A Brief History of Just-In-Time

John Aycock
Department of Computer Science
University of Calgary

Software systems have been using "just-in-time" compilation (JIT) techniques since the 1960s. Broadly, JIT compilation includes any translation performed dynamically, after a program has started execution. We examine the motivation behind JIT compilation, constraints imposed on JIT compilation systems, and present a classification scheme for such systems. This classification emerges as we survey forty years of JIT work, from 1960–2000.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors; K.2 [**Computing Milieux**]: History of Computing—*Software*

General Terms: Languages, Performance

Additional Key Words and Phrases: Just-in-time compilation, dynamic compilation

1. INTRODUCTION

> Those who cannot remember the past are condemned to repeat it.
> *George Santayana, 1863–1952* [Bartlett 1992]

This oft-quoted line is all too applicable in computer science. Ideas are generated, explored, set aside — only to be reinvented years later. Such is the case with what is now called "just-in-time" or dynamic compilation, which refers to translation that occurs after a program begins execution.

Strictly speaking, JIT compilation systems ("JIT systems" for short) are completely unnecessary. They are only a means to improve the time and space efficiency of programs. After all, the central problem JIT systems address is a solved one: translating programming languages into a form that is executable on a target platform.

What is translated? The scope and nature of programming languages that require translation into executable form covers a wide spectrum. Traditional programming languages like Ada, C, and Java are included, as well as little languages [Bentley 1988] such as regular expressions.

Traditionally, there are two approaches to translation: compilation and interpretation. Compilation translates one language into another — C to assembly language, for example — with the implication that the translated form will be

This work was supported in part by a grant from the National Science and Engineering Research Council of Canada.
Name: John Aycock
Affiliation: Department of Computer Science, University of Calgary
Address: 2500 University Drive N.W., Calgary, Alberta, Canada T2N 1N4
Address: EMAIL: aycock@cpsc.ucalgary.ca

more amenable to later execution, possibly after further compilation stages. Interpretation eliminates these intermediate steps, performing the same analyses as compilation, but performing execution immediately.

JIT compilation is used to gain the benefits of both (static) compilation and interpretation. These benefits will be brought out in later sections, so we only summarize them here:

—Compiled programs run faster, especially if they are compiled into a form which is directly executable on the underlying hardware. Static compilation can also devote an arbitrary amount of time to program analysis and optimization. This brings us to the primary constraint on JIT systems: speed. A JIT system must not cause untoward pauses in normal program execution as a result of its operation.

—Interpreted programs are typically smaller, if only because the representation chosen is at a higher level than machine code, and can carry much more semantic information implicitly.

—Interpreted programs tend to be more portable. Assuming a machine-independent representation, such as high-level source code or virtual machine code, only the interpreter need be supplied to run the program on a different machine. (Of course, the program still may be doing nonportable operations, but that's a different matter.)

—Interpreters have access to run-time information, such as input parameters, that may be undecidable using static analysis.

To narrow our focus somewhat, we only examine software-based JIT systems that have a nontrivial translation aspect. Keppel, Eggers, and Henry eloquently build an argument for the more general case of run-time code generation, where this latter restriction is removed [Keppel et al. 1991].

Note that we use the term "execution" in a broad sense — we call a program representation executable if it can be executed by the JIT system in any manner, either directly as in machine code, or indirectly using an interpreter.

2. JIT COMPILATION TECHNIQUES

Work on JIT compilation techniques often focuses around implementation of a particular programming language. We have followed this same division in this section, ordering from earliest to latest where possible.

2.1 Genesis

Self-modifying code has existed since the earliest days of computing, but we exclude that from consideration because there is typically no compilation or translation aspect involved.

Instead, we suspect that the earliest published work on JIT compilation is McCarthy's 1960 LISP paper. He mentions compilation of functions into machine language, a process fast enough that the compiler's output needn't be saved [McCarthy 1960]. This can be seen as an inevitable result of having programs and data share the same notation [McCarthy 1981].

Another early published reference to JIT compilation dates back to 1966. The University of Michigan Executive System for the IBM 7090 explicitly notes that the assembler [University of Michigan 1966b, page 1] and loader [University of Michigan 1966a, page 6] can be used to translate and load during execution. (The manual's preface says that most sections were written before August 1965, so this likely dates back further.)

Thompson's 1968 *CACM* paper is frequently cited as "early work" in modern publications. He compiles regular expressions into IBM 7094 code in an ad hoc fashion, code which is then executed to perform matching [Thompson 1968].

2.2 LC^2

The Language for Conversational Computing, or LC^2, was a language designed for interactive programming [Mitchell et al. 1968]. Although used briefly at Carnegie-Mellon University for teaching, LC^2 was primarily an experimental language [Mitchell 2000]. It might otherwise be consigned to the dustbin of history, if not for the techniques used by Mitchell in its implementation [Mitchell 1970], techniques that later influenced JIT systems for Smalltalk and Self.

Mitchell observed that compiled code can be derived from an interpreter at run-time, simply by storing the actions performed during interpretation. This only works for code that has been executed, however — he gives the example of an if-then-else statement, where only the else-part is executed. To handle such cases, code was generated for the unexecuted part which re-invoked the interpreter should it ever be executed (the then-part, in the example above).

2.3 APL

The seminal work on efficient APL implementation is Abrams' dissertation [Abrams 1970]. Abrams concocted two key APL optimization strategies, which he described using the connotative terms "drag-along" and "beating." Drag-along defers expression evaluation as long as possible, gathering context information in the hopes that a more efficient evaluation method might become apparent; this might now be called lazy evaluation. Beating is the transformation of code to reduce the amount of data manipulation involved during expression evaluation.

Drag-along and beating relate to JIT compilation because APL is a very dynamic language; types and attributes of data objects are not, in general, known until run-time. To fully realize these optimizations' potential, their application must be delayed until run-time information is available.

Abrams' "APL Machine" employed two separate JIT compilers. The first translated APL programs into postfix code for a D-machine[1], which maintained a buffer of deferred instructions. The D-machine acted as an 'algebraically simplifying compiler' [Abrams 1970, page 84] which would perform drag-along and beating at run-time, invoking an E-machine to execute the buffered instructions when necessary.

