analyses and optimizations that may prove too expensive to apply on-line. Moreover, even for analyses that are not too expensive to apply on-line, some candidate optimizing transformations that are safe to apply at one time may subsequently become invalid if the code base evolves in a way that violates their safety.

Fortunately, persistent object systems usually treat the code base as an integral part of the persistent store. For example, the PJama prototype of orthogonal persistence for Java captures all classes loaded by a persistent application and stores them in the persistent store. This code base approximates the notion of “whole-program” that has been exploited in other optimization frameworks.

We argue for an analysis and optimization framework that operates against the code base of the PJama persistent store, and which couples the results of analysis and optimization with PJama's run-time system to ensure continued correctness of the resulting code. The framework can perform extensive analysis over the code in the persistent store, supporting optimizations that cut across class boundaries in ways that could not be safely achieved with stand-alone Java classes.

Demand-Driven Type Analysis: An Efficient Means for Just-In-Time Compilation?

Jens Knoop, U. Dortmund

Replacing late by early method binding, where possible, is a major source of improving the performance of object-oriented programs. Classical type analysis (TA) techniques, however, enabling this are often too costly, particularly for usage in just-in-time (JIT) compilers as information is typically computed in an exhaustive fashion. Recently, however, demand-driven analysis techniques have been developed aiming at computing information selectively by need. These techniques have successfully been used for a variety of problems ranging from GEN/KILL-analyses over constant propagation to array bounds check elimination on demand in Java. In particular, their ability of focusing on "hot" program regions suggests their adequacy for JIT compilation. Though the idea of adopting these approaches to TA is obvious, there are no such approaches in practice. In this talk we investigate the inherent reasons of this depressive fact. To this end we reconsider the state-of-the-art approaches for demand-driven DFA. Extracting their strengths and limitations, we demonstrate why TA is beyond their scope, before pointing to extensions aiming at overcoming the limitations.

Object Inlining

Peeter Laud, Universität des Saarlandes

Java has a uniform object model, which means that each object is represented as a pointer to the actual data of the object, which resides in the heap. Implementing the language in this way creates several kinds of overhead: more pointer dereferences, more work for the garbage collector, less locality of data (this reduces cache efficiency).

Object inlining is an analysis (or a set of analyses), attempting to determine, which objects could be stored together as a "compound object", without changing the semantics imposed by the uniform object model.

Being a kind of representation analysis, object inlining needs some sort of heap analysis. We have attempted to craft an analysis which is no more powerful than it is necessary for our purposes. This analysis is somewhat weaker than a full-blown alias analysis.

The actual set of analyses includes the type inference and several simple bit-vector analyses. Thus the cost of doing them should not be prohibitive. In our talk we described our analysis and tried to give an impression that it is possible to automatically inline as many of the fields as the programmer might be able to inline by hand.

Towards structuring the object store

Arnd Poetzsch-Heffter, Universität Hagen

In most object-oriented languages, the object store is an unstructured container of arbitrarily linked objects. Language implementation and programming techniques can benefit from better structured object stores. In particular, more accurate data flow information can be computed and garbage collection and memory management can be improved. In addition, enhanced encapsulation concepts (e.g. control of representation exposure) can be better supported.

The talk presents a model for a hierarchically structured object store and investigates typing techniques and syntactic constructs for access restriction to statically enforce the structure.

Heap Analysis on SSA

Florian Liekweg, Universität Karlsruhe; Martin Trapp, IBM T.J. Watson Research Centre, New York; Uwe Aßmann, Universität Karlsruhe; Gerhard Goos, Universität Karlsruhe

We present the VDG-based intermediate format FIRM, which builds on the works of C. Cliff, B. Steensgard, et. al. This format includes abstractions for heap memory and heap analysis. It is used in the Karlsruhe compiler for the modern object-oriented language Sather-K which supports multiple inheritance, generics, garbage collection, and value types. We present approaches and ideas for memory fragmentation, use-def-based optimisations, and static garbage collection.

Interprocedural, Path-specific Optimization

David Melski, U. Wisconsin-Madison, USA

Ammons and Larus presented a technique for improving the results of data-flow analysis along hot paths [Ammons-Larus, PLDI'96]. This work is an extension of their work based on our interprocedural path-profiling technique. Rather than using the intraprocedural path-profiling technique of Ball and Larus, we use a technique that collects information about interprocedural paths (i.e., paths that may cross procedure boundaries). Hot interprocedural paths are duplicated, and dataflow analysis is performed; this may lead to sharper data-flow facts along the duplicated (hot) paths.

Preliminary results suggest that this is not a fruitful extension of the original work. In the experiments on constant propagation, only small increases in the number of constants (on the order of hundreds, weighted dynamically) are found. Meanwhile, the transformation for duplicating interprocedural paths incurs a much higher run-time overhead than the gain made from using the new constants.

Points-to-Analysis for Java

Mirko Streckenbach, University of Passau

Points-to analysis for Java is different from points-to for C or even C++. We present a framework which generalizes popular points-to algorithms and generates set constraints from full Java bytecode. The framework exploits previously computed points-to sets in a fixpoint iteration for precise resolution of dynamic binding. We then compare implementations of this framework for unification-based and subset-based analysis. It turns out that – in contrast to the C situation – both approaches have about the same running time, while the subset-based algorithm is still more precise. The unification-based method is slowed down because its inherent imprecision accumulates during fixpoint iteration.

2.5 Demos

A Demo of the Animorphic Smalltalk System (Demo)

Gilad Bracha, Sun Microsystems

The Animorphic Smalltalk system was a high-performance (2.5-4X commercial implementations) Smalltalk environment that included a variety of novel features, including: A high-performance VM combining dynamic profile based compilation and interpretation that was the precursor to the Hotspot Java virtual machine, mixins as a built-in feature in the VM and optional typechecking using the Strongtalk type system as well as a glyph-based UI, novel code browsers, mirror-based reflection and a typed blue-book library.

Active Objects in Oberon

Jürg Gutknecht, ETH Zürich

This demo presents the AOS Oberon system. AOS contains an extremely compact light-weight kernel, running on Intel SMP (symmetric multiprocessor). It is based on the metaphor of active objects and on the Oberon programming language.

The new ASF+SDF Meta-Environment

Mark van den Brand, CWI

ASF+SDF, the algebraic specification language for specifying the syntax and semantics of programming languages, is supported by an integrated programming environment: the ASF+SDF Meta-Environment. The second generation of this environment puts its emphasis on advanced rewriting techniques, the use of a scannerless parsing techniques, and reuse of components to develop efficient stand-alone environments.

JAX - A Tool for Analyzing and Optimizing Java Class Hierarchies

Frank Tip, IBM T.J. Watson Research Center

We have implemented various novel call graph construction algorithms in the context of JAX, an application extractor for Java, and shown that they all scale to a 325,000-line program.

Realising Connectors in Java with the Software Composition System COMPOST (Demo)

Uwe Aßmann, Universität Karlsruhe/Linköpings Universitet

COMPOST (the Compostion System) is a Java library which can be employed to compose and connect Java components. This demo shows how connectors in COMPOST can transform classes to insert appropriate communication statements. This mechanism increases reuse of classes and leads to scalable systems.