Value Specialization using PLTO

Tal Shaked

May 1, 2002

Abstract

It is quite common in programs for some variables and registers to take on a small set of values throughout the course of execution. A variable that takes on only one value during execution is referred to as a runtime constant. Compilers cannot optimize runtime constants since these are not known at compile time. Value specialization is the process of optimizing runtime constants as well as other variables that take on a specific value most of the time. It involves instrumenting an executable in order to keep track of the values of different variables and to find those with skewed distributions. These program points can then be specialized to run more quickly for the common value. Although the idea of value specialization is not new, implementing it for the Pentium architecture has posed some interesting challenges. This thesis describes the details involved in augmenting PLTO (Pentium Link-time Optimizer) with value specialization.

Contents

1 Introduction **3**

2 Background **3**

3 Value Profiles **4**
- 3.1 Profile Points 5
- 3.2 Value Profiles Tables 5
- 3.3 Insert Code 6

4 Value Specialization **7**
- 4.1 Determining Specialization Points 7
- 4.2 Implementing Specialization 7
- 4.3 Updating Edge Weights 9

5 Example and Results **10**
- 5.1 Program Flow 10
- 5.2 Specialized Program 13
- 5.3 Experimental Results 13

6 Related Work **15**

7 Summary **15**

8 Acknowledgements **16**

9 References **16**

1 Introduction

Computers get faster on a regular basis, but this does not meet the demand of many application developers and users who desire their programs to run even faster. The process of taking a program written in a high-level language like C and compiling it into an executable binary provides many opportunities to improve the running time of programs.

Generally optimizations that take place during the compilation process are done directly by the compiler. At this point the only information the compiler has is the source code. There is no knowledge of how execution will flow or where specifically to focus optimizations.

Recently there has been some exploration into optimizing programs via binary rewriting systems (BRS), which can transform one binary program into a different binary program that is functionally equivalent. PLTO (Pentium link-time optimizer) is a BRS developed at the University of Arizona which can read in a relocatable object program, manipulate it, and output an optimized executable. PLTO currently applies a number of optimization techniques ranging from function inlining to constant propagation to code layout.

This paper focuses on the implementation and potential effectiveness of value specialization, another optimization technique, in the context of PLTO. Value specialization is the process of optimizing programs based on knowing that a given variable at a specific program point is expected to contain a certain value most of the time. This is different than a compile-time constant that never changes and may be optimized away completely.

Past studies have shown that value specialization can be quite effective [1, 2, 3]. Moreover, many programs within the scientific computing community that take a long time to run may benefit greatly from this. More specifically MPI programs have a pattern of calling functions almost exclusively with the same arguments, but since these are run-time constants rather than compile-time constants, it is not possible for compilers effectively to optimize such calls. However, through value specialization, it will be possible to profile these variables and realize that they are basically run-time constants which can be specialized for an expected value.

Although currently PLTO is not able to value specialize on its own, much progress has been made towards achieving this goal. The first step for value specialization is obtaining value profiles, which are distribution tables for variables at specific program points. This has been implemented and works on arbitrary programs. The next step is to use these value profiles automatically to apply value specialization, which is underway. We are now able manually to select program points to specialize and hand-guide the process necessary for related optimizations.

We begin by giving an overview of PLTO. Then we look at the details on obtaining value profiles and how these are implemented. Afterwards we discuss how to use value profiles to first select program points for specialization and then optimize the program. We conclude with an example using PLTO that illustrates how value specialization works and its potential effectiveness.

2 Background

The process of optimizing a program using PLTO [5, 6] requires several stages. A program must first be compiled into a statically linked relocatable object file. This is PLTO's input binary in phase 1 of Figure 1.

The first thing PLTO does is disassemble the object file in order to create a control flow graph (CFG) of the program. A CFG is an internal representation of the program that can be easily manipulated and later converted back into a binary executable. It consists of basic blocks, which contain sequences of assembly instructions, and edges, which connect these basic blocks together and represent the flow of control of a program.

A basic block is a sequence of continuous instructions that will be executed. Naturally the boundaries of basic blocks are defined by changes in flow of execution through jumps either into a section or out of a section of code. It is easy to identify the boundaries of basic blocks by looking at the flow of execution based on jumps and function calls.

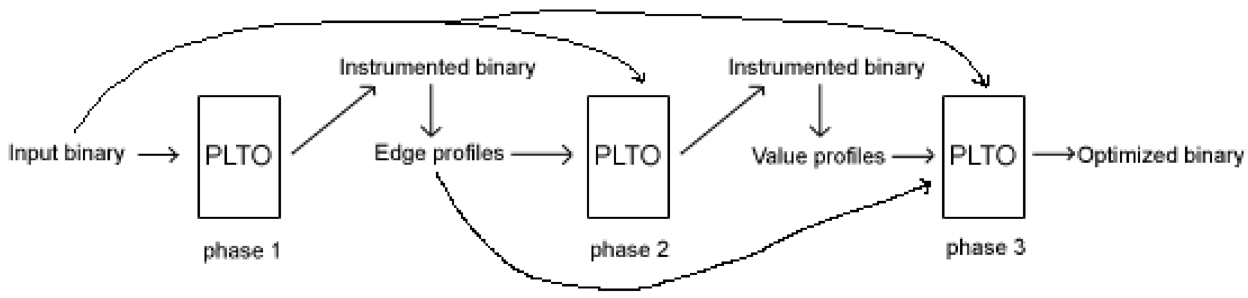

Figure 1: PTLO overview

Edges connect these basic blocks together based on the flow of control. For example, a basic block that terminates with a conditional branch will have two outgoing edges, one representing the path taken when the branch is true, and the other taken when the branch is false. This means that edges must have some semantics since flow of execution can go from one section of code (basic block) to another for different reasons (i.e. function calls, returns, conditional jumps, unconditional jumps, etc.).

