Towards Identifying and Monitoring Optimization Impacts *

Thomas P. Way *Lori L. Pollock*

High Performance Computing Software Laboratory
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716

{*way,pollock*}*@cis.udel.edu*

Abstract

Optimizing compilers apply code-improving transformations in phases over a source program in an effort to emit the fastest or most compact executable code possible. The effectiveness of these optimizations is limited by phase-ordering problems, manifested as interactions among optimizations and the attending impact on system resource utilization. Incorporating into each optimization module an awareness of its effect on resources, program characteristics, and other optimizations can lead to better code than the typical iterative phase-based approach.

This paper investigates the monitoring of these complex interactions, particularly as they apply to architectures with instruction-level parallelism (ILP). The paper examines the issues regarding how to determine what type of information adequately reflects the effect of individual optimizations on individual as well as the entire system of optimizations, and how to collect and quantify this information. A general framework for constructing an optimization interaction monitor and the design of a prototype tool based on this framework are described. Implications of using this monitoring tool in research directed towards the development of a tunable optimizing compiler for ILP architectures are examined.

1 Introduction

Programmers rely on the optimizing compiler to perform code-improving transformations and translate a source program into executable code that exhibits improved performance, smaller size, or both. The automatic application of these code-improving transformations, or *optimizations*, allows the programmer to code at a more intuitive, high level of abstraction while still exploiting the characteristics of the underlying machine architecture. The importance of this conceptual decoupling of the program and machine has increased steadily along with the complexity of computer architectures, the size of program codes, and the desire for higher performance.

Instruction-level parallelism (ILP) is a key architectural feature of high performance microprocessors [39], and is the focus of much active research in compiler optimization. Beyond the classical machine independent optimizations, program performance on architectures with ILP has been improved by many well-known techniques such as aggressive global instruction scheduling, software pipelining, speculative code motion, static branch prediction schemes, sophisticated dependence and aliasing analysis to expose more parallelism, increasing locality for effective cache use, hiding memory access latencies, careful register allocation, profile-based optimizations, and optimizations to take advantage of predicated execution. We refer to all of these as optimizations.

There is no question that performance gains are regularly attained experimentally and in production compilers with these optimizations. Less understood is what further performance gains might yet be achieved given an application, the target architecture, and the elusive answers to a number of open questions, such as:

1. Which optimizations should be applied?
2. Where in the code should a particular optimization be applied? and
3. What is the sequence in which these optimizations should be applied?

While easy to state, these questions are quite difficult to address for a number of reasons, including:

*Appeared in Proceedings of the 3rd Annual Mid-Atlantic Student Workshop on Programming Languages and Systems (MASPLAS), East Stroudsburg University in cooperation with ACM SIGPLAN, 1997.

- The goals of optimizations span the enhancement, exposure, and exploitation of ILP, cache use, register use, memory use and the overall complexity and quantity of computation. *Often, optimizing for one goal conflicts with optimizing for another goal.*

- *Optimizations are usually performed over a whole program or procedure as a phase, with the ordering of these phases determined by the compiler writer based on some heuristic.* Applying optimizations in more flexible combinations and orderings on variable-sized code partitions would allow for tailoring the optimization of a region to compensate for detrimental effects that it suffers due to optimization of nearby regions.

- The application of one optimization after another is typically predetermined by some heuristic which guides the optimization strategy in a general sense. However, *earlier optimizations can unpredictably prevent or enable the safety and effectiveness of later ones in the sequence.*

- *The effects of optimizations are architecture-dependent.*

- *Optimizations vary in their effect depending on program characteristics.* Thus, the best strategy for performing optimizations can vary from application to application, and even from region to region within the same application.

- *Interactions of optimizations performed on varying levels of representation of the code are not well understood.*

Currently, information about the impact of optimizations on system resource utilization, and therefore on subsequent optimizations, is rarely gathered and exploited in the optimizer's strategy. There are isolated instances where this information is used to good effect, such as when combining instruction scheduling and register allocation [3, 5, 6, 19, 20, 30, 33, 34], or software pipelining and register allocation [16, 17, 21, 23, 27, 32, 38, 44]. While these techniques can improve program performance, they focus narrowly on the interaction of a single pair of optimizations, rather than more generally on the entire collection of optimizations to be applied to a program.

Provided that enough useful information can be gathered and analyzed regarding the interactions of optimizations and their effects on resources, it might be possible to find satisfactory answers to the above questions which would lead to more robust and powerful code optimizers. To this end, we are exploring the design of a general framework for systematically incorporating information about the impact of optimizations throughout the optimization process, which encompasses the optimization phases from source code to target code. This framework will be used in the development of more powerful optimizers that have the capability to dynamically tune the optimization strategy by exploiting information about the impact of optimizations.

