A Tree-Based Alternative to Java Byte-Codes

Thomas Kistler and Michael Franz
Department of Information and Computer Science
University of California at Irvine
Irvine, CA 92697-3425

Abstract. Despite the apparent success of the Java Virtual Machine, its lackluster performance makes it ill-suited for many speed-critical applications. Although the latest just-in-time compilers and dedicated Java processors try to remedy this situation, optimized code compiled directly from a C program source is still orders of magnitude faster than software transported via Java byte-codes. This is true even if the Java byte-codes are subsequently further translated into native code. In this paper, we claim that these performance penalties are not a necessary consequence of machine-independence, but related to Java's particular intermediate representation. We have constructed a prototype and are further developing a software transportability scheme founded on a tree-based alternative to Java byte-codes. This tree-based intermediate representation is not only twice as compact as Java byte-codes, but also contains more detailed semantic information, some of which is critical for advanced code optimizations. Our architecture not only provides on-the-fly code generation from this intermediate representation, but also continuous re-optimization of the existing code-base by a low-priority background process. The re-optimization process is guided by up-to-the-minute profiling data, leading to superior optimization results.

1 Introduction

In recent months, the Java Virtual Machine [LYJ96] has rapidly become a standard platform for building portable Internet "applets" and applications. For these applications, portability is achieved by compiling Java source files into Java byte-codes (instruction sequences for the Java Virtual Machine) that are completely independent of the eventual target architecture. These byte-codes can easily be distributed over the Internet and interpreted on any given machine.

For small Internet applets, electronics, and household appliances, interpreting Java byte-codes yields adequate performance in most cases. For most other application areas, however, the performance penalty associated with interpreting byte-codes makes such an approach unsuitable—higher performance is required.

To remedy this situation, major software distributors have introduced *just-in-time* compilers. Just-in-time compilers translate Java byte-codes into a sequence of native machine instructions on a method-by-method basis upon first activation of a method; the compiled version is then cached for subsequent activations. According to manufacturers of such just-in-time compiler, e.g. [Sun95], the quality of the generated code is "reasonably good" and "almost indistinguishable from native C or C++".

Just-in-time compilers improve the situation relative to interpreted execution, but they still cannot compete with true optimizing compilers. Certain advanced optimizations rely on information that, although present in the source program, is lost in the transition to Java byte-codes, and whose reconstruction is extraordinarily difficult. Hence, compilers that take Java byte-codes as their input are not easily capable of performing intermodular optimizations, global trace scheduling, or code-parallelizations, which makes them intrinsically inferior to optimizing C or C++ compilers.

Since just-in-time compilers cannot always meet the required performance goals, most performance-critical applications continue to be compiled directly from source code into the machine language of the target

machine. The current proliferation of native plug-ins (software programs that extend the capabilities of Web browsers) rather than Java applets for high-performance applications clearly illustrates this point (e.g. Shockwave, PDFViewer, Live3D).

In this paper, we demonstrate that portability and high-performance are two goals that need not necessarily be irreconcilable. In the first part of this paper we describe an alternative intermediate representation that is based on high-level abstract syntax-trees rather than on low-level byte-codes. Abstract syntax trees provide the necessary foundation for advanced code optimizations and impose no artificial barriers to it.

In the second part, we introduce the concepts of *dynamic runtime optimization* and *adaptive profiling*. In our system, a dynamic runtime optimizer performs code optimizations *continuously*, based on runtime profile data. A background process regularly generates faster program versions that then replace earlier, less optimal versions. Basing compilation on an adaptive profiler allows the code optimizer to make superior optimization decisions, improving even the quality of already optimized code on subsequent re-optimization iterations.

2 A Tree-Based Intermediate Representation

Rather than compiling source files into a sequence of Java byte-codes or into a register transfer language [Wal86], source files in our implementation are translated into an intermediate representation called Slim Binaries [Fra94, FK96]. The Slim Binary representation is based on abstract syntax-trees and describes the actions of the original program similar to a parse tree. In contrast, abstract machine representations such as Java byte-codes are linear. In the Slim Binary representation, every node in the tree is strongly typed by a reference to the symbol table, in the byte-code representation, this type information as well as the block structure of the program are only implicitly present and not directly accessible.

