Hey, You Got Your Language In My Operating System!

Jon Howell*
Mark Montague

Technical Report PCS-TR98-340†

Department of Computer Science
Dartmouth College
Hanover, NH 03755-3510
{jonh,montague}@cs.dartmouth.edu

December 9, 1998

Abstract

Several projects in the operating systems research community suggest a trend of convergence among features once divided between operating systems and languages. We describe how partial evaluation and transformational programming systems apply to this trend by providing a general framework for application support, from compilation to run-time services. We contend that the community will no longer think of implementing a static collection of services and calling it an operating system; instead, this general framework will allow applications to be flexibly configured, and the "operating system" will simply be *the application support that is supplied at run-time.*

1 Introduction

In ten or fifteen years, we bet, this workshop will become the "Workshop on Hot Topics in Run-Time Application Support." As a community, we used to define an *operating system* as "hardware abstraction and resource management" [SS94, Tan87]. It is the authors' opinion that there are a continuum of tools that provide application support, from compiler to thread scheduler, and that the choice of parts to be in the run-time "operating system" should be made by the application or the user, not by the operating system designer. What we now call the operating system is merely the run-time aspect of the entire application support system.

In the conventional programming model, a program is specified as source code. A compiler evaluates that specification as much as possible statically, and produces an executable image that depends on run time services from the operating system to provide the dynamic features used in the program. Conventional operating systems provide protection, concurrency, hardware abstraction, and simple communication primitives. Conventional languages provide much richer abstraction and communication, but no concurrency, and protection not against adversaries but only against the programmer shooting his own foot.

However, modern operating systems are offering features once most associated with languages, and modern languages are providing features once left to the operating system. We conjecture that the boundary between operating systems and languages is fracturing, and that the term "operating system" is gradually losing any useful, precise definition. The operating system and language domains are converging because the existing boundary between them is only artificial.

In this paper, we first provide evidence of the convergence of operating systems and languages. Then, in Section 3, we discuss a model of compilation based on partial evaluation and transformation that allows applications to compile in assumptions early for better performance, or to defer implementation decisions to allow run-time flexibility. In Section 4, we propose extending the model into the operating system domain, and describe how the extended model sub-

*Jon Howell is supported by a research grant from the USENIX Association. Both authors are graduate students, and would like to be considered for the TCOS award.

†Submitted to the Seventh Workshop on Hot Topics in Operating Systems (HotOS-VII)

sumes the ad-hoc systems described in Section 2. Finally, we mention some problems with our approach, and draw conclusions about how it will apply to the operating systems research community.

2 The convergence of operating systems and languages

In this section, we will examine six ways in which languages are looking more like operating systems, and operating systems are looking more like languages.

2.1 Threads of execution

Applications have increasing needs for internal concurrency, to provide multiple client support in a server, or to implement a user-interface with fast response time. This is evidenced by the wide number of threads packages available for traditional languages, and the introduction of threads directly into language cores, such as in Java [Fla97].

Operating systems, on the other hand, are finding ways to provide application-specific control over scheduling, such as the inheritance scheduling scheme used in Fluke [FS96, FHL+96]. On a parallel machine, an application may be able to completely monopolize several nodes. To exploit this, some parallel operating systems are implemented as application-level library frameworks [MCK91].

2.2 Protection and context switching

Applications are also demanding greater internal protection. Many applications allow macros, plug-ins, or downloaded content. Some, such as Adobe Photoshop or Macromedia FreeHand, have ad-hoc binary extension (plug-in) interfaces. Microsoft's Component Object Model is a generic mechanism for unprotected application extension [OHE96]. Other applications restrict extensions to a limited language environment; examples include Microsoft Word's macros and Netscape Navigator's JavaScript and Java.

An interpreter provides operating-system-like services for the application running inside it. For example, the Agent Tcl system implements protection and resource allocation in an interpreter to allow execution of foreign mobile agents [KGN+97].

Druschel et al. point out that programmers are currently forced to decide at development time which module interfaces to protect (by using context switches, an operating system service), and which to make efficient (by using function calls, a language service). They propose decoupling modularity from protection in their Lipto system by offering a cross-module call that, based on a run-time decision, is implemented either as a fast function call or a protected context switch [DPH92].