The capability to tune requires identifying the relative importance of resources and the impact of transformations on these resources, program characteristics, and other optimizations. The first step in the development of this system is the identification of information that reflects the impacts of optimizations on other optimizations, the resources of the target machine, and the program characteristics. An important part of this step involves the development of a technique for collecting and analyzing this information. Not only do the impacts of individual optimizations have to be considered, but also the cumulative impact of applying a sequence of optimizations. In order to be useful, this information must be quantified to enable its use in the evaluation of optimization impacts.

This paper describes our work in monitoring the complex interactions of optimizations. In particular, the remainder of this paper explores issues regarding the kinds of information to collect, methods for collection, a framework for monitoring optimization interactions, and the design of a prototype optimization interaction monitoring tool based on this framework. This tool will be used to investigate the complex interactions of optimizations with each other, program characteristics, and system resources of ILP architectures. Ultimately, the contribution of the implementation of this monitoring tool and the accompanying research will be to support efforts to develop a tunable optimizing compiler for ILP architectures.

2 Measuring the Impacts

Before designing a framework and tool for monitoring the interactions of optimizations, it is necessary to (1) identify the kinds of information that are potentially beneficial, (2) determine how this information can be quantified and represented, and (3) develop methods for gathering the information. Some isolated progress on each of these fronts has been made in different corners of compiler optimization research. That is, we can draw from the individual experiences of those examining the interactions of specific phases and developing integrated approaches, those combining specific machine-independent optimiza-

tions, those unifying sets of transformations into a single mechanism, and the use of profiling information in optimization decision-making. In this section, we summarize the aspects of these various efforts relevant to the goal of monitoring optimization impacts.

Based on the enabling conditions for individual optimizations and the previous work on optimization interactions, we can classify the kinds of information that will be useful to monitor the impacts of optimizations into three categories:

- machine resource utilization,
- program characteristics, and
- performance characteristics.

The goal of an optimization often can be stated in terms of its improvement in the use of some architectural resource. Thus, some of the impacts of an optimization can be determined by identifying changes in the utilization of resources of primary concern to other optimizations. Machine resources can be represented as architectural characteristics. Examples include instruction costs, register set size, cache size and configuration, memory access costs, and high-performance architectural features such as ILP.

Another aspect of the impact of an optimization is its effect on changing the code so that other optimizations are either enabled or disabled. That is, the conditions for applying another optimization can either become true or false, due to performing an optimizing transformation to the code. Similarly, program characteristics can also be used to measure the use of machine resources. An example is using live variables to measure the demand for registers. Another example is data dependences to measure the restrictions on parallelism in the code, and the subsequent limitations on exploiting available parallelism in the architecture. Program characteristics of interest include data and instruction locality, branching likelihoods, live ranges of variables, data and control dependences, loop structures, algorithmic efficiency, coding style, and inherent parallelism.

Another measure of the impact of optimizations is how the code actually performs given a sequence of optimizations. This information can be gathered through code instrumentation to profile the execution of the resulting code, and analysis of the run-time performance information. Performance characteristics to monitor include a program execution profile, memory and cache utilization patterns, and communication costs.

In the next few subsections, we examine more closely the relevant experiences in gathering, quantifying, and using the three classes of information for compiler optimization decision-making.

2.1 Machine Resources

Information about machine resources has been utilized mainly for two distinct purposes: (1) to enable cooperation between two specific phases of optimization, and (2) to enable easy retargeting of an optimizer, flexibility in optimization ordering, and use of machine-specific information in higher-level optimizations.

To enable cooperation between pairs of phases, the phases are designed with the goal of maintaining some kind of balance among the levels of demand for specific machine resources of particular interest to the two phases, and the supply and configuration of the target machine's resources. The most well known examples of this work focus on the interactions between software pipelining register allocation [16, 17, 21, 23, 27, 32, 38, 44], instruction scheduling and register allocation [3, 5, 6, 19, 20, 30, 33, 34], instruction scheduling and cache usage [28], and scalar replacement and register allocation [8]. All have in common the goal of creating a good match between the program characteristics, such as instruction placement and register usage, and architectural features such as the availability of registers, memory access overhead, cache usage costs, and the parallelism of the target machine. They attempt to find a good match through cooperation between the two phases in the way that they utilize the resources, rather than greedily using one resource in order to optimize the use of another resource. From these efforts, we can derive what kind of information about machine resources is useful in monitoring optimization impacts for the specific optimization phases already examined for interactions.

