Using Data Values to Aid Branch-Prediction

Zachary S. Smith

Project Report
Submitted in Partial Fulfillment
of the
Requirements for the MS ECE Degree

Advisor
Prof. James E. Smith

Dept. of Electrical and Computer Engineering
University of Wisconsin - Madison

December 17, 1998

Abstract

My research was a concerned with gaining an understanding of some program constructs which create "hard to predict" branches, and figuring out how to use data-value information to predict branches. I started with a low-level investigation of the source-level constructs corresponding to the static branches with the most misses. Next, I investigated the predictability of register values leading up to these branches. I constructed several predictors for some particular branches in an effort to improve accuracy by utilizing available program information.

Data-values have the potential to help branch prediction accuracies a great deal — even more than 20% accuracy improvement over gshare for particular branches — though work remains to be done to determine an implementable, efficient method for exploiting this potential.

This research was meant to provide some basic insight into the problem so that further, more complete, studies could be done. The purpose of my research was not to be a comprehensive study, but instead a time-effective investigation of possible heuristics to aid branch-prediction. Finally, I contributed to a study of an implementable scheme which used data-values to aid branch prediction.

Contents

Section 1

Introduction

High branch prediction accuracy is desired in modern processors because it reduces pipeline stalls due to branches and exposes more instruction-level parallelism for superscalar, multi-scalar, or trace processors. Most conventional branch predictors use only branch history and branch PC information to index into a predictor table. These predictors work very well for the common case, but as Amdahl warned, the uncommon case then limits performance.

To accurately predict the remaining "hard to predict" branches, we must go beyond local or global branch history. There are many other places to look: register values, patterns, structural information, etc. If we can figure out what information is relevant to certain classes of "hard to predict" branches, then we can design predictors which recognize and exploit this information.

My research was focussed on two things:

1. to gain an understanding of some program constructs which create these "hard to predict" branches, and

2. to propose mechanisms that use data-value information to predict branches.

This is not a comprehensive study. My research, and this report, was meant to provide a first step so that further more complete studies could be done. One such study, a paper submitted to the 26^{th} International Symposium on Computer Architecture, is included in Appendix A. Since this was a project, the scope of investigation was limited by available time. I generally moved from small experiment to small experiment when I thought that enough insight had been gained to move forward.

Section 2

"Bad" Branch Inspection

A first step in understanding how to build a better branch predictor is to understand why conventional branch predictors behave poorly on some branches. In an effort to get a handle on these "hard-to-predict" branches, I examined the source code constructs corresponding to the 25 branches which contributed the most branch misses to the program run. The programs examined were chosen from the SPEC95 suite.

2.1 Methodology

Simulations were done on a version of the simple, in-order SimpleScalar[1] simulator which had been modified to do both conventional branch prediction and branch profiling. The conventional branch predictor used was gshare with a 12-bit history, giving 4k entries. At the end of a run, each static branch would have an associated count of the number of misses it contributed to the total misses for the run.

After the run completed, each of the branches in the top 25 misses was traced back to assembly-level and source-level code. In addition, graphs drawn to display the prediction accuracy for these top 25 mispredicted branches and a cumulative miss distribution which shows what fraction of the total run misses come from what fraction of the static branches.

2.2 Distribution of Misses

The distribution of misses as a function of number of misses per branch is important because it shows whether most of the total program misses come from few branches which miss many times, or most of the total program misses come from a larger number of branches which miss fewer times. If most misses come from fewer branches which miss many times, then it suggests that there are only certain cases which conventional branch-predictors do not handle well.

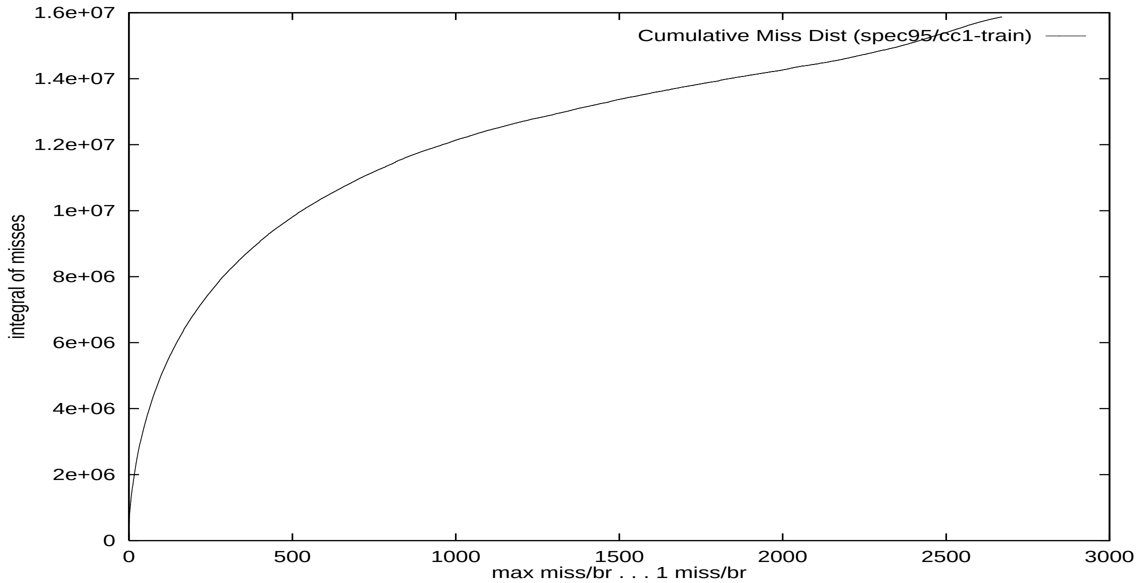
The following graphs show the cumulative distribution of pairs:
(X misses per branch, total number of misses from branches with X misses),

[1]Douglas C. Burger, Todd M. Austin, "The SimpleScalar Tool Set, Version 2.0," Univ. of Wisconsin - Madison Comp. Sci. Dept Technical Report #1342, June 1997.

sorted in decreasing order by X. The value on the X-axis is merely the rank of the data-pairs in sorted order. For example, the contribution of the right-most element is the sum of all the misses which come from branches which miss once.

The naming convention is <program>-<input set>. For example, "cc-train" is the cc benchmark training input.

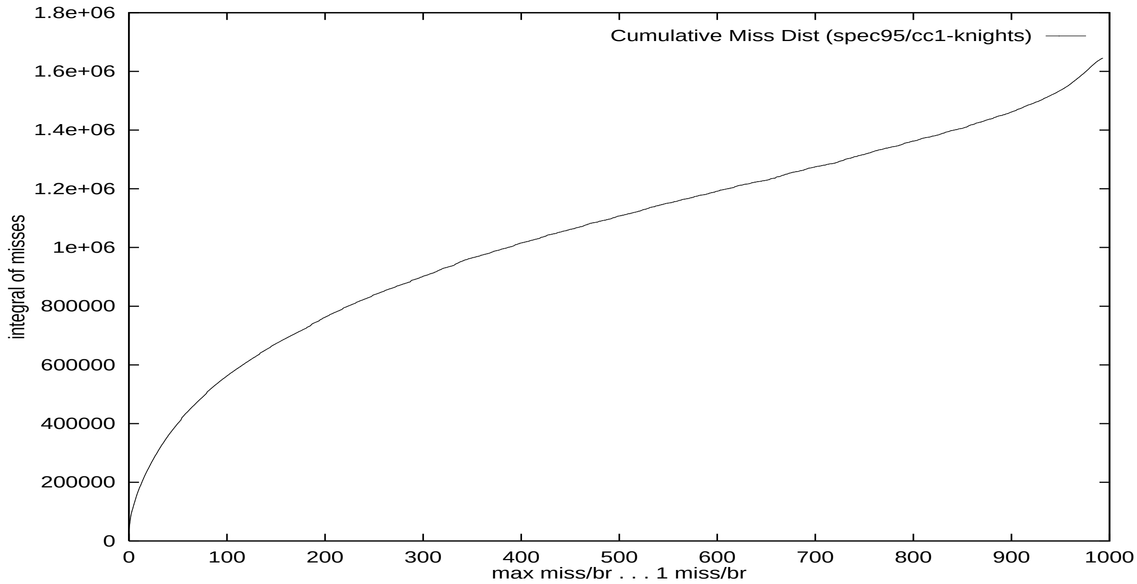
2.2.1 cc-train



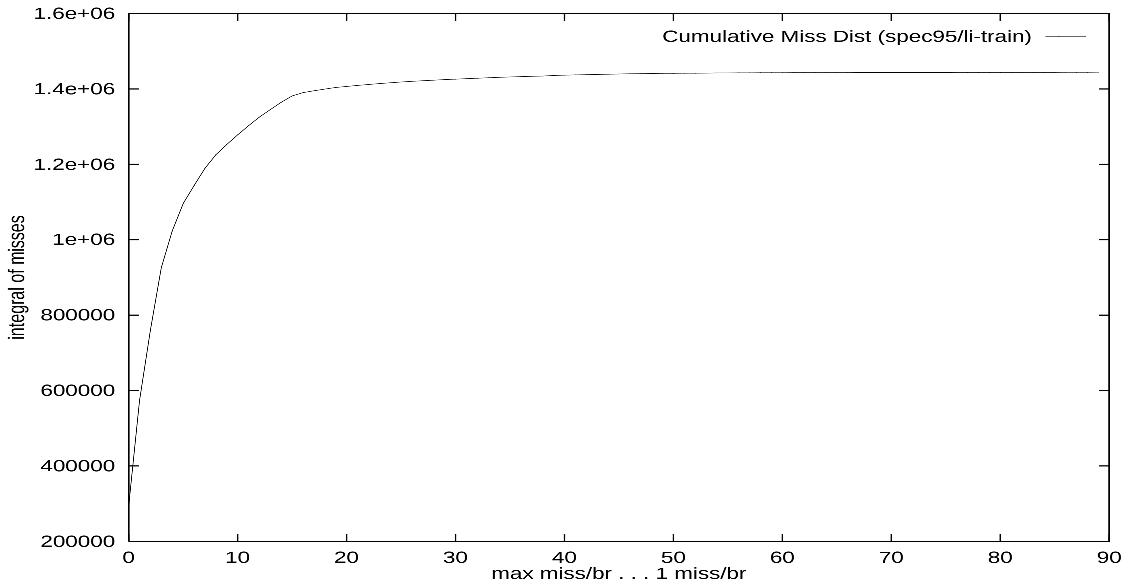
A curve which is initially steep and then flattens indicates that most of the misses come from relatively few static branches. In this case, we can see that approximately 60% (10M on Y-axis) of the misses come from branches whose number of misses is in the top 16% (500 on X-axis).

The "tail" on the right-hand end indicates how many misses come from branches which have few misses. This number is large when there are lots of branches which are missed only a small number of times. As the number of total static branches encountered increases, this number increases because the total misses due to predictor training increases.

2.2.2 cc-knights

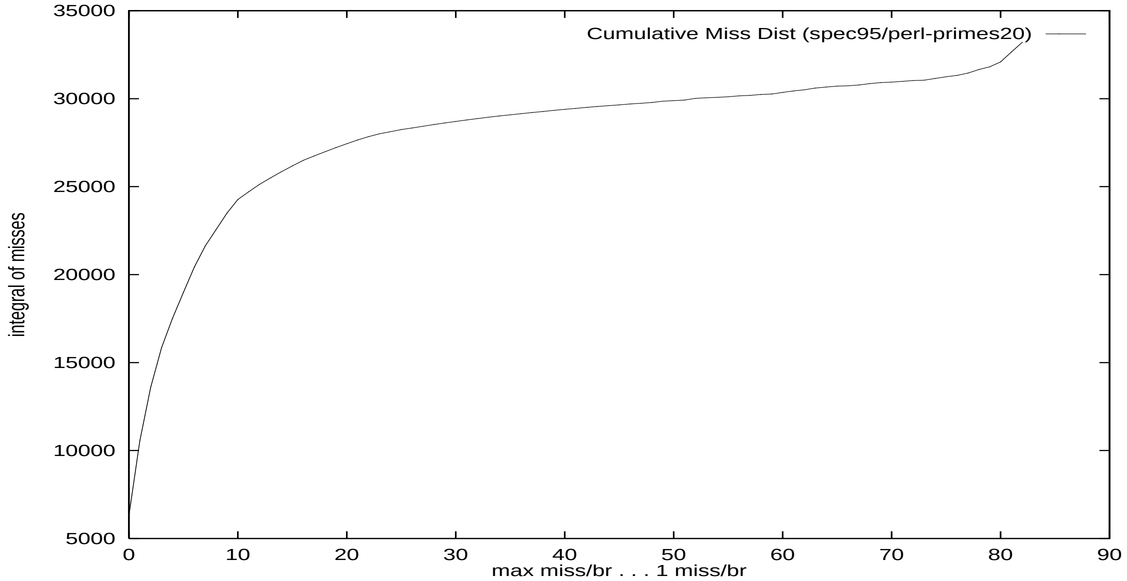


2.2.3 li-train

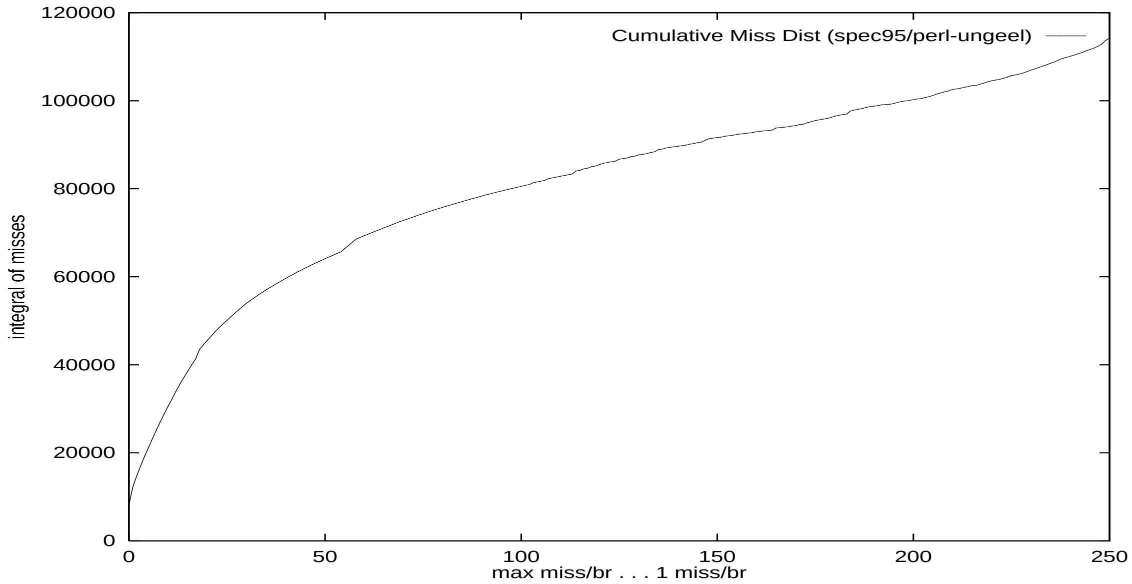


The lisp interpreter's misses are highly concentrated in just a few static branches: approximately 60% of the misses come from only 8% of the branches.

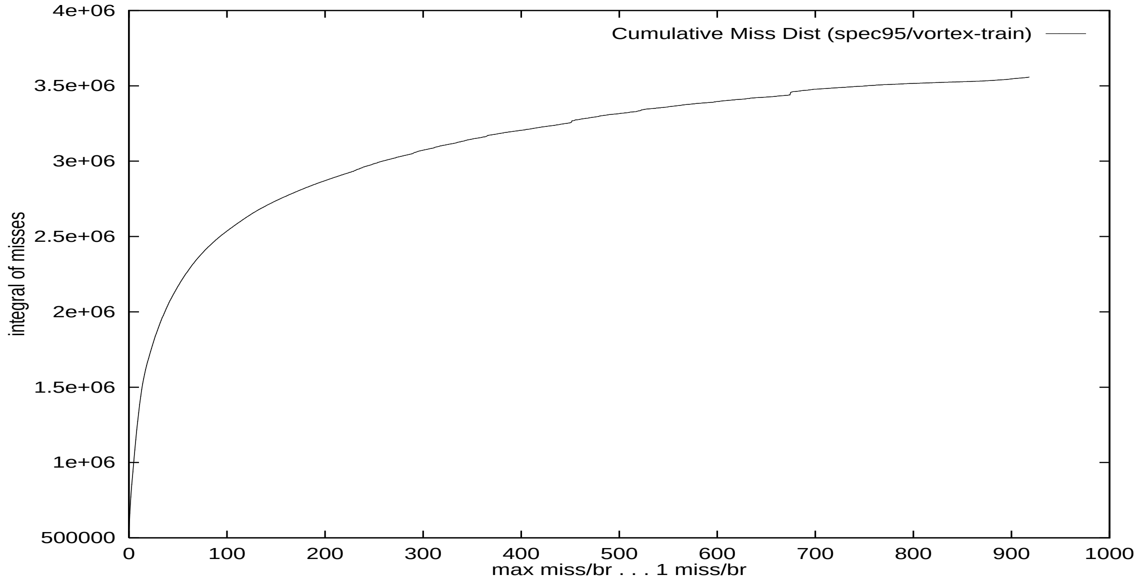
2.2.4 perl-primes20



2.2.5 perl-ungeek



2.2.6 vortex-train



2.2.7 Summary

As we can see from these graphs, the majority of the misses come from a relatively small number of static branches. The misses are *not* evenly distributed over the branches in a program.

From observing the graph "tails", we can see that the impact of the training misses is relatively small. Said another way, the sum of the misses from branches with few misses is small compared to the sum of the misses from branches with many misses, the "bad" branches.

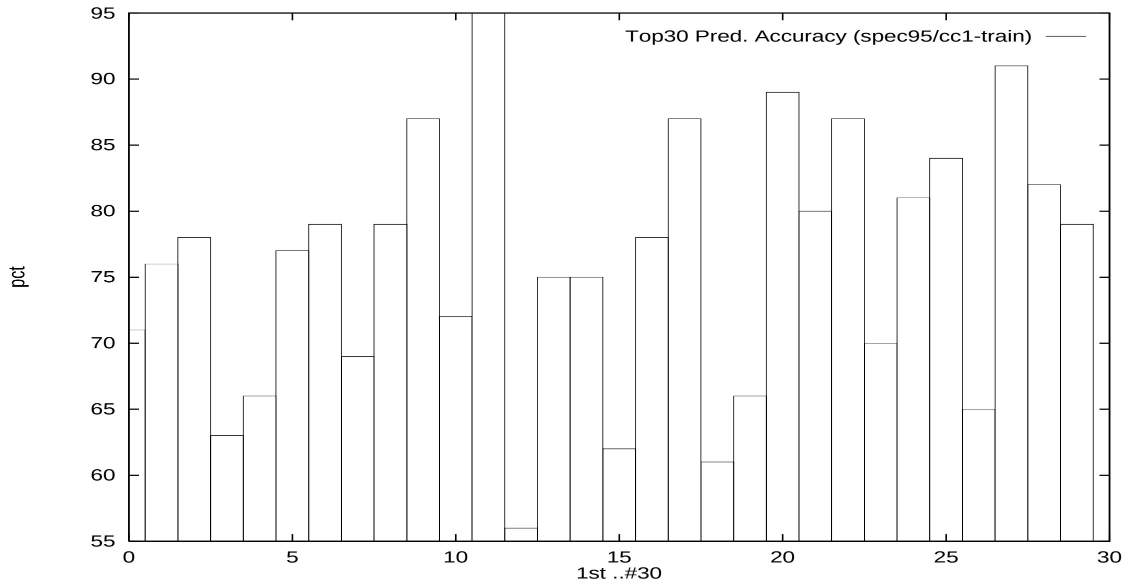
2.3 Prediction Accuracy for "Bad" Branches

Another important issue to address is: *how predictable are these "bad" branches?* Do the branches with the most misses have a low predictability[2] which causes the misses? Or do they have more-or-less average predictability but the branch is executed so many times that n * *mispredict rate* is very large?

