



# Vizer: A System to Vectorize Intel x86 Binaries

Keith D. Cooper, Anshuman Dasgupta, and Ken Kennedy

*Rice University  
Houston, Texas*

## 1 Introduction

Traditional compilers conduct optimizations on intermediate representations derived from high level source code. However, it is sometimes necessary and fruitful to optimize executables or compiled object files. This paper describes the Vizer system which automatically vectorizes object code for the Intel x86 architecture.

Binary optimization offers the opportunity to improve performance in situations where the optimizer cannot have access to the source code. A binary optimizer can analyze and modify compiled code to increase performance. It can also edit the code to insert instrumentation and sensor code that lets other tools observe the running program's behavior. While opportunities for binary optimization arise in many contexts, our interest is motivated from two distinct sources. In the Grid Application Development Software (GrADS) project, we are developing tools that optimize an executable version of the program after resources have been chosen and a mapping from the problem to those resources has been established. The technology developed in Vizer is part of the GrADS compilation system. Our second motivation comes from the opportunities present in legacy binaries—executable programs that we cannot recompile. The Vizer technology gives us a way to apply code optimization techniques directly to these programs and to rewrite them in ways that take advantage of new hardware features such as those on the Intel x86 line of processors.

The widespread use of the Intel x86 architecture make it an attractive target for binary optimization. Additionally, Intel regularly adds new features to new models of this line. For example, recent Pentiums support Single Instruction Multiple Data (SIMD) instructions called the Multimedia Extensions, Streaming SIMD (SSE), and Streaming SIMD 2 (SSE2) that operate on packed integer and floating point data. At the same time, however, these machines are backwards-compatible with older models in the line. Thus, code compiled for older processors such as the 80386 can run on the latest Pentium processors. Because of its popularity, there exists a large base of applications originally compiled for older x86 processors. Binary optimization offers a way to update these legacy applications so that they make good use of the modern features available on the latest Pentium models. The Vizer system demonstrates the potential for such improvement by implementing an object-level vectorizer that has the potential to dramatically reduce execution time for some codes.

Although vectorization of programs written in higher-level languages has been well studied [1, 10, 6], vectorizing low-level assembly code poses a number of additional challenges. A source-level

<pre>#include &lt;stdio.h&gt;  void fillArray(int* array, int N) {   for(int j = 1; j &lt; N; ++j) {     array [j] = array[j-1] + 33;   } }</pre>	<pre>.globl fillArray_FPii .globl main fillArray_FPii: _0:  push %ebp _1:  mov %esp,%ebp _3:  sub \$0x8,%esp _6:  movl \$0x1,0xffffffff8(%ebp) _d:  lea 0x0(%esi),%esi _10: mov 0xffffffff8(%ebp),%eax _13: cmp 0xc(%ebp),%eax _16: jl _1c _18: jmp _44 _1a: mov %esi,%esi _1c: mov 0xffffffff8(%ebp),%eax _1f: imul \$0x4,%eax,%ecx _22: mov 0x8(%ebp),%edx _25: mov 0xffffffff8(%ebp),%eax _28: mov %eax,%eax _2a: shl \$0x2,%eax _2d: add 0x8(%ebp),%eax _30: sub \$0x4,%eax _33: mov (%eax),%eax _35: add \$0x21,%eax _38: mov %eax,(%edx,%ecx,1) _3b: lea 0xffffffff8(%ebp),%eax _3e: incl (%eax) _40: jmp _10 _42: mov %esi,%esi _44: leave _45: ret</pre>
<b>C++ Source Code</b>	<b>Assembly File Recreated by Vizer</b>

Figure 1: A Simple Vector Addition

vectorizer identifies the array reads and writes, performs dependence analysis followed by some transformations, and emits a vectorized version of the program. The source-level representation makes it easy to identify both control-flow and array references, since they are explicit in the program. In contrast, the assembly program encodes both the control-flow and the data structures of the source program in a form where they are much harder to discover. As a result, an object-level vectorizer must somehow regenerate high-level structures from finely detailed machine instructions.

To see this, consider the simple vector addition shown in Figure 1. The left side of the figure shows the C++ source code. The right side of the figure shows the corresponding assembly code. (Vizer uses a disassembler on the compiled code to produce this form of the program.) Where the loops, array references, and induction variables are easy to find in the C++ program, they are well hidden in the assembly code. From the assembly code, it is not obvious that the instructions

represent a vector addition. To vectorize the assembly code, Vizer must find the control-flow, identify the array references and induction variables, and perform dependence analysis on this low-level form of the code. If the analysis shows that vectorization is safe, Vizer rewrites the code into vector form.