Protected Shared Libraries provide an intermediate level of protection between a full context switch and a function call, which allows a shared library to offer services whose implementations are protected from the main application [BTC97].

2.3 Services

Various services have typically been provided either in the operating system or a language library, as implementors have seen fit. In this section, we mention a few cases where services have crossed the "red line" as performance tradeoffs and implementation challenges have changed. For example, while persistence and replication are often implemented as features of a cluster-wide operating system [CLFL94, MWSK94], one can dynamically insert services such as persistence or replication into CORBA objects, using a mechanism not unlike binary rewriting [MCD95].

Conventional operating systems typically provide persistence service in the form of a file system accessible by function calls from the language level. However, languages that offer persistence as a first-class feature (*orthogonal* persistence [MH96]) can make some applications much easier to implement [MCKM96, Jor96, PAD+97]. Unfortunately, orthogonal persistence maps clumsily to a conventional file system, so sometimes orthogonal persistence is provided directly via the operating system [How98, Lie93, DdBF+94].

Sometimes library-like features, such as graphical user-interface components and sound processing services, are packaged into the operating system. Commercial desktop operating systems such as MacOS and Windows NT are notable examples. While Unix provides examples of how to provide such features at user-level, performance concerns motivate operating-system-level implementations.

2.4 Extensibility

Application-specific extensibility is a significant trend in operating systems research. Applications supply kernel extensions to replace inappropriate or inefficient default services. The VINO extensible operating system protects its kernel from errant extensions using software fault isolation [SESS96]. The SPIN kernel relies on language type safety to protect itself from misbehaved extensions [BCE+95, BSP+95, PB96]. The Exokernel achieves extensibility by pushing most services out of the kernel into user level, where applications simply replace default library implementations with their own [EK95, MK97, KEG+97].

The drive toward extensible operating systems can be viewed as trying to make the operating system as flexible as a language/library environment. Features that were once "run-time constants," decided at kernel compilation time, become run-time variable, with alternate values or implementations supplied at run time.

2.5 Profiling and specialization

Many profiling and tracing systems rewrite binaries to insert code, a task that requires clever tricks to fix up addresses that have already been wired down as constant [LS95, SE94].

Seltzer and Small have proposed a profiling and adaptation system for VINO that can simulate policy changes inside the kernel, and install those deemed effective [SS97]. In the Morph system, an operating system extension profiles a running application, then a language-domain code layout tool rewrites the application binary to optimize instruction cache and branch prediction performance [ZWG+97].

Self is a pure, dynamically-typed object-oriented language that requires an aggressive implementation to run efficiently. The run-time environment includes dynamic profiling, specialization, and compilation to discover and exploit any quasi-static features of the code in the system, while maintaining the illusion that the system is dynamically typed. The result is a system with the convenience and functionality of an interpreter, but the speed of a compiled language [HU94, CU91, Höl93].

'C is an extension to C that includes syntax for dynamic code generation. Its compiler `tcc` generates code optimized with respect to run-time constants [EHK96, PEK97]. Consel and Noël demonstrate a general system for efficient run-time specialization in C [CN96].

A hot topic in operating systems is the dynamic specialization of code to optimize performance for current conditions. For example, the Synthesis kernel and its successor Synthetix perform run-time specialization of operating system services for efficiency. Specialization occurs when the system knows that certain variables in a function will, for a certain time, remain constant; this condition is called a *quasi-invariant*. The system recompiles the function, exploiting the new static information at compile time to save run-time instructions. The specialized function is discarded when the quasi-invariant becomes false. An example of the use of specialization in Synthetix is a specialized `read()` function with no concurrency checks for use with files opened exclusively for reading [Mas92, CAK+96, CBK+96, VCMC97, CBK+96].

Franz has also proposed making run-time code generation a central operating system service. In his system, mostly-compiled code is compiled just-in-time when an application is launched, then recompiled for optimization as the program runs [Fra97a].

3 Partial evaluation and transformational programming

In the previous section, we examined evidence that features once provided by languages are being offered by operating systems, and vice-versa. In this section, we look at some systems that attempt to unify the compiler tool chain.

Franz points out that compiling, linking, and loading are aspects of a single problem, and that by compiling to the right machine-independent representation, one can avoid the "fix-up" steps usually associated with dynamic linking [Fra97b]. In fact, observe that in a traditional C tool chain, preprocessing, compiling, linking, and loading tools each manipulate special-purpose file formats, so that the same tools cannot be used to perform similar operations on code at different stages of compilation.