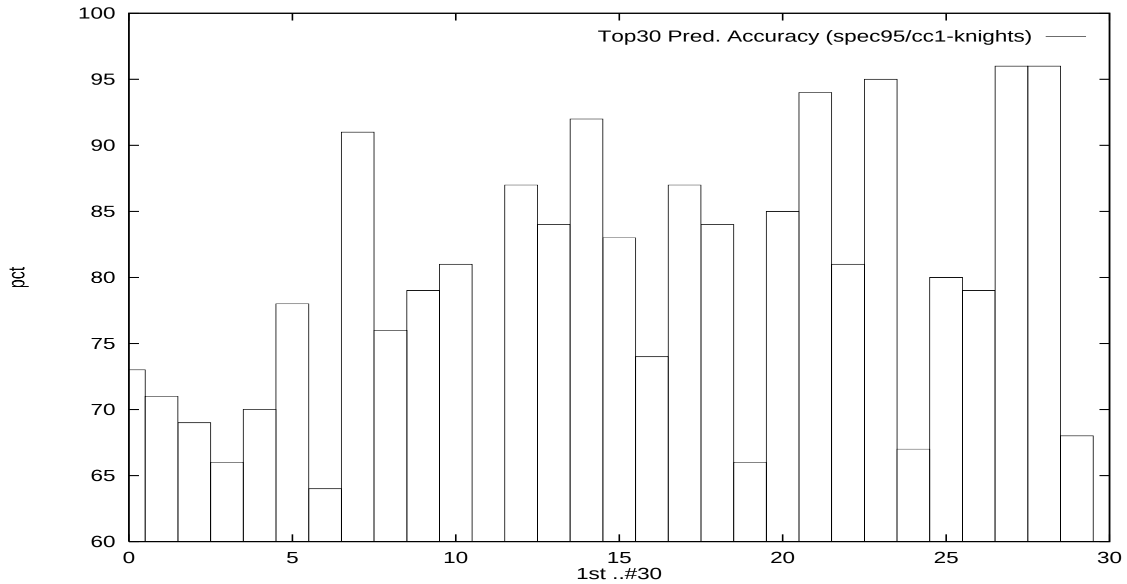
To give some insight into this question, here are plots of the predictability of the 30 branches which have the most misses.

[2]. . . in the context of the predictor: gshare.

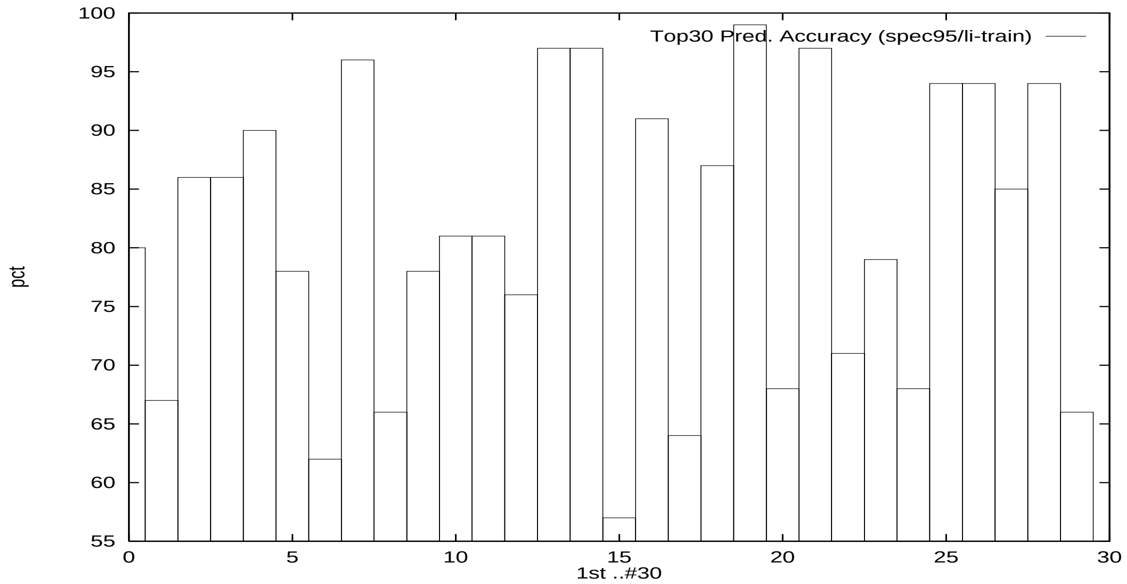
2.3.1 cc-train



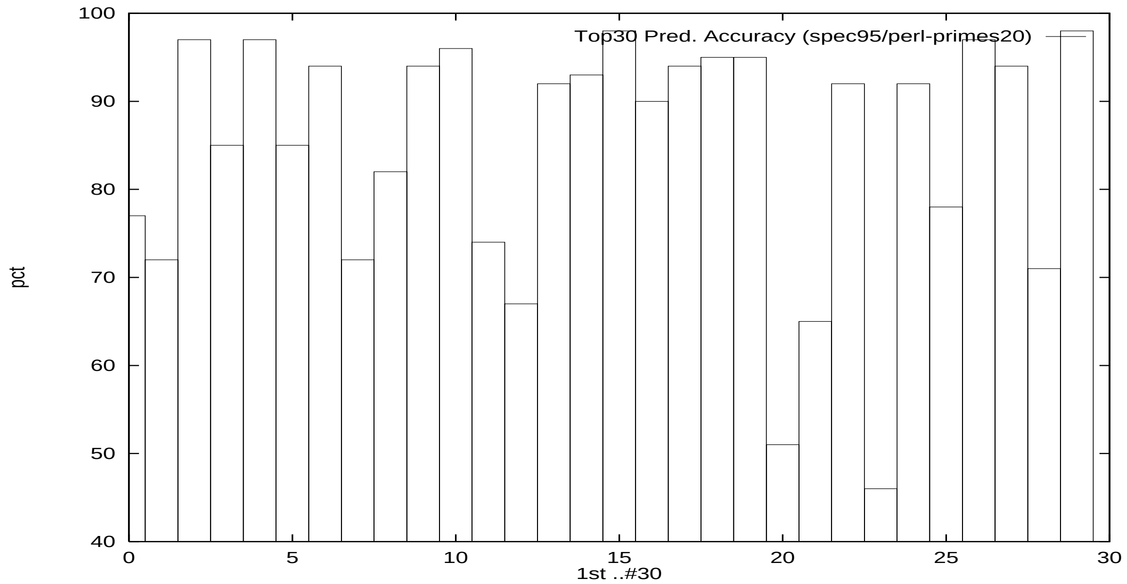
2.3.2 cc-knights



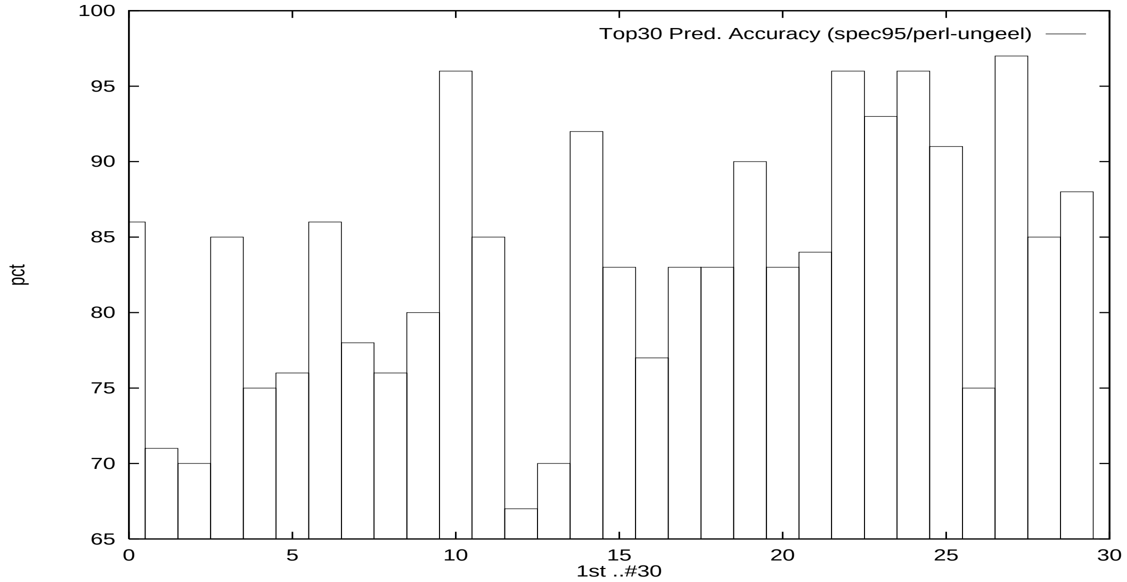
2.3.3 li-train



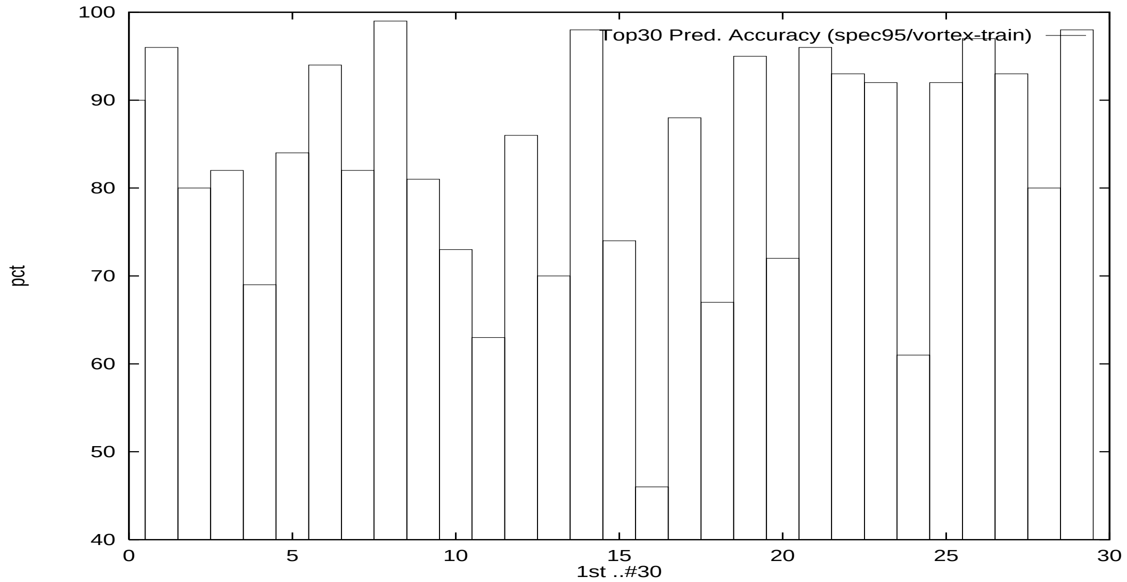
2.3.4 perl-primes20



2.3.5 perl-ungeek



2.3.6 vortex-train



2.3.7 Summary

If it were the case that all the branches which produce lots of misses were fairly unpredictable, all of the points in the graph would have been below 75% or so. We can see that some of these branches have high predictability (95% or so), and some have low predictability (50% to 75%).

There is no strong correlation between branch predictability and how many misses it produces. Some of these branches are more predictable but occur more often, and some are less predictable but occur less often.

2.4 Program/Data Structures

The next step is to understand why some of these branches produce lots of misses. I focus on why they are unpredictable: what are ways to improve the prediction accuracy for these currently poorly-predicted branches? The other method for reducing the number of misses is to reduce the number of times these poorly-predicted branches are executed, but that is out of the scope of my research at this point as it might be better addressed by compiler research.

My purpose at this point was to find common structures, or general cases, which caused unpredictability. It was not to exhaustively explain every specific case.

I will go into detail for cc-train, and just include an explanation of the other programs I examined. Full raw data for cc-train, cc-knights, and li-train is included at the end of the report in the appendices.

2.4.1 cc-train

After examining the misses for cc-train, it was clear that the majority were *similar in spirit* to the following cases. In cc-train, the first 4 worst branches stem from a common piece of code in cse.c, the code for common subexpression elimination:

```
1  for ( hash=0 ; hash < NBUCKETS ; hash++ )
2       for ( p = table[hash] ; p ; p = p->next )
3            {
4            if ( p->exp->code != REG ) continue;
5            . . misc code . .
6            }
```

cc-train-#1

Let's call the branch with the most misses for cc-train cc-train-#1. It corresponds to line 4 of the above code segment.

The normal gshare predictor gets 71% for this test. This will be called the "buckets-data" case because it is testing data which is indirect off a bucket linked list element pointer.

cc-train-#2

```
1  for ( hash=0 ; hash < NBUCKETS ; hash++ )
2       for ( p = table[hash] ; p ; p = p->next )
3            {
4            if ( p->exp->code != REG ) continue;
5            . . misc code . .
6            }
```

The normal gshare predictor gets 76% for this test. This will be called the "buckets" case because it is testing the bucket linked list element.

cc-train-#3

```
for ( hash=0 ; hash < NBUCKETS ; hash++ )
     for ( p = table[hash] ; p ; p = p->next )
          {
          if ( p->exp->code != REG ) continue;
          . . misc code . .
          }
```

This corresponds to the same source construct as cc-train-#2, but the compiler makes a special case for the first iteration.

The normal gshare predictor gets 78% for this test. This is again the "buckets" case.

cc-train-#4

cc-train-#4 has the same structure as #1, #2, and #3, but it occurs at a different point in the source code:

```
for ( hash=0 ; hash < NBUCKETS ; hash++ )
     for ( p = table[hash] ; p ; p = p->next )
          {
          if ( p->exp->code != REG ) continue;
          . . misc code . .
          }
```

The normal gshare predictor gets 63% for this test. This is again the "buckets-data" case.

The rest of the source code extracts are included at the end of this report.

Summary of cc-train

The majority of the misses are due to branches which test the contents of the table[] data structure. This structure is mostly constant, but changes slowly as a small number (usually less than 4) of elements are added or deleted from the linked lists which hang off of table[].

If table[i] has 3 elements hanging off of it, the branch pattern for the "p" (cc-train-#2) test will be T T T N. If one of these is deleted, the new pattern will be T T N. Since gshare uses global history, subsequent branches (ie, while processing table[i+1]), will not get the same ("correct") bimodal predictor as before, because the history has now changed. Even worse, as the data changes ("p"), the number of history bits which are affected changed because now table[i] will contribute only 3 history bits instead of 4, so the mapping has changed making correct prediction unlikely. I call this the "variable-length history problem".

This history problem affects cc-train-#1, the case in which p->exp->code is tested, as well. The data hanging off each linked list element is relatively constant as long as the element is not deleted (or replaced). It should be possible to use the data values to help solve this variable-length history problem.

A related problem is that if the code in the body of the loop contains many branches unrelated to the data tested, the relevant history will be flushed out of the global history buffer by the next iteration.

Another case which happens in cc-train is a case in which a very commonly called library routine (from memset.c) is called from many many different places in the program. The global branch history is "diluted" because it is called from so many places, whose histories do not correlated. This problem also occurs in some recursive program structures. I will call this the "history dilution problem". It should be possible to help these branches by using the caller ID or data information. I later addressed this problem by using data-values from registers to form a local context. I never tried using caller ID information.

2.4.2 cc-knights

"Knights" is a short C program I wrote a few years ago to solve a modified knight's tour problem[3]. It is recursion-intensive and the "meat" of most functions are in the body of the "return" statement. This is again a test of the C compiler.

As would be expected, the "buckets" and "buckets-data" cases are prevalent in the top 25 "bad" branches. The rest of the cases I believe behave badly because of the "history-dilution problem". The first worst branch is in a routine which recurses on a linear list, thus polluting the history with a variable number of bits depending on the data in the list. Another variation of the variable-history length problem is apparent in large compound tests: the previous branches may not have any correlation to the current branch.

2.4.3 li-train

Most of the top 25 "bad" branches for the lisp interpreter are of the form "if (p->value == CONST)". In the rest of the cases, the tested variable is the induction variable in a linked-list traversal. I believe these branches are unpredictable because nodes are commonly inserted and removed from the lists, which dilutes the global history. If the data on a certain node changes slowly, then it should be possible to use data information (ie, "p") to predict what the test of "p->value" should yield.

2.4.4 perl-primes20

Most of the "bad" branches in perl-primes20 are in library routines and, not surprisingly, have the "history-dilution" problem due to many callers.

[3]A knight must move on a restricted chess board and end up at a certain position

2.4.5 vortex-train

Like, perl, lots of vortex's "bad" branches are from library routines. This is interesting because most are of the form "memcpy(dest, src, CONST)", but they are called from so many places that the history is diluted. We should be able to predict these all the time, because the length (and thus pattern) is constant for each caller!

The rest of the "bad" branches for vortex-train test various attributes of database objects or are related to whether or not a certain key was found. The attributes should be correlated to the address of the object they reside in. This is very similar to the "p->exp->code" test in cc-train. In that case, the data ("code") was relatively constant with respect to the node ("p", or "exp"). In this case, the data is relatively constant with respect to the address of the object address.

2.5 Summary Remarks

I have identified several effects which produce poorly-predicted branches.

1. "Popular subroutines" have a hard time getting useful history. Possible solution: keep track of caller, or carefully choose data-values to use.

2. Program or data structures which produce a variable number of history bits as the data changes — variable-length history problem. Possible solution: use data values or structural information to keep the predictor "better synchronized" with data.

3. Control locality. The branches within the global history length may not have correlation with the current branch, or the relevant history might be too far away. Each branch has some previous code which is more relevant, so either focus on that history, or figure out what is the globally "important" history.

Although this is not a comprehensive description of all the constructs, we now have an idea of some heuristics which might be used to aid branch prediction.

Section 3

Data — Branch Correlation

As a next step in investigating possible methods for branch prediction improvement, I tried to correlate some of the input data to the branch outcome. In effect, instead of the lookup key being the low PC bits XOR'd with the global history as in gshare, the lookup key is now a register value known some time before the time the branch is evaluated. The base predictor is still a bimodal predictor.

The motivation for doing this was to measure the correlation between available data values (e.g., "p", "p->exp", or "p->exp->code") and the branch outcome. The "p->exp->code" case should have perfect correlation to the branch outcome, because it is the tested value.

This experiment was only conducted for the cc-train "p->exp->code" related cases as an intermediate step before moving to the full value-prediction experiments which follow in the next sections. It is expected that addresses "closer" to the value tested should be more highly correlated to the branch outcome. That is, "exp" should be more closely correlated to the branch outcome than "p".

3.1 Methodology

I modified the simulator so that when the PC for cc-train-#1 was encountered, in addition to the gshare lookup and update, my special "correlated predictor" code was run. The outcome of the "correlated predictor" (CP) was then tested for accuracy against the actual outcome. At the end of the run, the prediction accuracy for both gshare and the CP are available for comparison.

In addition, a third value was generated: the accuracy of a predictor which was correct if either CP or gshare was correct. This value will show if the CP predicts correctly when gshare does not.

The simulations were run on the "p->exp->code == CONST" case (cc-train-#1), for the first 200,000 executions of the static branch (to limit execution time).

3.2 Results

(accuracy %)	p	exp	code
gshare	77.7	77.7	77.7
CP	64.2	99.9	99.998
gshare + CP	91.7	99.97	99.998

For example, when the branch outcome was predicted based on the value of "p", the accuracy was 64.2%. When it was predicted on "exp", the accuracy was 99.9%, and when it was predicted based on "code", the accuracy was 100% minus one training miss per unique data value.

3.3 Summary

In this data structure, data within the nodes rarely changes. Changes are mostly nodes being added or deleted. This is why the accuracy of correlation on "exp" is so high. Correlating on "code" is equivalent to just evaluating the branch. From the last row, we can see that the correlated predictor does indeed predict correctly for some cases that gshare does not.

Instead of dwelling on this, I chose to move on to doing full value prediction to figure out how predictable branch inputs are, and if the branch outcome can be predicted using the value-prediction mechanism.

Section 4

Value Prediction

Although data correlation was somewhat interesting, to get any real improvement, we need to know the data value sooner. If data values could be predicted (for example, the "exp" in "p->**exp**->code"), then we could either continue evaluation or simply use a correlation of the predicted value to the branch outcome. Either way, we need some sort of idea *how predictable tested values are.*

4.1 Simple Value Prediction

4.1.1 Methodology

A context-based value predictor was used to predict values near the branch test value. Context-based value predictors use the last n values produced[1] by an instruction as the key to a base predictor, which in this case was *bimodal-ish* in that it would only change to a new value on the 2^{nd} miss.

