Profit-driven Scalar Optimization

MIN ZHAO and BRUCE R. CHILDER
University of Pittsburgh
and
MARY LOU SOFFA
University of Virginia

Although optimizations have been applied for a number of years to improve the performance of software, problems with respect to the application of optimizations have not been adequately addressed. For example, optimizations may degrade performance in certain circumstances. However, there is no efficient way to know when a degradation will happen. In this research, we investigate the profitability of optimizations, which is useful for determining the benefit of applying optimizations. We develop a framework that enables us to predict profitability using analytic models. The profitability of an optimization depends on code context, the particular optimization and machine resources. Thus, our framework has analytic models for each of these components. As part of the framework, there is also a profitability engine that uses models to predict the profit. In this paper, we target scalar optimizations, and in particular, describe the models for partial redundancy elimination (PRE), loop invariant code motion (LICM) and value numbering (VN). We implemented the framework for predicting the profitability of these optimizations. Based on the predictions, we can selectively apply profitable optimizations. We compared the profit-driven approach with an approach that uses a heuristic in deciding when optimizations should be applied. Our experiments demonstrate that the profitability of scalar optimizations can be accurately predicted by using models. That is, without actually applying a scalar optimization, we can determine if an optimization is beneficial and should be applied.

Categories and Subject Descriptors: D3.4 [Programming Languages]: Processors - Compiler; Optimization
General Terms: Performance, Design and Experimentation
Additional Key Words and Phrases: Profitability, Prediction, Profit-driven

1. INTRODUCTION

Optimizations, introduced in the late 1950s, have become essential components of compilers. Although concentrated research efforts have been devoted to the development of particular optimizations, certain problems with respect to the application of optimizations have yet to be adequately addressed. First, it is recognized that optimizations may degrade performance in certain circumstances [BrCo94, ZhCS03]. But so far we have no efficient way of determining the impact of optimizations and choosing not to apply optimizations to avoid performance degradation. Second, we also know that optimizations enable and disable other optimizations so the order of applying optimizations can have an impact on performance (i.e., phase ordering problem) [WhSo97, TVVA03, ACGH04, KHHW04]. However, typically, compilers apply optimizations in a predetermined order. The choice of the order is guided by a compiler writer’s expertise and used for all programs. Because of these problems, compilers are not achieving the maximum benefits from applying optimizations.

A number of events are occurring that demand systematic solutions to these problems. Because of the continued growth of embedded systems and the critical importance of time-to-market in this domain, there is an energetic movement to write embedded software in high-level languages. The use of high-level languages in this area requires a high quality optimizing compiler that can intelligently apply optimizations to achieve the best performance improvement. Another activity that has brought optimization problems to the forefront is the trend toward dynamic optimization. To be effective, dynamic optimization requires a good understanding of the benefit and cost of applying
optimizations. Currently, it is unclear when and where to apply optimizations dynamically and how aggressive optimizations can be and still be profitable after factoring in the cost of applying optimizations.

Traditionally, heuristics have been used to address some of the problems of applying optimizations. However, heuristics tend to be ad hoc and focus specifically on a single or a small class of optimizations. Heuristics also require tuning parameters to select appropriate threshold values. The success of a heuristic can depend on these values and the best choice can vary for different optimizations and code contexts.

To systematically tackle these problems, we need to better understand the profitability of optimizations. An experimental approach has been used in previous research to address the profitability [CoST01, KiKO00, ACGH04, KHHW04, ChCH05]. That is, the profitability is evaluated by actually applying optimizations and executing the optimized code. Based on the evaluations, the order and the configuration (e.g., tile size for loop tiling) to apply optimizations are determined.

Due to the high cost of applying optimizations and executing the optimized code, our research focuses on determining the profitability of optimizations through analytic models. In this paper, we present a framework of analytic models for predicting the profitability of scalar optimizations. In particular, we address the specific problems of how scalar optimizations impact registers, computation (i.e., functional units) and overall performance. Optimization profitability depends on code context, particular optimizations and machine resources, all of which need to be modeled. As part of the framework, there is a profitability engine that uses the models to predict the profitability of applying an optimization at any code point where the optimization is applicable.

We have developed models for a number of scalar optimizations, including copy propagation, constant propagation, dead code elimination, partial redundancy elimination (PRE), loop invariant code motion (LICM) and value numbering (VN). In this paper, we focus on the models for PRE, LICM and VN. Models for the other optimizations are useful when considering the impact of an optimization sequence, which is beyond the scope of this paper. We implemented the models and the profitability engine, which are used to predict the profitability of PRE, LICM and VN. Based on the prediction, we either apply the optimization or not. We compare our profit-driven PRE and LICM with a heuristic-driven approach that considers the register pressure when applying an optimization. We also compare profit-driven PRE, LICM and VN with an approach that always applies the optimization if it is safe (i.e., applicable). Experimental results demonstrate that our model-based framework can accurately predict the profitability of scalar optimizations with reasonable overhead. That is, without actually applying a scalar optimization, we can predict whether it is beneficial and then selectively apply it.

The contributions of this paper include:

- Analytic models for code, optimizations, and machine resources;
- A framework that uses models to predict the profitability of scalar optimizations; and
- An implementation of the framework and an experimental evaluation demonstrating that the model-based approach for predicting the profitability of scalar optimizations is effective and efficient.

In Section 2, we show the conceptual structure of our model-based profitability framework. A framework instance to determine the impact of PRE, LICM and VN on registers and computation is described in Section 3. Experimental results are presented in Section 4, followed by related work and conclusions.
2. PROFITABILITY FRAMEWORK

To determine the profitability of an optimization, we require models that are useful for predicting the impact of the optimization on performance (i.e., execution time). Performance is generally affected by registers, computation and other resources (e.g., cache). Thus, we need to determine the profit of an optimization for each resource and then combine the profits. Importantly, to determine profitability, we do not require exact profit numbers but numbers “accurate enough” that the right decision as to when and what optimizations to apply can be made.

Our framework, given in Figure 1, has three types of analytic models (code, optimization and resource models) and a profitability engine that processes the models and computes the profit. The models are plug-and-play components. When the new models for the code, optimizations or machine resources are needed, they can be developed and easily added into the framework.

![Figure 1. A model-based framework for predicting the profitability](image)

### 2.1 Code Models

The code model expresses those characteristics of the code segment that are changed by an optimization and impact a resource. In our framework, there is a code model for each machine resource. For example, there is a register code model to express live range information because live ranges can be changed by an optimization and impact the registers. There is also a computation code model to specify the frequency of the occurrence for operations. In some cases where optimizations (e.g., loop optimizations) have a significant impact on cache, a code model for cache is also needed to specify the array reference sequence [ZhCS03]. In this paper, we focus on scalar optimizations, which have negligible effect on cache (i.e., loop behavior dominates cache performance [McCT96]) and we consider only registers and computation.

