An Approach Toward Profit-Driven Optimization

MIN ZHAO and BRUCE R. CHILDERS
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, in certain circumstances, optimizations may degrade performance. However, there is no efficient way to know when a degradation will occur. 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

Authors' addresses: Min Zhao and Bruce R. Childers, Department of Computer Science, University of Pittsburgh, Pittsburgh, PA, 15260; Mary Lou Soffa, Department of Computer Science, University of Virginia, Charlottesville, VA, 22904.

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 [Briggs and Cooper 1994; Zhao et al. 2003]. 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) [Whitfield and Soffa 1997; Triantafyllis et al. 2003; Almagor et al. 2004; Kulkarni et al. 2004]. 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 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. In general, heuristics can work well. 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 [Cooper et al. 2001; Kisuki et al. 2000; Almagor et al. 2004; Kulkarni et al. 2004; Chen et al. 2005]. 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.

Because of 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;
- 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 future work. Section 7 concludes the paper.

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 new models for the code, optimizations, or machine resources are needed, they can be developed and easily added into the framework.

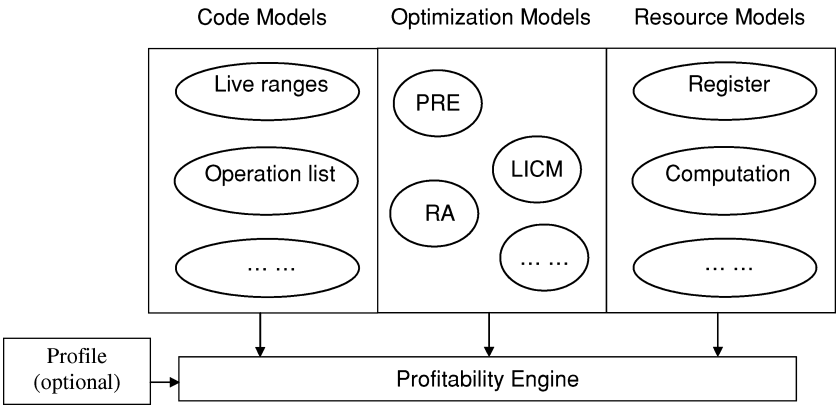

Fig. 1. A model-based framework for predicting the profitability.

2.1 Code Models

The code model expresses those characteristics of the code segment that are changed by an optimization and impact a machine 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 [Zhao et al. 2005a]. In this paper, we focus on scalar optimizations, which have negligible effect on cache (i.e., loop behavior dominates cache performance [McKinley et al. 1996]) 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 [Duesterwald et al. 1997]) 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 nonstructural 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 [Bivens and Soffa et al. 1990]. 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 [Chaiten 1982] or linear scan [Poletto and Sarkar 1989]. 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 one for 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. The profit engine can also use profile information (e.g., the basic block frequencies) in computing the profits.

For example, assume the impact of an optimization on registers is desired. The engine inputs the register code model, an optimization model, a register

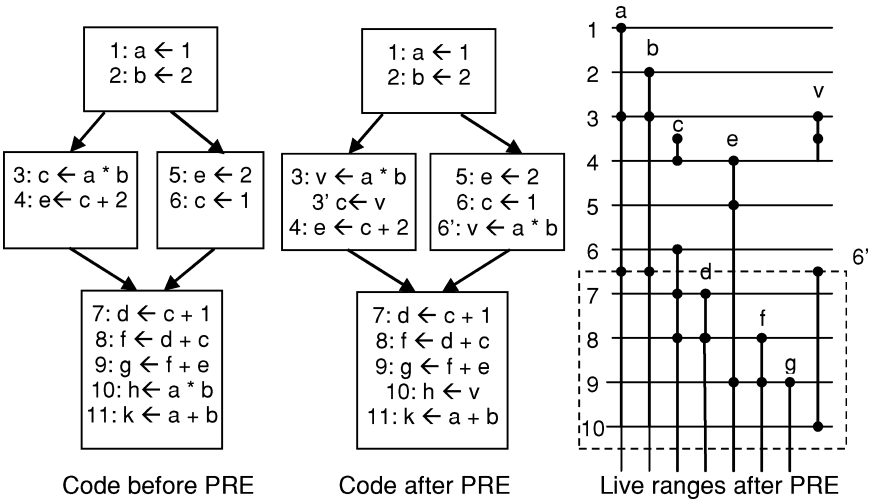

Fig. 2. An example of PRE impacting registers.

allocation model, and a register resource model. It then 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 data-flow algorithm [Pollock and Soffa 1989]. 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. The PRE algorithm is lazy code motion, which

inserts the computation as late as possible [Knoop et al. 1992]. 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 seven 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.

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 $LR^{x}_{[n,\ldots,m]}$, where x is a variable name and $[n, \ldots, m]$ 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 $LR^{v}_{[3..4,\ 6'..10]}$, where [6'..10] is a shorthand to represent a contiguous range. When a variable v is defined outside a loop at n and used inside the loop at m, we still use $[n, \ldots, m]$ to represent its live range for the simplicity of the notation. However v's live range includes the whole loop.

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 $\langle fB1, fB2, \ldots, fBn\rangle^{op}$, where f_{Bi} is the number of op in block B_i and opappears in blocks $B_1, B_2, \ldots, B_n$.

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 "$x \Downarrow v$," 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 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 [Knoop et al. 1992]. Although LCM considers register pressure, there are still cases where PRE introduces more register spills (as shown in Figure 2).

Eliminate the partial redundant expression EXP (y op z) at S_s

Insert a statement:

$S_d' = S_d + 1$

Insert < DEF v USE y, z OP op > @ S_d'

Replace the computation:

Delete < USE y, z OP op > @ S_s

Insert < USE v OP $copy$ > @ S_s

Update the same expressions:

$\forall T$ | $T = w \leftarrow EXP(y\ op\ z)$ at S_w

Delete < DEF w > @ S_w

Insert < DEF v > @ S_w

$S_w' = S_w + 1$

Insert < DEF w USE v OP $copy$ > @ S_w'

Fig. 3. PRE optimization model.

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 statement "$a \leftarrow b + 1$" is moved out of the loop body, because each of its operands is either defined outside of the loop or a constant.

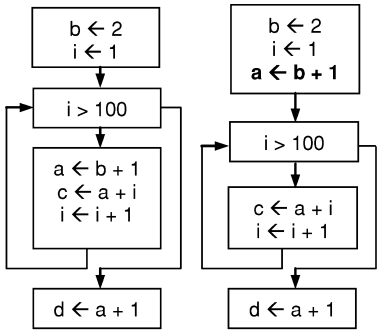

Fig. 4. An example of LICM.

Move a loop invariant statement $x \leftarrow y$ op z

Insert a statement:
$S_d' = S_d + 1$
Insert < DEF x USE y, z OP op > @ S_d'

Delete the statement:
Delete < DEF x USE y, z OP op > @ S_s

Fig. 5. LICM optimization model.

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 lexically 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.

We model dominator-based VN, which is a global technique that uses hashing to discover redundant computations and to fold constants [Briggs et al. 1997]. It works on Static Single Assignment (SSA) intermediate code. An example of VN is shown in Figure 6. Because the expression "$d_0 + c_0$" at statement 4 has the same value number as "$a_0 + c_0$" at statement 2, it is redundant and can be replaced by the destination of "$a_0 + c_0$". Thus, statement 4 is replaced by a copy from b_0 to e_0.