Figure 2: Overview of Vizer

Figure 2 shows a high-level view of vizer. Vizer takes as input one or more object files in the Executable and Linkable Format (ELF) [15]. The front end disassembles an object file into its corresponding assembly instructions. These serve as the intermediate representation (IR) for the rest of the system. The analysis phase of Vizer searches for opportunities for vectorization in the IR. The transformation phase takes advantage of the opportunities identified by analysis by inserting appropriate Streaming SIMD instructions. It also emits the final optimized-object file.

## 2 Related Work

Some prior work has been done on binary optimization for different architectures. A number of systems provide mechanisms to analyze, insert instrumentation, and conduct transformations on binaries. Etch, a system to analyze Windows executables, allows the user to write tools which monitor key features of the program during runtime [11]. The user can also use the framework provided by the system to conduct optimizations by rewriting the executables. The IMPACT binary reoptimization framework also optimizes Windows executables [14]. IMPACT generates an intermediate format called Mcode which can then be used to transform the code. The Executable Editing Library (EEL) is a similar system for Solaris and SunOS executables [9]. EEL generates abstractions in the form of C++ classes that can be used to manipulate the program. OM is a system for code optimization at link time for MIPS binaries [13]. Unlike IMPACT and EEL, OM analyzes object files and libraries instead of executables [13]. The system conducts inter-procedural code motion to move loop-invariant code out of loops. ATOM extends OM to enable the addition of instrumentation to MIPS object code [12].

These systems address many of the issues that arise in manipulating object-level and binary representations of programs. They do not, however, perform transformations as ambitious as automatic vectorization on the low-level code. Vizer tackles a particularly difficult problem — vectorization of this low-level code for the Intel processors. This particular mission was heavily influenced by the needs of the GrADS project; Vizer will serve as the basis for the GrADS component that tailors a program at load-time.

### 3 Disassembling and Parsing the Input

The first phase of Vizer must read the input program, in object-code form, and construct an intermediate representation (IR) for use by the later phases. It relies heavily on other tools to understand the ELF-format code. It uses the `objdump` utility, from the GNU binutils tools to disassemble the object code [2]. `Objdump` translates the object code into a series of code and data sections, along with relocation information. It also disassembles the the code sections.

The output of `objdump` is a text file with sufficient detail to let a user read the code. Vizer needs a file that can be parsed into a useful IR and, eventually, be re-assembled into object code. Thus, it rewrites the output of `objdump` into a valid assembly language program. It converts absolute addresses into relative addresses, recreates the labels on procedures and constant values, rewrites uses of those labels in loads and procedure calls,<sup>1</sup> and rewrites the data sections into the form needed by the GNU assembler. Vizer passes this rewritten file to a modified version of the GNU assembler that builds a simple, low-level IR for use by Vizer's later phases. The modified assembler preserves critical information about labels and pseudo-instructions.

### 4 Analysis of the Intermediate Representation

Vizer's second phase analyzes the IR version of the program to find opportunities for vectorization. It must reconstruct the control-flow and data-flow of the code from the assembly code, build a model of the data structures that the code uses, and perform dependence analysis to determine if individual operations can be performed in parallel.

#### 4.1 Control and Data-flow Analysis

As the first step in the analysis phase, Vizer derives a control-flow graph (CFG) from the IR version of the program. To simplify later analysis, it treats each assembly operation as a distinct node in the graph; edges in the CFG then represent the control flow between instructions. Vizer solves a series of data-flow problems over the CFG. It computes dominators, postdominators, dominance frontiers, and control dependences [1]. Using this information, it converts the code into Static Single Assignment form (SSA) [5, 3]. SSA concisely represents the uses and definitions of values in the program and relates them to the flow of control; this representation simplifies the later analysis and transformation.

**SSA for the x86** Some features of the x86 instruction set pose challenges for the SSA construction. The algorithms assume, implicitly, that all definitions and uses are explicit and visible. Some x86 instructions operate on registers that are not explicitly names. Vizer's SSA construction algorithm relies on a list of instructions that have implicit uses and definitions. This lets it insert enough information to correctly model the implicit effects.

The SSA construction algorithms assume, implicitly, that operations are non-destructive. De-

---

<sup>1</sup>Because the code being analyzed can call other procedures, it must undo the effects of name-mangling on external references so that the re-assembly process will produce results that match the symbols in those external code and data segments.