Getting edge profiles, which are counts of how many times edges between basic blocks are traversed, is critical in applying later optimizations because they indicate the hot spots in a program. Therefore the first thing to do with PLTO is to instrument the code and output a binary that when run will output edge profiles. This is the output from phase 1. We then execute this binary using a simple training set to output edge profiles. The next time PLTO reads in the same input file, it will also read in the edge profiles. This allows PLTO to identify areas of code that are frequently executed, often referred to as hot spots.

After phase 1, PLTO is usually run a second time with the input file in order to apply optimizations. Since the hot spots are now identified, many optimizations can take place that are generally not possible at compile time. The result is an optimized binary that is expected to run faster.

With the addition of value specialization, a second phase must take place. Just as we needed a separate phase to instrument the code for edge profiles, we also need a phase to instrument the code for value profiles. Phase 2 in Figure 1 shows PLTO taking in the input file and reading the edge profiles in order to output an instrumented binary which will create value profiles when run with (the same) sample input. It is important to generate edge profiles before phase 2 since they are used to guide the instrumentation process for creating value profiles.

Phase 3 shows PLTO being run a third time with the same input file, this time reading in both the edge profiles and value profiles. This allows PLTO to apply all of its normal optimizations as well as value specialization.

Chapter 3 examines phase 2 and describes the steps involved in instrumenting the code in order to obtain and output value profiles. Chapter 4 looks at phase 3 and specifically discusses how value specialization is applied and implemented when creating an optimized binary. Chapter 5 walks through these three phases, taking a program from beginning to end resulting in an optimized binary that runs significantly faster than the original. Phases 1 and 2 are operational; value specialization in phase 3 currently requires hand-guidance to tell PLTO exactly how to specialize.

3 Value Profiles

Creating value profiles occurs at phase 2 of Figure 1. We first describe how to select which program points to profile. Then we examine how the profiles are stored and maintained during program execution. This is important in understanding how code is inserted to create an instrumented binary. Finally the instrumented binary is run with representative input; it dumps the value profile table to a file just before exiting. This file is read by PLTO during phase 3.[1]

[1]There is also a tool that can read the value profiles and output the information in ASCII for easy inspection.

Value	Count
0	1023
2	356
3	567
4	453
23	234
42	-1
other	456

Figure 2: Sample Value Profile Table

3.1 Profile Points

Our current implementation allows us to track the values of general purpose registers and stack locations. This means that for every instruction, we can profile up to three different points (two sources and a destination). Since programs have thousands of instructions, and therefore potentially thousands of program points to profile, it is necessary to limit which points to consider.

A good way to determine which points to profile is to compute the potential benefit that can be obtained if we know a certain value of a variable at and after a given program point. Our current benefit analysis looks ahead from a given program point to find out how many instructions can be simplified or eliminated if a value is known. It is important to notice that this process is transitive in the sense that knowing the value of one variable can lead to knowing the value of other variables, which can simplify other instructions. This benefit analysis gives an absolute estimate of how many instructions can be eliminated.

Knowing the cost of specializing a program point helps determine whether the potential benefit is worthwhile. We will show in chapter 4 that it costs at least a couple instructions to test for a specialized value. If this cost outweighs the maximum expected benefit, then clearly it is not worthwhile to profile a point since even a distribution of 1.0 for a given value will not allow any optimizations.

This combination of computing the cost and benefit of a variable at a program point is known as *cost-benefit analysis*. A cost-benefit model does this computation and is used to determine whether a point is worth profiling. Designing an accurate cost-benefit model is quite complicated and is outside the scope of this thesis. Refer to [4] for more details on this topic.

3.2 Value Profiles Tables

The instrumented binary generates value profiles during execution; these are stored in value profile tables. Prior to execution we do not know what range of values a variable will take. However, we do know that we are only interested in variables that take on a certain value a large percentage of the time. Therefore rather than keeping track of all values seen, we use a method that keeps track of only the most common values at any given time.

Figure 2 shows a distribution table with six entries that consist of (value, count) pairs, where *value* represents the value of the variable, and *count* keeps track of how many times this value was seen. The seventh (last) entry keeps count of all other values seen beyond the first six, and therefore has no specific associated *value*.

This is a dynamic distribution table because we sort and clean the table every 1000 times a variable is profiled. We sort the table in non-decreasing order of *count*, and then clean, or empty the last three entries into *other*. This keeps a total count of how many times the variable was profiled, and allows three new values to enter the table. This means that if a common value is seen after six unique values have been profiled (and filled the table), it will still have a chance to enter the table after sorting and cleaning. In our implementation a value of -1 in the *count* field represents an empty entry (meaning that the value of *value* is garbage). Table entries are initialized with this value and are reset to this value during cleaning.

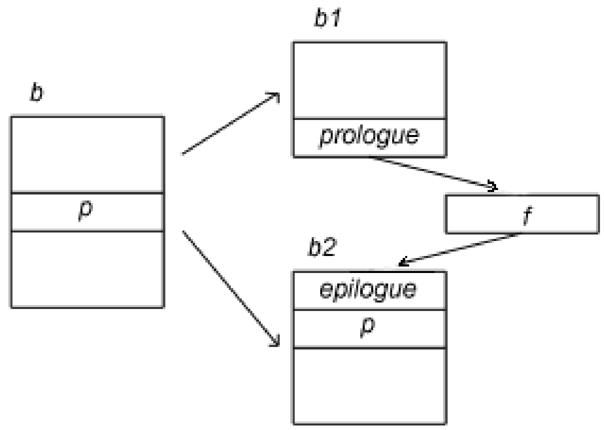

Figure 3: Inserting Profile Function f

The distribution tables for each profile point can be initialized as soon as all the profile points have been marked. At this point we know exactly how many tables are needed and can allocate the appropriate amount of space in the data section. We also create a mapping of distribution tables to profile points. This allows us to find the correct table for a given profile point during execution. Since every profile point has a unique instruction ID and source/destination within the instruction, we can use a pair of (instruction ID, operand) to map to a table. One final point is that we must also allocate space in the data section to write out this mapping of (instruction ID, operand). Then when the value profiles are read back in during phase 3, we can find the program point corresponding to a given table (assuming that instruction IDs match, which they do in PLTO for the same input binary).

