RICE UNIVERSITY

Vizer: A Framework to Analyze and Vectorize Intel x86 Binaries

by

Anshuman Dasgupta

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Keith D. Cooper, Professor, Co-Chair
Computer Science

Ken Kennedy, Professor, Co-Chair
Computer Science

Robert Jump, Professor
Electrical and Computer Engineering

Rice University, Houston, Texas

November, 2002
Vizer: A Framework to Analyze and Vectorize Intel x86 Binaries

Anshuman Dasgupta

Abstract

Binary optimization has become increasingly important due to the widespread use of legacy programs and the presence of environments that require dynamic optimizers. Although processor technology changes constantly, software developers usually compile their programs for older processors to maintain backward compatibility. These legacy applications can be specialized by binary optimizers to take advantage of hardware features found only on newer processors. Other binary optimizers, such as just-in-time compilers and dynamic optimizers for computational grids, strive to improve the efficiency of programs at execution time. These optimizers for executables and object codes typically conduct low-level optimization and do not attempt many of the transformations that are carried out for programs written in high-level languages.

This thesis presents Vizer, which is a framework for conducting high-level optimizations on binary programs. Vizer analyzes binary files and reconstructs data structures and control flow that were present in the high-level source code used to create the binary. This information can be used to implement optimizations that are otherwise not possible in binary optimizers. Vizer conducts one such optimization: the vectorization of Intel x86 object code. Vizer technology can also be used to instrument x86 executables on grid computing environments such as the Grid Application Development Software infrastructure.
Acknowledgements

I would like to thank my advisors Keith Cooper and Ken Kennedy for guiding me throughout this project. The meetings we had during the course of the project were very productive and helpful. My fellow graduate students, Tim Harvey and Todd Waterman, helped immensely in editing and shaping this thesis. Tim suggested the changes in the SSA algorithm to keep track of floating point values on the stack.

I would especially like to thank my parents and sister who have always supported and inspired me to achieve my academic goals.
Contents

Abstract i
List of Illustrations v
List of Tables vi

1 Introduction 1
  1.1 Contributions ........................................ 3

2 Related Work 4

3 Designing a framework for analysis and vectorization of binaries 7
  3.1 Introduction to Vectorization ............................. 7
  3.2 Vectorizing low-level code ................................ 8
  3.3 Disassembling and Parsing the Input ..................... 11
  3.4 Analysis and Vectorization of the Intermediate Representation ... 12
    3.4.1 Analysis of the Intermediate Representation .......... 12
    3.4.2 Transformation Phase: Vectorization and Final Emission of the Code ........................................ 19
  3.5 Experimental Results .................................... 21
  3.6 Complexity ................................................. 23
  3.7 Conclusion .................................................. 24

4 Using Vizer technology to instrument binaries on GrADS 26
  4.1 Introduction ................................................. 26
5 Conclusion

Bibliography
Illustrations

3.1 Overview of Vizer ............................................. 9
3.2 Effects of Pop and Push Operations on the Floating Point Register
   Stack .................................................................. 14
3.3 Definitions of the High-level Entities in Vizer .................. 17
3.4 Some of the High-level Entities Identified in the Example ...... 18
3.5 Algorithm for dead-code elimination .............................. 21

4.1 Overview of GrADS ............................................ 27
3.1 A Simple Vector Addition ........................................... 10
3.2 Execution Time of Benchmarks Before and After Running Vizer (sec) 22
Chapter 1

Introduction

Processor performance has increased at a tremendous rate roughly following the increase in transistor density predicted by Moore's law. Several factors have fueled this surge in CPU performance. The introduction of hardware that exploits instruction-level parallelism, the addition of specialized functional units on processors that target certain types of codes, and the development and evolution of architectural paradigms such as superscalar and very long instruction word architectures have contributed to effective processor performance. However, the mere presence of complex hardware does not guarantee good performance from software. To obtain speedier execution, software applications must exploit these new features present on recent processors. Thus, compiler technology must progress as rapidly as processors to ensure that compiled programs target newer CPUs.