Abrams' work was directed towards an architecture for efficient support of APL, hardware support for high-level languages being a popular pursuit of the time. Abrams never built the machine, however; an implementation was attempted a few years later [Schroeder and Vaughn 1973].[2] The techniques were later expanded upon by others [Miller 1977], although the basic JIT nature never changed, and were used

[1] Presumably "D" stood for "Deferral" or "Drag-Along."

[2] In the end, Litton never built the machine [Mauriello 2000].

slower	⟵		⟶	*faster*
	interpreted source code	interpreted virtual machine code	native code	
smaller	⟵		⟶	*larger*

Fig. 1. The time-space tradeoff.

for the software implementation of Hewlett-Packard's APL\3000 [Johnston 1977; van Dyke 1977].

2.4 Mixed Code, Throw-Away Code, and BASIC

The tradeoff between execution time and space often underlies the argument for JIT compilation. This tradeoff is summarized in Figure 1. The other consideration is that most programs spend the majority of time executing a minority of code, based on data from empirical studies [Knuth 1971]. Two ways to reconcile these observations have appeared: mixed code and throw-away compiling.

"Mixed code" refers to the implementation of a program as a mixture of native code and interpreted code, proposed independently by [Dakin and Poole 1973] and [Dawson 1973]. The frequently-executed parts of the program would be in native code, the infrequently-executed parts interpreted, hopefully yielding a smaller memory footprint with little or no impact on speed. A fine-grained mixture is implied: implementing the program with interpreted code and the libraries with native code would *not* constitute mixed code.

A further twist to the mixed code approach involved customizing the interpreter [Pittman 1987]. Instead of mixing native code into the program, the native code manifests itself as special virtual machine instructions; the program is then compiled entirely into virtual machine code.

The basic idea of mixed code, switching between different types of executable code, is still applicable to JIT systems, although few researchers at the time advocated generating the machine code at run-time. Keeping both a compiler and an interpreter in memory at run-time may have been considered too costly on the machines of the day, negating any program size tradeoff.

The case against mixed code comes from software engineering [Brown 1976]. Even assuming that the majority of code will be shared between the interpreter and compiler, there are still two disparate pieces of code (the interpreter proper and the compiler's code generator) which must be maintained and exhibit identical behavior.

(Proponents of partial evaluation, or program specialization, will note that this is a specious argument in some sense, because a compiler can be thought of as a specialized interpreter [Jones et al. 1993]. However, the use of partial evaluation techniques is not currently widespread.)

This brings us to the second manner of reconciliation: throw-away compiling [Brown 1976]. This was presented purely as a space optimization: instead of static compilation, parts of a program could be compiled dynamically on an as-needed basis. Upon exhausting memory, some or all of the compiled code could be thrown away; the code would be regenerated later if necessary.

BASIC was the testbed for throw-away compilation. Brown essentially characterized the technique as a good way to address the time-space tradeoff [Brown 1976];

Hammond was somewhat more adamant, claiming throw-away compilation to be superior except when memory is tight [Hammond 1977].

A good discussion of mixed code and throw-away compiling may be found in [Brown 1990].

2.5 FORTRAN

Some of the first work on JIT systems where programs automatically optimize their "hot spots" at run-time is due to Hansen [Hansen 1974].[3] He addressed three important questions:

(1) What code should be optimized? Hansen chose a simple, low-cost frequency model, maintaining a frequency-of-execution counter for each block of code (we use the generic term "block" to describe a unit of code; the exact nature of a block is immaterial for our purposes).

(2) When should the code be optimized? The frequency counters served a second rôle: crossing a threshold value made the associated block of code a candidate for the next "level" of optimization, as described below. "Supervisor" code was invoked between blocks, which would assess the counters, perform optimization if necessary, and transfer control to the next block of code. The latter operation could be a direct call, or interpreter invocation — mixed code was supported by Hansen's design.

(3) How should the code be optimized? A set of conventional machine-independent and machine-dependent optimizations were chosen and ordered, so a block might first be optimized by constant folding, by common subexpression elimination the second time optimization occurs, by code motion the third time, and so on. Hansen observes that this scheme limits the amount of time taken at any given optimization point (especially important if the frequency model proves to be incorrect), as well as allowing optimizations to be incrementally added to the compiler.

Programs using the resulting Adaptive FORTRAN system reportedly were not always faster than their statically compiled-and-optimized counterparts, but performed better overall.

Returning again to mixed code, Ng and Cantoni implemented a variant of FORTRAN using this technique — in a sense [Ng and Cantoni 1976]. Their system could compile functions at run-time into "pseudo-instructions," probably a tokenized form of the source code rather than a lower-level virtual machine code. The pseudo-instructions would then be interpreted. They claimed that run-time compilation was useful for some applications and avoided a slow compile-link process. They did not produce mixed code at run-time; their use of the term referred to the ability to have statically-compiled FORTRAN programs call their pseudo-instruction interpreter automatically when needed via linker trickery.

[3] [Dawson 1973] mentions a 1967 report by Barbieri and Morrissey where a program begins execution in interpreted form, and frequently-executed parts 'can be converted to machine code.' However, it is not clear if the conversion to machine code occurred at run-time. Unfortunately, we have not been able to obtain the cited work as of this writing.

2.6 Smalltalk

Smalltalk source code is compiled into virtual machine code when new methods are added to a class [Goldberg and Robson 1985]. The performance of naïve Smalltalk implementations left something to be desired, however.

Rather than attack the performance problem with hardware, Deutsch and Schiffman made key optimizations in software. The observation behind this was that they could pick the most efficient representation for information, so long as conversion between representations happened automatically and transparently to the user [Deutsch and Schiffman 1984].

JIT conversion of virtual machine code to native code was one of the optimization techniques they used, a process they likened to macro-expansion. Procedures were compiled to native code lazily, when execution entered the procedure; the native code was cached for later use. Their system was linked to memory management in that native code would never be paged out, just thrown away and regenerated later if necessary.

In turn, Deutsch and Schiffman credit the dynamic translation idea to Rau [Rau 1978]. Rau was concerned with "universal host machines" which would execute a variety of high-level languages well (compared to, say, a specialized APL machine). He proposed dynamic translation to microcode at the granularity of single virtual machine instructions. A hardware cache, the dynamic translation buffer, would store completed translations; a cache miss would signify a missing translation, and fault to a dynamic translation routine.