3.3 Insert Code

The original program must be modified in order to profile points during execution. This is done by inserting code around the profile points that does the required bookkeeping for the distribution tables. Since we may profile many points, and therefore insert code in many places, we create a function that does this bookkeeping and call it from every profile point.

In order to profile a program point, we need to know the value seen and the corresponding distribution table. Thus our bookkeeping function is f = ProfilingFunction(int value_of_variable, void* table_reference). For each profile point, we add a call to $f(value, table)$ where *value* equals the value of a register or stack location profiled, and *table* equals the distribution table for this program point.

Since we need to insert this code in the context of PLTO, it is necessary to construct f within PLTO as well. Therefore we have to create a control flow graph for f by hand, using the PLTO data structures so that it can easily be added to the instrumented program. This is done by first writing the profiling function in C, and then compiling it into optimized assembly. Then within PLTO we create structures for all of the instructions, generate the basic blocks containing these instructions, add the edges to these basic blocks based on the flow of execution, and finally wrapp these blocks into a function.

The final task is to insert calls to f at all the profile points. Figure 3 shows how this works. Suppose that we are profiling p in basic block b. In order to insert a call to the profiling function, we must divide b into two basic blocks. The first basic block, b_1, contains everything before p, plus the prologue for a function call, which includes saving the state of the registers and process status word (psw). We also create instructions to push on the stack the value being profiled and a reference to the distribution table for p. The second basic block, b_2, is entered when f returns. It contains an epilogue that restores the program state, profile point p, and the rest of basic block b.

It is worth noting that inserting calls to f at all the program points being profiled can increase the number of basic blocks and code size very quickly. Every instruction can potentially have multiple program points to profile. For example, suppose an instruction has two registers to compare. Then we would have to insert a call to f twice prior to this instruction. In Figure 3, this would result in b_2 splitting again at p.

4 Value Specialization

Value specialization is based on the distributions obtained from the value profiles. The basic idea is that if a variable often contains the same value, this value can be tested for, and then a section of code can be duplicated and optimized for this specific value. This technique retains the original code for other values, and creates a new path that should run faster for the common case.

Applying value specialization requires first choosing which program points to specialize. Then for each point a test is inserted to check if the variable holds the specialized value. This creates a branch in the control flow graph. One edge leads to the normal section of code, while the other leads to the cloned section, which will be optimized based on knowing the value of the specialized variable. Inserting code and cloning paths of execution alters the control flow graph and requires updating edge weights of the new control flow graph to retain consistency.

4.1 Determining Specialization Points

For each profiled point, we once again use the cost-benefit model to estimate whether or not it is worthwhile to specialize. However, now the distributions obtained from the value profiles are used to more accurately determine the cost and benefit of specialization. The immediate cost to applying specialization is the test of a variable for a specific value and the resulting branch. This cost will be incurred regardless of whether or not the specialized path is taken. Another cost to take into consideration is the increase in size of the code as a result of cloning the specialized path. We can get a better estimate of the benefit by using the distribution of the most common value to see how often we take the specialized path.

Simply computing the cost-benefit for each program point is insufficient for determining which points to specialize. The cost-benefit model tells us how much we expect to gain or lose from specialization, but this is only a local calculation independent of other program points which may be specialized. Recall that specialization requires cloning a path for optimizations. It is difficult to determine the effects of specialization of multiple variables if their cloned paths overlap. One solution to this problem is to avoid nested specialization. This can be done by choosing program points to specialize based on which ones appear first in the code, and eliminating all other program points which overlap. More complicated heuristics such as determining the density of specialization points and paths are also possible. In what follows we will assume that specialization points and cloned paths do not overlap. See [3] for more details on selecting specialization points.

4.2 Implementing Specialization

Value specialization requires inserting a test to check if a variable holds a certain value, and then creating a cloned path which is taken when the variable holds the common value. The actual optimizations occur later when PLTO examines the new control flow graph and realizes that the cloned path is entered only when the profiled variable holds a specific value. This allows various optimizations including constant folding and simplification or elimination of conditional statements based on this variable.

Consider the subgraph in Figure 4. On the left we have the original graph prior to specialization. Program point p contains a variable we want to specialize on. The labeled areas with an s are the paths of execution that have been marked to be relevant to specializing p. The areas with an n are normal sections not affected by specialization. In other words, knowing the value of a variable at p will only affect areas marked with s.

The graph on the right in Figure 4 shows how the control flow graph will be marked after specialization. Suppose that we specialize for register r at program point p when $r = 100$. We will have to add *test* to check the value of r at this point. In our example, we assume that we want to know the value just after p. The *test*

