1995

Optimizing Object Invocation Using Optimistic Incremental Specialization

Jon Inouye
Oregon Graduate Institute of Science & Technology

Andrew P. Black
Oregon Graduate Institute of Science & Technology, black@cs.pdx.edu

Charles Consel
Oregon Graduate Institute of Science & Technology

Calton Pu
Oregon Graduate Institute of Science & Technology

Jonathan Walpole
Oregon Graduate Institute of Science & Technology

(POSITION PAPER)

Optimizing Object Invocation Using Optimistic Incremental Specialization *

Jon Inouye, Andrew Black, Charles Consel, Calton Pu, and Jonathan Walpole

Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology

1 Introduction

To make object invocation efficient, it is important to minimize overhead. In general, overhead is incurred in order to maintain transparency; with the advent of mobile computer systems, persistence, increasing security and privacy concerns, transparency becomes more expensive and overhead is increasing. Invocation mechanisms maintain transparency by finding objects, choosing communication media, performing data translation into common formats (e.g., XDR), marshalling arguments, encrypting confidential data, etc. Performing all of these operations on every invocation would lead to unacceptable performance, so designers often avoid operations by specializing object invocation for more restricted environments. For example, the Emerald compiler [9] performs several optimizations when an object is known to be always local: the object is referenced with a location-dependent pointer that saves both space and access time and the invocation code performs no residency checks. Additionally, if the concrete type of the implementation is known, operations on the object can be in-lined. Unfortunately, if the object cannot be guaranteed to be local at compile-time, the Emerald compiler cannot perform any of these optimizations.

Contemporary distributed object systems remove overhead by building invocation mechanisms out of multiple modules. Each module provides functionality for a specific situation. Run time checks are inserted into the invocation path to interpret the situation and select the appropriate module. COOL [12] optimizes local invocation by making use of the C++ virtual function mechanism to convert from remote calls to direct calls and vice-versa. During every invocation[1] COOL implicitly checks the server's location. When the client and server are located in the same address space, the private virtual pointer of the interface object is modified to point directly to the virtual table of the server's class. The problem with this approach is that the invocation interface has to interpret the caller's context in order to choose the appropriate specializations.

This paper advocates a general technique, called *optimistic incremental specialization*, that addresses two limitations mentioned previously. First, can we optimize on "invariants" that are not guaranteed? Second, can we use specialized implementations and avoid inserting run-time checks in the invocation path? Section 2 describes optimistic incremental specialization and section 3 discusses our current status and open issues. We review related research in section 4 and summarize in section 5.

*This research is partially supported by ARPA grant N00014-94-1-0845 and grants from the Hewlett-Packard Company.

[1] Checks on every invocation are performed only on COOL systems that support mobility.

2 Optimistic Incremental Specialization

Optimistic incremental specialization (OIS) has its conceptual roots in the programming language field of partial evaluation [5, 6]. Partial evaluation is a program transformation technique aimed at specializing a program with respect to parts of its input that are invariant. It provides the basis for a simple and automatic approach for specializing programs. OIS extends partial evaluation in two ways. First, it allows specializing over time, and second, it allows specialization on assumptions.

Specializing over time. The term incremental in OIS refers to specialization after the compilation stage. With OIS, specialization may take place at any time when information becomes available, including compile, boot, link, and execution time. Long-lived systems, like operating systems, gather information from a variety of sources not available to a compiler. As invariants become known, the system can use that information to optimize itself.

Specializing on assumptions. OIS also allows specialization on assumptions that may not be true at all times; we refer to these assumptions as *quasi-invariants*. In a distributed object system, there are many things that are *likely* to be constant for long periods of time, but may occasionally vary. For example, object location might be considered a quasi-invariant. When a object becomes resident, it is likely to remain resident for a significant amount of time. OIS specializes using these quasi-invariants by assuming they will remain true.

Preserving correctness. If specialized code is generated and used, based on quasi-invariants that hold most of the time, then performance should improve. However, the system must correctly handle the cases where the quasi-invariants do not hold. Correctness can be preserved by checking, or *guarding*, every place where quasi-invariants may become false. Thus OIS still performs checks, but it moves the checks out of the frequently traversed paths into sporadically used paths. In the example above, rather than place the residency guard in the object invocation path, we place the guard in the module that supports object mobility, where it is less likely to be executed (since objects are probably invoked more frequently than they are moved). These guards are pure software constructs, i.e., guards do not require processor hardware to detect the invalidation of quasi-invariants.[2]

Re-specialization. If the guard is triggered, the object invocation mechanism must be "unspecialized", either to the completely generic routine (with the run-time checks) or, more accurately, to another specialized version that still capitalizes on the other invariants and quasi-invariants that remain valid. We call the process of replacing one version of a routine by another *replugging*.