The code models are extracted from the low level intermediate representation of the program. When safe conditions for applying an optimization are detected, the code models are automatically generated by the optimizer. Note, in this work, we assume that data flow information is available to determine if an optimization is legal. If legal, we then apply profitability analysis. However, we could also do the reverse: we could determine the hot regions of the code and the profitability of an optimization in a region and if the transformation is profitable, use data flow analysis (in particular, demand-driven data flow analysis [DuGS97]) to determine if the optimization is legal.
2.2 Optimization Models

Optimization models are written by the compiler engineer when developing a new optimization. An optimization model expresses the semantics (i.e., effect) of an optimization, from which the impact of the optimization on each resource can be determined. For example, the PRE optimization model describes the semantics of PRE, from which we can infer how PRE changes the code models under consideration.

The effect of an optimization is determined from the code changes that the optimization introduces. Optimizations can cause non-structural and structural code changes, which can be expressed by editing changes on a control flow graph. These edits are Insert/Delete a statement (including its operation and operators), Insert/Delete a block and Insert/Delete an edge. All optimization code changes can be expressed in terms of these edits [BiSo90]. Thus, the code changes of a particular optimization can be described as a series of basic edits. For example, constant propagation can be represented as “Delete variable v at statement P; Insert constant c at statement P”.

To determine the impact of an optimization on registers, an optimization model for the register allocator must be developed (shown as “RA” in Figure 1). The characteristics of the register allocator that need to be modeled are whether the allocator is local or global and how it spills the live ranges (i.e., the number of additional loads and stores that are inserted into the code). A model for the register allocator can be constructed that approximates a particular register allocation scheme, say graph coloring [Chai82, GeAp96] or linear scan [PoSa99]. In this work, we are interested in the impact of other optimizations on registers rather than the impact of a particular register allocation scheme. Hence, we only need a representative register allocation model, such as coloring.

2.3 Resource Models

The framework has a model for each machine resource, which describes the resource configuration and benefit/cost information in using the resource. A resource model is developed based on a particular platform. For example, to determine the register profit, we need to know the number of available hardware registers and the cost of memory accesses (loads and stores). When considering computation, the computational operations available in the architecture and their execution latencies are needed. Since we do not consider instruction scheduling, the profit deduced using the computational resource model is an approximation, as is true for most of the resource models.

2.4 Profitability Engine

The models in the framework are descriptive and provide the information needed to compute profitability. The other important component of our framework is the profitability engine. When conditions for an optimization are detected, the code, the optimization, and the resource models are input into the profitability engine. The engine uses the information in the models to compute the profit of an optimization at a program point. The profit can be computed for one resource or for combined resources. From an optimization model, the engine determines the code model changes caused by the application of the optimization. It then applies these changes to the code model and generates a new code model that represents the effect of the optimization. Finally, it uses the resource model to determine the impact of the changes on the resource.

For example, assume the impact of an optimization on registers is desired. The engine inputs the register code model, an optimization model, a register allocation model and a register resource model. Then the engine determines the changes on the live ranges (i.e., the register code model) based on the optimization model. Since an optimization model expresses the semantics of the optimizations as basic edits, the engine takes the edits and
computes the changes on the live ranges using an incremental dataflow algorithm [PoSo89]. The profitability engine then uses a register allocation model to determine how the spills (i.e., loads and stores) are changed according to these live range changes. Finally, the engine computes the profit associated with the change in the spills.

Using our framework, the optimizer can perform profit-driven scalar optimization. Once it is known that an optimization (i.e., a single instance of an optimization) is applicable, the optimizer generates the code models involved in the optimization and triggers the profitability engine to predict the profit of the optimization. When the engine is triggered, it takes the code models, optimization models and resources models, updates the code models and determines the profit on resources under consideration (i.e., registers and computation for scalar optimizations). Based on whether there is a benefit or not, the optimizer applies the optimization accordingly.

3. FRAMEWORK INSTANCE

In this section, we describe an instance of our framework for predicting the profitability of PRE, LICM and VN. The impact of PRE, LICM and VN on computation is clear: they insert or delete operations at some program points. Their impact on registers is more complicated and depends on the code context. Sometimes they may introduce register spills, while in other cases they may decrease the number of spills. In Figure 2, we show an example where PRE increases register pressure by introducing one more spill. In the figure, PRE moves the use of $a$ and $b$ at statement 10 up in the code. Because $a$ and $b$ are used after statement 10, their live ranges remain the same. PRE also introduces a new live range for the temporary variable, $v$. Thus, if there are 7 hardware registers, the inserted live range will cause a spill to memory. However, if $a$ and $b$ were not used after statement 10, their live ranges would be shortened. In that case, the total number of live ranges would be decreased by one, leading to fewer spills.

PRE increases the number of register spills by one, if there are 7 hardware registers.

Figure 2: An example of PRE impacting registers
3.1 Code Models for Registers and Computation

The register code model represents the code as live ranges of global and local variables (including temporaries and parameters). We represent a live range by $LR_{[n..m]}^x$, where $x$ is a variable name and $[n..m]$ is the range of statements over which $x$ is defined and referenced. The live range of a variable is not necessarily contiguous. For example, in Figure 2 after PRE, the live range of $v$ consists of two parts and can be expressed as $LR_{[7..10]}^v$, where $[6'..10]$ is a shorthand to represent a contiguous range.

The computation code model represents the frequency of occurrence for each operation in the code. For an operation $op$, its frequency is represented as a sequence $\{f_{B_1}, f_{B_2}, ..., f_{B_n}\}_op$, where $f_{B_i}$ is the number of $op$ in block $B_i$ and $op$ appears in blocks $B_1, B_2, ... B_n$.

3.2 Optimization Models

An optimization model expresses the semantics (i.e., code changes) of the optimization as a series of basic edits. We represent a basic edit by its action and code location. For example, we represent “insert a statement $x \leftarrow a + b$ at code location $S$” by “Insert <DEF $x$ USE $a, b$ OP add $> @ S$ ”. In some cases, only a part of a statement is involved in a basic edit. For example, to replace the statement “$x \leftarrow a + b$” at code location $S$ with the statement “$x \leftarrow v$”, only the use variables and the operations are involved in the replacement. We represent the replacement by:

“Delete <USE $a, b$ OP add $> @ S$ ”

“Insert <USE $v$ OP copy $> @ S$ ”.

Next, we describe the optimization models for PRE, LICM, VN and the register allocation.

3.2.1 PRE Optimization Model

PRE inserts and deletes computations in the flow graph so that after PRE each path contains no more occurrences of the computation than before. The PRE algorithm that we model is lazy code motion (LCM), which takes register pressure into account by hoisting an expression no earlier than necessary [KnRS92]. Although LCM considers register pressure, there are still cases where PRE introduces more register spills (as shown in Figure 2).