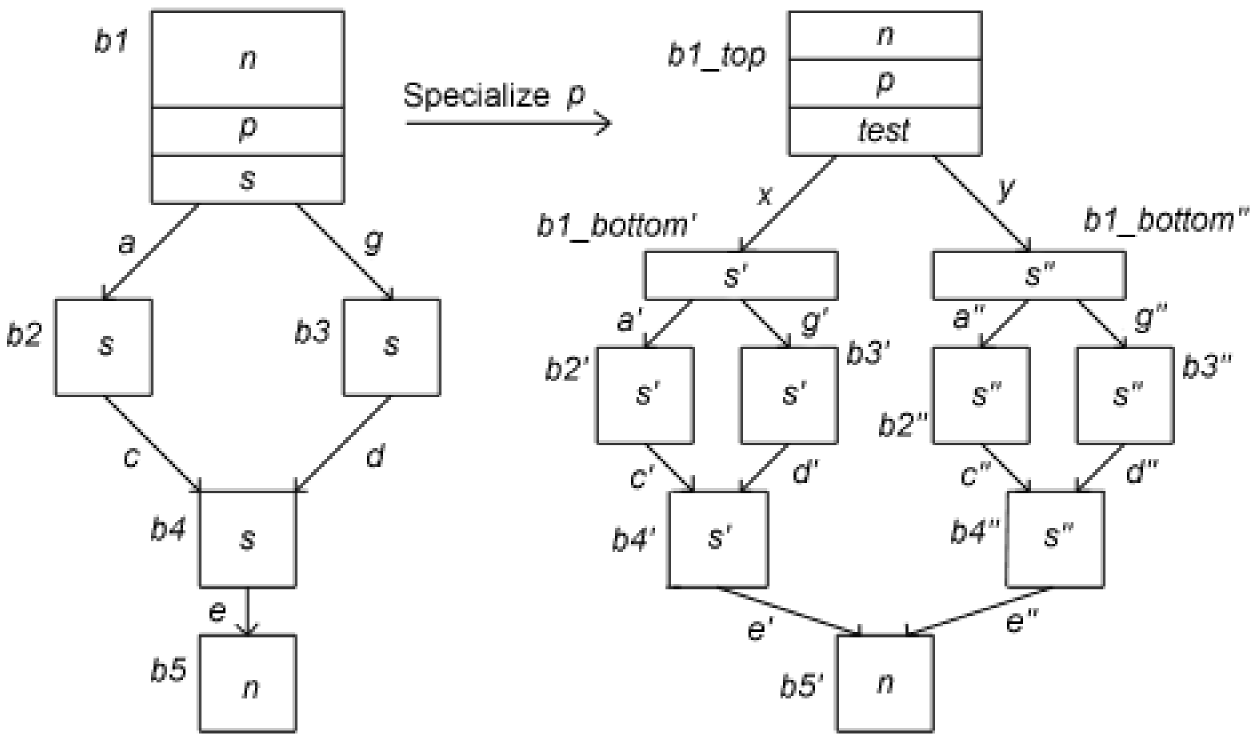

Figure 4: Specialization Example

code compares r to 100. If r does not equal 100, then execution flows as normal, and therefore will take the left path represented by x, with basic blocks marked by s' . If r equals 100, then we direct execution to the right. On the right side we have cloned the areas of the original graph marked with s. These nodes are now labeled with s''. Note that we needed to split basic block $b1$ in order to insert *test*. Then $b1\_bottom$ becomes the entry point to the specialized path of execution. Therefore we must clone all basic blocks which include $b1\_bottom$ and those marked with s. Eventually execution in the cloned or specialized area will return to the normal areas marked with n. Note that if we add an edge from $b4$ to $b1$, we would have a graph that looks like the body of a loop. In such a case this specialization would mean that each time the body is entered (which can be quite a lot if this is a hot spot), checking a variable for a specific value can allow the execution to enter an optimized version each time. This means that even if only a few instructions are eliminated on this optimized path, if it is entered thousands of times, the overall savings can become significant.

Apart from cloning the basic blocks in the specialized path of execution, we also have to clone the edges between them. There are three types of edges to consider [2]:

1. Specialized basic block to specialized basic block. An example of this is edge c ($b2 \rightarrow b4$). In such a case we create an analogous edge c'' in the cloned section, connecting the cloned block to the corresponding cloned block. Note that c' is the same edge as c.

2. Specialized basic block to normal basic block. An example of this is edge e. In such a case we create a similar edge, which goes from the cloned block to the normal block. Once again e'' is the added edge, and e' is the same edge as e.

3. Normal basic block to specialized basic block. An example of this is the entry point to the cloned section, which does not occur in the original graph because it occurs exactly where $b1$ is divided for the

[2]Actually there are four, but the case of an edge from a normal basic block $\rightarrow$ normal basic block clearly does not affect cloning.

test. As we can see, we must create two edges. x represents the original path, which is now a branch from the *test* going to the left which is the original code, while y is for the specialized path going to the cloned region. It is important to point out that this branching edge after *test* is the only interesting edge of this type. In fact, it represents the total inflow into the specialized region. The only other kind of edge which can come from a normal basic block into a cloned basic block is from a function return.

Figure 4 shows how to insert a specialization test, clone the relevant paths of execution, and then connect the cloned paths back to the original graph. This in itself will not make the code run any faster. In fact, initially it will make the code run slower, since now there is an added test and branch. However, once the control flow graph is modified, then PLTO will be able to apply optimizations because it will know the value of r from the inserted *test*. This should reduce the length of the cloned path by more than the cost of the *test* which will generally result in an overall faster running time. One thing to keep in mind, however, is that the total amount of code has increased as a result of the cloning. Normally this should not be much of an issue, but it may affect what gets stored in cache, which can have drastic effects on execution time (e.g. if the footprint of a loop or function becomes too large).

4.3 Updating Edge Weights

Value specialization results in adding new basic blocks and edges to the control flow graph. Therefore in order to maintain consistency we have to update the edge weights and block counts to show how execution is expected to flow in the new graph. This is important since various optimization by PLTO, such as code layout, rely on these numbers.