2.7 Self

The Self programming language [Ungar and Smith 1987; Smith and Ungar 1995], in contrast to many of the other languages mentioned in this section, is primarily a research vehicle. Self is in many ways influenced by Smalltalk, in that both are pure object-oriented languages — everything is an object. But Self eschews classes in favor of prototypes, and otherwise attempts to unify a number of concepts. Every action is dynamic and changeable, and even basic operations, like local variable access, require invocation of a method. To further complicate matters, Self is a dynamically-typed language, meaning that the types of identifiers are not known until run-time.

Self's unusual design makes efficient implementation difficult. This resulted in the development of the most aggressive, ambitious JIT compilation and optimization up to that time. The Self group noted three distinct generations of compiler [Hölzle 1994], an organization we follow below; in all cases, the compiler was invoked dynamically upon a method's invocation, as in Deutsch and Schiffman's Smalltalk system.

2.7.1 *First Generation.* Almost all the optimization techniques employed by Self compilers dealt with type information, and transforming a program in such a way that some certainty could be had about the type of identifiers. Only a few techniques had a direct relationship with JIT compilation, however.

Chief among these, in the first generation Self compiler, was customization [Chambers et al. 1989; Chambers and Ungar 1989; Chambers 1992]. Instead of dynamically compiling a method into native code that would work for any invocation of the method, the compiler produced a version of the method that was customized to that particular context. Much more type information was available to the JIT compiler compared to static compilation, and by exploiting this fact the resulting code was much more efficient. While method calls from similar contexts could share customized code, "overcustomization" could still consume a lot of memory at run-time; ways to combat this problem were later studied [Dieckmann and Hölzle 1997].

2.7.2 *Second Generation.* The second generation Self compiler extended one of the program transformation techniques used by its predecessor, and computed much better type information for loops [Chambers and Ungar 1990; Chambers 1992].

This Self compiler's output was indeed faster than that of the first generation, but it came at a price. The compiler ran 15 to 35 times more slowly on benchmarks [Chambers and Ungar 1990; Chambers and Ungar 1991], to the point where many users refused to use the new compiler! [Hölzle 1994]

Modifications were made to the responsible algorithms to speed up compilation [Chambers and Ungar 1991]. One such modification was called "deferred compilation of uncommon cases."[4] The compiler is informed that certain events, such as arithmetic overflow, are unlikely to occur. That being the case, no code is generated for these uncommon cases; a stub is left in the code instead, which will invoke the compiler again if necessary. The practical result of this is that the code for uncommon cases need not be analyzed upon initial compilation, saving a substantial amount of time.[5]

Ungar, Smith, Chambers, and Hölzle [1992] contains a good presentation of optimization techniques used in Self and the resulting performance in the first and second generation compilers.

2.7.3 *Third Generation.* The third generation Self compiler attacked the issue of slow compilation at a much more fundamental level. The Self compiler was part of an interactive, graphical programming environment; executing the compiler on-the-fly resulted in a noticeable pause in execution. Hölzle argued that measuring pauses in execution for JIT compilation by timing the amount of time the compiler took to run was deceptive, and not representative of the user's experience [Hölzle 1994; Hölzle and Ungar 1994b]. Two invocations of the compiler could be separated by a brief spurt of program execution, but would be perceived as one long pause by the user. Hölzle compensated by considering temporally-related groups of pauses, or 'pause clusters,' rather than individual compilation pauses.

As for the compiler itself, compilation time was reduced — or at least spread out — by using adaptive optimization, similar to Hansen's FORTRAN work. Initial method compilation was performed by a fast, non-optimizing compiler; frequency-of-invocation counters were kept for each method to determine when recompilation should occur [Hölzle 1994; Hölzle and Ungar 1994b; Hölzle and Ungar 1994a]. They

[4]In Chambers' thesis, this is referred to as "lazy compilation of uncommon branches," an idea he attributes to a suggestion by John Maloney in 1989 [Chambers 1992, page 123]. However, this is the same technique used in [Mitchell 1970], albeit for different reasons.

[5]This technique can be applied to dynamic compilation of exception handling code [Lee et al. 2000].

make an interesting comment on this mechanism:

> ...in the course of our experiments we discovered that the trigger mechanism ("when") is much less important for good recompilation results than the selection mechanism ("what"). [Hölzle 1994, page 38][6]

This may come from the slightly counter-intuitive notion that the best candidate for recompilation is *not* necessarily the method whose counter triggered the recompilation. Object-oriented programming style tends to encourage short methods; a better choice may be to (re)optimize the method's caller and incorporate the frequently-invoked method inline [Hölzle and Ungar 1994b].

Adaptive optimization adds the complication that a modified method may already be executing, and have information (such as an activation record on the stack) that depends on the previous version of the modified method [Hölzle 1994]; this must be taken into consideration.[7]

The Self compiler's JIT optimization was assisted by the introduction of "type feedback" [Hölzle 1994; Hölzle and Ungar 1994a]. As a program executed, type information was gathered by the run-time system, a straightforward process. This type information would then be available if and when recompilation occurred, permitting more aggressive optimization. Information gleaned using type feedback was later shown to be comparable with, and perhaps complementary to, information from static type inference [Agesen and Hölzle 1995; Agesen 1996].

2.8 Slim Binaries and Oberon

One problem with software distribution and maintenance is the heterogeneous computing environment in which software runs: different computer architectures require different binary executables. Even within a single line of backwards-compatible processors, many variations in capability can exist; a program statically compiled for the least-common denominator of processor may not take full advantage of the processor on which it eventually executes.

In his doctoral work, Franz addressed these problems using "slim binaries" [Franz 1994; Franz and Kistler 1997]. A slim binary contains a high-level, machine-independent representation[8] of a program module. When a module is loaded, executable code is generated for it on-the-fly, which can presumably tailor itself to the run-time environment. Franz, and later Kistler, claimed that generating code for an entire module at once was often superior to the method-at-a-time strategy used by Smalltalk and Self, in terms of the resulting code performance [Franz 1994; Kistler 1999].

Fast code generation was critical to the slim binary approach. Data structures were delicately arranged to facilitate this; generated code that could be reused was noted and copied if needed later, rather than being regenerated [Franz 1994].

Franz implemented slim binaries for the Oberon system, which allows dynamic loading of modules [Wirth and Gutknecht 1989]. Loading and generating code for

---

[6] The same comment, with slightly different wording, also appears in [Hölzle and Ungar 1994a, page 328].

