Profile Guided Compiler Optimizations

Rajiv Gupta
The University of Arizona, Tucson, Arizona
and
Eduard Mehofer
Institute for Software Science, University of Vienna, Vienna, Austria
and
Youtao Zhang
The University of Arizona, Tucson, Arizona

Traditionally compile-time optimization algorithms are applied conservatively in situations where it is known that the optimization is both definitely applicable and beneficial. Such a conservative approach fails to exploit many valuable optimization opportunities. A profile-guided optimizer uses a program's execution profile in two ways to aggressively optimize the program. First the profiles can be used to identify new optimization opportunities that are frequently observed during program execution but are not detected by static analysis. Second the profiles can be used to carry out sophisticated cost-benefit analysis to apply transformations that improve the performance of one part of the program at the expense of a performance loss in another part of the program. Different types of optimizations require different types of profile data. In this chapter we illustrate the use of control flow, value, and address profiles to selected classical code optimizations and memory optimizations.

1. INTRODUCTION

Over the past several decades numerous compile-time optimizations have been developed to speed up the execution of programs. Application of a typical optimization can be viewed as consisting of two primary tasks: uncovering optimization opportunities through static analysis of the program; and transforming the program to exploit the uncovered opportunities. Most of the early work on classical code optimizations is based upon a very simple performance model. Optimizations are defined such that their application is always considered to have a positive impact on performance. Therefore, the focus of the research has been on developing aggressive analysis techniques for uncovering opportunities for optimization in greater numbers and designing powerful program transformations to exploit most if not all of the uncovered opportunities.

While the simplicity of the approach is attractive, in recent years it has been recognized that the above approach for optimizing programs has serious drawbacks. On one hand this approach fails to exploit many opportunities for optimization and on the other hand it may apply optimizations which may have an adverse impact on performance. The first drawback can be remedied by employing a combination of static analysis and execution profiles to detect optimization opportunities. The second drawback can be addressed by using a sophisticated performance model during the application of optimizations. In particular, a cost-benefit analysis of a program transformation can be carried out before actually applying the transformation. Given adequate profile data, the following broad categories of optimization opportunities can now be exploited:

—*Optimization opportunities uncovered using static analysis whose exploitation involves performance trade-offs.* In some situations the analysis phase of an optimization algorithm may be successful in uncovering an optimization opportunity; however, the transformation phase may require cost-benefit analysis to determine whether or not the optimization opportunity should be exploited. For example, the transformation may represent a trade-off between improved performance in one part of the program and degraded performance in another part of the program.

—*Optimization opportunities that cannot be uncovered by static analysis, but are frequently observed to exist during program execution.* Program optimizations can be designed to exploit characteristics of values, representing data or addresses, encountered by various instructions during program execution. While static analysis can be used to relate values of variables, it is not amenable to identifying specific values involved. For example, by examining the statement $i \leftarrow i + 1$, we can statically determine that the statement increments the value of i by one. However, in general, it is not likely that we will be able to determine the exact value of i as the value of i may depend upon a number of runtime factors such as the program input, the number of times the statement is executed etc.

To illustrate the above scenarios, consider the example shown in Fig. 1. Let us consider the first scenario. The original code contains an optimization opportunity. If both conditionals evaluate to true, the expression $x \times y$ is computed twice. Moreover we can make this determination by statically analyzing the program. To optimize the program we may wish to transform the code as shown in Fig. 1b. This transformation removes the redundant evaluation of $x \times y$ for true evaluations of the conditionals. However, if we consider the execution corresponding to false evaluations of the conditionals, we find that an evaluation of $x \times y$ is introduced in the transformed code where none was present prior to transformation. Therefore we can conclude that the transformation is useful only if line 4 is executed less frequently than line 7. A simple cost-benefit analysis based upon expected execution frequencies of various statements in the program can be used to decide whether or not the transformation should be applied.

Now let us consider an instance of the second scenario. Let us assume that static analysis cannot identify any specific values associated with variable y during program execution. However, by profiling the execution of the program we may determine that very frequently the value of y at line 4 is 1. Therefore using profiling we have identified an optimization opportunity that was not detected using static analysis, namely that the multiply operation at line 4 can be frequently optimized away. We can transform the program as shown in Fig. 1c. The multiply operation at line 4 is executed conditionally in the transformed code. Since extra instructions are required to check whether the value of y is 1 and accordingly update t, it is important that we ensure that the overall benefit of eliminating the multiply operation is greater than the overall cost of executing the extra instructions. A simple cost-benefit analysis based upon the expected frequencies with which variable y has

```
1.  if () then
2.      z ← x × y
3.  else
4.      ....
5.  endif
6.  if () then
7.      w ← x × y
8.  else
9.      ....
10. endif
```

(*a*) *Original Code.*

```
1.  if () then
2.      t ← z ← x × y
3.  else
4.      t ← x × y
5.  endif
6.  if () then
7.      w ← t
8.  else
9.      ....
10. endif
```

(*b*) *Redundancy Removal.*

```
1.  if () then
2.      t ← z ← x × y
3.  else
4.      t ← (y = 1)? x : x × y
5.  endif
6.  if () then
7.      w ← t
8.  else
9.      ....
10. endif
```

(*c*) *Strength Reduction.*

Fig. 1. Examples of Profile-Guided Transformations.

the value 1 or some other value at line 4 can be used to determine whether or not the transformation should be applied.

While the example in Fig. 1 illustrated that execution profiles can be used to both detect optimization opportunities and identify beneficial transformations in context of classical optimizations, similar situations also arise in the context of memory optimizations discussed later in this chapter.

Another reason for increased relevance of profile-guided optimizations is that many opportunities for optimizations which could not be exploited on processors of the past, can now be exploited due to the advanced hardware features present in modern processors. Support for safe speculative execution and predicated execution can be helpful in carrying out profile-guided optimizations. For example, if the multiple operation in Fig. 1 is replaced by a divide operation, then the placement of x/y at line 4 can result in a divide by zero exception which may not have occurred during the original program execution. Therefore the redundancy removal transformation is rendered illegal. However, the IA-64 processor provides hardware support for suppressing spurious exceptions and therefore allows the optimization of divide operation to be carried out [18]. Support for predicated execution, also provided by IA-64, can enable efficient implementation of the strength reduction transformation shown in the above example as it enables conditional execution of the multiply operation without introducing additional branch instructions.

From the above discussion it is clear that the cost-benefit analysis should be an integral part of an optimization algorithm. Moreover cost-benefit analysis for a given program execution can only be carried out if we are provided with estimates of execution frequencies of various runtime events that are both relevant to the program's performance and are impacted by the program transformation. While in the above example both the cost and the benefit was being measured in terms of number of instructions (or cycles), this may not always be the case. In some situations while the benefit may be measured in terms of anticipated reduction in the number of instructions, the cost may be in measured in terms of code growth resulting from the transformation. This is because many of the optimizations essentially perform code specialization which results in code growth.

The profile-guided compilation process is summarized in Fig. 2. Before profile-guided optimizations can be carried out, an instrumented version of the program must be run on one or more representative inputs to collect profile data. This profile data is then used by the optimizing compiler to recompile the program generating optimized code. The selection of representative inputs is important since the optimizations are expected to be beneficial only if the profile data that is collected is relatively insensitive acroos a wide range of inputs on which the program is expected to be executed. While this chapter is devoted to static profile-guided optimization of programs, the concept of profile-guided optimization is also used by dynamic optimizers. However, in contrast to the techniques described in this chapter, the profiling and optimization techniques employed during dynamic optimization must be extremely light weight.

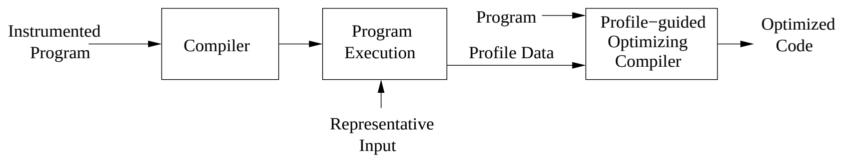


Fig. 2. Profile-Guided Optimizing Compiler.

The remainder of this chapter is organized as follows. In section 2 we provide a brief overview of types of profile data that is used to guide a wide range of optimizations. In section 3 we discuss the role of profiles in carrying out classical optimizations. We illustrates the principles of profile-guided optimization through an in depth look at a partial redundancy elimination algorithm. In section 4 we discuss the role of profiles in carrying out optimizations that improve the memory performance of a program. Concluding remarks are given in section 5.

2. TYPES OF PROFILE INFORMATION

Profiles provide summary information on past program executions that are used to guide program optimization. However, different types of optimizations require different types of profiles. In particular, three types of profile information is used in practice: control flow profiles, value profiles, and address profiles. In this section we briefly discuss each of the profile types including the types of optimizations where they are used.

2.1 Control Flow Profiles

One form of profile captures a trace of the execution path taken by the program. This trace represents the order in which the nodes corresponding to basic blocks in the program's control flow graph (CFG) are visited. We refer to such a profile as the program's *control flow trace* (CFT). By examining a CFT we can compute the execution frequency of any given program subpath. As expected, CFTs can be extremely large in size and therefore representations that maintain CFTs in compressed form have been considered. Such compressed forms are referred to as *Whole Program Paths* (WPPs). In [27] Larus used the Sequitur [31] algorithm for compression while in [43] Zhang and Gupta proposed alternative redundancy removal techniques for compression. However, even after compression, WPPs can be extremely large.

In practice, a number of approximations of CFT that directly measure the execution frequencies of selected program subpaths are used. These profiles differ in the degree of approximation involved and the cost for collecting them. The proposed approximations of control flow profiles include the following:

—*Node profiles* provide the execution frequencies of the basic blocks in the control flow graph. For some optimizations such profiles are adequate. For example, in making code placement decisions, as illustrated by the redundancy removal transformation of Fig. 1b, node profiles are sufficient.

—*Edge profiles* provide the execution frequencies of each edge in the control flow graph. The overhead of collecting edge profiles is comparable to the overhead of collecting node profiles. However, edge profiles are superior to node profiles because edge profiles cannot always be computed from node profiles while node profiles can always be computed from edge profiles. Edge profiles are widely used.

—*Two-edge profiles* [30] provide the execution frequencies of each pair of consecutive edges in the control flow graph. They are clearly superior to edge profiles. Edge profiles can always be computed from two-edge profiles. However, the reverse is not true. Two-edge profiles derive their increased power from their ability to capture the correlation between the executions of consecutive conditional branches.

—*Path profiles* [2] provide the execution frequencies of acyclic subpaths in the control flow graph such that they are acyclic and intraprocedural. Since a path is acyclic, it does not ever include a loop back edge and since it is intraprocedural, it terminates if an exit node of a procedure is reached. Path profiles are more precise than two-edge profiles for acyclic components of a control flow graph since they capture correlation across multiple conditional branches within an acyclic graph. However, two-edge profiles can capture correlation among a pair of conditional branches along a cyclic path while path profiles cannot do so.

Relative precision of control flow profiles. Each of the above profiles can be used to derive an estimate of the execution frequency of an arbitrary subpath in the program. However, the precision of the estimate can vary from one type of profile data to another. If we examine the subpaths whose execution frequencies are directly measured by each of the above profiling algorithms, they can be distinguished based upon two important characteristics. First the *lengths* of the subpaths whose execution frequencies are collected vary. Second the degree of *overlap* between two subpaths that may be executed one after another also varies. The table in Fig. 3 gives the length and overlap, in terms of number of nodes, for each of the profile types. From this table we can conclude the relationships of the precisions for any pair of profile types. For example, the two-edge and path profiles are not comparable as the former has greater degree of overlap while the latter allows for longer paths. The precision relationships are summarized by the hierarchy shown in Fig. 3.