In addition, to approximate multiple predictions into the future (i.e., speculative update), an offset parameter is available to increase the distance between the history window and the predicted next value. This offset is zero for a normal predictor.

In all of my value predictor experiments, there were no key-space hashing collisions (ie, an infinite table).

4.1.2 Results

I chose the following cases to run simulations because of two reasons:

1. they contributed many misses, and
2. they were code constructs to which value prediction could easily be applied in an intuitive sense.

[1]. . . or consumed.

cc-train-#1

histlen	offset	exp	code
1	0	0.604	0.323
2	0	0.831	0.515
4	0	0.829	0.771
5	0	0.830	0.817
10	0	0.823	0.845
20	0	0.784	0.800
10	1	0.784	0.808
10	2	0.753	0.771
10	3	0.723	0.743
10	4	0.695	0.718
10	5	0.672	0.698
10	10	0.588	0.624
10	20	0.480	0.543

The prediction accuracy starts off low, increases to a maximum plateau, and then decreases as the history length gets long. For very short history length, the accuracy is not as good because there is not as much information used in the prediction. Long history lengths have too much history dilution due to history changes.

The accuracy for "exp" is better for short history lengths because it has many values which are unique, while "code" has less than 10 different values which are not unique in the data structure.

As the offset is increased, the accuracy decreases. Data values have locality too!

In this case, "code" can only beat gshare for history length > 4 and not more than 5 iterations into the future.

I will merely summarize the rest of the data, as long lists of numbers are not interesting.

cc-train-#2

This is the "p" case. Gshare has an accuracy of 76%, and for no length of history length does the value predictor beat gshare!

cc-train-#4

Again, the "p->exp->code" case. I found that the value prediction accuracy on "p" cannot beat gshare's branch accuracy.

For "exp", the value predictor can beat gshare for history lengths greater than 1 for the next-value. Predicting more than 1 iteration into the future requires more history to beat gshare: for 2 iterations, history length of 3 is required.

For "code", the value predictor beats gshare for the next-iteration when the history length is greater than 3. Predicting more than 1 iteration into the future requires more history to beat gshare: for up to 2 iterations, history length 4 or greater is needed. For 3 iterations into the future, 5 history values are needed.

li-train-#1

The branch in question is "this->flags & CONST". Gshare already has a 80% accuracy. The VP predicts "this" at most 12.4% better than gshare at histlen 2. The VP predicts "flags" at most 8% better than gshare at histlen 23, but doesn't beat it at all until histlen 13 — much too long!

li-train-#2

The branch test is "prev->n_cdr", and gshare has 67% accuracy. The value predictor does better on "prev" for any history length, but "n_cdr" only beats gshare with history length > 10 — much too long.

li-train-#3

The branch test is "prev->n_flags & CONST", and gshare has 86% accuracy. With only histlen 2, "prev" has 93.6% accuracy, but "n_flags" has only 62.1%. Unfortunately, "n_flags" never beats gshare, but "prev" still maintains 92% looking 3 iterations into the future.

4.1.3 Summary Comments

I ran a number of other runs similar to those listed above. The results did not suggest anything conclusive with regard to how useful predicting these value will be for the purpose of evaluating the data to get the branch outcome. In some cases the values were predictable with small or moderate history length, but in other cases gshare nearly always beat the value predictor. I think this suggests that this mode of value prediction alone is not the answer.

4.2 Adaptive Value Prediction

Remember the cc-train-#1 case: a long string ("row") of values. When one node is added or deleted from the middle of the list, the disruption will probably cause histlen misses while the new value is passed through the history FIFO. Intuitively, it seems as though something intelligent could be done if the disruption could be detected so less misses are produced.

```
for ( hash=0 ; hash < NBUCKETS ; hash++ )
    for ( p = table[hash] ; p ; p = p->next )
        {
        if ( p->exp->code != REG ) continue;
        . . misc code . .
        }
```

One way to compensate for disruptions is to take advantage of structural information. In this code segment, when we come upon line 1 the first time, we know the data pattern is starting over. In addition, we know when we enter a new linked list because a new value of "hash" is produced ("hash++" in line 1). We can use this information to stay on track.

4.2.1 Row Techniques

I tried several techniques to try to resynchronize the value predictor each time the sequence was started over. This should prevent disruptions within histlen of the end of the previous row (the last time this code sequence was executed) from causing mispredicts near the beginning of this row.

Reset Histlen on Reset

The first method I tried was to reset histlen to 0 when the outer loop was started. Besides the normal hash table of predictors, a separate "length0" predictor existed holding the last *first value*. As the loop continued, the second branch test would use histlen 1, and histlen would be incremented once per update until it was back to the normal value. When using a deficient histlen (less than the maximum value), the system looks up the last predictor that matches the short history.

The following table shows accuracy for the normal and adaptive predictors. The value being predicted is "code".

max histlen	normal	adaptive
1	0.251	0.355
2	0.533	0.538
3	0.696	0.686
4	0.769	0.760
5	0.813	0.794

For histlen 1, the adaptive works better because *all history* is only 1 — the same as the "length0" special predictor. We avoid misses when the tail of the previous row changed. For the other cases, this effect is smaller, and eventually smaller history becomes a liability when it is a sub-sequence which occurs somewhere else.

Stuff History on Reset

Since the adaptive length did not work well in the previous experiment, I tried to just inject a number (approximately histlen) of constant values into the history buffer on a reset (starting the algorithm again). This is better because now all lookups use the full history. The constant chosen was a constant value which did not occur in the natural value stream.

histlen	normal	adaptive
1	0.251	0.292
2	0.533	0.546
3	0.696	0.705
4	0.769	0.778
5	0.813	0.818

The adaptive predictor does work better than the normal predictor. The improvement is slight because the average length of the string of values (the number of nodes in the structure) is long compared to the number of disruptions near the tail end. (Disruptions are fairly randomly distributed.)

nValue Base Predictor

After having some success with an adaptive technique, I thought that perhaps there is a better base predictor than a bimodal-type value predictor. As I was going through the value streams by hand, I thought that it might be a good idea to keep several candidate values with a count of how many times it was the correct prediction and then on a lookup choose the value with highest count. I started with n = 1.

histlen	bimod-ish	nValue (n=1)	(n=2)	(n=8)
1	0.251	0.169	0.251	0.405
2	0.533	0.471	0.430	0.498
3	0.696	0.667	0.471	0.554
4	0.769	0.773	0.565	0.620
5	0.813	0.835	0.678	0.697

These results were surprising! They show that for the histlen 4 and histlen 5 cases, it is better to change to a new value *right away* rather than have hysteresis like a bimodal-type predictor!

Another interesting but not as surprising result is that for very short history lengths (1 or 2), an nValue predictor with large n (2, 8) can do better than a bimodal predictor or nValue (n=1). For example, for history length 1, the nValue (n=8) predictor works well because it keeps track of, for example, all the different things that come after A: AA, AB, AC, AA, AD, etc, where the bimodal-ish and the nValue (n=1) keep track of a small number (1 or 2).

Once data changes, it is not likely to change *back*, so it is advantageous to update the predictor immediately for moderately long history lengths (3-5).

4.2.2 Bucket-Based Techniques

Previously, the only structural information I used was reset, that is, the beginning of the value sequence. We also have more information: the outer loop counter! To help solve the old variable-length history impact problem, we can notice when we start a new linked list (ie, moving from table[i] to table[i+1]) and try to make sure the value prediction mechanism is caught up.

Stuff Const on Outer Loop

Each time "hash" is incremented, a constant is inserted into the value history buffer. These runs also use the nValue (n=1) base predictor. Again, the constant chosen was one which did not occur in the natural value stream.

histlen	normal	stuff
1	0.158	0.229
2	0.455	0.431
3	0.650	0.618
4	0.763	0.736
5	0.827	0.810
6	0.843	0.870
7	0.847	0.888
8	0.847	0.902
9	0.842	0.902
10	0.835	0.900

This method yields almost a 5% increase for moderately long history lengths.

Stuff Many Const on Outer Loop

Each time "hash" is incremented, *histlen* constants are inserted into the history buffer. I tried this method for completeness, but its performance was abysmal.

Stuff Hash on Outer Loop

Each time "hash" is incremented, the value of hash is inserted into the value history buffer.

histlen	normal	stuff
1	0.158	0.657
2	0.455	0.891
3	0.650	0.920
4	0.763	0.923
5	0.827	0.921
6	0.843	0.919
7	0.847	0.914
8	0.847	0.910
9	0.842	0.906
10	0.835	0.900

As would be expected, this predictor uses more information than the "Stuff Const on Outer Loop" predictor, and so achieves almost 30% better accuracy at histlen 3. Instead of only knowing that there has been a transition, the predictor now also knows *where* it is in the sequence.

Catenate Hash

Keep regular value history, but catenate the current value of "hash" to the lookup key, which is now histlen history values along with the value of hash.

histlen	normal	stuff
1	0.158	0.792
2	0.455	0.923
3	0.650	0.918
4	0.763	0.913
5	0.827	0.906
6	0.843	0.897
7	0.847	0.887
8	0.847	0.879
9	0.842	0.871
10	0.835	0.862

This prediction scheme is very good! With only two history value, it can get over 90% accuracy. The previous method, "Stuff Hash on Outer Loop", adversely impacted the history by adding values. This method keeps the history intact and in effect makes each linked list have independent history modulo the boundary conditions.

4.3 Conclusion

The main purpose of these experiments was to investigate different ways to combine values to increase value-prediction accuracy. In addition, we saw that when dealing with a single value stream, it is better to change the value after one miss instead of using hysteresis.

With the potential for using value-prediction demonstrated here, I chose to move on to applying this directly to the problem of branch-prediction.

Section 5

Hybrid Branch Predictor

In the previous section, we saw that with a history length of only 2, if the value of "hash" was catenated onto the lookup key, a value-prediction accuracy of 92.3% could be attained for a particular branch. The next experiment applied this method to predicting the branch outcome.

5.1 Methodology

The prediction scheme used in these experiments was fundamentally a context-based value predictor which had been modified so that instead of predicting the next value in the stream, it predicts the branch outcome based on the context. In other words, the trained next-value outcome comes from a different value stream than the history values used as the lookup key. Additionally, an extra value can be catenated to the key for lookup, for example, if the value stream is "code" in "p->exp->code", then "exp" might be catenated to the key to add more information to the predictor.

5.2 Results

Because this is a much more directly useful result than simply predicting the value in the previous section, I simulated more cases, with the following in mind when selecting which cases to try:

1. choose those which are characteristic of a certain program construct type, and
2. choose those which miss very often.

For comparison purposes, I ran the old bimodal-type, the nValue n=1, and n=2 predictors with both hash catenation and no hash catenation. Again, this is from the cc-train-#1 case.

histlen	bimod, no hash	bimod, hash
1	0.641	0.911
2	0.739	0.930
3	0.818	0.924
4	0.846	0.917
5	0.869	0.908

histlen	nValue (n=1), no hash	(n=1), hash
1	0.504	0.873
2	0.644	0.934
3	0.779	0.931
4	0.839	0.926
5	0.878	0.918

Here again, with only a history length of 2, with hash catenation, we get a prediction accuracy of 93.4%! This is about 1.1% more than the "code" value prediction accuracy (92.3%). This difference is because while the value predictor must get the value exactly right, the branch outcome is only 0 or 1, thus the aliasing helps the accuracy.

histlen	(n=2), no hash	(n=2), hash
1	0.653	0.839
2	0.680	0.875
3	0.720	0.883
4	0.757	0.892
5	0.811	0.893

The nValue (n=2) predictor does not change as fast as the bimodal-type predictor, so it performs worse.

5.2.1 Expanded Results

The follow data was collected during Summer 1998 to expand results for the hybrid predictor, with outer-loop catenation, with the nValue predictor (n=1). I chose the following cases because they had an obvious outer loop counter or induction variable which could be used for catenation.

cc-train-#1

Without Catenation: ("histlen" is the value history length, and "ofs" is the offset of the history into the future, eg, ofs=0 predicts the next value.) Gshare gets approximately 75% on this branch.

histlen,	ofs=0	ofs=1	2	3
1	0.504	0.540	0.555	0.541
2	0.644	0.641	0.606	0.629
3	0.779	0.746	0.739	0.727
4	0.839	0.823	0.805	0.788
5	0.878	0.855	0.832	0.822
6	0.877	0.854	0.838	0.824
7	0.869	0.851	0.833	0.823
8	0.860	0.844	0.829	0.816

With Catenation:

histlen,	ofs=0	ofs=1	2	3
1	0.873	0.894	0.879	0.865
2	0.934	0.925	0.918	0.913
3	0.931	0.923	0.915	0.911
4	0.926	0.916	0.908	0.899
5	0.918	0.906	0.896	0.889
6	0.907	0.895	0.885	0.875
7	0.895	0.885	0.873	0.866
8	0.885	0.874	0.864	0.855

cc-train-#2

Without Catenation:

histlen,	ofs=0	ofs=1	2	3	4	5
1	0.651	0.636	0.632	0.617	0.621	0.614
2	0.759	0.737	0.745	0.715	0.702	0.680
3	0.808	0.779	0.761	0.734	0.718	0.701
4	0.794	0.762	0.737	0.715	0.697	0.677
5	0.738	0.713	0.690	0.670	0.652	0.634

With Catenation:

histlen,	ofs=0	ofs=1	2	3	4	5
1	0.924	0.729	0.693	0.670	0.676	0.664
2	0.884	0.770	0.753	0.737	0.725	0.708
3	0.830	0.762	0.739	0.712	0.697	0.670
4	0.768	0.720	0.690	0.662	0.640	0.615
5	0.706	0.666	0.634	0.606	0.583	0.561

Gshare predicts this branch with approximately 77% accuracy. With small offset and short history, the hybrid predictor with catenation can improve considerably (77% vs. 92%).

cc-train-#3

This case comes from the same code as cc-train-#2, but the compiler produced a special case, yet it is still a highly-missed branch. Gshare gets approximately 79% on this branch.

Without Catenation:

histlen,	ofs=0	ofs=1	2	3	4	5
1	0.793	0.773	0.793	0.793	0.827	0.813
2	0.756	0.762	0.767	0.788	0.776	0.772
3	0.728	0.717	0.729	0.726	0.721	0.728
4	0.701	0.693	0.689	0.685	0.688	0.684
5	0.653	0.650	0.646	0.647	0.646	0.639

With Catenation:

histlen,	ofs=0	ofs=1	2	3	4	5
1	0.908	0.901	0.903	0.902	0.904	0.899
2	0.832	0.829	0.829	0.827	0.822	0.820
3	0.771	0.769	0.770	0.766	0.764	0.760
4	0.717	0.716	0.715	0.712	0.708	0.706
5	0.666	0.664	0.663	0.659	0.655	0.652

This does even better for very short history lengths (eg, 1) because it is nearly always the same with respect to "hash" — this branch is the "p" for *only* the first iteration of the inner loop.

cc-train-#4

This case is similar, but not quite the same as cc-train-#1. Gshare gets only 61% on this branch.

Without Catenation:

histlen,	ofs=0	ofs=1	2	3
1	0.542	0.564	0.562	0.560
2	0.654	0.601	0.593	0.582
3	0.729	0.677	0.659	0.636
4	0.792	0.744	0.721	0.692
5	0.834	0.791	0.760	0.738
6	0.846	0.806	0.782	0.763
7	0.836	0.805	0.786	0.768
8	0.823	0.800	0.782	0.762

With Catenation:

histlen,	ofs=0	ofs=1	2	3
1	0.795	0.798	0.793	0.787
2	0.853	0.842	0.836	0.828
3	0.863	0.856	0.844	0.843
4	0.871	0.852	0.846	0.835
5	0.861	0.846	0.833	0.822
6	0.850	0.833	0.821	0.807
7	0.835	0.820	0.807	0.793
8	0.823	0.807	0.793	0.781

In this case, "hash" catenation has a significant effect for histlen > 2, and performance is not degraded very much with increased window offset.

cc-knights-#2

This is another case with an outer loop counter which can be catenated. Gshare gets 71% on this branch.

Without Catenation:

histlen,	ofs=0	ofs=1	2	3	4
1	0.874	0.839	0.812	0.792	0.783
2	0.863	0.848	0.828	0.816	0.803
3	0.826	0.813	0.796	0.779	0.768
4	0.797	0.784	0.764	0.750	0.738
5	0.763	0.748	0.732	0.719	0.708
6	0.727	0.716	0.701	0.689	0.677
7	0.694	0.683	0.670	0.658	0.647
8	0.663	0.653	0.640	0.629	0.619

With Catenation:

histlen,	ofs=0	ofs=1	2	3	4
1	0.948	0.925	0.912	0.900	0.890
2	0.884	0.864	0.846	0.829	0.815
3	0.837	0.813	0.792	0.773	0.757
4	0.794	0.770	0.747	0.728	0.712
5	0.755	0.730	0.708	0.690	0.673
6	0.718	0.694	0.673	0.655	0.637
7	0.683	0.660	0.639	0.621	0.605
8	0.650	0.628	0.608	0.591	0.575

cc-knights-#3

This is another case which tests "p". Gshare gets 69% on this branch.

Without Catenation:

histlen,	ofs=0	ofs=1	2	3	4
1	0.881	0.848	0.822	0.800	0.798
2	0.872	0.859	0.841	0.831	0.822
3	0.835	0.827	0.812	0.798	0.788
4	0.809	0.798	0.781	0.771	0.763
5	0.777	0.766	0.755	0.744	0.733
6	0.747	0.739	0.728	0.717	0.708
7	0.720	0.711	0.701	0.691	0.682
8	0.693	0.685	0.676	0.667	0.658

With Catenation:

histlen,	ofs=0	ofs=1	2	3	4
1	0.927	0.881	0.846	0.817	0.794
2	0.888	0.852	0.831	0.813	0.801
3	0.849	0.819	0.800	0.783	0.768
4	0.812	0.788	0.771	0.753	0.739
5	0.777	0.757	0.741	0.724	0.708
6	0.746	0.728	0.714	0.696	0.682
7	0.717	0.701	0.687	0.671	0.658
8	0.689	0.675	0.662	0.647	0.635

5.2.2 Non-outer-loop cases

As a slight aside, I evaluated the following test from cc-knights-#1: `fmt[i] == 'e'`. Gshare gets 73%. Doing regular hybrid prediction on "fmt[i]" with no catenation yielded the following:

histlen,	ofs=0	ofs=1	2	3	4
1	0.739	0.573	0.578	0.572	0.608
2	0.730	0.691	0.613	0.585	0.603
3	0.747	0.744	0.643	0.611	0.621
4	0.784	0.754	0.667	0.637	0.641
5	0.806	0.781	0.700	0.668	0.689
6	0.818	0.806	0.720	0.713	0.706
7	0.836	0.823	0.762	0.737	0.736
8	0.849	0.848	0.779	0.768	0.756

I ran the simulation again, catenating the value of "i" to the history of "fmt[i]":

histlen,	ofs=0	ofs=1	2	3	4
1	0.875	0.748	0.744	0.730	0.744
2	0.883	0.858	0.761	0.742	0.753
3	0.888	0.894	0.775	0.759	0.766
4	0.921	0.897	0.797	0.766	0.774
5	0.923	0.917	0.809	0.788	0.802
6	0.936	0.917	0.819	0.814	0.809
7	0.933	0.928	0.841	0.823	0.828
8	0.939	0.933	0.850	0.842	0.840

This clearly illuminates another heuristic for using value history to attain higher prediction accuracies: use the array index as addition key information into the predictor table.

5.3 Summary

Structural information, value information, and value history can all be combined to achieve very high branch prediction accuracies. It is clear from this data that the potential for impressive improvement exists for some branches when value history is combined with other related values: prediction accuracies can be increased by up to 20%, but more importantly, the miss rate was decreased in some cases by 3x.

If the tested-value is used to drive a value-predictor (or hybrid predictor, in this case), then even more accuracy can be achieved by careful selection of the catenation variable. Possible heuristics are: next-outer loop test register, induction variable, array index, and node address for struct data.

In addition, the preceding tables show that it is possible to predict several iterations in advance with relatively small accuracy loss.

Section 6

Conclusions

These are conclusions from my solitary work only. The insight gained here helped to suggest methods and heuristics used in the ISCA paper included in Appendix A.

6.1 Hard to Predict Branches

Most of the "hard to predict" branches can be classified into groups based on the reasons conventional predictors fail.