A push from memory onto the stack <code>fld -8(%ebx)</code> turns into 7 pseudo-copies: <code>psmov st(6) =&gt; st(7)</code> <code>psmov st(5) =&gt; st(6)</code> <code>...</code> <code>psmov st(1) =&gt; st(2)</code> <code>psmov -8(%ebp) =&gt; st(0)</code>	A pop from the stack into memory <code>fstp -8(%ebp)</code> turns into 7 pseudo-copies: <code>psmov st(0) =&gt; -8(%ebp)</code> <code>psmov st(1) =&gt; st(0)</code> <code>psmov st(2) =&gt; st(1)</code> <code>...</code> <code>psmov st(7) =&gt; st(6)</code>
<b>Effect of a Push</b>	<b>Effect of a Pop</b>

Figure 3: Effects of Pop and Push Operations on the Floating Point Register Stack

structive operations, such as the two-address operations in the x86, conflict with SSA’s principle of giving each static definition site a unique name. For example, in  $x \leftarrow x + y$ , the right-hand  $x$  is distinct from the left-hand  $x$ . In SSA, they would have distinct subscripts, as in  $x_4 \leftarrow x_2 + y_{13}$ . To resolve this problem, Vizer’s SSA module maps the reused operand into distinct read and write operands and provides a mechanism to differentiate between the two. This has the effect, internally, of treating two-address operations as if they were three-address operations, without sacrificing Vizer’s ability to regenerate the necessary destructive operations when it emits the final code.

**Stack-based Floating-point Registers** The x86 architecture treats its floating point registers as a stack, with wrap-around operations [8]. The eight floating-point registers are numbered R0 through R7. Memory operations can only operate on the register that is currently the top of the stack. Thus, a load operation corresponds to a push and a store operation corresponds to a pop of the stack.

Rather than moving the data on each stack operation, the processor changes the mapping from stack names to register names. All addressing of the data registers is relative to the stack-top register. Thus, the data registers are referenced as ST(0) through ST(7) where ST(0) is the top of the stack and ST(7) is the bottom element in the stack. Push and pop operations change the mapping from register names (R0 through R7) to stack names (ST(0) through ST(7)). If ST(0) is R7 and the processor executes a push operation, the stack top wraps so ST(0) becomes R0. If the next operation is a pop, ST(0) corresponds to R7 after the pop executes.

This feature creates a fundamental ambiguity; the name ST(0) can refer to any of the eight floating point registers. To handle this problem, Vizer expands pushes and pops into a sequence of register-to-register pseudo-copies. The copies explicitly model the stack’s behavior. Figure 3 shows this translation.

**Push** The left side details the effects of a push, or load, operation. Starting from the stack bottom, or ST(7), pseudo-copies are generated to move the value from the  $i^{th}$  register in the stack to the  $(i + 1)^{st}$  register in the stack. A pseudo-copy is also generated to move the value from memory into ST(0).

**Pop** The right side details the effects of a pop, or store, operation. An initial pseudo-copy simulates the move of  $ST(0)$  to the memory location specified in the instruction. Then, starting from  $ST(1)$ , pseudo-copies are generated to move the value from the  $(i + 1)^{st}$  register in the stack to the  $i^{th}$  register in the stack.

These pseudo-copies make all of the data movement explicit and provide an unambiguous name space for the floating point register. They introduce a seven-fold expansion into the code. Fortunately, copy-folding on the SSA can remove these extraneous copies at a later stage in translation. Thus, Vizer can safely insert the copies to simplify analysis, without having them appear in the final code.

## 4.2 Extracting Higher Level Entities

After control-flow and data-flow analysis, Vizer tries to identify vectorizable operations. In a high-level language, such as C++, the loops, arrays, and induction variables needed for such analysis are explicit. Thus, a source-level vectorizer can easily extract and analyze such constructs. Working from compiled code, however, these constructs are encoded in the assembly code and hidden. However, as shown in Figure 1, compiling the source code into assembly code can discard much of the high-level information and hide the data structures quite effectively.

To vectorize the assembly-level IR, Vizer must reconstruct a high-level model of the program that includes loops, arrays, and induction variables. It can then perform dependence tests on these constructs to discover which operations can be safely vectorized. As a matter of terminology, we refer to the registers and memory locations in the assembly code that correspond to the high-level constructs as *high-level entities*. Identifying high-level entities is a challenging prospect, but it is crucial to the success of Vizer.