Let us illustrate the estimation of the execution frequency of a given program path from various types of profiles. When such an estimate is not exact, we will obtain lower and upper bounds on the execution frequency. If the lower and upper bounds for a path are found to be l and h, then it means that the path was definitely

	Node	Edge	Two-edge	Path
Length (nodes)	1	2	3	≥ 1
Overlap (nodes)	0	0	2	0

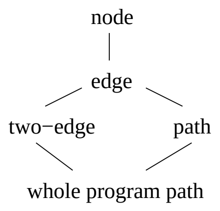
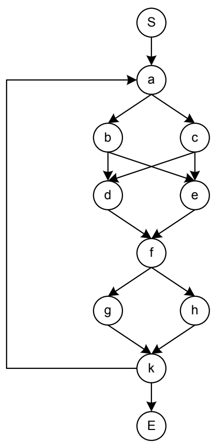


Fig. 3. Relative Precision of Control Flow Profiles.

executed l times and potentially executed as many as h times. Fig. 4 shows a sample control flow graph and the complete control flow trace for a program execution. The node, edge, two-edge, and path profiles corresponding to this program execution are also given. Let us estimate the frequency of path `abefgk` from various profiles.



Control Flow Trace:
S(abdfgk)^{10}(acefhk)^{10}
(abdfhk)^{10}(acefgk)^{10}
(abefgk)^{50}(abefhk)^{10}E

Node	Frequency
S	1
a	100
b	80
c	20
d	20
e	80
f	100
g	70
h	30
k	100
E	1

Edge	Frequency
Sa	1
ab	80
ac, ce	20
bd, df	20
be	60
ef	80
fg, gk	70
fh, hk	30
ka	99
kE	1

Two-Edge	Frequency
Sab	1
abd, bdf	20
abe, bef	60
ace, cef	20
dfg	10
dfh	10
efg	60
efh	20
fgk	70
fhk	30
gka	80
hka	20
kac	20
kab	79
hkE	1

Path	Frequency
Sabdfgk	1
abdfgk	9
abdfhk	10
abefgk	50
acefhk	10
acefgk	10
abefhk	9
abefhkE	1

Profile $\rightarrow$	Node	Edge	Two-edge	Path
Freq. Estimate for Path `abefgk`	30-70	30-60	40-60	50

Fig. 4. An Example of Control Flow Profiles.

—*Estimate based upon node profiles.* If we examine the execution frequencies of the nodes on path `abefgk`, the minimum frequency encountered is 70 for node `g`. Therefore the upper bound on the execution frequency of the path is 70. To find the lower bound on the execution frequency of path `abefgk`, first consider the lower bound on the execution frequency of subpath `efgk`. Since node `d` is executed 20 times, the lower bound on the execution frequency of `efgk` is 50. Now further considering that the execution frequency of node `c` is 20, the lower bound on the execution frequency of path `abefgk` is 30. Therefore we conclude that path `abefgk` is executed at least 30 times and potentially as many as 70 times.

—*Estimate based upon edge profiles.* By examining the edge profiles, we can conclude that the subpaths `abef`, `fgk` and `fhk` are executed exactly 60, 70 and 30 times respectively. Therefore we can conclude that the path `abefgk` is executed at least 30 times and potentially 60 times. An algorithm for deriving these estimates can be found in [3].

—*Estimate based upon two-edge profiles.* As in the case of edge profiles, we can deduce that the subpath `abef` has a frequency of 60. The frequency of subpaths `efg` and `efh` are measured directly to be 60 and 20 respectively. Therefore we can conclude that the path `abefgk` is executed at least 40 times and possibly as many as 60 times.

—*Estimate based upon path profiles.* During path profiling the execution frequency of path `abefgk` is directly measured since it is an acyclic path. Therefore we know that its execution frequency is exactly 50.

Cost of collecting profiles. To collect the profiles we must execute instrumented versions of the program. The instrumentation code that is introduced depends upon the type of profiles being collected. While in general the overhead of instrumented code is linear to the length of the program execution, techniques can be employed to reduce the overhead.

During the collection of node profiles, instead of instrumenting each basic block to collect its execution frequency, we can introduce a counter for each control dependence region in the program. The execution frequency of a basic block is equal to the sum of the execution frequencies of the control dependence regions to which it belongs [32]. Since there are far fewer control dependence regions than there are basic blocks, this approach reduces the overhead of profiling. In Fig. 4 the basic blocks `a`, `f`, and `k` belong to the same control dependence region and they all have identical execution frequencies. Therefore a single shared counter can be used to measure their execution frequencies.

Similarly each edge in the control flow graph need not be instrumented as execution frequencies of some edges can be computed from execution frequencies of other edges. In Fig. 4 the edges `fg` and `gk` always have the same frequency, so that at most one of them should be instrumented. Moreover if the frequencies of edges `df`, `ef`, and `fh` are known, and there are no exceptions, we can deduce the frequency of edge `fg`.

The collection of two-edge and path profiles is more expensive. However, their computation can also be optimized by reducing instrumentation points. In [2] an algorithm is presented to reduce the overhead of instrumentation code during collection of path profiles. Each path in the acyclic graph is assigned a unique number. The form of the instrumentation is such that it computes the path number as it is traversed. Upon reaching the end of the path, this path number is available and used to update the frequency counter associated with the path. By carefully placing the instrumentation code, the number of instructions needed to compute the path number can be minimized.

2.2 Value Profiles

Value profiles identify the specific values encountered as an operand of an instruction as well as the frequencies with which the values are encountered. The example in Fig. 5 illustrates the form of these profiles. With this information the compiler can recognize operands that are almost always constant and utilize this information to carry out value specialization optimizations such as constant folding, strength reduction, and motion of nearly invariant code out of loops.

Code:

```
I1: load R3, 0(R4)
I2: R2 ← R3 & 0xff
```

Value profile:

(instruction, register)	profiles (value,freq)
(I1,R3)	(0xb8d003400,10) ...
...	...
(I1,R2)	(0,1000)
(I2,R3)	(0,100),(0x8900,200) …,(0x2900,100)

Fig. 5. Sample Value Profiles.

Since the number of instructions in a program is large, and each operand of an instruction may potentially hold a very large number of values, collection of complete value profiles is not practical. Therefore to reduce the size of the profile data and the execution time overhead of profiling the following two steps are taken.

First only the most frequently appearing `N` values are collected for a given operand. Calder *et al.* [9] have proposed maintaining a top-n-value table (TNV) for a register being written by an instruction. Each TNV table entry contains a pair of values: the *value* and the *frequency* with which that value is encountered. Least frequently used (LFU) replacement policy is used to choose an entry for replacement when the table is full. If we exclusively use the LFU policy for updating the TNV, the values that are encountered later in the program may not be able to reside in the table even if they are encountered frequently. This is because they may be repeatedly replaced. To avoid this situation, at regular intervals the bottom half of the table is cleared. By clearing part of the table free entries are created that can be used by values encountered later in the program. Both the number of entries in the table and clearing interval are carefully tuned to get good results. Collecting only top `N` values not only reduces the profiling overhead, but also makes the convergence to a steady state easier and faster to reach.

Second complimentary approach to reducing profiling overhead is to collect value profiles for only interesting instructions. Watterson and Debray [35] use a cost-benefit model where the cost is the testing cost of whether a register has a special value, and the benefit is the direct and indirect instruction savings that can be achieved by optimizing the program with this information. Control flow profiles are collected first to carry out cost-benefit analysis and identify candidates for value profiling.

2.3 Address Profiles

Address profiles can be collected in form of a stream of memory addresses that are referenced by a program. These profiles are usually used to apply data layout and placement transformations for improving the performance of memory hierarchy. Depending upon the optimization, the address traces can be collected at different levels of granularity. At the finest level of granularity, each memory address can be traced. Coarser level traces record references to individual objects rather than individual addresses.

A program's address space can be divided into three parts: stack, heap, and globals. Stack data typically exhibits good cache locality. Therefore most of the research is focused on improving the data access behavior of globals and heap data. Programmers usually organize their data structures logically. They tend to put logically related data objects or data fields together. However, the logical relationships may be different from the order in which the data structures are actually accessed at runtime. Using the information provided by address profiles, compilers can reorganize the placement of data objects with respect to each other or the placement of fields within a data object such that closely accessed items are placed next to each other. In this way the cache behavior is improved.

A complete address trace of a program run can be extremely large. In order to compress the size of the address trace, Chilimbi [13] has proposed using the Sequitur algorithm to generate a compressed *whole program stream* (WPS) representation of the address trace in much the same way as Sequitur is used to compress a program's control flow trace. To guide the application of data layout and placement transformations, the WPS representation is analyzed to identify *hot address streams*. These streams represent subsequences of addresses that are encountered very frequently during the program run.

While the above approach first collects complete address profiles and then processes them to identify information useful in guiding data layout and placement transformations, another approach is to directly identify the useful information. Calder *et al.* [10] have proposed an algorithm based upon such an approach. The information that they collect is represented by a graph named the *temporal relationship graph* (TRG). The nodes in this graph are data items of interest. Weighted links are established between pairs of nodes.

If references to a pair of data items are separated by fewer than a threshold number (say N) of other data references, then the weight associated with the link between the two items is incremented. To maintain the weights of all the links, an N-entry queue is maintained which records the latest N data items that are referenced by the program. The weights on the links at the end of the program run can be used by the compiler to identify data items that should be placed close to each other for achieving good cache behavior. Fig. 6 shows an example of the information collected using this approach.

Sample code:

```
for(i=0;i<2000;i++) {
   swtich (flag) {
   case 1:
      xa = *pa;  … ; break;
   case 2:
      xb = *pb;  … ; break;
   case 3:
      xc = *pc;  … ; break;
   case 4:
      xd = *pd;  … ; break;
   }
   ….
   pa = buf[i]
   ….
}
```

Declarations:

```
int flag;
int *pa,*pb,*pc,*pd;
int buf[2000];
…
int xa,xb,xc,xd;
```

Address profile:

Link	Weight
(A(xa),A(pa))	500
(A(xb),A(pb))	20
(A(xc),A(pc))	2
(A(xd),A(pd))	10
….	….
(A(pa),A(buf))	2000

Fig. 6. Address Profiles.

3. PROFILE GUIDED CLASSICAL OPTIMIZATIONS

Simple optimization algorithms typically optimize statements that are determined to be optimizable under all conditions through static analysis of the program. On the other hand more aggressive algorithms also optimize statements that are *conditionally optimizable* where the optimization opportunities are discovered either through static analysis or through profiling. As a consequence, often such algorithms involve replicating statements and creating unoptimized and optimized copies of them. Depending upon the conditions that hold, appropriate copy of the statement is executed. The above process is also commonly referred to as *code specialization*. In some optimizations specialization leads to elimination of a copy (e.g., redundancy and dead code elimination) while during other optimizations specialization leads to simpler and more efficient code (e.g., strength reduction and constant folding).

In this section we begin by providing a brief overview of various code specialization transformations followed by identification of specific optimizations that rely on them. We describe the critical role that profiling plays in carrying out these optimizations. Later we present a profile-guided partial redundancy algorithm to illustrate how profile data can be exploited in a systematic way in developing a profile-guided optimization algorithm.

3.1 Transformations

Different code specialization algorithms carry out *code replication* at different levels of *granularity*. For example, function inlining replicates entire functions, partial inlining replicates code along selected paths through a function, and code motion based algorithms replicate individual statements. There are primarily two classes of transformations that are used to carry out code replication and enable specialization of conditionally optimizable code: *code motion* of different types and *control flow restructuring* with varying scope.