Though using advanced compiler technology to create processor-specific programs can result in a large increase in performance, many application writers frequently trade off performance for portability. Software developers tend to target the lowest common denominator in a processor line to maintain backward compatibility. Newer programs can thus be run on older processors. Consequently, these programs do not make use of additional hardware present on recent processors. While programs can be recompiled to create processor-specialized binaries, software developers are, for a number of reasons, unwilling to do so. The creation of several binaries each targeted at a processor is neither practical nor cost-effective. The large-scale presence of non-specific binaries provides a huge opportunity for binary optimizers. Binary optimizers take as input compiled and assembled programs and output optimized binary code.
As a result, programs that have been compiled for older processors can be modified to utilize effectively the hardware present on newer processors.

A binary optimizer can also be used profitably on a grid computing infrastructure as in the Grid Application Development Software (GrADS). In the 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 resources assigned to execute a binary running on the grid are not known until runtime. Due to the presence of computers with different architectures on the grid, the characteristics of the target resources can vary widely. In such a scenario, a binary optimizer is critical to ensure good performance from the system. A binary optimizer is invoked to conduct dynamic optimization of binaries executing on the grid to fine-tune the program for a particular processor.

In this thesis, we describe the Vizer system which automatically vectorizes object code for the Intel x86 architecture. The Intel x86 architecture is an attractive target for binary optimization because of its widespread use and the regular addition of features to new models in this line. For example, recent Pentiums support 3 types of Single Instruction Multiple Data (SIMD) instructions: Multimedia Extensions, Streaming SIMD Extensions (SSE), and Streaming SIMD Extensions 2 (SSE2) that operate on packed integer and floating point data. At the same time, however, these machines are backward-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 compiled for older x86 processors. Binary optimization offers a way to update these legacy applications so that they make effective 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 reduces execution time for some codes. In addition to vectorizing object code, the infrastructure developed in Vizer has also been extended to analyze and instrument executables. Vizer safely inserts instrumen-
tation in binaries to enable monitoring of applications running on a computational grid. Vizer is being used to instrument executables on the GrADS framework.

1.1 Contributions

This thesis makes two important contributions. First, the thesis describes the development of Vizer, a system that automatically vectorizes Intel binaries. The implementation of Vizer and the encouraging results obtained by running Vizer on benchmarks show that vectorization can be conducted on low-level object code. More importantly, the infrastructure developed by Vizer can be used to recreate the control flow and structures of the original high-level language code. This information is extremely useful to a binary optimizer. The framework developed by Vizer to extract high-level structures can be used not only for vectorization but also for numerous other optimizations that are also not traditionally conducted at the binary level. Secondly, the thesis describes the extension of Vizer’s abilities to instrument executables in a grid computing environment such as GrADS. Vizer technology is currently being used in GrADS to enable the monitoring of programs running on the system. Once instrumented by Vizer, the program reports execution metrics which are used to ensure that the executable meets certain performance contracts.
Chapter 2

Related Work

While binary translators have been profitably used in the past, they have gained importance in the last few years due to the widespread use of legacy binary code and Java programs. Presently there are a large number of closed-source and legacy applications available without readily accessible source code. Many of these binaries were compiled for older processors to maintain backward compatibility. Binary optimizers can specialize these applications to run on recent processors. Executable-level optimizers have also become more prevalent due to the widespread use of Java bytecode. Machine-independent Java bytecode can be written on one hardware platform and run on another. Though this results in the creation of a write-once-run-anywhere binary, the bytecode cannot contain any architecture-specific feature. To increase the performance of such code, just-in-time (JIT) compilers conduct dynamic optimization of Java binaries to ensure that frequently executed code is recognized and optimized for the target machine. Binary optimizers can be roughly classified into three major types:

**Binary translators** Binary translators convert compiled applications from one architecture to another. This can be done statically or at runtime. UQBT is a research project that develops tools to translate programs between different machine environments [10]. Based on the provided specifications of machines and operating systems, UQBT tools produce a machine-independent representation of a binary that can then be ported to another system. Several other projects conduct dynamic translation of binaries between operating systems allowing the end user to run applications compiled for any operating system irrespective of the environment present on his or her
computer. For instance, VMWare provides a mechanism to run Windows executables on Linux machines and Linux executables on Windows machines [29]. Wine is an open-source project that provides an interface for Windows applications to run on an Unix system [31]. A number of other translators convert binaries between systems [24, 32, 20].