A deeper observation is that at each stage of compilation, each tool does the same basic job, that of partial evaluation: it replaces symbols with values where values are known (constant), and leaves the other symbols (still variable) alone. Some steps include an optimization pass, where the discovery of symbols with constant value leads to the simplification of sections of code. Continuing with our example of the C tool chain, the preprocessor replaces known text strings with macro values, and the compiler replaces known constructs (such as loops and arithmetic) with static code fragments. The static linker replaces references to known functions with the address of the function in the linked executable. The dynamic linker does the same job at run time.

Jones describes beautifully how the simple, central concept of partial evaluation allows one to write abstract, pure code, while still generating efficient executables. Partially evaluating an interpreter with respect to a program results in a compiled version of the program, eliminating the usual interpreter run-time overhead, and allowing the programmer to create efficient code without creating a compiler [Jon96].

Partial evaluation is the guiding concept; transformational systems are a general way to implement specific evaluations. In transformational systems, programs are transformed from abstract, formal representations into efficient executables by repeatedly applying transformations from a collection of rules. Each transformation preserves the correctness of the program while making it more concrete by introducing some implementation decision while rejecting alternative implementations. Some driver directs the application of transformations until the program is completely concrete; that is, executable. Boyle gives a nice introduction to transformational systems, in the context of his TAMPR driver [Boy89]. In CIP-S, a human being is the driver, interactively selecting and applying transformations [Vul94].

Metaprogramming is a form of transformational system, wherein transformations are specified as imperative programs rather than declarative rewrite rules. TXL, or tree transformation language, lets one express transformations as "by-example" modifications to source text in the native language of the program being transformed [CS92]. In intentional programming, transformations are automatically matched against the target program's structure, and exhaustively applied until no transformation's precondition occurs in the program [Sim96, ADK$^+$97].

4 Extending partial evaluation into the kernel

In conventional systems, and to our knowledge in most transformational programming systems, the process of partial evaluation stops once it produces a binary executable. Systems like Synthetix may provide specialized implementations of operating system functions behind the kernel "red line;" extensible systems like VINO and SPIN allow applications to supply such replacements at run-time. But observe two facts: (1) The partial evaluation does not affect the application-kernel interface, and (2) at each stage (including inside the kernel of specializing and extensible kernels), the mechanism and interface to partial evaluation is different and ad-hoc.

We propose to extend the concept behind transformational programming systems to subsume operating system functionality. The artificial boundary we mentioned in Section 1 is the application binary interface (ABI): compilers create binary executables, operating systems load and run those executables. As we mentioned above, transformational systems allow fragments of code to be partially evaluated, either sooner (for performance) or later (for flexibility). This flexibility would be useful even beyond the creation of application binaries, into the realm of tasks we refer to as the operating system software. The ultimate goal of partial evaluation is not a binary file, but a complete computation.

In the following sections, we examine how partial evaluation applies to each of the examples we listed in Section 2.

4.1 Threads of execution

An application programmer could specify the threads of an application in an abstract manner, expressing as much concurrency as she finds appropriate. The actual decision as to whether the threads share an address space, protection domain, or even a node, could be deferred until late in the compilation process. A high-performance web server might resolve the thread abstraction with a transformational module that places all the threads into supervisor mode inside the kernel; on a multi-user machine the same code might be compiled to run each thread in a separate hardware context.

4.2 Protection and context switching

Each choice in the smorgasbord of current protection schemes makes a tradeoff between performance and safety. Instead of designing the protection scheme into each plug-in interface, imagine if plug-in software could be compiled by transformational units to meet an appropriate interface. Trusted plug-ins (shipped with an application, or signed by a trusted signature) would be compiled to fast binary code. Untrusted plug-ins would be compiled to a verifiable language, or have fault isolation checks inserted. We apply the general technique of program transformation to decouple modularity from protection.

4.3 Services

Our examples of inserting services such as persistence and replication into object implementations included one solution that works by rewriting the binary code of methods and others that work by making persistent or replicated the entire environment that objects run inside. In a transformational system, we would expect to see similar approaches, but defined in a more general way, perhaps with better source-language or operating-system portability.