Code motion of statements. The basic form of code motion, namely *safe code motion*, in addition to honoring the program's data dependences, guarantees that for every execution of a statement during the execution of optimized code, there exists a corresponding execution of the statement during the execution of unoptimized code. As a consequence, it must be the case that if an exception occurs during the execution of optimized code, it would have also occurred during the execution of unoptimized code. Hardware support present in modern processors such as IA-64 allows relaxation of the above constraint and yet preserves the program semantics [18]. In particular, *speculative code motion* allows compiler to introduce executions of a statement in the optimized code that are not present in the unoptimized code. To be beneficial, the added cost of these extra executions must be offset by savings resulting from speculative code motion. Therefore profile based cost-benefit analysis is typically required to take advantage of speculative code motion. *Predicated code motion* further creates opportunities for performing code motion more freely [18]. Statements can be moved out of control structures and executed at a different program point under the original conditions by predicating its execution with an appropriately constructed predicate expression. Profiling can assist in estimating whether the benefit achieved by code motion outweighs the cost of evaluating the predicate expression.

Control flow restructuring. If the conditions under which a statement can be optimized hold along some execution paths but not others, then control flow restructuring can be employed to enable the optimization of the statement as follows. Using restructuring we can create a program such that when we reach the statement, based upon the incoming edge along which we arrive at the statement, we can determine whether or not the statement can be optimized. We can create unoptimized and optimized copies of the statement and place them along the incoming edges. The scope of control flow restructuring determines the degree to which the code can be optimized. For example, the changes due to intraprocedural control flow restructuring are localized to within a procedure body while interprocedural restructuring changes the flow of control across procedure boundaries. The primary cost of using this transformation is the resulting growth in code size. Increasing the scope of restructuring also increases the amount of resulting growth in code size. Function inlining is one way to achieve interprocedural control flow restructuring. However, it is also accompanied with significant code growth. To limit code growth while performing interprocedural optimizations a couple of alternative techniques have been proposed: partial inlining of frequently executed paths through a procedure [23] and creating procedures with multiple entries and multiple exits [4].

3.2 Optimizations

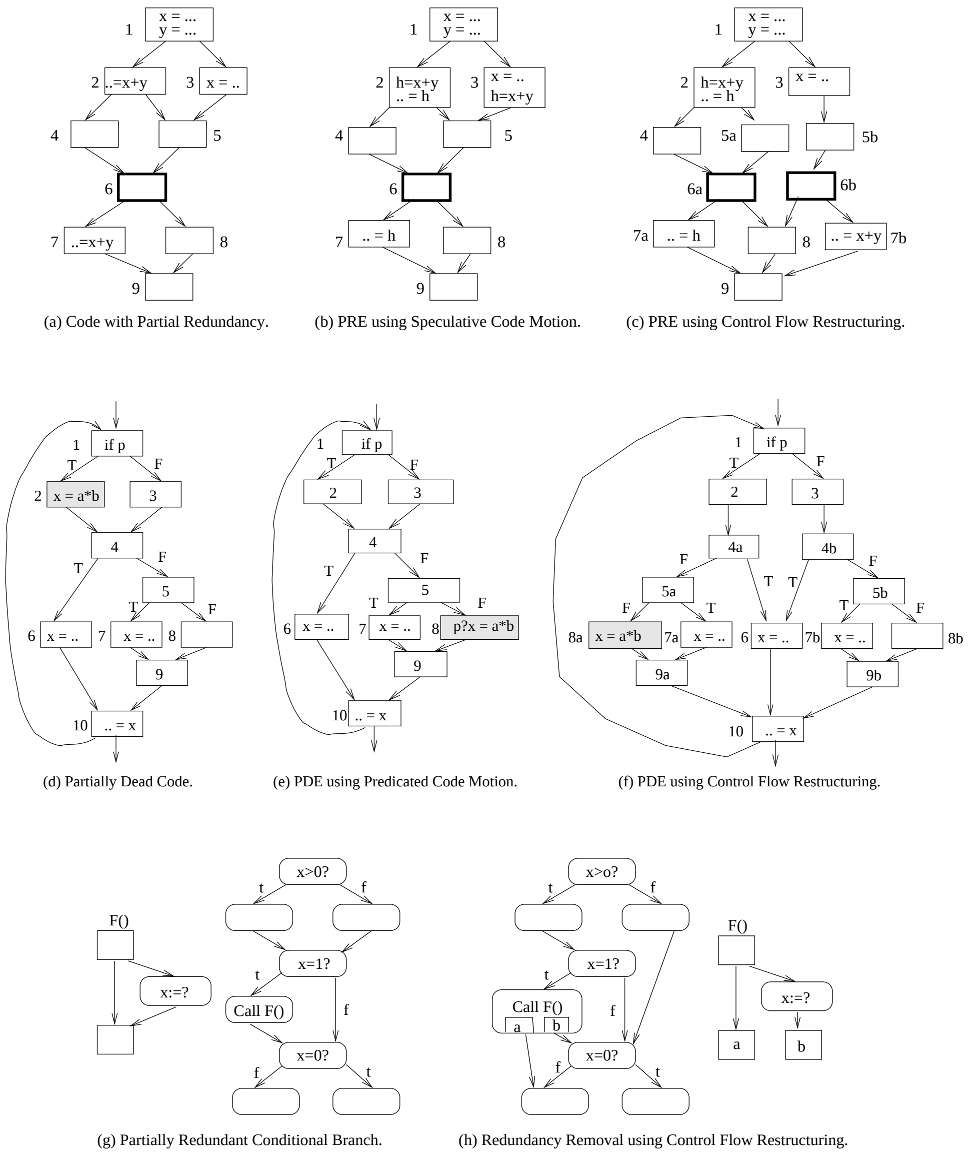
Partial Redundancy Elimination (PRE). The PRE of expressions is traditionally performed using *safe code motion* [25; 36; 15]. However, many opportunities for redundancy removal cannot be exploited using this restricted form of code motion. For example, in Fig. 7a the evaluation of expression `x+y` in node 7 is partially redundant because if node 2 is visited prior to reaching node 7, then the expression has already been evaluated at node 2 and need not be evaluated again. However, the partial redundancy of `x+y` cannot be either reduced or eliminated using safe code motion. If *speculative code motion* is used to hoist `x+y` above node 6, as shown in Fig. 7b, a net reduction in partial redundancy of `x+y` results if the frequency of node 3 is less than the frequency of node 7. If *control flow restructuring* is employed in the manner shown in Fig. 7c, then the redundancy is entirely eliminated. However, some code growth also results. Cost-benefit functions based upon profile data can be designed to selectively carry out the above transformations.

The use of speculation was first proposed in [20; 21] and that of control flow restructuring was first proposed in [40]. Using the combination of all of the above transformations to achieve greater benefits at lower costs is discussed in [6]. While we have only considered redundancy evaluation in context of arithmetic expressions, partial redundancy elimination can also be applied in other contexts such as *load removal* [7] and *array bounds checks elimination* [8; 19].

Partial dead code elimination (PDE). The PDE of an assignment is achieved by delaying the execution of the statement to a point where its execution is definitely required [26]. Using conservative code motion all opportunities for PDE cannot be exploited. For example, the assignment in node 2 of Fig. 7d is partially dead because if control reaches nodes 6 or 7, the value computed by the assignment is never used. However, we cannot simply delay the execution of the assignment to node 8 since in some cases placement of the assignment at node 8 will block the correct value of `x` from reaching its use in node 10. Straightforward code motion will block the sinking of the assignment to node 4, and therefore any further; thus preventing any optimization to occur. By *predicating* the statement we can enable its sinking past node 4 and down to 8 and thus achieve PDE (see Fig. 7e). This approach is clearly beneficial if the frequency of node 8 is less than that of node 2. Control flow restructuring can also be applied to carry out PDE as shown in Fig. 7f. As in the case of PRE, profile based cost-benefit analysis is required to selectively apply the above forms of PDE. Predication based PDE was first proposed in [22]. A control flow restructuring algorithm that minimizes code growth associated with PDE can be found in [5]. Just as partially dead assignments can be eliminated using the motion and restructuring transformations, *partially dead stores* can also be eliminated.

Conditional branch elimination. A conditional branch is considered to be partially redundant if along some paths its outcome can be determined at compile-time. Elimination of a partially redundant conditional branch requires *flow graph restructuring* to be performed. During flow graph restructuring, the path(s) along which the outcome of a conditional branch is known at compile-time are separated from the path(s) along which its outcome is unknown through code duplication. The basic idea of intraprocedural restructuring was first proposed in [37] and then a more general algorithm based upon interprocedural demand driven analysis as well as profile-guided interprocedural control flow restructuring was given in [4].

The last example in Fig. 7 illustrates this optimization. The flow graph in Fig. 7g contains two paths, an intraprocedural one and an interprocedural one, along which conditional branch elimination is applicable. The first path is established when $x > 0$ is false and hence $x = 1$ is known to be false. The second path arises because if $x = 1$ is true, and value of x is preserved during the call to function $F()$, the condition $x = 0$ will evaluate to false. The first opportunity is exploited through intraprocedural control flow restructuring and the second requires interprocedural restructuring that is achieved by splitting the exit of function $F()$ as shown in Fig. 7h.



(a) Code with Partial Redundancy. (b) PRE using Speculative Code Motion. (c) PRE using Control Flow Restructuring.

(d) Partially Dead Code. (e) PDE using Predicated Code Motion. (f) PDE using Control Flow Restructuring.

(g) Partially Redundant Conditional Branch. (h) Redundancy Removal using Control Flow Restructuring.

Fig. 7. Code Motion and Control Flow Restructuring for Profile Guided Classical Optimizations.

3.3 Partial Redundancy Elimination via Speculative Code Motion

A computation is called partially redundant if there exists at least one path on which it is computed twice. Such redundancies can be eliminated by doing the computation once at an appropriate program point, assigning the value to an auxiliary variable, and replacing the original computations by the auxiliary variable. The problem of finding an appropriate program point is guided by the following transformational idea. Unnecessary recomputations can be avoided by moving computations in the opposite direction of the control flow and placing them in a more general context maximizing in this way the potential of redundant code which can be eliminated thereafter.

Under the restriction that no new computations are inserted along any path, computationally optimal results can be obtained statically [25]. However it is assumed that all paths are equally important which is not true in practice. In this section we present a PRE algorithm which takes profile information into account to reduce the number of recomputations further. By executing expressions speculatively, it enables the removal of redundancies along more frequently executed paths at the expense of introducing additional expression evaluations along less frequently executed paths. More specifically, speculation is the process of hoisting expressions such that an expression whose execution is controlled by a conditional prior to speculation, is executed irrespective of the outcome of that conditional after speculation. In this way new computations may be introduced on some paths which can alter the semantics of a program in case these computations can cause exceptions. However, modern architectures like Intel IA-64 support speculative execution of instructions by suppressing exceptions making this kind of transformation safe.

The remainder of this section is organized as follows. We first present a *cost model* which will be used to identify profitable opportunities for speculation. Next we present a practical algorithm for carrying out *cost-benefit* analysis based upon probabilistic data flow analysis. We show how under this analysis the application of the cost model is approximated. Finally we present the details of the *code motion framework* used to carry out PRE using a combination of safe code motion and selectively enabled speculative code motion based upon cost-benefit analysis.

Cost model. For a given expression *exp* and a conditional node n we formulate a condition under which it is potentially profitable to enable speculative hoisting of *exp* above n. Essentially we identify the paths through n which can *benefit* from speculation of *exp* and paths that incur an additional execution time *cost* due to speculation of *exp*. If the probability of following paths that benefit is greater than the paths that incur a cost, then speculation of *exp* is enabled at n. The computation of these probabilities is based upon an execution profile.

