Achieving High Performance via Co-Designed Virtual Machines

J. E. Smith
Timothy Heil
Dept. of Electrical and Computer Engr.
University of Wisconsin-Madison
{jes,heilt}@ece.wisc.edu

Subramanya Sastry
Todd M. Bezenek
Computer Sciences Dept.
University of Wisconsin-Madison
{sastry,bezenek}@cs.wisc.edu

Abstract

A virtual machine (VM) uses software to support a virtual instruction set architecture on a hardware platform executing a native instruction set. By co-designing the hardware and software elements of a VM, and by using an implementation-dependent native instruction set, there will be many new opportunities for improved performance and flexibility. Because the hardware-supported instruction set is implementation dependent, performance optimizations can be more easily passed from software through to hardware, and performance feedback information can be more easily passed from hardware up to the software. Furthermore, optimizations can be performed by software dynamically, as the program runs. A co-designed virtual machine may include adaptive hardware performance features, continuous hardware performance feedback, and on-the-fly optimizing re-compilation by the VM. Hardware and software can cooperate in finding instruction level parallelism across large blocks of dynamic instructions, and can efficiently implement of a number of advanced microarchitecture techniques involving control independence, prediction, speculation, and cache hierarchy management.

1. Introduction

Today's virtual machines (VMs) use a layer of software that allows programs compiled in one instruction set to be run on a processor executing a (different) native instruction set. VMs have become popular in recent years for providing platform independence; however, VMs also open many new opportunities for enhancing performance. The co-design of VM software and the underlying hardware microarchitecture will enable enhanced instruction level parallelism (ILP) and new adaptable performance mechanisms that cannot be realized when hardware and application software are separated by a fixed instruction set architecture as is conventionally done.

In future high performance computers, we propose that a *virtual instruction set architecture* (V-ISA) will be the level for maintaining architectural compatibility. The V-ISA is implemented with a VM that blends software and hardware in a symbiotic manner via co-design. The hardware supports an implementation-dependent architecture, or *implementation instruction set architecture* (I-ISA). As such, the I-ISA has special features keyed to the specific hardware optimizations built into the microarchitecture. The co-designed VM implementation includes fast compilation of the V-ISA to the I-ISA, efficient hardware performance feedback, optimizing re-compilation, and adaptive hardware performance features, enabling performance improvements beyond those that can be achieved with conventional hardware and software.

Specifically, the software implementing the VM becomes available for the hardware microarchitect to use -- in contrast to the current paradigm where the hardware designer is isolated from software via a fixed, often inflexible instruction set architecture. This application of VMs is also in contrast with most current VM proposals, which focus on maintaining compatibility and portability across platforms supporting existing ISAs, and whose performance objectives are typically no more ambitious than minimizing performance losses that occur in the process.

2. The Search for High Performance

Processor microarchitecture has been evolving for many years. Performance improvements have primarily come from exploiting larger amounts of ILP and from the increased use of prediction and speculation. Microarchitectures have evolved from serial to pipelined to superscalar implementations. For this evolution to continue, it is becoming evident that a substantially different, more far-reaching approach must be used. Some of the current obstacles to increased performance are:

(1) Instruction set compatibility has become a performance obstacle. It is difficult to change an existing instruction set to enhance performance. In fact, it is sometimes inadvisable because changes that enhance performance in one generation may lead to added design complications (with no performance benefits) in future generations, e.g.

delayed branches. Furthermore, removing old instructions is difficult because of the need to maintain compatibility with older generation software.

(2) Designers often make tradeoffs to provide the best performance when averaged over a number of application programs. However, these tradeoffs may not give the best performance for any of the individual programs, or for individual phases of larger programs. The ability to adapt for a specific program or program phase is often limited by decisions made at hardware design time.

(3) Processors have limited visibility to ILP. It is well known that the window of instructions visible to the hardware has to be increased to provide higher ILP. Using hardware alone to increase the instruction window size typically leads to greater complexity, however. For substantial advances to take place, it is likely that the instruction window will have to become larger than hardware alone can support. Potentially, a compiler could make more instructions visible in the window, but software's view is static, and is often restricted. In particular, the object-oriented programming paradigm is becoming widely adopted, and by its nature, the compile-time visible instruction window can be limited due to the presense of dynamically dispatched method calls.