1. "Popular subroutines" have a hard time getting useful history. Possible solution: keep track of caller, or more carefully choose data stream. The possibilities for exploiting caller information were not explored in my research.

2. Program or data structures which produce a variable number of history bits as the data changes — variable-length history problem. Possible solution: use data values or structural information to keep the predictor "better synchronized" with data.

3. Control locality. The branches within the global history length may not have correlation with the current branch, or the relevant history might be too far away. Each branch has some previous code which is more relevant, so either focus on that history, or figure out what is the globally "important" history.

Points 2 and 3 were addresses by using a local value-history stream for prediction instead of a global history.

Knowing these problems allows us to design predictors which overcome them.

6.2 Value Prediction

In some cases, it is possible to use a context-based value predictor to predict the value to be tested by the branch and have the accuracy beat gshare. Unfortunately, this is not always the case.

There is more information available! Using structural and value information, it is possible to achieve very high prediction accuracies with small history lengths. A context value predictor with length 2 could achieve 92.3% accuracy on a particular branch when the outer loop counter was used as additional information.

Hysteresis is not always good in a predictor. In particular, when the history length is long enough to differentiate between sequences well enough (in my runs, > 2), it is better to update the predictor with the new value right away, on the first miss. Once data changes, it tends to stay changed!

6.3 Hybrid Branch Prediction

If a normal context-based value predictor is modified to predict branch outcomes based on a value stream, with the ability to catenate additional values onto the key, it is possible to accurately predict branches using a short value history and predict a few iterations into the future with good accuracy.

I demonstrated a context-based predictor using structural and value information which could achieve 93.4% accuracy, beating gshare by over 20% for a few particular branches. This suggests that it is possible to tame these "hard to predict" branches!

6.4 Future Work

Future work in the area of using data-values to aid branch-prediction should be focussed on:

1. better heuristics for combining data values,
2. using caller information for popular subroutines,
3. better basic value predictors, and
4. implementable schemes.

6.5 Acknowledgements

I was directed in this work by Prof. Jim Smith. I have discussed some of these ideas with Yiannakis Sazeides, and have worked with Timothy Heil, both graduate students.

Appendix A

ISCA '99 Paper

"Using Data Values to Predict Branches," Timothy Heil, Zak Smith, J.E. Smith, Dept. of Electrical and Computer Engr., University of Wisconsin - Madison. Submitted to the 26^{th} International Symposium on Computer Architecture.

Using Data Values to Predict Branches

Timothy Heil, Zak Smith, J. E. Smith
Dept. of Electrical and Computer Engr.
University of Wisconsin-Madison

Oct. 23, 1998

Abstract

Most dynamic branch predictors use control flow history for making predictions. We investigate the alternative of adding certain data values to the prediction process. A preliminary study of hard-to-predict branches suggests two heuristics for selecting data values that may be useful for increasing branch prediction accuracy. The first is to use a data value tested by the branch itself. The second is to use data values tested by the loop-terminating branch enclosing the branch to be predicted.

We consider a gshare-like predictor structure which exclusive-ORs data value information together with the PC and global branch history when indexing a prediction table. This predictor is designed to use older, committed data values in order to accommodate delays in generating data values. Because the use of data values can cause degraded performance for some branches, profiling is used to select a subset of the branches to be predicted with data values.

Execution-driven simulations using interference-free tables show that the overall performance improvement is relatively small, but a few frequently-executed individual branches benefit considerably, yielding a 25 to 50% reduction in miss rate. A second study with bounded tables reveals that interference eliminates the improvement gained by using data values for most benchmarks. Refining the branch profiling process helps solve this interference problem, however relative improvements are not as good as those with interference-free tables. When comparing the two heuristics for selecting data values, we find that using values tested by the branch subsumes the use of the value tested by the enclosing loop's branch; hence, the simpler heuristic of using tested values may be adequate by itself.

1. Introduction

Branch prediction has been studied at great length, and a wide variety of dynamic branch predictors have been proposed [YEH93, PAN92, CHA95, YEH91, JUA98]. Virtually all predictors proposed to date use control flow history, usually past branch outcomes, as the basis for making predictions. Some use local branch history [SMI81] (the history of the branch being predicted); others use global history [PAN92, CHA95] (the history of other branches); some use combinations of the histories [KES98, MCF93, YEH93, YEH91]. The goal in this work has been to select some set of history bits and combine them so that prediction accuracy is improved.

Generally, history bits are selected so they correlate well with the branch being predicted, and the history bits are combined in ways intended to reduce interference due to aliasing in prediction tables [MIC97, MCF93, CHA96, SPR97]. Refinements include hybrid predictors which select among multiple predictors depending on which best matches the branch being predicted [CHA95, KES98, MCF93, YEH93, YEH91], and more recently, dynamic history length predictors that search for the proper global history length at runtime [JUA98].

Despite these refinements, common sense says that not all mispredictions are caused by failure to combine just the right branch history bits or by interference. There are some branches that will be mispredicted regardless of the branch history bits used even when there is no interference. On the other hand, it is unlikely that all such mispredicted branches are fundamentally unpredictable (a precise definition of "fundamentally unpredictable" is admittedly elusive; we only use the term in an intuitive sense here). In particular, there may be other information besides control flow history that may aid in branch prediction. This study considers a new form of information to predict branches -- program data values.

1.1. A Search for a Better Predictor

As a way of improving branch predictors, we first looked at specific branches that are hard to predict using a conventional branch predictor (gshare [MCF93]). We studied these branches in isolation, looking at their specific program context, and tried to evaluate their predictability in an intuitive way. Examining each specific branch, we tried to invent a special predictor that only had to be a accurate for that particular branch. We found that in many cases, we could devise good, simple predictors by using certain selected data values as inputs to the prediction process.

We identified two common causes for mispredictions using conventional predictors, and, consequently, two simple heuristics for identifying data values that could be used to improve the prediction process. A summary of the two causes for mispredictions follow.

"Loss of Information" in reducing tested values to a single branch outcome A branch instruction performs a many-to-one mapping on its input data, and information is lost by considering only branch outcomes[1].

[1]We use the MIPS instruction set, and consider a SET instruction that feeds a branch to be part of the branch. When we speak of an input value of such a branch, we are actually referring to an input value to the SET instruction.

For example, consider the following value stream for variable X.

X: 3 2 1 0 2 1 0 3 2 1 0 1 0 2 1 0 1 0

If the branch is testing for X==0, the corresponding branch outcome stream follows.

B: N N N T N N T N N N T N T N N T N T

There is no immediately apparent pattern in this sequence of branch outcomes, and it could be relatively difficult to predict. However, the original stream of X values have an obvious pattern: The sequence repeatedly starts at a value and counts down to zero. If a predictor has access to the value sequence, then whenever a non-0 appears, the predictor can immediately predict the point where a 0 eventually will occur. For example if a 1 is observed, then the next time the branch executes, it should be predicted taken. However, if the predictor only looks at the branch outcomes, then this information is lost; in effect all the non-0s are reduced to *not taken* and all 0s are reduced to *taken* by the branch evaluation process. A branch with just this behavior was observed in *gcc*. A similar effect was noted in [SAZ98] as a common source for loss of data predictability.

Loss of control flow "context" A motivation for using global branch history is that a given branch may be correlated with earlier branches in the instruction stream. These earlier branches, as summarized in the global history, provide a context for making the prediction. However, the global history context is not necessarily easily identifiable. Some programs follow paths with different numbers of dynamic branches separating executions of two correlated branches. The bit in the global history that refers to a specific static instruction shifts around, so that this global context becomes blurred or completely obscured. Consider the following loop from *gcc*.

```
for ( hash=0 ; hash < NBUCKETS ; hash++ )
  for ( p = table[hash] ; p ; p = p->next )
    {
    if ( p->exp->code != REG ) continue;
    . . misc code . .
    }
```

This code loops through a hash table, following short linked lists that emanate from the hash buckets (the table[i]). In *gcc*, this data structure changes slowly over time. Between passes through the outer loop, perhaps four of the 32 linked lists will change. Hence, if a predictor could determine the value of i, indicating the context, it could predict that particular list search accurately. For example, if entry table[i] has 3 elements in its linked list, the branch pattern for the test of ‘‘p’’ (line #2) will be T T T N. If the predictor somehow knew it was working on entry table[i], it would be able to predict the repeating pattern of local branch outcomes, as long as that one list did not change.

A typical history-based branch predictor would guess which list it is working on by looking at global branch history. If the predictor for the branch on line #2 could synchronize on the outer loop branch, it could accurately determine the context (provided the global history register were long enough). In this example, however, even a very long global history register will not help because other dynamic branches are mixed into the global history register with the loop branch, and there is a variable number of these dynamic branches due to the small changes in the overall data structure. For example, if a list node were deleted from table[i-1], then a predictor looking at the global history would lose its place with respect to the outer loop and not know when the search of table[i] begins. This context problem also affects line #4, where p->exp->code is tested. The data hanging off each linked list element is relatively constant and would be predictable, as long as the context (value of i, or p) were known.

If a predictor could use the value tested by the loop-closing branch as an input, then an accurate context could be established. For the particular section of code shown above, we found that this approach cut the misprediction rate roughly in half.

Our conclusion from the initial study was that data values could help branch prediction. Furthermore, we identified two specific classes of data values that should be considered for aiding the prediction process: the input value that the branch itself tests (referred to as the *tested value*), and the value tested by the branch terminating the innermost enclosing loop of the branch being predicted (referred to as the *loop value*).

1.2. Other Related Work

Besides the branch prediction studies cited above, there is other related work directed toward the same branch prediction problems we are attempting to solve.

Mahkle and Natarajan [MAH96] used a software-based approach for applying data values to branch prediction. An execution profile is used by the compiler to generate branch-specific prediction functions in software, and the results are applied to hardware predictors at runtime.

Branch anticipation [FAR98] is similar to our work in that it carefully targets and evaluates specific difficult to predict branches. Instead of trying to further improve a branch's prediction accuracy, however, anticipation attempts to use data value regularity and predictability to execute the branch ahead of the rest of the execution stream. The resulting early resolution of the branch reduces misprediction penalties.

The work in [NAI95] does not use data values, but it does attempt to provide more accurate "context" for making branch predictions. Rather than using branch outcomes, control path information (e.g. sequences of branch PC's) is used for predicting branches. This provides better program context for the predictor, and for certain branches shows performance improvements.

1.3. Paper Overview

In Section 2, we discuss issues related to branch predictors that use data values as inputs and describe the structure of branch predictors we use in our study. In Section 3, we describe the performance evaluation methodology used in the paper. In Section 4, we use a cycle-level simulator to determine the potential of branch prediction using data values. Prediction tables are large and free of aliasing, but the timing of data value updates is accurately modeled. In Section 5, we consider more realistic fixed-size tables with aliasing. Both Sections 4 and 5 compare predictors using data values with equivalently-sized gshare predictors. In Section 6, we draw conclusions and discuss possibilities for future research.

2. Constructing Branch Predictors that Use Data Values

First, we briefly describe two issues that affect the construction and use of branch predictors that use data values as inputs.

Acquiring the data values We have observed that a branch's tested values are useful for predicting the branch's outcome. An apparent flaw in this approach is that the value will often not be ready at the time the branch is fetched and therefore can not be used for prediction. The prediction process typically runs well ahead of computation, and several instances of the static instruction producing the tested value may be pending when the value is needed for branch prediction. To solve this problem, we use the most recent known-correct (committed) data value and concatenate a count of the number of outstanding updates to that value. In other words, *we use an older data value combined with the number of expected updates as an approximation of the data value we are really interested in*. The count is used to prediction ahead from the most recent known value to the current position in the sequence. For instance, in the example in Section 1.1, if the data value for X is 2, and the count is two, then current value of X to be tested by a current branch is expected to be 0. If the value is part of a repeating sequence, then this approximation should be reasonably accurate.

Interference in the prediction table Interference due to aliasing is a problem in conventional branch predictors, and injecting data values into the process can potentially make interference much worse. For example, consider a simple loop of length 1000. Predicting the loop-terminating branch is very easy. But, if we combine the value of the loop counter, which ranges from 1 to 1000, into an index hashing function, the branch will now be spread across as many as 1000 different table entries. Prediction accuracy will decrease because the learning time will increase and added interference will hurt both this branch and others. This means it is important to select only certain branches for prediction with data values. Consequently, we use profiling to select the branches which benefit from prediction using data values; data values are only used for those branches.

2.1. Branch Predictor Design

The predictor implementation we propose is in Fig. 1. It contains a set of prediction registers (PREGs) one corresponding to each architected register. Each PREG contains a recently committed value (VALUE) for the corresponding architected register, concatenated with a count (CNT) of the number of

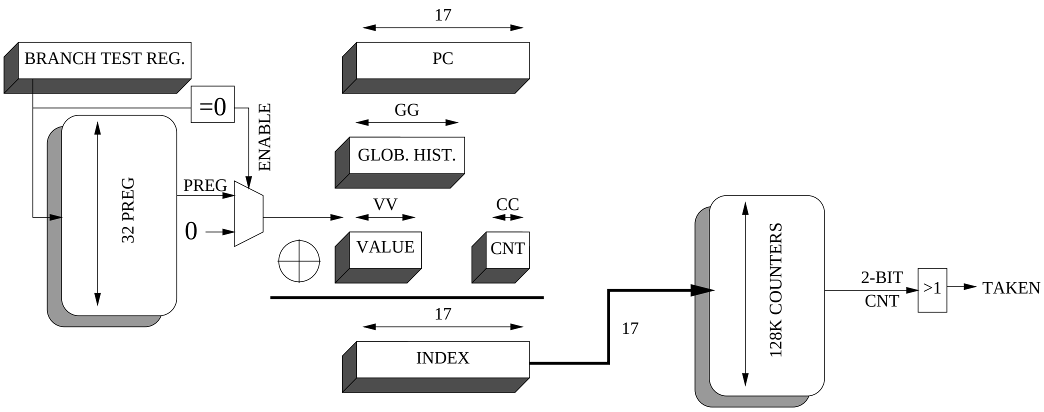


Fig. 1. A branch predictor that combines the program counter, global history, and data value information.

pending updates to the register. In Section 5 we show that three bits are sufficient for the CNT field.

A branch instruction accesses the PREG table with the designator of a register whose value has been determined to be useful for prediction (this will depend on the heuristic and is explained later in this section). For the addressed PREG, the VALUE and CNT fields are read out, and these bits are selectively exclusive-ORed with some low order PC bits and global history bits (similar to gshare [MCF93]). We use the notation Ppp_Ggg_Vvv_Ccc_size to describe such a predictor. pp is the number of PC bits, gg the number of global branch history bits, vv the number of VALUE bits, and cc the number of CNT bits exclusive-ORed together to generate the index. Our baseline predictor is P17_G12_V00_C00_128K; we use 17 PC bits and 12 global history bits. The 17-bit result of the hash function is used to access a table of 128K 2-bit saturating counters which provide the prediction in the conventional way [SMI83].

The prediction table size is larger than is used in current microprocessors, but we chose to use large prediction tables to allow for improvements in future processor designs.

The selection of the VALUE and CNT fields is enabled by a flag in the branch instruction that is set by a tagging/profiling process described later. If the enable is not active, then a conventional gshare hash is used; if it is active, then the VALUE and CNT fields are also included.

2.2. Multiple Branch Values

The predictor only provides for one entry to be read from the PREG file, and branch instructions typically have two operands. Due to features in the MIPS instruction set and compilers, however, most MIPS branch instructions compare a register value with an immediate operand, typically zero. Hence, when the tested value heuristic is used, only the non-immediate branch operand is useful for improving branch prediction. Our current implementation reads only the PREG corresponding to first operand when a branch instruction is encountered. The Table 1 shows the fraction of branches where the second operands is zero. In addition to these, some of the remaining two source branch instructions may be comparing to an immediate value other than zero. These percentages are high, but not overwhelmingly so. In future work, we will consider methods that use both branch operands (in a way similar to the way we handle SET instructions discussed below.)

Table 1. Dynamic branches with the second operand zero.

Benchmark	% Dynamic Branches
COMPRESS	77
GCC	65
GO	46
IJPEG	92
LI	70

2.3. SET Instructions

One reason that branches often test an immediate value is that the MIPS instruction set uses a SET instruction to compare two values, and the one/zero result is later tested by a branch instruction comparing for (in)-equality with zero. Clearly, when this happens, the predictor should use values input to the SET instruction rather than the branch. To implement this, when a SET instruction is executed, the zero/one result is not stored in the PREG VALUE field. Rather, the two input architectural register operands of the SET instruction are subtracted, and this value is used to update the PREG in place of the comparison result. Hence, we form a composite of the two tested values for future use. The PREG count field is handled normally. When the subsequent branch is predicted, the branch predictor will use this composite value as a matter of course.

2.4. Instruction Tagging

As noted earlier, not all branches should be predicted using a data value; only branches whose performance improves should be selected to use data values. To implement selective use of value information and to implement the two value selection heuristics, we assume that instructions can be tagged. The tag values are set by profiling software and analysis of binaries and serve as inputs to the prediction hardware. For our study, using instruction tagging was the the most straightforward approach for passing profile information into the prediction process; of course, in a real implementation other methods are possible. For the two heuristics, we now list the tag values that are assigned, and their meaning.

Tagging for the tested value heuristic

-- Instructions that produce tested values to be used for prediction are tagged. These are the instructions that should update the PREG associated with their architectural result register.

-- Branch instructions are tagged to indicate if a PREG should be used when accessing the predictor. The PREG corresponding to the 1st register operand of the branch is used.

Tagging for the loop value heuristic

-- Instructions that produce values used by a branch that terminates an enclosing loop are tagged. These instructions should update the PREG corresponding to their result register. These instructions only need to be tagged if a branch enclosed by the loop will uses the updated PREG.

-- Branch instructions are tagged to indicate that a PREG should be used as input to the predictor. For the loop heuristic, this tag also has to indicate which PREG should be used, because it is not necessarily specified as an operand by the branch instruction being predicted; rather, the PREG corresponding to the loop value tested by the enclosing loop branch should be accessed.

When an instruction tagged to update a PREG is fetched and decoded, the count field of the PREG is incremented, reflecting another outstanding update to the value field. If the instruction is squashed due to a misprediction the count must decremented. When the instruction commits, the PREG value is updated with value produced by the instruction and the count is decremented.

3. Simulation Methodology

3.1. Simulator

To study PREG-based prediction we augmented the SimpleScalar [BUR97] cycle-level out-of-order simulator. Detailed timing simulation is essential for studies that use data values for prediction because the timing of data value availability is a very important consideration. Below we list the features of the aggressive superscalar model used for this study.

Issue width:	8-way out-of-order
Load/store:	4 instructions per cycle
Issue queue:	64 Instructions deep
Minimum misprediction penalty:	8 cycles
Branch Target Buffer:	16K Entry, 4-way set associative
Return Address Stack:	16 Entry
L1 I-Cache:	64KB, 16byte lines, 2 way set assoc., LRU
L1 D-Cache:	64KB, 16byte lines, 2 way set assoc., LRU
L2 Cache:	1MB, 16byte lines, 4 way set assoc., LRU, 10 cycle latency
Memory:	120 cycle latency

3.2. Benchmarks

Five of the more difficult-to-predict SPEC95 benchmarks were simulated. All were simulated for 200M instructions, except for *go*, which completed at 132M instructions. Reduced input data sets were used to make the execution profile more complete for 200M instruction simulations. The benchmarks are documented in Table 2. The *Branch Working Set* is the number of static branches that account for 90 percent of the dynamic branches. The misprediction rate and instructions per cycle (IPC) are given for a gshare predictor using 12 bits of global branch history with a 128K entry (256K bit) table.

3.3. Profiling and Tagging

To support our simulations studies to follow, we implemented an idealized profiling method and methods for tagging instructions. For each of the two heuristics studied, we first simulated each benchmark program using data values to predict all branches. These runs use interference-free prediction tables (described in the next section). Then, those static branches whose prediction showed improvement were tagged in the binary to enable the use of data information for their prediction in later simulation runs.

We used the same data for profiling as for the simulation experiments; however, as we shall see, the branches responsible for overall performance benefit tended to be a static few branches that individually show major improvements, and therefore appear to be fundamentally more predictable when using data values.

For tagging instructions, we implemented the following methods.