The Slim Binary representation, as its name suggests, is exceptionally dense, more so than compressed source code or compressed object code, accelerating the transfer of executable content over a network. It is a variation of adaptive compression schemes, such as the popular LZW algorithm [Wel84], tailored towards syntax trees. It is based on the observation that different parts of programs often look very similar. As an example, expressions like `j++` or subexpressions like `...*pi/360` might be used several times within the same scope. The same holds for procedure calls. Procedures might be called repeatedly with similar parameter sets (e.g. `formatfloat(..., 10, 2)`). These similarities can be exploited by the use of a predictive algorithm that encodes recurring expressions and subexpressions efficiently both in terms of space and time.

In our implementation, the abstract syntax tree is reconstructed at load-time and native code is generated on-they. Slim Binaries cannot be easily interpreted at runtime which, at first sight, might be a disadvantage. Their structure is less suited in the area for which Java was originally invented—embedded systems, and advanced consumer electronics. This area mainly distinguishes itself by limited memory capacity and computing resources. However, this argument is becoming less relevant considering the recent increase of computing power and the recent reduction of memory prices. For personal computers, interpreted execution isn't very appealing at all.

Because code-generation is performed at load-time, and because generating code takes more time than merely linking programs, we have built a code generating loader with the explicit design goal of fast loading times. In this context, the importance of Slim Binaries being compact becomes even more significant. The time saved by the faster downloading of object files can be compensated for the on-they compilation phase. Measurements show that the resulting loading times are well within the range of what users are willing to tolerate even for large applications. Surprisingly the goal of fast load-times does not even go at the expense of code quality. The code generated by our loader is comparable in quality to commercial C and Java just-in-time compilers. In contrast to Java interpreters and just-in-time compilers, however, the full native speed of applications is brought into action from the very beginning of executing an application.

Slim Binaries have several advantages over Java byte-codes. First, a tree representation is likely to be more secure than byte-codes. The very definition of adaptive compression schemes limits the vocabulary at all times to symbols that can legally be accessed at the current position in the program. It is therefore hardly possible to construct a program that violates the scoping rules of the source language. Even if malicious applications could be constructed, scoping violations can easily be detected and handled during code generation, without resorting to mechanisms as complex as Java's byte-code verification. Byte code verification is a time-consuming process as it requires extensive data flow analysis.

Second, and much more important, the information available in our intermediate representation builds the foundation for advanced code optimizations. In contrast to Java, as we will show in the next section, we are able to apply more aggressive algorithms without large pre-processing costs, since essential data about control and data flow is preserved in the abstract syntax-tree.

3 Advanced Code Optimizations

In a runtime environment that is based on byte-codes, two categories of optimizations can basically be carried out. The first category encompasses optimizations that are completely independent of the eventual target architecture. Examples are constant folding, dead-code elimination, loop-invariant code motion, and to some extent, even procedure inlining. These optimizations can entirely be performed at *compile-time*, and on the level of the source language.

The second category comprises optimizations that depend on processor-specific information. Because this information is only available at *load-time*, these optimizations must operate on byte-code sequences. To improve performance, instructions can be rearranged to achieve a better instruction mix, or unnecessary and expensive register-spills can be eliminated by smart register allocation algorithms. Peephole optimizations can also be classed with this category of optimizations.

Yet, there is a third important category of optimizations that, like optimizations of the first group, operate on the level of the source language but also depend on processor specific information that is only available at load-time. These optimizations cannot be performed at all on Java byte-codes. Cache blocking [WL91] and loop-unrolling are two examples of these techniques. Analyzing and recognizing access patterns, as well as having precise information about important cache parameters (e.g. cache size, line size) are prerequisites for these optimizations. While the former can be accomplished at compile-time, the latter cannot in practice. Value numbering [CS70] poses a similar problem. If done at all at compile-time, byte-code instructions that cannot be mapped to the underlying architecture on a one-to-one basis, but have to be translated into a sequence of native instructions (e.g. `invokevirtual`, `invokestatic`, `invokeinterface`) cannot reasonably be taken into consideration. Delaying value numbering until load-time is also impractical. A further problem that belongs to the third category of optimizations is parallelizing instruction streams. Analyzing properties of data-structures can only be realized at compile-time. However, important information about underlying hardware parameters (e.g. number of processors) is not available until load-time.

Not being able to perform any of these optimizations is an immense disadvantage, which will be of prime importance in the near future. This holds especially for optimizations that parallelize instruction streams, since the tendency to cope with increasing performance requirements is rather to build multi-processor systems than single-processor systems.