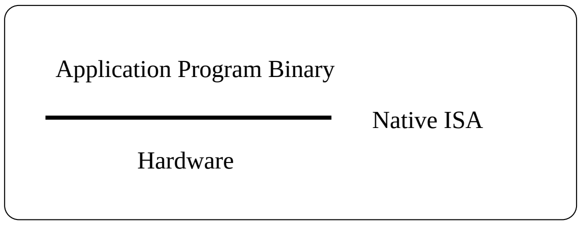
(4) Researchers have become clever in devising hardware performance enhancements. Usually these are targeted at some particular aspect of the design and give a relatively small overall performance improvement. Although each of these features makes sense by itself, any individual program may only be able to take advantage of a few of them. And, implementing them all tends to weigh down a clean pipeline structure with many small, complicated, special-purpose "appendages" -- possibly to such an extent that the overall complexity wipes out the individual benefits.

VM co-design can overcome all of the above obstacles and will open many new opportunities for computer architects. It can provide a wider scope for finding ILP, allow the hardware designer more flexibility in ISA design, allow more dynamic adaptivity in hardware performance features, and allow a cleaner pipeline design by removing hardware appendages (by moving complexity to software where it can be selectively applied). Furthermore, VM co-design can provide high performance for object-oriented programs.

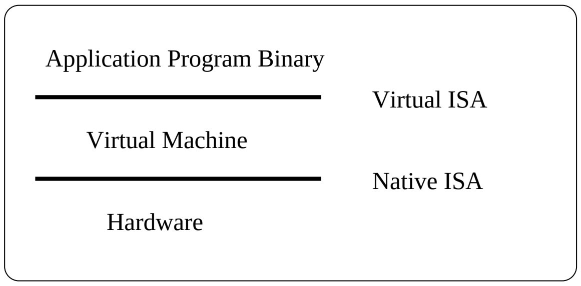
3. Virtual Architectures

Traditional processor architecture is based on an inflexible division between hardware and software (see Figure 1a). The inflexibility of this hardware/software division has become inhibiting to microarchitecture design. Microarchitects support this interface through innovations in hardware, and the ability to reach above the hardware/software interface is very limited. As instruction sets have solidified, opportunities for architectural (instruction set) innovations have become greatly reduced. Only occasionally are current hardware designers able to innovate by modifying instruction sets -- multimedia instruction extensions are a recent example.

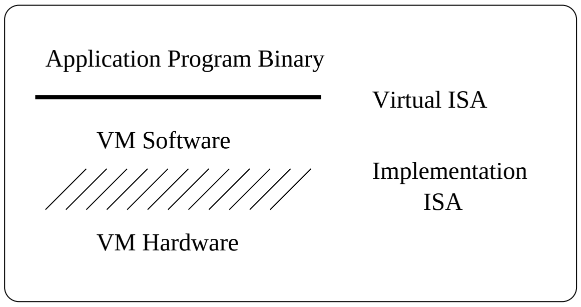
The virtual architecture abstraction provides revolutionary opportunities to computer architects. This



a) Conventional hardware/software interface



b) Conventional virtual machine interface



c) Implementing a virtual architecture with a hardware/software co-designed virtual machine.

Figure 1. Virtual machine co-design.

abstraction has become popular recently in the form of Java bytecodes and Java virtual machine (JVM) implementations. Figure 1b illustrates the VM as it is used in the Java environment. The VM layer effectively widens the traditional ISA interface between hardware and software. Application software is compiled to the virtual architecture specification (i.e. Java bytecodes), and the hardware is an implementation of an existing instruction set, or native ISA. The VM is implemented with an additional software layer that is placed at the hardware/software interface. The VM and the underlying hardware implement the V-ISA through interpretation, just-in-time compilation, adaptive recompilation, or some combination of these techniques.

