To appear in the ACM SIGMICRO Symposium on Code Generation and Optimization, March 2005

A Model-based Framework: an Approach for Profit-driven Optimization

Min Zhao
University of Pittsburgh
lilyzhao@cs.pitt.edu

Bruce R. Childers
University of Pittsburgh
childers@cs.pitt.edu

Mary Lou Soffa
University of Virginia
soffa@cs.virginia.edu

Abstract

Although optimizations have been applied for a number of years to improve the performance of software, problems that have been long-standing remain, which include knowing what optimizations to apply and how to apply them. To systematically tackle these problems, we need to understand the properties of optimizations. In our current research, we are investigating the profitability property, which is useful for determining the benefit of applying an optimization. Due to the high cost of applying optimizations and then experimentally evaluating their profitability, we use an analytic model framework for predicting the profitability of optimizations. In this paper, we target scalar optimizations, and in particular, describe framework instances for Partial Redundancy Elimination (PRE) and Loop Invariant Code Motion (LICM). We implemented the framework for both optimizations and compare profit-driven PRE and LICM with a heuristic-driven approach. Our experiments demonstrate that a model-based approach is effective and efficient in that it can accurately predict the profitability of optimizations with low overhead. By predicting the profitability using models, we can selectively apply optimizations. The model-based approach does not require tuning of parameters used in heuristic approaches and works well across different code contexts and optimizations.

1. Introduction

The field of optimization has been extremely successful over the past 40+ years. As new languages and new architectures have been introduced, novel and effective optimizations have been developed to target and exploit both the software and hardware innovations. Many reports from research and commercial projects have indicated that the performance of software improves significantly through aggressive optimizations.

Most successes in the field have come from the development of particular optimizations, such as loop optimizations and path sensitive optimizations. Although there were several long-standing problems with optimizations, they have not been adequately addressed because optimizations were yielding performance improvements. These problems included knowing what optimizations to apply and when, where, in which order (i.e., phase ordering) and in which configuration (e.g., the tile size in loop tiling) to apply them for the best improvement.

A number of events are occurring that demand solutions to these problems. First, 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 highest 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 certain properties of optimizations. Currently, it is unclear when and where to apply optimizations dynamically and how aggressive optimization can be and still be profitable after factoring in the cost of applying the optimization. Last, although new optimizations continue to be developed and applied, the performance improvement is shrinking. The question then is whether the optimization field has reached its limit or do further improvements depend on solutions to these problems. We believe the latter is true.

Traditionally, heuristics have been used to address some of the challenges 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 the 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 properties of optimizations, especially operational properties. We define optimization properties as either semantic or operational. Semantic properties deal with the semantics of the optimizations and include correctness, soundness and optimization specification. Operational properties target the application

of optimizations and include profitability and interaction of optimizations. Although research on many of these properties has been limited, there has recently been a flurry of activity focusing on optimization properties. There are two approaches to explore the properties of optimizations. One is through formal techniques, which include developing formal specifications, analytic models, and proofs through model checking and theorem provers [20, 24, 33, 15, 32]. Another approach is experimental. That is, the properties are evaluated by actually applying optimizations and executing the optimized code. This approach is mostly used for exploring operational properties, which are useful for determining when, where and how to apply optimizations [31, 9, 6, 17, 1, 18].

Because of the high cost of applying optimizations and experimentally evaluating their properties [9, 18], our research focuses on formally investigating operational properties of optimizations through analytic models. With analytic models, we can study, for example, the profitability of optimizations. Also our goal is to model the interactions among optimizations and then use the models to predict the impact of a sequence of optimizations without actually applying them.

In this paper, as a step toward our goal, we present a framework of analytic models for exploring the profitability of optimizations. In particular, we address the specific problems of how scalar optimizations impact registers, computation (i.e., functional units) and overall performance. A number of research efforts have shown that applying an optimization can degrade performance [4, 36]. To avoid this degradation, we use our framework to first predict the profitability of applying an optimization at a program point. Then based on whether there is a profit or not, we either apply it or not. The profitability of optimizations depends on code context, particular optimizations and machine resources, all of which need to be modeled. Thus, the framework includes models for code context, optimizations and resources. As part of the framework, we have a Profitability Engine that uses the models to predict the profitability of applying an optimization at any code point where it is applicable.

We developed models for a number of optimizations including copy propagation, constant propagation, dead code elimination, Partial Redundancy Elimination (PRE) and Loop Invariant Code Motion (LICM). In this paper, we focus on the models for PRE and LICM. Models for the other optimizations are useful when considering the impact of a sequence of optimizations, which is beyond the scope of this paper. We implemented the models and the profitability engine for both optimizations and compared profit-driven PRE and LICM with a heuristic-driven approach. Our experiments demonstrate that a model-based approach is effective and efficient in that it can accurately predict the profitability of optimizations with reasonable overhead. By determining the profitability, we can intelligently select profitable optimizations to apply in a systematic way.

The contributions of this paper include:

- A conceptual framework for investigating optimization profitability. The framework includes analytic code models, optimization models, and resource models for cache, registers and computation, and a profitability engine that uses the models to determine the performance profit.
- An implementation of the framework for scalar optimizations (in particular PRE and LICM) that uses the profitability of PRE and LICM to determine when to apply them.
- An experimental evaluation demonstrating that the model-based approach for predicting the profitability of optimizations is effective and efficient.
- A general model-based technique that can be used to study properties of optimizations.