The main idea behind readjusting the edge weights is to realize that the total flow through the control flow graph before value specialization should be equivalent to the flow afterwards. The only changes that take place are the cloning of specialized paths, which will divert some of the flow from the original control flow graph to these cloned sections. For simplicity let *original* refer to the section of code before modifications, *cloned* refer to the section for the specialized path, and *residual* refer to what is left that is not diverted to the cloned section. The challenge is to figure how much of the flow is diverted, and specifically what paths it takes within the cloned sections. Then we can reduce this flow from the original sections to calculate the flow in the residual sections. Observe that for any edge, the sum of this edge's weight in the residual and cloned sections should equal the original edge weight. Similarly for any basic block, the node count in the original should equal the sum of the corresponding block counts in the residual and cloned sections. We can determine how much flow is diverted to the specialized path based on the distribution table on the tested variable. Then in a top-down fashion we can simulate the flow through the cloned section, and finally subtract this flow from the original section to get the counts for the residual code.

Consider the transformation in Figure 4 from the control flow graph on the left to that on the right. We start by knowing the basic block count for $b1$. For specialization $b1$ is divided, and the bottom part of this basic block is cloned and becomes $b1\_bottom'$ and $b1\_bottom''$. We know that the sum of the flows to these two blocks must equal the flow through $b1$; in other words $x + y$ is the block count for $b1$. The distribution table for the variable tested tells us what fraction of the flow will go to $b1\_bottom''$ (although not exactly, since distribution tables are periodically cleaned). From this we can get the block counts for both $b1\_bottom'$ and $b1\_bottom''$. It is important to notice that $b1\_bottom''$ is the only entry point into the cloned section, so it represents the total flow into this area[3].

In order to assign edge weights to the rest of the cloned section, we must first determine the frequency that each edge is traversed. Consider a'' and g'' which correspond to a branch. There are two cases to consider:

1. The specialized variable tells us nothing about the frequency of execution on these paths.
2. The specialized variable affects which path will be taken.

[3] Actually there can be other incoming edges into the cloned path. However, they will only be function return edges. This will not increase the flow since there will be a corresponding edge with an outflow for the function call edge.

In case 1, the cloned section alone does not help us predict the flow between these two paths. Then it is logical to assign the frequency of execution on these paths based on the original control flow graph. We can compute the frequency of a and b, and multiply these by the inflow to $b1\_bottom''$ to get the respective edge weights for a'' and b''.

For case 2, we will get a new frequency for each path based on the specialized value. This is likely to be 1.0 for one path, and 0.0 for the other if the variable we specialized on allows us to compute the result of a conditional branch. Then we can use the frequencies and the node count for $b1\_bottom''$ to compute the edge weights, and hence to compute the counts of the destination basic blocks.

After computing a'' and b'', we need to update a' and b'. Keep in mind that the total flow from the original control graph (left) for edges and nodes should equal the sum of the flows in the new control flow graph for the cloned regions and residual regions. This means that $a = a' + a''$ and $b = b' + b''$. Since both equations have one unknown, a' and b', they are easily solved assuming we compute the edges in the cloned section first. Since c'', d'', and e'' are all lone edges leaving a basic block, they have a frequency of 1.0, and thus have the edge weight equal to the node count of the basic block they are leaving. Note that whenever possible we compute basic block counts and edge weights in a top-down fashion.

The above method of assigning edge weights needs refinement to accurately represent the flow in cycles. A top-down assignment will have trouble determining edge weights for basic blocks in a loop by just looking at the source's block count. A future task is to develop an algorithm that can deal with cycles.

5 Example and Results

In this section we step through a concrete example that illustrates the potential effectiveness of value specialization, and how this relates to what PLTO is currently capable of. The starting point is the *Sample Program* in Figure 5. We will describe the control flow graphs produced by PLTO both before and after using value specialization and show that the specialized program runs 32% faster than the original program.

5.1 Program Flow

The program in Figure 5 is quite simple. There is one for loop that does a series of calculations based on a sequence of comparisons. At the top of the loop, we see that `spec_val` is assigned a value of either 100 or 0. In fact, 90% of the time `spec_val` is 100, and 10% of the time it is 0.

The important thing to notice in this program is that once the value of `spec_val` is known, the outcomes of the five comparisons can be computed in advance. Hence we should be able to specialize on `spec_val` and create an optimized version which can eliminate the conditional statements and just perform the calculations in the chosen branch. To get a better idea of the structure, look at the control flow graph of this program in Figure 6. This was obtained with gcc using the -O3 option, and then running PLTO with full optimizations (not including value specialization).

Basic block B19215 at the top of the diagram is the beginning of the for loop. The modulus operation at `(i % 10 > 0)` corresponds to the instructions starting with `idivl %eax <- %esi`. The test at instruction 141 determines whether `%eax` will hold 100 or 0 after instruction 145, which corresponds to `spec_val`. The staircase of basic blocks that follows represents the five conditional statements.

Instruction 146 is the beginning of the first conditional statement, comparing `spec_val` to 100. We see that the edge connecting to B19205 has a frequency of 1.0 because both 100 and 0 are not greater than 100, so the true branch is never taken. Note that `%ecx` corresponds to variable `x`.

In basic block B19206, `spec_val` is compared to 80. The outgoing edges show that the true branch (relative to the source code) is taken with a frequency of 0.9, while the false branch is taken with a frequency of 0.1. We can see that this structure repeats for the remaining three conditional statements. Basic block B19214 is the end of the for loop. Here the counter is incremented and then tested to see if the loop should be entered again.

From Figure 6 it is clear that we can specialize on `%eax`. However, where is the best place to do this? After instruction 145, `%eax` holds either 100 or 0, so checking before instruction 146 is possible.

```
int x = 0;
int main() {
  int i;
  int spec_val;
  for(i = 0; i < 300000000; i++) {
    if(i % 10 > 0) {
      spec_val = 100;
    else
      spec_val = 0;
    /* now we should be able to specialize on the value of spec_val */
    if(spec_val > 100)
      x += 1;
    else
      x += 2;
    if(spec_val > 80)
      x += 10;
    else
      x += 20;
    if(spec_val > 60)
      x += 100;
    else
      x += 200;
    if(spec_val > 40)
      x += 1000;
    else
      x += 2000;
    if(spec_val > 20)
      x += 10000;
    else
      x += 20000;
  }
}
```

Figure 5: *Sample Program.* This code has a series of conditional statements that depend on `spec_val`. By specializing on the value of `spec_val` just prior to where the series of conditional statements begins, it is possible to eliminate them because the result of each condition can be computed in advance.