To develop the above cost model for hoisting an expression *exp* above a conditional node n, it is useful to categorize the program subpaths that either originate or terminate at n as follows:

(1) *Available subpaths* which are subpaths from the *start* node to n along which an evaluation of *exp* is encountered and is not killed by a redefinition of any of its operands prior to reaching n.

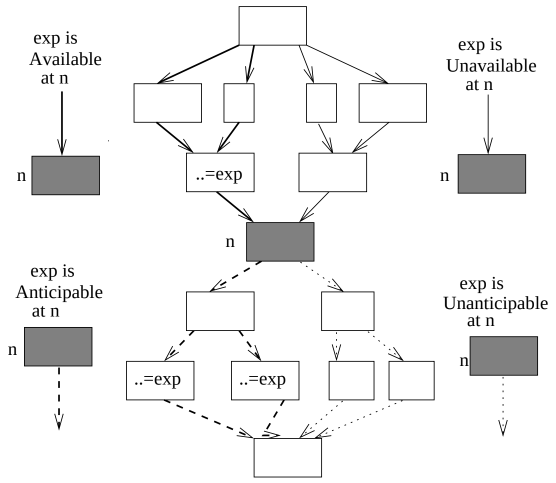


Fig. 8. Path Classification.

(2) *Unavailable subpaths* which are subpaths from the *start* node to n along which *exp* is not available at n. These subpaths include those along which either an evaluation of *exp* is not encountered or if *exp* is encountered, it is killed by a redefinition of one of its operands prior to reaching n.

(3) *Anticipable subpaths* which are subpaths from n to the *end* node along which *exp* is evaluated prior to any redefinition of the operands of *exp*.

(4) *Unanticipable subpaths* which are subpaths from n to the *end* node along which *exp* is not anticipable at n. These subpaths include those along which either *exp* is not evaluated or if *exp* is evaluated, it is done after a redefinition of one of its operands.

The paths that can potentially *benefit* from hoisting of *exp* above a conditional node n are the paths that are obtained by concatenating *available* subpaths with *anticipable* subpaths at n. This is because first of all along these paths there is redundancy. An evaluation of *exp* prior to reaching n causes an evaluation of *exp* performed after visiting n to become redundant. Secondly, the redundancy cannot be removed unless *exp* is hoisted above node n. Therefore we can now state that, given a path p that passes through the conditional node n, the hoisting of an expression *exp* above n can **benefit** path p only if *exp* is *available* and *anticipable* at n's entry along p. We denote the set of paths through n that benefit from hoisting of *exp* above n as $BenefitPaths_{exp}(n)$. The expected overall benefit of speculating *exp* can be computed by summing up the execution frequencies of paths in this set as shown below.

$$Benefit_{exp}(n) = \sum_{p \in BenefitPaths_{exp}(n)} Freq(p).$$

The paths that incur a *cost* due to hoisting of *exp* above a conditional node n are the paths that are obtained by concatenating *unavailable* subpaths with *unantici-*

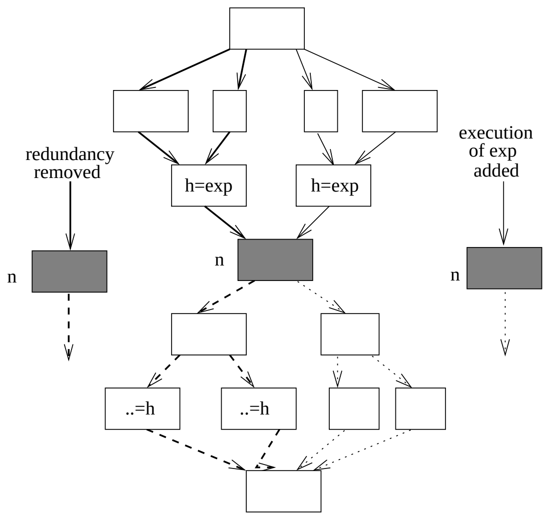


Fig. 9. Paths that Benefit vs Paths that Incur a Cost.

pable subpaths at n. This is because along these paths an additional evaluation of *exp* prior to reaching n is introduced. Therefore, given a path p that passes through a conditional node n, the hoisting of an expression *exp* above n **costs** path p only if *exp* is *unavailable* and *unanticipable* at n's entry along p. We denote the set of paths through n that incur a cost due to hoisting of *exp* above n as $CostPaths_{exp}(n)$. The expected overall cost of speculating *exp* above n can be computed by summing up the execution frequencies of paths in this set as shown below.

$$Cost_{exp}(n) = \sum_{p \in CostPaths_{exp}(n)} Freq(p).$$

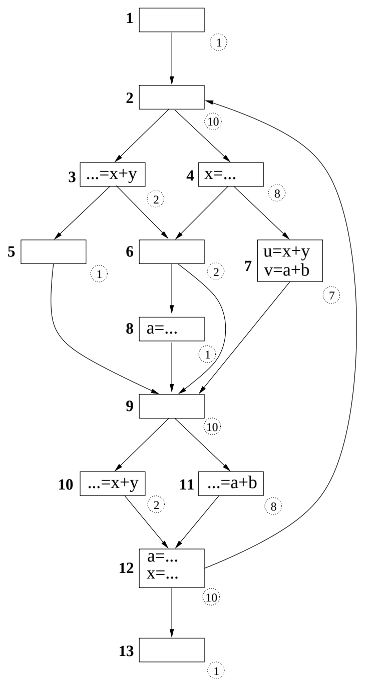
Speculation should be enabled if based upon the profile data we conclude that the expected benefit exceeds the expected cost. A boolean variable $EnableSpec_{exp}(n)$ is associated with an expression and conditional node pair (exp, n) which is set to true if speculation for *exp* at n is enabled; otherwise it is set to false. For convenience we restate the speculation enabling condition in terms of probabilities as the detailed analysis algorithm that we present later is based upon probabilistic data flow analysis. As shown below, speculation is enabled if the probability of following a path through n that benefits is greater than the probability of following a path that incurs an additional cost. $Freq(n)$ is the execution frequency of node n. Note that the sum of the two probabilities used below need not be 1 because there may be other paths which are unaffected by speculation.

$$EnableSpec_{exp}(n) = ProbCost_{exp}(n) < ProbBenefit_{exp}(n), where$$

$$ProbCost_{exp}(n) = \frac{Cost_{exp}(n)}{Freq(n)} \quad \& \quad ProbBenefit_{exp}(n) = \frac{Benefit_{exp}(n)}{Freq(n)}.$$

It should be noted that if an expression exp is never available or never anticipable at n, then $ProbBenefit_{exp}(n)$ is 0 and therefore speculation is never enabled at n. On the other hand if exp is always available or always anticipable at n, then $ProbCost_{exp}(n)$ is 0 and, if the $ProbBenefit_{exp}(n)$ is non-zero, speculation is enabled at n.

Let us apply the above cost model to the example in Fig. 10. We assume that we are given the entire control flow trace shown in the figure according to which the loop iterates 10 times. The encircled numbers at the right side of the nodes denote the number of times the node is visited during execution. Hence, $x+y$ is evaluated 11 times and $a+b$ is evaluated 15 times with 2 evaluations of $x+y$ and 7 evaluations of $a+b$ being redundant. Given the entire control flow trace, the cost model we have described is used to compute the precise probabilities given in the figure below. From these probabilities we conclude that we should enable speculative hoisting of both $x+y$ and $a+b$ above conditional node 9. However, the speculation for neither of the expressions should be enabled at conditional nodes 6, 3 and 4.



$\underline{Control\ Flow\ Trace:}$
$1-2-3-5-9-10-12$
$-2-3-6-9-10-12$
$-(2-4-7-9-11-12)^7$
$-2-4-6-8-9-11-12-13$

Precise Probabilities	n			
	9	6	3	4
$ProbCost_{x+y}(n)$	$\frac{1}{10}$	$\frac{1}{2}$	$\frac{0}{2}$	$\frac{1}{8}$
$ProbBenefit_{x+y}(n)$	$\frac{2}{10}$	$\frac{1}{2}$	$\frac{0}{2}$	$\frac{0}{8}$
$ProbCost_{a+b}(n)$	$\frac{2}{10}$	$\frac{1}{2}$	$\frac{0}{2}$	$\frac{0}{8}$
$ProbBenefit_{a+b}(n)$	$\frac{7}{10}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{8}$

Fig. 10. Enabling Speculation Using Idealized Cost Model.

Cost-benefit analysis based upon probabilistic data flow analysis. The analysis we described in the preceding section is idealized in two respects. First the availability and anticipability data flow information is required for each interesting path that passes through a conditional. Second it is assumed that the entire control flow trace is available so that the execution frequencies of each of these interesting paths can be determined. While it is possible to compute path-based data flow information and maintain complete control flow traces using the techniques described in [20] and [27; 43] respectively, for practical reasons it may be desirable to explore less expensive alternatives. In this section we present a detailed cost-benefit algorithm which uses *probabilistic data flow analysis* (PDFA) guided by *two-edge* profiles. Therefore this algorithm neither requires the entire control flow trace nor does is compute data flow information on a per path basis.

While traditional data flow analysis calculates whether a data flow fact holds or does not hold at some program point, PDFAs calculate the probability with which a data flow fact will hold at some program point. Therefore when applied to the data flow problems of *availability* and *anticipability*, PDFA provides us with the following probabilities: $AvailProb_{exp}(n)$ is the probability that expression exp is available at node n and $AntiProb_{exp}(n)$ is the probability that exp is anticipable at node n. Therefore we reformulate the speculation enabling condition for expression exp at a conditional node n in terms of these probabilities as follows:

$$EnableSpec_{exp}(n) = ProbCost_{exp}(n) < ProbBenefit_{exp}(n),$$

$$where$$

$$ProbBenefit_{exp}(n) \approx AvailProb_{exp}(n) \times AntiProb_{exp}(n) \quad and$$

$$ProbCost_{exp}(n) \approx (1 - AvailProb_{exp}(n)) \times (1 - AntiProb_{exp}(n)).$$

It is important to note that the above formulation is an approximation of the prior cost model because it is based upon the assumption that *availability* and *anticipability* at a conditional node are *independent events*. Therefore we have computed the probability of following a path that benefits (incurs a cost) by taking the product of probabilities for availability (not availability) and anticipability (not anticipability). However, as we know, in practice the outcomes of conditionals in the program may be correlated. Therefore the above model is approximate even though it uses precise probabilities for availability and anticipability.

Let us reconsider the example of Fig. 10 in light of the above cost model. Before we can use the above model, we should compute the availability and anticipability probabilities. We compute the precise probabilities using the control flow trace. As we can see from the results in Fig. 11, even though the estimates for cost and benefit probabilities have changed, their relationship has not changed. Therefore as before we again conclude that speculation should only be enabled at conditional node 9 for the two expressions.

While in the above example we used an approximation of the cost model, we still used precise availability and anticipability probabilities derived from the control flow trace. Next we discuss how these probabilities can be approximated using PDFA based upon two-edge profiles.

Precise Probabilities	n			
	9	6	3	4
$AvailProb_{x+y}(n)$	$\frac{9}{10}$	$\frac{1}{2}$	$\frac{0}{2}$	$\frac{0}{8}$
$AntiProb_{x+y}(n)$	$\frac{2}{10}$	$\frac{1}{2}$	$\frac{2}{2}$	$\frac{7}{8}$
$AvailProb_{a+b}(n)$	$\frac{7}{10}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{8}$
$AntiProb_{a+b}(n)$	$\frac{8}{10}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{7}{8}$

Approximate Probabilities	n			
	9	6	3	4
$ProbCost_{x+y}(n)$	0.08	0.25	0	0.125
$ProbBenefit_{x+y}(n)$	0.18	0.25	0	0
$ProbCost_{a+b}(n)$	0.06	1	1	0.125
$ProbBenefit_{a+b}(n)$	0.56	0	0	0

Fig. 11. Enabling Speculation Using Precise Availability and Anticipability Probabilities.