VN has three actions for a basic block: (1) it removes redundant or meaninglessΦ-instructions (Φ-instruction is a pseudo-assignment that introduces a new definition point at the merge point in the control-flow graph [Briggs et al. 1998]); (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

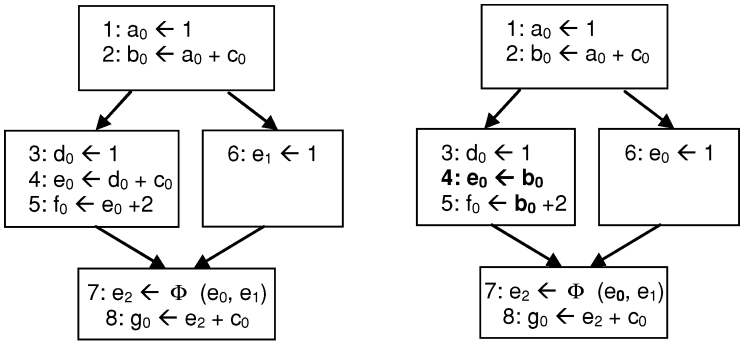

Fig. 6. An example of VN.

Φ-instructions should be replaced by copy instructions in predecessor blocks. Because the inputs of theΦ-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 [Briggs et al. 1998] 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 *EXP* (y op z) at S_s is redundant, it is replaced by a copy from its value number v. That is, the variables of *EXP* are deleted as uses with the operation *op* at S_s. The expression's value number v is inserted as a use with the operation *copy* at S_s. All uses of the defined variable x are also replaced by v. 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 *EXP* (y op z) at S_s can be simplified by constant folding, *EXP* is deleted. The third rule shows if a redundant Φ-instruction is deleted, all the uses of the defined variable x are also 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, which is inserted in the replacing the computation step. Here, the variable y is deleted as a use and

#1: Replace a redundant statement $x \leftarrow y$ op z with $x \leftarrow$ VN[x] at S_s
Replace the computation:
Delete < USE y, z OP op > @ S_s
Insert < USE v OP $copy$ > @ S_s
Replace all uses of x with its value number v:
$\forall$u | u is use of x at S_u
Delete < USE x > @ S_u
Insert < USE v > @ S_u

#2: Fold constant a statement $x \leftarrow y$ op z at S_s
Delete the computation:
Delete < USE y, z OP op > @ S_s

#3: Delete a redundant Φ -instruction $x \leftarrow \Phi$ ($x_1, x_2, \ldots$)
Replace all uses of x with its value number v:
$\forall$u | u is use of x at S_u
Delete < USE x > @ S_u
Insert < USE v > @ S_u

#4: Delete a useless copy instruction $x \leftarrow y$ at S_s
Delete the copy instruction:
Delete < DEF x USE y OP $copy$ > @ S_s

Fig. 7. VN optimization model.

the defined variable x is deleted as a definition with the operation *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 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 because of 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 $LR^c_{[n,\ldots,m]}$, we determine how spills are changed. If $LR^c_{[n,\ldots,m]}$is inserted or lengthened, it may introduce one more spill. Within the range $[n, \ldots, m]$, if the insertion of a new live range causes the number of live ranges to exceed the number of available hardware registers (HR), we select a live range to spill to memory, which introduces more loads and stores. We use $LR^{all}_{[n,\ldots,m]}$to represent the live ranges in $[n, \ldots, m]$. 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, $LR^s_{[n,\ldots,m]}$is selected to be spilled. If there is no definition of s before a use of s or there is no use

Determine how spill changes for every live range change $LR^{c}_{[n,\ldots,m]}$

IF Inserted($LR^{c}_{[n,\ldots,m]}$) ∪ Lengthened($LR^{c}_{[n,\ldots,m]}$)

  IF $| LR^{all}_{[n,\ldots,m]} + LR^{c}_{[n,\ldots,m]} | > | HR |$

    Select { $LR^{s}_{[n,\ldots,m]}$ } → MEM

    $\forall d \mid d$ is definition of s at $S_d \cap S_d \in [n,\ldots,m]$

      Insert < OP *store* > @ S_d

    $\forall u \mid u$ is use of s at $S_u \cap S_u \in [n,\ldots,m]$

      Insert < OP *load* > @ S_u

ELSE

  IF $| LR^{all}_{[n,\ldots,m]} - LR^{c}_{[n,\ldots,m]} | \leqslant | HR |$

    Select { $LR^{s}_{[n,\ldots,m]}$ } → MEM

    $\forall d \mid d$ is definition of s at $S_d \cap S_d \in [n,\ldots,m]$

      Delete < OP *store* > @ S_d

    $\forall u \mid u$ is use of s at $S_u \cap S_u \in [n,\ldots,m]$

      Delete < OP *load* > @ S_u

Fig. 8. Register allocation optimization model.

of s within the range $[n, \ldots, m]$, a store or load is inserted at the boundary of $[n, \ldots, m]$. If the boundary of $[n, \ldots, m]$ is within a loop, a store or load is inserted outside the loop. Otherwise, at all the uses or definitions of $s$within $[n, \ldots, m]$, a load or store will be inserted. Alternatively, if $LR^{c}_{[n,\ldots,m]}$ 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, resource models, and profiles. It then 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.

From an optimization model, the profitability engine determines how the optimization changes the register code model with an incremental data-flow algorithm [Pollock and Soffa 1989]. Table I 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 $LR^{v}_{[n,\ldots,m]}$, is lengthened. If the inserted use at statement s is the last use (i.e., s postdominates other uses), the new live range for v becomes $LR^{v}_{[n,\ldots,s]}$. Otherwise, the new live range consists of the original live range and a range to the use statement s. This range is represented as $LR^{v}_{[n,\ldots,m,\ldots,s]}$.

Table I. Incremental Computation of the New Register Code Model

Code Change	Incrementally Compute the New Register Code Model
Insert a use of variable v at statement s	IF v *is live at post-s* THEN no change; ELSE /* *lengthen v's live range**/ The original live range $LR^v_{[n,\ldots,m]}$ changes to $LR^v_{[n,\ldots,m]\cup[n,\ldots,s]} = \begin{cases} LR^v_{[n,\ldots,s]} & s \text{ postdominate other uses} \\ LR^v_{[n,\ldots,m,\ldots,s]} & \text{otherwise} \end{cases}$
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 $LR^v_{[n,\ldots,m]}$ changes to $LR^v_{[n,\ldots,m]\cap[s,\ldots,m]} =$ $\begin{cases} LR^v_{[s,\ldots,m]} & s \text{ postdominate other definition} \\ LR^v_{[n,\ldots,s,\ldots,m]} & \text{otherwise} \end{cases}$
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 $LR^v_{[n,\ldots,s]}$ changes to $LR^v_{[n,\ldots,s]\cap[n,\ldots]} = \begin{cases} LR^v_{[n,\ldots,m]} & m \text{ postdominate other uses} \\ LR^v_{[n,\ldots,m,\ldots,]} & \text{otherwise} \end{cases}$
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 $LR^v_{[s,\ldots,m]}$ changes to $LR^v_{[s,\ldots m]\cup[\ldots,m]} = \begin{cases} LR^v_{[n,\ldots,m]} & n \text{ postdominate other definition} \\ LR^v_{[\ldots,n,\ldots,m]} & \text{otherwise} \end{cases}$
Delete an edge from block B_s to bloc B_d	Delete all uses of any variable that is live at the beginning of B_d from the B_s and all predecessors of B_s where the variable is no longer live by any path.
Insert an edge from block B_s to bloc B_d	Insert all uses of any variable that is live at the beginning of B_d to the B_s and all predecessors of B_s.

