Hey, You Got Your Compiler in My Operating System!

Jon Howell*
Mark Montague
Dartmouth College Department of Computer Science
Hanover, NH 03755-3510
E-mail: {jonh,montague}@cs.dartmouth.edu

Abstract

Several operating systems projects revolve around moving functionality above or below the kernel "red line" to increase flexibility or performance. We describe how a general model of partial evaluation encompasses this trend. The operating systems community should not be content with a single interface between applications and the operating system, even if that interface allows extension below the red line; we contend that partial evaluation will be most effective when it is free of that arbitrary, static interface. Extending partial evaluation from the language level down to the hardware provides a consistent, global framework for application support.

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" [32, 34]. 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 which tools to apply statically and which to apply at run-time "in the operating system" should not be frozen by the operating system design, but should be fluid, controlled by the dead-code eliminator of a partial evaluator.

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 limited protection (not against adversaries but only against the programmer shooting his own foot).

However, modern operating systems are offering features once only associated with languages, and modern languages (and their compilers) 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 run-time and compile-time domains are converging because the existing boundary between them is artificial.

We propose that the concept of partial evaluation be extended to subsume operating system functionality and organization. The artificial boundary is the application binary interface (ABI): compilers create binary executables conforming to the ABI, operating systems load and run those executables. Partial evaluation allows functions to be made concrete 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 should not be a binary executable, but a complete computation.

Note that we are *not* proposing a single-language system, which would merely move the artifical boundary up to the level of that language. Instead, we argue for the use of partial evaluation as a general mechanism for transforming programs from specifications in (multiple) high-level languages all the way to execution at run-time.

2. Partial Evaluation

To explain our proposal, we first describe the general mechanism of partial evaluation. Partial-evaluation systems start with a very high-level specification of an application and an interpretive description of the language in which the application is specified. That interpreter has two inputs: the

*Jon Howell is supported by a research grant from the USENIX Association.

In Proceedings of the Seventh Workshop on Hot Topics in Operating Systems,
March 29-30, 1999
URL: http://www.cs.dartmouth.edu/~jonh/research/

application itself, and the "run-time" input to the application. The system can effectively compile the application by partially evaluating the interpreter with respect to one of its inputs, the application program. The result is that the general system eliminates unused parts of the interpreter, and optimizes those parts of the interpreter that are used with respect to the program being run. The arbitrary interface to the interpreter (the specification language) disappears at run-time; only those parts of the interface remain that are necessary for the dynamic flexibility expressed in the application.

Partial evaluation can be realized many ways. Transformational systems maintain a pool of manually-written transformation steps, each of which reduces a program from abstract towards concrete by expressing some implementation decision. In CIP-S, for example, a human operator interactively selects and applies transformations [35]. In other systems, transformations include manually-written rules of applicability, and the system applies transformations automatically and exhaustively until the program is reduced to an executable [3, 7, 31].

In general, partial evaluation can be fully automatic. Jones describes a generic partial evaluator that elegantly converts pure, abstract specifications into efficient executable code [16]. One can view his partial evaluator as a fully-automatic transformation system. Transformations are expressed as an interpreter for some language. That interpreter expresses the desired result of a transformation, rather than the transformation itself; partial evaluation effects the automatic transformation.

Partial evaluation is not a foreign concept. Each tool in a typical compiler chain performs partial evaluation; it replaces symbols with values where values are known (constant), and leaves the other symbols (still variable) alone. Consider the C tool chain: The preprocessor replaces known text strings with macro values. 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. A given step is performed exactly once for the entire application. Our complaint is that in conventional systems, each use of partial evaluation is ad-hoc, and is constrained to work between artificial boundaries above and below that limit the opportunity for global system optimization.

3. Transcending the kernel with partial evaluation

In conventional language systems, and in existing transformational programming systems, the process of partial evaluation stops once it produces a binary executable that conforms to an OS application binary interface (ABI). Below the ABI, the so-called "red line," systems like Synthetix specialize implementations of operating system functions, exploiting run-time invariants to expose opportunities for temporary optimizations. Extensible systems like VINO and SPIN allow applications to replace OS functionality with new implementations, making what was once a run-time invariant now variable under application control. But observe two facts: (1) In these systems 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.

Our ideal operating system would be supplied in three parts: The first is a general partial evaluation mechanism. The second is a collection of operating system services (such as CPU scheduling, resource allocation, and device drivers), delivered in a form suitable for partial evaluation. The third is a set of high-level language "compilers," described interpretively in terms of available lower-level components and interfaces. There are still conceptual interfaces between high-level services ("language" features) and low-level services ("operating system" features), but the partial evaluator may remove any trace of those interfaces in the implementation it generates on a particular machine.

In contrast, a conventional OS is supplied as an immutable program (the kernel), some immutable libraries, and a compiler to transform applications from a source language into executables; the running system consists of the kernel and the executables. In our proposed system, the partial evaluator would evaluate an application with respect to an appropriate language description to reduce an application to a more concrete form (analogous to object files); further services would be implemented by reducing the application with respect to implementations of run-time services. A running system would not exhibit the strict, immutable boundaries between kernel and library and application; instead, for example, the scheduler loop might be unrolled to handle the three currently-running processes on the system. Perhaps a program specified in a high-level language is transformed by type-safe transformations, and is run safely in supervisor mode. A matrix application may depend upon code that performs strided disk accesses in terms of block accesses; the partial evaluator would automatically coalesce the operations into a single fast path.

"Compilers" for different languages would be delivered as transformational units that implement the syntax and semantics of a source language in terms of the lower-level services available in the system. Applications with aggressive requirements would provide their own low-level (perhaps non-portable) implementations for some services; extensibility is manifest when the partial evaluator chooses for run-time support the replacement services instead of the system defaults.