[7] Hansen's work could ignore this possibility; the FORTRAN of the time did not allow recursion, and so activation records and a stack were unnecessary [Sebesta 1999].

[8] An abstract syntax tree, to be precise.

a slim binary was not faster than loading a traditional binary [Franz 1994; Franz and Kistler 1997], but Franz argued that this would eventually be the case as the speed discrepancy between processors and I/O devices increased [Franz 1994].

Using slim binaries as a starting point, Kistler's work investigated "continuous" run-time optimization, where parts of an executing program can be optimized *ad infinitum*. He contrasts this to the adaptive optimization used in Self, where optimization of methods would eventually cease [Kistler 1999].

Of course, re-optimization is only useful if a new, better, solution can be obtained; this implies that continuous optimization is best suited to optimizations whose input varies over time with the program's execution.[9] Accordingly, Kistler looked at cache optimizations — rearranging fields in a structure dynamically to optimize a program's data-access patterns [Kistler 1999; Kistler and Franz 1999] — and a dynamic version of trace scheduling, which optimizes based on information about a program's control flow during execution [Kistler 1999].

The continuous optimizer itself executes in the background, as a separate low-priority thread which executes only during a program's idle time [Kistler 1997; Kistler 1999]. Kistler uses a more sophisticated metric than straightforward counters to determine when to optimize, and observes that deciding *what* to optimize is highly optimization-specific [Kistler 1999].

2.9 Templates, ML, and C

ML and C make strange bedfellows, but the same approach has been taken to dynamic compilation in both. This approach is called "staged compilation," where compilation of a single program is divided into two stages: static and dynamic compilation. Prior to run-time, a static compiler compiles "templates," essentially building blocks which are pieced together at run-time by the dynamic compiler, which may also place run-time values into holes left in the templates. Typically these templates are specified by user annotations, although some work has been done on deriving them automatically [Mock et al. 1999].

As just described, template-based systems do not fit our description of JIT compilers, since there would appear to be no nontrivial translation aspect. However, templates may be encoded in a form which requires run-time translation before execution, or the dynamic compiler may perform run-time optimizations after connecting the templates.

Templates have been applied to (subsets of) ML [Leone and Lee 1994; Lee and Leone 1996; Wickline et al. 1998]. They have also been used for run-time specialization of C [Consel and Noël 1996; Marlet et al. 1999], as well as dynamic extensions of C [Auslander et al. 1996; Engler et al. 1996; Poletto et al. 1997].

2.10 Simulation, Binary Translation, and Machine Code

Simulation is the process of running native executable machine code for one architecture on another architecture.[10] How does this relate to JIT compilation? One

[9] Although, making the general case for run-time optimization, he discusses intermodule optimizations where this is not the case [Kistler 1997].

[10] We use the term "simulate" in preference to "emulate" as the latter has the connotation that hardware is heavily involved in the process. However, some literature uses the words interchangeably.

of the techniques for simulation is binary translation; in particular, we focus on dynamic binary translation that involves translating from one machine code to another at run-time. Typically, binary translators are highly specialized with respect to source and target; research on retargetable and "resourceable" binary translators is still in its infancy [Ung and Cifuentes 2000]. Altman, Kaeli, and Sheffer [2000] has a good discussion of the challenges involved in binary translation, and Cmelik and Keppel [1994] compares pre-1995 simulation systems in detail. Rather than duplicating their work, we will take a higher-level view.

May [1987] proposed that simulators could be categorized by their implementation technique into three generations. To this, we add a fourth generation to characterize more recent work.

(1) First generation simulators were interpreters, which would simply interpret each source instruction as needed. As might be expected, these tended to exhibit poor performance due to interpretation overhead.

(2) Second generation simulators dynamically translated source instructions into target instruction one at a time, caching the translations for later use.

(3) Third generation simulators improved upon the performance of second generation simulators by dynamically translating entire blocks of source instructions at a time. This introduces new questions as to what should be translated. Most such systems translated either basic blocks of code or extended basic blocks [Cmelik and Keppel 1994], reflecting the static control flow of the source program. Other static translation units are possible: one anomalous system, DAISY, performed page-at-a-time translations from PowerPC to VLIW instructions [Ebcioğlu and Altman 1996; Ebcioğlu and Altman 1997].

(4) What we call fourth generation simulators expand upon the third generation by dynamically translating paths, or traces. A path reflects the control flow exhibited by the source program at run-time, a dynamic instead of a static unit of translation. The most recent work on binary translation is concentrated on this type of system.

Fourth generation simulators are predominant in recent literature [Bala et al. 1999; Chen et al. 2000; Deaver et al. 1999; Gschwind et al. 2000; Klaiber 2000; Zheng and Thompson 2000]. The structure of these is fairly similar:

(1) Profiled execution. The simulator's effort should be concentrated on "hot" areas of code that are frequently executed. For example, initialization code that is executed only once should not be translated or optimized. To determine which execution paths are hot, the source program is executed in some manner and profile information is gathered. Time invested in doing this is assumed to be recouped eventually.

When source and target architectures are dissimilar, or the source architecture is uncomplicated (such as a RISC processor) then interpretation of the source program is typically employed to execute the source program [Bala et al. 1999; Gschwind et al. 2000; Transmeta Corporation 2001; Zheng and Thompson 2000]. The alternative approach, direct execution, is best summed up by Rosenblum, Herrod, Witchel, and Gupta [1995, page 36]:

> By far the fastest simulator of the CPU, MMU, and memory system of an SGI multiprocessor is an SGI multiprocessor.

In other words, when the source and target architectures are the same, as in the case where the goal is dynamic optimization of a source program, the source program can be executed directly by the CPU. The simulator regains control periodically as a result of appropriately modifying the source program[Chen et al. 2000] or by less direct means such as interrupts [Gorton 2001].

(2) Hot path detection. In lieu of hardware support, hot paths may be detected by keeping counters to record frequency of execution [Zheng and Thompson 2000], or by watching for code that is structurally likely to be hot, like the target of a backwards branch [Bala et al. 1999]. With hardware support, the program's program counter can be sampled at intervals to detect hot spots [Deaver et al. 1999].

Some other considerations are that paths may be strategically *excluded* if they are too expensive or difficult to translate [Zheng and Thompson 2000], and choosing good stopping points for paths can be as important as choosing good starting points in terms of keeping a manageable number of traces [Gschwind et al. 2000].