To enable easy retargeting of an optimizer, flexibility in optimization ordering, and use of machine-specific information in higher-level optimizations, specific information about the target architecture in the form of a machine description is used throughout optimization [2]. This work is relevant to our tasks of presenting a machine description of the machine resources to the optimizer, and using that machine resource information throughout optimization to increase the effectiveness of each optimization.

2.2 Program Characteristics

For the task of monitoring program characteristics as an impact of an optimization, we can learn from the work in combining optimizations and in unifying sets of transformations into a single mechanism. In both approaches, the methods take into account the interactions of two or more optimizations.

There have been several research groups examining key classical, machine-independent optimizations with the goal of determining whether combining these opti-

mizations into a single pass can lead to more efficient generated code than separate phases, and if so, developing techniques for achieving this combination. Partial redundancy elimination (PRE) is a global optimization introduced by Morel and Renvoise [31] and extended and modified by several others [7, 10, 13, 14, 26]. PRE combines common subexpression elimination, loop invariant code motion, and code motion to move code from frequently used paths to less frequently used paths.

Another example of combining optimizations is global value numbering (GVN) [10, 40], which attempts to replace a set of instructions that each compute the same value with a single instruction, by combining constant folding and propagation, static single assignment, dead code elimination, loop invariant code motion, and common subexpression elimination. Click and Cooper [11] present a framework for combining constant propagation, value numbering, and unreachable code elimination, and describe how to reason about the properties of the resulting framework. In particular, they discuss how the framework for describing optimizations can indicate whether combining the optimizations will be profitable. Other efforts include combining cache and ILP [9] and combining loop transformations [25].

There have been several efforts toward unifying transformations into a single mechanism, and applying search techniques to the transformation space. In particular, one framework for unifying loop transformations is based on unimodular matrix theory [41, 47]. Each transformation of the loop is encoded in a matrix, and applied to the dependence vectors of the loop; the form of the vector obtained by multiplying the matrix and the vector reveals whether the transformation is legal, and the effect of applying a sequence of transformations is determined by the product of the corresponding matrices. Algorithms for handling different kinds of loop nests and loop transformations have been developed by others [24, 36, 37, 42].

Whitfield and Soffa [46] developed a framework that unifies the specification of classical and parallelizing optimizations with the goal of examining the interactions between transformations and aiding in the ordering of optimization phases. The framework is based on an axiomatic specification of optimizations using program code to represent state, preconditions to represent the conditions for safely applying an optimization, and postconditions to represent the effect of applying the optimization to the state. Using these specifications, actions that enable or disable conditions for specific optimizations can be identified. The enabling and disabling conditions determine the interactions between optimizations, which can be used as guidelines for ordering the optimizations. The framework and the study of interactions of optimizations focuses strictly on the safety of optimizations after other optimizations, and examines the interactions on a pairwise basis.

In summary, the program characteristics that are used by these previous efforts for dealing with optimization interactions include: dependence analysis based on the *Static Single-Assignment* (SSA) form [10, 11, 12], assigning costs to assignment statements to guide elimination of partial redundancies [26], and characteristics of loop structures [25] such as nesting level and dependences. The more unified approaches take a broader look, considering the use of an axiomatic scheme to represent and analyze characteristics [46] and a matrix-based theory of representation and analysis [41, 47]. Most of the program characteristics that are used are gathered through static program analysis typically performed by an optimizing compiler.

2.3 Performance Characteristics

An execution profiler is a very common tool that is used in research and industry to assist in optimizing program code. Typically, the programmer compiles the program with some form of code-instrumentation flag turned on. Upon running this instrumented code, a file of profile information about the program execution is produced. Among the most useful information in this file is the actual time and percentage of total execution time utilized by each function. The programmer uses this information to guide efforts to rewrite either the source or assembly code, making the algorithms that are used in the program more efficient. This method is usually very successful, but requires a great deal of proficiency in algorithm design and a strong understanding of the features of the underlying architecture.

Efforts to guide the optimization process from a higher-level include the use of profile-based optimizations [18, 29, 22], and compile-time performance prediction [45]. These studies demonstrate how to use information about performance in optimization decision-making. They are concerned indirectly with machine and program characteristics insofar as they are reflected by the measured performance of a given program. However, for monitoring, we want to learn what kind of code instrumentation will best demonstrate the impact of an optimization on code performance, in addition to using run-time code performance to drive an optimization.