2. Conceptual Profitability Framework

To determine the profitability of an optimization, we require models that are useful for predicting the impact of the optimization on performance. Performance is generally affected by cache, registers and functional units. Thus, we need to be able to determine the profit of an optimization for each resource and then combine the profits**.** Importantly, to determine the profitability, we do not require exact numbers but numbers "**accurate enough**" that the right decision as to when to apply an optimization can be made.

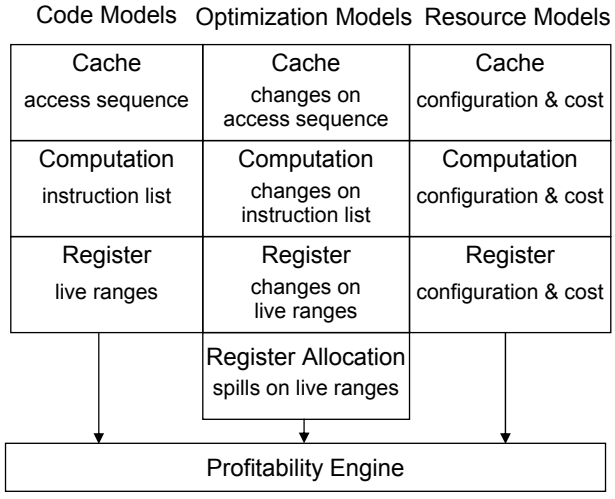


Figure 1. Profit-driven optimization framework

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 resources considered are cache, registers and functional units. The focus of this paper is the performance profit (i.e., execution time). However, other resources, such as code size and power/energy can also be modeled and included in the framework. Register allocation is an optimization but it also plays a part in determining the impact of other optimizations on registers. Thus, an optimization model for register allocation is shown separately in Figure 1.

2.1. Code Models

The code model expresses those characteristics of the code segment that are changed by an optimization and impact a resource. For example, array access sequences affect the cache so the code model would specify the access sequences. Live ranges can be changed by an optimization and impact the registers so the code model would express live range information for the code segment. Computation is also affected by an optimization and the code model is the list of instructions.

When safe conditions for applying an optimization are detected, a model of the code is 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 [10]) 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 of an optimization and its impact on the resources under consideration. Similar to code models, each optimization can have multiple models, one for each resource affected by the optimization. For example, if an optimization impacts registers and computation, then there is a model for that optimization's effect on registers and another one for its effect on computation.

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 small editing changes on a control flow graph. These edits are Insert/Delete an instruction (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 [25]. Thus, the changes for a particular optimization can be expressed as a series of basic edits. For example, constant propagation can be expressed as "Delete variable *v* at statement *P*; Insert constant *c* at statement *P*".

To determine the impact of the optimizations on registers, an optimization model for the register allocator must be developed. 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 [7, 11] or linear scan [26]. 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 resource, which describes the resource configuration and benefit/cost information in using the resource. This 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 the caches, the cache configuration and the cost of a cache miss/hit are needed. The functional unit model has the computational instructions available in the architecture and their execution latencies. 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 models (generated by the optimizer), the optimization models (developed offline by the compiler engineer), and resource models (developed offline for particular resources) are input into the Profitability Engine. This engine uses the information in the models to compute the profit of an optimization at a program point where it is safe. The profit can be computed for one resource or for combined resources. From the code and the optimization models, the engine determines the changes on the code models caused by the application of an optimization. It does this without applying the optimization. It then uses the resource models to determine the impact of the changes on the resource.

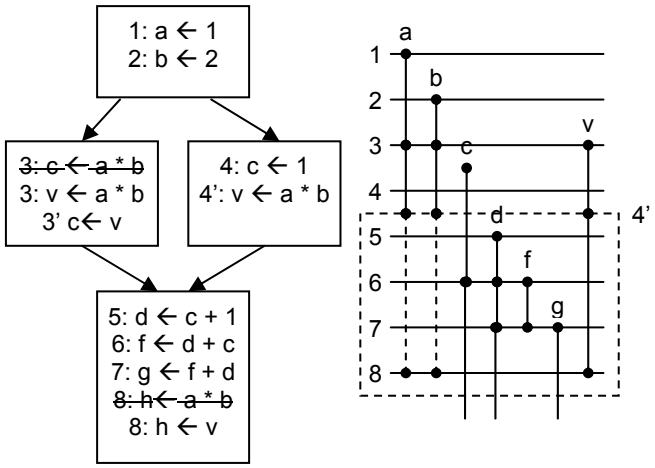
Table 1. Incremental updates of live ranges