**Identifying Loops** The statements that Vizer can vectorize are all contained in loops. The analyzer must reconstruct loops from the branching structure of the assembly code. Vizer uses the dominance information computed on the control-flow graph to identify loops. It currently focuses on loops with a single exit. It considers all branch instructions in the CFG which post-dominate a compare instruction. If branch uses the result of the compare instruction and the compare post-dominates one of the branch’s targets, then a cyclic path exists from the branch instruction passing through the compare instruction and back to the branch—forming a loop. Vizer marks all nodes which lie on the cycle as part of the loop.

**Identifying Induction Variables** An induction variable is a variable defined on every iteration of the loop and incremented by a constant amount in every iteration [1]. If the induction variables are held in registers, they can be identified by analyzing the SSA graph and the operations in the loop [4]. However, many compilers for the Intel x86 architecture, like the GNU C++ compiler, g++, generate induction variables that are held in memory — typically the runtime stack — because of the small number of general purpose registers present in the architecture. Focusing exclusively on register-based induction variables misses many of the induction variables in the disassembled

Definition of High-level Entity		Potential High-level Construct
<i>StackLocation</i>	::= <All exprs pointing to some location in the stack>	
<i>StackValue</i>	::= <i>DeRef(StackLocation)</i>	
<i>InductionVar</i>	::= <i>PrimaryInductionVar</i>   <i>SecondaryInductionVar</i>   < <i>InductionVar From Outer Loop</i> >	
<i>PrimaryInductionVar</i>	::= <Value either in register or stack incremented on each iteration and controls the exit condition of the loop>	
<i>SecondaryInductionVar</i>	::= <i>Constant * PrimaryInductionVar</i>	
<i>ArrayPointer</i>	::= <i>StackValue</i>   <i>StackLocation</i>   <i>ArrayPointer Op Constant</i>   <i>DeRef(ArrayPointer Op InductionVar)</i>	<i>Array[0]</i> <i>&amp;(Array[C Op K])</i> <i>&amp;(Array[InductionVariable])</i> <i>*ArrayPointer</i>
	$K = \text{Constant} / (\text{sizeof(Data)} / \text{Addressable Size})$	
<i>ArrayValue</i>	::= <i>ArrayPointer (except StackLocations)</i>   <i>ArrayPointer Op Constant</i>	<i>Array[i]</i> <i>Array[i]</i>
<i>ArrayReads</i>	::= <i>DeRef(ArrayPointer + InductionVar)</i>	
<i>ArrayWrites</i>	::= <i>DeRef(ArrayPointer + InductionVar)</i>	

Figure 4: Definitions of the High-level Entities in Vizer

code. Thus, Vizer tracks some local variables through the stack to improve its ability to identify induction variables.

The register-use convention on the Intel x86 uses one register, called the base pointer, for most accesses to the runtime stack. Vizer takes advantage of this fact. It iterates over the instructions in the loop and locates operands that refer to a stack location. It uses the SSA form to discover which of these stack locations are, in fact, used as induction variables. The analysis identifies the expressions and registers pointing to locations in the stack. It uses this information to identify the expressions and registers that contain values loaded from the stack. It uses simple symbolic analysis to determine which of these stack values have the same symbolic value and marks them as equivalent.<sup>2</sup> This information is used in the dependence testing phase.

**Identifying Other High Level Entities** To identify vectorizable operations, Vizer must recognize other high-level entities. In particular, pointers to arrays, array reads, and array writes are essential. To find array pointers, Vizer examines all the operands used in a loop. Operands

<sup>2</sup>For example, the g++ compiler often generates symbolically equivalent induction variables.

```

*****
For Loop Number:  0
*****
Induction Variables
Counter0:  -8 (ebp (1002))           Counter0:  %eax (1010)
Counter9:  %eax (1023)             Counter9:  %ecx (1016)
...
Array Pointers:
&(StackValue0[] ):  (8 (ebp (1002)))
&(StackValue0[Counter9] [] ):  ((edx (1019),ecx (1016)))
&(StackValue0[Counter9 + -1 ] [] ):  ((eax (1025)))
&(StackLocation6[] [] ):  ((eax (1028)))
&(StackValue0[] ):  (%edx (1019))
...
Array Values:
StackValue0           StackValue0[Counter9]
StackValue0[Counter9 + -1 ]  StackLocation6[]
...
Array Reads:
StackValue0[Counter9 + -1 ]

Array Writes:
StackValue0[Counter9] = StackValue0[Counter9 + -1] + 33 (Block #:  21)

```

Figure 5: Some of the High-level Entities Identified in the Example

used as memory addresses are potential array pointers, unless they point to an induction variable. Any expression consisting of a constant added to an array pointer is also a potential array pointer. Array reads and writes are identified by looking for expressions that dereference an array pointer, indexed by an induction variable.