Table II. Updates of the Computation Code Model

Code Change	Update the Computation Code Model
Insert an operation *op* at block B_s	The original operation list $\langle fB_1, fB_2, \ldots, fB_s, \ldots, fB_n\rangle^{op}$ changes to $\langle fB_1, fB_2, \ldots, fB_s + 1, \ldots, fB_n\rangle^{op}$
Delete an operation *op* at block B_s	The original operation list $\langle fB_1, fB_2, \ldots, fB_s, \ldots, fB_n\rangle^{op}$ changes to $\langle fB_1, fB_2, \ldots, fB_s - 1, \ldots, fB_n\rangle^{op}$

The profitability engine also infers how an optimization changes the computation code model. As shown in Table II, the code changes from an optimization that can be classified as either inserting 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_{Bs}) is increased by one. If an operation *op* is deleted at a block B_s, $f_{Bs}$is decreased by one. Thus, the profitability engine can determine the impact of an optimization on the computation.

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.

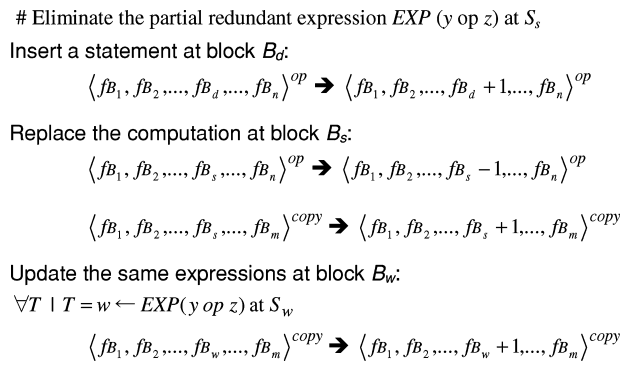

Fig. 9. Impact of PRE on computation code model.

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_{total}, the profitability engine needs to combine the effects of the optimization on registers, R_{total} and computation, C_{total}. To compute R_{total}, the profitability engine sums the register profit associated with every step in the optimization model. Similarly, to compute C_{total}, the profitability engine sums the computation profit for every step. Table III shows how to compute R_{total} and C_{total} for PRE, LICM, and VN. For example, to compute the profit of eliminating a redundant expression in VN (3rd row in Table III), the engine needs to compute the register profit, which includes the register profit of replacing the computation $R_{replacecomp}$ and updating of the uses of the defined variable $R_{replaceuse}$. Further, $R_{replacecomp}$ is computed by deleting a use, $R_{deleteuse}(EXP, S_s)$ and inserting a use, $R_{insertuse}(v, S_s)$. The engine also needs to compute the computation profit of replacing the computation $C_{replacecomp}$ (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 Φ-instruction. The engine should also consider the deletion. Thus, the engine multiplies $R_{insertuse}(v, S_s)$ and $C_{insert}(copy, B_s)$ by a factor of α. α 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.

Table III. Computation of Profitability for Registers (R_{total}) and Computation (C_{total})

Optimization	Compute the Profit on Registers and Computation
PRE: eliminate a redundant expression	$R_{total} = R_{insertstat} + R_{replacecomp} + R_{update\,\exp}$ $= R_{insertuse}(EXP, S_d) + R_{insertdef}(v, S_d)$ $+R_{deleteuse}(EXP, S_s) + R_{insertuse}(v, S_s)$ $+\sum_w (R_{deletedef}(w, S_w) + R_{insertdef}(v, S_w)$ $+R_{insertdef}(w, S_w + 1) + R_{insertuse}(v, S_w + 1))$ $C_{total} = C_{insertstat} + C_{replacecomp} + C_{update\,\exp}$ $= C_{insert}(op, B_d)$ $+ C_{delete}(op, B_s) + C_{insert}(copy, B_s)$ $+\sum_w C_{insert}(copy, B_w)$
LICM: move an invariant statement	$R_{total} = R_{insertstat} + R_{deletestat}$ $= R_{insertuse}(EXP, S_d) + R_{insertdef}(x, S_d)$ $+R_{deleteuse}(EXP, S_s) + R_{deletedef}(x, S_s)$ $C_{total} = C_{insertstat} + C_{deletestat}$ $= C_{insert}(op, B_d) + C_{delete}(op, B_s)$
VN: eliminate a redundant expression	$R_{total} = R_{replacecomp} + R_{replaceuse}$ $= R_{deleteuse}(EXP, S_s) + \alpha \times R_{insertuse}(v, S_s)$ $+\sum_u (R_{deleteuse}(x, S_u) + R_{insertuse}(v, S_u))$ $C_{total} = C_{replacecomp} + C_{replaceuse}$ $= C_{delete}(op, B_s) + \alpha \times C_{insert}(copy, B_s)$
VN: fold constant a statement	$R_{total} = R_{deletecomp}$ $= R_{deleteuse}(EXP, S_s)$ $C_{total} = C_{deletecomp}$ $= C_{delete}(op, B_s)$

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 10a. Figure 10b 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 10b) and the VN optimization model. The engine generates the optimized code model using the incremental data-flow algorithm (shown in Table I). In this case, c_0 and d_0 are deleted as uses. Because c_0 and d_0 are