Tagging Method for Tested Value Heuristic

During the profiling phase described above, a list is generated containing all the static instructions that produce values consumed by a branch instruction or a SET instruction feeding a branch. The instructions in this list are tagged as instructions that update PREG entries.

Tagging Method for Loop Value Heuristic

The program binary is analyzed to determine for each branch instruction, the branch instruction which terminates its innermost enclosing loop. The register tested by the loop branch is used to tag the enclosed branch under consideration. To find enclosing loop branches, a list of starting addresses of all procedures

Table 2. Benchmark Characteristics

Benchmark	Input	Branch Working Set	128K entry gshare, 12-bit history Mispredict Rate (%)	IPC
compress	400000 e	26	10.1	1.40
gcc	ref/input/gcc.i	3126	7.4	1.29
go	9 9	1092	12.7	1.20
ijpeg	test/input/specmun.ppm	84	7.5	3.53
li	queen.lsp	63	3.3	3.15

is extracted from the binary. For each branch in each procedure ("branch A"), an enclosing branch is said to exist if there is a backward branch in the same procedure, coming after branch A, whose target is sequentially before the address of branch A. Although there may be multiple enclosing branches, only the first one found is reported for tagging. In the case of nested enclosing loops, this will be the innermost one. Overall, this process is straightforward, and the details are omitted for brevity.

4. Interference-Free Prediction

To determine the overall potential of using data values for branch prediction, we first used an interference-free variation of the proposed branch predictor. This predictor is illustrated in Fig. 2. The full PC of the branch, the PREG, and a 16-bit global branch history are concatenated to form a direct index into a table of 2-bit counters. No hashing is necessary; the simulated table is large enough to assign a separate 2-bit counter to all patterns that occur.

For comparing performance, we use an interference-free gselect predictor [MCF93] that uses the full PC of the branch concatenated with 16 bits of global history. Table 3 compares the results for the two interference-free prediction methods. For each method, performance in both misprediction percentage and

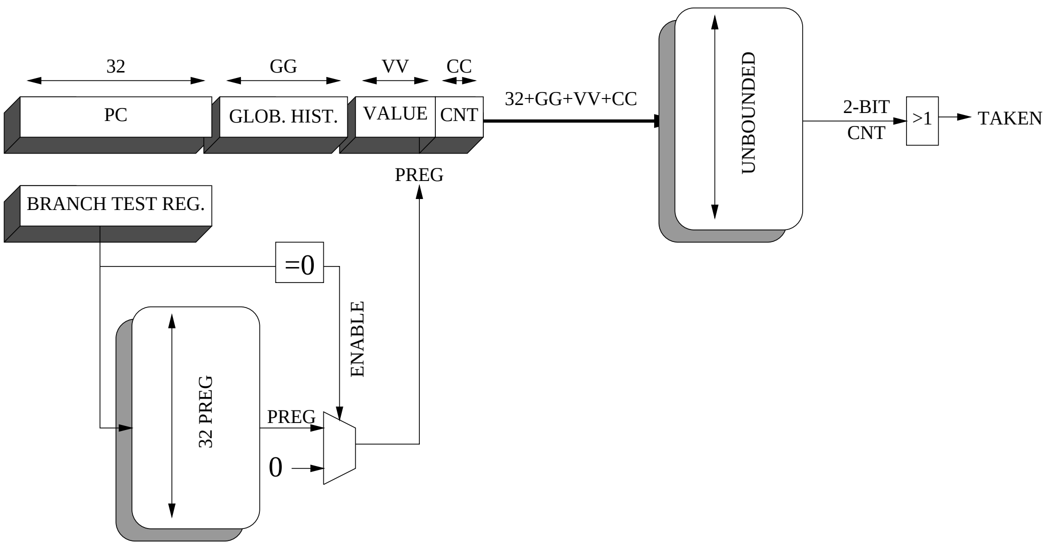


Fig. 2. Interference-free version of branch predictor using data values.

instructions per cycle (IPC) are given. The tested value heuristic outperforms the loop value heuristic in almost every case (except *go* where the difference is small), so we show only the improvement of the tested value heuristic over gselect in the last two columns. The *ijpeg* benchmark improves misprediction rate by 23 percent and overall performance by 4.5 percent. However, considering all the benchmarks, improvements are relatively modest.

Table 3. Performance with interference-free predictor tables

Benchmark	gselect 16-bit		PREG 16_16_06 Test heuristic		PREG 16_16_06 Loop heuristic		Improvement (%) Loop heuristic vs. gselect	
	Miss %	IPC	Miss %	IPC	Miss %	IPC	Miss %	IPC
COMPRESS	8.4	1.45	7.8	1.50	8.1	1.50	6.7	2.8
GCC	4.4	1.39	4.1	1.40	4.3	1.40	8.2	0.6
GO	8.8	1.27	8.4	1.27	8.3	1.27	4.3	0.2
IJPEG	7.1	3.57	5.4	3.73	5.9	3.70	23.1	4.5
LI	3.1	3.18	2.6	3.26	2.9	3.21	15.5	2.6

To get a clearer picture of the potential of using data values for branch prediction, we focussed our attention on those branches whose prediction accuracy is improved. Table 4 summarizes the static branches that account for 90% of the overall performance improvement. For example, we observe that in *compress*, four static branches account for 90% of the observed improvement, and the average miss rate for these four branches is reduced from 9.2 percent down to 6.2 percent; i.e. about a third of the mispredictions are eliminated. Furthermore, these four static branches account for only 15.4 percent of the overall branch working set (see Table 2). The most dramatic example is the *li* benchmark, where 3 static branches account for 90% of the improvement, and on average their miss rate is cut in half -- from 18% down to 8.9%.

These results indicate that using data values can potentially provide very large performance improvements, but only for certain static branches. We take this as a promising sign -- our benchmark sample was confined to SPEC benchmarks; there may be other programs which have higher percentages of static branches that can benefit from using data values for branch prediction. These results also reinforce the need to focus PREG prediction on specific problem branches.

Table 4. Static branches yielding 90% of performance improvement

Benchmark	Number	% of Working Set	gselect Miss Rate (%)	PREG Miss Rate (%)
COMPRESS	4	15.4	9.2	6.9
GCC	123	3.9	7.7	4.2
GO	19	1.7	12.0	9.0
IJPEG	20	23.8	11.8	7.8
LI	3	4.8	18.0	8.9

PREG branch prediction is based indirectly on predicting the data values the branch depends on. That is, an older, committed data value plus the pending count is used as a *de facto* prediction of the real data value that is not yet available. To get an idea of how accurate these "stand-in" values are, we used them to predict the value they represent -- the value actually tested by the branch. In the simulator, we implemented a data value predictor alongside the branch predictor. This data value predictor was only used to collect data, and did not affect the simulations. Each time we predicted a branch in the simulation, we also predicted the data value the branch was testing, by using the PREG VALUE and CNT to index an interference-free data value prediction table.

The results in Table 5 indicate that an improved data value prediction scheme is needed to lower branch miss rates. Column 1 shows the misprediction rate for the data value predictor. Compared with other studies [SAZ97] these results are not exceptional. Columns 2 through 4 show the direct effect data value prediction has on branch prediction. For comparison, Column 2 shows the misprediction rate for all PREG predicted branches. Column 3 demonstrates the the misprediction rate is low, less than 1.5% for all benchmarks but *go*, when the data value is correctly predicted. However when data value is mispredicted (Column 4), the branch prediction is, in essence, based on faulty information and suffers greatly. The final column shows that 82.9 to 96.2% of the remaining branch mispredictions for PREG predicted branches are correlated with data value mispredictions. Improved data value prediction has the potential to reduce these misses, greatly increasing the effectiveness of PREG prediction.

According to [SAZ97], a context predictor containing sequences of data values should aid prediction. However, in our scheme this could be hampered when multiple instructions updating the same PREG. In this case, additional context values are likely to come from instructions related to other branches, unrelated to this branch, and only weakly improve the context. This situation occurs often when SET instructions are

Table 5. Correlation of data prediction and branch prediction

Benchmark	Data Value Mispredict Rate (%)	Branch Mispredict Rate (%)			% Mispredicts correlated w/ Value Mispredict
		All Branches	Value Prediction Correct	Value Prediction Incorrect	
COMPRESS	49.4	8.0	0.6	15.0	96.2
GCC	12.9	4.5	0.8	28.9	82.9
GO	39.7	9.0	3.5	17.3	76.4
IJPEG	27.7	7.5	1.0	24.6	90.6
LI	36.3	6.6	1.4	15.9	86.9

used to evaluate the branch condition. The compiler that compiled the benchmarks, gcc, tends to use integer register $2 as a temporary register for the results of set instructions. Thus most SET instructions update a single PREG, PREG $2, and most branches are using SET instructions. More research is needed to improve this case.

Finally, we considered the range of CNT values. In the previous simulations we used six bits for the CNT value. We find that three bits are sufficient. Fig. 3 shows a histogram of the six bit count values dynamically encountered by PREG predicted branches. The results shown are for *compress*. All other benchmarks have similar or smaller CNT values. The CNT value rarely ranges above 7, and never above 15.

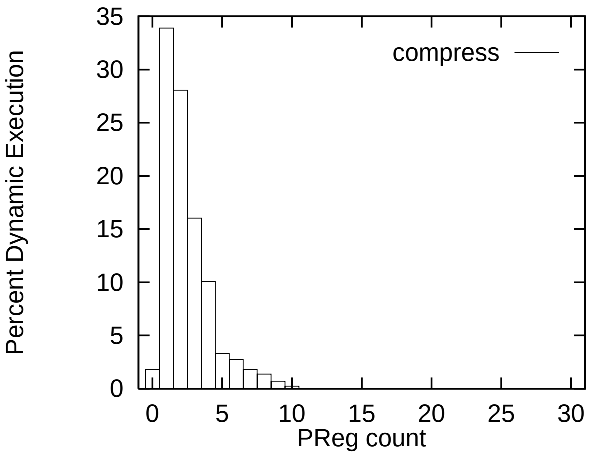


Fig. 3. Values taken by CNT field in PREG entries.

5. Prediction Tables with Interference

Our second set of simulations uses the predictor proposed in Fig. 1 to explore the effects of interference. In accordance with [JUA98, SEC96], we found that, in general, gshare predictors should use fewer bits of global history than the number of bits used to index the table. We chose a table with 128K 2-bit counters, and found that 12 or so bits of global history will give good performance. Table 6 shows the results.

Table 6. Performance with table interference

Benchmark	GSHARE 12-bit		PREG 12_08_03				PREG 12_08_03 w/ Threshold Filter				Improvement Tested v. GSHARE	
			Test		Loop		Test		Loop			
	Miss	IPC	Miss	IPC	Miss	IPC	Miss	IPC	Miss	IPC	Miss	IPC
COMPRESS	10.13	1.40	8.78	1.44	10.13	1.40	8.78	1.44	10.13	1.40	13.33	2.92
GCC	7.42	1.29	8.00	1.27	7.67	1.29	7.42	1.29	7.44	1.29	0.0	0.0
GO	12.7	1.20	13.9	1.18	13.3	1.19	13.4	1.19	13.06	1.19	-5.1	-0.9
IJPEG	7.52	3.53	6.87	3.58	7.51	3.53	6.77	3.59	7.52	3.53	9.97	1.75
LI	3.33	3.15	3.41	3.13	3.18	3.18	3.18	3.16	3.18	3.18	4.5	0.33

The initial results (columns 3 and 4) indicate that PREG prediction increased the amount of interference in the bounded table to the point where performance no longer improved. To solve this problem, we filtered more tightly the branches for which PREG prediction was enabled. Rather then taking any branch for which there was an improvement, however slight, we placed a threshold on the amount of improvement required. For each branch PREG prediction was enabled only if profiling indicated it would eliminate 0.02% or more of the total mispredictions in the benchmark. This made definite improvements in the misprediction rate. Table 6 shows the improvement in misprediction rate and IPC between tested value with filtering, and gshare. All benchmarks improved, except for *gcc*, which broke even, and *go*, which still becomes worse.

We also notice that the loop value heuristic never performs as well as the tested value heuristic. We expected combining the two heuristics, selecting the best heuristic for each branch, to further increase performance. However, we obtained poor results using tables with interference. To understand why, we returned again to interference-free predictor tables. Table 7 shows these results.

Table 7. Performance with Combined Heuristics

Benchmark	Test heuristic	G12_V08_C03 no inter. Loop heuristic	Combined	Improvement Combined vs. Test
COMPRESS	8.7	10.1	8.7	0.0
GCC	4.2	4.5	4.1	1.4
GO	9.0	9.3	8.8	1.9
IJPEG	5.8	6.7	5.8	0.9
LI	3.1	3.1	2.9	5.8

Combining the two heuristics did provide a slight improvement with unbounded tables, but increased interference in bounded tables swamped the improvement. Examining individual branches, we could find few branches where both the loop value heuristic did well and the tested value heuristic did poorly. Table 7 shows the number of static branches for which the loop value heuristic eliminated more mispredicts than the tested value heuristic by at least 0.02% of the total mispredicts. The tested value heuristic appears to subsume the loop value heuristic for all benchmarks, except perhaps LI.

6. Conclusions

The bottom line is that by using the proposed prediction method based on data values significant improvements are possible for certain branches, but overall improvements are modest. The success we have had on certain previously difficult branches is tantalizing, and this motivates us to further study.

We have only scratched the surface of possible predictor designs. This study started with the simple gshare predictor, and made relatively small modifications. The variety of possible hashing functions have only been lightly examined, and a wide array of interference-reduction techniques developed for branch predictors [MIC97, MCF93, CHA96, SPR97] have not yet been applied, even though interference is an important aspect in of data-value based predictors.

Because the loop value heuristic is subsumed by the tested value heuristic, our future research will focus on variations of the tested value heuristic. Possibly, prediction could be based on values generated farther up the dependency chain from the branch, rather then the value directly tested. This research will be driven by re-examining individual branches still not helped by PREG prediction.

The results also indicate that PREGs need to be handled better. A general solution for branches with two input values needs to be developed. Also, a tighter connection should be maintained between the instruction producing the value and the branch using the value for prediction. The cross-talk caused by multiple instructions updating a single PREG hampers branch prediction. Eliminating this cross-talk would enable contexts to contain sequences of data values. This should improve the data prediction rate [SAZ97], which should aid the prediction process.

This initial study has left us confident that continued innovations will produce improved overall results. Using data values provides a new direction for increased branch prediction accuracy, but much additional research remains.

Acknowledgments

The authors would like to acknowledge Rushan Chen for her participation in the initial investigation of hard-to-predict branches. This work was supported in part by NSF Grant MIP-9505853 and by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Army Intelligence Center and Fort Huachuca, or the U.S. Government.

References

[BUR97] Douglas C. Burger, Todd M. Austin, "The SimpleScalar Tool Set, Version 2.0," Univ. of Wisconsin - Madison Comp. Sci. Dept. Technical Report #1342, June 1997.

[CHA95] Po-Yung Chang, Eric Hao, Yale Patt, "Alternative implementations of hybrid branch predictors," In *Proc. of the 28th Intl. Symp. on Microarch.,* November 1995.

[CHA96] Po-Yung Chang, Marius Evers, Yale Patt, "Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference," *Intl. Conf. on Parallel Arch. and Compilation Techniques,* October 1996.

[EVE98] Marius Evers, Sanjay J. Patel, Robert S. Chappell, Yale N. Patt, "An analysis of correlation and predictability: What makes two-level branch predictors work," In *Proc. of the 25th Intl. Symp. on Comp. Arch.,* Barcelona, Spain, June 1998.

[FAR98] Alexandre Farcy, Olivier Temam, Roger Espasa, Toni Juan, "Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes," To appear in *Proc. of the 31th Intl. Symp. on Microarch.,* 1999.

[JUA98] Toni Juan, Sanji Sanjeevan, Juan J. Navarri, "Dynamic history-length fitting: A third-level of adaptivity for branch-prediction," In *Proc. of the 25th Intl. Symp. on Comp. Arch.,* June 1998.

[KES98] R.E.Kessler, "The Alpha 21264 Microprocessor: Out-Of-Order Execution at 600MHz," Presented at *Hot Chips 10,* August 1998.

[LEE97] C.-C. Lee, I.-C.K. Chen, and T.N. Mudge, "The Bi-Mode Branch Predictor," In *Proc. of the 24th Intl. Symp. on Microarch.,* pp. 4-13, December 1997.

[MAH96] Scott Mahlke, Balas Natarajan, "Compiler Synthesized Dynamic Branch Prediction," *IEEE* 1996

[MCF93] Scott McFarling, “Combining branch predictors,” *Technical Report TN-36*, Digital WRL, June 1993

[MIC97] Pierre Michaud, Andre Seznec, Richard Uhlig, “Trading conflict and capacity aliasing in conditional branch predictors,” In *Proc. of the 24th Intl. Symp. on Comp. Arch.*, June 1997.

[NAI95] R. Nair, “Dynamic Path-Based Branch Correlation,” In *Proc. of the 28th Intl. Symp. on Microarch.*, pp. 15-23, November 1995.

[PAN92] S. T. Pan, K. So, J.T. Rahmeh, “Improving the Accuracy of Dynamic Branch Prediction using Branch Correlation,” In *Proc. of the 5th Intl. Conf. on Arch. Support for Prog. Lang. and Op. Sys.*, October 1992.

[SAZ97] Yiannakis Sazeides, James E. Smith, “The Predictability of Data Values,” *Thirtieth Intl. Symp. on Microarch.*, pp. 248-257, December 1997.

[SAZ98] Y. Sazeides, J. Smith, “Modeling Program Predictability,” In *Proc. of the 25th Intl. Symp. on Comp. Arch.*, pp. 73-84, June 1998.

[SEC96] S. Sechrest, C-C Lee, Trevor Mudge, “Correlation and Aliasing in Dynamic Branch Predictors,” In *Proc. of the 23rd Intl. Symp. on Comp. Arch.*, pp. 22-32, May 1996.

[SMI81] J. E. Smith, “A study of branch prediction strategies,” In *Proc. of the Eigth Intl. Symp. on Comp. Arch.*, pp. 135-148, 1981.

[SPR97] E Sprangle, Robert Chappell, M Alsup, Yale Patt “The Agree Prediction: A Mechanism for Reducing Negative Branch History Interference,” In *Proc. of the 24th Intl. Symp. on Comp. Arch.*, May 1997

[YEH91] T-Y Yeh, Yale Patt, “Two-Level Adaptive Training Branch Prediction” In *Proc. of the 24th Intl. Symp. on Microarch.*, Nov 1991.

[YEH93] T-Y Yeh, Yale Patt, “A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History,” In *Proc. of the 20th Intl. Symp. on Comp. Arch.*, May 1993.