2.4 Quantification and Representation

From the prior work on each of these fronts, we can summarize the various kinds of information that has proved

to be useful in characterizing the interactions of optimizations. In particular, we focus here on how to quantify and represent this information. This is not necessarily an exhaustive list, but a good representative of how machine resources, program characteristics, and performance information are quantified and represented:

- A machine architecture description, including features such as numbers of registers and instruction equivalences [2]. Machine characteristics include instruction costs in terms of execution cycles used, pipeline characteristics, cache size and configuration, memory access costs, communication overhead, and ILP and other features of high-performance architectures.
- A control flow graph (CFG) and an SSA form to represent a program.
- A program dependence graph representation of the program to represent data and control dependences.
- Annotations on an intermediate program representation to keep track of the number of live variables and their live ranges.
- A register interference graph.
- Counts of the number of uses and definitions for variables in a region of code.
- The number of executed instructions as a percentage of total instructions before and after an optimization is applied.
- A specific value associated with each assignment statement to represent the cost of the statement as a way to guide optimization. Presumably, costs of other types of statements could be assigned and used as well.
- Various frameworks to describe the profitability of combinations of optimizations.
- Instrumentation of code using profiling tools to collect a wide variety of information about run-time performance characteristics. An example is profiling information to characterize branching behaviors to determine whether or not to fill delay slots.

This list includes information from all three categories: machine resource characteristics, program characteristics, and performance characteristics. Some information is embedded in the program representations, while other information is represented separately.

3 Monitoring Issues

In this section, we describe some of the issues to be addressed in designing a monitoring system to be used in conjunction with an optimizer to monitor the impact of optimizations.

Which machine resources should be monitored? Identifying which resources are involved in optimization interactions and therefore should be monitored is key to providing the most appropriate feedback to the optimizer. We looked at related work in this area in the previous section. There are many potential system resources and parameters to target for monitoring usage: register set sizes, memory structure and access latency, cache organization and size, processor speed, run-time costs of instructions, communications overhead, and available parallelism. The choice of monitoring targets depends on how each optimization changes the utilization of the resources, and how this affects the safety and profitability of other optimizations. We need not monitor the usage of a resource that is not affected by optimizations, but need to carefully monitor resources for which usage is highly affected. Resources with usage only slightly affected could be monitored at a lower priority.

What program representation(s) must be kept to enable monitoring of interactions of various optimizations? In order to monitor the interactions between optimizations performed on different levels of the code and allow for interleaving of optimizations that operate at different code levels, the internal program representation must be able to correlate the different levels simultaneously. In particular, a current representation of the code at high and low levels must be maintained simultaneously, preferably in the same structure for easy mapping between representations.

What information is fixed and does not need to be gathered at compile- or run-time? Some information about optimizations, such as enabling and disabling conditions of individual optimizations is static, and is already present before optimization begins. In this case, no monitoring is needed during optimization for this information.

What are the best techniques for collecting this information? Among the possibilities are static analysis techniques and run-time profiling. In addition to analyses such as those for dependence and data flow information, static techniques include annotating the program representation data structure to reflect various program characteristics such as live ranges and locality information.

How often should the information be collected, and in what fashion should it be updated? In order to effectively use the monitoring information throughout the optimization process to deal with the interactions, the information needs to remain current. This suggests a need for up-

dating this information as optimizations are performed. This can be done by algorithms for the update that work as an incremental approach [35] or a demand-driven approach [15]. The granularity of information update can be at anywhere from the lowest level (instruction level) to the highest level (source level), and could be done after each application of an optimization for the most accurate reflection of the interactions of optimizations with resources and each other.

How long should information be kept? Keeping a detailed and exhaustive optimization history has the potential to generate a large volume of information. If this quantity is too large to be practical or useful, older or less useful information should be discarded. This would affect the ability to step backwards through the optimization process beyond a certain point, so a tradeoff between information volume and capabilities is required.

Is there anything to be gained by looking beyond second-order interactions to third- and higher-order interactions of optimizations? Perhaps there is a point of stabilization, where the negative effects of a misapplied transformation can be counteracted by the appropriate sequence of following transformations. The answer to this question may have a large effect on the complexity of the monitoring system and eventual self tuning optimizer.

What capabilities in a tool would allow for thorough experimentation with optimization interaction? We believe that useful experimentation can be performed using an optimization interaction monitoring tool that would offer flexible and interactive optimization application, ordering and undoing, and would collect and display relevant resource utilization and program analysis information continuously. The tool needs to be flexible to allow for experimentation in different approaches to monitoring that will give the most useful information for tuning optimizations.