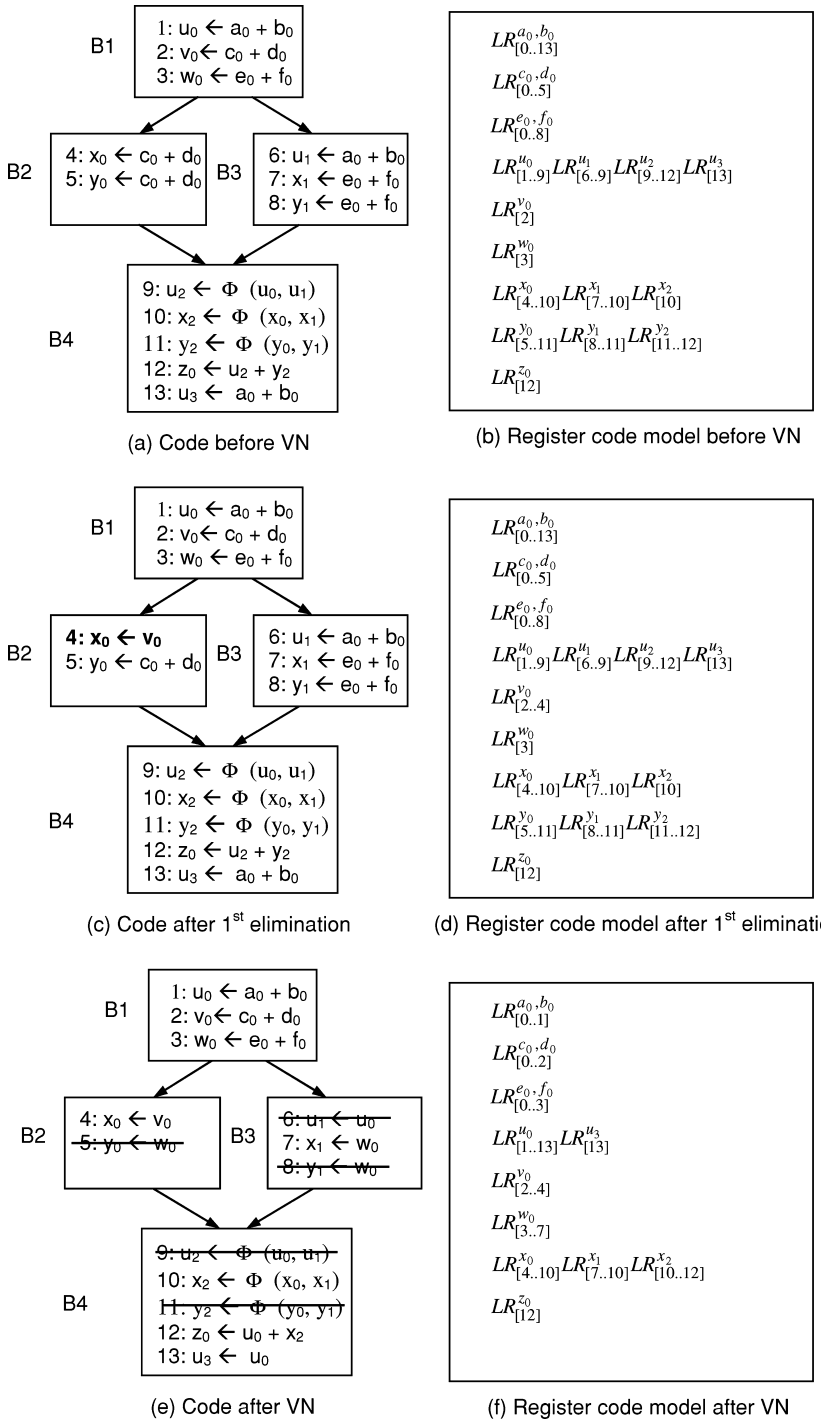

Fig. 10. An example of profit-driven VN.

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 $LR^{v_0}_{[2]}$to $LR^{v_0}_{[2..4]}$. Figure 10d 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. However, 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 six 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 10e shows the code after VN (assuming all six redundant expressions are profitable). The register code model after VN is shown in Figure 10f, where all the live ranges are changed except for $LR^{z_0}_{[12]}$.

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 [Smith and Holloway]. For the implementation, we used the dead-code elimination pass from Mach SUIF, extended the PRE pass from Rolaz, 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*) and Mibench benchmarks (*bitcount*, *dijkstra*, *fft*, *jpeg* and *sha*). We run our experiments on a Pentium III 1.4G machine, with 512 MB of memory and an AMD Athlon MP 1800 1.4 GHz machine, running RedHat Linux. The experimental results show the same trend for both machines. For brevity, we only report the results on the Pentium III machine. Complete results can be found in [Zhao et al. 2005b]. We performed node profiling on the training data sets with the HALT library (included in Mach SUIF) to get the basic block frequency counts used in the profitability engine. In all experiments, each

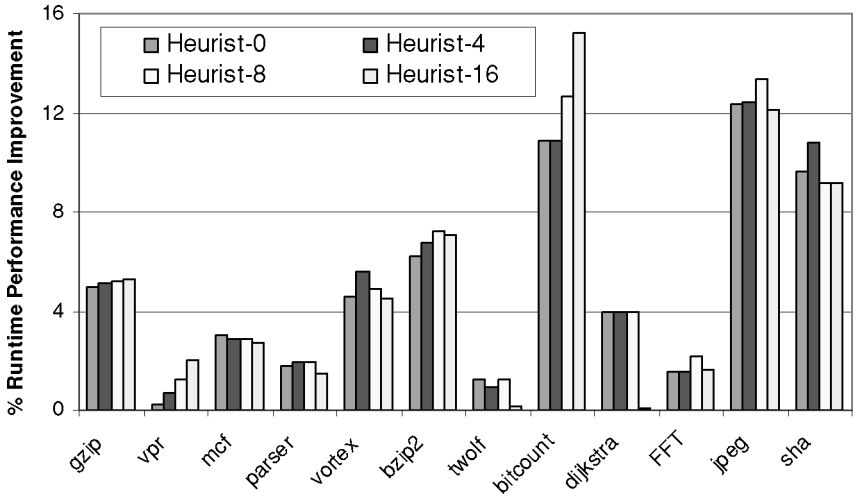

Fig. 11. Improvement of heuristic-driven PRE with different limits.

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 [Gupta and Bodik 1999]. We implemented a similar heuristic. We set the upper limit 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. Figures 11 and 12 show the runtime performance improvement of heuristic-driven PRE and LICM over the baseline. The baseline compiler applies register allocation and simple instruction scheduling. To enable more opportunities for PRE and LICM, we also 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

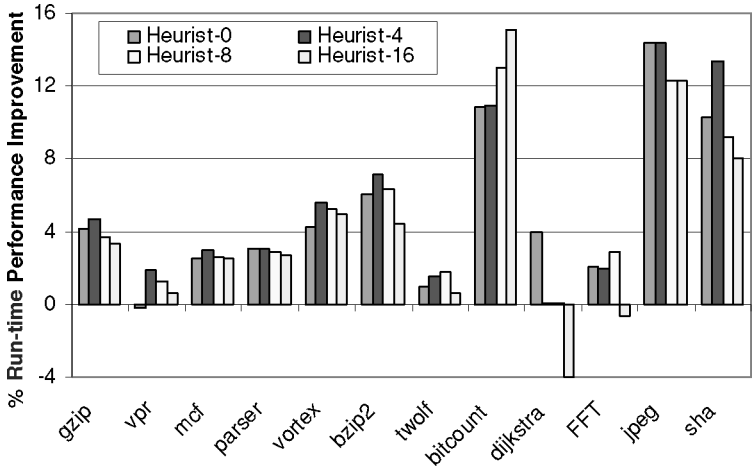

Fig. 12. Improvement of heuristic-driven LICM with different limits.

because there are eight hardware registers that can be allocated for a byte-type variable on x86. Four and sixteen are used to examine stricter or looser limits.

From the figures, we can see that different benchmarks need different limits to perform the best. For example, for PRE, *gzip* can achieve an improvement of 5.25% when the limit is set to sixteen, while *mcf* needs the limit set to zero to achieve the best improvement of 3.01%. Also, some benchmarks are sensitive to the limit (e.g., bitcount), 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 four for LICM. If we fix the limit, then we can not always achieve the best improvement with a heuristic.

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 compare profit-driven VN and always applying VN.

Figures 13 and 14 show the comparisons of several PRE approaches, in terms of the improvement in the dynamic number of memory accesses and run-time performance over the baseline. We also compared the improvement in dynamic instruction count for the different approaches. It shows the same trend as the run-time performance improvement. Thus, we omit the results for brevity (see [Zhao et al. 2005b] for all the results). In the figures, 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 15 and 16 show the comparisons with the same configurations except for LICM.

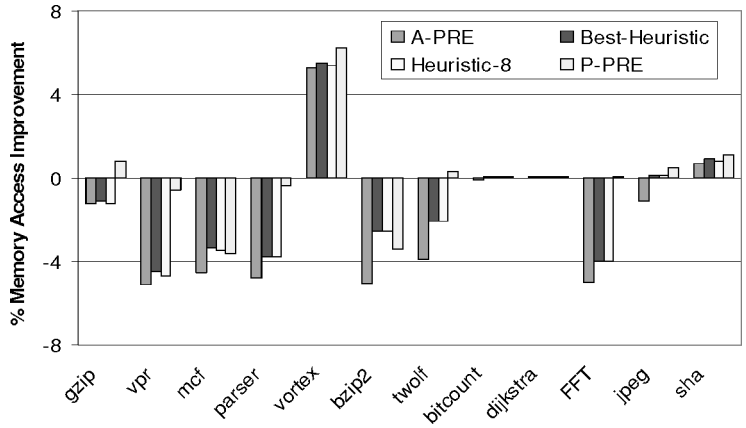

Fig. 13. Memory access improvement for PRE.