A primary goal in using VMs in this manner is to provide platform independence. That is, the Java bytecodes can be used on any hardware platform, provided that a VM is implemented for that platform. However, this additional layer of abstraction typically results in lost performance because of inefficiencies in matching the V-ISA and the native ISA via interpretation and just-in-time (JIT) compilation. The V-ISA and the native ISA are defined independently, so there is no real opportunity for the two to mesh well. Consequently, a typical goal in this environment is to provide performance approaching that of a native ISA execution.

Through VM co-design, the V-ISA abstraction can be exploited to exceed native processor performance. This approach is illustrated in Figure 1c. Given a fixed V-ISA (e.g. Java bytecodes), the VM software and underlying hardware are co-designed. The underlying native ISA is implementation dependent, and is *not* an existing ISA as is the case in Figure 1b. We refer to this low-level ISA as the *implementation* ISA (I-ISA) to emphasize its implementation dependence. Because the I-ISA is implementation dependent, the hardware and software along this boundary can be co-designed, blurring the division between the two. The resulting flexibility enables the distribution of performance features between *and across* the hardware/software interface.

In the current environment, Java bytecodes are an obvious choice as a V-ISA, but the V-ISA could, in fact, be a conventional RISC or CISC ISA. Or, if it is desirable to support a conventional "native" ISA and Java bytecodes, both could be supported (or for that matter multiple ISAs could be supported) by the same I-ISA. In this case, the I-ISA may be better suited for some of the V-ISAs than others -- but at least the designer can choose where the tradeoffs should be. And, finally, because platform independence is a good feature, it is still provided at the V-ISA level.

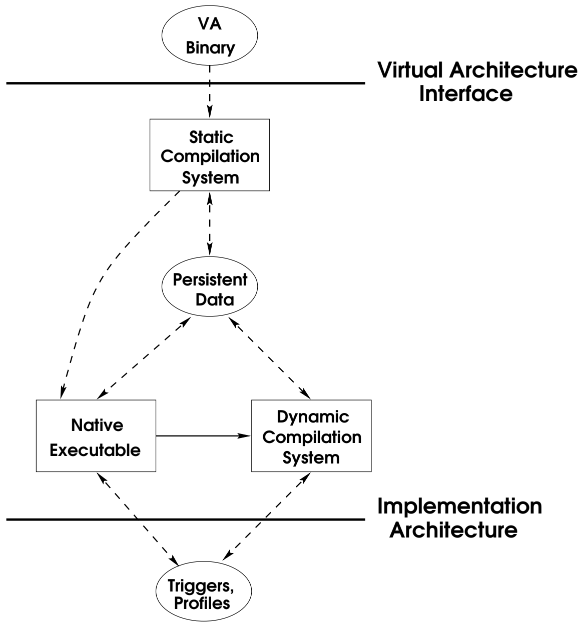

VA Binary
Virtual Architecture Interface
Static Compilation System
Persistent Data
Native Executable
Dynamic Compilation System
Implementation Architecture
Triggers, Profiles

Figure 2. Overview of a co-designed VM system.

4. VM Implementation

Figure 2 illustrates the co-designed virtual machine system that we envision. A V-ISA binary is executed by a combination of VM hardware and software. In this system, progressively more aggressive software optimizations can be phased in. For example, when a V-ISA binary is executed for the first time, it is compiled to the I-ISA on demand by a JIT compiler that is fast and performs simple optimizations. As the program runs, the VM hardware and software work cooperatively to optimize the program's execution. The hardware collects relevant performance information that is in a form efficiently obtained and interpreted by the software; i.e. it is designed to match the software's needs. This component of the optimization is termed *dynamic compilation*.

Using hardware and software cooperatively to optimize performance offers many opportunities for innovation. For example, the VM hardware collects performance information specific to the current VM software optimization implementation. This data can be collected on a per-instruction basis via software selection. Performance information is then passed to the software via instructions that read performance data by polling, by hardware triggers that cause the VM software to be invoked when a certain event occurs, or when some performance characteristic is found to shift outside an operating envelope -- as when a program changes processing phases.