PRE has three semantic actions that create code changes:

- Insert a statement: inserts the redundant expression $EXP$ and introduces a temporary $v$ to hold the result at a destination code location;
- Replace the computation: replace $EXP$ with a copy from the temporary $v$ at the source code position; and
- Update the same expressions (that have the same operation and operands as $EXP$): replace each same expression’s destination with the temporary and insert a copy statement following it.

The PRE optimization model showing these code changes is given in Figure 3. A code movement or replacement can be expressed as a deletion of the statement at the source location and an insertion of a statement at the destination location. For clarity, we use $S_s$ to represent the source location and $S_d$ for the destination location.
In the figure, an assignment from the expression \( EXP \) to a temporary \( v \) is inserted at a destination code location after \( S_d \). That is, the variables of \( EXP \) are inserted as uses and the temporary \( v \) is inserted as the definition with the operation \( op \) at \( S_d' \), where \( S_d' = S_d + 1 \).

We use \( S' \) to represent the new code location. At the source code location \( S_s \), the expression \( EXP \) is deleted and a copy from the temporary \( v \) is inserted. The definition variable is unchanged. Finally, for each expression \( T \) that has the same computation at the code location \( S_w \), the destination \( w \) is replaced by the temporary \( v \) and a copy from \( v \) to \( w \) is inserted at the new location \( S_w' \).

After PRE, the temporary \( v \) can be propagated and copy statements can be deleted by applying copy propagation, which is modeled by a separate optimization model for copy propagation.

### 3.2.2 LICM Optimization Model

LICM moves a statement from a loop body to the outside of the loop. There are certain conditions that must be met to safely apply LICM. An example is shown in Figure 4, where the invariant code “\( a \leftarrow b + 1 \)” is moved out of the loop body because each of its operands is either defined outside of the loop or a constant.
The semantic action of LICM is simply a code movement. The optimization model for LICM is shown in Figure 5. At the destination code location (i.e., the loop preheader), an invariant statement can be inserted and at the source location (i.e., inside the loop), the invariant statement is deleted.

### 3.2.3 VN Optimization Model

VN tries to find and remove redundant expressions that are equivalent based on their values (unlike PRE which considers the lexical equivalent expressions). It assigns an identifying number to each expression in a particular way and then uses the number to find and remove redundant computations.

VN has three actions for a basic block: 1) it removes redundant or meaningless Φ-instructions (Φ-instruction is a pseudo-assignment that introduces a new definition point at the merge point in the control-flow graph [BCHS98]); 2) it simplifies computation (constant folding) or removes the redundant computation; and 3) it adjusts the inputs of Φ-instructions in successor blocks. When converting SSA to non-SSA intermediate code, some Φ-instructions should be replaced by copy instructions in predecessor blocks. Because the inputs of the Φ-instructions have been adjusted, they do not show where they were originally defined (i.e., where the copy should be inserted). A general algorithm can be used to replace the Φ-instructions with copy instructions [BCHS98] and to accurately predict the impact of VN, the replacement algorithm should be modeled.

A simplification is to incrementally add the copy statements as VN progresses. In our VN implementation, we replace the redundant computations with copy statements (instead of removing them) and retain the inputs of Φ-instructions when processing each basic block. We then use Φ-instructions to keep the useful copy statements and remove the useless ones. In this way, no copy statements will be inserted when converting SSA to non-SSA code.

The VN optimization model describes the code changes from VN. The model is given in Figure 7. In the figure, VN[x] is the value number of x, where x can be a variable, an expression or a Φ-instruction. Each value number is a variable name. For an expression,
its value number is the variable name of the first occurrence of the expression in this path in the dominator tree.

In Figure 7, if an expression \( \text{EXP}(y \text{op} z) \) at \( S_s \) is redundant, it is replaced by a copy from its value number \( v \). That is, the variables of \( \text{EXP} \) are deleted as uses with the operation \( \text{op} \) at \( S_s \). And the expression’s value number \( v \) is inserted as a use with the operation \( \text{copy} \) at \( S_s \). In the example shown in Figure 6, at statement 4, the redundant expression \( d_0 + c_0 \) is deleted and a copy from its value number \( b_0 \) is inserted. At statement 5, the definition variable \( e_0 \) is used and is replaced by \( b_0 \).

In our VN algorithm, we also find statements for constant folding. The second rule in Figure 7 shows if an expression \( \text{EXP}(y \text{op} z) \) at \( S_s \) can be simplified by constant folding, \( \text{EXP} \) is deleted. The third rule shows if a redundant \( \Phi \)-instruction is deleted, all the uses of the defined variable \( x \) are replaced by the value number \( v \). Thus, at the statement \( S_u \) where the defined variable \( x \) is used, \( x \) is deleted as a use and \( v \) is inserted as a use. The last rule models the deletion of a useless copy statement at statement \( S_s \). Here, the variable \( y \) is deleted as a use and the defined variable \( x \) is deleted as a definition with the operation \( \text{copy} \) at the location \( S_s \).

### 3.2.4 Register Allocation Optimization Model

To determine the impact of scalar optimizations on registers, we also need a model for register allocation. By applying register allocation, hardware registers are assigned to the live ranges. If the number of hardware registers is not enough, the register allocator selects live ranges to spill to memory, which impact the overall performance. Thus, to predict the impact of optimizations, we need to compute spills for the original live ranges and the live ranges changed by the optimization and compare them. This is a time
consuming process. Instead, we use an incremental approach that computes how spills are changed due to each live range change. Our register allocation model reflects this incremental approach.

We model a global graph coloring register allocator. Figure 8 shows the register allocation optimization model. For each changed live range change \( LR^c_{[n,...,m]} \), we determine how spills are changed. If \( LR^c_{[n,...,m]} \) is inserted or lengthened, it may introduce one more spill. Within the range \([n,...,m]\), if the insertion of a new live range causes the number of live ranges to exceed the number of available hardware registers, we select a live range to spill to memory, which introduces more loads and stores. We use \( LR^{all}_{[n,...,m]} \) to represent the live ranges in \([n,...,m]\). To select a live range to spill, we choose the one that has the least number of uses and definitions within the range, under the assumption that the register allocator typically performs well. Thus, we need to represent all variables’ uses and definitions within the range. Suppose, \( LR^t_{[n,...,m]} \) is selected to be spilled. If there is no definition of \( s \) before a use of \( s \) or there is no use of \( s \) within the range \([n,...,m]\), a store or load is inserted at the boundary of \([n,...,m]\). Otherwise, at all the uses or definitions of \( s \) within \([n,...,m]\), a load or store will be inserted. Alternatively, if \( LR^c_{[n,...,m]} \) is deleted or shortened, it may decrease one spill. This register allocation model is input to the profitability engine (see the next section) to predict the impact of the other optimizations on registers.

3.3 Profitability Engine

The profitability engine inputs the code models, optimization models and resource models, determines the changes on the code models (for both registers and computation)
and generates the optimized code models. Finally, it computes the profit on registers and computation.