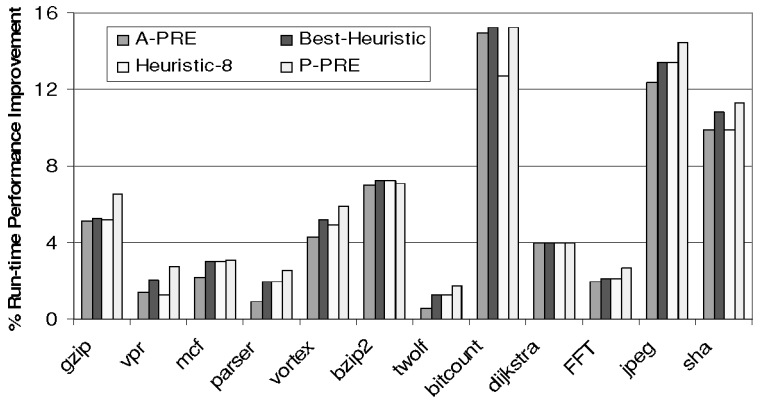

Fig. 14. Run-time performance improvement for PRE.

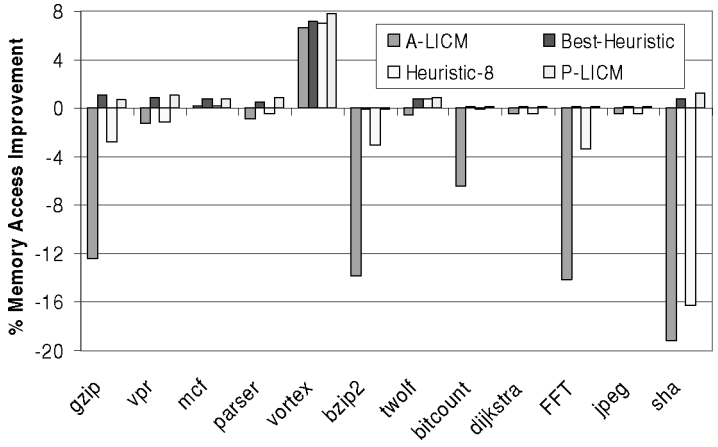

Fig. 15. Memory access improvement for LICM.

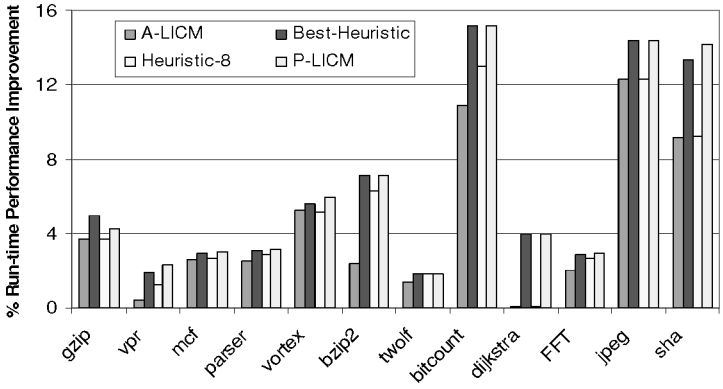

Fig. 16. Run-time performance improvement for LICM.

As Figure 13 shows, the problem with always applying PRE when it is applicable is that it may increase register pressure and incur more spills. In most cases, A-PRE increases the number of memory accesses. For example, in *vpr*, A-PRE increases the memory accesses by 5.11%. Both the heuristic approach and P-PRE can avoid the unprofitable instances of PRE, thus decreasing the memory accesses. However, P-PRE considers the registers in a more accurate way (as demonstrated by the prediction accuracy in Section 4.4). It improves the memory access count more than the heuristic approach. For example, in *gzip*, the best-heuristic increases the memory access by 1.1%, while P-PRE decreases the memory accesses by 0.82%. Because of the mispredictions, P-PRE increases the memory accesses more than the heuristic approach for *mcf* and *bzip2*.

Figure 14 shows the run-time performance improvement for different PRE approaches over the baseline. Both H-PRE and P-PRE achieve performance benefits over always applying PRE. However, the choice of the limits in heuristic-PRE (H-PRE) is very important (as described in Section 4.1). For example, in *vortex*, when the limit is set to four, H-PRE improves performance by 5.61%. While when the limit is eight, H-PRE improves performance by 4.89%. P-PRE considers both register pressure and computation to predict the profitability of PRE. Thus, in the case where P-PRE increases memory accesses more than H-PRE (*mcf*), P-PRE still improves the overall run-time performance. P-PRE consistently performs as good as or better than the best-heuristic for PRE, except for *bzip2*, where predictions are sometimes incorrect. In the cases where P-PRE decreases the number of memory accesses, it improves the run-time performance more (e.g., *gzip, twolf*, and *jpeg*). That is, the performance benefit comes from the careful consideration of register pressure. On a register limited machine, like x86, it is particularly important to consider the register pressure as these results indicate.

Figures 15 and 16 show a comparison of the different approaches for applying LICM. As shown in Figure 15, A-LICM can increase register pressure greatly. For example, in *sha*, A-LICM increases the memory accesses by 19.17%. Heuristic LICM and profit-driven LICM selectively choose profitable LICM instances

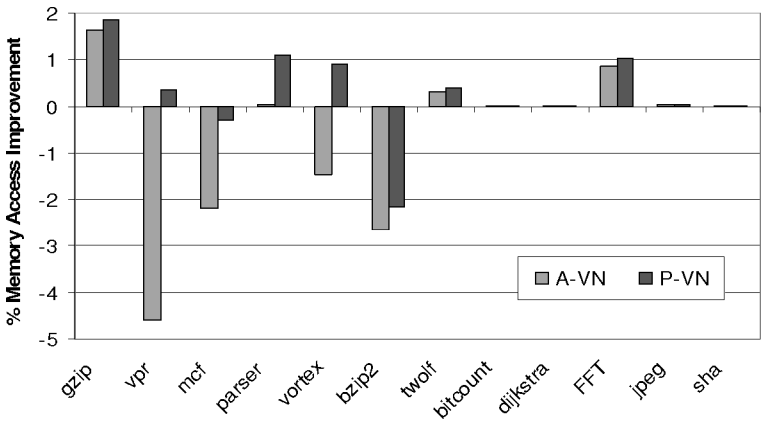

Fig. 17. Memory access improvement for VN.

to apply. Thus, in *sha*, best-heuristic LICM decreases the memory accesses by 0.74% and P-LICM decreases the accesses by 1.24%.

Figure 16 shows the run-time performance improvement for different LICM approaches over the baseline. From the figure, we can see that the overall performance of A-LICM can be improved by not applying unprofitable ones. Although the heuristic-driven LICM achieves a performance improvement over always applying LICM, it is important to choose the right limit. For example, in *vortex*, with a register pressure limit of eight, the heuristic-driven LICM is worse than always applying LICM. While in the best-heuristic (where the limit is sixteen), it is better than always applying LICM. P-LICM can perform at least as well as the best-heuristic LICM in most cases, without tuning the parameters used in H-LICM. However, in one case (*gzip*), because of incorrect predictions, P-LICM has worse performance than the heuristic-driven approach.