The first probabilistic data flow system which is based upon edge probabilities was developed by Ramalingam [38]. A demand-driven frequency analyzer whose cost can be controlled by permitting a bounded degree of imprecision was proposed by Bodik *et al.* [6]. Since edge frequencies do not capture any branch correlation, the approximations introduced in computed probabilities can be considerable. Therefore Mehofer and Scholz [30] proposed a data flow system based upon two-edge probabilities. Instead of relating the data flow facts at a node only with the data flow facts at the immediate predecessor nodes in the flow graph using one-edge probabilities, the data flow facts at a node are related to two predecessor nodes using a flow graph with two-edge probabilities. In this way significantly better results can be achieved as presented in [30]. Of course, this approach can be made more precise by extending it beyond two edges; however, it will undermine our goal of computing the probabilities efficiently.

Availability and anticipability are calculated based on the traditional formulations of the data-flow problems as shown in Fig. 12a. Availability analysis is guided by the local Boolean predicates *LocAvail* and *LocBlkAvail*. $LocAvail_{exp}(n)$ equals *true*, if there is an occurrence of *exp* in n which is not blocked by a subsequent statement of n, that is, a statement which modifies variables used in *exp*. $LocBlkAvail_{exp}(n)$ equals *true*, if *exp* is blocked by some statement of n, that is, a statement modifying a variable used in *exp*. Boolean conjunction is denoted by $\wedge$, Boolean disjunction by $\vee$, and Boolean negation is denoted by a bar. Prefix *N* and *X* are used as abbreviations of entry and exit, respectively. The confluence operator $\bigwedge$ indicates that the availability problem is a forward all-problem.

Availability Analysis:

—$LocAvail_{exp}(n)$: There is an expression *exp* in n which is available at the exit of n (downwards exposed).

—$LocBlkAvail_{exp}(n)$: The expression *exp* is blocked by some statement of n.

$$\begin{aligned} N\text{-}Avail_{exp}(n) &= \begin{cases} false & if \;\; n = start \;\; node \\ \bigwedge\limits_{m \in pred(n)} X\text{-}Avail_{exp}(m) & otherwise \end{cases} \\ X\text{-}Avail_{exp}(n) &= LocAvail_{exp}(n) \vee (N\text{-}Avail_{exp}(n) \wedge \overline{LocBlkAvail_{exp}(n)}) \end{aligned}$$

Anticipability Analysis:

—$LocAnti_{exp}(n)$: There is a hoisting candidate *exp* in n (upwards exposed).

—$LocBlkAnti_{exp}(n)$: The hoisting of *exp* is blocked by some statement of n.

$$\begin{aligned} N\text{-}Anti_{exp}(n) &= LocAnti_{exp}(n) \vee (X\text{-}Anti_{exp}(n) \wedge \overline{LocBlkAnti_{exp}(n)} \\ X\text{-}Anti_{exp}(n) &= \begin{cases} false & if \;\; n = end \;\; node \\ \bigwedge\limits_{m \in succ(n)} N\text{-}Anti_{exp}(m) & otherwise \end{cases} \end{aligned}$$

(a) Original Data Flow Equations.

Dual Availability Analysis:

$$\begin{aligned} N\text{-}NoAvail_{exp}(n) &= \begin{cases} true & if \;\; n = start \;\; node \\ \bigvee\limits_{m \in pred(n)} X\text{-}NoAvail_{exp}(m) & otherwise \end{cases} \\ X\text{-}NoAvail_{exp}(n) &= (LocBlkAvail_{exp}(n) \wedge \overline{LocAvail_{exp}(n)}) \vee \\ &\quad\; (N\text{-}NoAvail_{exp}(n) \wedge \overline{LocAvail_{exp}(n)}) \end{aligned}$$

Dual Anticipability Analysis:

$$\begin{aligned} N\text{-}NoAnti_{exp}(n) &= (LocBlkAnti_{exp}(n) \wedge \overline{LocAnti_{exp}(n)}) \vee \\ &\quad\; (X\text{-}NoAnti_{exp}(n) \wedge \overline{LocAnti_{exp}(n)} \\ X\text{-}NoAnti_{exp}(n) &= \begin{cases} true & if \;\; n = end \;\; node \\ \bigvee\limits_{m \in succ(n)} N\text{-}NoAnti_{exp}(m) & otherwise \end{cases} \end{aligned}$$

(b) Data Flow Equations After Transformation.

Fig. 12. Availability and Anticipability Analysis.

Similarly, anticipability analysis is guided by the local predicates *LocAnti* and *LocBlkAnti*. $LocAnti_{exp}(n)$ equals *true*, if there is an occurrence of *exp* in n which is not blocked by a preceding statement of n, that is, a statement which modifies variables used in *exp*. $LocBlkAnti_{exp}(n)$ equals *true*, if *exp* is blocked by some statement of n, that is, a statement modifying a variable used in *exp*. The confluence operator $\wedge$ indicates that the anticipability problem is a backward all-problem.

Since probabilistic data-flow systems are based on any-problems in which the meet operator is union, the analyses problems have to be transformed as described by Reps et al. [39]. If a "must-be-X" problem is an intersection problem, then the "may-not-be-X" problem is a union problem. The solution of the "must-be-X" problem is the complement of the solution of the "may-not-be-X" problem. Fig. 12b presents the data flow equations after complementing the original ones. Note that the initializations of the start and end node have to be changed appropriately as well. Since now the confluence operator for both problems is union, the PDFA framework can be applied in a straightforward way. The solutions $N\text{-}NoAvail_*$, $X\text{-}NoAvail_*$, $N\text{-}NoAnti_*$, and $X\text{-}NoAnti_*$ of the equation systems denote the probabilities of the complementary problems. Thus, the probabilities for availability and anticipability are given by $N\text{-}Avail_* = 1 - N\text{-}NoAvail_*$, $X\text{-}Avail_* = 1 - X\text{-}NoAvail_*$, $N\text{-}Anti_* = 1 - N\text{-}NoAnti_*$, and $X\text{-}Anti_* = 1 - X\text{-}NoAnti_*$.

Next we discuss PDFAs in more detail and specify the equation system for the two-edge approach. PDFAs as presented in [38; 30] are applicable for a large class of data flow problems called *finite bi-distributive subset problems*. This class of data flow problems requires (i) a finite set of data flow facts and (ii) data flow functions that distribute over both set union and set intersection. This class of data flow problems is more general than bit-vector problems but less general than the class of finite distributive subset problems introduced by Reps *et al.* [39].

PDFA equations are built by utilizing a so-called *exploded control flow graph* (ECFG) [39] which is created from the original CFG and a data flow problem. Fig. 13 depicts the ECFG for the availability problem for a subgraph of our example consisting of the nodes 1, 2, 3, 4, and 5.

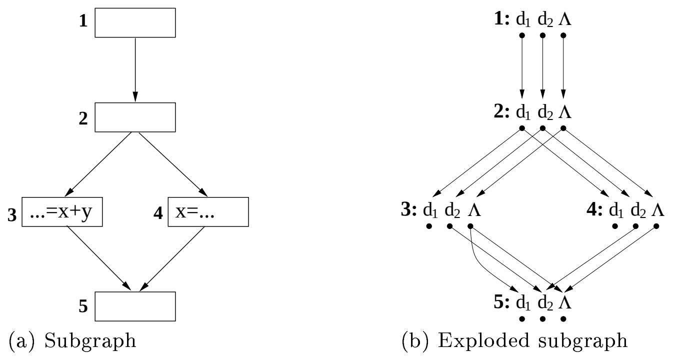


Fig. 13. CFG and ECFG of subgraph of our example.

The ECFG has $N \times (D \cup \{\Lambda\})$ nodes with N denoting the node set of the corresponding CFG and D denoting the data flow information set extended by the special symbol Λ used to calculate node frequencies. Data flow facts d_1 and d_2 designate expressions $x + y$ and $a + b$, respectively. The edges of the ECFG are derived from the representation relation [39] of a data flow function f, $R_f \subseteq (D \cup \{\Lambda\}) \times (D \cup \{\Lambda\})$, as follows:

$$\begin{aligned} R_f = \quad & \{(\Lambda, \Lambda)\} \\ \cup \ & \{(\Lambda, y) | y \in f(\phi)\} \\ \cup \ & \{(x, y) | y \in f(\{x\}) \wedge y \notin f(\phi)\}. \end{aligned}$$

Note that nodes (n, Λ) are always connected with the successor nodes. Nodes 1 and 2 do not affect availability which is described by the identity function where all nodes are connected with the corresponding "exploded" successor nodes. However, since data flow fact d_1 is generated at node 3, we have an edge from $(3, \Lambda)$ to $(6, d_1)$. On the other hand, since data flow fact d_1 is killed at node 4, there is no outgoing edge from node $(4, d_1)$. In this way an ECFG represents DFA functions explicitly.

For the two-edge approach we require two-edge probabilities $p(u, v, w)$ which specify the probability that execution will follow edge `vw` once edge `uv` has been reached.

$$p(u, v, w) = \begin{cases} \frac{\mathit{occurs}(\mathtt{uvw}, \pi)}{\mathit{occurs}(\mathtt{uv}, \pi)} & \text{if } \mathit{occurs}(\mathtt{uv}, \pi) \neq 0, \\ 0 & \text{otherwise.} \end{cases}$$

Function *occurs* denotes the number of occurrences of a path in the control flow trace π. Further, we need the notion of "predecessor" edge similar to predecessor node.

$$\mathit{In}(\mathtt{vw}, \delta) = \{(u, \delta'') | (u, \delta'') \rightarrow (v, \delta') \rightarrow (w, \delta) \in \mathit{ECFG}\}.$$

$\mathit{In}(\mathtt{vw}, \delta)$ denotes the set of predecessor edges of edge `vw` with data fact δ in the ECFG. Finally, Fig. 14 gives the equation system of the two-edge approach.

The unknowns of the equation system are related to the ECFG edges. Unknown $y(\mathtt{vw}, d)$ denotes the expected frequency of data flow fact d to hold true at node `w` under the condition that edge `vw` has been taken, and $y(\mathtt{vw}, \Lambda)$ denotes the expected number of times that edge `vw` is executed. The core of the equation system is Equation 3. The unknown related to edge `vw` and fact δ is linked to all unknowns related to precessor edges weighting it with a two-edge probability.

For initialization reasons we introduce an artificial "root" edge $\epsilon\mathtt{s}$ from the pseudo node ϵ to the start node s. Equation 1 and Equation 2 show the initialization of the equation system with $c(d)$ denoting the initial values of the corresponding data flow problem. Equation 4 describes the relation between nodes and edges in the ECFG. Once the equation system with the frequency unknowns has been solved, $\mathit{Prob}(v, d) = y(v, d)/y(v, \Lambda)$ denotes the probability of data flow fact d to hold true in node v. The two-edge equation system of Fig. 14 has been presented in terms of forward data flow problems. For backward problems the two-edge probability and the equation system must be adapted accordingly.