4 Design of a Monitoring System

Without being too specific about the particular resources and program characteristics to be monitored, we describe a framework for an optimization monitoring system. This framework forms the basis for an experimental optimization interaction monitoring tool that will be described in the next section. Figure 1 depicts the general framework and the relationship between the monitoring system and the optimizer. The monitoring system consists of an optimization monitor, resource monitor, sequencer/controller, and a static analyzer. The static analyzer may be partially or completely part of the optimizer. The monitoring system utilizes and maintains information in a program representation, architecture description, and profile information database. The next few subsections describe the relationship between the optimizer and the monitoring system, each component of the monitoring system, and the capabilities provided by the user interface.

4.1 Relationship to Optimizer

The monitoring system is not embedded in the optimizer, but rather acts as a separate entity, to be used in conjunction with an optimizer. However, the optimizer and the monitoring system share information and possibly parts of the static analysis component. The application of the optimizer is controlled by the sequencer/controller. The optimizer itself makes use of optimization modules, which isolate the functionality of each optimization phase, allowing for easy modification and extension of the optimization library. The types and levels of optimizations performed will be determined by the particular compiler infrastructure utilized.

4.2 Features and Capabilities

The **front end** of the system parses program source code, translating it into an intermediate program representation. For this, we use a pre-existing compiler front end. The **program representation** is an intermediate representation in the form of an integration of an abstract syntax tree and a low-level instruction list, and also attempts to associate lines of source code with the intermediate representation when possible. Facilities are included for annotating the intermediate program representation with an optimization profile and relevant resource utilization information. This information is linked to the representation at the line-, block-, region-, and function-level, where possible.

The complete description of the **target machine architecture** is available to all optimization phases and to the optimization monitor. The information contained in the architecture description is used to guide, quantify and compare the impacts of optimizations on system resource utilization, as well as for many machine-dependent optimizations.

The **static analyzer** provides a wealth of information about the current state of the program. Common techniques are data flow analysis, value numbering, memory usage summarization, and alias analysis [1].

The **optimization monitor** coordinates the efforts of the other functional units. It is responsible for incorporating initial machine information from the architectural description, updating the program representation with new information received from the static analyzer, optimizer, profile monitor, and resource monitor, and communicat-

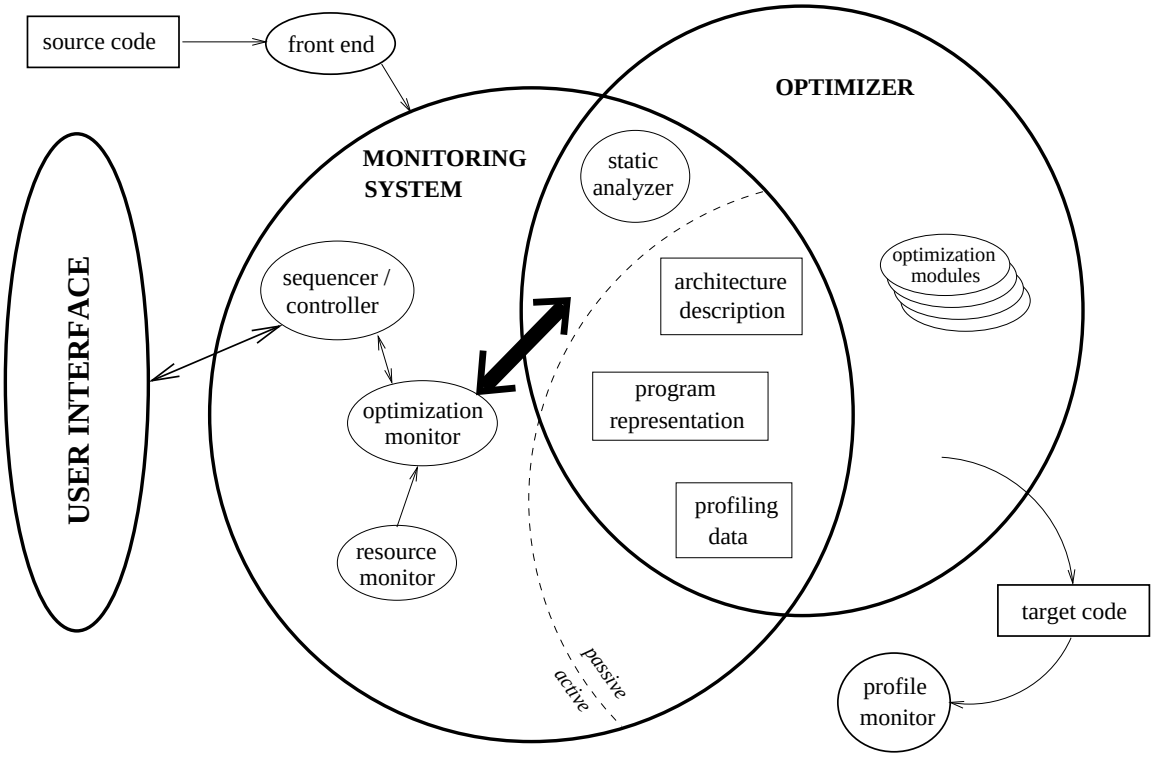

Figure 1: An Optimization Monitoring System Framework.

ing commands to and from the sequencer/controller to the optimizer.