Appendix B

cc-train Top 25 Misses

```
 **** #1 miss at 0050c008: cse.c:1530
   0050bfd8:                       : <invalidate+708> addiu $s2[18],$s2[18],16768
   cse.c:1526
   0050bfe0:                       : <invalidate+710> lw $a0[4],0($s2[18])
   0050bfe8:  (     196176,   1.24): <invalidate+718> beq $a0[4],$zero[0],0050c0c0 <invalidate+7f0>
   cse.c:1530
   0050bff0:                       : <invalidate+720> lw $v1[3],0($a0[4])
   cse.c:1528
   0050bff8:                       : <invalidate+728> lw $s0[16],4($a0[4])
   cse.c:1530
   0050c000:                       : <invalidate+730> lhu $v0[2],0($v1[3])
** 0050c008:  (     312230,   1.97): <invalidate+738> bne $v0[2],$s6[22],0050c0b0 <invalidate+7e0>
   0050c010:                       : <invalidate+740> lw $a1[5],4($v1[3])
   0050c018:                       : <invalidate+748> slti $v0[2],$a1[5],64
   0050c020:  (      59019,   0.37): <invalidate+750> beq $v0[2],$zero[0],0050c0b0 <invalidate+7e0>
   cse.c:1535
   0050c028:                       : <invalidate+758> lw $v0[2],0($v1[3])
   0050c030:                       : <invalidate+760> sll $v0[2],$v0[2],0x10
   0050c038:                       : <invalidate+768> sra $v0[2],$v0[2],0x18
   0050c040:                       : <invalidate+770> sll $v0[2],$v0[2],0x2
   0050c048:                       : <invalidate+778> addu $v0[2],$v0[2],$s5[21]


---- Source Extract [cse.c] [1530]
---- (cse.c)
 1518:        in_table |= TEST_HARD_REG_BIT (hard_regs_in_table, i);
 1519:        CLEAR_HARD_REG_BIT (hard_regs_in_table, i);
 1520:        delete_reg_equiv (i);
 1521:        reg_tick[i]++;
 1522:      }
 1523:
 1524:    if (in_table)
 1525:      for (hash = 0; hash < NBUCKETS; hash++)
 1526:        for (p = table[hash]; p; p = next)
 1527:  {
 1528:    next = p->next_same_hash;
 1529:
*1530:    if (GET_CODE (p->exp) != REG
 1531:        || REGNO (p->exp) >= FIRST_PSEUDO_REGISTER)
 1532:      continue;
 1533:
 1534:    tregno = REGNO (p->exp);
 1535:    tendregno
 1536:      = tregno + HARD_REGNO_NREGS (tregno, GET_MODE (p->exp));
 1537:    if (tendregno > regno && tregno < endregno)
 1538:    remove_from_table (p, hash);
 1539:  }
 1540:  }
 1541:
 1542:       return;


 **** #2 miss at 0050c0b8: cse.c:1526
   cse.c:1537
   0050c080:                       : <invalidate+7b0> slt $v0[2],$s3[19],$v0[2]
   0050c088:  (      22149,   0.14): <invalidate+7b8> beq $v0[2],$zero[0],0050c0b0 <invalidate+7e0>
   0050c090:                       : <invalidate+7c0> slt $v0[2],$a1[5],$s4[20]
   0050c098:  (      16392,   0.10): <invalidate+7c8> beq $v0[2],$zero[0],0050c0b0 <invalidate+7e0>
   cse.c:1538
   0050c0a0:                       : <invalidate+7d0> addu $a1[5],$zero[0],$s1[17]
   0050c0a8:                       : <invalidate+7d8> jal 0050a390 <remove_from_table>
   cse.c:1526
   0050c0b0:                       : <invalidate+7e0> addu $a0[4],$zero[0],$s0[16]
** 0050c0b8:  (     259900,   1.64): <invalidate+7e8> bne $a0[4],$zero[0],0050bff0 <invalidate+720>
```

```
   cse.c:1525
   0050c0c0:                        : <invalidate+7f0> addiu $s2[18],$s2[18],4
   0050c0c8:                        : <invalidate+7f8> addiu $s1[17],$s1[17],1
   0050c0d0:                        : <invalidate+800> slti $v0[2],$s1[17],31
   0050c0d8:  (      30349,   0.19): <invalidate+808> bne $v0[2],$zero[0],0050bfe0 <invalidate+710>
   cse.c:1542
   0050c0e0:                        : <invalidate+810> j 0050c250 <invalidate+980>
   cse.c:1545
   0050c0e8:                        : <invalidate+818> addiu $v0[2],$zero[0],54


---- Source Extract [cse.c] [1526]
---- (cse.c)
 1514:    CLEAR_HARD_REG_BIT (hard_regs_in_table, regno);
 1515:
 1516:    for (i = regno + 1; i < endregno; i++)
 1517:      {
 1518:        in_table |= TEST_HARD_REG_BIT (hard_regs_in_table, i);
 1519:        CLEAR_HARD_REG_BIT (hard_regs_in_table, i);
 1520:        delete_reg_equiv (i);
 1521:        reg_tick[i]++;
 1522:      }
 1523:
 1524:    if (in_table)
 1525:      for (hash = 0; hash < NBUCKETS; hash++)
*1526:        for (p = table[hash]; p; p = next)
 1527:  {
 1528:    next = p->next_same_hash;
 1529:
 1530:    if (GET_CODE (p->exp) != REG
 1531:        || REGNO (p->exp) >= FIRST_PSEUDO_REGISTER)
 1532:      continue;
 1533:
 1534:    tregno = REGNO (p->exp);
 1535:    tendregno
 1536:      = tregno + HARD_REGNO_NREGS (tregno, GET_MODE (p->exp));
 1537:    if (tendregno > regno && tregno < endregno)
 1538:    remove_from_table (p, hash);


 **** #3 miss at 0050bfe8: cse.c:1526
   0050bfa8:  (      17354,   0.11): <invalidate+6d8> beq $t4[12],$zero[0],0050c250 <invalidate+980>
   cse.c:1525
   0050bfb0:                        : <invalidate+6e0> addu $s1[17],$zero[0],$zero[0]
   0050bfb8:                        : <invalidate+6e8> addiu $s6[22],$zero[0],52
   0050bfc0:                        : <invalidate+6f0> lui $s5[21],4099
   0050bfc8:                        : <invalidate+6f8> addiu $s5[21],$s5[21],-17788
   0050bfd0:                        : <invalidate+700> lui $s2[18],4099
   0050bfd8:                        : <invalidate+708> addiu $s2[18],$s2[18],16768
   cse.c:1526
   0050bfe0:                        : <invalidate+710> lw $a0[4],0($s2[18])
** 0050bfe8:  (     196176,   1.24): <invalidate+718> beq $a0[4],$zero[0],0050c0c0 <invalidate+7f0>
   cse.c:1530
   0050bff0:                        : <invalidate+720> lw $v1[3],0($a0[4])
   cse.c:1528
   0050bff8:                        : <invalidate+728> lw $s0[16],4($a0[4])
   cse.c:1530
   0050c000:                        : <invalidate+730> lhu $v0[2],0($v1[3])
   0050c008:  (     312230,   1.97): <invalidate+738> bne $v0[2],$s6[22],0050c0b0 <invalidate+7e0>
   0050c010:                        : <invalidate+740> lw $a1[5],4($v1[3])
   0050c018:                        : <invalidate+748> slti $v0[2],$a1[5],64


---- Source Extract [cse.c] [1526]
---- (cse.c)
 1514:    CLEAR_HARD_REG_BIT (hard_regs_in_table, regno);
 1515:
 1516:    for (i = regno + 1; i < endregno; i++)
 1517:      {
 1518:        in_table |= TEST_HARD_REG_BIT (hard_regs_in_table, i);
 1519:        CLEAR_HARD_REG_BIT (hard_regs_in_table, i);
 1520:        delete_reg_equiv (i);
 1521:        reg_tick[i]++;
 1522:      }
 1523:
 1524:    if (in_table)
 1525:      for (hash = 0; hash < NBUCKETS; hash++)
*1526:        for (p = table[hash]; p; p = next)
 1527:  {
 1528:    next = p->next_same_hash;
 1529:
 1530:    if (GET_CODE (p->exp) != REG
 1531:        || REGNO (p->exp) >= FIRST_PSEUDO_REGISTER)
 1532:      continue;
 1533:
 1534:    tregno = REGNO (p->exp);
 1535:    tendregno
 1536:      = tregno + HARD_REGNO_NREGS (tregno, GET_MODE (p->exp));
 1537:    if (tendregno > regno && tregno < endregno)
 1538:    remove_from_table (p, hash);
```

```
 **** #4 miss at 0050c9b8: cse.c:1711
   0050c988:                        : <invalidate_for_call+258> addiu $s2[18],$s2[18],16768
   cse.c:1707
   0050c990:                        : <invalidate_for_call+260> lw $a0[4],0($s2[18])
   0050c998:  (      70733,   0.45): <invalidate_for_call+268> beq $a0[4],$zero[0],0050cad0 <invalidate_for_call+3a0>
   cse.c:1711
   0050c9a0:                        : <invalidate_for_call+270> lw $a1[5],0($a0[4])
   cse.c:1709
   0050c9a8:                        : <invalidate_for_call+278> lw $s1[17],4($a0[4])
   cse.c:1711
   0050c9b0:                        : <invalidate_for_call+280> lhu $v0[2],0($a1[5])
** 0050c9b8:  (     117492,   0.74): <invalidate_for_call+288> bne $v0[2],$s5[21],0050cac0 <invalidate_for_call+390>
   0050c9c0:                        : <invalidate_for_call+290> lw $v1[3],4($a1[5])
   0050c9c8:                        : <invalidate_for_call+298> slti $v0[2],$v1[3],64
   0050c9d0:  (      21832,   0.14): <invalidate_for_call+2a0> beq $v0[2],$zero[0],0050cac0 <invalidate_for_call+390>
   cse.c:1716
   0050c9d8:                        : <invalidate_for_call+2a8> lw $v0[2],0($a1[5])
   0050c9e0:                        : <invalidate_for_call+2b0> sll $v0[2],$v0[2],0x10
   0050c9e8:                        : <invalidate_for_call+2b8> sra $v0[2],$v0[2],0x18
   0050c9f0:                        : <invalidate_for_call+2c0> sll $v0[2],$v0[2],0x2
   0050c9f8:                        : <invalidate_for_call+2c8> addu $v0[2],$v0[2],$s4[20]
```

```
---- Source Extract [cse.c] [1711]
---- (cse.c)
 1699:       }
 1700:
 1701:   /* In the case where we have no call-clobbered hard registers in the
 1702:      table, we are done.  Otherwise, scan the table and remove any
 1703:      entry that overlaps a call-clobbered register.  */
 1704:
 1705:   if (in_table)
 1706:     for (hash = 0; hash < NBUCKETS; hash++)
 1707:       for (p = table[hash]; p; p = next)
 1708:  {
 1709:    next = p->next_same_hash;
 1710:
*1711:    if (GET_CODE (p->exp) != REG
 1712:        || REGNO (p->exp) >= FIRST_PSEUDO_REGISTER)
 1713:      continue;
 1714:
 1715:    regno = REGNO (p->exp);
 1716:    endregno = regno + HARD_REGNO_NREGS (regno, GET_MODE (p->exp));
 1717:
 1718:    for (i = regno; i < endregno; i++)
 1719:      if (TEST_HARD_REG_BIT (regs_invalidated_by_call, i))
 1720:        {
 1721:  remove_from_table (p, hash);
 1722:  break;
 1723:        }
```

```
 **** #5 miss at 0051fa88: cse.c:6852
   0051fa38:                        : <cse_insn+4b80> beq $s4[20],$zero[0],0051fa50 <cse_insn+4b98>
   0051fa40:                        : <cse_insn+4b88> lb $v0[2],37($s0[16])
   0051fa48:                        : <cse_insn+4b90> bne $v0[2],$zero[0],0051fa68 <cse_insn+4bb0>
   0051fa50:                        : <cse_insn+4b98> lw $a0[4],0($s0[16])
   0051fa58:                        : <cse_insn+4ba0> jal 0050e7d8 <cse_rtx_addr_varies_p>
   0051fa60:                        : <cse_insn+4ba8> beq $v0[2],$zero[0],0051fa80 <cse_insn+4bc8>
   0051fa68:                        : <cse_insn+4bb0> addu $a0[4],$zero[0],$s0[16]
   0051fa70:                        : <cse_insn+4bb8> addu $a1[5],$zero[0],$s1[17]
   0051fa78:                        : <cse_insn+4bc0> jal 0050a390 <remove_from_table>
   0051fa80:                        : <cse_insn+4bc8> addu $s0[16],$zero[0],$s2[18]
** 0051fa88:  (     108471,   0.68): <cse_insn+4bd0> bne $s0[16],$zero[0],0051fa18 <cse_insn+4b60>
   0051fa90:                        : <cse_insn+4bd8> addiu $s3[19],$s3[19],4
   0051fa98:                        : <cse_insn+4be0> addiu $s1[17],$s1[17],1
   0051faa0:                        : <cse_insn+4be8> slti $v0[2],$s1[17],31
   0051faa8:  (      11888,   0.07): <cse_insn+4bf0> bne $v0[2],$zero[0],0051fa08 <cse_insn+4b50>
   cse.c:6853
   0051fab0:                        : <cse_insn+4bf8> jal 0050c730 <invalidate_for_call>
   cse.c:6861
   0051fab8:                        : <cse_insn+4c00> lw $t1[9],84($s8[30])
   0051fac0:                        : <cse_insn+4c08> sw $zero[0],76($s8[30])
```

```
---- Source Extract [cse.c] [6852]
---- (cse.c)
 6840:       sets[i].src_elt = src_eqv_elt;
 6841:
 6842:   invalidate_from_clobbers (&writes_memory, x);
 6843:
 6844:   /* Some registers are invalidated by subroutine calls.  Memory is
 6845:      invalidated by non-constant calls.  */
 6846:
 6847:   if (GET_CODE (insn) == CALL_INSN)
 6848:     {
 6849:       static struct write_data everything = {0, 1, 1, 1};
 6850:
 6851:       if (! CONST_CALL_P (insn))
*6852:  invalidate_memory (&everything);
 6853:       invalidate_for_call ();
 6854:     }
```

```
 6855:
 6856:   /* Now invalidate everything set by this instruction.
 6857:      If a SUBREG or other funny destination is being set,
 6858:      sets[i].rtl is still nonzero, so here we invalidate the reg
 6859:      a part of which is being set.  */
 6860:
 6861:   for (i = 0; i < n_sets; i++)
 6862:     if (sets[i].rtl)
 6863:       {
 6864:  register rtx dest = sets[i].inner_dest;


 **** #6 miss at 004d6f38: rtlanal.c:930
   004d6f08:                        : <rtx_equal_p+48> sw $s0[16],24($sp[29])
   rtlanal.c:923
   004d6f10:  (      64481,   0.41): <rtx_equal_p+50> beq $a0[4],$a1[5],004d7270 <rtx_equal_p+3b0>
   rtlanal.c:925
   004d6f18:  (       1225,   0.01): <rtx_equal_p+58> beq $a0[4],$zero[0],004d7120 <rtx_equal_p+260>
   004d6f20:  (        657,   0.00): <rtx_equal_p+60> beq $a1[5],$zero[0],004d7120 <rtx_equal_p+260>
   rtlanal.c:928
   004d6f28:                        : <rtx_equal_p+68> lhu $a2[6],0($a0[4])
   rtlanal.c:930
   004d6f30:                        : <rtx_equal_p+70> lhu $v0[2],0($a1[5])
** 004d6f38:  (     102417,   0.65): <rtx_equal_p+78> bne $a2[6],$v0[2],004d7120 <rtx_equal_p+260>
   rtlanal.c:936
   004d6f40:                        : <rtx_equal_p+80> lb $v1[3],2($a0[4])
   004d6f48:                        : <rtx_equal_p+88> lb $v0[2],2($a1[5])
   004d6f50:  (      16302,   0.10): <rtx_equal_p+90> bne $v1[3],$v0[2],004d7120 <rtx_equal_p+260>
   rtlanal.c:941
   004d6f58:                        : <rtx_equal_p+98> addiu $v0[2],$zero[0],52
   004d6f60:  (      16244,   0.10): <rtx_equal_p+a0> bne $a2[6],$v0[2],004d6fd8 <rtx_equal_p+118>
   rtlanal.c:946
   004d6f68:                        : <rtx_equal_p+a8> lw $v1[3],4($a0[4])


---- Source Extract [rtlanal.c] [930]
---- (rtlanal.c)
 918:   register int i;
 919:   register int j;
 920:   register enum rtx_code code;
 921:   register char *fmt;
 922:
 923:   if (x == y)
 924:     return 1;
 925:   if (x == 0 || y == 0)
 926:     return 0;
 927:
 928:   code = GET_CODE (x);
 929:   /* Rtx's of different codes cannot be equal.  */
*930:   if (code != GET_CODE (y))
 931:     return 0;
 932:
 933:   /* (MULT:SI x y) and (MULT:HI x y) are NOT equivalent.
 934:      (REG:SI x) and (REG:HI x) are NOT equivalent.  */
 935:
 936:   if (GET_MODE (x) != GET_MODE (y))
 937:     return 0;
 938:
 939:   /* REG, LABEL_REF, and SYMBOL_REF can be compared nonrecursively.  */
 940:
 941:   if (code == REG)
 942:     /* Until rtl generation is complete, don't consider a reference to the


 **** #7 miss at 0050ea90: cse.c:2415
   cse.c:2411
   0050ea60:  (        131,   0.00): <canon_reg+68> bne $s5[21],$zero[0],0050ea78 <canon_reg+80>
   cse.c:2412
   0050ea68:                        : <canon_reg+70> addu $v0[2],$zero[0],$zero[0]
   0050ea70:                        : <canon_reg+78> j 0050ee30 <canon_reg+438>
   cse.c:2414
   0050ea78:                        : <canon_reg+80> lhu $a0[4],0($s5[21])
   cse.c:2415
   0050ea80:                        : <canon_reg+88> addiu $v1[3],$a0[4],-39
   0050ea88:                        : <canon_reg+90> sltiu $v0[2],$v1[3],22
** 0050ea90:  (     101943,   0.64): <canon_reg+98> beq $v0[2],$zero[0],0050ebd0 <canon_reg+1d8>
   0050ea98:                        : <canon_reg+a0> sll $v0[2],$v1[3],0x2
   0050eaa0:                        : <canon_reg+a8> lui $at[1],4097
   0050eaa8:                        : <canon_reg+b0> addu $at[1],$at[1],$v0[2]
   0050eab0:                        : <canon_reg+b8> lw $v0[2],9656($at[1])
   0050eab8:                        : <canon_reg+c0> jr $v0[2]
   cse.c:2437
   0050eac0:                        : <canon_reg+c8> lw $a0[4],4($s5[21])
   0050eac8:                        : <canon_reg+d0> slti $v0[2],$a0[4],64
   0050ead0:  (      37009,   0.23): <canon_reg+d8> bne $v0[2],$zero[0],0050ee28 <canon_reg+430>


---- Source Extract [cse.c] [2415]
---- (cse.c)
 2403: canon_reg (x, insn)
 2404:      rtx x;
 2405:      rtx insn;
```

```
 2406: {
 2407:   register int i;
 2408:   register enum rtx_code code;
 2409:   register char *fmt;
 2410:
 2411:   if (x == 0)
 2412:     return x;
 2413:
 2414:   code = GET_CODE (x);
*2415:   switch (code)
 2416:     {
 2417:     case PC:
 2418:     case CC0:
 2419:     case CONST:
 2420:     case CONST_INT:
 2421:     case CONST_DOUBLE:
 2422:     case SYMBOL_REF:
 2423:     case LABEL_REF:
 2424:     case ADDR_VEC:
 2425:     case ADDR_DIFF_VEC:
 2426:       return x;
 2427:


 **** #8 miss at 0050cac8: cse.c:1707
   0050ca90:                       : <invalidate_for_call+360> addu $a1[5],$zero[0],$s0[16]
   0050ca98:                       : <invalidate_for_call+368> jal 0050a390 <remove_from_table>
   cse.c:1722
   0050caa0:                       : <invalidate_for_call+370> j 0050cac0 <invalidate_for_call+390>
   cse.c:1718
   0050caa8:                       : <invalidate_for_call+378> addiu $a1[5],$a1[5],1
   0050cab0:                       : <invalidate_for_call+380> slt $v0[2],$a1[5],$a2[6]
   0050cab8:  (        156,   0.00): <invalidate_for_call+388> bne $v0[2],$zero[0],0050ca50 <invalidate_for_call+320>
   cse.c:1707
   0050cac0:                       : <invalidate_for_call+390> addu $a0[4],$zero[0],$s1[17]
** 0050cac8:  (      99858,   0.63): <invalidate_for_call+398> bne $a0[4],$zero[0],0050c9a0 <invalidate_for_call+270>
   cse.c:1706
   0050cad0:                       : <invalidate_for_call+3a0> addiu $s2[18],$s2[18],4
   0050cad8:                       : <invalidate_for_call+3a8> addiu $s0[16],$s0[16],1