**Static optimizers and analyzers**  Several systems provide mechanisms to statically analyze, instrument, and transform binaries. In contrast to binary translators described in the previous paragraph, the primary focus of these tools is to optimize binary code, staying on the same platform. Etch, a system to analyze Windows executables, allows the user to write tools which monitor key features of the program during runtime [23]. The user can also use the framework provided by the system to conduct optimizations by rewriting the executables. The IMPACT binary reoptimization framework optimizes Windows executables by generating an intermediate format called Mcode that can be used to transform the code [27]. The Executable Editing Library (EEL) is a similar system for Solaris and SunOS executables [18]. EEL generates abstractions in the form of C++ classes that can be used to manipulate the program. OM is a system for code optimization for MIPS binaries [26]. Unlike IMPACT and EEL, OM analyzes object files and libraries instead of executables. 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 [25].

**Dynamic optimizers** While transforming binaries after compilation can be beneficial, it is sometimes necessary to optimize executables at runtime. Dynamic optimizers can analyze and optimize running programs. Dyninst is a tool to manipulate programs while they are executing [8]. It provides a machine-independent interface that allows users to modify executables and insert instrumentation into the binary at runtime. The popularity of the Java programming language has led to much re-
search being conducted on extracting good performance from JIT compilers. JITs such as Java HotSpot and Latte optimize frequently executed code to increase performance [19, 34]. The Jalapeno Java Virtual Machine compiles Java bytecode into native machine instructions at runtime to increase performance on Symmetric Multi-Processor machines [9]. Other systems such as Dynamo and Transmeta’s Code Morphing technology dynamically modify code for faster execution [3, 16].

These systems address many of the issues that arise in manipulating object-level and binary representations of programs. They do not, however, perform high-level transformations such as automatic vectorization on the low-level code. Vectorization is typically conducted on a near-source representation of the code because it requires the analysis of data structures and control flow not easily identifiable in low-level binaries. 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. Vizer’s framework can also be used to conduct other high-level transformations on object files and executables. Another application for Vizer is instrumentation of executables. As will be described in Chapter 4, Vizer’s infrastructure is currently being used to instrument binaries executing on the GrADS environment.
Chapter 3

Designing a framework for analysis and vectorization of binaries

3.1 Introduction to Vectorization

Vectorization is a technique to conduct arithmetic operations on several operands simultaneously. Early supercomputers introduced hardware which made it possible to conduct element-wise operations on two vector registers. Nowadays, vector hardware is present in a number of consumer-level processors. For instance, the Intel x86 line of processors contain vector instructions called Streaming SIMD Extensions and Multimedia Extensions. Other modern architectures have similar features. 3D Now! technology present in AMD processors, PowerPC's AltiVec instructions, and the vector units in the PlayStation2's Emotion Engine CPU all implement vector operations. Vector hardware has largely become popular due to the rise in usage of multimedia applications. Since many multimedia programs conduct the same operation on streaming data, this provides an ideal setting for the use of SIMD vector hardware.

In spite of the hardware support for vector instructions, most programs are written in high-level languages which do not allow the programmer to specify vector operations directly. Such languages maintain a high level of abstraction and do not want the programmer to be aware of the specific features present in the target architecture. As a result, to obtain good performance, compilers have to seek out sequential scalar loops and attempt to vectorize them. Before transforming scalar loops to a vectorized form, the compiler must conduct an analysis of the loop to ensure the safety of transformation. The vectorizer must ensure that the results after vectorization
are consistent with the results produced by the sequential loop. To guarantee safety, the vectorizer must conduct a dependence analysis on the unoptimized code. The presence of certain dependences, called vectorization-preventing dependences provide an impediment to vectorization. Consider the following loop which illustrates the problem vectorizers face due to dependences:

\[
\text{DO } I = 2, 50 \\
\]

\text{ENDDO}

If this loop is transformed without analyzing dependences, it may be naively vectorized as:

\[
\]

where the vectorized loop is written in Fortran-90 syntax. However, semantically, the scalar and the vectorized loop differ. The array \( A \) present in the left hand side of the vectorized loop reads in the old values of \( A \). In contrast, the scalar loop accesses the updated values of \( A \). Thus the loop cannot be trivially vectorized as the transformation will be unsafe. If a dependence analysis is conducted on the scalar loop before transformation, then the vectorizer can realize that due to the true, loop-carried dependence present from \( A[I] \) to \( A[I - 1] \) in the scalar loop, it cannot directly vectorize the loop. This dependence is an example of a vectorization-preventing dependence. If such dependences are discovered, the vectorizer can attempt to transform the loop or reschedule the array accesses to make the loop vectorizable. Loop distribution, loop peeling, and loop reversal are a few common transformations that may enable the vectorization of a scalar loop.