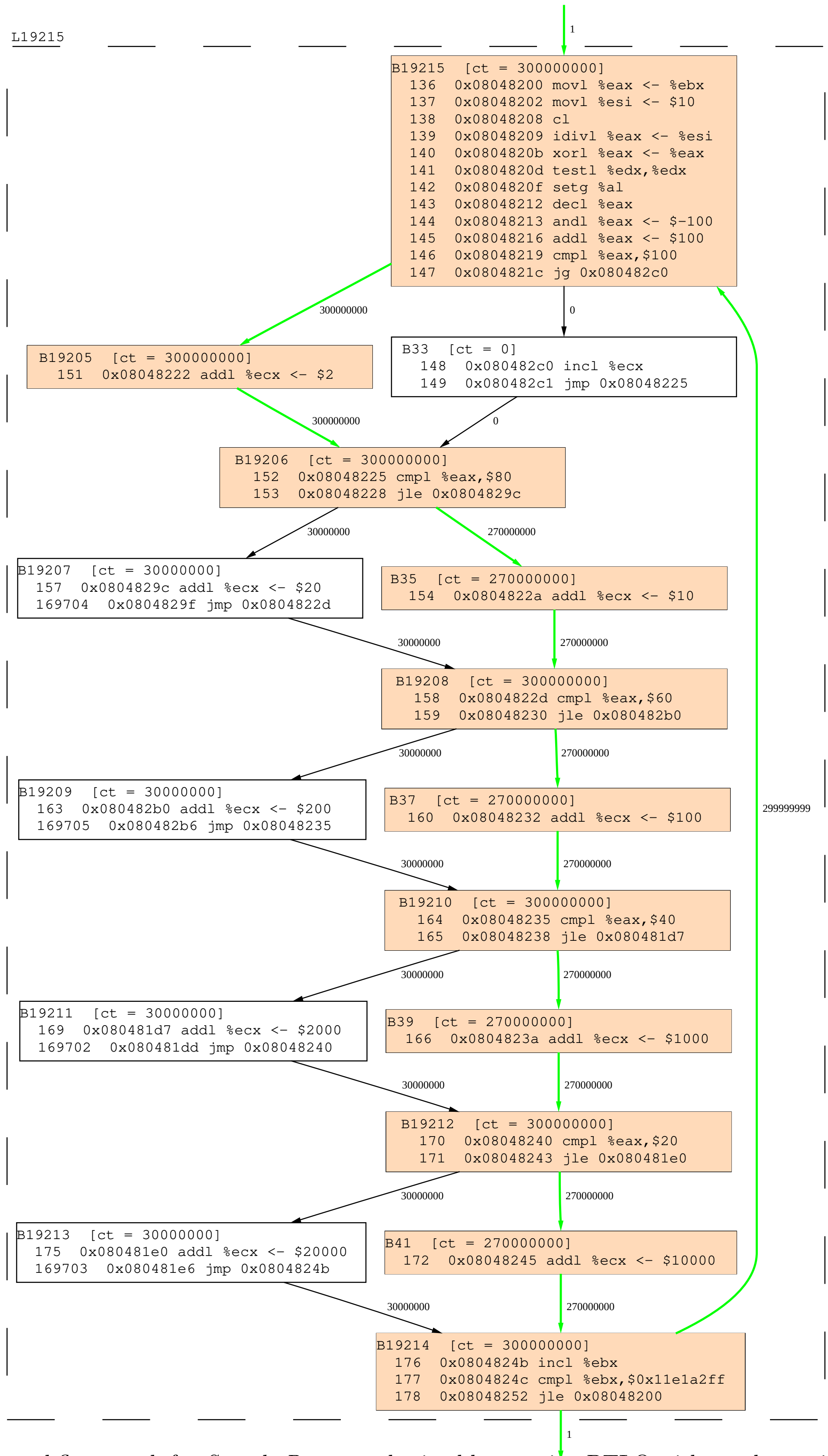

Figure 6: Control flow graph for *Sample Program* obtained by running PTLO with regular optimizations. It illustrates the flow of execution prior to value specialization.

Using the distributions from the value profiles as well as the estimates from the cost-benefit model, we can more easily pinpoint the most efficient place to insert the test. According to the cost-benefit model, the destination at instruction 143 can be specialized for the most benefit. When we look at the distribution table for this profile point, we see that `%eax` has a value of 1 90% of the time, and 0 the other %10. This means that after instruction 143 executes (which is a decrement), `%eax` will hold either 0 or -1. This is exactly the point where the test should be inserted. Since `%eax` will hold 0 90% of the time, we want to specialize for this value. Therefore we will test if `%eax` is not equal to 0, and branch to the normal code if true, and to the specialized section if false[4]. To complete the value specialization at this point, we must clone the rest of basic block B19215, as well as all of the other basic blocks in the for loop.

5.2 Specialized Program

Figure 7 shows the new control flow graph with value specialization implemented as described. PLTO is not capable of automatically applying value specialization at this point, so it was necessary to determine by hand where to insert the test and which basic blocks to clone for the specialized path. This only required modifying the CFG to include the cloned path for the value specialized; PLTO did the specialization to optimize this path. Some of the edge weights are inaccurate since the method of assigning them as described earlier has not yet been implemented.