(1) $y(\epsilon\mathtt{s}, \Lambda) = 1$

for all d in D:
(2) $y(\epsilon\mathtt{s}, d) = c(d)$

for all $\mathtt{vw}$ in E: for all δ in D:

$$(3) \quad y(\mathtt{vw}, \delta) = \sum_{(u,\delta') \in In(\mathtt{vw},\delta)} p(u, v, w) * y(\mathtt{uv}, \delta')$$

for all w in N: for all δ in D:

$$(4) \quad y(\mathtt{w}, \delta) = \sum_{u \in pred(v)} y(\mathtt{vw}, \delta)$$

Fig. 14. Two-Edge Equation System.

The probabilities for availability of the two-edge approach coincide with the precise results for our example. Note that for solving the availability of $x + y$ precisely at conditional node 9, it is important to figure out whether path $3 - 6 - 9$ or $4 - 6 - 9$ has been taken. Fortunately, the two-edge approach succeeds in enabling hoisting of both expressions at node 9. However, in general, the probabilities may deviate from the precise results such that beneficial opportunities for improvements are missed. Thus there exists a trade-off between preciseness of the results having possibly direct influence on the optimizations performed and the effort required to obtain those results.

Speculative code motion framework. We now present the speculative code motion framework that is used to carry out the optimization. This framework hoists expressions to appropriate points in the program using safe code motion across conditionals at which speculation is disabled and speculative code motion across conditionals where speculation is enabled. A number of PRE algorithms can be found in the literature. We adapt the safe code motion based PRE algorithm proposed by Steffen [41].

The original analysis in [41] consists of a backward data flow analysis phase followed by a forward data flow analysis phase. The backward data flow is used to identify all *down-safe* points, that is, points to which expression evaluations can be safely hoisted. Forward data flow analysis identifies the *earliest* points at which expression evaluations can be placed. Finally, the expression evaluations are placed at points that are both *earliest* and *down-safe*. We modify the *down-safety* analysis to take advantage of speculation past the conditional nodes where speculation has been enabled. The *earliestness analysis* remains unchanged.

The placement points identified by the above algorithm are entry or exit points of nodes in the control flow graph. In some situations the flow graph may not contain a node at the appropriate placement point. To ensure this never happens, certain so-called *critical edges* are split and a synthetic node is introduced. These are edges from nodes with more than one successor to nodes with more than one predecessor.

Therefore in the flow graph of Fig. 10, edge $6 \to 9$ is a critical edge. Thus a node is introduced along this edge before performing the analysis for carrying out PRE.

The data flow equations for computing down-safety of an expression exp are as follows. An expression exp is down-safe at entry of node n if it is either computed in n (i.e., $Used_{exp}(n)$ is true) or it is down-safe at n's exit and preserved by n (i.e., $Pres_{exp}(n)$ is true). The expression exp is down-safe at the exit of a conditional node n if one of the following conditions is true: exp is down-safe at entry points of all of n's successor nodes; or speculation of exp has been enabled at n (i.e., $AppEnabSpec_{exp}(n)$ is true) and exp is down-safe at least one of n's successor nodes. The first of the above two conditions was used in Steffen's original code motion framework as it carries out safe code motion. The second condition has been added to allow useful speculative motion of exp to occur. If the node being examined is not a conditional node, then only the first of the two conditions is checked.

$$N-DSafe_{exp}(n) = Used_{exp}(n) \vee (Pres_{exp}(n) \wedge X-DSafe_{exp}(n))$$

$$X{-}DSafe_{exp}(n) = \begin{cases} False & n \text{ is the exit node} \\ \bigwedge_{m \in Succ(n)} DSafe_{exp}(m) & n \text{ is a conditional} \\ \vee(AppEnabSpec_{exp}(n) \wedge \bigvee_{m \in Succ(n)} Dsafe_{exp}(m)) & \\ \bigwedge_{m \in Succ(n)} DSafe_{exp}(m) & otherwise \end{cases}$$

The outcome of applying the above algorithm to the example of Fig. 10 is shown in Fig. 15. From the results of down-safety analysis we observe the following. The expressions $x + y$ and $a + b$ are both down-safe at the entry of node 9 because speculation is enabled at 9 for both expressions. Although we did not formally describe the earliestness analysis, it is easy to informally observe that the earliest points at which $x + y$ is down-safe are the entry point of node 3 (since $x + y$ is not down-safe at the exit of node 2) and the exit point of node 4 (since $x + y$ is not down-safe at the entry of node 4). Therefore evaluations of $x + y$ are placed at these points and assigned to the temporary tx which replaces all of the original occurrences of $x + y$ in the program. Similarly we also observe that the earliest points at which $a+b$ is down-safe are the entry point of node 5, entry point of node 7, exit point of node 8, and entry point of node 8a. Evaluations of $a + b$ are placed at these four points and assigned to the temporary ta which replaces all original occurrences of $a + b$.

The optimized placement results in 10 evaluations of $x + y$ and $a + b$ reducing the evaluations of $x + y$ by 1 and $a + b$ by 5, respectively. Hence, the enabling of speculation for the two expressions at node 9 was profitable and resulted in more efficient code. This example also illustrates the need for breaking critical edges since an evaluation of $a + b$ has been placed in the newly created node $8a$.

3.4 Cost of Analysis

When applied to the entire program, the compile-time cost of a profile-guided optimization algorithm can be expected to be higher than its non profile-guided coun-

n	exp			
	x+y		a+b	
	X-DSAFE	N-DSAFE	X-DSAFE	N-DSAFE
1	F	F	F	F
2	F	F	F	F
3	T	T	F	F
4	T	F	F	F
5	T	T	T	T
6	T	T	F	F
7	T	T	T	T
8	T	T	T	F
8a	T	T	T	T
9	T	T	T	T
10	F	T	F	F
11	F	F	F	T
12	F	F	F	F
13	F	F	F	F

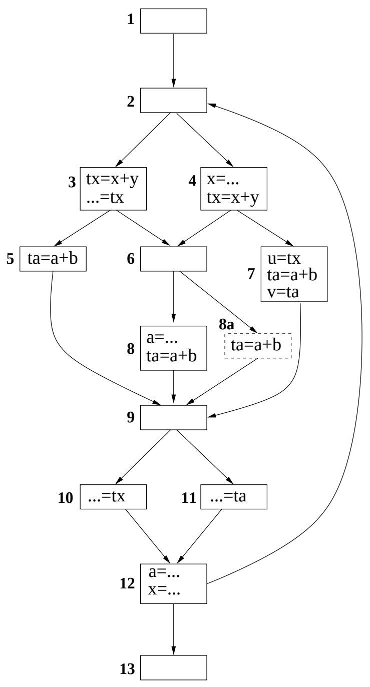


Fig. 15. Speculative PRE based Upon Modified Down-Safety Analysis.

terpart. For example, the speculative PRE algorithm of the preceding section is more expensive than the traditional safe code motion based algorithm. Additional cost results from the cost-benefit analysis associated with enabling speculation. However, the following approaches can be applied to limit the cost of profile-guided algorithms:

—The application of optimization algorithms can be *limited* so that only the code belonging to frequently executed (hot) program regions is aggressively optimized.

—Instead of carrying out exhaustive data flow analysis, we can employ *demand driven analysis* techniques [16; 17; 24]. These techniques limit the cost of data flow analysis by only computing data flow facts that are relevant to a code optimization. This approach is particularly beneficial for expensive analyses (e.g., interprocedural analysis).

—Conservatively *imprecise frequency analysis* techniques can be used to limit the cost of frequency analysis. A frequency analyzer whose cost can be controlled by permitting a bounded degree of imprecision was proposed by Bodik *et al.* [6]. In addition, this analyzer is demand-driven.

4. PROFILE GUIDED MEMORY OPTIMIZATIONS

Over the past decade, while the processor speeds have risen by 55% per year, the memory speeds have only improved by 7% per year. As a result, the cache miss penalties have increased from several cycles to over 100 cycles. Since the memory accesses, especially those due to loads, are on the critical path, cache misses greatly reduce the ability of a modern processor to effectively exploit instruction level parallelism.

Broadly speaking, there are two classes of optimization techniques aimed at improving cache performance: those that reduce the number of cache misses to minimize memory stalls and others that better tolerate cache miss penalty. The focus of this section is on optimizations of the first type.

A program's address space is divided into three parts: heap, stack, and global space. Stack data typically exhibits good cache behavior. Therefore most of the research has been aimed at improving the cache locality for statically allocated global data and dynamically allocated heap data. While the optimizations for statically allocated data are typically carried out at compile-time, optimizations applicable to dynamically allocated data must use a combination of compile-time decisions and runtime actions. The techniques used to improve data locality can be divided into following categories:

—*Object placement* techniques determine a placement of data objects in relation to each other which provides improved cache behavior.

—*Object layout* techniques determine a layout of fields within a large data object to improve cache locality.

—*Object layout and placement* techniques alter both the layout of fields with in an object as well as the placement of data objects in relation to each other.

—*Object compression* techniques improve cache behavior by reducing the data memory footprint of a program

4.1 Object Placement

The address profiles provide the compiler with the information useful in organizing the placement of objects in memory in relation to each other. For example, we can categorize objects as frequently referenced *hot objects* and infrequently accessed *cold objects.* The hot objects can be placed together separate from the cold objects. By further seeing which hot objects are frequently used together, the placement of hot objects in relation to each other can be determined.

For statically allocated global data, object placement can be carried out at one time during compilation. However, the placement of heap objects can only be carried incrementally and at runtime as they are dynamically allocated. Standard system provided memory allocators (e.g., malloc) do not provide any control over data placement. There are two proposed approaches for placement of heap objects: *object migration* and *memory clustering.*

Object migration. In this approach standard memory allocators are used and therefore when the heap objects are initially allocated their placement is not optimized. At runtime the objects are migrated from their original locations to other ones in order to improve cache locality. There are two main challenges of carrying out effective object migration.

First there is a substantial overhead of object migration. There are two sources of this overhead: object migration itself requires extra execution time and access to migrated objects may require additional operations is some situations. If the gain of improved cache locality outweighs the migration overhead, only then is the object migration optimization profitable. The role of address profiles is to identify objects that should be colocated through migration.

Second challenge is that of maintaining program correctness in presence of object migration. If the programmer makes use of location of data in developing the code, clearly object migration will lead to correctness problems. For example, consider the objects corresponding to variable `a` and structure pointed to by `p` shown below. The user may assume that the objects are colocated and use address arithmetic in accessing the fields. In particular, the user may access $p \rightarrow c$ data field using $\&a + 2 * sizeof(int)$. Clearly object migration will cause such code to fail.

```
int   a;
struct {
      int   b;
      int   c;
}  *p;
```

One approach to addressing the correctness issue is to provide programming guidelines that restrict the manner in which data can be accessed. Address arithmetic can be allowed but only within an object, that is, we can prohibit the user from deriving the address of one object from the address of another object. This restriction would eliminate the problem illustrated in the above example. Another approach is to provide hardware support that may enable safe application of object migration in some situations.

Luk and Mowry [28] have proposed hardware support for *memory forwarding* that uses limited hardware support to enable aggressive object migration. The problem addressed by this support is as follows. When an object is migrated, pointers to that object may exist elsewhere in the program. Since these pointers will be out of date, any accesses through them will be illegal. To correctly handle accesses to migrated objects through such pointers the following support is provided. An extra bit is attached to each machine word which has the following semantics. If the bit attached is set, it indicates that the object that resided at that location has been migrated and the location now contains a forwarding address for the migrated object. The load and store instructions are implemented to carry out the extra level of dereferencing based upon the contents of the bit attached to the memory location being addressed. Thus this approach enables migration of objects if forwarding addresses are left behind at locations where the objects originally resided. It should be noted that extra overhead is associated with memory forwarding both in terms of extra storage that is needed to hold forwarding addresses and extra execution time required to carry out extra level of dereferencing.