The **resource monitor** actively measures system resource utilization as characterized by values such as the number of available registers at a given point in the code, cache utilization, memory access patterns, and pipeline utilization. The current information is compared against the architecture description and previous resource monitoring information to determine the effectiveness of optimizations and their impacts on system resources. By running the target code generated after optimization, valuable **run-time profiling** information is gathered. This information can characterize the overall effects of optimizations on performance, and can guide further optimization strategy.

The **sequencer/controller** is the contact point between the optimization monitor and the user interface. This unit is also a rudimentary user-controlled optimization *strategist*, allowing the use of the collected monitoring information during the optimization strategy decision-making process.

The sequencer provides a means for the user to specify the ordering of optimizations. The user can undo the effects of an optimization, and can also specify regions for exclusive optimization. Exclusive application of optimizations enables experimentation into region-based optimization [22].

The controller allows the user to interact with the optimization process. In order to directly examine the effects of optimizations, and their impact on various resources and other optimizations, the ability to step forward and backward through the optimization process is needed. Both forward and backward stepping can be done on the instruction-, block-, region-, and function-level, or directly to the beginning or end of the optimization process.

4.3 User Interface

The user interface provides the control of the optimization monitoring system and a display of relevant information. The key functionality of the user interface includes:

- An intuitive graphical interface that provides a convenient way to control the forward and backward stepping of the optimization process, interactive optimization selection and steering, and display of the code, optimization profile, resource utilization and other information to the user.

- Clicking on a line, block, region or function causes the associated monitoring information to be displayed.
- Every time the code changes, the display is updated. This update includes new resource utilization and optimization profile information.
- Viewing of multiple levels of code is possible, allowing the effects of source-level transformations to be witnessed in the low-level code, and vice-versa where possible.
- Regions ranging from the line- to function-level, and even larger, are selectable. In addition to the display of information related to a given region, the localized application of an optimization to the selected region can be performed.
- Displaying of higher-level information such as the current function being optimized, the optimization being performed, some transformation sequence number, a total count of all transformations performed, current count of the number of lines of target code, the use of breakpoints (set, unset, list), and overall resource measurements.

These user interface capabilities together provide the user with a monitoring tool which is user-friendly, flexible in control over the optimization process, and responsive in providing feedback on the monitoring of optimization impacts.

5 Prototype Monitoring Tool

In this section, we describe our implementation of a prototype of an optimization interaction monitoring tool based on the design concepts outlined earlier. This tool will be used to experiment with the interactions of optimizations and measure their impacts on resources, program characteristics and each other. Eventually, this tool will be extended to explore the use of this information in guiding optimization strategies for improved performance, particularly for ILP architectures.

5.1 Providing The Base

The tool is being built as an extension of the Stanford University Intermediate Format (SUIF) [43] and the Very Portable Optimizer (*vpo*) [2] compilers. This combination provides a versatile research testbed for studying the interactions of both machine-independent and machine-dependent optimizations, as demonstrated by Table 1.

SUIF is a flexible platform, written in C++, for research on compiler techniques for high-performance machines. It is easily modified and extended with new optimization phases, and can also be used to perform advanced program analysis. SUIF currently accepts as input C and Fortran (via *f2c*) source programs. It excels at source-level and other machine-independent optimizations which are successful for improving performance on ILP and other high-performance architectures. SUIF is currently used for research on topics including: scalar data flow optimizations, array data dependence analysis, loop transformations for both locality and parallelism, and software prefetching.

Developed at the University of Virginia, *vpo* is an easily retargetable optimizing compiler that relies on a machine description to enable better-informed low-level optimizations and easier porting to new machines. The *vpo* compiler is a very stable C compiler, and is itself written in C. It uses an intermediate representation called a *Register Transfer List (RTL)*. The RTL is used to describe the source program at a low-level. Retargeting is accomplished by providing a new machine description for a target architecture in the form of RTL instructions for each instruction on the target machine that is needed to implement the requirements of the source language. *Vpo* is currently targeted to at least seven architectures, including Mips, Sparc and DEC Alpha machines.