Conversely, the VM software can manage the VM hardware through implementation-dependent instructions (or instruction fields) for specific hardware performance features. These instructions are statically inserted into the I-ISA binary when the V-ISA is compiled, or dynamically, as the I-ISA version of the program executes. For example, given several load/store opcodes to control which memory accesses trigger prefetches and the type of prefetch, the prefetching behavior of the I-ISA binary can be modified as information about data locality is received from the VM hardware. VM software can also directly execute special instructions to tune hardware performance features to match the characteristics of the code currently being executed. For example, an instruction may reach into the hardware and adjust a branch predictor global history length for a particular program or region of a program.

The optimization process could potentially occur many times during the execution of a program. Some of the ways that optimizations are triggered by are (1) the procurement of useful information from the VM hardware, (2) changes in the behavior of the dynamic I-ISA instruction stream, or (3) changes in input data. The modified binary persists during a single program execution, and may persist across program executions. Saving the modified binary between executions will allow object code to continuously improve, and to automatically conform to a new data set or to changes in the system hardware.

5. Example Performance Optimizations

Many current topics of interest in microarchitecture research will benefit from VM co-design. The following subsections focus on a few of them that are of interest to us.

5.1. Trace Selection and Control Independence

Maintaining a large and accurate window of executable dynamic instructions is important for any high-performance microarchitecture. This problem has recently been addressed with the introduction of trace processors [1]. In a trace processor, dynamic instruction supply relies on accurate trace prediction and good trace cache performance.

Trace processors also provide the potential of maintaining a large instruction window in spite of mispredicted branches by exploiting *control independence*. A trace is control independent of a prior branch if the trace is executed regardless of the outcome of the branch. Control independent traces do not have to be squashed and restarted if the branch is mispredicted.

A recent study [2] explores the potential and limitations of control independence in the context of superscalar processors. It predicts that control independence can reduce the performance gap between real and oracle branch prediction by half. In a detailed simulation, overall performance improvements on the order of 30% were observed.

Trace processors offer a complexity-effective solution for implementing control independence mechanisms -- the distributed window organization naturally isolates traces from one another. However, traces must be selected to expose the control independent (re-convergent) points in the program. That is, the hardware trace selection algorithm (heuristics used to divide the dynamic instruction stream into traces) must identify and align traces at these control independent points.

Currently, trace selection is constrained by the limited view of program control flow available to hardware. Hardware is able to expose some control independence, but software can be used to apply sophisticated heuristics to information gathered from a program’s control-flow graph and profile information gathered by the hardware.

VM co-design provides the vehicle for this aggressive trace selection. First, software can implement the otherwise overly-complex analysis. Second, there is a transfer of information between software and hardware, through the I-ISA: hardware provides dynamic profile information, and software supplies the hardware with enough information to select good traces. Co-designed trace selection can likewise be applied to improving trace-prediction accuracy and trace-cache performance.

5.2. Improving Control Prediction with Data Values

Conventional branch predictors use branch history to predict future branch outcomes. Prediction methods have become more elaborate, and clever ways have been found to make predictors more efficient, e.g. by reducing aliasing. However, performance improvements are leveling off.

For substantial improvements in branch prediction, additional information must be used. In particular, certain hard-to-predict branches can be made much more predictable by using data values -- i.e. the results of non-branch instructions. However, folding data values into the branch prediction process in a productive way is very difficult because there is such a large number of data values to choose from, and there are some branches for which using data values hinders predictability.

Using virtual machines is a method for solving this problem. Software can experiment on-the-fly with branch prediction algorithms that use data values. This is similar in spirit to the hardware tuning method for global branch histories in [3]. Furthermore, opcodes in the implementation ISA can be used to enable/disable the use of data values on a per-instruction basis.

5.3. Memory Systems for Object Oriented Applications

The increasing relative delay of memory is a growing performance problem. The poor data locality between objects and the poor instruction locality between methods for different objects aggravates this problem.