The example in Fig. 16 illustrates the above approach. It shows a link list where initially all the items in the link list reside at their originally assigned locations. The link list after some of the elements have been migrated to adjacent memory locations is shown next. The list elements are migrated so that fewer cache misses occur when the list is traversed. Note that the original locations of the migrated list elements now contain forwarding addresses. If there are any dangling pointers that still point to these old addresses, they are forwarded to the new addresses by the hardware.

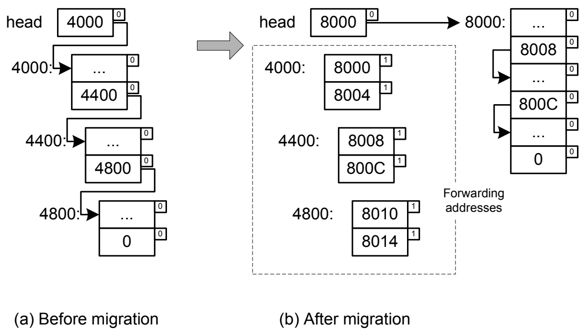


Fig. 16. Memory Forwarding.

Memory clustering. This approach supports memory management routines that provide limited user level control over object placement. Therefore unlike object migration which dynamically changes object placement, this approach places objects in more desirable locations to begin with. As a result the migration overhead is avoided. However, this approach burdens the programmer with the task of providing hints to the memory allocator for making good object placement decisions.

An example of a user level memory allocator is `ccmalloc` allocator proposed by Chilimbi *et al.* [11]. In addition to providing the size of the object during memory allocation, the user provides a pointer to an existing object that is likely to be accessed contemporaneously with the newly allocated object. Whenever possible, `ccmalloc` allocates the new object in the same cache block as the existing object. The address profiles can be used by the user to identify objects that are accessed contemporaneously.

In Fig. 17 we show the grouping of nodes of a binary tree with and without clustering. Group of nodes belonging to the same cache block are shown by the shading in the figure. The grouping prior to clustering reflects the order in which the nodes are created. The grouping following the use of `ccmalloc` allocates a group of neighboring nodes to the same cache block. Assuming that this decision was based upon address profiles of perhaps a breadth first traversal of the tree, the optimized placement of tree nodes will lead to better cache performance.

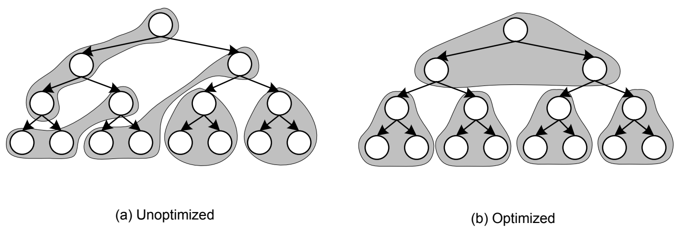


Fig. 17. A `ccmalloc` Example.

While the above approach requires programmer's involvement, it is also possible to automate this process using address profiles. We can identify references that appear close to each other in the address profiles and generate a signature for these group of references. When creating nodes at runtime, attempt can be made to colocate the nodes with the same signature.

4.2 Object Layout

A data structure is often defined by the programmer to support code readability. Therefore logically related data fields are often put together. The compiler simply uses a memory layout for the fields that mirrors the order in which the fields are declared. However, this order may not be consistent with the ordering that incurs fewer cache misses. In order to improve cache performance, the layout of fields within an object can be reordered so that the fields that are accessed frequently and contemporaneously are placed close to each other. Of course this transformation is only useful if the object is large enough that it extends across multiple cache blocks. Truong *et al.* [34] evaluated this approach and showed that when a node spans several cache blocks, object layout optimization implicitly takes advantage of cache line prefetching and reduces cache pollution; thus improving cache performance.

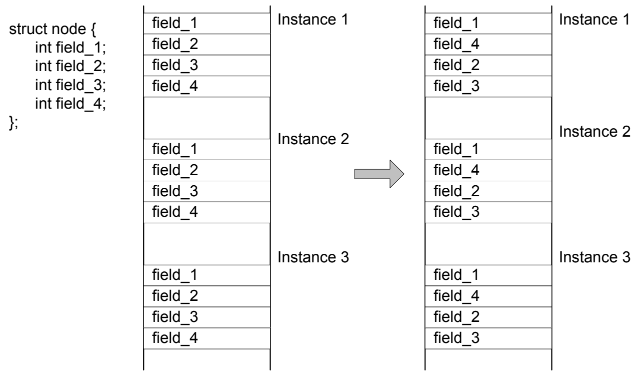


Fig. 18. Field Reorganization.

Fig. 18 shows the default memory layout used by the compiler on the left. On the right the layout generated after profile-guided field reorganization is shown under the assumption that the cache block is large enough to hold two fields and fields 1 and 4 are used together. A pair of accesses of fields 1 and 4 would generate two cache misses before reorganization and only one cache miss after reorganization.

4.3 Object Layout and Placement

Some highly aggressive locality improving transformations result is simultaneous changing of object layouts and their placement with respect to each other. Consider a data structure that contains multiple objects (or nodes) of the same type. Moreover each object contains several fields, some are hot (frequently used) and while others are cold (rarely used). Two transformations, *instance interleaving* and *object splitting*, have been proposed to improve the cache behavior of such data structures.

Instance interleaving. Instead of allocating an object in contiguous set of memory locations, this approach proposed by Truong *et al.* [34], interleaves the storing of multiple instances of object instances of the same type. Hot fields from different object instances are stored together and so are the cold instances. The application of this transformation to the example of Fig. 18, is illustrated in Fig. 19. The hot fields (1 and 4) from different object instances are stored together in one place and the cold fields (2 and 3) from different object instances are also stored together. Clearly from this example we can observe that both the internal layout and relative placement of objects is changed by this transformation.

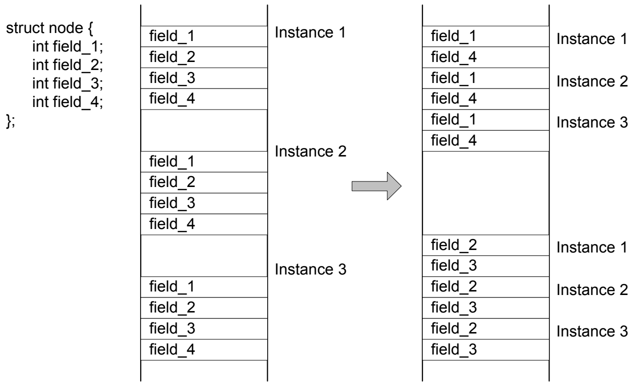


Fig. 19. Interleaving Instances of the Same Type.

Object splitting. Chilimbi *et al.* [12] also proposed a transformation that separates hot fields within an object from the cold fields. In fact they split an object into two parts, the hot primary part and the cold secondary part. Hot fields are accessed directly, while a pointer to the cold part is stored within the hot part and thus an extra level of indirection is involved in accessing cold fields. This transformation does not explicitly interleave the hot (cold) parts from different object instances. The locality across objects can be improved by using memory clustering (e.g., `ccmalloc`) to group together hot (or cold) fields from different object instances. The example in Fig. 20 illustrates this transformation. Compared to instance interleaving, this approach is much cleaner and easy to manage as the modification to the original source code is easier to perform.

It is important to note that changing an object's layout alone is not useful if the object is small and fits within a cache block. However, when layout and placement are simultaneously changed, it is beneficial to even split small objects as the splitting will allow clustering of a greater number of hot parts of objects into a single cache block.

4.4 Object Compression

A complimentary technique for improving data cache behavior is that of object compression. This technique improves cache performance by reducing the memory requirement by packing greater amounts of data per cache block. Higher data

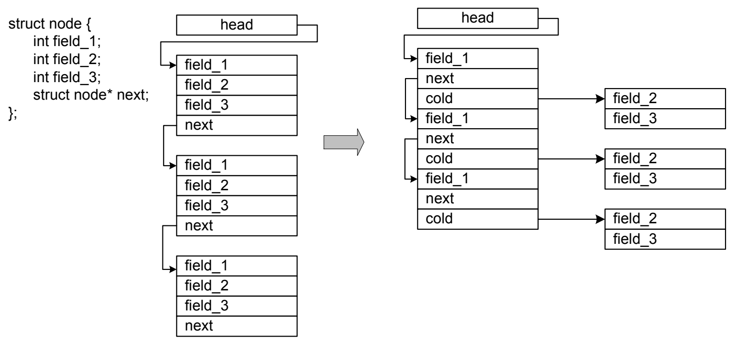


Fig. 20. Object Splitting.

density per cache block reduces the number of cache misses that take place when accessing a given amount of data from memory.

Compile-time techniques have been developed to exploit presence of narrow width data to achieve compression. Stephenson *et al.* [33] have proposed bitwidth analysis that is used to discover narrow width data by performing value range analysis. For example, a flag declared as an integer may take only 0 or 1 values. Once the compiler has proven that certain data items do not require a complete word of memory, they are compressed to a smaller size. This approach is particularly useful for programs with narrow width multimedia data which is not packed by the user. The work by Davidson and Jinturkar [14] also carries out memory coalescing using compile-time analysis. The above techniques share one common characteristic, they are applicable only when the compiler can determine that the data being compressed is *fully compressible* and they only apply to *narrow width non-pointer* data.

Zhang and Gupta [45] have proposed profile-guided compression transformations that apply to *partially compressible* data and, in addition to handling narrow width non-pointer data, they also apply to *pointer data.* Therefore these transformations are quite effective in compressing heap objects. The generality of this approach is quite important because experience has shown that while heap allocated data structures are highly compressible, they are almost never fully compressible. This approach is also simpler in one respect. It does not require complex compile-time analysis to prove that the data is always compressible. Instead simple profile-guided heuristics can be used by the compiler to determine that the data structure being compressed contains mostly compressible data.

Compression of partially compressible data structures is based upon the observation by Zhang *et al.* [44] that majority of the data values encountered in data memory allocated to a program can be divided into two categories: very small constant values and very large address values. In a node belonging to a heap allocated data structure we are likely to have some data fields which may contain small constants and pointer fields which contain large address values. If the 18 higher order bits of small constants are the same, then we can truncate the value to 15 lower order bits as the remaining bits can be simply obtained by replicating the highest order bit of this truncated entity. If the pointer field stored in a node shares a common 17 bit prefix with the address at which the node itself is stored, then we

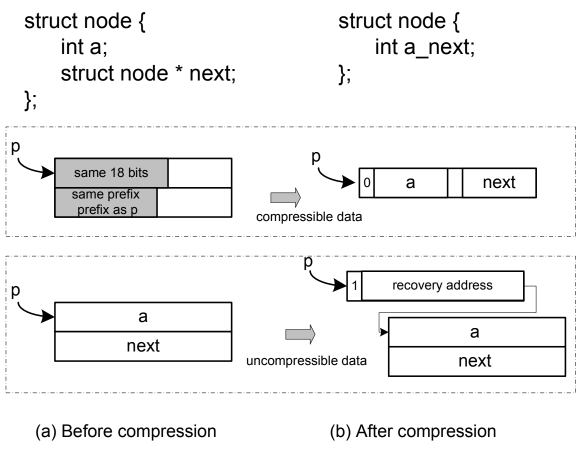


Fig. 21. Compression of Partially Compressible Data Structure.

can truncate the pointer field also to 15 bits. This is because the full contents of the pointer can always be reconstructed from the address at which the node resides. Two truncated 15 bit entities, each representing either integer data or a pointer, can be compressed into a single word. Of the two remaining bits of a 32 bit word, one is unused and the other is used to indicate whether or not the word contains two compressed values. The word will not contain two compressed values in case the values are not found to be compressible at runtime. In this case it contains a pointer to a location where the two fields are stored in uncompressed form.