Moreover, Java byte-codes have additional disadvantages. Directly mapping byte-codes onto the underlying architecture is much more difficult than generating machine instructions from an abstract syntax-tree. Code generators that are based on a high-level representation do not have to deal with unfavorable peculiarities of Java byte-codes but can tailor their output towards advanced and specific processor features, such as special purpose instructions, size of register sets, and cache architectures. This is especially true for today's most common RISC processors which are less suited for byte-code's heavily used stack operations. Whether

dedicated Java processors, such as Sun Microsystems recently announced picoJava architecture, will overcome this disadvantage is still an open question.

In contrast to Java byte-codes, Slim Binaries are optimally suited for *all categories* of code optimizations and do not have to deal with any of the bytecodes' disadvantages. At the time of loading, the abstract syntax tree, which can be efficiently decoded, contains the same amount of information that is available at compile-time. It not only preserves the control and data flow of programs, but also the structure and property of data-structures and data-types. This information is essential for aggressive code optimizations.

4 Runtime Optimization and Adaptive Profiling

Slim Binaries reconcile portability and efficiency by providing the foundation for code-optimizations at the time of loading. Unfortunately performing optimizations at load-time has one problem: it is quite time-consuming. In many cases it takes at least five times as long as simply compiling the program [Bra95]. This might be feasible for small applications, or large numerical applications in which the time saved by the optimizations is much more substantial than the additional time required to optimize the program. For all other applications, however, a different solution is necessary.

Therefore, in our design (Fig. 1), program optimization is performed at runtime, taking advantage of idle cycles (we measured idle times of more than 90%). At load-time, a fast code-generating loader transforms the intermediate representation into a first unoptimized code-image. The optimizer then continuously generates faster versions of the program in the background, replacing older code images "in situ". This step is repeated until a fixpoint is reached and further optimizations do not continue contributing to the overall system performance.

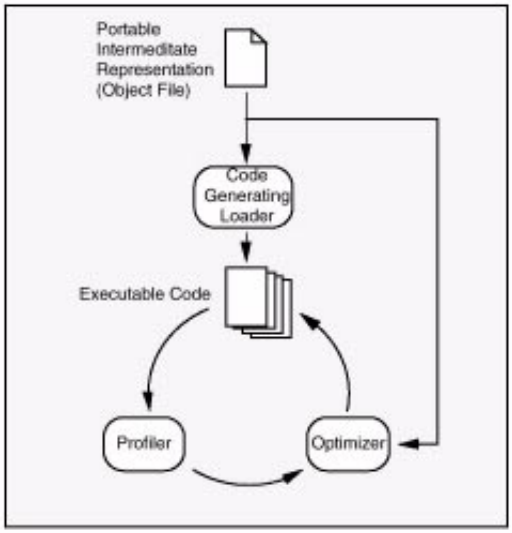

Fig. 1. Architecture

Performing optimizations at runtime also enables a completely new set of *intermodular* optimizations. Because the configuration of the system (i.e. which components are active, and how they interact) is known at runtime, optimizations are not restricted to local algorithms. Previous studies have shown that the impact of intermodular optimizations on runtime performance can be dramatic in some cases [Höl94]. Examples of intermodular optimizations are intermodular inlining, intermodular register allocation, and global cache optimizations [Kis96].

Runtime optimization is only one aspect of our architecture. Equally important is the adaptive profiler that continuously collects information about the system's runtime behavior. The profiler's primary goal is to pinpoint the program parts that account for most of the execution time. That way, optimizations can be

concentrated on high payoff areas rather than being applied uniformly to each section of the program. Less frequently executed sections are optimized sparsely, and no optimization is performed on rarely executed sections or sections in which optimizations would not yield profitable results.