3.2 Vectorizing low-level code

Although vectorization of programs written in higher-level languages has been well studied [2, 21, 17], vectorizing low-level assembly code poses a number of additional
challenges. A source-level 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 Table 3.1. 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 3.1: Overview of Vizer](image)

Figure 3.1 shows a high-level view of Vizer. Vizer takes as input one or more object files in the Executable and Linkable Format (ELF) [28]. Vizer can also analyze executables as will be discussed in Section 4.3. 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
```c
#include <stdio.h>

.globl fillArray_FPi
.globl main

void fillArray(int* array, int N) {
    for(int j = 1; j < N; ++j) {
        array[j] = array[j-1] + 33;
    }
}
```

<table>
<thead>
<tr>
<th>C++ Source Code</th>
<th>Assembly File Recreated by Vizer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fillArray_FPi:</td>
</tr>
<tr>
<td></td>
<td>.0: push %ebp</td>
</tr>
<tr>
<td></td>
<td>.1: mov %esp,%ebp</td>
</tr>
<tr>
<td></td>
<td>.3: sub $0x8,%esp</td>
</tr>
<tr>
<td></td>
<td>.6: movl $0x1,0xfffffffff8(%ebp)</td>
</tr>
<tr>
<td></td>
<td>.d: lea 0x0(%esi),%esi</td>
</tr>
<tr>
<td></td>
<td>.10: mov 0xfffffffff8(%ebp),%eax</td>
</tr>
<tr>
<td></td>
<td>.13: cmp 0xc(%ebp),%eax</td>
</tr>
<tr>
<td></td>
<td>.16: jle .1c</td>
</tr>
<tr>
<td></td>
<td>.18: jmp .44</td>
</tr>
<tr>
<td></td>
<td>.1a: mov %esi,%esi</td>
</tr>
<tr>
<td></td>
<td>.1c: mov 0xfffffffff8(%ebp),%eax</td>
</tr>
<tr>
<td></td>
<td>.1f: imul $0x4,%eax,%ecx</td>
</tr>
<tr>
<td></td>
<td>.22: mov 0x8(%ebp),%edx</td>
</tr>
<tr>
<td></td>
<td>.25: mov 0xfffffffff8(%ebp),%eax</td>
</tr>
<tr>
<td></td>
<td>.28: mov %eax,%eax</td>
</tr>
<tr>
<td></td>
<td>.2a: shl $0x2,%eax</td>
</tr>
<tr>
<td></td>
<td>.2d: add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td></td>
<td>.30: sub $0x4,%eax</td>
</tr>
<tr>
<td></td>
<td>.33: mov (%eax),%eax</td>
</tr>
<tr>
<td></td>
<td>.35: add $0x21,%eax</td>
</tr>
<tr>
<td></td>
<td>.38: mov %eax,(%edx,%ecx,1)</td>
</tr>
<tr>
<td></td>
<td>.3b: lea 0xfffffffff8(%ebp),%eax</td>
</tr>
<tr>
<td></td>
<td>.3e: inc %eax</td>
</tr>
<tr>
<td></td>
<td>.40: jmp .10</td>
</tr>
<tr>
<td></td>
<td>.42: mov %esi,%esi</td>
</tr>
<tr>
<td></td>
<td>.44: leave</td>
</tr>
<tr>
<td></td>
<td>.45: ret</td>
</tr>
</tbody>
</table>

Table 3.1: A Simple Vector Addition

opportunities for vectorization in the IR. The transformation phase takes advantage of opportunities that the analysis identifies by inserting appropriate Streaming SIMD instructions. It also emits the final optimized object file.
3.3 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 [5]. Objdump translates the object code into a series of code and data sections, along with relocation information. It also disassembles the code sections.