The benefits of the above optimization increase with the extent to which large and critical data structures in the program can be compressed. The cost of the optimization is the additional instructions for carrying out data compression, expanding compressed data prior to their use in computations, checking the compressibility of data, and accessing uncompressible data through an extra level of indirection. If profiling indicates that the values in a data structure are mostly compressible, then the benefits are found to outweight the costs.

In Fig. 21 the node structure contains an integer field `a` and a pointer field `next`. Lets assume that profiling indicates that these fields are mostly compressible. In this situation the compiler replaces them by a single combined field. At runtime when a node is created, reduced amount of storage is allocated to accommodate field `a_next`. If the data being stored is found to be compressible then they are stored in this single word as shown in the figure. The most significant bit is set to 0 if the data is held by this compressed field. On the other hand, if the data being stored is not compressible, additional storage is allocated to store the data and a pointer to this location is placed in `a_next`. The most significant bit of `a_next` is set to 1.

4.5 Profile Guided Code Layout

The techniques discussed so far are applicable to program data. Techniques have also been proposed to improve instruction cache behavior [42; 29]. The goal of such techniques is to layout the code in a manner that increases the fraction of instructions per fetched instruction cache block that are actually executed. Note that all instructions belonging to a cache block may not be executed due to the presence of branches in the code.

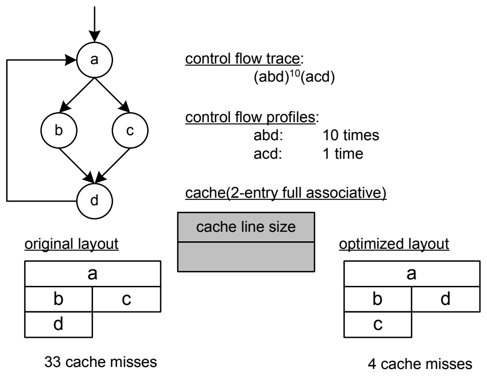


Fig. 22. Code Placement.

Consider the example in Fig. 22. The sizes of cache line and basic blocks (a, b, c, and d) are shown in the figure. For illustration purposes, let us assume that we have a 2-entry fully associative cache. If no layout optimizations are performed, it is likely that the compiler will generate the layout marked as the original layout in the figure. This layout introduces a large number of instruction cache misses primarily because the blocks a, b and d that are on the frequently executed path cannot be all kept in the cache. Meanwhile block c is fetched repeatedly but rarely executed. Using path profile data we can easily determine that it is more beneficial to use the optimized layout shown in the figure in which blocks b and d are packed into the same cache line. The number of cache misses is greatly reduced in this case. Here we have illustrated the application of code layout optimization at the basic block level. Techniques for layout optimization at procedural level have also been developed [29].

5. CONCLUDING REMARKS

In this chapter we have identified optimization opportunities that may exist during program execution but cannot be exploited without the availability of profile data. Different types of profile data that are useful for code optimization were identified. The use of this profile data in carrying out profile-guided classical and memory optimizations was discussed. We identified the key issues involved in developing profile-guided optimization algorithms and demonstrated these issues by developing a speculative PRE algorithm.

References

[1] A-H. A. Badawy, A. Aggarwal, D. Yeung, and C-W. Tseng, "Evaluating the Impact of Memory System Performance on Software Prefetching and Locality Optimizations," *The 15th Annual ACM International Conference on Supercomputing* (ICS), pages 486–500, Sorrento, Italy, June 2001.

[2] T. Ball and J.R. Larus, "Efficient Path Profiling," *The 29th IEEE/ACM International Symposium on Microarchitecture* (MICRO), pages 46–57, December 1996.

[3] T.Ball, P. Mataga and M. Sagiv, "Edge Profiling Versus Path Profiling: The Showdown," *The 25th ACM SIGPLAN-SIGACT Symposium on Principle of Programming Languages* (POPL), pages 134–148, San Diego, CA, January 1998.

[4] R. Bodik, R. Gupta, and M.L. Soffa, "Interprocedural Conditional Branch Elimination," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–158, Las Vegas, Nevada, June 1997.

[5] R. Bodik and R. Gupta, "Partial Dead Code Elimination using Slicing Transformations," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 159–170, Las Vegas, Nevada, June 1997.

[6] R. Bodik, R. Gupta, and M.L. Soffa, "Complete Removal of Redundant Expressions," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14, Montreal Canada, June 1998.

[7] R. Bodik, R. Gupta, and M.L. Soffa, "Load-Reuse Analysis: Design and Evaluation," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 64–76, Atlanta, Georgia, May 1999.

[8] R. Bodik, R. Gupta, and V. Sarkar, "ABCD: Eliminating Array Bounds Checks on Demand," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–333, Vancouver B.C., Canada, June 2000.

[9] B. Calder, P. Feller and A. Eustace, "Value Profiling," *The 30th IEEE/ACM International Symposium on Microarchitecture* (MICRO), pages 259–269, December 1997.

[10] B. Calder, C. Krintz, S. John, and T. Austin, "Cache-Conscious Data Placement," *The 8th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pages 139–149, San Jose, California, October 1998.

[11] T.M. Chilimbi, M.D. Hill, and J.R. Larus, "Cache-Conscious Structure Layout," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 1–12, Atlanta, Georgia, May 1999.

[12] T.M. Chilimbi, B. Davidson, and J.R. Larus, "Cache-Conscious Structure Definition," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 13–24, Atlanta, Georgia, May 1999.

[13] T.M. Chilimbi, "Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 191–202, Snowbird, Utah, June 2001.

[14] J. Davidson and S. Jinturkar, "Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 186–195, Orlando, FL, June 1994.

[15] D.M. Dhamdhere, "Practical Adaptation of Global Optimization Algorithm of Morel and Renvoise," *ACM Transactions on Programming Languages*, Vol. 13, No. 2, pages 291–294, 1991.

[16] E. Duesterwald, R. Gupta, and M.L. Soffa, "A Practical Framework for Demand-Driven Interprocedural Data Flow Analysis," *ACM Transactions on Programming Languages and Systems*, Vol. 19, No. 6, pages 992–1030, November 1997.

[17] E. Duesterwald, R. Gupta, and M.L. Soffa, "Demand-driven Computation of Interprocedural Data Flow," *ACM SIGPLAN-SIGACT 22nd Symposium on Principles of Programming Languages* (POPL), pages 37–48, San Francisco, California, January 1995.

[18] C. Dulong, "The IA-64 Architecture at Work," *IEEE Computer*, Vol. 31, No. 7, pages 24–32, July 1998.

[19] R. Gupta, "A Fresh Look at Optimizing Array Bound Checks," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 272–282, White Plains, NY, June 1990.

[20] R. Gupta, D. Berson, and J.Z. Fang, "Path Profile Guided Partial Redundancy Elimination Using Speculation," *IEEE International Conference on Computer Languages* (ICCL), pages 230-239, Chicago, Illinois, May 1998.

[21] R. Gupta, D. Berson, and J.Z. Fang, "Resource-Sensitive Profile-Directed Data Flow Analysis for Code Optimization," *IEEE/ACM 30th International Symposium on Microarchitecture* (MICRO), pages 358-368, Research Triangle Park, North Carolina, December 1997.

[22] R. Gupta, D. Berson, and J.Z. Fang, "Path Profile Guided Partial Dead Code Elimination Using Predication," *International Conference on Parallel Architectures and Compilation Techniques* (PACT), pages 102-115, San Francisco, California, November 1997.

[23] R.E. Hank, W-M. Hwu, and B.R. Rau, "Region-based Compilation: An Introduction and Motivation," *The 28th IEEE/ACM International Symposium on Microarchitecture* (MICRO), pages 158–168, Ann Arbor, Michigan, 1995

[24] N. Heintze and O. Tardieu, "Demand-Driven Pointer Analysis," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 24–34, Snowbird, Utah, June 2001.

[25] J. Knoop, O. Ruthing, and B. Steffen, "Lazy Code Motion," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 224–234, San Francisco, CA, June 1992.

[26] J. Knoop, O. Ruthing, and B. Steffen, "Partial Dead Code Elimination," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 147–158, Orlando, Florida, June 1994.

[27] J. Larus, "Whole Program Paths," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 259–269, Atlanta, GA, May 1999.

[28] C-K. Luk and T.C. Mowry, "Memory Forwarding: Enabling Aggressive Layout Optimizations by Guaranteeing the Safety of Data Relocation," *The 26th IEEE/ACM International Symposium on Computer Architecture* (ISCA), pages 88–99, May 1999.

[29] S. McFarling, "Program Optimization for Instruction Caches," *The 3th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pages 183–191, April 1989.

[30] E. Mehofer and B. Scholz, "A Novel Probabilistic Data Flow Framework," *The Tenth International Conference on Compiler Construction* (CC), LNCS 2027, Springer Verlag, pages 37–51, Genova, Italy, April 2001.

[31] C. G. Nevil-Manning and I.H. Witten, "Linear-time, Incremental Hierarchy Inference for Compression," *Data Compression Conference*, Snowbird, Utah, IEEE Computer Society, pages 3–11, 1997.

[32] V. Sarkar, "Determining Average Program Execution Times and their Variance," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 298–312, Portland, Oregon, June 1989.

[33] M. Stephenson, J. Babb, and S. Amarasinghe, "Bitwidth Analysis with Application to Silicon Compilation," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 108–120, Vancouver B.C., Canada, June 2000.

[34] D.N. Truong, F. Bodin, and A. Seznec, "Improving Cache Behavior of Dynamically Allocated Data Structures," *International Conference on Parallel Architectures and Compilation Techniques* (PACT), pages 322–329, Paris, France, 1998.

[35] S. Watterson and S. Debray, "Goal-Directed Value Profiling," *The Tenth International Conference on Compiler Construction* (CC), LNCS 2027, Springer Verlag, Genova, Italy, April 2001.

[36] E. Morel and C. Renvoise, "Global Optimization by Suppression of Partial Redundancies," *Communications of the ACM*, Vol. 22, No. 2, pages 96–103, 1979.

[37] F. Mueller and D. Whalley, "Avoiding Conditional Branches by Code Replication," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 56–66, La Jolla, CA, June 1995.

[38] G. Ramalingam, "Data Flow Frequency Analysis," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 267-277, Philadelphia, PA, 1996.

[39] T. Reps and S. Horwitz and M. Sagiv, "Precise Interprocedural Dataflow Analysis Via Graph Reachability," *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (POPL), pages 49–61, San Francisco, CA, January 1995.

[40] B. Steffen, "Property Oriented Expansion," *International Static Analysis Symposium* (SAS), LNCS 1145, Springer Verlag, pages 22–41, Germany, September 1996.

[41] B. Steffen, "Data Flow Analysis as Model Checking," *TACS*, Sendai, Japan, Springer-Verlag, LNCS 526, pages 346–364, 1991.

[42] C. Young, D.S. Johnson, D.R. Karger, and M.D. Smith, "Near-optimal Intraprocedural Branch Alignment," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 183–193, Las Vegas, Nevada, June 1997.

[43] Y. Zhang and R. Gupta, "Timestamped Whole Program Path Representation," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 180–190, Snowbird, Utah, June 2001.

[44] Y. Zhang, J. Yang, and R. Gupta, "Frequent Value Locality and Value-Centric Data Cache Design," *The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pages 150–159, Cambridge, MA, November 2000.

[45] Y. Zhang and R. Gupta, "Data Compression Transformations for Dynamically Allocated Data Structures," *International Conference on Compiler Construction*, Grenoble, France, April 2002.