Table 1. Incremental computation of the new register code model

<table>
<thead>
<tr>
<th>Code Change</th>
<th>Incrementally compute the new register code model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Insert a use of variable v at statement s</strong></td>
<td>IF ( v ) is <em>live at post-s</em> THEN no change; ELSE /* lengthen ( v )'s live range*/</td>
</tr>
<tr>
<td></td>
<td>The original live range ( LR_{[n, \ldots, m]}^v ) changes to</td>
</tr>
</tbody>
</table>
| | \[
| LR_{[n, \ldots, m]}^v \quad | | s \) post - dominate other uses \| \[
| LR_{[n, \ldots, m]}^v \quad | | otherwise \| \[
| | |
| **Insert a definition of variable v at statement s** | IF \( v \) is not *live at post-s* THEN no change; ELSE /* shorten \( v \)'s live range*/ |
| | The original live range \( LR_{[n, \ldots, m]}^v \) changes to |
| | \[
| LR_{[n, \ldots, m]}^v \quad | | s \) post - dominate other definition \| \[
| LR_{[n, \ldots, m]}^v \quad | | otherwise \| \[
| | |
| **Delete a use of variable v at statement s** | IF \( v \) is *live at post-s and v is not only use in a loop* THEN no change; ELSE /* shorten \( v \)'s live range*/ |
| | The original live range \( LR_{[n, \ldots, m]}^v \) changes to |
| | \[
| LR_{[n, \ldots, m]}^v \quad | | m \) post - dominate other uses \| \[
| LR_{[n, \ldots, m]}^v \quad | | otherwise \| \[
| | |
| **Delete a definition of variable v at statement s** | IF \( v \) is not *live at post-s* THEN no change; ELSE /* lengthen \( v \)'s live range*/ |
| | The original live range \( LR_{[n, \ldots, m]}^v \) changes to |
| | \[
| LR_{[n, \ldots, m]}^v \quad | | n \) post - dominate other definition \| \[
| LR_{[n, \ldots, m]}^v \quad | | otherwise \| \[
| | |

From an optimization model, the profitability engine determines how the optimization changes the register code model with an incremental dataflow algorithm [PoSo89]. Table 1 shows how to incrementally compute the new register code model (i.e., live ranges) for each edit given by the optimization model. In this table, \( post-s \) means the point immediately after statement \( s \). We use \( n \) to represent a statement where there is a definition of the variable \( v \) and \( use \) \( m \) to represent a statement where there is a use of the variable \( v \). For example, the effect on the live ranges from inserting a use of \( v \) (1st row of the table) depends on the current code. If \( v \) is already live at \( post-s \), there is no change. Otherwise, the original live range \( LR_{[n, \ldots, m]}^v \), is lengthened. If the inserted use at statement \( s \) is the last use (i.e., \( s \) post-dominates other uses), the new live range for \( v \) becomes \( LR_{[n, \ldots, m]}^v \). Otherwise, the new live range consists of the original live range and a range to the use statement \( s \). This range is represented as \( LR_{[n, \ldots, m]}^v \).

The profitability engine also infers how an optimization changes the computation code model. As shown in Table 2, the code changes from an optimization can be classified as either inserting an operation or deleting an operation. If an operation \( op \) is inserted at a
block \( B_s \), the number of \( op \) in block \( B_s \) (i.e., \( f_{B_s} \)) is increased by one. If an operation \( op \) is deleted at a block \( B_s \), \( f_{B_s} \) is decreased by one. Thus, the profitability engine can determine the impact of an optimization on the computation.

Table 2. Updates of the computation code model

<table>
<thead>
<tr>
<th>Code Change</th>
<th>Update the computation code model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert an operation ( op ) at block ( B_s )</td>
<td>The original operation list ( { f_{B_1}, f_{B_2}, ..., f_{B_n} } ) changes to ( { f_{B_1}, f_{B_2}, ..., f_{B_n} + 1, ..., f_{B_n} } )</td>
</tr>
<tr>
<td>Delete an operation ( op ) at block ( B_s )</td>
<td>The original operation list ( { f_{B_1}, f_{B_2}, ..., f_{B_n} } ) changes to ( { f_{B_1}, f_{B_2}, ..., f_{B_n} - 1, ..., f_{B_n} } )</td>
</tr>
</tbody>
</table>

For example, the impact of PRE on computation can be determined by the profitability engine, as shown in Figure 9. To insert a statement, the operation \( op \) is inserted at block \( B_d \) (the destination code location \( S_d \) is in block \( B_d \)). To replace the computation, the operation \( op \) is deleted at block \( B_s \) and a copy is inserted at block \( B_s \) (the source location \( S_s \) is in block \( B_s \)). Finally, to update the same expression \( T \) at the code location \( S_w \), a copy is inserted in block \( B_w \), where \( S_w \) is in block \( B_w \).

```
# Eliminate the partial redundant expression \( EXP(y \ op \ z) \) at \( S_s \)
Insert a statement at block \( B_d \):  
\[ \{ f_{B_1}, f_{B_2}, ..., f_{B_n} \} \Rightarrow \{ f_{B_1}, f_{B_2}, ..., f_{B_n} + 1, ..., f_{B_n} \} \]

Replace the computation at block \( B_s \):  
\[ \{ f_{B_1}, f_{B_2}, ..., f_{B_n} \} \Rightarrow \{ f_{B_1}, f_{B_2}, ..., f_{B_n} - 1, ..., f_{B_n} \} \]

\[ \{ f_{B_1}, f_{B_2}, ..., f_{B_n} \} \Rightarrow \{ f_{B_1}, f_{B_2}, ..., f_{B_n} + 1, ..., f_{B_n} \} \]

Update the same expressions at block \( B_w \):  
\[ \forall T \mid T = w \leftarrow EXP(y \ op \ z) \) at \( S_w \)  
\[ \{ f_{B_1}, f_{B_2}, ..., f_{B_n} \} \Rightarrow \{ f_{B_1}, f_{B_2}, ..., f_{B_n} + 1, ..., f_{B_n} \} \]
```