The user interface example concerned whether to put user interface components into the operating system kernel for performance. In a transformational system, that decision (like other protection decisions) can be deferred to system configuration time. Embedded systems or single-user systems might load the user interface alongside other supervisor mode services for speed (saving some context switching); a multiuser system would package it in its own protection domain as is done with the X Windows server.

4.4 Extensibility

Extensibility is a natural consequence of partial evaluation. The programs that serve as the "operating system" in our hypothetical system are programs like any other, and can thus defer certain operations (including linking in code) to run time. Extensibility is expressed by linking new code into parts of the operating system at run time. Safety, or protection, would work as we described above: depending on the system, or perhaps even mixed in a single system, extensions would be subject to a pass through a code verifier, a software fault isolator, or perhaps no check at all.

4.5 Profiling and specialization

The opposite of extensibility is specialization. When a code path is specialized, it is no longer as general and flexible as before, but now has fewer tests and branches, and so executes faster. Specialization is partial evaluation working in the other direction from extensibility. If the transformation system is still available to operating system code at run time, it can be used to recompile a code fragment, treating some variables as quasi-invariants. As in SPIN or Synthetix, guards would still need to be installed to detect when the assumptions used in the specialization fail; perhaps related transformational modules would be able to automatically generate guard code.

In summary, by introducing the general concept of partial evaluation, several ad-hoc systems can be subsumed by a single transformational approach that allows implementation decisions to be made sooner for performance or deferred for flexibility. The conventional tool chain partitions various services (syntactic parsing, variable allocation, collection of code into reusable libraries, persistence, scheduling) into specific tools (compiler, linker, operating system), which sacrifices flexibility: when a given step is performed, it is performed exactly once for the entire application. In our proposal, partial evaluation extends into the operating system, so that tradeoffs between performance and flexibility can be made at all levels without requiring the introduction of ad-hoc interfaces and implementations.

5 Objections

We usually assume that operating systems should be language independent, so they can run a variety of applications, but the system we describe seems to be an integrated system, from language all the way through to run time kernel. In fact, the only central element is the transformation driver or partial evaluator. Programs and transformations can be expressed in a single "wide-spectrum language," in a general abstract syntax tree form, or in a series of intermediate syntaxes. One can still use multiple parser transformations to implement various source languages.

Another concern is the visibility of concrete semantics at the source code level. In our discussion of thread and protection issues, we suggested that code might express as much concurrency as the programmer finds appropriate, with thread and protection domain decisions deferred. The (potential) presence of a protection domain or address space boundary, however, means that the programmer cannot assume that two threads can access the same object in shared memory. Formal languages can hide reference versus copy semantics and other concrete semantics from the programmer. As Boyle describes, the ultimate goal is to express the desired computation in a purely abstract, formal way; then to introduce all implementation decisions (such as the use of references versus copies) as applications of transformations [Boy89].

6 Conclusion

The community used to define the operating system by what was in it: "hardware abstraction and resource management." However, we argue that what comprises an operating system is only "that application support that is provided at run time." In Section 3, we described the partial evaluation model, in which the concept of "compile time" vs. "run time" is extended to a continuum between static and dynamic evaluation; and in Section 4, we extended that model beyond the application binary interface into the realm of traditional operating system functions. Such an extended model would allow services to migrate freely between the "operating system" (run-time environment) and "language" (compile-time environment).

We argue that the boundary between operating systems and languages is fracturing because it is artificial, and that the term "operating system" is gradually losing any useful, precise definition. Current systems break through the artificial boundary by creating ad-hoc mechanisms to introduce flexibility or performance where it was previously unavailable. We posit that a partial evaluation system captures and manifests the continuum from language to operating system in a general way. Systems based on transformational programming will subsume both extensible and specializing operating-system designs.

Acknowledgements

Thanks to our advisors David Kotz and Javed Aslam for tolerating our flights of fancy. We would also like to thank John Chapin for bringing to our attention the literature on transformational programming.

References

[ADK+97] William Aitken, Brian Dickens, Paul Kwiatkowski, Oege de Moor, David Richter, and Charles Simonyi. Transformation in Intentional Programming. Microsoft Research white paper, September 1997. Available at: `http://www.research.microsoft.com/research/ip/overview/TrafoInIP.pdf`.