4. Application of partial evaluation to operating systems problems

In the following section, we examine how our proposal fits in with several contemporary research topics.

4.1. 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 [30]. The SPIN kernel relies on language type safety to protect itself from misbehaved extensions [2]. 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 [18].

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.

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 to run time. The system is extended when the partial evaluator incorporates new code into the runtime system. Competing protection mechanisms could be available in the same system: extensions would be subject to a pass through a code verifier, a software fault isolator, or perhaps no check at all.

4.2. Profiling and specialization

A hot topic in operating systems and language environments is the dynamic specialization of code to optimize performance for current conditions.

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 by a linker [20, 33]. 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 [29]. 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 [36].

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 [14, 4].

'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 [28]. Consel and Noël demonstrate a general system for efficient run-time specialization in C [6].

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, if only for a 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 [23, 8].

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 [13].

Specialization is partial evaluation working in the other direction from extensibility. 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. Because the partial evaluator is constantly refining the running system, it can recompile a code fragment, treating some variables as quasi-invariants. The problem of discovery and guarding of quasi-invariants is still challenging. 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.

4.3. 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 large number of threads packages available for traditional languages, and the introduction of threads directly into language cores, such as in Java [11].

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 [12]. 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 [22].

With a partial evaluator, 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 processor, could be deferred until late in the compilation process. It could perhaps be deferred even to run-time, with different organizations created for different users of the application. 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.

4.4. 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 [27]. Other applications restrict extensions to a limited language environment; examples include Microsoft Word's macros and Netscape Navigator's JavaScript and Java. The Agent Tcl system implements protection and resource allocation in an interpreter to allow safe execution of foreign mobile agents [19]. Instead of designing the protection scheme into each plug-in interface, imagine if plug-in software could be reduced by partial evaluation 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.

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 [10]. 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 [1]. The general technique of partial evaluation implicitly decouples modularity from protection. Modularity decisions are expressed in the specification of a program; protection decisions are expressed in the concrete implementation created by the partial evaluator.

4.5. 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 [5, 26], one can dynamically insert services such as persistence or replication into CORBA objects, using a mechanism not unlike binary rewriting [24].

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 [25]) can make some applications much easier to implement [17]. Unfortunately, orthogonal persistence maps clumsily to a conventional file system, so sometimes orthogonal persistence is provided directly via the operating system [15, 21, 9].

These examples include 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. With a general partial evaluator, we would expect to see similar approaches, but defined in a more general way, perhaps with better source-language or operating-system portability.

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. With partial evaluation, this 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.

5. 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*. We contend that the boundary between operating systems and compilers is fracturing because it is artificial and restrictive. Current systems break through the artificial boundary by creating ad-hoc mechanisms to introduce flexibility or performance. We assert that a partial evaluation system captures and manifests the continuum from language to operating system in a general way.

We claim that the presence of a general mechanism for partial evaluation would subsume several ad-hoc systems by a single approach that allows implementation decisions to be made concrete for performance or deferred for flexibility. In contrast to the rigid interfaces between tools in the conventional tool chain, our proposal blurs the dichotomy between "compile time" and "run time" into a continuum between static and dynamic evaluation. Such a model allows services to migrate freely between the "language" (compile-time environment) and "operating system" (run-time environment).

Partial evaluation should be extended to subsume operating system functionality and organization. Our proposal (1) offers extensibility unrestricted by an ABI and (2) exposes optimizations that span (indeed, eliminate) the red line.

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 and partial evaluation, as well as the anonymous reviewers whose comments helped reshape this paper.

References

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

[2] B. Bershad, S. Savage, P. Pardyak, E. Gün Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. 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, Dec. 1995. ACM Press.

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

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

[5] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single address space operating system. *ACM Transactions on Computer Systems*, pages 271–307, Nov. 1994.

[6] C. Consel and F. 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.

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

[8] C. Cowan, T. Autrey, C. Krasic, C. Pu, and J. Walpole. Fast Concurrent Dynamic Linking for an Adaptive Operating System. In *International Conference on Configurable Distributed Systems (ICCDS'96)*, Annapolis, MD, May 1996.

[9] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindstrom, J. Rosenberg, and F. Vaughan. Grasshopper: An orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, Summer 1994.

[10] P. Druschel, L. L. Peterson, and N. 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.

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

[12] B. Ford and S. Susarla. CPU inheritance scheduling. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 91–105, Oct. 1996.

[13] 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.

[14] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. *SIGPLAN Notices*, 29(6):326–336, June 1994.

[15] J. Howell. Straightforward Java persistence through checkpointing. In *Advances in Persistent Object Systems*, pages 322–334. Morgan Kaufmann, 1999.

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

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

[18] M. Kaashoek, D. Engler, G. Ganger, H. 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, Oct. 1997.

[19] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla, and G. Cybenko. Agent Tcl: Targeting the needs of mobile computers. *IEEE Internet Computing*, 1(4):58–67, July/August 1997.

[20] J. R. Larus and E. 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.

[21] J. 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.

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

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

[24] 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.

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

[26] 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, Jan. 1994.

[27] R. Orfali, D. Harkey, and J. Edwards. *The essential distributed objects survival guide*. John Wiley & Sons, 1996.

[28] M. Poletto, D. Engler, and M. 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.

[29] 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.

[30] M. I. Seltzer, Y. Endo, C. Small, and K. 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, Oct. 1996.

[31] C. 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`.

[32] M. Singhal and N. G. Shivaratri. *Advanced concepts in operating systems*. McGraw-Hill, 1994.

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

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

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

[36] X. Zhang, Z. Wang, N. Gloy, J. Chen, and M. Smith. System support for automatic profiling and optimization. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 15–26, Oct. 1997.