Figure 9. Impact of PRE on computation code model

After determining the changes on the code models, the profitability engine generates the optimized code model and computes the profit for the resource under consideration. For example, to compute the profit for registers, the engine computes the benefit/cost in terms of spills (i.e., loads and stores) based on the register allocation model. That is, for each live range change, the engine determines the impacted region and compares the total number of live ranges with the available hardware registers. If the total number of live ranges is larger, inserting a live range will introduce one more spill. To select a live range to spill to memory, the engine records the uses and definitions of all variables in the region and chooses the one that has the least number of uses and definitions. The benefit/cost associated with the spill is the profit of the optimization on registers.
To compute the profit of an optimization on overall performance $P_{\text{total}}$, the profitability engine needs to combine the effects of the optimization on registers, $R_{\text{total}}$ and computation, $C_{\text{total}}$. To compute $R_{\text{total}}$, the profitability engine sums the register profit associated with every step in the optimization. Similarly, to compute $C_{\text{total}}$, the profitability engine sums the computation profit for every step. Table 3 shows how to compute $R_{\text{total}}$ and $C_{\text{total}}$ for PRE, LICM and VN. For example, to compute the profit of eliminating a redundant expression in VN (3rd row in Table 3), the engine needs to compute the register profit, which includes the register profit of replacing the computation $preplacecom$ and updating of the uses of the defined variable $replaceuse$. Further, $R_{\text{replacecomp}}$ is computed by deleting a use, $R_{\text{deleteuse}}(\text{EXP}, S_x)$ and inserting a use, $R_{\text{insertuse}}(v, S_x)$. The engine also needs to compute the computation profit of replacing the computation $C_{\text{replacecomp}}$ (i.e., removing the computation and inserting a copy). However,
the inserted copy statement may be deleted later as a useless statement if it is not an argument of a $\Phi$-instruction. The engine should also consider the deletion. Thus, the engine multiplies $R_{\text{interm}}(v,S_i)$ and $C_{\text{insert}}(\text{copy},B_j)$ by a factor of $\alpha$. $\alpha$ is a number between zero and one and can be determined by profiling.

To combine the profits for registers and computation, they must have the same metric. If the computation profit considers the frequency of a node, the register profit also needs to consider the execution frequency of the loads or stores.

3.4 An Example of Selectively Applying VN

To illustrate how our framework works, we show an example of profit-driven VN applied to a code segment, shown in Figure 10 (a). Figure 10 (b) gives the corresponding register code model, where all the live ranges are expressed.

VN processes each block in the dominator tree. The first block processed will be $B_1$. Since none of the expressions in $B_1$ are in the hash table, the value number of the defined variables and the expressions will be the defined variables themselves. For example, VN[\(u_0\)] is $u_0$ and VN[\(a_0+b_0\)] is $u_0$.

The next block processed is $B_2$. Since the expression $c_0 + d_0$ is defined in block $B_1$, the first redundant expression, $x_0 \leftarrow c_0 + d_0$, is found. The optimizer calls the profitability engine to predict the profit of eliminating this redundancy. The profitability engine computes the profit on both registers and computation. To predict the profit on registers, the engine first takes the register code model (shown in Figure 10 (b)) and the VN optimization model. The engine generates the optimized code model using the incremental dataflow algorithm (shown in Table 1). In this case, $c_0$ and $d_0$ are deleted as uses. Because $c_0$ and $d_0$ are live after statement 4, there is no change on the register code model for the deletions. Also, $v_0$ is inserted as a use. Thus, the live range of $v_0$ is lengthened from $LR_{[2]}$ to $LR_{[2..4]}$. Figure 10 (d) shows the updated register code model after replacing this redundancy.

Using the register allocation optimization model, the engine determines how the spills changed based on the live range updates. For this example, there is no spill change from deleting $c_0$ and $d_0$. But inserting $v_0$ will increase the spills by one if the number of hardware registers is less than 8. Indeed, the number of live ranges at statement 3 changes from 7 to 8. Choosing which variable to spill depends on the register allocator’s spill strategy. In our register allocation model, we pick the one that has the fewest number of uses and definitions, which is $u_0$. This introduces a store before statement 2 and a load after statement 4. The cost associated with the inserted load and store is the profit on registers as predicted by the engine.

The profit on computation is more easily predicted, which includes the benefit of removing an add statement and the cost to insert a copy statement. To compute the overall profit, the profitability engine uses the functions described in the previous section. If the overall profit is positive, redundancy elimination is applied. Otherwise, it is not applied.

There are 6 redundant expressions that can be eliminated in this example. For every redundant expression, the profitability engine is triggered to predict the profit of applying the redundancy elimination. Figure 10 (e) shows the code after VN (assuming all 6 redundant expressions are profitable). The register code model after VN is shown in Figure 10 (f), where all the live ranges are changed except for $LR_{[2]}$. 
Figure 10. An example of profit-driven VN

(a) Code before VN

(b) Register code model before VN

(c) Code after 1st elimination

(d) Register code model after 1st elimination

(e) Code after VN

(f) Register code model after VN
4. EXPERIMENTAL RESULTS

To evaluate the effectiveness and usefulness of our framework, we implemented our models and the profitability engine for copy propagation, constant propagation, dead code elimination, PRE, LICM and VN. We integrated our models into the Mach SUIF compiler [SmHo02]. For the implementation, we used the dead code elimination pass from Mach SUIF, extended the PRE pass from Rolaz [Rolla03] and implemented copy propagation, constant propagation, LICM and VN. We compared the performance and compile-time of profit-driven PRE and LICM with always applying PRE and LICM and a heuristic-driven PRE and LICM. Because the heuristic used for PRE and LICM is not useful for VN, we compared profit-driven VN only with always applying VN.

For the experiments, we used several SPEC2K benchmarks (gzip, vpr, mcf, parser, vortex, and twolf). These are the only SPEC2K benchmarks that can be compiled by the current release of the Mach SUIF compiler. We used an AMD Athlon 1.4 GHz machine, with 2 GB of memory, running RedHat Linux. We performed node profiling on the training data sets with the HALT library (included in Mach SUIF) to get the frequency counts used in our models. In all experiments, each benchmark was run three times on a lightly loaded machine and the average execution time was computed to factor out system effects.

4.1 Heuristic-driven Approach

Always applying an applicable optimization can sometimes lead to a performance degradation. Such a simple heuristic of “always applying” is not sufficient in making decisions about when to apply an optimization. Previous work has focused on developing heuristics to decide when to apply optimizations, such as register pressure sensitive redundancy elimination, which sets upper limits on allowable register pressure and performs redundancy elimination within these limits [GuBo99]. We implemented a similar heuristic. We set the upper limit (i.e., a threshold) on allowable live ranges at the places where the redundant expressions will be moved. Redundancy elimination is performed only when the number of live ranges is within the limit. In VN, we eliminate full redundancies and there is no code movement. Thus, the heuristic described here is not useful for VN. In this section, we show the experimental results for heuristic-driven PRE and LICM.

One problem with a heuristic-driven approach is how to select a limit that can achieve good performance across all the benchmarks. Our experiments show that different benchmarks need different limits to achieve the best performance. Table 4 shows the performance improvement of heuristic-driven PRE and LICM over the baseline. The baseline compiler applies register allocation and simple instruction scheduling. Also, to enable more opportunities for PRE and LICM, we apply copy propagation, constant propagation and dead code elimination before applying PRE and LICM. We varied the limit on register pressure from zero to sixteen. For PRE, if the limit is zero, only full redundancies are eliminated. In practice, the limits are usually chosen to be the number of available hardware registers. Hence, eight may be a good limit because there are eight hardware registers that can be allocated for a byte-type variable on the x86. Four and sixteen are used to examine stricter or looser limits. In the table, the best performance improvement is shown in bold.