[BCE+95] Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemysław Pardyak, Stefan Savage, and Emin Gün Sirer. SPIN: An extensible microkernel for application-specific operating system services. *ACM Operating Systems Review*, 29(1):74–77, January 1995.

[Boy89] James M. Boyle. Abstract programming and program transformation. In Ted J. Biggerstaff, editor, *Software Reusability*, chapter 15, pages 361–413. ACM Press, 1989.

[BSP+95] Brian Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, December 1995. ACM Press.

[BTC97] Arindam Banerji, J. M. Tracey, and David L. Cohn. Protected shared libraries — a new approach to modularity and sharing. In *Proceedings of the 1997 USENIX Technical Conference*, pages 59–75, January 1997.

[CAK+96] Crispin Cowan, Tito Autrey, Charles Krasic, Calton Pu, and Jonathan Walpole. Fast Concurrent Dynamic Linking for an Adaptive Operating System. In *International Conference on Configurable Distributed Systems (ICCDS'96)*, Annapolis, MD, May 1996.

[CBK+96] Crispin Cowan, Andrew Black, Charles Krasic, Calton Pu, Jonathan Walpole, Charles Consel, and Eugen-Nicolae Volanschi. Specialization classes: An object framework for specialization. In *Proceedings of the Fifth International Workshop on Object Orientation in Operating Systems*, pages 72–77, Seattle, WA, October 1996. IEEE Computer Society Press.

[CLFL94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single address space operating system. *ACM Transactions on Computer Systems*, pages 271–307, November 1994.

[CN96] Charles Consel and François Noël. A general approach to run-time specialization and its application to C. In *23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburgh Beach, FL, January 1996.

[CS92] J.R. Cordy and M. Shukla. Practical metaprogramming. In *Proceedings of the 1992 IBM Centre for Advanced Studies Conference*, pages 215–224, November 1992.

[CU91] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *Proceedings of the Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–15, October 1991.

[DdBF+94] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindstrom, John Rosenberg, and Francis Vaughan. Grasshopper: An orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, Summer 1994.

[DPH92] Peter Druschel, Larry L. Peterson, and Norman C. Hutchinson. Beyond micro-kernel design: Decouping modularity and protection in Lipto. In *Proceedings of the Twelfth International Conference on Distributed Computer Systems*, pages 512–520, June 1992.

[EHK96] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 131–144, 1996.

[EK95] Dawson R. Engler and M. Frans Kaashoek. Exterminate all operating system abstractions. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS)*, pages 78–83, 1995.

[FHL+96] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 137–151, October 1996.

[Fla97] David Flanagan. *Java in a Nutshell*. O'Reilly & Associates, second edition, 1997.

[Fra97a] M. Franz. Run-time code generation as a central system service. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems (HotOS)*, pages 112–117, 1997.

[Fra97b] Michael Franz. Dynamic linking of software components. *IEEE Computer*, pages 74–81, March 1997.

[FS96] Bryan Ford and Sai Susarla. CPU inheritance scheduling. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 91–105, October 1996.

[Höl93] Urs Hölzle. Integrating independently-developed components in object-oriented languages. In *Proceedings of the Seventh European Conference on Object-Oriented Programming*, pages 36–56, July 1993.

[How98] Jon Howell. Straightforward Java persistence through checkpointing. In *Proceedings of the Third International Workshop on Persistence and Java*, Tiburon, CA, September 1998. Available at: `http://www.sunlabs.com/research/forest/com.sun.labs.pjw3.main.html`.

[HU94] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *SIGPLAN Notices*, volume 29, pages 326–336, June 1994.

[Jon96] Neil D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, September 1996.

[Jor96] Mick Jordan. Early experiences with persistent Java. In *Proceedings of the First International Workshop on Persistence and Java*, September 1996. Available at: `http://www.sunlabs.com/research/forest/UK.Ac.Gla.Dcs.PJW1.pj1.html`.

[KEG+97] M.F. Kaashoek, D.R. Engler, G.R. Ganger, H.M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 52–65, October 1997.

[KGN+97] David Kotz, Robert Gray, Saurab Nog, Daniela Rus, Sumit Chawla, and George Cybenko. Agent Tcl: Targeting the needs of mobile computers. *IEEE Internet Computing*, 1(4):58–67, July/August 1997.