(3) Code generation and optimization. Once a hot path has been noted, the simulator will translate it into code for the target architecture, or perhaps optimize the code. The correctness of the translation is always at issue, and some empirical verification techniques are discussed in [Zheng and Thompson 2000].

(4) "Bail-out" mechanism. In the case of dynamic optimization systems (where the source and target architectures are the same), there is the potential for a negative impact on the source program's performance. A bail-out mechanism [Bala et al. 1999] heuristically tries to detect such a problem and revert back to the source program's direct execution; this can be spotted, for example, by monitoring the stability of the working set of paths. Such a mechanism can also be used to avoid handling complicated cases.

Another recurring theme in recent binary translation work is the issue of hardware support for binary translation, especially for translating code for legacy architectures into VLIW code. This has attracted interest because VLIW architectures promise legacy architecture implementations which have higher performance, greater instruction-level parallelism [Ebcioğlu and Altman 1996; Ebcioğlu and Altman 1997], higher clock rates [Altman et al. 2000; Gschwind et al. 2000], and lower power requirements [Klaiber 2000]. Binary translation work in these processors is still done by software at run-time, and is thus still dynamic binary translation, although occasionally packaged under more fanciful names to enrapture venture capitalists [Geppert and Perry 2000]. The key idea in these systems is that, for efficiency, the target VLIW should provide a superset of the source architecture [Ebcioğlu and Altman 1997]; these extra resources, unseen by the source program, can be used by the binary translator for aggressive optimizations or to simulate troublesome aspects of the source architecture.

2.11 Java

Java is is implemented by static compilation to bytecode instructions for the Java virtual machine, or JVM. Early JVMs were only interpreters, resulting in less-than-stellar performance:

> Interpreting bytecodes is slow. [Cramer et al. 1997, page 37]

> Java isn't just slow, it's *really* slow, *surprisingly* slow. [Tyma 1998, page 41]

Regardless of how vitriolic the expression, the message was that Java programs had to run faster, and the primary means looked to for accomplishing this was JIT compilation of Java bytecodes. Indeed, Java brought the term "just-in-time" into common use in computing literature.[11] Unquestionably, the pressure for fast Java implementations spurred a renaissance in JIT research; at no other time in history has such concentrated time and money been invested in it.

A early view of Java JIT compilation is given by Cramer, Friedman, Miller, Seberger, Wilson, and Wolczko [1997], who were engineers at Sun Microsystems, the progenitor of Java. They make the observation that there is an upper bound on the speedup achievable by JIT compilation, noting that interpretation proper only accounted for 68% of execution time in a profile they ran. They also advocated the direct use of JVM bytecodes, a stack-based instruction set, as an intermediate representation for JIT compilation and optimization. In retrospect, this is a minority viewpoint; most later work, including Sun's own [Sun Microsystems 2001], invariably began by converting JVM code into a register-based intermediate representation.

The interesting trend in Java JIT work [Adl-Tabatabai et al. 1998; Bik et al. 1999; Burke et al. 1999; Cierniak and Li 1997; Ishizaki et al. 1999; Krall and Grafl 1997; Krall 1998; Yang et al. 1999] is the implicit assumption that mere translation from bytecode to native code is not enough: code optimization is necessary too. At the same time, this work recognizes that traditional optimization techniques are expensive, and looks for modifications to optimization algorithms that strike a balance between speed of algorithm execution and speed of the resulting code.

There have also been approaches to Java JIT compilation besides the usual interpret-first-optimize-later. A compile-only strategy, with no interpreter whatsoever, was adopted by [Burke et al. 1999], who also implemented their system in Java; improvements to their JIT directly benefited their system. Agesen [1997] translated JVM bytecodes into Self code, to leverage optimizations already existing in the Self compiler. Annotations were tried by Azevedo, Nicolau, and Hummel [1999] to shift the effort of code optimization prior to run-time: information needed for efficient JIT optimization was precomputed and tagged on to bytecode as annotations, which were then used by the JIT system to assist its work. Finally, Plezbert and Cytron [1997] proposed and evaluated the idea of "continuous compilation" for Java in which an interpreter and compiler would execute concurrently, preferably

[11] Gosling [2001] points out that the term "just-in-time" is borrowed from manufacturing terminology, and traces his own use of the term back to about 1993.

on separate processors.[12]

3. CLASSIFICATION OF JIT SYSTEMS

In the course of surveying JIT work, some common attributes emerged. We propose that JIT systems can be classified according to three properties.

(1) Invocation. A JIT compiler is explicitly invoked if the user must take some action to cause compilation at run-time. An implicitly invoked JIT compiler is transparent to the user.
(2) Executability. JIT systems typically involve two languages: a source language to translate from, and a target language to translate to (although these languages can be the same, if the JIT system is only performing optimization on-the-fly). We call a JIT system monoexecutable if it can only execute one of these languages; polyexecutable if more than one can be executed. Polyexecutable JIT systems have the luxury of deciding when compiler invocation is warranted, since either program representation can be used.
(3) Concurrency. This property characterizes how the JIT compiler executes, relative to the program itself. If program execution pauses under its own volition to permit compilation, it is not concurrent; the JIT compiler in this case may be invoked via subroutine call, message transmission, or transfer of control to a coroutine. In contrast, a concurrent JIT compiler can operate as the program executes concurrently: in a separate thread or process, even on a different processor.

JIT systems that function in hard real-time may constitute a fourth classifying property, but there seems to be little research in the area at present; it is unclear if hard real-time constraints pose any unique problems to JIT systems.

4. TOOLS FOR JIT COMPILATION

General, portable tools for JIT compilation that help with the dynamic generation of binary code did not appear until relatively recently. To varying degrees, these toolkits address three issues:

(1) Binary code generation. As argued in [Ramsey and Fernández 1995], and as personal experience attests, emitting binary code such as machine language is a situation rife with opportunity for error. There are associated bookkeeping tasks too: information may not yet be available upon initial code generation, like the location of forward branch targets. Once discovered, the information must be backpatched into the appropriate locations.
(2) Cache coherence. CPU speed advances have far outstripped memory speed advances in recent years [Hennessy and Patterson 1996]. To compensate, modern CPUs incorporate a small, fast cache memory, the contents of which may get temporarily out of sync with main memory. When dynamically generating code, care must be taken to ensure that the cache contents reflect code written to main memory before execution is attempted. The situation is even

[12] Only compilation occurred concurrently, and only happened once, as opposed to the ongoing optimization of Kistler's "continuous optimization" [Kistler 2001].