Co-designed VMs can be used to improve memory hierarchy performance by reorganizing cached data and for improving cache hierarchy management (i.e. constraining the highest cache where a data item can reside.) These methods can be tightly integrated with features in the performance monitoring architecture to collect performance data on a per-instruction (or per region) basis. Optimizations include the ability to re-organize memory data -- permitted by Java semantics, and the ability to prefetch and otherwise move data in the hierarchy based on type information available in Java bytecodes. Such information can be directly communicated via the I-ISA, by using special opcodes, for example. Additionally, performance feedback from the hardware can allow dynamic runtime optimizations based on usage patterns.

5.4. Performance Tradeoffs

For VM co-design to work, there are some important performance tradeoffs that must be resolved. In particular, the overhead of performing optimizations in software must be offset by performance gains eventually realized. This means that very fast and simple software methods must be used when possible. Expensive optimizations are amortized over many executions of the program by saving the results of dynamic optimizations, creating persistent changes in an application's binary. Costs can also be minimized by using a hierarchical implementation of different algorithms, with more complex, better optimizing algorithms being invoked only for portions of a program that are heavily used.

6. Related Research

The virtual architecture co-design methodology presented here builds on a long history of VM research, as well as related research in other contexts. *Virtual machines* were originally an OS concept. They permitted multiple, possibly different, OSes to coexist on one machine [5], but did not involve a V-ISA or forms of runtime compilation. *Guest OSes* interacted with a *virtual machine monitor* to manipulate system resources. The monitor maintained control of all system resources, and the ISA was crafted to isolate each guest OS from the others [6]. This represents the first in many steps toward virtual architectures.

A pioneering machine was the IBM System/38 [7] which dynamically converted program code into internal micro-code before it was executed. Hiding the details of the micro-code from the virtual architecture provided flexibility for future innovations. The follow-on AS/400 line from IBM provides the same flexibility using binary translation [8]. Application programs are provided as a *program template*. Before a program is run, an AS/400 platform compiles the program template into an executable binary suitable for that platform. The success of this approach was demonstrated by IBMs recent migration of the AS/400 to the PowerPC architecture [9].

Another existing application treats a conventional ISA -- which has traditionally been directly executed by hardware -- as a virtual ISA, in order to provide cross-platform compatibility. FX!32 transparently executes Intel x86 binaries on DEC Alpha processors. Given an Intel x86 binary, FX!32 interprets the x86 operations in order to complete the first execution and provide profile information. An optimizing compiler can then translate the x86 code into high-performance Alpha code, and transparently cache the compiled code on disk. Later invocations of the application will use the optimized code. As profile information accumulates from successive runs, FX!32 re-optimizes the application.

Currently, much virtual machine development is motivated by the desire for platform independence. A number of recent virtual architectures besides Java attempt to satisfy the desire. Oberon [10], Inferno [11], Python [12], ANDF [13, 14], Objective Caml [15, 16] and Oblique [17] are such virtual architectures. Other languages take advantage of these machine-independent platforms by compiling to a virtual architecture. Scheme, in the form of Kawa [18]; and IBM REXX, in the from of NetRexx [19]; obtained machine independence by compiling into Java bytecodes.

Virtual machines have also been a preferred tool of language designers. Early research in language design spawned a number of virtual architectures. Languages such as LISP [20], Smalltalk [21], and SELF [22] gained popularity and attracted considerable attention in the research community. The intent of each was to produce an architecture with improved safety, security, reusability, and extensibility. But these goals are difficult to obtain with a low-level architecture designed for direct execution. The resulting semantics and structure of these new languages made them difficult to compile efficiently; interpreted execution was natural. This quickly led to the concept of a virtual architecture executed by a virtual machine. Platform independence was an added advantage.

The poor performance of interpreters spurred research into making virtual machines faster. Dynamic compilers built into VMs translated from the virtual ISA to the native ISA and optimized the resulting code. A VM for SELF included multiple compilers for different levels of optimization [23]. Guided by profile information, the VM frequently executed code for heavy optimization, performing runtime adaptive optimizations such as inlining of dynamically dispatched function calls. This is much the same as the dynamic runtime optimizations employed in the contemporary virtual architecture model.