Further, with the availability of accurate profiling-data at the time of optimization, the optimizer never has to resort to inexact heuristics. This leads to superior results in most cases. Many of today's aggressive optimization algorithms are based on heuristics, in order to achieve good results. However, this can be a double-edged sword. On the one hand, if the system's runtime behavior is properly predicted, considerable performance increases may be expected. On the other hand, if predictions do not come true, these optimizations will lead to performance penalties. As an example, in trace scheduling, traces (also called execution-paths) are selected and scheduled in decreasing order of their execution frequency. The most frequently executed path is scheduled first, as if it were one single basic block. However, in order to preserve semantic correctness, corresponding code motions have to be performed in off-trace paths. If, at runtime, the trace which was assumed to be executed most often is indeed executed most of the time, this optimization yields superior results. If that is not the case, and off-trace paths are executed more often, then this optimization will deteriorate the overall performance. Loop-unrolling which depends on loop-frequency estimates and cache parameters, or inlining and partial evaluation which depend on call-frequency estimates are other examples of optimizations that highly depend on heuristics.
In order to make the profiler as unobtrusive as possible, it uses a combination of dynamic instrumentation of the object code and statistical profiling techniques. It also varies the granularity at which it monitors the system's execution, and is only applied when it can contribute to the overall system performance, pushing the profiling overhead below 5%. Previous studies have reported profiling overheads of 5%-91% [BL94].

5 Results

In the last few months, we have implemented an experimental system that is based on our proposed architecture. The system, named "Juice", enables the seamless integration of Slim Binary encoded executables into HTML-pages. It is based on a family of Netscape plug-ins that contain an on-the-fly code generator and the Juice runtime environment. Juice is currently publicly available for Intel based computers running Windows 95 and for PowerPC based Macintosh computers.

Beside being reliable and simple to use, Juice is also efficient. Table 1 shows time-measurements for basic operations, such as assignments, additions, and method calls. The benchmark was executed on an Intel Pentium processor clocked at 166Mhz (Dell OptiPlex GXM 5166). Since neither the optimizer nor the profiler have yet been fully implemented and integrated into the Juice architecture, they have not been taken into account for all of the benchmarks (this special configuration that only applies on-the-fly compilation but no optimizations is subsequently called Juice Level I). Juice does very well in comparison to just-in-time compilers. The runtime-differences are only minimal. Both runtime systems achieve an average speed-up factor of 12 to 18 in contrast to byte-code interpretation.

	Internet Explorer 3.0 (Interpreted)	Netscape Navigator 3.0 (Just-In-Time)	Internet Explorer (Just-In-Time)	Juice Level I (No Opt.)
Local Var. Assignment	0.220	0.011	0.006	0.015
Instance Var. Assignment	0.440	0.010	0.007	0.046
Array Elem. Assignment	0.590	0.050	0.051	0.045
Byte Addition	0.680	0.044	0.030	0.021
Short Addition	0.660	0.044	0.030	0.047
Int Addition	0.570	0.015	0.013	0.017
Float Addition	0.570	0.046	0.045	0.054
Double Addition	0.500	0.140	0.044	0.110
Method Call	1.500	0.092	0.091	0.120
Average	0.637	0.050	0.035	0.053

Table 1. Suite 1—Basic Operations. All numbers are given in microseconds per operation.

Yet, speed-up factors in this range are not realistic in most cases. This has to be attributed to the fact that larger applications often call Java library routines that are distributed as native binaries, already optimized for speed. The more native libraries are called, the less just-in-time compilers boost performance. The results of the second benchmark, which comprises of several long-running, computational intensive tasks, emphasizes this statement. It compares the execution times of Juice, just-in-time compilers, and optimized C++ to the execution time of byte-code interpretation (Table 2). The speed-ups are remarkably smaller than the ones measured in the first test suite. Performance comprehensibly degrades with the number of library calls down to disappointing ratios of 2:1-4:1. Examples are the "String Sort" benchmark that frequently invokes the native "System.arraycopy" method and the "Fourier Analysis" benchmark that frequently calls the math library (Math.sin, Math.cos, Math.exp).

	Netscape Navigator 3.0 (Just-In-Time)	Internet Explorer 3.0 (Just-In-Time)	Juice Level I (No Opt.)	C++ Optimized
Numeric Sort	11.17	13.21	9.63	79.69
String Sort	3.50	4.72	1.55	6.70
Bitfield Operations	17.61	15.92	20.82	64.94
Fourier Analysis	0.87	2.76	2.45	4.27
IDEA Encryption	4.54	3.24	6.69	16.30
Huffman Compression	11.87	16.14	20.68	35.32
LU Decomposition	7.63	7.18	6.18	36.69
Average	8.43	9.19	10.05	35.33

Table 2. Suite 2—Computational Intensive Operations. All numbers are given in multiples of the performance of interpreted byte-codes using Internet Explorer.