From the table, we can see that different benchmarks need different limits to perform the best. For example, for PRE, gzip can achieve an improvement of 4.1% when the limit is set to sixteen, while mcf needs the limit set to zero to achieve the best improvement of 2.37%. Also, some benchmarks are sensitive to the limit (e.g., twolf), while others are not (e.g., mcf). Further, we see that different optimizations may need different limits for the
same benchmarks. For example, gzip needs the limit set to sixteen for PRE but needs the limit set to eight for LICM. If we fix the limit, then we can not always achieve the best improvement with a heuristic.

Table 4. Improvement of heuristic-driven PRE and LICM with different limits

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>3.50</td>
<td>3.75</td>
<td>3.78</td>
<td>4.10</td>
<td>2.90</td>
<td>3.29</td>
<td>5.40</td>
<td>3.27</td>
</tr>
<tr>
<td>vpr</td>
<td>1.22</td>
<td>0.75</td>
<td>1.81</td>
<td>1.83</td>
<td>-0.40</td>
<td>-0.38</td>
<td>0.52</td>
<td>0.69</td>
</tr>
<tr>
<td>mcf</td>
<td>2.37</td>
<td>2.35</td>
<td>2.31</td>
<td>2.22</td>
<td>2.50</td>
<td>2.62</td>
<td>2.58</td>
<td>2.47</td>
</tr>
<tr>
<td>parser</td>
<td>1.25</td>
<td>1.50</td>
<td>1.70</td>
<td>1.35</td>
<td>2.55</td>
<td>2.86</td>
<td>1.99</td>
<td>2.23</td>
</tr>
<tr>
<td>vortex</td>
<td>4.73</td>
<td>5.25</td>
<td>4.66</td>
<td>3.86</td>
<td>4.88</td>
<td>5.69</td>
<td>4.99</td>
<td>5.28</td>
</tr>
<tr>
<td>bzip2</td>
<td>7.35</td>
<td>7.52</td>
<td>8.19</td>
<td>7.91</td>
<td>7.02</td>
<td>7.35</td>
<td>6.70</td>
<td>4.57</td>
</tr>
<tr>
<td>twolf</td>
<td>1.07</td>
<td>0.88</td>
<td>1.14</td>
<td>0.02</td>
<td>0.52</td>
<td>0.38</td>
<td>2.14</td>
<td>1.91</td>
</tr>
</tbody>
</table>

4.2 Performance benefit of profit-driven PRE, LICM and VN

Using our model-based framework, we can determine the profitability of an optimization and selectively apply it. The cases where optimizations degrade performance can be avoided. In this section, we first compare profit-driven PRE and LICM with always applying PRE and LICM and the heuristic-driven PRE and LICM. We then show a performance comparison between profit-driven VN and always applying VN.

Figures 11 and 12 show the performance benefit of profit-driven PRE and LICM over the baseline. In Figure 11, A-PRE is the improvement of always applying PRE when it is applicable. Heuristic-driven PRE is described as above and has two versions based on the register pressure allowed. Best-heuristic is the best case among the limits for each benchmark, while Heuristic-8 uses a fixed limit of eight. Lastly, P-PRE is the performance benefit of profit-driven PRE. Figure 12 shows the same configurations except for LICM.

As Figure 11 shows, the problem with always applying PRE when it is applicable is that it may increase register pressure, which may incur more spills and thus degrade performance. Both the heuristic approach and P-PRE can avoid the unprofitable instances of PRE. However, the selection of limits in the heuristics plays an important role in its performance (as described in Section 4.1). P-PRE considers both register pressure and
computation to predict the profitability of PRE and applies it accordingly, without requiring parameters to be tuned. It consistently performs as good as or better than the Best-Heuristic for PRE, except for bzip2, where predictions are sometimes incorrect.

Figure 12. Improvement of P-LICM compared with A-LICM and H-LICM

Figure 12 shows the performance benefit of the different approaches for applying LICM. Due to the increased register pressure caused by some instances of LICM, the overall performance of A-LICM can be improved by not applying the unprofitable ones. Although the heuristic-driven LICM achieves a performance improvement over always applying in some cases, it is important to choose the right limit. For example, in parser, with a register pressure limit of eight, the heuristic-driven LICM is worse than always applying. While in the best-heuristic, it is better than always applying LICM. P-LICM can perform at least as well as the best-heuristic LICM in most cases. However, in one case (gzip), due to incorrect predictions, P-LICM has worse performance than the heuristic-driven approach.

Figure 13. Improvement of P-VN compared with A-VN

Figure 13 shows the performance improvement of profit-driven VN over the baseline, compared with always applying VN. Unlike PRE and LICM, we did not apply other optimizations (e.g., copy propagation or constant propagation) before VN because VN eliminates the redundancies by value, not by name. Constant or copy propagation cannot enable more opportunities for VN. Always applying VN degraded the performance in
some cases because of the increased register pressure caused by eliminating some redundancies. For example, for vortex, the overall performance was degraded by 1.48%. However, using our framework, profit-driven VN can selectively apply only profitable redundancy elimination, achieving a performance benefit. For vortex, profit-driven VN improved performance by 1.69% over the baseline.

From these figures, we see that our framework is useful for a variety of optimizations, whether the optimization operates on SSA or non-SSA intermediate code formats. We conclude that a model-based approach can be effectively used to explore and determine the profitability of optimizations. The profitability property is useful in deciding when to apply optimizations.

4.3 Compile-time

Because our approach uses models to make decisions, we investigated how compile-time is impacted by profit-driven optimization. Tables 5, 6 and 7 show the compile-time for different optimization strategies for PRE, LICM and VN.

Table 5. Compile-time for PRE

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Compile time (seconds)</th>
<th>Percentage difference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A-PRE</td>
<td>H-PRE</td>
</tr>
<tr>
<td>gzip</td>
<td>42</td>
<td>45.14</td>
</tr>
<tr>
<td>vpr</td>
<td>128.38</td>
<td>193</td>
</tr>
<tr>
<td>mcf</td>
<td>20.89</td>
<td>29</td>
</tr>
<tr>
<td>parser</td>
<td>100.67</td>
<td>123</td>
</tr>
<tr>
<td>vortex</td>
<td>490.48</td>
<td>575.33</td>
</tr>
<tr>
<td>bzip2</td>
<td>33.77</td>
<td>42.6</td>
</tr>
<tr>
<td>twolf</td>
<td>755.55</td>
<td>1087</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From Table 5, the A-PRE compile-time varies from approximately 20 seconds to 755 seconds. The compile-time shown for the heuristic approach is the average for the different limits. It increases from 7% to 50% over A-PRE, with an average of 30%. Here the heuristic-driven PRE has to compute and update live range information. The compile-time for profit-driven PRE increases over A-PRE by 16% to 68%, with an average of 40%. Because P-PRE considers computation and register pressure in a more precise way than the heuristic-driven PRE, it incurs a modest overhead increase over the heuristic approach.