5.2 The Underlying Technology

In order to use both SUIF and *vpo* as a single optimizer, the optimizers needs to be able to exchange information

Table 1: Optimizations performed by SUIF and *vpo*.

Optimization	SUIF	*vpo*
constant folding	x	
constant propagation	x	
copy propagation	x	
dead code elimination	x	
forward propagation	x	
code motion	x	x
loop invariant motion	x	
induction variable elimination	x	x
reduction variable elimination	x	
loop strength reduction		x
common subexpression elimination		x
branch delay slot filling		x
instruction scheduling		x
global register allocation		x
evaluation order determination		x

to be used in monitoring and eventually in the self tuning optimizer. SUIF has two levels of representation. High-SUIF is an abstract syntax tree, while Low-SUIF is more of an instruction list representation. *Vpo* uses a low-level RTL representation that is closest to Low-SUIF. To connect SUIF and *vpo* we are developing a stand-alone code generator that parses a Low-SUIF file and emits RTL that can be used by *vpo*. We are also investigating the possibility of designing a SUIF phase that will traverse the SUIF intermediate representation and emit the corresponding RTL.

We are reusing the static analysis capabilities that exist in the SUIF and *vpo* compilers, rather than rewriting the same phases. We are exploring ways to extract information using these techniques, possibly through modifying or enhancing the algorithms already in place. SUIF allows the dynamic addition of annotations to the representations, which enables our monitoring tool to easily attach and update information about the optimization process to the actual intermediate code. *Vpo* currently does not support easy addition of annotations to the representation. Some extension of the capabilities of *vpo* to handle annotations may be needed.

For the machine resource characteristics, we are reusing the *vpo* machine descriptions. Further investigation is needed to determine whether additional information about the architecture will be included to enable better monitoring of optimization impacts on the machine resources.

The monitoring system is being designed with an X-Windows interface to provide an easy interface between the user and the functionality of the monitoring tool. The basic design of the interface window bears some resemblance to that of the *xvpodb* [4] interface, which uses a VCR analogy in its design. The technology that links the the user interface to the optimizers is currently based on the visualization tools that are known to work with SUIF and *vpo*. The *spp* tool [43] generates a graphical representation of the SUIF control flow structure, and *xvpodb* [4] is a graphical debugging tool that interactively displays *vpo* optimization information.

Our tool is being designed to allow the user to move forward and backward through each step of the optimization process, observing the resultant changes to the code. The debugging tool *xvpodb* retains information about the optimization process by requesting *vpo* to generate a list of each instance of the application of transformations to individual instructions as they are performed. This list is manipulated by *xvpodb* to provide the ability to step forward and backward through the sequence of transformations.

6 Summary and Future Work

Using limited information that characterizes the interactions of optimizing transformations has proven successful at improving the effectiveness of small sets of optimizations. The natural conclusion is that, if these techniques are extended to all optimizations, then further improvements in the effectiveness of an optimization strategy should follow. However, this extension requires the development of techniques to collect and quantify pertinent information.

This paper describes the first steps towards this goal, namely, our work towards the development of an interactive optimization monitoring tool, applicable to high-performance ILP architectures. We hope that experiments using this tool will help us to gain a comprehensive understanding of the complex interactions between optimizations and their impacts on the use of machine resources. We believe that these experiments can help provide answers to the questions of which optimizations should be applied, and where and in what sequence should they be applied, to achieve the most effective optimization for a given program and target machine.

References

[1] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. Technical Report CSD-93-781, Computer Science Division, University of California, Berkeley, 1993.

[2] Manuel E. Benitez and Jack W. Davidson. A portable global optimizer and linker. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 329–338, June 1988.

[3] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. Resource spackling: A framework for integrating register allocation in local and global schedulers. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 135–145, Montreal, Canada, August 1994.

[4] Mickey R. Boyd and David B. Whalley. Graphical visualization of compiler optimizations. *Journal of Programming Languages*, 3:69–94, 1995.

[5] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–131, April 1991.

[6] Thomas S. Brasier, Philip H. Sweany, and Steven J. Beaty. CRAIG: A practical framework for combining instruction scheduling and register assignment. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 11–18, June 1995.

[7] Preston Briggs and Keith Cooper. Effective partial redundancy elimination. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1994.

[8] David Callahan, Steve Carr, and Ken Kennedy. Register allocation via hierarchical graph coloring. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 53–65, White Plains, New York, June 1990.

[9] Steve Carr. Combining optimization for cache and instruction-level parallelism. In *Parallel Architectures and Compilation Techniques (PACT)*, 1996.

[10] Cliff Click. Global code motion global value numbering. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1995.

[11] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *ACM SIGPLAN Symposium on the Principles of Programming Languages*, pages 25–35, 1989.