Potential Rewards. If the optimistic assumptions about a program's behavior are correct, the specialized code will function correctly. If one or more of the assumptions become false, the specialized code must be replugged. This transformation will be a net performance win if the specialized code is executed many times, i.e., if the savings that accrue from the optimistic assumption being right, weighted by the probability that it is right, exceed the additional costs of the replugging step, weighted by the probability that it is necessary, plus the initial specialization cost.

[2] Copy-on-write memory is an optimistic technique that uses hardware features of the processor's memory management unit (MMU) to detect writes to logically separate, but physically shared, pages.

Application to Distributed Object Systems Consider the case of an object that is heavily used but rarely modified. COOL optimizes object invocation by mapping called objects into the address space of the calling object. COOL relies on the Chorus Distributed Virtual Memory server to maintain consistency between the replicated objects. OIS provides an alternative solution. Objects can be assumed immutable and replicated on demand. Guards can be placed in methods that may modify the object; triggering a guard initiates a replugging operation that invalidates remote replicas. The differences between OIS and COOL are that the OIS mechanism requires no virtual memory support from the operating system and provides consistency at the object-level rather than at the page level. If the replicated objects are rarely modified, this will lead to higher availability and more local invocations. However, if the objects are modified frequently, the system will generate too many invalidation actions and performance will suffer.

3 Discussion

Optimistic incremental specialization is a general methodology that combines both compiler and operating system technology. We are currently experimenting by applying OIS by hand to both a commercial file system [4] and the COOL [10] object invocation module. We are using this experience to develop tools for automating this process.

Current Status. Automating OIS involves building several tools including compile- and run-time specializers for C programs and a guard tracker that assists in identifying locations where quasi-invariants can be invalidated and supports run-time re-specialization. We currently have a static partial evaluator that specializes C programs with respect to invariants known at compile-time. We are completing the development of a dynamic partial evaluator that specializes C programs with respect to invariants known at run-time. It includes a template generator and dynamic linker.

Open Issues. There are a number of difficult problems that we have not yet resolved. Included are issues involving quasi-invariant selection, classifying invariants, and run-time support for guards and replugging. Identifying good quasi-invariants often requires domain knowledge. In our last example, if objects were assumed immutable and yet frequently modified, specialization would probably degrade object invocation. We really want to replicate only those objects unlikely to be modified; this can be difficult for a compiler to do without having some form of profiling information. One solution would be to use a meta-interface [11], that would allow the programmer to inform the compiler and system that selected objects are unlikely to be modified. Another solution would be to use software feedback [14].

Invariants. Different types of invariants require different optimization strategies. Actual invariants, those that are guaranteed, are specialized without guards. Quasi-invariants that are changed infrequently, can be specialized are unstable to different degrees. Some quasi-invariants are rarely invalidated, while others are invalidated quite frequently but are restricted to a small set of values in their range. For example, an integer value changes at each invocation, but only between the values of 0, 1, and 2. It is also possible to specialize on such a highly-unstable quasi-invariant but a different type of specialization is required.

Guard Placement. Preserving correctness requires guarding all the places where quasi-invariants can become false. As we have mentioned previously, OIS attempts to move checks out of the fre-

quently used paths into infrequently used paths. Object-oriented languages provide strong data encapsulation that facilitates guard placement. Quasi-Invariants, e.g., data values, can be made accessible only through well-defined interfaces. It is important for the programmer to code in a manner where each quasi-invariant is encapsulated as a predicate on an object. Some quasi-invariants, such as those associated with the semantics of a particular systems, need to be explicitly encoded as data values.

Replugging. Run-time support for replugging needs to be carefully implemented. In order for re-specialization to be efficient, the compiler and run-time system need to have some form of agreement on how code segments will be replaced. Code cannot be safely replaced while it is being used. The dynamic specializer also needs to take into account which quasi-invariants are being used, i.e., guards left by previous unrelated specializations should not be removed by the current specialization.

4 Related Work

There has already been a considerable amount of research in optimizing local communication including the Firefly RPC [15], LWRPC [3], and L3 [13] implementations. This work has concentrated on developing a specialized implementation for specific situations and does not address minimizing the overhead involved in analyzing the current situation.