Table 6. Compile-time for LICM

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Compile-time (seconds)</th>
<th>Percentage Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A-LICM</td>
<td>H-LICM</td>
</tr>
<tr>
<td>gzip</td>
<td>47.8</td>
<td>59.04</td>
</tr>
<tr>
<td>vpr</td>
<td>128</td>
<td>147</td>
</tr>
<tr>
<td>mcf</td>
<td>20.8</td>
<td>27.6</td>
</tr>
<tr>
<td>parser</td>
<td>109.3</td>
<td>133.47</td>
</tr>
<tr>
<td>vortex</td>
<td>492.1</td>
<td>547.39</td>
</tr>
<tr>
<td>bzip2</td>
<td>38.59</td>
<td>48.55</td>
</tr>
<tr>
<td>twolf</td>
<td>591</td>
<td>817.76</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Similar compile-time trends can be seen for A-LICM, H-LICM and P-LICM. The A-LICM compile-time varies from approximately 20 seconds to 591 seconds. The heuristic-driven LICM increases compile-time over A-LICM from 11% to 38% (average 24%) and profit-driven LICM increases compile-time over A-LICM by 15% to 55% (average 32%). Finally, in comparison to heuristic-driven LICM, profit-driven LICM increases compile-time by an average of 6.2%.

Table 7. Compile-time for VN

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Compile-time (seconds)</th>
<th>Percentage Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A-VN</td>
<td>P-VN</td>
</tr>
<tr>
<td>gzip</td>
<td>46.93</td>
<td>54.33</td>
</tr>
<tr>
<td>vpr</td>
<td>127.06</td>
<td>146.30</td>
</tr>
<tr>
<td>mcf</td>
<td>25.57</td>
<td>29.41</td>
</tr>
<tr>
<td>parser</td>
<td>96.25</td>
<td>113.59</td>
</tr>
<tr>
<td>vortex</td>
<td>508.94</td>
<td>583.93</td>
</tr>
<tr>
<td>bzip2</td>
<td>28.35</td>
<td>33.28</td>
</tr>
<tr>
<td>twolf</td>
<td>283.35</td>
<td>331.03</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From Table 7, the compile-time of always applying VN varies from 25 seconds to 508 seconds. The profit-driven VN increases the compile-time over always applying VN from 14% to 18%, with an average of 16%. Compared with P-PRE and P-LICM, the compile-time increased by P-VN is smaller. One reason is that there are fewer instances of VN than PRE and LICM (shown in the next section). The overhead of the profit-driven approach depends on how many instances of the optimization appear in the code and the impact of every instance.

As the tables show, the increase in compile-time of our profit-driven approach is modest and about the same as the heuristic-driven approach. These small increases show that our approach is feasible and efficient. However, our prototype has several implementation artifacts that hurt performance; a production implementation could decrease the compile-time further. We conclude that the compile-time increase is worth the benefit of applying the optimizations more effectively.

4.4 Model Verification

We validated our models by determining their accuracy when predicting the profitability of an optimization. We validated the prediction accuracy by considering only registers. We did not evaluate the computation profit because the computation is exact in terms of instruction count, given relative node frequencies from a profile. If the relative frequencies in the profile do not match what happens in an actual run, then there can be an inaccuracy in predicting the computation profit. However, this inaccuracy is a property of the profile – not of the models to compute the computation profit.

For deciding when to apply optimizations, a correct prediction is one in which we predict there is a benefit/cost for registers (i.e., if register profit is positive, it indicates a spill reduction; otherwise, it shows a spill increase) and actual execution has the same result. For those cases where the actual execution shows there was no impact on registers, we consider the prediction to be correct. The accuracy prediction is measured by how often we make a correct prediction. To validate the prediction accuracy, we checked every prediction and compared the value predicted with the actual execution (i.e., we use the number of memory accesses before and after applying an optimization to reflect the spill changes).
Tables 8, 9 and 10 show the prediction accuracy of PRE, LICM and VN. In the tables, “TP” is the total number of predictions and “% accuracy” is the prediction accuracy for both heuristic-driven and profit-driven approaches. In the heuristic-driven PRE and LICM, we set the limit to eight.

Table 8. Prediction accuracy of H-PRE and P-PRE

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Heuristic-8 PRE</th>
<th></th>
<th>Profit-driven PRE</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TP</td>
<td>% accuracy</td>
<td>TP</td>
<td>% accuracy</td>
</tr>
<tr>
<td>gzip</td>
<td>43</td>
<td>79.07</td>
<td>48</td>
<td>89.58</td>
</tr>
<tr>
<td>vpr</td>
<td>290</td>
<td>80.34</td>
<td>303</td>
<td>96.04</td>
</tr>
<tr>
<td>mcf</td>
<td>51</td>
<td>88.23</td>
<td>51</td>
<td>86.27</td>
</tr>
<tr>
<td>parser</td>
<td>239</td>
<td>75.73</td>
<td>293</td>
<td>87.87</td>
</tr>
<tr>
<td>vortex</td>
<td>513</td>
<td>79.72</td>
<td>530</td>
<td>81.13</td>
</tr>
<tr>
<td>bzip2</td>
<td>58</td>
<td>81.03</td>
<td>56</td>
<td>78.57</td>
</tr>
<tr>
<td>twolf</td>
<td>484</td>
<td>76.03</td>
<td>475</td>
<td>91.12</td>
</tr>
<tr>
<td>average</td>
<td>80.02</td>
<td></td>
<td>87.23</td>
<td></td>
</tr>
</tbody>
</table>

As Table 8 shows, in some cases heuristic-driven PRE had a different number of predictions than profit-driven PRE because of the interactions among PRE instances. The prediction accuracy for heuristic-driven PRE varies from 75% to 88%, with an average of 80%. Compared with heuristic-driven PRE, profit-driven PRE makes more correct predictions generally, with the prediction accuracy from 78% to 96% (average 87%). Profit-driven PRE considers the impact on register pressure in a more precise way. In some cases, like mcf, although the prediction accuracy of P-PRE is lower than H-PRE, P-PRE achieves a better performance benefit than H-PRE because P-PRE also considers the computation.

A similar trend can be seen in Table 9 for LICM. The prediction accuracy for heuristic-driven LICM varies from 75% to 88%, with an average of 79%. Profit-driven LICM has a higher prediction accuracy, varying from 82% to 94% (average 88%). Because profit-driven PRE and LICM can make more correct predictions than the heuristic-driven approach, the performance improvement of P-PRE and P-LICM is generally better than Heuristic-8 PRE and Heuristic-8 LICM (shown in Figures 8 and 9). Table 10 shows the prediction accuracy of our framework for profit-driven VN. It varies from 81% to 94%, with an average of 87%.