Figures 17 and 18 show the improvement of memory accesses and run-time performance of profit-driven VN over the baseline, compared to 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 redundancies by value, not by name. Constant or copy propagation cannot enable more opportunities for VN. Always applying VN-degraded performance, in some cases, because of the increased register pressure, caused by eliminat some redundancies, as shown in Figure 17. For example, for *vortex*, A-VN increases the memory accesses by 1.46% and, thus, the run-time performance was degraded by 1.37%. However, using our framework, profit-driven VN can selectively apply only profitable redundancy elimination, achieving a performance benefit. For *vortex*, profit-driven VN decreases the memory accesses by 0.91% and, thus, improves run-time performance by 1.28% 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.

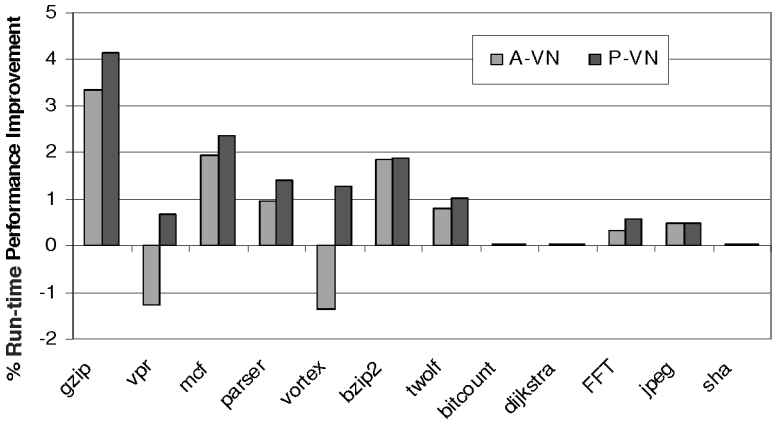

Fig. 18. Run-time performance improvement for VN.

Table IV. Compile-Time for PRE

	Full Compile-Time			One Pass Compile-Time		
Benchmark	A-PRE	H over A%	P over A%	A-PRE	H over A%	P over A%
gzip	44.99	9.18	17.63	10.44	36.78	65.90
vpr	142.46	52.23	61.86	37.61	77.45	103.56
mcf	21.84	37.36	48.49	4.68	57.39	72.91
parser	106.74	25.10	34.00	26.7	69.06	94.23
vortex	518.5	19.11	29.64	88.49	56.78	79.76
bzip2	35.58	22.85	27.15	10.77	68.25	86.56
twolf	767.27	46.05	58.24	199.49	90.29	104.82
bitcount	6.59	7.13	10.93	1.79	56.98	61.45
dijkstra	1.15	11.30	13.91	0.29	24.14	48.28
FFT	4.61	8.89	13.02	1.07	41.12	55.14
jpeg	35.08	40.34	53.62	7.49	80.32	104.74
sha	3.04	10.53	15.13	0.66	21.21	36.36
average	–	24.17	31.97	–	56.65	76.14

4.3 Compile-Time

Because our approach uses models to make decisions, we investigated how compile-time is impacted by profit-driven optimization. We need to ensure that evaluating the models does not overly increase compile-time. Tables IV, V, and VI show the compile-time for different optimization strategies for PRE, LICM, and VN. In the tables, the compile-time for all compilation passes, including the front-end, optimizations, and back-end passes ("Full Compile-Time"), and for the optimization pass under consideration ("One Pass Compile-Time") are shown.

From Table IV, the full compile-time for A-PRE varies from approximately 1.2 to 767.3 s. The compile-time shown for the heuristic approach is the average for the different limits. It increases from 7 to 52% over A-PRE, with an average of 24%. Here the heuristic-driven PRE has to compute and update live range information, which causes the compile-time increase. The compile-time for profit-driven PRE increases over A-PRE by 11 to 62%, with an average of

Table V. Compile-Time for LICM

	Full Compile-Time			One Pass Compile-Time		
Benchmark	A-LICM	H over A (%)	P over A (%)	A- LICM	H over A (%)	P over A (%)
gzip	45.97	23.65	27.65	12.94	57.26	69.63
vpr	127.84	18.80	27.35	32.36	58.19	79.49
mcf	20.51	32.42	39.10	4.73	49.89	72.94
parser	106.08	21.86	30.82	29.53	58.42	88.93
vortex	511.8	11.34	15.48	98.87	36.41	47.25
bzip2	34.63	22.81	30.26	11	57.55	79.55
twolf	579.97	37.73	55.50	165.49	88.14	132.64
bitcount	6.63	4.52	7.39	1.88	16.49	25.53
dijkstra	1.19	7.56	10.08	0.35	11.43	14.29
FFT	4.58	35.37	41.48	1.21	60.33	85.12
jpeg	25.26	20.23	28.82	6.38	56.99	70.82
sha	2.78	17.63	25.90	0.81	38.27	54.32
average	–	21.16	28.32	–	49.11	68.38

Table VI. Compile-Time for VN

	Full Compile-Time		One Pass Compile-Time	
Benchmark	A-VN	P over A (%)	A-VN	P over A (%)
gzip	47.02	15.82	6.82	26.83
vpr	127.93	14.88	18.17	26.25
mcf	25.98	15.97	3.61	22.44
parser	97.2	17.78	13.56	33.48
vortex	511.68	14.72	61.95	27.44
bzip2	28.59	17.59	3.47	48.99
twolf	284.34	16.93	40.4	34.16
bitcount	7.33	12.55	1.81	26.52
dijkstra	1.67	13.17	0.25	24.00
FFT	5.66	17.49	0.84	44.05
jpeg	29.11	15.94	4.27	37.24
sha	3.58	12.29	0.55	27.27
average	–	15.43	–	31.56

32%. 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 IV also shows compile-time for only the PRE optimization pass. The one-pass compile-time for A-PRE varies from approximately 0.3 to 199.49 s. The compile-time for H-PRE increases from 21 to 90% over A-PRE, with an average of 57%. The compile-time for P-PRE increases over A-PRE by 36 to 105%, with an average of 76%.

Similar compile-time trends can be seen for A-LICM, H-LICM, and P-LICM in Table V. The full compile-time for A-LICM varies from approximately 1.2 to 579.9 s. The heuristic-driven LICM increases compile-time over A-LICM from 5 to 38% (average 21%) and profit-driven LICM increases compile-time over A-LICM by 7 to 56% (average 28%). The one-pass compile-time for A-LICM varies from approximately 0.35 to 165.49 s. The compile-time for H-LICM increases from 11 to 88% over A-LICM, with an average of 49%. The compile-time for P-PRE increases over A-PRE by 14 to 132%, with an average of 68%.

From Table VI, the full compile-time for A-VN varies from 1.7 to 512 s. The profit-driven VN increases the compile-time over always applying VN from 12 to 18%, with an average of 15%. The one pass compile-time for A-VN is from 0.25 to 21 s. The P-VN increase compile-time over A-VN from 22 to 49%, with an average of 32%. 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 without tuning parameters.

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 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 VII and VIII 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 and profit-driven approaches. In the heuristic-driven PRE and LICM, we set the limit to eight.