Code Change	Update code model
Insert a use u of variable v in block B at statement s	If *v is live at post-s*: no change; Else */* insertion lengthen v's live range*/* If *there is a use or definition before s in the block B:* no change to global code model; record the local change; Else: add v to IN(B) and all reachable predecessors of B;
Insert a definition d of variable v in block B at statement s	If *v is not live at pre-s:* no change; Else /* insertion shorten v's live range*/ If *there is a use or definition before s in the block B:* no change to global code model; record the local change; Else delete v from IN(B) and all reachable predecessors of B;
Delete a use u of variable v in block B at statement s	If *there is a use after s in the block B:* no change; Else /* delete shorten v's live range*/ If *there is a use or definition before s in the block B:* no change to global code model; record the local change; Else delete v from IN(B) and all reachable predecessors of B;
Delete a definition d of variable v in block B at statement s	If *v is not live at pre-s:* no change; Else /* insertion lengthen v's live range*/ If *there is a use or definition before s in the block B:* no change to global code model; record the local change; Else add v to IN(B) and all reachable predecessors of B;
Insert an edge from Bs to Bd	Add all variables in IN(Bd) to the Bs and all reachable predecessors of Bs;
Delete a edge from Bs to Bd	Delete all variables in IN(Bd) from the Bs and all reachable predecessors of Bs;

For example, assume the impact of an optimization on registers is desired. The engine inputs the code model for registers, a model for this optimization, an optimization model for register allocation, and a resource model of registers. Then it determines the changes on the live ranges (i.e., the code model for registers), using an incremental dataflow algorithm [25]. Since an optimization models its changes by basic edits [3], the engine takes the edits and computes the changes in live ranges using Table 1. The table describes how the code changes of an optimization affect live ranges. In this table, *pre-s* means the point immediately before statement *s* while *post-s* means the point immediately after statement *s*. For example, the effects on live ranges of inserting a use of *v* (1st row) depend on the current code. If *v* is already live at *post-s*, there is no change. If there is a use in the block of *u* before *s*, then the only change is to the local live range of *v*. Otherwise, the live range of *v* has changed and *v* has to be added to the set of live variables at the beginning of the block, IN, and all reaching predecessors. And then the profitability engine uses a register allocation model to determine the spills (i.e., loads and stores) caused by these live range changes. The last step for the engine is to use the number of spills and where these loads and stores are inserted or deleted to compute the profit.

3. Framework for Scalar Optimizations

In this section, we describe an instance of our framework for predicting the profitability of scalar optimizations, and in particular, PRE and LICM. Since scalar optimizations have negligible effects on cache (i.e., loop behavior dominates cache performance [22]), we do not further consider cache models but only registers and computation.



PRE decreases the number of register spills by one
(Assume there are 5 available hardware registers)

Figure 2: An example of PRE impacting registers

The impact of PRE and LICM on computation is clear: they insert or delete instructions at some program points. Their impact on registers is more complicated and depends on the code context. Sometimes PRE or LICM may introduce more register spills, while in other cases they may decrease the number of spills. In Figure 2, we show an example where PRE improves the register pressure by decreasing one register spill. In the figure, PRE moves the last use of *a* and *b* up in the code and thus shortens their live ranges but introduces a new live range