Figure 4 shows the rules used to classify operands as various kinds of high-level entities. This classification scheme allows Vizer to raise the level of abstraction with which it views the disassembled code. At this point, it can reason about the important higher-level constructs instead of working with the lower-level assembly constructs. Figure 5 shows a subset of the high-level entities that Vizer finds in the assembly code from Figure 1.

**Finding Vectorizable Statements** As the last step in its analysis phase, Vizer looks for vectorizable array accesses. The current version of Vizer has a limited range: it only considers loops that contain exactly one operation that writes to an array. It assumes that array pointers present in two distinct stack locations point to different arrays. This is equivalent to assuming that formal parameters are not aliased, a common assumption in Fortran systems. Vizer uses a simple symbolic analysis of the induction variables to check for dependences between array reads and writes. This analysis also ensures that reads and writes of the arrays occur to consecutive mem-

ory locations. Vizer marks a loop for vectorization only if it can prove that the loop contains no vectorization-preventing data dependences.

We plan to extend this phase of the analysis in several distinct ways. A more complete dependence test will increase the set of loops that can be recognized as vectorizable. Similarly, it will allow vectorization of loops containing writes to more than one array. Finally, we will add a mechanism for identifying and marking loops that need a runtime check for safety; this will let the transformation phase emit a test that selects either the scalar or the vector version of a loop, as appropriate.

## 5 Transformation Phase: Vectorization and Final Emission of the Code

Once the analysis phase has identified the vectorizable loops, the final phase in Vizer must rewrite the assembly code into vector form. In general, the vectorized code loads each operand of the vectorizable statement into a vector register. If the operand is constant in the loop, the load is scheduled before entry into the loop. After the loads of all the operands have been scheduled, Vizer schedules the series of necessary vector operations. Finally, it schedules the store operations to move the result back into the appropriate memory locations.

**Sum Reductions** Some vectorizable loops require special treatment. Sum-reduction loops are a good example. Scalar sum-reductions cannot be vectorized without transformation. Consider a sum reduction of the form:

```
DO i = 1, 100
  S = A[i] * B[i] + S
ENDDO
```

This reduction cannot be trivially vectorized. However, if the sum-reduction is recognized, then the loop can be vectorized as:

```
Temp1[1:4] = 0
DO i = 1, 100, 4
  Temp2[1:4] = A[i:i+4] * B[i:i+4]
  Temp1[1:4] = Temp1[1:4] + Temp2[1:4]
ENDDO
S = S + Temp1[1];
S = S + Temp1[2];
S = S + Temp1[3];
S = S + Temp1[4];
```

Vizer both recognizes this sum-reduction and implements the transformation required to vectorize it in this manner.

Benchmark	Original Runtime	After Vizer Vectorizes	Improvement
Matrix Addition	3.503	1.593	54.53%
Matrix Multiplication	3.680	1.700	53.53%
Erlebacher	20.463	18.280	10.67%

Figure 6: Execution Time of Benchmarks Before and After Running Vizer (sec)

**Changing Loop Bounds** The x86 Streaming SIMD operations operate on four elements at a time. This requires that the loop bounds be rewritten to provide the correct behavior. The current version of Vizer does not do this; instead, it assumes that each loop iterates a multiple of four times. We will enhance the transformation stage of Vizer to insert the appropriate pre-loop and post-loop code to handle the general case.

**Dead Code Elimination and Emission of Vectorized Code** After it has inserted the vectorized loop, Vizer must eliminate the original scalar operations from the loop. When it inserts the vector store operation, Vizer marks the original scalar store instruction as dead. After all possible loops are vectorized, the SSA built from the original IR is no longer valid. Vizer regenerates SSA for the modified program and performs dead code elimination [5]. This eliminates the redundant scalar operations. Finally, Vizer uses the GNU Assembler to produce the final, optimized, vectorized object file.

## 6 Experimental Results

To assess the effectiveness of Vizer, it was tested on a number of programs. Because it does not perform the traditional loop transformations that enhance opportunities for vectorization, its range of effectiveness is somewhat limited. Hence, we tested it on benchmarks that contain statements vectorizable without loop transformations.