Table 9. Prediction accuracy of H-LICM and P-LICM

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Heuristic-8 LICM</th>
<th></th>
<th>Profit-driven LICM</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TP</td>
<td>% accuracy</td>
<td>TP</td>
<td>% accuracy</td>
</tr>
<tr>
<td>gzip</td>
<td>53</td>
<td>88.68</td>
<td>45</td>
<td>84.44</td>
</tr>
<tr>
<td>vpr</td>
<td>251</td>
<td>75.70</td>
<td>230</td>
<td>94.35</td>
</tr>
<tr>
<td>mcf</td>
<td>68</td>
<td>76.47</td>
<td>52</td>
<td>82.69</td>
</tr>
<tr>
<td>parser</td>
<td>89</td>
<td>79.78</td>
<td>75</td>
<td>90.67</td>
</tr>
<tr>
<td>vortex</td>
<td>361</td>
<td>77.56</td>
<td>346</td>
<td>87.57</td>
</tr>
<tr>
<td>bzip2</td>
<td>92</td>
<td>82.60</td>
<td>88</td>
<td>89.77</td>
</tr>
<tr>
<td>twolf</td>
<td>367</td>
<td>77.93</td>
<td>345</td>
<td>88.70</td>
</tr>
<tr>
<td>average</td>
<td>79.82</td>
<td></td>
<td>88.31</td>
<td></td>
</tr>
</tbody>
</table>
Table 10. Prediction accuracy of P-VN

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Profit-driven VN</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TP</td>
<td>% accuracy</td>
</tr>
<tr>
<td>gzip</td>
<td>30</td>
<td>93.33</td>
</tr>
<tr>
<td>vpr</td>
<td>77</td>
<td>87.01</td>
</tr>
<tr>
<td>mcf</td>
<td>35</td>
<td>82.86</td>
</tr>
<tr>
<td>parser</td>
<td>32</td>
<td>84.38</td>
</tr>
<tr>
<td>vortex</td>
<td>71</td>
<td>94.37</td>
</tr>
<tr>
<td>bzip2</td>
<td>48</td>
<td>87.5</td>
</tr>
<tr>
<td>twolf</td>
<td>101</td>
<td>81.19</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td>87.23</td>
</tr>
</tbody>
</table>

On average, 12% of the time, our framework made inaccurate predictions. The inaccuracy is primarily from a simplified assumption used in the register optimization model about how the register allocator spills registers. The model assumes that the allocator will select the spill priority based solely on the number of uses and definitions in a live range. However, the Mach SUIF register allocator also uses the number of conflicting edges in the interference graph to make the decisions. Note that even without the detailed implementation information, our models achieve good prediction accuracy. If more accuracy is needed, the models can be improved by incorporating more implementation information.

5. RELATED WORK

There has recently been a flurry of research focusing on optimization properties. We categorize optimization properties as either semantic or application. Semantic properties deal with the semantics of the optimizations and include correctness, soundness and optimization specification. Application properties include profitability, interaction and automatic generation of the optimizations. There are two approaches to explore these properties. One is through formal techniques, which include developing formal specifications, analytic models, and proofs with model checking and theorem provers [LJWF02, LeMC03, Necu00, WhSo97, JaGS99, WhSo90]. Another approach is experimental, which is mostly used for exploring application properties. In the introduction, we indicated the previous work that uses this approach for determining the order and the configuration to apply optimizations.

In this paper, we extend our previous paper [ZhCS05] by unifying the model notations, providing a simpler interface for a compiler engineer to develop optimization models, adding a new optimization model for VN and presenting more experimental results. In our current research, we use analytical models to predict the profitability of optimizations. Thus, in this section, we focus on discussing the prior work that relates to using models (including analytical or experimental models) to address the problems of the application of optimizations. To our knowledge, ours is the first work that uses analytical models to predict the impact of scalar optimizations on registers and computation.

Our previous paper developed a framework that had code, loop optimization and cache models and demonstrated that the benefit of applying loop optimizations on cache could be predicted [ZhCS03]. The work relied on models that had already been developed for modeling the cache and array access sequences [GhMM99, HKVI02]. It did not consider scalar optimizations, registers or computation. In this paper, we develop a more powerful and general framework that has a profitability engine as well as models and thus can be used for many types of optimizations.
An approach to discover a best optimization configuration uses an analytic model of machine resources to statically estimate the performance of the optimized code instead of executing it [TVVA03]. However, because optimizations are not modeled in this approach, they still need to be applied.

Another approach is to select an optimization level to recompile the methods based on an experimental resource model [AFGH00, HoUn96]. The optimizer uses a simple benefit-cost analysis to decide whether to recompile a method at a higher optimization level. The benefit of an optimization level is estimated as a constant by offline experiments. However, this model does not include some aspects of optimization behavior (e.g., the effect of optimizations depends on the code context).

The last approach uses analytic models of code, optimizations and resources [WoLa91, WoMC96, Pugh91, McCT96, CoMc95, Sark97, CCCM91, KCRB99, GhMM99, SaMe00, YLRC03]. The idea is to use a resource cost model (e.g., cache cost) and optimization models (e.g., unimodular matrix transformations) to select a program-specified sequence or configuration to apply optimizations that maximizes the benefit. These techniques demonstrate that analytic models are efficient in driving the application of optimizations. However, all these techniques use models that express only a small set of optimizations (loop optimizations and data optimizations) and mainly attack a single problem; i.e., to improve the performance of cache [CoST02].

Another relative work is the register pressure sensitive PRE [GuBo99]. It sets upper limits on allowable register pressure and then performs redundancy elimination within these limits. In this paper, we develop independent models of optimizations, while register pressure sensitive PRE uses data flow analysis to determine register pressure, which is integrated with the PRE algorithm and only works for PRE. They also do not consider the impact of PRE on computation.

6. CONCLUSIONS
In this paper, we presented a novel model-based framework that can be used to predict the profitability of scalar optimizations. This work coupled with prior work, which considered loop optimizations, has a wide range of applicability in terms of optimizations and machine resources. Our model-based technique can make accurate predictions without applying and executing the optimized code. Using the framework, the optimizer can selectively apply an optimization based on whether the optimization is profitable or not. Since we do not have to actually apply the optimizations or run the resulting code, we can also use the framework for faster searches over different optimization sequences to determine an effective optimization order. Although the focus in this paper was on exploring the profitability property, other properties can be explored using the model-based approach. For example, we can explore the interaction property of optimizations. Thus, using models, a good sequence of optimizations can be found without the added expense of applying and then removing the optimization (undoing the optimization or storing two versions of the code).

ACKNOWLEDGEMENTS
This research is supported in part by the National Science Foundation, Next Generation Software, grants CNS-0305198 and CNS-0203945. Bruce R. Childers was also partially supported on this work by an IBM Faculty Partnership Award and DARPA contract.
REFERENCES