[13] D. M. Dhamdhere. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, 1991.

[14] K.-H. Drechsler and M. P. Stadel. A solution to a problem with Morel and Renvoise's "global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, 1988.

[15] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *ACM SIGPLAN Symposium on the Principles of Programming Languages*, pages 37–48, January 1995.

[16] A. E. Eichenberger and E. S. Davidson. Stage scheduling: A technique to reduce the register requirements of a modulo schedule. In *International Symposium on Microarchitecture (MICRO)*, November 1995.

[17] C. Eisenbeis and D. Windheiser. Optimal software pipelining in presence of resource constraints. In *Parallel Architectures and Compilation Techniques (PACT)*, August 1993.

[18] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.

[19] Stefan M. Freudenberger and John C. Ruttenberg. Phase ordering of register allocation and instruction scheduling. In Robert Giegerich and Susan L. Graham, editors, *Code Generation - Concepts, Tools, Techniques: Proceedings of the International Workshop on Code Generation*, pages 146–170, May 1992.

[20] James R. Goodman and Wei-Cheung Hsu. Code scheduling and register allocation in large basic blocks. In *ACM International Conference on Supercomputing*, pages 442–452, November 1988.

[21] R. Govindarajan, E. R. Altman, and G. R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Parallel Architectures and Compilation Techniques (PACT)*, 1994.

[22] Richard E. Hank, Wen mei W. Hwu, and B. Ramakrishna Rau. Region-based compilation: An introduction and motivation. In *International Symposium on Microarchitecture (MICRO)*, 1995.

[23] R. A. Huff. Lifetime-sensitive modulo scheduling. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1993.

[24] Wayne Kelly and William Pugh. A unifying framework for iteration reordering transformations. In *Proceedings of the IEEE First International Conference on Algorithms and Architectures for Parallel Processing*, 1995.

[25] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, 1993.

[26] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. The power of assignment motion. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1995.

[27] J. Llosa, M. Valero, and E. Ayguade. Bidirectional scheduling to minimize register requirements. In *Fifth Workshop on Compilers for Parallel Computers*, June 1995.

[28] Jack L. Lo and Susan J. Eggers. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1995.

[29] S. A. Mahike, P. Change, and W. W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 1991.

[30] Soo-Mook Moon and Kemal Ebcioğlu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *International Symposium on Microarchitecture (MICRO)*, 1992.

[31] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2), February 1979.

[32] Q. Ning and G. R. Gao. A novel framework of register allocation for software pipelining. In *ACM SIGPLAN Symposium on the Principles of Programming Languages*, January 1993.

[33] Cindy Norris and Lori L. Pollock. Register allocation sensitive region scheduling. In *Parallel Architectures and Compilation Techniques (PACT)*, Limassol, Cyprus, June 1995.

[34] S. S. Pinter. Register allocation with instruction scheduling: A new approach. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1993.

[35] Lori L. Pollock and Mary Lou Soffa. An incremental version of iterative data flow analysis. In *ACM SIGPLAN Symposium on the Principles of Programming Languages*, pages 152–164, January 1985.

[36] William Pugh. Uniform techniques for loop optimizations. In *ACM International Conference on Supercomputing*, 1991.

[37] J. Ramanujam. A linear algebraic view of loop transformations and their interaction. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, 1993.

[38] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

[39] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: History, overview and perspective. *The Journal of Supercomputing*, 7(1):9–50, January 1993.

[40] Barry Rosen, Mark Wegman, and Kenneth Zadeck. Global value numbers and redundant computations. In *ACM SIGPLAN Symposium on the Principles of Programming Languages*, January 1988.

[41] Rafael H. Saavedra, Weihua Mao, Daeyeon Park, Jacqueline Chame, and Sungo Moon. The combined effectiveness of unimodular transformations, tiling, and software prefetching. In *Proceedings of the International Parallel Processing Symposium*, 1996.

[42] Vivek Sarkar and Radhika Thekkath. A general framework for iteration-reordering loop transformations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1992.

[43] Stanford Compiler Group. *The SUIF Parallelizing Compiler Guide*. Stanford University, 1994. Version 1.0.

[44] Jian Wang, Andrease Krall, and M. Anton Ertl. Decomposed software pipelining with reduced register requirement. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 277–280, 1995.

[45] Ko-Yang Wang. Precise compile-time performance prediction for superscalar-based computers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 73–84, Orlando, Florida, June 1994.

[46] Debbie Whitfield and Mary Lou Soffa. An approach to ordering optimizing transformations. In *ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 137–146, March 1990.

[47] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4), October 1991.