	Binary Code Generation	Cache Coherence	Execution	Abstract Interface	Input
[Engler 1996]	•	•	•	•	ad hoc
[Engler and Proebsting 1994]	•	•	•	•	tree
[Fraser and Proebsting 1999]	•	•	•	•	postfix
[Keppel 1991]		•	•	•	n/a
[Ramsey and Fernández 1995]	•				ad hoc

Table 1. Comparison of JIT toolkits.

more complicated when several CPUs share a single memory. Keppel contains a detailed discussion [Keppel 1991].

(3) Execution. The hardware or operating system may impose restrictions which limit where executable code may reside. For example, memory earmarked for data may not allow execution (i.e., instruction fetches) by default, meaning that code could be generated into the data memory, but not executed without platform-specific wrangling. Again, refer to Keppel [Keppel 1991].

Only the first issue is relevant for JIT compilation to interpreted virtual machine code — interpreters don't directly execute the code they interpret — but there is no reason why JIT compilation tools cannot be useful for generation of non-native code as well.

Table 1 gives a comparison of the toolkits. In addition to indicating how well the toolkits support the three areas above, we have added two extra categories. First, an "abstract interface" is one that is architecture-independent. Use of a toolkit's abstract interface implies that very little, if any, of the user's code needs modification in order to use a new platform. The drawbacks are that architecture-dependent operations like register allocation may be difficult, and the mapping from abstract to actual machine may be suboptimal, such as a mapping from RISC abstraction to CISC machinery.

Second, "input" refers to the structure, if any, of the input expected by the toolkit. With respect to JIT compilation, more complicated input structures take more time and space for the user to produce and the toolkit to consume [Engler 1996].

Using a tool may solve some problems but introduce others. Tools for binary code generation help avoid many errors compared to manually emitting binary code. These tools, however, require detailed knowledge of binary instruction formats whose specification may itself be prone to error. Engler and Hsieh [2000] present a "metatool" that can automatically derive these instruction encodings by repeatedly querying the existing system assembler with varying inputs.

5. CONCLUSION

Dynamic, or just-in-time, compilation is an old implementation technique with a fragmented history. By collecting this historical information together, we hope to shorten the voyage of rediscovery.

ACKNOWLEDGMENTS

Thanks to Nigel Horspool, Shannon Jaeger, and Mike Zastre, who proofread and commented on drafts of this paper. Also, thanks to Rick Gorton, James Gosling, Thomas Kistler, Ralph Mauriello, and Jim Mitchell for supplying historical information and clarifications. Evelyn Duesterwald's PLDI 2000 tutorial notes were helpful in preparing Section 2.9.

REFERENCES

Abrams, P. S. 1970. *An APL Machine*. Ph. D. thesis, Stanford University. SLAC Report 114.

Adl-Tabatabai, A.-R., Cierniak, M., Lueh, G.-Y., Parikh, V. M., and Stichnoth, J. M. 1998. Fast, effective code generation in a just-in-time Java compiler. In *PLDI '98* (1998), pp. 280–290.

Agesen, O. 1996. *Concrete Type Inference: Delivering Object-Oriented Applications*. Ph. D. thesis, Stanford University. Also as SMLI TR-96-52, Sun Microsystems, January 1996.

Agesen, O. 1997. Design and implementation of Pep, a Java just-in-time translator. *Theory and Practice of Object Systems 3*, 2, 127–155.

Agesen, O. and Hölzle, U. 1995. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *OOPSLA '95* (1995), pp. 91–107.

Altman, E., Gschwind, M., Sathaye, S., Kosonocky, S., Bright, A., Fritts, J., Ledak, P., Appenzeller, D., Agricola, C., and Filan, Z. 2000. BOA: The architecture of a binary translation processor. Technical Report RC 21665, IBM Research Division.

Altman, E. R., Kaeli, D., and Sheffer, Y. 2000. Welcome to the opportunities of binary translation. *IEEE Computer 33*, 3 (March), 40–45.

Auslander, J., Philipose, M., Chambers, C., Eggers, S. J., and Bershad, B. N. 1996. Fast, effective dynamic compilation. In *PLDI '96* (1996), pp. 149–159.

Azevedo, A., Nicolau, A., and Hummel, J. 1999. Java annotation-aware just-in-time (AJIT) compilation system. In *JAVA '99* (1999), pp. 142–151.

Bala, V., Duesterwald, E., and Banerjia, S. 1999. Transparent dynamic optimization. Technical Report HPL-1999-77, Hewlett-Packard.

Bartlett, J. 1992. *Familiar Quotations* (16^{th} ed.). Little, Brown and Company. J. Kaplan, editor.

Bentley, J. 1988. Little languages. In *More Programming Pearls*, pp. 83–100. Addison-Wesley.

Bik, A. J. C., Girkar, M., and Haghighat, M. R. 1999. Experiences with Java JIT optimization. In *Innovative Architecture for Future Generation High-Performance Processors and Systems* (1999), pp. 87–94. IEEE.

Brown, P. J. 1976. Throw-away compiling. *Software — Practice and Experience 6*, 423–434.

Brown, P. J. 1990. *Writing Interactive Compilers and Interpreters*. Wiley.

Burke, M. G., Choi, J.-D., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M. J., Sreedhar, V. C., and Srinivasan, H. 1999. The Jalapeño dynamic optimizing compiler for Java. In *JAVA '99* (1999), pp. 129–141.

Chambers, C. 1992. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. Ph. D. thesis, Stanford University.

Chambers, C. and Ungar, D. 1989. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *PLDI '89* (1989), pp. 146–160.

Chambers, C. and Ungar, D. 1990. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *PLDI '90* (1990), pp. 150–164.

CHAMBERS, C. AND UNGAR, D. 1991. Making pure object-oriented languages practical. In *OOPSLA '91* (1991), pp. 1–15.

CHAMBERS, C., UNGAR, D., AND LEE, E. 1989. An efficient implementation of Self, a dynamically-typed object-oriented programming language based on prototypes. In *OOPSLA '89* (1989), pp. 49–70.