In Figure 7 we can see exactly how value specialization altered the control flow graph. First look at the bottom of basic block B19215. Right after instruction 143 we see the specialization test. The left branch goes to the standard path where we have a new basic block (B25761) that contains the rest of B19215 from the original control flow graph. The right branch goes to the specialized path which consists of only five basic blocks, the last of which is used to test the condition of the for loop.

Initially the specialized branch on the right starting at the bottom of basic block B19215 in Figure 7 was identical to the left branch since we only cloned that path. Let us trace through the optimizations PLTO applied after knowing the value of `%eax` from the specialization test. We will refer to the right branch in order to describe the changes that transformed it into the left branch. Instructions 144 and 145 were removed because knowing the value of `%eax` meant that the outcome of both instructions could be computed in advance. The conditional branch at instructions 146 and 147 were unnecessary since the outcome can also be computed if `%eax` is known. This resulted in eliminating basic block B33 completely. Basic block B19206 was also removed since the outcome of the conditional branch can be computed in advance as well. The remaining three branches were eliminated for the same reasons. All that is left is the single path of execution taken when the values of all the conditionals are known.

It should be pointed out that PLTO has not completely optimized the right path. Although several instructions have been eliminated simplifying the path to only five instructions within the body of the loop, it is clear upon inspection that those five instructions can simplified to just one! Each instruction adds a constant value to `%ecx`. Therefore these constant values can be summed up in advance and then all added directly to `%ecx` in one instruction. We are still looking into why PLTO failed to optimize this path fully, but believe that it may have to do with the order in which the various optimizations are executed since some only become possible after others have occurred (e.g. the basic blocks cannot be merged if compare and jump instructions have not been removed).

The program from Figure 6 which was fully optimized by PLTO (without value specialization) obtained a running time of 30.39 seconds for each of three trials. The program from Figure 7 which included value specialization had a running time of 20.38 seconds for each of three trials. This is an improvement of

5.3 Experimental Results

Figure 7 shows that the optimized path has approximately a dozen less instructions executed than the normal path. Actually more than a dozen instructions were eliminated, but many were not directly in the path of

[4]We specialize on the false branch since since fall-throughs generally execute more quickly on the Pentium.

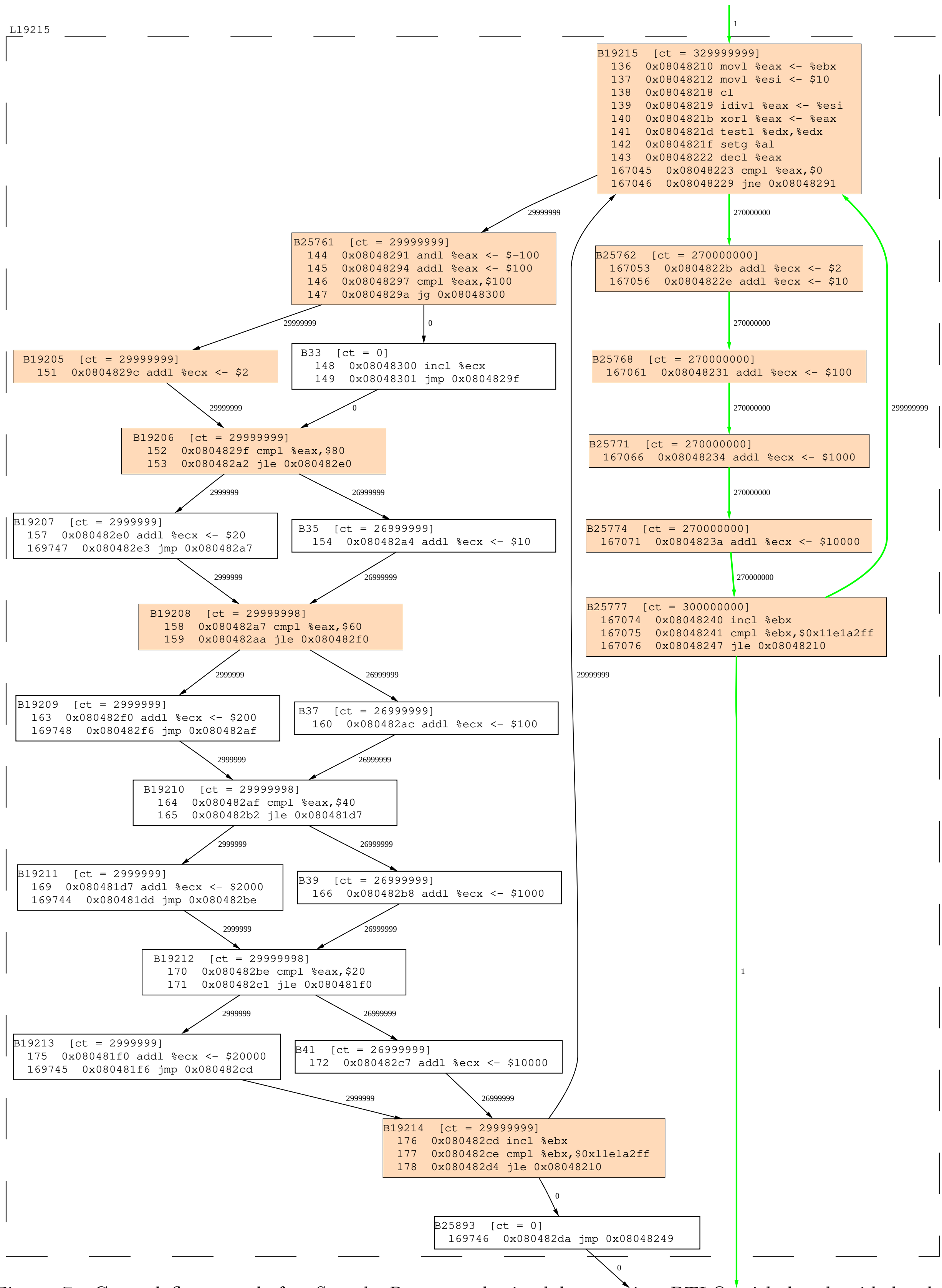

Figure 7: Control flow graph for *Sample Program* obtained by running PTLO with hand-guided value specialization. We see the normal path of execution on the left, and the specialized (and much shorter) path on the right.

execution (i.e. those in the bodies of the branches not taken). Not surprisingly timing tests comparing the programs which correspond to the CFGs just described show drastic differences in performance.