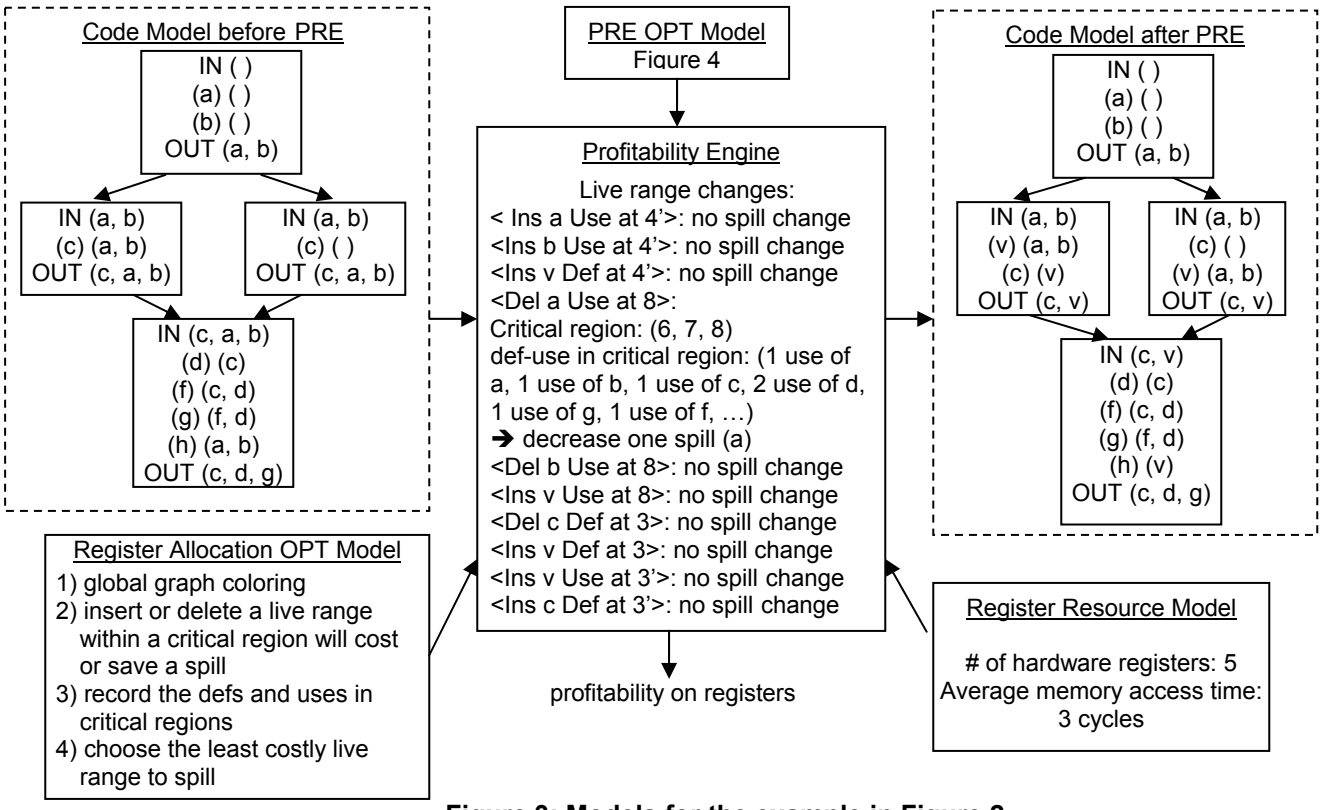


Figure 3: Models for the example in Figure 2

for the temporary variable, *v*. However, if *a* and *b* were used later, their live ranges would remain the same. In this case, the total number of live ranges increases by one due to the temporary variable.

In the next sections, we present models for PRE and LICM. Figure 3 shows our framework to predict the impact of PRE on registers for the example in Figure 2. The models and the profitability engine in this example are explained below.

3.1. Code Models for Registers and Computation

The code model for registers is the same for all scalar optimizations. The code model for registers represents the code as live ranges of global and local variables (including temporaries and parameters). We represent live ranges by the set of live variables at the point immediately before a basic block, IN, and at the point immediately after the block, OUT. The model also includes the code blocks and edges for the code involved in the optimization and statements in terms of uses and definitions. In Figure 3, the boxes labeled "Code Model before/after PRE" show the code model of registers for the example in Figure 2. In a basic block, in addition to IN and OUT information, uses and definitions are also included. For example, "(c) (a, b)" expresses there is a definition of *c* and a use of *a* and *b*.

The code model for computation represents the type and locations of instructions involved in the optimization. We represent each instruction element as $\langle op, \{\langle B1, N1\rangle,...,\langle Bm, Nm\rangle\}\rangle^*$, where *op* is the type of instruction, *Bi* represents the block number and *Ni* expresses the static number of *op* instruction in block *Bi*.

3.2. Register Allocation Optimization Model

To more accurately determine the impact of scalar optimizations on registers, we need a model for register allocation that represents the allocator's spilling strategy. For the impact of registers on profit, we need to compute spills for the original live ranges and the live ranges changed by the optimization and compare them, which is a time consuming process. Indeed, we take an incremental approach by computing the number of spills increased or decreased due to each code change. Our register allocation model reflects this incremental approach.

The register allocator that we model does global graph coloring. Within the region of a code change, we express the number of spills increased or decreased due to inserting or removing a live range for each *critical region*. A critical region is the code segment where the number of live ranges is equal or larger than the number of available hardware registers. So inserting or removing a live range will cost or save a spill in this region. Within

the critical region, we represent definitions and uses of each variable and temporary, and choose to spill the live range that is the least costly to spill, under the assumption that the register allocator typically performs well. That is, we choose the live range which spans the critical region with the smallest number of definitions and uses. This register allocation optimization model is input to the profitability engine (see the next section) which then computes the critical regions for each basic code change, records the definitions and uses of variables and determines the spills. In Figure 3, the box labeled "Register Allocation OPT Model" shows the optimization model for the register allocator described above.

3.3. Scalar Optimization Models

PRE Optimization Models: PRE moves a statement from one code location to another and introduces a temporary to reference the common expression. The PRE optimization models express how these changes impact the code models for registers and functional units.

PRE has three semantic actions: move a computation, say X, (i.e., move X to a safe code location), replace its destination (i.e., replace X's destination with a temporary at the original code position) and replace the redundant expressions (i.e., replace the redundant expression's destination with the temporary and insert a copy instruction following it). A move or a replace can be expressed as a deletion of the instruction at the original site and an insertion at the moved site.

Figure 4 presents the **PRE optimization model for registers**, where the code changes that PRE creates are given. Each change is described by the action insert/delete (Ins/Del), an abstract variable name (also whether it is a use or definition), and what block and where in the block the variable will be inserted/deleted. The semantic actions of PRE are represented as three steps in Figure 4. In step 1, the redundant expression is moved to a safe code location in the statement *Sd* as uses and the temporary V is inserted as the definition. In step 2, at the original position, *Ss*, the expression is deleted and the temporary is inserted as the use. In step 3, for every expression that made the statement redundant, the destination is replaced by the temporary and a copy from the temporary to the destination is placed.

The **optimization model for computation** describes how PRE changes the computation code model. The model essentially shows how instruction elements are changed by the application of PRE. The model is given in Figure 5. The first rule describes that the computation of *op* is deleted at the original block *Bs* and inserted into the destination block *Bd*. The second rule represents the inserted copy instructions at the original block *Bs* and the blocks *Baj* for all redundant expressions.