This benchmark also unequivocally demonstrates that native code, compiled from an abstract syntax tree, is at least equivalent in quality to code generated by just-in-time compilers (Fig. 2). In some cases, the results even surpass the fastest available Java runtime systems—notably without applying any optimizations or profiling. However, the benchmarks also clearly demonstrate the current deficiencies of just-in-time compilers—they cannot yet compete with true optimizing compilers. Optimized C++ code is still an order of magnitude faster. In order to narrow this gap, aggressive and advanced optimizations are a necessity. The proposed tree-based intermediate representation fulfills all the requirements for achieving this goal.

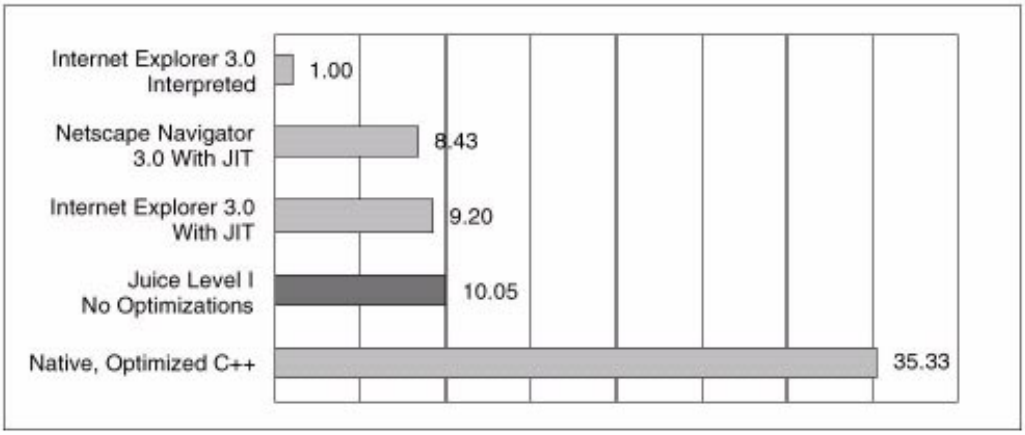

Fig. 2. Average Speed-Up Results

One of the initial claims of this paper was that a tree-based intermediate representation not only provides the basis for closing the efficiency gap between Java byte-codes and optimized C++ code, but also reduces the overhead for transferring files over a network. Not only are Slim Binary object files more than twice as dense as Java class files, and a factor of 3 to 4 smaller than traditional native object files, Slim Binaries are

even smaller than compressed native code. Figure 3 summarizes the results for the above test suite (consisting of 12 source files, approximately 130kBytes in size).

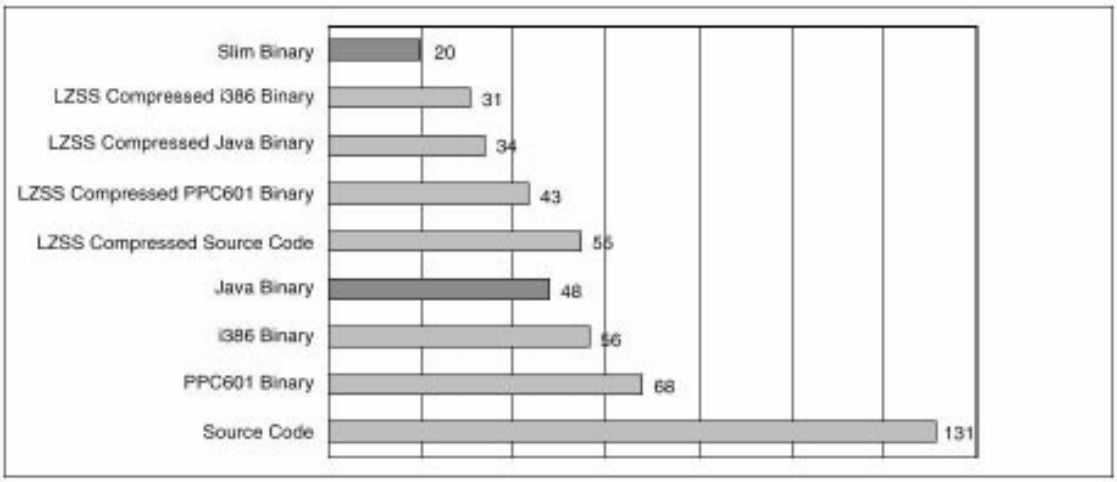

Fig. 3. Size Comparison Between Different Distribution Formats. All numbers are given in kBytes.