   0050cae0:                       : <invalidate_for_call+3b0> slti $v0[2],$s0[16],31
   0050cae8:  (       9637,   0.06): <invalidate_for_call+3b8> bne $v0[2],$zero[0],0050c990 <invalidate_for_call+260>
   cse.c:1725
   0050caf0:                       : <invalidate_for_call+3c0> lw $ra[31],40($sp[29])
   0050caf8:                       : <invalidate_for_call+3c8> lw $s5[21],36($sp[29])
   0050cb00:                       : <invalidate_for_call+3d0> lw $s4[20],32($sp[29])


---- Source Extract [cse.c] [1707]
---- (cse.c)
 1695:  if (reg_tick[regno] >= 0)
 1696:    reg_tick[regno]++;
 1697:
 1698:  in_table |= TEST_HARD_REG_BIT (hard_regs_in_table, regno);
 1699:       }
 1700:
 1701:   /* In the case where we have no call-clobbered hard registers in the
 1702:      table, we are done.  Otherwise, scan the table and remove any
 1703:      entry that overlaps a call-clobbered register.  */
 1704:
 1705:   if (in_table)
 1706:     for (hash = 0; hash < NBUCKETS; hash++)
*1707:       for (p = table[hash]; p; p = next)
 1708:  {
 1709:    next = p->next_same_hash;
 1710:
 1711:    if (GET_CODE (p->exp) != REG
 1712:        || REGNO (p->exp) >= FIRST_PSEUDO_REGISTER)
 1713:      continue;
 1714:
 1715:    regno = REGNO (p->exp);
 1716:    endregno = regno + HARD_REGNO_NREGS (regno, GET_MODE (p->exp));
 1717:
 1718:    for (i = regno; i < endregno; i++)
 1719:      if (TEST_HARD_REG_BIT (regs_invalidated_by_call, i))


 **** #9 miss at 00508fc8: cse.c:680
   00508f90:                       : <rtx_cost+40> sw $s3[19],36($sp[29])
   00508f98:                       : <rtx_cost+48> sw $s2[18],32($sp[29])
   00508fa0:                       : <rtx_cost+50> sw $s1[17],28($sp[29])
   cse.c:673
   00508fa8:  (        987,   0.01): <rtx_cost+58> beq $s0[16],$zero[0],00509358 <rtx_cost+408>
   cse.c:679
   00508fb0:                       : <rtx_cost+60> lhu $s5[21],0($s0[16])
   cse.c:680
   00508fb8:                       : <rtx_cost+68> addiu $v1[3],$s5[21],-36
   00508fc0:                       : <rtx_cost+70> sltiu $v0[2],$v1[3],37
** 00508fc8:  (      91200,   0.57): <rtx_cost+78> beq $v0[2],$zero[0],00509070 <rtx_cost+120>
   00508fd0:                       : <rtx_cost+80> sll $v0[2],$v1[3],0x2
   00508fd8:                       : <rtx_cost+88> lui $at[1],4097
   00508fe0:                       : <rtx_cost+90> addu $at[1],$at[1],$v0[2]
   00508fe8:                       : <rtx_cost+98> lw $v0[2],8816($at[1])
```

```
   00508ff0:                        : <rtx_cost+a0> jr $v0[2]
   cse.c:685
   00508ff8:                        : <rtx_cost+a8> lw $a0[4],8($s0[16])
   00509000:                        : <rtx_cost+b0> lhu $v1[3],0($a0[4])
   00509008:                        : <rtx_cost+b8> addiu $v0[2],$zero[0],47


---- Source Extract [cse.c] [680]
---- (cse.c)
 668:   register int i, j;
 669:   register enum rtx_code code;
 670:   register char *fmt;
 671:   register int total;
 672:
 673:   if (x == 0)
 674:     return 0;
 675:
 676:   /* Compute the default costs of certain things.
 677:      Note that RTX_COSTS can override the defaults.  */
 678:
 679:   code = GET_CODE (x);
*680:   switch (code)
 681:     {
 682:     case MULT:
 683:       /* Count multiplication by 2**n as a shift,
 684:   because if we are considering it, we would output it as a shift.  */
 685:       if (GET_CODE (XEXP (x, 1)) == CONST_INT
 686:    && exact_log2 (INTVAL (XEXP (x, 1))) >= 0)
 687:  total = 2;
 688:       else
 689:  total = COSTS_N_INSNS (5);
 690:       break;
 691:     case DIV:
 692:     case UDIV:


 **** #10 miss at 004d7c98: rtlanal.c:1202
   find_reg_note():
   rtlanal.c:1202
   004d7c90:                        : <find_reg_note> lw $v1[3],28($a0[4])
** 004d7c98:  (      76725,   0.48): <find_reg_note+8> beq $v1[3],$zero[0],004d7cf8 <find_reg_note+68>
   rtlanal.c:1203
   004d7ca0:                        : <find_reg_note+10> lw $v0[2],0($v1[3])
   004d7ca8:                        : <find_reg_note+18> sll $v0[2],$v0[2],0x10
   004d7cb0:                        : <find_reg_note+20> sra $v0[2],$v0[2],0x18
   004d7cb8:  (      34647,   0.22): <find_reg_note+28> bne $v0[2],$a1[5],004d7ce8 <find_reg_note+58>
   004d7cc0:  (       4490,   0.03): <find_reg_note+30> beq $a2[6],$zero[0],004d7cd8 <find_reg_note+48>
   004d7cc8:                        : <find_reg_note+38> lw $v0[2],4($v1[3])
   004d7cd0:  (       5226,   0.03): <find_reg_note+40> bne $a2[6],$v0[2],004d7ce8 <find_reg_note+58>
   rtlanal.c:1205


---- Source Extract [rtlanal.c] [1202]
---- (rtlanal.c)
 1190:
 1191: /* Return the reg-note of kind KIND in insn INSN, if there is one.
 1192:    If DATUM is nonzero, look for one whose datum is DATUM.  */
 1193:
 1194: rtx
 1195: find_reg_note (insn, kind, datum)
 1196:      rtx insn;
 1197:      enum reg_note kind;
 1198:      rtx datum;
 1199: {
 1200:   register rtx link;
 1201:
*1202:   for (link = REG_NOTES (insn); link; link = XEXP (link, 1))
 1203:     if (REG_NOTE_KIND (link) == kind
 1204:  && (datum == 0 || datum == XEXP (link, 0)))
 1205:       return link;
 1206:   return 0;
 1207: }
 1208:
 1209: /* Return the reg-note of kind KIND in insn INSN which applies to register
 1210:    number REGNO, if any.  Return 0 if there is no such reg-note.  Note that
 1211:    the REGNO of this NOTE need not be REGNO if REGNO is a hard register;
 1212:    it might be the case that the note overlaps REGNO.  */
 1213:
 1214: rtx


 **** #11 miss at 004dceb0: emit-rtl.c:1795
   emit-rtl.c:1794
   004dce70:                        : <next_active_insn+28> lw $a0[4],12($a0[4])
   emit-rtl.c:1795
   004dce78:  (       4488,   0.03): <next_active_insn+30> beq $a0[4],$zero[0],004dcef0 <next_active_insn+a8>
   004dce80:                        : <next_active_insn+38> lhu $v0[2],0($a0[4])
   004dce88:                        : <next_active_insn+40> addu $v0[2],$v0[2],$a3[7]
   004dce90:                        : <next_active_insn+48> andi $v0[2],$v0[2],65535
   004dce98:                        : <next_active_insn+50> sltiu $v0[2],$v0[2],2
   004dcea0:  (      14755,   0.09): <next_active_insn+58> bne $v0[2],$zero[0],004dcef0 <next_active_insn+a8>
   004dcea8:                        : <next_active_insn+60> lhu $v0[2],0($a0[4])
```

```
** 004dceb0:  (      76013,   0.48): <next_active_insn+68> bne $v0[2],$a2[6],004dce70 <next_active_insn+28>
   004dceb8:  (        490,   0.00): <next_active_insn+70> beq $v1[3],$zero[0],004dcef0 <next_active_insn+a8>
   004dcec0:                        : <next_active_insn+78> lw $v0[2],16($a0[4])
   004dcec8:                        : <next_active_insn+80> lhu $v0[2],0($v0[2])
   004dced0:                        : <next_active_insn+88> addu $v0[2],$v0[2],$a1[5]
   004dced8:                        : <next_active_insn+90> andi $v0[2],$v0[2],65535
   004dcee0:                        : <next_active_insn+98> sltiu $v0[2],$v0[2],2
   004dcee8:  (      23315,   0.15): <next_active_insn+a0> bne $v0[2],$zero[0],004dce70 <next_active_insn+28>
   emit-rtl.c:1804
   004dcef0:                        : <next_active_insn+a8> addu $v0[2],$zero[0],$a0[4]


---- Source Extract [emit-rtl.c] [1795]
---- (emit-rtl.c)
 1783:
 1784: /* Find the next insn after INSN that really does something.  This routine
 1785:    does not look inside SEQUENCEs.  Until reload has completed, this is the
 1786:    same as next_real_insn.  */
 1787:
 1788: rtx
 1789: next_active_insn (insn)
 1790:      rtx insn;
 1791: {
 1792:   while (insn)
 1793:     {
 1794:       insn = NEXT_INSN (insn);
*1795:       if (insn == 0
 1796:    || GET_CODE (insn) == CALL_INSN || GET_CODE (insn) == JUMP_INSN
 1797:    || (GET_CODE (insn) == INSN
 1798:        && (! reload_completed
 1799:    || (GET_CODE (PATTERN (insn)) != USE
 1800:        && GET_CODE (PATTERN (insn)) != CLOBBER))))
 1801:  break;
 1802:     }
 1803:
 1804:   return insn;
 1805: }
 1806:
 1807: /* Find the last insn before INSN that really does something.  This routine


 **** #12 miss at 00600f38: ../sysdeps/generic/memset.c:62
   ../sysdeps/generic/memset.c:58
   00600f10:                        : <memset+c0> sw $a3[7],-4($v0[2])
   ../sysdeps/generic/memset.c:59
   00600f18:                        : <memset+c8> sw $a3[7],0($v0[2])
   ../sysdeps/generic/memset.c:60
   00600f20:                        : <memset+d0> addiu $v0[2],$v0[2],32
   00600f28:                        : <memset+d8> addiu $t0[8],$t0[8],32
   ../sysdeps/generic/memset.c:61
   00600f30:                        : <memset+e0> addiu $v1[3],$v1[3],-1
   ../sysdeps/generic/memset.c:62
** 00600f38:  (      73309,   0.46): <memset+e8> bne $v1[3],$zero[0],00600ee0 <memset+90>
   ../sysdeps/generic/memset.c:63
   00600f40:                        : <memset+f0> andi $a2[6],$a2[6],31
   ../sysdeps/generic/memset.c:66
   00600f48:                        : <memset+f8> srl $v1[3],$a2[6],0x2
   ../sysdeps/generic/memset.c:67
   00600f50:  (       7371,   0.05): <memset+100> beq $v1[3],$zero[0],00600f78 <memset+128>
   ../sysdeps/generic/memset.c:69
   00600f58:                        : <memset+108> sw $a3[7],0($t0[8])
   ../sysdeps/generic/memset.c:70


---- Source Extract [../sysdeps/generic/memset.c] [62]
---- (/pong/usr3/s/smithz/research/tools/src/glibc-1.09/sysdeps/generic/memset.c)
 50:       while (xlen > 0)
 51:  {
 52:    ((op_t *) dstp)[0] = cccc;
 53:    ((op_t *) dstp)[1] = cccc;
 54:    ((op_t *) dstp)[2] = cccc;
 55:    ((op_t *) dstp)[3] = cccc;
 56:    ((op_t *) dstp)[4] = cccc;
 57:    ((op_t *) dstp)[5] = cccc;
 58:    ((op_t *) dstp)[6] = cccc;
 59:    ((op_t *) dstp)[7] = cccc;
 60:    dstp += 8 * OPSIZ;
 61:    xlen -= 1;
*62:  }
 63:       len %= OPSIZ * 8;
 64:
 65:       /* Write 1 `op_t' per iteration until less than OPSIZ bytes remain.  */
 66:       xlen = len / OPSIZ;
 67:       while (xlen > 0)
 68:  {
 69:    ((op_t *) dstp)[0] = cccc;
 70:    dstp += OPSIZ;
 71:    xlen -= 1;
 72:  }
 73:       len %= OPSIZ;
 74:     }
```

```
 **** #13 miss at 0050acd0: cse.c:1299
   cse.c:1286
   0050aca8:                        : <insert+508> sb $t2[10],38($s2[18])
   cse.c:1294
   0050acb0:                        : <insert+510> lw $v1[3],0($v0[2])
   0050acb8:  (      64364,   0.41): <insert+518> beq $v1[3],$zero[0],0050acc8 <insert+528>
   cse.c:1295
   0050acc0:                        : <insert+520> sw $s2[18],8($v1[3])
   cse.c:1296
   0050acc8:                        : <insert+528> sw $s2[18],0($v0[2])
   cse.c:1299
** 0050acd0:  (      73285,   0.46): <insert+530> beq $s1[17],$zero[0],0050adc8 <insert+628>
   cse.c:1301
   0050acd8:                        : <insert+538> lw $s1[17],20($s1[17])
   cse.c:1302
   0050ace0:                        : <insert+540> lw $a1[5],28($s2[18])
   0050ace8:                        : <insert+548> lw $v0[2],28($s1[17])
   0050acf0:                        : <insert+550> slt $v0[2],$a1[5],$v0[2]
   0050acf8:  (      42850,   0.27): <insert+558> beq $v0[2],$zero[0],0050ad48 <insert+5a8>
   cse.c:1310
   0050ad00:                        : <insert+560> addu $v0[2],$zero[0],$s1[17]


---- Source Extract [cse.c] [1299]
---- (cse.c)
 1287:     /* GNU C++ takes advantage of this for 'this'
 1288:        (and other const values).  */
 1289:     || (RTX_UNCHANGING_P (x)
 1290:         && GET_CODE (x) == REG
 1291:         && REGNO (x) >= FIRST_PSEUDO_REGISTER)
 1292:     || FIXED_BASE_PLUS_P (x));
 1293:
 1294:   if (table[hash])
 1295:     table[hash]->prev_same_hash = elt;
 1296:   table[hash] = elt;
 1297:
 1298:   /* Put it into the proper value-class.  */
*1299:   if (classp)
 1300:     {
 1301:       classp = classp->first_same_value;
 1302:       if (CHEAPER (elt, classp))
 1303:  /* Insert at the head of the class */
 1304:  {
 1305:    register struct table_elt *p;
 1306:    elt->next_same_value = classp;
 1307:    classp->prev_same_value = elt;
 1308:    elt->first_same_value = elt;
 1309:
 1310:    for (p = classp; p; p = p->next_same_value)
 1311:      p->first_same_value = elt;


 **** #14 miss at 0051fa10: cse.c:6852
   cse.c:6852
   0051f9c8:                        : <cse_insn+4b10> lw $v0[2],-26604($gp[28])
   0051f9d0:                        : <cse_insn+4b18> addu $s1[17],$zero[0],$zero[0]
   0051f9d8:                        : <cse_insn+4b20> lui $s3[19],4099
   0051f9e0:                        : <cse_insn+4b28> addiu $s3[19],$s3[19],16768
   0051f9e8:                        : <cse_insn+4b30> sll $s5[21],$v0[2],0x3
   0051f9f0:                        : <cse_insn+4b38> sra $s5[21],$s5[21],0x1f
   0051f9f8:                        : <cse_insn+4b40> sll $s4[20],$v0[2],0x2
   0051fa00:                        : <cse_insn+4b48> sra $s4[20],$s4[20],0x1f
   0051fa08:                        : <cse_insn+4b50> lw $s0[16],0($s3[19])
** 0051fa10:  (      73059,   0.46): <cse_insn+4b58> beq $s0[16],$zero[0],0051fa90 <cse_insn+4bd8>
   0051fa18:                        : <cse_insn+4b60> lb $v0[2],36($s0[16])
   0051fa20:                        : <cse_insn+4b68> lw $s2[18],4($s0[16])
   0051fa28:  (       5748,   0.04): <cse_insn+4b70> beq $v0[2],$zero[0],0051fa80 <cse_insn+4bc8>
   0051fa30:  (        163,   0.00): <cse_insn+4b78> bne $s5[21],$zero[0],0051fa68 <cse_insn+4bb0>
   0051fa38:                        : <cse_insn+4b80> beq $s4[20],$zero[0],0051fa50 <cse_insn+4b98>
   0051fa40:                        : <cse_insn+4b88> lb $v0[2],37($s0[16])
   0051fa48:                        : <cse_insn+4b90> bne $v0[2],$zero[0],0051fa68 <cse_insn+4bb0>
   0051fa50:                        : <cse_insn+4b98> lw $a0[4],0($s0[16])
   0051fa58:                        : <cse_insn+4ba0> jal 0050e7d8 <cse_rtx_addr_varies_p>


---- Source Extract [cse.c] [6852]
---- (cse.c)
 6840:       sets[i].src_elt = src_eqv_elt;
 6841:
 6842:   invalidate_from_clobbers (&writes_memory, x);
 6843:
 6844:   /* Some registers are invalidated by subroutine calls.  Memory is
 6845:      invalidated by non-constant calls.  */
 6846:
 6847:   if (GET_CODE (insn) == CALL_INSN)
 6848:     {
 6849:       static struct write_data everything = {0, 1, 1, 1};
 6850:
 6851:       if (! CONST_CALL_P (insn))
*6852:  invalidate_memory (&everything);
 6853:       invalidate_for_call ();
```

```
 6854:     }
 6855:
 6856:   /* Now invalidate everything set by this instruction.
 6857:      If a SUBREG or other funny destination is being set,
 6858:      sets[i].rtl is still nonzero, so here we invalidate the reg
 6859:      a part of which is being set.  */
 6860:
 6861:   for (i = 0; i < n_sets; i++)
 6862:     if (sets[i].rtl)
 6863:       {
 6864:  register rtx dest = sets[i].inner_dest;


 **** #15 miss at 0050c998: cse.c:1707
   cse.c:1706
   0050c958:                       : <invalidate_for_call+228> addu $s0[16],$zero[0],$zero[0]
   0050c960:                       : <invalidate_for_call+230> addiu $s5[21],$zero[0],52
   0050c968:                       : <invalidate_for_call+238> lui $s4[20],4099
   0050c970:                       : <invalidate_for_call+240> addiu $s4[20],$s4[20],-17788
   0050c978:                       : <invalidate_for_call+248> addiu $s3[19],$gp[28],-22784
   0050c980:                       : <invalidate_for_call+250> lui $s2[18],4099
   0050c988:                       : <invalidate_for_call+258> addiu $s2[18],$s2[18],16768
   cse.c:1707
   0050c990:                       : <invalidate_for_call+260> lw $a0[4],0($s2[18])
** 0050c998:  (      70733,   0.45): <invalidate_for_call+268> beq $a0[4],$zero[0],0050cad0 <invalidate_for_call+3a0>
   cse.c:1711
   0050c9a0:                       : <invalidate_for_call+270> lw $a1[5],0($a0[4])
   cse.c:1709
   0050c9a8:                       : <invalidate_for_call+278> lw $s1[17],4($a0[4])
   cse.c:1711
   0050c9b0:                       : <invalidate_for_call+280> lhu $v0[2],0($a1[5])
   0050c9b8:  (     117492,   0.74): <invalidate_for_call+288> bne $v0[2],$s5[21],0050cac0 <invalidate_for_call+390>
   0050c9c0:                       : <invalidate_for_call+290> lw $v1[3],4($a1[5])
   0050c9c8:                       : <invalidate_for_call+298> slti $v0[2],$v1[3],64


---- Source Extract [cse.c] [1707]
---- (cse.c)
 1695:  if (reg_tick[regno] >= 0)
 1696:    reg_tick[regno]++;
 1697:
 1698:  in_table |= TEST_HARD_REG_BIT (hard_regs_in_table, regno);
 1699:      }
 1700:
 1701:   /* In the case where we have no call-clobbered hard registers in the
 1702:      table, we are done.  Otherwise, scan the table and remove any
 1703:      entry that overlaps a call-clobbered register.  */
 1704:
 1705:   if (in_table)
 1706:     for (hash = 0; hash < NBUCKETS; hash++)
*1707:       for (p = table[hash]; p; p = next)
 1708:  {
 1709:    next = p->next_same_hash;
 1710:
 1711:    if (GET_CODE (p->exp) != REG
 1712:        || REGNO (p->exp) >= FIRST_PSEUDO_REGISTER)
 1713:      continue;
 1714:
 1715:    regno = REGNO (p->exp);
 1716:    endregno = regno + HARD_REGNO_NREGS (regno, GET_MODE (p->exp));