IF *meet the partial redundant computation exp (X op Y):*
T ← exp is partial redundant at $[Bs, Ss]$;
move it to $[Bd, Sd]$ and assign a new temporary *V*;
T_1 at $[Ba1, Sa1] \ldots T_n$ at $[Ban, San]$ are redundant expressions

THEN

step 1: $\langle \text{Ins exp USE } [\text{Bd, Sd}] \rangle$
$\langle \text{Ins V DEF } [\text{Bd, Sd}] \rangle$

step 2: $\langle \text{Del exp USE } [\text{Bs, Ss}] \rangle$
$\langle \text{Ins V USE } [\text{Bs, Ss}] \rangle$

step 3: $\forall$ Ti at $[\text{Bai, Sai}]$:
$\langle \text{Del Ti DEF } [\text{Bai, Sai}] \rangle$
$\langle \text{Ins V DEF } [\text{Bai, Sai}] \rangle$
$\langle \text{Ins Ti DEF } [\text{Bai} + 1, \text{Sai} + 1] \rangle$
$\langle \text{Ins V USE } [\text{Bai} + 1, \text{Sai} + 1] \rangle$

Figure 4. PRE optimization model for registers

IF *meet the partial redundant computation exp (X op Y):*
T ← exp is partial redundant at $[Bs, Ss]$;
move it to $[Bd, Sd]$ and assign a new temporary *V*;
T_1 at $[Ba1, Sa1] \ldots T_n$ at $[Ban, San]$ are redundant expressions

THEN

$\langle \text{op}, \{\forall\, \text{i}(\text{i} \neq \text{s and i} \neq \text{d})\langle \text{Bi, Ni} \rangle, \langle \text{Bs, Ns} \rangle, \langle \text{Bd, Nd} \rangle\} \rangle$ ➔
$\langle \text{op}, \{\forall\, \text{i}(\text{i} \neq \text{s and i} \neq \text{d})\langle \text{Bi, Ni} \rangle, \langle \text{Bs, Ns} - 1 \rangle, \langle \text{Bd, Nd} + 1 \rangle\} \rangle$

$\langle \text{copy}, \{\forall\, \text{i}\, \langle \text{Bi, Ni} \rangle, \forall\, \text{j} \in [1, \text{n}] \langle \text{Baj, Naj} \rangle, \langle \text{Bs, Ns} \rangle\} \rangle$ ➔
$\langle \text{copy}, \{\forall\, \text{i}\, \langle \text{Bi, Ni} \rangle, \forall\, \text{j} \in [1, \text{n}] \langle \text{Baj, Naj} + 1 \rangle, \langle \text{Bs, Ns} + 1 \rangle\} \rangle$

Figure 5. PRE optimization model for computation

LICM Optimization Models: LICM moves a statement from the body of a loop and places it outside the loop. There are certain conditions that must be met to safely apply LICM. The actions are similar to PRE (and in fact can be thought of as a subcase of PRE) and the resulting optimization models for registers and computation are similar. Based on code movements, the models can predict register impact (with live ranges, as described for PRE) and computation (with code edits and motions, as described for PRE). We do not show these models for brevity because they are similar to PRE.

3.4. Profitability Engine

The profitability engine takes the code models, optimization models and resources models and determines the profit on resources. For example, assume the impact of an optimization on registers is desired. The

engine inputs the code model for registers, a model for this optimization, a register allocation optimization model, and a resource model for registers. It determines the changes in the live ranges according to the optimization model. Then it computes the benefit/cost in terms of spills (i.e., loads and stores) changed by the optimization according to the register allocation optimization model. That is, for each live range change, the engine finds the critical regions and records the number of definitions and uses in the critical regions. When a live range is inserted or deleted within the critical region, the engine chooses the least costly live range to spill and computes the cost or benefit associated with the spill. An example is shown in Figure 3 (see the box labeled "Profitability Engine"). The Profitability Engine determines the changes on the code model and for each change, determines how the spills are affected. For brevity, only detailed actions for "deleting the use of *a* at the statement 8" are shown. The critical region is from line 6 to line 8. Also the uses and definitions are recorded for the critical region. When one live range is deleted, one spill is decreased and *a* is chosen. The benefit of this PRE on registers comes from saving this spill.

If the profits for all the resources, namely registers and computation are combined, they must have the same metric. The computation profit considers the frequency of a node, and therefore, the register profit also need to consider the execution frequency of the loads or stores, based on either profiling or input from the user.

4. Experimental Results

To evaluate the effectiveness of our framework and the usefulness of profit-driven optimization, we implemented models for PRE and LICM, described in Section 3.3, and integrated them into the Mach SUIF compiler [30]. We compared our profit-driven PRE and LICM with always applying an optimization and a heuristic-driven PRE and LICM, which takes register pressure into consideration. We extended the PRE pass implemented by Rolaz [19] in Mach SUIF and implemented LICM [23]. To enable more PRE or LICM opportunities, we also applied passes of copy propagation, constant propagation, and dead code elimination before PRE or LICM. For experiments, we used a number of SPEC2K benchmarks (*gzip, vpr, mcf, parser, vortex,* and *twolf*), which are the SPEC2K benchmarks that can be compiled by the currently available Mach SUIF compiler. We used a dual-processor AMD Athlon 1.4 GHz machine, with 2 GB of memory running RedHat Linux. Using the training data sets, we performed node profiling with the HALT library (included in Mach SUIF) to get the frequency counts used in our computation 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.