[Lie93] Jochen Liedtke. A persistent system in real use: Experiences of the first 13 years. In *Proceedings of the Third International Workshop on Object Orientation in Operating Systems*, pages 2–11, 1993.

[LS95] James R. Larus and Eric Schnarr. EEL: machine-independent executable editing. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, June 1995.

[Mas92] Henry Massalin. *Synthesis: An efficient implementation of fundamental operating system services*. PhD thesis, Columbia University, 1992.

[MCD95] A. Mohindra, G. Copeland, and M. Devarakonda. Dynamic insertion of object services. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, pages 13–20, 1995.

[MCK91] Peter W. Madany, Roy H. Campbell, and Panos Kougiouris. Experiences building an object-oriented system in C++. Technical Report UIUC-DCS-R-91-1671, The University of Illinois at Urbana-Champaign, March 1991.

[MCKM96] Ron Morrison, Richard Connor, Graham Kirby, and David Munro. Can Java persist? In *Proceedings of the First International Workshop on Persistence and Java*, September 1996. Available at: `http://www.sunlabs.com/research/forest/UK.Ac.Gla.Dcs.PJW1.pj1.html`.

[MH96] J. Eliot B. Moss and Tony L. Hosking. Approaches to adding persistence to Java. In *Proceedings of the First International Workshop on Persistence and Java*, September 1996. Available at: `http://www.sunlabs.com/research/forest/UK.Ac.Gla.Dcs.PJW1.pj1.html`.

[MK97] D. Mazieres and M.F. Kaashoek. Secure applications need flexible operating systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems (HotOS)*, pages 56–61, 1997.

[MWSK94] K. Murray, T. Wilkinson, T. Stiemerling, and P. Kelly. Angel: resource unification in a 64-bit microkernel. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pages 106–115, January 1994.

[OHE96] Robert Orfali, Dan Harkey, and Jeri Edwards. *The essential distributed objects survival guide*. John Wiley & Sons, 1996.

[PAD+97] Tony Printezis, Malcolm Atkinson, Laurent Daynès, Susan Spence, and Pete Bailey. The design of a new persistent object store for PJama. In *Proceedings of the Second International Workshop on Persistence and Java*, August 1997. Available at: `http://www.sunlabs.com/research/forest/COM.Sun.Labs.Forest.PJava.PJW2.pjw2.html`.

[PB96] Przemysław Pardyak and Brian N. Bershad. Dynamic binding for an extensible system. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 201–212, October 1996.

[PEK97] M. Poletto, D.R. Engler, and M.F. Kaashoek. `tcc`: a system for fast, flexible, and high-level dynamic code generation. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 109–121, 1997.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: a system for building customized program analysis tools. Technical Report WRL-94/2, Digital Western Research Laboratory, March 1994.

[SESS96] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 213–227. USENIX Association, October 1996.

[Sim96] Charles Simonyi. Intentional Programming – innovation in the legacy age. In *Proceedings of the IFIP WG 2.1 Meeting*, June 1996. Available at: `http://www.research.microsoft.com/research/ip/ifipwg/ifipwg.htm`.

[SS94] Mukesh Singhal and Niranjan G. Shivaratri. *Advanced concepts in operating systems*. McGraw-Hill, 1994.

[SS97] M. Seltzer and C. Small. Self-monitoring and self-adapting operating systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems (HotOS)*, pages 124–129, 1997.

[Tan87] Andrew S. Tanenbaum. *Operating Systems — Design and Implementation*. Prentice-Hall, 1987.

[VCMC97] Eugen N. Volanschi, Charles Consel, Gilles Muller, and Crispin Cowan. Declarative specialization of object-oriented programs. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 286–300, Atlanta, GA, October 1997.

[Vul94] Ton Vullinghs. Transformational program development using CIP-S. In *Proceedings of the Systems for Computer-Aided Specification, Development and Verification Workshop*, October 1994. Available at: `http://www.informatik.uni-ulm.de/abt/pm/publikationen/Vul94.html`.

[ZWG+97] Xiaolan Zhang, Zheng Wang, N. Gloy, J.B. Chen, and M.D. Smith. System support for automatic profiling and optimization. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 15–26, October 1997.