Specialized hardware has also been used to improve performance of virtual machines. The virtual instruction set can be compiled into an instruction set that is easily executed by a RISC processor. In addition, various extensions to a RISC ISA can be used to improve the performance of operations commonly used within the virtual architecture. SOAR [24] and Mushroom [25] are excellent examples of this approach. These processors use special hardware to support type checking with tagged memory, garbage collection with software managed caches, object-oriented heap management with object-oriented memory hierarchies, and function calls with register windows. These specialized-hardware systems provide early examples of using hardware/software co-design. The virtual machine compiled the virtual ISA into a RISC ISA with direct support for virtual ISA features. Tuning the virtual machine compiler and the RISC ISA together allowed the implementation of performance enhancements that would not have been possible without using the co-design method.

Research has been done which supports our conjecture that virtual architectures can be used to improve performance. The DAISY (Dynamically Architected Instruction Set from Yorktown) project at IBM's T.J. Watson Research Center articulates the microarchitect's desire to improve performance using innovations incompatible with a standard ISA [26]. The goal of DAISY is to enhance instruction-level parallelism using VLIW architectures that are incompatible with the architecture being addressed. DAISY provides simple hardware support for completely emulating other architectures including x86, S/390, PowerPC, and Java [27].

DAISY contains a kernel -- unseen by the virtual architecture -- which translates instructions from the virtual ISA to the VLIW architecture. V-ISA code is translated in virtual-memory-page-sized blocks, and is triggered when program execution first enters a page. The translation is cached in a separate region of memory. The addition of aggressive VLIW scheduling provides improved performance.

All aspects of the emulated architecture are preserved, including precise exceptions and self-modifying code semantics; even operating system kernel code is emulated. Hardware support is provided for unusual aspects of the virtual architectures, such as different floating point formats, status and comparison flags, and the ability to access subsections of a register.

Many of the advantages of our virtual architecture model are found in DAISY. For instance, DAISY uses a dynamic optimizing VM co-designed with hardware supporting special I-ISA features added to improve performance. We extend the advantages of DAISY with more adaptive, dynamic hardware optimizations and greater reliance on hardware performance monitoring. Further, our focus on a higher-level V-ISA enables better global compiler optimization.

Contemporary compiler research is also moving toward the virtual architecture model. Many advanced compiler optimizations rely on information not available at compile time [28]. These include link-time optimizations [29], post-link-time optimizations [30], and run-time optimizations [31, 32, 33, 34, 35, 36]. Profiling is often advocated as a way to provide this run-time information to the compiler. Profiling can increase the performance gain or reduce the code expansion of optimizations [37, 38, 39, 40, 41, 42, 43]. Virtual machines naturally provide this information to the compiler. When the compiler is integrated within a VM, profile information specific to the particular execution being optimized is provided. This allows our virtual architecture model to adjust the binary for each execution of the program -- whether it is different because it has different input data, resides in different physical memory pages, is executing with a different set of other processes, or because a processor upgrade recently took place.

Finally, research in hardware assisted profiling [44, 45, 46] will aid the dynamic optimizations within a virtual machine. Hardware gathers detailed statistics down to the level of specific instructions with very little overhead. By taking advantage of the co-design model, the VM will retrieve this information from the CPU through software readable hardware registers and traps.

7. Summary

Using hardware/software co-design to implement a virtual machine will provide a powerful new way of dynamically optimizing executing programs. By exploiting observed run-time performance characteristics, optimizations can improve the execution of code using dynamic compilation, improve the efficiency of the memory system by changing the data or instruction reference behavior of the program, or make adjustments to underlying hardware performance mechanisms. These optimizations will be done on a broad scale; not limited by a relatively small issue window as in hardware-only approaches. Higher performance will result from combining the strengths of software compilation and dynamic hardware execution.

Acknowledgements

This work was supported by NSF Grant MIP-9505853, by an IBM Partnership Award, and by Sun Microsystems.

References

[1] E. Rotenberg, Q. A. Jacobson, Yiannakis Sazeides, J. E. Smith, " Trace Processors," *Thirtieth Intl. Symp. on Microarch.*, pp. 138-148, December 1997.

[2] E. Rotenberg, Q. A. Jacobson, J. E. Smith, "A Study of Control Independence in Superscalar Processors," *Fifth Intl. Symp. on High Perf. Comp. Arch.*, pp. 115-124, January 1999.