Section 4.1 presents the performance improvement of a heuristic-driven PRE and LICM. In Section 4.2, we compare our profit-driven PRE and LICM with always applying and the heuristic-driven PRE and LICM in terms of performance improvement and compile time. Section 4.3 describes the verification of our models in terms of their prediction accuracy.

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 [13]. 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. One problem in a heuristic-driven approach is to the choice of the limit that can generally achieve good performance across all the benchmarks. Our experiments show different benchmarks need different limits to achieve the best performance for both heuristic-driven PRE and LICM.

Tables 2 and 3 show the performance improvement of heuristic-driven PRE and LICM over the baseline. The baseline compiler applies only register allocation and simple instruction scheduling. 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 we choose eight as a limit because there are eight hardware registers that can be allocated for a byte-type variable in the x86. Four and sixteen are used to examine stricter or looser limits. In the tables, the best performance improvement is shown in bold. From the tables, we can see that different benchmarks need different limits in the heuristics 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 very sensitive to the selection of the limits (e.g., *twolf*), while others are not (e.g., *mcf*). We can see that different optimizations may need different limits for the same benchmarks. For example, *gzip* needs the limit set to for PRE but needs the limit set to eight for LICM. So if we fixed the limit in the heuristic-driven approach, the usual

approach, we can not always achieve the best performance improvement.

Table 2. Performance improvement of heuristic-driven PRE with different limits

Benchmark	Heuristic-driven PRE			
	0	4	8	16
gzip	3.50	3.75	3.78	**4.10**
vpr	1.22	0.75	1.81	**1.83**
mcf	**2.37**	2.35	2.31	2.22
parser	1.25	1.50	**1.70**	1.35
vortex	4.73	**5.25**	4.66	3.86
bzip2	7.35	7.52	**8.19**	7.91
twolf	1.07	0.88	**1.14**	0.02

Table 3. Performance improvement of heuristic-driven LICM with different limits

Benchmark	Heuristic-driven LICM			
	0	4	8	16
gzip	2.90	3.29	**5.40**	3.27
vpr	-0.40	-0.38	0.52	**0.69**
mcf	2.50	**2.62**	2.58	2.47
parser	2.55	**2.86**	1.99	2.23
vortex	4.88	**5.69**	4.99	5.28
bzip2	7.02	**7.35**	6.70	4.57
twolf	0.52	0.38	**2.14**	1.91

4.2. Comparing Profit-driven PRE and LICM with Heuristic-driven PRE and LICM

4.2.1. Performance Benefit

Using our model-based framework, we can determine the profitability of an optimization and selectively apply it. The cases where optimizations degrade the performance can be avoided. Figures 6 and 7 show the performance benefit of profit-driven PRE and LICM over the baseline, compared with always applying and the heuristic-driven PRE and LICM. In Figure 6, 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 performance across various register pressures for each benchmark and *Heuristic-8* uses a fixed limit of eight. Lastly, P-PRE represents the performance benefit of our profit-driven PRE. Figure 7 shows the same configurations only applied to LICM.

In Figure 6, the performance benefit of different approaches to decide when to apply PRE is shown. 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 our approach can avoid the unprofitable PREs. However, the selection of limits in the heuristics plays an important role in the performance benefit, as described in Section 4.1. Our 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.

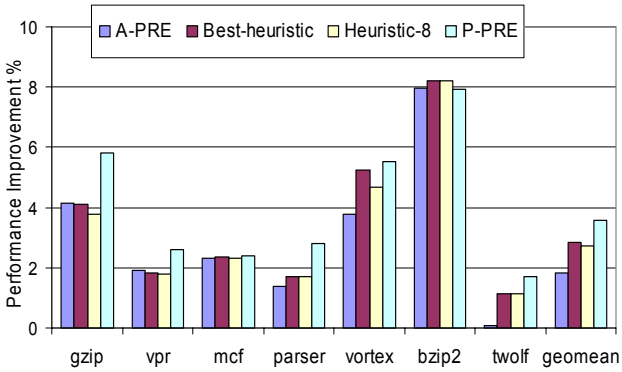


Figure 6. Performance benefit of profit-driven PRE compared with heuristic-driven PRE

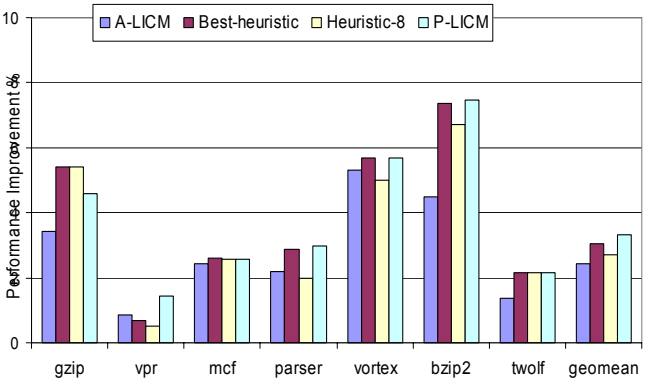


Figure 7. Performance benefit of profit-driven LICM compared with heuristic-driven LICM

Figure 7 shows the performance benefit of the different approaches for applying LICM. Due to the register pressure increase caused by some LICMs, the overall performance of A-LICM can be improved by not applying the unprofitable ones. Although the heuristic-driven LICM can achieve performance improvement over always applying in some cases, it is very important to choose the right limit on allowed register pressure. For example in *parser*, with a register pressure limit of 8, heuristic-driven LICM is worse than always applying. While in the best-heuristic, it is better than always applying. Our profit-driven LICM can perform at least as well as best-heuristic LICM in most cases. However, in one case (*gzip*), due to incorrect predictions, profit-driven

LICM has worse performance than the heuristic-driven approach.