We ran the programs in Figure 6 and Figure 7 and measured their execution times. The tests were run on a 600MHz Pentium III. Times were measured using the Unix time command. Each test was run three times to ensure accuracy. The results were as follows:

Before Specialization	(Figure 6 program)	30.39 seconds
After Specialization	(Figure 7 program)	20.38 seconds

This is an improvement of over 32%, which is expected considering that 90% of the time the optimized path is entered resulting in much fewer total instructions executed.

6 Related Work

Value specialization as a technique for improving the performance of programs has been explored for quite a while. Some of the earliest work in this area was done by Calder *et al.* [1,2]. A specific implementation of value specialization for the Alpha architecture was later done at the University of Arizona by Watterson, Debray, and Muth [3, 4]. This work showed that value specialization for the Alpha resulted in non-trivial gains in performance for certain programs.

Implementing value specialization requires an infrastructure that allows a relocatable object file to be disassembled and manipulated in order to output a functionally equivalent binary. This is what PLTO provides for the Pentium. The more general problems of selectively choosing which areas to profile, how to instrument the binary to obtain these value profiles, and the process of actually modifying the input binary in order to apply value specialization have been researched across several projects. In particular, ALTO (Alpha Link-time Optimizer) has dealt with these issues in the context of a RISC architecture. Both selective value profiling [4] and the implementation of value specialization [3] are closely related to the work with PLTO. Since the Pentium is a SISC architecture, many variables are stored on the stack rather than registers, which has been an additional challenge for PLTO because tracking variables on the stack becomes necessary for value profiling.

7 Summary

Value specialization has been shown to be an effective way to optimize programs on non-Pentium architectures. This paper describes an implementation of value specialization for the IA32 (Pentium) architecture.

Many pieces are necessary in order for value specialization to take place. Initially PLTO provided the infrastructure to disassemble object files, manipulate them, and output new binaries that are functionally equivalent. What is required for value specialization is a method of selecting program points to profile, instrumenting the input binary to obtain value profiles, and then using these distribution tables to actually specialize the code. Currently we have a benefit model that roughly estimates the number of instructions that can be saved if a variable's value is known at a given program point. This is the key component to constructing a cost-benefit model which is used for selecting program points to profile and later specialize. We are able to instrument input binaries and obtain value profiles for all general purpose registers and stack locations. We have the building blocks for automating the value specialization. This includes functionality for cloning paths of execution and inserting test code to branch for specialization.

The example in chapter 5 shows what is currently working. Here we directed the value profiler to select all variables in the main function. We used the distribution tables and the benefit model to determine the best point to profile. Then we directed PLTO where to insert the specialization test and which basic blocks to clone for the specialized path. The rest of the optimizations were done automatically by PLTO. Our future work will consist of automating the rest of the value specialization process.

The *Sample Program* in Figure 5 showed a decrease in running time of over 32%. This, combined with past work done on the Alpha, indicates that value specialization specifically for the Pentium can lead to significant performance improvements. This is not surprising considering the general structure of programs. It is common that a variable takes on a value almost all the time, and therefore it makes sense to specialize these cases. As an example, many programs contain error checks and other safety tests that guard sections of code that are rarely executed. These tests may occur multiple times throughout a section of code; they could be reduced to just one test through value specialization.

Although we believe that value specialization in general can improve the performance of most programs, we are particularly interested in applying value specialization to MPI[5] programs which are used for parallel scientific computing (and hence take a long time to run so almost any performance gain can be significant). MPI programs generally have a structure that should lead to significant improvements in performance through value specialization. This is because the communication between machines is done through repeated function calls that often have arguments which are expected to be runtime constants. Recall that compile-time constants can be optimized through a compiler, but runtime constants can only be noted and optimized through value specialization.

8 Acknowledgements

I would like to thank my advisor, Greg Andrews, for his guidance and time throughout this project and up through the final revisions of this thesis. I am also appreciative of the help and support I received from Ben Schwarz, Matt Legendre, and Saumya Debray, with PLTO and other aspects of this work. This research was supported by the National Science Foundation through grants ACR-9720738 and CCR-0113633.

9 References

1. B. Calder, P. Feller, and A. Eustace, "Value Profiling and Optimization", *Journal of Instruction-Level Parallelism* 1 (1999), 1-6.

2. B. Calder, P. Feller, and A. Eustace, "Value Profiling", *Proc. 30th International Symposium on Microarchitecture*, Dec. 1997, pp. 259-269.

3. S. Debray, R. Muth, S, Watterson, "Code Specialization based on Value Profiles", *Proc. 7th. International Static Analysis Symposium (SAS 2000)*, June 2000. *Springer LNCS vol. 1824*, pp. 340-359.

4. S. Debray, S. Watterson, "Goal-Directed Value Profiling", *Proc. Compiler Construction 2001 (CC 2001)*, April 2001.

5. B. Schwarz, S. Debray, G. Andrews, "PLTO: A Link-Time Optimizer for the Intel IA-32 Architecture", *Proc. 2001 Workshop on Binary Rewriting* (WBT-2001), Sept. 2001.

6. B. Schwarz, "Post-Linktime Optimization on the Intel IA-32 Architecture", Honors Thesis, May 2002.

[5]MPI is the Message Passing Interface library.