The results of running Vizer on three such programs: the Erlebacher benchmark, matrix multiply, and matrix addition are shown in Figure 6. The Erlebacher benchmark written by Thomas Edison at ICASE consists of around 600 lines of FORTRAN code which performs 3-dimensional tridiagonal solves [7]. Matrix multiply and matrix add are both conducted on 2-dimensional matrices of dimensions  $512 * 512$ . The experiments were performed on a Pentium-4 processor with 512 MB memory, running Redhat Linux 7.1. The matrix addition and multiplication benchmarks were compiled with the GNU C++ compiler, `g++`. The Erlebacher benchmark was compiled with the GNU Fortran compiler, `g77`.

As can be seen from the table, Vizer performed extremely well on the matrix addition and multiplication benchmarks. improving the runtime by over 50%. On the Erlebacher benchmark, Vizer performs well on the Erlebacher loops attaining an improvement of over 10%. In the Erlebacher benchmark, nine procedures contained loops. Loops in two of these nine procedures could

be vectorized by Vizer resulting in the improvement.

## 7 Conclusions and Future Directions

This paper describes Vizer, a vectorizer for Intel x86 binaries. Vizer can successfully derive enough high-level information from object files to identify vectorization opportunities. The experimental results show that Vizer, in its current form, can identify and exploit simple opportunities in object code. The potential for improvement is significant; Vizer produced significant improvements in compiled (and optimized) programs.

The current version of Vizer is a prototype that misses many opportunities for vectorization. We intend to enhance its symbolic analysis capabilities and add more sophisticated dependence testing. We will implement loop distribution and loop peeling to expose more opportunities for vectorization. Finally, we will add other optimizations to Vizer to improve the original object code; candidates for implementation include constant propagation, lazy code motion, register scavenging and assignment, and value numbering.

Even with its current limitations, Vizer has demonstrated the potential for performing powerful transformations, like vectorization, in a binary-translation framework. This approach has the potential to allow optimization of legacy codes, binary-only libraries, and other executables where source code is not available. The Vizer system will serve as an initial load-time optimizer in the GrADS program execution system.

## 8 Acknowledgements

This work was supported jointly by the NSF-sponsored GrADS project, NSF grant number 9975020, and the Los Alamos Computer Science Institute. Tim Harvey suggested the changes in the SSA algorithm to keep track of floating point values on the stack. Todd Waterman and Tim Harvey read earlier drafts of this document and made many important suggestions. The many members of the compiler groups at Rice provided us with support, encouragement, advice, and tools. their support. To all these people go our heartfelt thanks.

## References

- [1] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, first edition, 2001.
- [2] GNU Binutils. The GNU Project, <http://www.gnu.org/software/binutils/binutils.html>.
- [3] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. *Software – Practice and Experience*, 28(8):859–881, 1998.
- [4] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. Operator strength reduction. *ACM Transactions on Programming Languages and Systems*, 23(5):603–625, September 2001.
- [5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
- [6] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe. The Structure of an Advanced Vectorizer for Pipelined Processors. In *IEEE Computer Society Fourth International Computer Software and Applications Conference*, 1980.
- [7] T. M. Edison and G. Erlebacher. Implementation of a Fully-balanced Periodic Tridiagonal Solver on a Parallel Distributed Memory Architecture. Technical report, ICASE, 1994. Technical Report TR-94-37.
- [8] Intel Corp. *IA-32 Intel Architecture Software Developer’s Manual*, 2001.
- [9] James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1995.
- [10] Randy Allen and Ken Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. In *ACM Transactions on Programming Languages and Systems*, pages 491–542, 1987.
- [11] Ted Romer, Geoff Voelker Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian N. Bershad, and J. Bradley Chen. Instrumentation and optimization of Win32/Intel executables using etch. In *USENIX Windows NT Workshop*, 1997.
- [12] Amitabh Srivastava and Alan Eustace. ATOM. A System for Building Customized Program Analysis Tools. In *SIGPLAN ’94 Conference on Programming Language Design and Implementation (PLDI)*, pages 49–60, 1994.
- [13] Amitabh Srivastava and David W. Wall. A Practical System for Intermodule Code Optimization at Link-time. *Journal of Programming Languages*, 1(1):1–18, December 1992.
- [14] M. Thiems. Optimization and Executable Regeneration in the IMPACT Binary Reoptimization Framework. Master’s thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1998.
- [15] Tool Interface Standard Committee. *Tool Interface Standard Portable Formats Specification Version 1.1*, 1993.