Of all object invocation architectures, Zenith's "management by exception" technique is the most similar to optimistic incremental specialization [7]. Zenith is a persistent mobile object system in which object invocation is tailored to an *active stationary object*. Zenith maintains an ordered list of address hints; the first hint is always used. If this reference fails, the exception handler will clean things up. The Zenith exception handler is similar to the OIS guard in that it must resolve the problem and "do the right thing". The primary difference is the Zenith exception handler may have little semantic information passed to it. In terms of detecting invalid references, a hardware fault is passed back up to the TLB miss handler which fails to handle it and passes it to the page fault handler, etc., until it finally reaches the application. Layering is used to rebuild semantic information; this fault was caused because something the programmer thought might happen really did occur. An OIS guard and replugger performs the same operation above the operating system without having to go through all the preprocessing *in* the operating system.

There has been considerable interest in allowing user-specified modules to replace system modules. Two other object invocation models, the Substrate Object Model [2, 1] and the Kernel Tool Kit (KTK) [8] allow an invocation implementation to be extended or replaced using a meta-interface. In this paper, we have concentrated on removing overhead for efficient object invocation given a fixed set of implementations. Substrates and KTK objects allow new implementations to be added dynamically, which complicates the use of OIS. Interoperability with binary modules is another open issue we need to consider.

5 Summary

We believe optimistic incremental specialization will be effective in minimizing overhead in object invocation implementations. Reducing invocation overhead will become more important as distributed object systems continue to maintain various forms of transparency in dynamically diverse environments.

OIS is a novel optimization methodology based on both compiler and operating system technology. It is this combination that allows OIS to perform specializations over time in addition to specializing on quasi-invariants. While the technology is still at an early stage of development, it has great potential for being applied to a wide variety of areas where interpretation costs are significant.

References

[1] Arindam Banerji and David L. Cohn. An Infrastructure for Application-Specific Customization. In *Proceedings of the ACM European SIGOPS Workshop*, September 1994.

[2] Arindam Banerji, Dinesh Kulkarni, John Tracey, and David L. Cohn. The Substrate Object Model and Architecture. In *Proceedings of International Workshop on Object-Orientation in Operating Systems*, Asheville, NC, December 1993.

[3] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight Remote Procedure Call. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 102–113, December 3-6 1989.

[4] Andrew Black, Charles Consel, Calton Pu, Jonathan Walpole, Crispin Cowan, Tito Autrey, Jon Inouye, Lakshmi Kethana, and Ke Zhang. Dream and Reality: Incremental Specializatoin in a Commercial Operating System. Technical Report CSE-95-001, Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, March 1995.

[5] Charles Consel and Olivier Danvy. Tutorial Notes on Partial Evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.

[6] Charles Consel, Calton Pu, and Jonathan Walpole. Incremental Partial Evaluation: The Key to High Performance, Modularity, and Portability in Operating Systems. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Copenhagen, June 1993.

[7] N. Davies, M. Davy, G. S. Blair, and J. A. Mariani. Object Invocation and Management in Distributed Multimedia Systems. Technical Report MPG-92-09, Department of Computing, Lancaster University, 1992.

[8] Ahmed Gheith, Bodhisattwa Mukherjee, Dilma Silva, and Karsten Schwan. KTK: Configurable Objects and Invocations. In *Proceedings of Third International Workshop on Object Orientation in Operating Systems*, pages 236–240, Asheville, NC, December 1993.

[9] Norman Hutchinson. *Emerald: An Object-Based Language for Distributed Programming.* PhD thesis, Department of Computer Science, University of Washington, January 1987. Also available as technical report 87-01-01.

[10] C. Jacquemont, F. Herrman, P. S. Jenson, P. Gautron, J. Mukerji, H. G. Baumgarten, and H. Hartlage. COOL: The CHORUS CORBA Compliant Framework. In *Proceedings of the IEEE Computer Conference (COMPCON)*, pages 132–141, San Francisco, California, February 1994.

[11] Gregor Kiczales and John Lamping. Operating Systems: Why Object-Oriented? In *Proceedings of Third International Workshop on Object Orientation in Operating Systems (IWOOOS-III)*, pages 25–30, Asheville, NC, December 1993.

[12] Rodger Lea, Christian Jacquemont, and Eric Pillevesse. COOL: System Support for Distributed Programming. *Communications of the ACM*, 36(9):37–46, September 1993.

[13] Jochen Liedtke. Improving IPC by Kernel Design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 175–188, December 5-8 1993.

[14] H. Massalin and C. Pu. Fine-grain adaptive scheduling using feedback. *Computing Systems*, 3(1):139–173, Winter 1990. Special Issue on selected papers from the Workshop on Experiences in Building Distributed Systems, Florida, October 1989.

[15] Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.