Thus, our experiments show that a model-based approach can be used to explore and determine the profitability of optimizations, and this profitability property can be useful in deciding when to apply optimizations. Also, the profitability measure predicted by our framework has other uses, such as being used as the fitness value in a heuristic search for the best optimization sequence [1, 18].

4.2.2. Compile-time

Because our approach uses analytic models to make decisions about applying optimizations, we investigated how compile-time is impacted by profit-driven optimization. Tables 4 and 5 show the compile-time for different optimization strategies for PRE and LICM.

Table 4. Compile time for PRE (in seconds)

Benchmark	A-PRE	Heuristic PRE	P-PRE
gzip	42	45.14	48.78
vpr	128.38	193	216
mcf	20.89	29	30.5
parser	100.67	123	136.31
vortex	490.48	575.33	633.1
bzip2	33.77	42.6	44.1
twolf	755.55	1087	1187.16

Table 5. Compile time for LICM (in seconds)

Benchmark	A-LICM	Heuristic LICM	P-LICM
gzip	47.8	59.04	61.09
vpr	128	147	161
mcf	20.8	27.6	28.97
parser	109.3	133.47	138.19
vortex	492.1	547.39	569
bzip2	38.59	48.55	51.67
twolf	591	817.76	916.26

From Table 4, the A-PRE compile-time varies from approximately 20 seconds to 755 seconds. Compile-time shown for the heuristic approach is the average for the different limits. It increases 7 % to 50% over A-PRE, with an average of 30% because 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 by an average of 8.3%.

From Table 5, similar compile-time trends can be seen for A-LICM, heuristic LICM and P-LICM. The A-LICM compile-time varies from approximately 20 seconds to 591 seconds. Heuristic LICM increases compile-time over A-LICM from 11% to 38% (average 24%) and P-LICM increases compile time over A-LICM by 15% to 55% (average 31%). Finally, in comparison to heuristic LICM, our P-LICM increases compile time by an average of 6.2%.

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 modest compile-time increase is worth the benefit of applying the optimizations more effectively.

4.3. 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 our computation model 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 the computation profit. However, this inaccuracy is a property of the profile – not of the computation models.

For deciding when to apply optimizations, a correct prediction is one in which we predict there is a benefit for registers (i.e., if register profit is positive, it indicates a spill reduction) and the actual executions show the same result. The accuracy prediction is measured by how often we make the 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).

Table 6. Prediction accuracy of our framework

Benchmark	PRE			LICM		
	TP	CP	%Acc	TP	CP	%Acc
gzip	48	43	89.58	45	38	84.44
vpr	303	291	96.04	230	217	94.35
mcf	51	44	86.27	52	43	82.69
parser	293	210	87.87	75	68	90.67
vortex	530	431	81.13	346	303	87.57
bzip2	56	44	78.57	88	79	89.77
twolf	475	433	91.12	345	306	88.70

Table 6 shows the prediction accuracy of our framework for PRE and LICM. In the table, "TP" is the total number of predictions and "CP" is the number of

correct predictions when using our framework. "%Acc" is the overall percentage accuracy of our framework.

As the table shows, the prediction accuracy varies from 78% to 96%, with an average of 88%. The results demonstrate that our models are indeed accurate and can correctly predict the profit (or cost) and the profit-driven optimizations can achieve performance benefit.

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 accuracy of our models can be improved by incorporating implementation information.

5. Related Work

In the introduction, we indicated prior work on optimization properties. In this section, we discuss prior work that relates to profitability of optimizations. To our knowledge, ours is the first work that focuses on predicting the impact of scalar optimizations and the impact 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 [36]. The work relied on models that had already been developed for modeling the cache and array access sequences [12, 14]. 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.

There have been several approaches to address the problems of the application 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 [31]. 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 [2, 21]. 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 is based on analytic models of code, optimizations and resources [34, 35, 27, 22, 8, 29, 5, 16, 12, 28]. 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 [9].

Research using register pressure sensitive PRE [13] 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 determine the profitability of optimizations. This work coupled with prior work, which considered loop optimizations, has a wide range of applicability in terms of optimizations and resources. Here, we demonstrate the value of our framework for the scalar optimizations PRE and LICM. Our model-based technique can make accurate predictions without applying and executing the optimized code. As such the potential exists for faster searches over different optimization sequences to determine an effective optimization order since we do not have to actually apply the optimizations or run the resulting code. Although our focus was on exploring the profitability property, other properties can be explored using the model-based approach. For example, we believe that models can be used to explore the interaction property. 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).