[3] Toni Juan, Sanji Sanjeevan, Juan J. Navarro, "Dynamic History-Length Fitting: A third level of adaptivity for branch prediction," *Fifth Intl. Symp. on Comp. Arch.*, pp. 155-166, June 1998.

[4] David A. Solomon, "The Windows NT Kernel Architecture," *IEEE Computer* 31(10), pp. 40-47, October 1998.

[5] R. A. Meyer, L. H. Seawright, "A virtual machine time-sharing system," *IBM System Journal* 9(3), pp. 199-217, 1970.

[6] Gerald J. Popek, Robert P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Comm. of the ACM*, 17(7), pp. 412-421, July 1974.

[7] Viktors Berstis, "Security and Protection of Data in the IBM System/38," *Seventh Int. Symp. on Comp. Arch.*, pp. 245-252, May 1980.

[8] Frank G. Soltis, Paul Conte, *Inside the AS/400: Featuring the AS/400E Series*, 2nd Ed., November 1997.

[9] Brian E. Clark, "64 Bits, No Buts -- Implementing a True 64-Bit Computer System," presentation, Engineering Hall, Univ. of Wisconsin at Madison, March 1997.

[10] M. Franz, T. Kistler, "Slim Binaries," Technical Report No. 96-24, Dept. of Info. and Comp. Sci., Univ. of California, Irvine, June 1996.

[11] Lucent Technologies, *Inferno Release 2.0 Reference Manual*, 1997.

[12] Jeff Bauer, "An Introduction to Python," *Linux Journal* #21, January 1996.

[13] Christian Fabre, Francois de Ferriere, Fred Roy, "Java-ANDF Feasibility Study," Final Report, Open Software Foundation Research Institute, March 1997.

[14] Gianluigi Castelli, proj. dir., "ESPRITE Project 6062," Project description OMI/GLUE/R/01-10-92, January 1992.

[15] Didier Remy, Jerome Vouillon, "Objective ML: An effective object-oriented extension to ML," *Theory And Practice of Objects Systems*, 4(1), pp. 27-50, 1998.

[16] Xavier Leroy, "The ZINC experiment, an economical implementation of the ML language," INRIA Technical Report 117, 1990.

[17] Luca Cardelli, "A Language with Distributed Scope," DEC SRC Technical Report, May 1995.

[18] Per Bothner, "Kawa -- Compiling Dynamic Languages to the Java VM," *USENIX Annual Tech. Conf.*, June 1998.

[19] Peter Heuchert, Frederik Haesbrouck, Norio Furukawa, Ueli Wahil, "Creating Java Applications using NetRexx," IBM ITSO Redbook SG24-2216-00, September 1997.

[20] Guy Steele Jr., Richard Gabriel, "The Evolution of Lisp," *SIGPLAN Notices* 28(3), pp. 231-270, March 1993.

[21] Adele Goldberg, David Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.

[22] David Ungar, Randall B. Smith, "SELF: The Power of Simplicity," *Proc. of Obj. Oriented Sys. and Prog. Lang.*, pp. 227-241, 1987.

[23] Urs Holzle, "Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming," Sun Microsystems Laboratories Technical Report TR-95-35, 1995.

[24] David Ungar, Ricki Blau, Peter Foley, Dain Samples, David Patterson, "Architecture of SOAR: Smalltalk on a RISC," *Eleventh Intl. Symp. on Comp. Arch.*, pp. 188-197, June 1984.

[25] Mario Wolczko, Ifor Williams, "The influence of the object-oriented language model on a supporting architecture," *Twenty-sixth Hawaii Intl. Conf. on Sys. Sci.*, January 1993.

[26] Kemal Ebcioglu, Erik R. Altman, "DAISY: Dynamic Compilation for 100% Architecture Compatibility," IBM Research Report RC 20538, August 1996.

[27] Kemal Ebcioglu, Erik R. Altman, Erdem Hokenek, "A Java Processor Based on Fast Dynamic VLIW Compilation," *Intl. Workshop On Security and Efficiency Aspects of Java*, January 1997.