The output of objdump is a text file with sufficient detail to let a user read the code. However, it lacks information such as labels and procedure calls, so the file cannot be re-assembled. Vizer needs a file that it can parse into a useful IR and, eventually, re-assemble into object code. Thus, it rewrites the output of objdump into a valid assembly language program.

Using the additional information provided by objdump, Vizer recreates correct assembly code. In particular, Vizer uses relocation information to fix call labels and loads of constant data into registers. Data from disassembled data sections are also added to the objdump disassembly. For instance, the load of a constant from the data section may be disassembled by objdump as:

\[
\text{mov } 0x8, \%eax
\]

Since the first operand is an absolute address, it is changed to an appropriate relative offset so that the file containing the instruction can be assembled and linked correctly. Vizer uses relocation information to deduce where the operand is located in the data section and changes the instruction appropriately. After modifying the instruction, the output from Vizer may look like:

\[
\text{mov .data+0x8, \%eax}
\]

Similarly, Vizer changes the operands of procedure call instructions to point to the correct procedure name. The output after Vizer modifies the assembly is a file containing legal assembly code. This file is then used as an input into the parser. The
parser used in the GNU Assembler was modified for Vizer so that it emits a list of structures containing the parsed instructions. The new parser also maintains critical information about labels and pseudo-instructions. This list of structures forms the IR used by Vizer’s later stages.

3.4 Analysis and Vectorization of the Intermediate Representation

3.4.1 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.

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 dependencies [2]. Using this information, it converts the code into minimal Static Single Assignment form (SSA) [13, 7]. 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 SSA construction algorithms assume that all definitions and
uses are explicit and visible. Some x86 instructions operate on registers that are not explicit 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 also assume, implicitly, that operations are non-destructive. Destructive 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 [15]. 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.
<table>
<thead>
<tr>
<th>A push from memory onto the stack</th>
<th>A pop from the stack into memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fld -8(%ebx)</code></td>
<td><code>fstp -8(%ebp)</code></td>
</tr>
<tr>
<td>turns into 7 pseudo-copies:</td>
<td>turns into 7 pseudo-copies:</td>
</tr>
<tr>
<td><code>psmov st(6) =&gt; st(7)</code></td>
<td><code>psmov st(0) =&gt; -8(%ebp)</code></td>
</tr>
<tr>
<td><code>psmov st(5) =&gt; st(6)</code></td>
<td><code>psmov st(1) =&gt; st(0)</code></td>
</tr>
<tr>
<td>...</td>
<td><code>psmov st(2) =&gt; st(1)</code></td>
</tr>
<tr>
<td><code>psmov st(1) =&gt; st(2)</code></td>
<td>...</td>
</tr>
<tr>
<td><code>psmov -8(%ebp) =&gt; st(0)</code></td>
<td><code>psmov st(7) =&gt; st(6)</code></td>
</tr>
</tbody>
</table>

| **Effect of a Push** | **Effect of a Pop** |

Figure 3.2: Effects of Pop and Push Operations on the Floating Point Register Stack

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 3.2 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 explicitly 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 of translation. Thus, Vizer can safely insert the copies to simplify analysis, without having them appear in the final code.
Extracting High 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, effectively, hidden. As shown in Figure 3.1, 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, including some abstract version of the data structures. 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 [2,
If the induction variables are held in registers, they can be identified by analyzing the SSA graph and the operations in the loop [12]. 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 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.¹ 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 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 3.3 shows the rules used to classify operands as various kinds of high-level