7. Acknowledgements

This research is supported in part by the National Science Foundation, Next Generation Software, grants CNS-0305198 and CNS-0203945. We would also like to thank John Regehr and the anonymous reviewers for their useful suggestions on how to improve the paper.

8. References

[1] L. Almagor, K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon and T. Waterman. Finding effective compilation sequences. *ACM 2004 Conf. On Languages, Compilers, and Tools for Embedded Systems.*

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM 2000 Conf. on Object-oriented Programming, systems, languages, and applications.*

[3] M. P. Bivens and M. L. Soffa. Incremental register reallocation. *Software Practice & Experience,20(10), 1990.*

[4] P. Briggs and K. D. Cooper. Effective partial redundancy elimination. *SIGPLAN'94 Conf. on Programming Language Design and Implementation.*

[5] B. Chandramouli, J. Carter, W. Hsieh, and S. McKee. A cost framework for evaluating integrated restructuring optimizations. *Int'l. Conf. on Parallel Architectures and Compilation Techniques, September 2001*.

[6] K. Cooper, T.J. Harvey, D. Subramanian, and L. Torczon. Compilation order matters. *Technical Report, Rice University, 2002.*

[7] G. Chaitin. Register allocation and spilling via graph coloring. *ACM SIGPLAN Symp. on Compiler Construction, June 1982.*

[8] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data Layout. *SIGPLAN'95 Conference on Programming Language Design and Implementation, June 1995.*

[9] K. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing, August 2002.*

[10] E. Duesterwald, R. Gupta, M. L. Soffa. Practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems, November 1997.*

[11] L. George and A. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems, May 1996.*

[12] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning behavior. *ACM Transactions on Programming Languages and Systems, July 1999.*

[13] R. Gupta and R. Bodík. Register pressure sensitive redundancy elimination. *8th Int'l. Conf. on Compiler Construction , 1999.*

[14] J. S. Hu, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, H. Saputra, and W. Zhang. Compiler directed cache polymorphis, *Proc. of LCTES/SCOPES, June 2002.*

[15] C. Jaramillo, R. Gupta, and M.L. Soffa. Comparison Checking: An approach to avoid debugging of optimized code. *Proceedings of Foundation of Software Engineering, 1999.*

[16] M. Kandemir, J. Ramanujam, and A. Choudhary. Improving cache locality by a combination of loop and data transformations. *IEEE Transactions on Computers, February 1999.*

[17] T. Kisuki, P. M. W. Knijnenburg and M. F. P. O'Boyle. Combined selection of tile Size and unroll factors using iterative compilation. *Int'l Conference on Parallel Architectures and Compilation Techniques, 2000.*

[18] P.Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. *SIGPLAN'04 Conf. on Programming Language Design and Implementation, 2004.*

[19] L. Rolaz. An implementation of lazy code motion for MachSUIF. URL: *http://lapwww.epfl.ch/dev/machsuif/opt_passes/lcm.pdf*

[20] S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. *SIGPLAN'03 on Programming Language Design and Implementation, 2003.*

[21] U. Holzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems, July 1996.*

[22] K. McKinley, S. Carr, and C. Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems, July 1996.*

[23] S. S. Muchnick. Advanced Compiler Design Implementation. *Morgan Kaufmann Publishers, 1997.*

[24] G. C. Necula. Translation validation for an optimizing compiler. *SIGPLAN 2000 conference on Programming language design and implementation.*

[25] L. Pollock and M.L. Soffa. An Incremental Version of Iterative Data Flow Analysis. *IEEE Transactions on Software Engineering, December 1989.*

[26] M. Poletto and V. Sarkar. Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems, September 1999.*

[27] W. Pugh. Uniform Techniques for Loop Optimization. *5th International Conference on Supercomputing, 1991.*

[28] V. Sarkar and N. Megiddo. An Analytic Model for Loop Tiling and its Solution. *Int'l. Symp. on Performance Analysis of Systems and Software, 2000.*

[29] V. Sarkar, Automatic Selection of high-order transformations in the IBM XL FORTRAN compilers, *IBM Journal of Research and Development, May 1997.*

[30] M. D. Smith and G. Holloway. An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization. URL: *http://www.eecs.harvard.edu/hube/software*

[31] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler Optimization-space Exploration. *Int'l. Symp. on Code Generation and Optimization, 2003.*

[32] D. Whitfield and M. L. Soffa. An Approach to Ordering optimizing transformations. *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, 1990.*

[33] D. Whitfield and M. L. Soffa. An Approach for Exploring Code Improving Transformations. *ACM Transactions on Programming Languages, November 1997.*

[34] M. Wolf and M. Lam. A Data Locality Optimizing Algorithm. *SIGPLAN'91 Conference on Programming Language Design and Implementation.*

[35] M. E.Wolf, D. E. Maydan and D. Chen. Combining Loop Transformations Considering Caches and Scheduling. *Int'l. Symp. on Mircoarchitecture, 1996.*

[36] M. Zhao, B. R. Childers, and M. L. Soffa. Predicting the Impact of Optimizations for Embedded Systems. *ACM Conf. On Languages, Compilers, and Tools for Embedded Systems, 2003.*