CHEN, W.-K., LERNER, S., CHAIKEN, R., AND GILLIES, D. M. 2000. Mojo: A dynamic optimization system. In *Third ACM Workshop on Feedback-directed and Dynamic Optimization (FDDO-3)* (December 2000).

CIERNIAK, M. AND LI, W. 1997. Briki: an optimizing Java compiler. In *Proceedings IEEE COMPCON '97* (1997), pp. 179–184.

CMELIK, B. AND KEPPEL, D. 1994. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 Conference on Measurement and Modeling of Computer Systems* (1994), pp. 128–137.

CONSEL, C. AND NOËL, F. 1996. A general approach for run-time specialization and its application to C. In *POPL '96* (1996), pp. 145–156.

CRAMER, T., FRIEDMAN, R., MILLER, T., SEBERGER, D., WILSON, R., AND WOLCZKO, M. 1997. Compiling Java just in time. *IEEE Micro 17*, 3 (May/June), 36–43.

DAKIN, R. J. AND POOLE, P. C. 1973. A mixed code approach. *The Computer Journal 16*, 3, 219–222.

DAWSON, J. L. 1973. Combining interpretive code with machine code. *The Computer Journal 16*, 3, 216–219.

DEAVER, D., GORTON, R., AND RUBIN, N. 1999. Wiggins/Redstone: An on-line program specializer. In *Proceedings IEEE Hot Chips XI Conference* (August 1999).

DEUTSCH, L. P. AND SCHIFFMAN, A. M. 1984. Efficient implementation of the Smalltalk-80 system. In *POPL '84 Proceedings* (1984), pp. 297–302.

DIECKMANN, S. AND HÖLZLE, U. 1997. The space overhead of customization. Technical Report TRCS 97-21 (December), University of California at Santa Barbara.

EBCIOĞLU, K. AND ALTMAN, E. R. 1996. DAISY: Dynamic compilation for 100% architectural compatibility. Technical Report RC 20538, IBM Research Division.

EBCIOĞLU, K. AND ALTMAN, E. R. 1997. Daisy: Dynamic compilation for 100% architectural compatibility. In *ISCA '97* (1997), pp. 26–37.

ENGLER, D. R. 1996. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *PLDI '96* (1996), pp. 160–170.

ENGLER, D. R. AND HSIEH, W. C. 2000. DERIVE: A tool that automatically reverse-engineers instruction encodings. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo '00)* (2000), pp. 12–22.

ENGLER, D. R., HSIEH, W. C., AND KAASHOEK, M. F. 1996. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *POPL '96* (1996), pp. 131–144.

ENGLER, D. R. AND PROEBSTING, T. A. 1994. DCG: An efficient, retargetable dynamic code generation system. In *ASPLOS VI* (1994), pp. 263–272.

FRANZ, M. 1994. *Code-Generation On-the-Fly: A Key to Portable Software.* Ph. D. thesis, ETH Zurich.

FRANZ, M. AND KISTLER, T. 1997. Slim binaries. *CACM 40*, 12 (December), 87–94.

FRASER, C. W. AND PROEBSTING, T. A. 1999. Finite-state code generation. In *PLDI '99* (1999), pp. 270–280.

GEPPERT, L. AND PERRY, T. S. 2000. Transmeta's magic show. *IEEE Spectrum 37*, 5 (May), 26–33.

GOLDBERG, A. AND ROBSON, D. 1985. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley.

GORTON, R. 2001. Private communication.

GOSLING, J. 2001. Private communication.

GSCHWIND, M., ALTMAN, E. R., SATHAYE, S., LEDAK, P., AND APPENZELLER, D. 2000. Dynamic and transparent binary translation. *IEEE Computer 33*, 3, 54–59.

HAMMOND, J. 1977. BASIC — an evaluation of processing methods and a study of some programs. *Software — Practice and Experience 7*, 697–711.

HANSEN, G. J. 1974. *Adaptive Systems for the Dynamic Run-Time Optimization of Programs.* Ph. D. thesis, Carnegie-Mellon University.

HENNESSY, J. L. AND PATTERSON, D. A. 1996. *Computer Architecture: A Quantitative Approach* (Second ed.). Morgan Kaufmann.

HÖLZLE, U. 1994. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming.* Ph. D. thesis, Carnegie-Mellon University.

HÖLZLE, U. AND UNGAR, D. 1994a. Optimizing dynamically-dispatched calls with run-time type feedback. In *PLDI '94* (1994), pp. 326–336.

HÖLZLE, U. AND UNGAR, D. 1994b. A third-generation Self implementation: Reconciling responsiveness with performance. In *OOPSLA '94* (1994), pp. 229–243.

ISHIZAKI, K., KAWAHITO, M., YASUE, T., TAKEUCHI, M., OGASAWARA, T., SUGANUMA, T., ONODERA, T., KOMATSU, H., AND NAKATANI, T. 1999. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *JAVA '99* (1999), pp. 119–128.

JOHNSTON, R. L. 1977. The dynamic incremental compiler of APL\3000. In *APL '79 Conference Proceedings*, Volume 9 of *APL Quote Quad* (June 1977), pp. 82–87. Number 4, Part 1.

JONES, N. D., GOMARD, C. K., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice Hall.

KEPPEL, D. 1991. A portable interface for on-the-fly instruction space modification. In *ASPLOS IV* (1991), pp. 86–95.

KEPPEL, D., EGGERS, S. J., AND HENRY, R. R. 1991. A case for runtime code generation. Technical Report 91-11-04, University of Washington Department of Computer Science and Engineering.

KISTLER, T. 1997. Dynamic runtime optimization. In *Proceedings of the Joint Modular Languages Conference (JMLC '97)* (1997), pp. 53–66.

KISTLER, T. 1999. *Continuous Program Optimization.* Ph. D. thesis, University of California, Irvine.

KISTLER, T. 2001. Private communication.

KISTLER, T. AND FRANZ, M. 1999. The case for dynamic optimization: Improving memory-hierarchy performance by continuously adapting the internal storage layout of heap objects at run-time. Technical Report 99-21 (May), University of California, Irvine. Revised September, 1999.

KLAIBER, A. 2000. The technology behind Crusoe processors. Technical report (January), Transmeta Corporation.

KNUTH, D. E. 1971. An empirical study of Fortran programs. *Software — Practice and Experience 1*, 105–133.

KRALL, A. 1998. Efficient JavaVM just-in-time compilation. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)* (1998), pp. 205–212.