As Table VII 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 100%, with an average of 82.5%. Compared with heuristic-driven PRE, profit-driven PRE generally makes more correct predictions, with the prediction accuracy from 78 to 100% (average 92.6%). Profit-driven PRE considers the impact on register pressure in a more precise way. In some cases, like *mcf*,

Table VII. Prediction Accuracy of H-PRE and P-PRE

Benchmark	Heuristic-8 PRE		Profit-Driven PRE	
	TP	% accuracy	TP	% accuracy
gzip	43	79.07	48	89.58
vpr	290	80.34	303	96.04
mcf	51	88.23	51	86.27
parser	239	75.73	293	87.87
vortex	513	79.72	530	81.13
bzip2	58	81.03	56	78.57
twolf	484	76.03	475	91.12
bitcount	5	100	5	100
dijkstra	2	100	2	100
FFT	3	33	3	100
jpeg	58	96.55	58	100
sha	5	100	5	100
average	–	82.48	–	92.55

Table VIII. Prediction Accuracy of H-LICM and P-LICM

Benchmark	Heuristic-8 LICM		Profit-Driven LICM	
	TP	% accuracy	TP	% accuracy
gzip	53	88.68	45	84.44
vpr	251	75.70	230	94.35
mcf	68	76.47	52	82.69
parser	89	79.78	75	90.67
vortex	361	77.56	346	87.57
bzip2	92	82.60	88	89.77
twolf	367	77.93	345	88.70
bitcount	3	66.67	3	100
dijkstra	5	40	5	80
FFT	23	86.96	23	95.65
jpeg	82	97.56	79	100
sha	21	76.19	21	95.24
average	–	77.18	–	90.76

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 (shown in Figure 14).

A similar trend can be seen in Table VIII for LICM. The prediction accuracy for heuristic-driven LICM varies from 40 to 97%, with an average of 77. Profit-driven LICM has a higher prediction accuracy, varying from 82 to 100% (average 91%). 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. Table IX shows the prediction accuracy of our framework for profit-driven vn. It varies from 81 to 100%, with an average of 87%. In some cases, VN had no effect, so no accuracy is reported (i.e., *bitcount*, *dijkstra*, and *sha*).

On average, our framework made inaccurate predictions 10% of the time. The inaccuracy is primarily from a simplified assumption used in the register optimization model about how the register allocator spills registers. The

Table IX. Prediction Accuracy of P-VN

	Profit-Driven VN	
benchmark	TP	% accuracy
gzip	30	93.33
vpr	77	87.01
mcf	35	82.86
parser	32	84.38
vortex	71	94.37
bzip2	48	87.5
twolf	101	81.19
bitcount	0	–
dijkstra	0	–
FFT	4	75
jpeg	1	100
sha	0	–
average	–	87.29

model assumes that the allocator will select the spill priority based solely on the number of uses and definitions in a live range. However, Mach SUIF's register allocator also uses the number of conflicting edges in the interference graph to make spill decisions. Even without detailed implementation information, our models achieve good accuracy. If greater accuracy is needed, the models can be improved by incorporating more implementation information. In our framework, the prediction inaccuracy also does not accumulate. The profitability engine incrementally updates the code models. The incremental update is accurate. That is, the updated code model is the same as performing the optimization and reconstructing the code models. The inaccuracy of the prediction only comes from computing the profit associated with every update in an optimization. Thus, the prediction of an optimization does not impact the prediction accuracy of later optimizations.

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 [Lacey et al. 2002; Lerner et al. 2003; Necula 2000; Whitfield and Soffa 1997; Jaramillo et al. 1999; Whitfield and Soffa 1990]. 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 [Zhao et al. 2003] 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 [Zhao et al. 2003]. The work relied on models that had already been developed for modeling the cache and array access sequences [Ghosh et al. 1999; Hu et al. 2002]. 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 [Triantafyllis et al. 2003, 2005]. Our approach is different, because we also model the code and the optimizations. Although optimization writers have to develop models for each optimization, they are valuable because they enable us to predict profitability without actually applying them. Thus, there is no need to roll back an optimization if not profitable. To determine whether an optimization should be applied or not using Triantafyllis' approach, one needs to apply the optimization in consideration, followed by register allocation and then estimate the performance using profile information. Applying register allocation for every optimization and the optimization itself is expensive. Instead, we use models and an incremental approach to determine the impact of the optimizations. The benefit of the incremental approach has been demonstrated in the previous work. Thus, the compile-time overhead of prediction in our approach is less than Triantafyllis' approach.

Another approach is to select an optimization level to recompile the methods based on an experimental resource model [Arnold et al. 2000; Hölzle and Ungar 1996]. 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 [Wolf and Lam 1991; Wolf et al. 1996; Pugh 1991; McKinley et al. 1996; Coleman and McKinley 2005; Sarkar 1997; Chandramouli et al. 2001; Kandemir et al. 1999; Ghosh et al. 1999; Sarkar and Megiddo 2000; Yotov et al. 2003]. 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 [Cooper et al. 2001].

Another related work is the register pressure-sensitive PRE [Gupta and Bodík 1999]. 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. FUTURE WORK

The goal of our work is to use models to systematically investigate optimization properties and to find a better way to apply optimizations. In this paper, we present the models that can be used to predict the profitability of scalar optimizations. Based on the profit, the optimizer can selectively apply profitable optimizations. Our model-based approach can be used for other problems regarding the application of optimizations. In the future, we can extend our work in the following ways.

6.1 Using the Profitability Models

One recent research direction uses heuristic search to find program-specific optimization sequences [Almagor et al. 2004; Kulkarni et al. 2004]. In this approach, a large number of optimization sequences are applied and each is evaluated. When the evaluation involves dynamic measures (e.g., dynamic instruction count or cycle count), the execution of the program for all sequences is required. Thus, the search time can be significant. With our model-based approach, we do not have to actually apply the optimizations or run the resulting code to evaluate the profitability. We can use our framework to improve the search performance by avoiding the execution of the programs. Because the prediction of an optimization does not impact the prediction accuracy of the later optimizations (as described in section 4.4), the models are suitable for predicting the profit of a sequence of optimizations.

6.2 Modeling More Resources and Optimizations

In this paper, the resources that we model are registers and computation without code scheduling. In our previous work, we modeled the cache [Zhao et al. 2003]. Our models for the registers are more suitable for x*86* and other processors where there are few registers. In the future, we may need to model more resources based on different machine architectures (for example, computation with code scheduling). To predict the profit on computation with code scheduling, a code model (e.g., dependence graph), a resource model, and an optimization model for code scheduling are needed. The profitability engine should also be able to infer the changes of an optimization on the computation code model directly from the optimization model. For some architecture, we may also need to combine all the resources (cache, registers, and computation) to make more accurate predictions.

6.3 Modeling Optimization Interactions

Another important problem is to model optimization interactions. Optimizations can interact with each other through shared resources. For example, in our experiments, we found a case where applying one optimization prevents the application of another because the first one increases the register pressure. If the second optimization has more benefits, we will miss the performance improvement opportunity. If we can detect the interfering transformations, order their profitability, and make decisions considering the interactions, we will achieve more performance benefits. Interactions also interact with each other in enabling and disabling ways. To determine the enabling and disabling interactions, we need to model the conditions under which an optimization can be applicable (i.e., precondition) and the impact of the optimization (i.e., postcondition). The existence of optimization interactions also depends on code context, which should be modeled as well. The optimization interactions can then be used to drive the search for a program-specific optimization sequence.

7. 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. We implemented the framework for predicting the profitability of several scalar 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 predicted by using models and it is useful for selectively applying optimizations.

ACKNOWLEDGMENTS

This research is supported in part by the National Science Foundation, Next Generation Software, grants CNS-0305198 and CNS-0203945. We are grateful to the reviewers for their helpful suggestions to improve the paper.

REFERENCES

ALMAGOR, L., COOPER, K., GROSUL, A., HARVEY, T., REEVES, S., SUBRAMANIAN, D., TORCZON, L., AND WATERMAN, T. 2004. Finding effective compilation sequences. *ACM Conference on Languages, Compilers, and Tools for Embedded Systems*.