[28] Sarita V. Adve, Doug Burger, Rudolf Eigenmann, Alasdair Rawsthorne, Michael D. Smith, Catherine H. Gebotys, Mahmut T. Kandemir, David J. Lija, Alok N. Choudhary, Jesse Z. Fang, Pen-Chung Yew, "Changing Interaction of Compiler and Architecture," *IEEE Computer* 30(12), pp. 51-58, December 1997.

[29] Amir H. Hashemi, David R. Kaeli, Brad Calder, "Efficient Procedure Mapping Using Cache Line Coloring," *Prog. Lang. Design and Impl.*, pp. 171-182, June 1997.

[30] David. W. Goodwin, "Interprocedural Dataflow Analysis in an Executable Optimizer," *Prog. Lang. Design and Impl.*, pp. 122-133, June 1997.

[31] Joseph A. Fisher, "Walk-Time Techniques: Catalyst for Architectural Change," *IEEE Computer* 30(9), pp. 40-42., September 1997.

[32] Todd B. Knoblock, Erik Ruf, "Data Specialization," *Prog. Lang. Design and Impl.*, pp. 215-226, May 1996.

[33] Mark Leone, Peter Lee, "A Declarative Approach to Run-Time Code Generation," *Workshop on Comp. Support for Sys. Soft.*, pp. 8-17, February 1996.

[34] Charles Consel, Luke Hornof, Francois Noel, Jacques Noye, Nicolae Volanschi, "A Uniform Approach for Compile-time and Run-time Specialization," INRIA Research Report RR-2775, January 1996.

[35] Massimiliano Poletto, Dawson R. Engler, M. Frans Kaashoek, "tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation," *Prog. Lang. Design and Impl.*, pp. 109-121, June 1997.

[36] Pedro C. Diniz, Martin C. Rinard, "Dynamic feedback: an effective technique for adaptive computing," *Prog. Lang. Design and Impl.*, pp. 71-84, June 1997.

[37] Robert G. Burger, R. Kent Dybvig, "An Infrastructure for Profile-Driven Dynamic Recompilation," *Intl. Conf. on Comp. Lang.*, pp. 240-249, May 1998.

[38] Glenn Ammons, James R. Larus, "Improving Dataflow Analysis with Path Profiles," *Prog. Lang. Design and Impl.*, pp. 72-84, June 1998.

[39] Raymond Lod, Fred Chow, Robert Kennedy, Shin-Ming Liu, Peng Tu, "Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores," *Prog. Lang. Design and Impl.*, pp. 26-37, June 1998.

[40] A. V. S. Sastry, Roy D. C. Ju, "A New Algorithm for Scalar Promotion Based on SSA Form," *Prog. Lang. Design and Impl.*, pp. 15-25, June 1998.

[41] Cliff Young, David S. Johnson, David R. Karger, Michael D. Smith, "Near-optimal Intraprocedural Branch Alignment," *Prog. Lang. Design and Impl.*, pp. 183-193, June 1997.

[42] Rastislav Bodik, Rajiv Gupta, Mary Lou Soffa, "Complete Removal of Redundant Expressions," *Prog. Lang. Design and Impl.*, pp. 1-14, June 1998.

[43] Andrew Ayers, Robert Gottlieb, Richard Schooler, "Aggressive Inlining," *Prog. Lang. Design and Impl.*, pp. 134-145, June 1997.

[44] Glenn Ammons, Thomas Ball, James R. Larus, "Exploiting Hardware Performance Counters With Flow and Context Sensitive Profiling," *Prog. Lang. Design and Impl.*, pp. 85-96, June 1997.

[45] Jeffrey Dean, James E. Hicks, Carl A Waldspurger, William E. Weihl, George Chrysos, "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors," *Thirtieth Symp. on Microarch.*, pp. 292-302, December 1997.

[46] Thomas M. Conte, Kishore N. Menezes, Mary Ann Hirsch, "Accurate and Practical Profile-Driven Compilation Using the Profile Buffer," *Twenty-ninth Symp. on Microarch.*, pp. 36-45, December 1996.