¹For example, the g++ compiler often generates symbolically equivalent induction variables.
<table>
<thead>
<tr>
<th>Definition of High-level Entity</th>
<th>Potential High-level Construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>StackLocation := (&lt;All\ exprs\ pointing\ to\ some\ location in\ the\ stack&gt;)</td>
<td></td>
</tr>
<tr>
<td>StackValue := DeRef(StackLocation)</td>
<td></td>
</tr>
<tr>
<td>InductionVar := PrimaryInductionVar</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SecondaryInductionVar</td>
</tr>
<tr>
<td></td>
<td>&lt;InductionVar From Outer Loop&gt;</td>
</tr>
<tr>
<td>PrimaryInductionVar := (&lt;Value\ either\ in\ register\ or\ stack\ incremented\ on\ each\ iteration\ and\ controls\ the\ exit\ condition\ of\ the\ loop&gt;)</td>
<td></td>
</tr>
<tr>
<td>SecondaryInductionVar := Constant * PrimaryInductionVar</td>
<td></td>
</tr>
<tr>
<td>ArrayPointer := StackValue</td>
<td>Array[0]</td>
</tr>
<tr>
<td></td>
<td>StackLocation</td>
</tr>
<tr>
<td></td>
<td>ArrayPointer Op Constant</td>
</tr>
<tr>
<td></td>
<td>DeRef(ArrayPointer Op InductionVar)</td>
</tr>
<tr>
<td>(K = Constant / (sizeof(Data) / Addressable\ Size))</td>
<td></td>
</tr>
<tr>
<td>ArrayValue := ArrayPointer (except StackLocations)</td>
<td>Array[i]</td>
</tr>
<tr>
<td></td>
<td>ArrayPointer Op Constant</td>
</tr>
<tr>
<td>ArrayReads := DeRef(ArrayPointer + InductionVar)</td>
<td></td>
</tr>
<tr>
<td>ArrayWrites := DeRef(ArrayPointer + InductionVar)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.3: Definitions of the High-level Entities in Vizer

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 3.4: Some of the High-level Entities Identified in the Example

Figure 3.4 shows a subset of the high-level entities that Vizer finds in the assembly code from Figure 3.1.
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 memory 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. More complete dependence testing 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.

3.4.2 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
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.

**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.
Procedure for Dead-code elimination for loop L:
Mark original scalar store as dead
changed = false
do {
    for all instructions I in loop L
        if I defines a value V and all uses of V are dead
            Mark I as dead
            changed = true
        endif
    endfor
} until (changed = false)

Figure 3.5: Algorithm for dead-code elimination

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 corresponding 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 to remove redundant scalar operations. As shown in Figure 3.5, a variant of the algorithm described in [13] is used to eliminate dead instructions. The algorithm that Vizer uses conducts conservative dead-code elimination by marking the scalar store as dead. It then uses the SSA def-use chains to identify resulting dead instructions until a fixed point is reached. After conducting dead-code elimination, Vizer uses the GNU Assembler to produce the final, optimized, vectorized object file.

3.5 Experimental Results
To assess the effectiveness of Vizer, it was tested on a number of benchmarks. Because it does not perform the traditional loop transformations that enhance opportunities
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original Runtime</th>
<th>After Vizer Vectorizes</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Addition</td>
<td>3.503</td>
<td>1.593</td>
<td>54.53%</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>3.680</td>
<td>1.700</td>
<td>53.53%</td>
</tr>
<tr>
<td>Erlebacher</td>
<td>20.463</td>
<td>18.280</td>
<td>10.67%</td>
</tr>
</tbody>
</table>

Table 3.2: Execution Time of Benchmarks Before and After Running Vizer (sec)

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 3.2. The Erlebacher benchmark written by Thomas Edison at ICASE consists of around 600 lines of FORTRAN code which performs 3-dimensional tridiagonal solves [14]. 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.

For these benchmarks, Vizer was able to analyze the object-code, abstract out high-level structure, identify potentially vectorizable statements, and emit a safely vectorized version of the benchmark. Vectorization requires the analysis of structures that are not explicit in binary files. These tests showed that Vizer was effectively able to extract the data structures and control flow that were present in the source code used to create the binaries. Vizer could use the high-level model of the code that it created to implement an optimization that is not typically conducted at the object-code level. The benefits of conducting such a high-level optimization on the benchmark binaries are evident from the results. 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. The results emphasize that high-level transformations can be profitably conducted on binary code.

3.6 Complexity

Vizer's analysis and transformation phase dominates its runtime. Building the IR takes time linear in the size of the input object file. Constructing the CFG, identifying loops, induction variables, stack locations, and array accesses each take time that is linear in proportion to the size of the input. However, some components of the analysis and transformation phases iterate over the IR several times to obtain the necessary high-level information. In the description of these components given below, the number of nodes in the CFG is denoted as N.