ARNOLD, M., FINK, S., GROVE, D., HIND, M., AND SWEENEY, P. F. 2000. Adaptive optimization in the Jalapeño JVM. *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

BIVENS, M. P. AND SOFFA, M. L. 1990. Incremental register reallocation. *Software Practice and Experience, 20*, 10 (Oct.).

BRIGGS, P. AND COOPER, K. D. 1994. Effective partial redundancy elimination. *In Proceedings of SIGPLAN'94 Conference on PLDI*.

BRIGGS, P., COOPER, K., AND SIMPSON, L. T. 1997. Value numbering. *Software Practice and Experience 27*, 6 (June).

BRIGGS, P., COOPER, K. D., HARVEY, T. J., AND SIMPSON, L. T. 1998. Practical improvements to the construction and destruction of static single assignment form. *Software Practice and Experience 28*, 8 (July).

CHANDRAMOULI, B., CARTER, J., HSIEH, W., AND MCKEE, S. 2001. A cost framework for evaluating integrated restructuring optimizations. *International Conference on Parallel Architectures and Compilation Techniques*, Barcelona, Spain, September.

CHAITIN, G. 1982. Register allocation and spilling via graph coloring. *ACM SIGPLAN Symposium on Compiler Construction*, June.

CHEN, C., CHAME, J., AND HALL, M. 2005. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. *International Symposium on Code Generation and Optimization*, March.

COLEMAN, S. AND MCKINLEY, K. S. 1995. Tile size selection using cache organization and data layout. *In Proceedings of SIGPLAN'95 Conference on Programming Language Design and Implementation, La Jolla, CA*, June.

COOPER, K., SUBRAMANIAN, D., AND TORCZON, L. 2001. Adaptive optimizing compilers for the 21st century. *Proceedings of the 2001 LACSI Symposium*, October.

DUESTERWALD, E., GUPTA, R., AND SOFFA, M. L. 1997. Practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems 19*, 6 (Nov.).

GHOSH, S., MARTONOSI, M., AND MALIK, S. 1999. Cache miss equations: A compiler framework for analyzing and tuning behavior. *ACM Transactions on Programming Languages and Systems 21*, 4 (July).

GUPTA, R. AND BODÍK, R. 1999. Register pressure sensitive redundancy elimination. *8th International Conference on Compiler Construction*, March.

HU, J. S., KANDEMIR, M., VIJAYKRISHNAN, N., IRWIN, M. J., SAPUTRA, H., AND ZHANG, W. 2002. Compiler directed cache polymorphis. *Proceedings of LCTES/SCOPES*, June.

HÖLZLE, U. AND UNGAR, D. 1990. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems 18*, 4 (July).

JARAMILLO, C., GUPTA. R., AND SOFFA, M. L. 1999. Comparison checking: An approach to avoid debugging of optimized code. *Proceedings of Foundation of Software Engineering.*

KANDEMIR, M., RAMANUJAM, J., AND CHOUDHARY, A. 1999. Improving cache locality by a combination of loop and data transformations. *IEEE Transactions on Computers 48*, 2 (Feb.).

KULKARNI, P., HINES, S., HISER, J., WHALLEY, D., DAVIDSON, J., AND JONES, D. 2004. Fast searches for effective optimization phase sequences. *SIGPLAN'04 Conference on Programming Language Design and Implementation.*

KISUKI, T., KNIJNENBURG, P. M. W., AND O'BOYLE, M. F. P. 2000. Combined selection of tile size and unroll factors using iterative compilation. *International Conference on Parallel Architectures and Compilation Techniques.*

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1992. Lazy code motion. In *Proceedings of SIGPLAN'92 Conference on Programming Language Design and Implementation*, San Francisco.

LACEY, D., JONES, N. D., WYK, E., AND FREDERIKSEN, C. C. 2002. Proving correctness of compiler optimizations by temporal logic. *ACM Symposium on Principles of Programming Languages.*

LERNER, S., MILLSTEIN, T., AND CHAMBERS, C. 2003. Automatically proving the correctness of compiler optimizations. *SIGPLAN'03 Conf. on Programming Language Design and Implementation.*

MCKINLEY, K., CARR, S., AND TSENG, C. 1996. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems 18*, 4 (July).

NECULA, G. C. 2000. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation,* Vancouver, British Columbia, June.

POLLOCK, L. AND SOFFA, M. L. 1989. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering 15*, 12 (Dec.).

POLETTO, M. AND SARKAR, V. 1999. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems 21*, 5 (Sept.).

PUGH, W. 1991. Uniform techniques for loop optimization. In *Proceedings of 5th International Conference on Supercomputing*, Cologne, West Germany, June.

ROLAZ, L. 1990. An implementation of lazy code motion for machSUIF. Web URL: *http://lapwww.epfl.ch/dev/machsuif/opt_passes/lcm.pdf*

SARKAR, V. AND MEGIDDO, N. 2000. An analytic model for loop tiling and its Solution. *International Symposium on Performance Analysis of Systems and Software,* April.

SARKAR, V. 1997. Automatic selection of high-order transformations in the IBM XL FORTRAN compilers. *IBM Journal of Research and Development*, May.

SMITH, M. D. AND HOLLOWAY, G. 1990. An introduction to machine SUIF and its portable libraries for analysis and optimization. URL: *http://www.eecs.harvard.edu/hube/software/nci/overview.html*

TRIANTAFYLLIS, S., VACHHARAJANI, M., AND AUGUST, D. I. 2005. Compiler optimization-space exploration.*The Journal of Instruction-level Parallelism (JILP)*, February.

TRIANTAFYLLIS, S., VACHHARAJANI, M., VACHHARAJANI, N., AND AUGUST, D. I. 2003. Compiler optimization-space exploration. *Int'l. Symposium on Code Generation and Optimization*, March.

WHITFIELD, D. AND SOFFA, M. L. 1990. An approach to ordering optimizing transformations. *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*.

WHITFIELD, D. AND SOFFA, M. L. 1997. An Approach for Exploring Code Improving Transformations. *ACM Transactions on Programming Languages 19*, 6 (Nov.).

WOLF, M. AND LAM, M. 1991. A data locality optimizing algorithm. In *Proceedings of SIGPLAN'91 Conference on Programming Language Design and Implementation*, Toronto, Canada.

WOLF, M. E., MAYDAN, D. E., AND CHEN, D. 1996. Combining loop transformations considering caches and scheduling. *International Symposium on Mircoarchitecture, Paris, France*, December.

YOTOV, K., LI, X., REN, G., CIBULSKIS, M., DEJONG, G., GARZARAN, M., PADUA, D., PINGALI, K., STODGHILL, P., AND WU, P. 2003. A comparison of empirical and model-driven optimization. Scheduling. In *Proceedings of SIGPLAN'03 Conference on Programming Language Design and Implementation*, San Diego, CA.

ZHAO, M., CHILDERS, B., AND SOFFA, M. L. 2003. Predicting the impact of optimizations for embedded systems. *ACM Conf. On Languages, Compilers, and Tools for Embedded Systems*, June.

ZHAO, M., CHILDERS, B., AND SOFFA, M. L. 2005. A model-based framework: An approach for profit-driven optimization. *International Symposium on Code Generation and Optimization*, March.

ZHAO, M., CHILDERS, B., AND SOFFA, M. L. 2005. Profit-driven optimization.*University of Pittsburgh, Department of Computer Science, Technical Report TR-05-129*, November.

Received May 2005; revised October 2005 and April 2006; accepted May 2006