Although Slim Binaries reduce network traffic by a factor of two in terms of network packets, the differences between downloading Java class files and Juice Slim Binaries are almost indistinguishable in terms of download-time. This stems from the fact that for small execution units, it is mostly the time to set up the connection to a server that accounts for most of the waiting-time. However, with the introduction of component packaging concepts (compact archive formats for packaging the components of a Java or Juice application) the size of executable units will become more significant.

Finally, we have also measured the time that is required to compile the abstract syntax tree into native code. As mentioned earlier, the compilation time is hardly noticeable by the user. On a PowerPC based computer (Power Macintosh 8500/120) it takes approximately 470ms to compile the 12 benchmark-files. In comparison to the time required to download these files using a fast connection (4s), or using a slow connection (40s), the overhead of on-the-fly code generation can be neglected.

6 Conclusions

In the last few months, Java and the Java Virtual Machine have become a standard environment for building portable Internet “applets” and applications. However, despite its success, its lack of performance makes it ill-suited for many performance critical applications. Although just-in-time compilers try to remedy this situation and achieve speed-ups of 10-15 compared to interpreted bytecodes, they still cannot compete with true optimizing compilers.

In this paper, we have shown, that portability and high performance need not necessarily be irreconcilable. We have proposed an alternative intermediate representation that is based on abstract syntax trees rather than on low-level byte-codes. This intermediate representation, bundled with a dynamic runtime optimizer and an adaptive profiler, builds the basis for advanced and aggressive optimizations that are difficult to perform on a lower-level representation—much of essential information is lost in the transition from source code to byte-codes. We have shown that our current version of the proposed architecture can compete with today’s fastest just-in-time compilers, although no optimizations have yet been implemented. With the availability and integration of the runtime optimizer and the adaptive profiler, we will be able to level the performance with optimized C++ and dismantle the current performance deficiency of portable transportability schemes.

Additional information about Juice and research related topics at the University of California at Irvine can be found on the World Wide Web at the following location: http://www.ics.uci.edu/~juice.

References

[Bra95] M. M. Brandis; *Optimizing Compilers for Structured Programming Languages*; (Doctoral Dissertation) Eidgenössische Technische Hochschule Zürich; 1995

[BL94] T. Ball, J. R. Larus; Optimally Profiling and Tracing Programs; *In ACM Transactions on Programming Languages and Systems*, 16(4), pp 1319-1360; July 1994

[CS70] J. Cocke, J. Schwartz; *Programming Languages and Their Compilers: Preliminary Notes*; Courant Institute of Mathematical Sciences, New York University; April 1970

[FK96] M. Franz, Th. Kistler; *Slim Binaries*; Technical Report 96-24, Department of Information and Computer Science, UC Irvine; 1996

[Fra94] M. Franz; *Code-Generation On-the-Fly: A Key to Portable Software*; (Doctoral Dissertation) Verlag der Fachvereine, Zürich; 1994

[Höl94] U. Hölzle; Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming; (Ph.D. Dissertation) Department of Computer Science, Stanford University; 1994

[Kis96] Th. Kistler; *Dynamic Runtime Optimization*; Technical Report 96-54, Department of Information and Computer Science, UC Irvine; 1996

[LYJ96] T. Lindholm, F. Yellin, B. Joy, K. Walrath; *The Java Virtual Machine Specification*; Addison-Wesley; 1996

[Mot93] Motorola, Inc.; *PowerPC 601: RISC Microprocessor User's Manual*; 1993

[Sun95] Sun Microsystems; *The Java Language: An Overview*; http://java.sun.com/doc/Overviews/java/java-overview-1.html; 1995

[Wal86] D. W. Wall; Global Register Allocation at Link Time; *In Proceedings of SIGPLAN `86 Symposium on Compiler Construction*, pp 264-275; July

[Wel84 T. A. Welch; A Technique for High-Performance Data Compression; *IEEE Computer*, 17(6), pp 8-19; June 1984

[Wir88] N. Wirth; The Programming Language Oberon; In *Software-Practice and Experience* 18(7), pp 671-690; July 1988

[WL91] M. Wolf, M. Lam; A Data Locality Optimization Algorithm; *In Proceedings of the SIGPLAN `91 Conference on Programming Language Design and Implementation*, pp 30-44, Published as SIGPLAN Notices 26(6); June 1991