- Computing dominators and SSA: Dominators are computed by using the rapid data-flow framework described in [11]. This information is then used to build minimal SSA. The SSA construction algorithm takes \( O(V \times N^2) \) time to terminate where \( V \) is the number of names in the CFG before SSA is constructed. Since the names in the CFG correspond to registers in the code, \( V \) is a constant and cannot exceed the number of registers on the Intel x86 architecture. Thus SSA is calculated in \( O(N^2) \) time.

- Identifying High-level entities: The nodes in the identified loops are analyzed to identify high-level entities. An iterative algorithm is used to locate these entities. A single operand can be identified as multiple high-level entities. If there are \( K \) high-level entities defined (where \( K \) is a small constant), this implies that in the worst case every operand in the loop will be identified as \( K \) high-level
entities. In the degenerate case, it is also possible that each pass only identifies one such entity. There can however be at most 3 operands per instruction. Thus, if there are \( L_i \) nodes in the \( i^{th} \) loop, the algorithm can make at most 3 * \( L_i \) * K passes through the loop. During each of these passes, the algorithm considers \( L_i \) nodes. As a result, for all loops, this step will take \( O(\sum(L_i^2)) \) time. Note that since \( \sum(L_i^2) \leq (\sum L_i)^2 \), and \( \sum L_i \leq N \), this step takes time \( O(N^2) \) time.

- Vectorizing the code: Each operand in a vectorizable statement is loaded into a vector register. There are 8 SSE registers that can be used by Vizer. Hence, Vizer can schedule only a constant number of load and arithmetic operations. The transformation phase considers each vectorizable loop nest and inserts vector instructions. This phase thus takes \( O(M) \) time, where \( M \) is the number of inner loops in the code.

- Dead-code elimination: The algorithm shown in Figure 3.5 is implemented to remove dead code. This algorithm takes \( O(\sum(L_i^2)) \) time in the worst-case which is \( O(N^2) \).

As can be seen from the runtime analysis, calculating SSA, identifying high-level entities, and dead-code elimination prove to be the most time-consuming tasks for Vizer. Each of these phases take \( O(N^2) \) time. Vizer's worst-case running behavior will thus be quadratic in the size of the input.

### 3.7 Conclusion

Conducting certain source-level optimizations at the binary-level requires substantial information about the high-level characteristics of the binary. The control flow and data structures that were once present in the source code are lost when the program is compiled and assembled. This information must be recalculated by analyzing the
binary. As shown in the experiments, Vizer can successfully provide a high-level perspective of the object file. This description of the machine-level instructions enables an optimizer to apply transformations that are traditionally not conducted on binaries. Vizer demonstrates the benefit of conducting high-level optimization on binary codes by implementing a vectorizer.

The experimental results show that Vizer, in its current form, can identify and exploit simple vectorization opportunities in object code. The potential for improvement is significant; Vizer produced significant improvements in compiled programs. The current version of Vizer is a prototype that misses many opportunities for vectorization. Vizer’s goal in designing the vectorizer was to prove that vectorization is possible on binaries. While the experiments underline that the goal was achieved, Vizer’s vectorization capabilities could be enhanced in a number of ways. More sophisticated symbolic analysis and dependence testing could be added to Vizer. Also, the addition of loop distribution and loop peeling in Vizer will expose more opportunities for vectorization.

Importantly, Vizer’s framework of extracting high-level information from binaries can be effectively used to implement other optimizations that are not conducted on binary code. Some of the candidates for implementation include lazy code motion, loop unrolling, enhanced scalar expansion, 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.
Chapter 4

Using Vizer technology to instrument binaries on GrADS

4.1 Introduction

The GrADS project aims to provide a simple and efficient way to use distributed heterogeneous computing [4]. GrADS comprises of several components, as shown in Figure 4.1. The front-end, the Problem Solving Environment (PSE), accepts programs written by users in various high level languages. These programs are then compiled by the program preparation system to a configurable object format. The GrADS execution environment monitors the status of resources on the entire system and schedules the object files to be executed on various computing systems on the network. Before scheduling the object files, the files are converted into native executables for the target machines. A dynamic optimizer fine-tunes the executables according to the specifications of the particular machine.

A program running on GrADS is monitored at various stages of execution to ensure that the performance contracts for the executable are being satisfied. If the execution environment realizes that a contract cannot be met, it can relocate the program to other resources available on the grid.