KRALL, A. AND GRAFL, R. 1997. A Java just-in-time compiler that transcends JavaVM's 32 bit barrier. In *PPoPP '97 Workshop on Java for Science and Engineering* (1997).

LEE, P. AND LEONE, M. 1996. Optimizing ML with run-time code generation. In *PLDI '96* (1996), pp. 137–148.

LEE, S., YANG, B.-S., KIM, S., PARK, S., MOON, S.-M., EBCIOĞLU, K., AND ALTMAN, E. 2000. Efficient Java exception handling in just-in-time compilation. In *Java 2000* (2000), pp. 1–8.

LEONE, M. AND LEE, P. 1994. Lightweight run-time code generation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (1994), pp. 97–106.

Marlet, R., Consel, C., and Boinot, P. 1999. Efficient incremental run-time specialization for free. In *PLDI '99* (1999), pp. 281–292.

Mauriello, R. 2000. Private communication.

May, C. 1987. Mimic: A fast System/370 simulator. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques* (June 1987), pp. 1–13.

McCarthy, J. 1960. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM 3*, 4, 184–195.

McCarthy, J. 1981. History of LISP. In R. L. Wexelblat Ed., *History of Programming Languages*, pp. 173–185. Academic Press.

Miller, T. C. 1977. Tentative compilation: A design for an APL compiler. In *APL '79 Conference Proceedings*, Volume 9 of *APL Quote Quad* (June 1977), pp. 88–95. Number 4, Part 1.

Mitchell, J. G. 1970. *The Design and Construction of Flexible and Efficient Interactive Programming Systems*. Ph. D. thesis, Carnegie-Mellon University.

Mitchell, J. G. 2000. Private communication.

Mitchell, J. G., Perlis, A. J., and van Zoeren, H. R. 1968. LC^2: A language for conversational computing. In M. Klerer and J. Reinfelds Eds., *Interactive Systems for Experimental Applied Mathematics*. Academic Press. Proceedings of 1967 ACM Symposium.

Mock, M., Berryman, M., Chambers, C., and Eggers, S. J. 1999. Calpa: A tool for automating dynamic compilation. In *Second ACM Workshop on Feedback-directed and Dynamic Optimization* (1999), pp. 100–109.

Ng, T. S. and Cantoni, A. 1976. Run time interaction with FORTRAN using mixed code. *The Computer Journal 19*, 1, 91–92.

Pittman, T. 1987. Two-level hybrid interpreter/native code execution for combined space-time program efficiency. In *SIGPLAN Symposium on Interpreters and Interpretive Techniques* (1987), pp. 150–152.

Plezbert, M. P. and Cytron, R. K. 1997. Does "just in time" = "better late then never"? In *POPL '97* (1997), pp. 120–131.

Poletto, M., Engler, D. R., and Kaashoek, M. F. 1997. tcc: A system for fast, flexible, and high-level dynamic code generation. In *PLDI '97* (1997), pp. 109–121.

Ramsey, N. and Fernández, M. 1995. The New Jersey machine-code toolkit. In *Proceedings of the 1995 USENIX Technical Conference* (1995), pp. 289–302.

Rau, B. R. 1978. Levels of representation of programs and the architecture of universal host machines. In *Proceedings of the 11th Annual Microprogramming Workshop (MICRO-11)* (1978), pp. 67–79.

Rosenblum, M., Herrod, S. A., Witchel, E., and Gupta, A. 1995. Complete computer system simulation: The SimOS approach. *IEEE Parallel and Distributed Technology 3*, 4 (Winter), 34–43.

Schroeder, S. C. and Vaughn, L. E. 1973. A high order language optimal execution processor: *F*ast *I*ntent *R*ecognition *S*ys*t*em (FIRST). In *Proceedings of a Symposium on High-Level-Language Computer Architecture*, Volume 8 of *SIGPLAN* (November 1973), pp. 109–116. Number 11.

Sebesta, R. W. 1999. *Concepts of Programming Languages* (Fourth ed.). Addison-Wesley.

Smith, R. B. and Ungar, D. 1995. Programming as an experience: The inspiration for Self. In *ECOOP '95* (1995).

Sun Microsystems. 2001. The Java HotSpot virtual machine. White paper.

Thompson, K. 1968. Regular expression search algorithm. *Commun. ACM 11*, 6 (June), 419–422.

Transmeta Corporation. 2001. Code morphing software. http://www.transmeta.com/-technology/architecture/code_morphing.html.

Tyma, P. 1998. Why are we using Java again? *Commun. ACM 41*, 6, 38–42.

Ung, D. and Cifuentes, C. 2000. Machine-adaptable dynamic binary translation. In *Dynamo '00* (2000), pp. 41–51.

UNGAR, D. AND SMITH, R. B. 1987. Self: The power of simplicity. In *OOPSLA '87* (1987), pp. 227–242.

UNGAR, D., SMITH, R. B., CHAMBERS, C., AND HÖLZLE, U. 1992. Object, message, and performance: How they coexist in Self. *IEEE Computer 25*, 10 (October), 53–64.

University of Michigan. 1966a. The System Loader. In *University of Michigan Executive System for the IBM 7090 Computer*, Volume 1.

University of Michigan. 1966b. The "University of Michigan Assembly Program" ("UMAP"). In *University of Michigan Executive System for the IBM 7090 Computer*, Volume 2.

VAN DYKE, E. J. 1977. A dynamic incremental compiler for an interpretive language. *Hewlett-Packard Journal 28*, 11 (July), 17–24.

WICKLINE, P., LEE, P., AND PFENNING, F. 1998. Run-time code generation and Modal-ML. In *PLDI '98* (1998), pp. 224–235.

WIRTH, N. AND GUTKNECHT, J. 1989. The Oberon system. *Software — Practice and Experience 19*, 9 (September), 857–893.

YANG, B.-S., MOON, S.-M., PARK, S., LEE, J., LEE, S., PARK, J., CHUNG, Y. C., KIM, S., EBCIOĞLU, K., AND ALTMAN, E. 1999. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In *International Conference on Parallel Architectures and Compilation Techniques* (1999), pp. 128–138. IEEE.

ZHENG, C. AND THOMPSON, C. 2000. PA-RISC to IA-64: Transparent execution, no recompilation. *IEEE Computer 33*, 3 (March), 47–52.