4.2 Instrumentation on GrADS

The GrADS system is under development. In its current state, the program running on the GrADS infrastructure is compiled to an ELF executable. To correctly monitor the program, instrumentation calls must be inserted into the file. Autopilot, a
system to dynamically observe and optimize programs running on heterogeneous computational grids, is used to monitor the performance of the GrADS executable [22]. Autopilot extracts performance metrics from the execution and reports these measures back to the contract monitor. The binary created by the user of the grid does not have the necessary calls to Autopilot. Hence, the binary has to be rewritten with the needed calls to Autopilot, along with Autopilot initialization and shutdown code.

Vizer technology is used to modify the uninstrumented binary and insert the necessary hooks into Autopilot code. Since Vizer was originally designed to analyze object files, the front-end of Vizer was extended to disassemble and analyze ELF executables. Using the information gathered by the front-end, Vizer analyzes the executable to detect the correct insertion points for the calls to Autopilot.
4.3 Changes required in Vizer to analyze executables

The extraction of relocation information provides the major challenge to analyzing executables. While vectorizing x86 object files, Vizer had to extract the relative offset of operands given absolute addresses. This information was encoded in the object file — every absolute address was annotated with the offset from the corresponding data section. However, in ELF executables, the offset information is not provided along with the operand. The linking process for the executable changed all relative addresses to absolute addresses. As a result, the disassembler only provides the name of the data section corresponding to the absolute address. Vizer needs to change such operands into an offset from a data section to produce a legal assembly file. Vizer analyzes the disassembled data sections that contain a table of absolute addresses along with the data stored at those addresses. Using this information, Vizer calculates the offset for every absolute address and replaces those addresses with the computed relative addresses.

4.4 Using Vizer on GrADS applications

Vizer was tested and used on three GrADS applications: ScalaPACK, fasta, and Cactus. ScalaPACK is a library to solve systems of linear equations, least squares problems, and eigenvalue problems [6]. Fasta is a biological application that computes the similarities between protein and DNA sequences [30]. Cactus implements a problem solving environment for a computational grid to solve large scale scientific problems [1]. These applications were complex programs designed to demonstrate the effectiveness of the GrADS infrastructure in executing scientific programs on the grid. Compiling and assembling these programs yielded an executable that was then given as input to Vizer. Vizer was able to successfully analyze the binary, detect insertion points for instrumentation code, and instrument all three applications.
4.5 Conclusion

The framework developed by Vizer to vectorize binaries can be profitably used to analyze Intel x86 ELF executables. Vizer creates abstractions from executables that can serve as a basis for instrumenting the binary. The insertion of instrumentation allows the capture of runtime information about the program while it executes on the grid. Vizer has successfully instrumented three reasonably complex scientific applications. This shows that the Vizer system can be used not only to vectorize Intel binaries but also to insert instrumentation into executables. Distributions of many legacy programs contain the executable files but not the corresponding object code. Extending Vizer's abilities to analyze executables enables it to potentially optimize a wider selection of binaries.
Chapter 5

Conclusion

The role of binary optimizers has become increasingly important due to the presence of legacy code and additional features in new processors. Analyzing and optimizing binaries can be profitable in scenarios where the source code for the program is inaccessible. In this thesis I presented Vizer, a binary optimizer for Intel x86 binaries.

Vizer implements an infrastructure to disassemble, analyze, and optimize x86 object files and executables. Vizer’s analysis yields a high-level perspective of the binary code. This information is critical to implementing transformations that are not typically conducted on object-code and executables. In contrast to other binary optimizers, the model that Vizer provides can be utilized to apply source-code level optimizations. Vizer implements one such optimization — it vectorizes object code. Vizer identifies vectorization opportunities, conducts dependence analysis, and safely vectorizes the code to increase performance. Our experiments show that vectorization is possible and profitable for machine-level code. Vizer is effective in decreasing execution time for binaries.

The underlying framework developed by Vizer can also be used to implement other high-level optimizations and modifications to binary files. We have shown that Vizer can be used to instrument binaries executing on the GrADS computing grid. Vizer has successfully instrumented a number of scientific applications running on GrADS.
Bibliography