 1717:
 1718:    for (i = regno; i < endregno; i++)
 1719:      if (TEST_HARD_REG_BIT (regs_invalidated_by_call, i))


 **** #16 miss at 00400560: bison.simple:355
   00400520:                       : <yyparse+330> addiu $a1[5],$a1[5],19064
   00400528:                       : <yyparse+338> addu $a2[6],$zero[0],$s3[19]
   00400530:                       : <yyparse+340> jal 005fe0c0 <fprintf>
   bison.simple:354
   00400538:                       : <yyparse+348> sll $v0[2],$s3[19],0x1
   00400540:                       : <yyparse+350> lui $s5[21],4096
   00400548:                       : <yyparse+358> addu $s5[21],$s5[21],$v0[2]
   00400550:                       : <yyparse+360> lh $s5[21],9412($s5[21])
   bison.simple:355
   00400558:                       : <yyparse+368> addiu $v0[2],$zero[0],-32768
** 00400560:  (      67960,   0.43): <yyparse+370> beq $s5[21],$v0[2],00400858 <yyparse+668>
   bison.simple:363
   00400568:                       : <yyparse+378> lw $v1[3],-21476($gp[28])
   00400570:                       : <yyparse+380> addiu $v0[2],$zero[0],-2
   00400578:  (      19251,   0.12): <yyparse+388> bne $v1[3],$v0[2],004005c0 <yyparse+3d0>
   bison.simple:366
   00400580:                       : <yyparse+390> lw $v0[2],-21484($gp[28])
   00400588:  (        177,   0.00): <yyparse+398> beq $v0[2],$zero[0],004005b0 <yyparse+3c0>
   bison.simple:367
   00400590:                       : <yyparse+3a0> lw $a0[4],-23992($gp[28])


---- Source Extract [bison.simple] [355]
---- (bison.simple)
!! Could not open /pong/usr3/s/smithz/research/bench/spec95-src/126.gcc/src/bison.simple !
```

```
 **** #17 miss at 0050ee20: cse.c:2449
   0050edd8:                        : <canon_reg+3e0> lw $v0[2],4($s2[18])
   0050ede0:                        : <canon_reg+3e8> lw $v0[2],0($v0[2])
   0050ede8:                        : <canon_reg+3f0> addiu $s1[17],$s1[17],1
   0050edf0:                        : <canon_reg+3f8> sltu $v0[2],$s1[17],$v0[2]
   0050edf8:                        : <canon_reg+400> bne $v0[2],$zero[0],0050ed90 <canon_reg+398>
   cse.c:2449
   0050ee00:                        : <canon_reg+408> addiu $s2[18],$s2[18],-4
   0050ee08:                        : <canon_reg+410> addiu $s7[23],$s7[23],-4
   0050ee10:                        : <canon_reg+418> addiu $s6[22],$s6[22],-1
   0050ee18:                        : <canon_reg+420> addiu $s4[20],$s4[20],-1
** 0050ee20:  (      65125,   0.41): <canon_reg+428> bgez $s4[20],0050ec48 <canon_reg+250>
   cse.c:2473
   0050ee28:                        : <canon_reg+430> addu $v0[2],$zero[0],$s5[21]
   cse.c:2474
   0050ee30:                        : <canon_reg+438> lw $ra[31],60($sp[29])
   0050ee38:                        : <canon_reg+440> lw $s8[30],56($sp[29])
   0050ee40:                        : <canon_reg+448> lw $s7[23],52($sp[29])
   0050ee48:                        : <canon_reg+450> lw $s6[22],48($sp[29])
   0050ee50:                        : <canon_reg+458> lw $s5[21],44($sp[29])
   0050ee58:                        : <canon_reg+460> lw $s4[20],40($sp[29])


---- Source Extract [cse.c] [2449]
---- (cse.c)
 2437:  if (REGNO (x) < FIRST_PSEUDO_REGISTER
 2438:      || ! REGNO_QTY_VALID_P (REGNO (x)))
 2439:    return x;
 2440:
 2441:  first = qty_first_reg[reg_qty[REGNO (x)]];
 2442:  return (first >= FIRST_PSEUDO_REGISTER ? regno_reg_rtx[first]
 2443:  : REGNO_REG_CLASS (first) == NO_REGS ? x
 2444:  : gen_rtx (REG, qty_mode[reg_qty[REGNO (x)]], first));
 2445:      }
 2446:    }
 2447:
 2448:   fmt = GET_RTX_FORMAT (code);
*2449:   for (i = GET_RTX_LENGTH (code) - 1; i >= 0; i--)
 2450:     {
 2451:       register int j;
 2452:
 2453:       if (fmt[i] == 'e')
 2454:  {
 2455:    rtx new = canon_reg (XEXP (x, i), insn);
 2456:
 2457:    /* If replacing pseudo with hard reg or vice versa, ensure the
 2458:       insn remains valid.  Likewise if the insn has MATCH_DUPs.  */
 2459:    if (insn != 0 && new != 0
 2460:        && GET_CODE (new) == REG && GET_CODE (XEXP (x, i)) == REG
 2461:        && (((REGNO (new) < FIRST_PSEUDO_REGISTER)


 **** #18 miss at 004d6f10: rtlanal.c:923
   004d6ec8:                        : <rtx_equal_p+8> sw $ra[31],56($sp[29])
   004d6ed0:                        : <rtx_equal_p+10> sw $s7[23],52($sp[29])
   004d6ed8:                        : <rtx_equal_p+18> sw $s6[22],48($sp[29])
   004d6ee0:                        : <rtx_equal_p+20> sw $s5[21],44($sp[29])
   004d6ee8:                        : <rtx_equal_p+28> sw $s4[20],40($sp[29])
   004d6ef0:                        : <rtx_equal_p+30> sw $s3[19],36($sp[29])
   004d6ef8:                        : <rtx_equal_p+38> sw $s2[18],32($sp[29])
   004d6f00:                        : <rtx_equal_p+40> sw $s1[17],28($sp[29])
   004d6f08:                        : <rtx_equal_p+48> sw $s0[16],24($sp[29])
   rtlanal.c:923
** 004d6f10:  (      64481,   0.41): <rtx_equal_p+50> beq $a0[4],$a1[5],004d7270 <rtx_equal_p+3b0>
   rtlanal.c:925
   004d6f18:  (       1225,   0.01): <rtx_equal_p+58> beq $a0[4],$zero[0],004d7120 <rtx_equal_p+260>
   004d6f20:  (        657,   0.00): <rtx_equal_p+60> beq $a1[5],$zero[0],004d7120 <rtx_equal_p+260>
   rtlanal.c:928
   004d6f28:                        : <rtx_equal_p+68> lhu $a2[6],0($a0[4])
   rtlanal.c:930
   004d6f30:                        : <rtx_equal_p+70> lhu $v0[2],0($a1[5])
   004d6f38:  (     102417,   0.65): <rtx_equal_p+78> bne $a2[6],$v0[2],004d7120 <rtx_equal_p+260>
   rtlanal.c:936


---- Source Extract [rtlanal.c] [923]
---- (rtlanal.c)
 911: /* Return 1 if X and Y are identical-looking rtx's.
 912:    This is the Lisp function EQUAL for rtx arguments.  */
 913:
 914: int
 915: rtx_equal_p (x, y)
 916:      rtx x, y;
 917: {
 918:   register int i;
 919:   register int j;
 920:   register enum rtx_code code;
 921:   register char *fmt;
 922:
*923:   if (x == y)
 924:     return 1;
```

```
 925:   if (x == 0 || y == 0)
 926:     return 0;
 927:
 928:   code = GET_CODE (x);
 929:   /* Rtx's of different codes cannot be equal.  */
 930:   if (code != GET_CODE (y))
 931:     return 0;
 932:
 933:   /* (MULT:SI x y) and (MULT:HI x y) are NOT equivalent.
 934:      (REG:SI x) and (REG:HI x) are NOT equivalent.  */
 935:


 **** #19 miss at 0050acb8: cse.c:1294
   0050ac80:                       : <insert+4e0> addiu $t2[10],$zero[0],1
   cse.c:1294
   0050ac88:                       : <insert+4e8> lui $v1[3],4099
   0050ac90:                       : <insert+4f0> addiu $v1[3],$v1[3],16768
   0050ac98:                       : <insert+4f8> sll $v0[2],$s3[19],0x2
   0050aca0:                       : <insert+500> addu $v0[2],$v0[2],$v1[3]
   cse.c:1286
   0050aca8:                       : <insert+508> sb $t2[10],38($s2[18])
   cse.c:1294
   0050acb0:                       : <insert+510> lw $v1[3],0($v0[2])
** 0050acb8:  (      64364,   0.41): <insert+518> beq $v1[3],$zero[0],0050acc8 <insert+528>
   cse.c:1295
   0050acc0:                       : <insert+520> sw $s2[18],8($v1[3])
   cse.c:1296
   0050acc8:                       : <insert+528> sw $s2[18],0($v0[2])
   cse.c:1299
   0050acd0:  (      73285,   0.46): <insert+530> beq $s1[17],$zero[0],0050adc8 <insert+628>
   cse.c:1301
   0050acd8:                       : <insert+538> lw $s1[17],20($s1[17])
   cse.c:1302


---- Source Extract [cse.c] [1294]
---- (cse.c)
 1282:   elt->prev_same_hash = 0;
 1283:   elt->related_value = 0;
 1284:   elt->in_memory = 0;
 1285:   elt->mode = mode;
 1286:   elt->is_const = (CONSTANT_P (x)
 1287:     /* GNU C++ takes advantage of this for `this'
 1288:        (and other const values).  */
 1289:     || (RTX_UNCHANGING_P (x)
 1290:         && GET_CODE (x) == REG
 1291:         && REGNO (x) >= FIRST_PSEUDO_REGISTER)
 1292:     || FIXED_BASE_PLUS_P (x));
 1293:
*1294:   if (table[hash])
 1295:     table[hash]->prev_same_hash = elt;
 1296:   table[hash] = elt;
 1297:
 1298:   /* Put it into the proper value-class.  */
 1299:   if (classp)
 1300:     {
 1301:       classp = classp->first_same_value;
 1302:       if (CHEAPER (elt, classp))
 1303:  /* Insert at the head of the class */
 1304:  {
 1305:    register struct table_elt *p;
 1306:    elt->next_same_value = classp;


 **** #20 miss at 0050e660: cse.c:2337
   cse.c:2336
   0050e620:                       : <refers_to_mem_p+1e8> addiu $s3[19],$v1[3],-1
   0050e628:  (        875,   0.01): <refers_to_mem_p+1f0> bltz $s3[19],0050e770 <refers_to_mem_p+338>
   0050e630:                       : <refers_to_mem_p+1f8> sll $v0[2],$s3[19],0x2
   0050e638:                       : <refers_to_mem_p+200> addu $s1[17],$v0[2],$s5[21]
   0050e640:                       : <refers_to_mem_p+208> addu $s2[18],$s3[19],$a0[4]
   0050e648:                       : <refers_to_mem_p+210> addu $s8[30],$zero[0],$a0[4]
   cse.c:2337
   0050e650:                       : <refers_to_mem_p+218> lb $v0[2],0($s2[18])
   0050e658:                       : <refers_to_mem_p+220> addiu $t0[8],$zero[0],101
** 0050e660:  (      62741,   0.40): <refers_to_mem_p+228> bne $v0[2],$t0[8],0050e6b0 <refers_to_mem_p+278>
   cse.c:2339
   0050e668:  (      18727,   0.12): <refers_to_mem_p+230> beq $s2[18],$s8[30],0050e4f8 <refers_to_mem_p+c0>
   cse.c:2345
   0050e670:                       : <refers_to_mem_p+238> lw $a0[4],4($s1[17])
   0050e678:                       : <refers_to_mem_p+240> addu $a1[5],$zero[0],$s4[20]
   0050e680:                       : <refers_to_mem_p+248> addu $a2[6],$zero[0],$s6[22]
   0050e688:                       : <refers_to_mem_p+250> addu $a3[7],$zero[0],$s7[23]
   0050e690:                       : <refers_to_mem_p+258> jal 0050e438 <refers_to_mem_p>
   0050e698:  (          2,   0.00): <refers_to_mem_p+260> beq $v0[2],$zero[0],0050e750 <refers_to_mem_p+318>


---- Source Extract [cse.c] [2337]
---- (cse.c)
 2325:   If the base addresses are not equal, there is no chance
 2326:   of the memory addresses conflicting.  */
```

```
 2327:        if (! rtx_equal_p (mybase, base))
 2328:  return 0;
 2329:
 2330:        return myend > start && mystart < end;
 2331:      }
 2332:
 2333:   /* X does not match, so try its subexpressions.  */
 2334:
 2335:   fmt = GET_RTX_FORMAT (code);
 2336:   for (i = GET_RTX_LENGTH (code) - 1; i >= 0; i--)
*2337:     if (fmt[i] == 'e')
 2338:       {
 2339:  if (i == 0)
 2340:    {
 2341:      x = XEXP (x, 0);
 2342:      goto repeat;
 2343:    }
 2344:  else
 2345:    if (refers_to_mem_p (XEXP (x, i), base, start, end))
 2346:      return 1;
 2347:       }
 2348:     else if (fmt[i] == 'E')
 2349:       {


 **** #21 miss at 0050cf30: cse.c:1833
   0050cf00:                        : <canon_hash+70> j 0050d520 <canon_hash+690>
   cse.c:1951
   0050cf08:                        : <canon_hash+78> addu $a2[6],$zero[0],$a1[5]
   cse.c:1952
   0050cf10:                        : <canon_hash+80> j 0050cef8 <canon_hash+68>
   cse.c:1832
   0050cf18:                        : <canon_hash+88> lhu $a3[7],0($a2[6])
   cse.c:1833
   0050cf20:                        : <canon_hash+90> addiu $v1[3],$a3[7],-36
   0050cf28:                        : <canon_hash+98> sltiu $v0[2],$v1[3],55
** 0050cf30:  (      60508,   0.38): <canon_hash+a0> beq $v0[2],$zero[0],0050d1a8 <canon_hash+318>
   0050cf38:                        : <canon_hash+a8> sll $v0[2],$v1[3],0x2
   0050cf40:                        : <canon_hash+b0> lui $at[1],4097
   0050cf48:                        : <canon_hash+b8> addu $at[1],$at[1],$v0[2]
   0050cf50:                        : <canon_hash+c0> lw $v0[2],9248($at[1])
   0050cf58:                        : <canon_hash+c8> jr $v0[2]
   cse.c:1837
   0050cf60:                        : <canon_hash+d0> lw $v1[3],4($a2[6])
   cse.c:1844
   0050cf68:                        : <canon_hash+d8> slti $v0[2],$v1[3],64


---- Source Extract [cse.c] [1833]
---- (cse.c)
 1821: {
 1822:   register int i, j;
 1823:   register int hash = 0;
 1824:   register enum rtx_code code;
 1825:   register char *fmt;
 1826:
 1827:   /* repeat is used to turn tail-recursion into iteration.  */
 1828:  repeat:
 1829:   if (x == 0)
 1830:     return hash;
 1831:
 1832:   code = GET_CODE (x);
*1833:   switch (code)
 1834:     {
 1835:     case REG:
 1836:       {
 1837:  register int regno = REGNO (x);
 1838:
 1839:  /* On some machines, we can't record any non-fixed hard register,
 1840:     because extending its life will cause reload problems.  We
 1841:     consider ap, fp, and sp to be fixed for this purpose.
 1842:     On all machines, we can't record any global registers. */
 1843:
 1844:  if (regno < FIRST_PSEUDO_REGISTER
 1845:      && (global_regs[regno]


 **** #22 miss at 004d6288: rtlanal.c:677
   004d6240:                        : <refers_to_regno_p+98> lhu $s0[16],0($s3[19])
   rtlanal.c:677
   004d6248:                        : <refers_to_regno_p+a0> addiu $v0[2],$zero[0],43
   004d6250:  (        799,   0.01): <refers_to_regno_p+a8> beq $s0[16],$v0[2],004d64d8 <refers_to_regno_p+330>
   004d6258:                        : <refers_to_regno_p+b0> sltiu $v0[2],$s0[16],44
   004d6260:  (       8127,   0.05): <refers_to_regno_p+b8> beq $v0[2],$zero[0],004d6280 <refers_to_regno_p+d8>
   004d6268:                        : <refers_to_regno_p+c0> addiu $v0[2],$zero[0],41
   004d6270:  (       3239,   0.02): <refers_to_regno_p+c8> beq $s0[16],$v0[2],004d64d8 <refers_to_regno_p+330>
   004d6278:                        : <refers_to_regno_p+d0> j 004d65f0 <refers_to_regno_p+448>
   004d6280:                        : <refers_to_regno_p+d8> addiu $a0[4],$zero[0],52
** 004d6288:  (      60238,   0.38): <refers_to_regno_p+e0> beq $s0[16],$a0[4],004d62a8 <refers_to_regno_p+100>
   004d6290:                        : <refers_to_regno_p+e8> addiu $v0[2],$zero[0],54
   004d6298:  (       5758,   0.04): <refers_to_regno_p+f0> beq $s0[16],$v0[2],004d63d8 <refers_to_regno_p+230>
   004d62a0:                        : <refers_to_regno_p+f8> j 004d65f0 <refers_to_regno_p+448>
```

```
   rtlanal.c:680
   004d62a8:                        : <refers_to_regno_p+100> lw $s2[18],4($s3[19])
   rtlanal.c:685
   004d62b0:                        : <refers_to_regno_p+108> addiu $v0[2],$zero[0],14
   004d62b8:  (       1248,   0.01): <refers_to_regno_p+110> beq $s2[18],$v0[2],004d62d0 <refers_to_regno_p+128>
   004d62c0:                        : <refers_to_regno_p+118> addiu $v0[2],$zero[0],30


---- Source Extract [rtlanal.c] [677]
---- (rtlanal.c)
 665:   register int i;
 666:   register RTX_CODE code;
 667:   register char *fmt;
 668:
 669:  repeat:
 670:   /* The contents of a REG_NONNEG note is always zero, so we must come here
 671:      upon repeat in case the last REG_NOTE is a REG_NONNEG note.  */
 672:   if (x == 0)
 673:     return 0;
 674:
 675:   code = GET_CODE (x);
 676:
*677:   switch (code)
 678:     {
 679:     case REG:
 680:       i = REGNO (x);
 681:
 682:       /* If we modifying the stack, frame, or argument pointer, it will
 683:   clobber a virtual register.  In fact, we could be more precise,
 684:   but it isn't worth it.  */
 685:       if ((i == STACK_POINTER_REGNUM
 686: #if FRAME_POINTER_REGNUM != ARG_POINTER_REGNUM
 687:     || i == ARG_POINTER_REGNUM
 688: #endif
 689:     || i == FRAME_POINTER_REGNUM)


 **** #23 miss at 0050c020: cse.c:1530
   0050bfe8:  (     196176,   1.24): <invalidate+718> beq $a0[4],$zero[0],0050c0c0 <invalidate+7f0>
   cse.c:1530
   0050bff0:                        : <invalidate+720> lw $v1[3],0($a0[4])
   cse.c:1528
   0050bff8:                        : <invalidate+728> lw $s0[16],4($a0[4])
   cse.c:1530
   0050c000:                        : <invalidate+730> lhu $v0[2],0($v1[3])
   0050c008:  (     312230,   1.97): <invalidate+738> bne $v0[2],$s6[22],0050c0b0 <invalidate+7e0>
   0050c010:                        : <invalidate+740> lw $a1[5],4($v1[3])
   0050c018:                        : <invalidate+748> slti $v0[2],$a1[5],64
** 0050c020:  (      59019,   0.37): <invalidate+750> beq $v0[2],$zero[0],0050c0b0 <invalidate+7e0>
   cse.c:1535
   0050c028:                        : <invalidate+758> lw $v0[2],0($v1[3])
   0050c030:                        : <invalidate+760> sll $v0[2],$v0[2],0x10
   0050c038:                        : <invalidate+768> sra $v0[2],$v0[2],0x18
   0050c040:                        : <invalidate+770> sll $v0[2],$v0[2],0x2
   0050c048:                        : <invalidate+778> addu $v0[2],$v0[2],$s5[21]
   0050c050:                        : <invalidate+780> lw $v1[3],0($v0[2])
   0050c058:                        : <invalidate+788> addiu $v0[2],$v1[3],3
   0050c060:  (        341,   0.00): <invalidate+790> bgez $v0[2],0050c070 <invalidate+7a0>


---- Source Extract [cse.c] [1530]
---- (cse.c)
 1518:        in_table |= TEST_HARD_REG_BIT (hard_regs_in_table, i);
 1519:        CLEAR_HARD_REG_BIT (hard_regs_in_table, i);
 1520:        delete_reg_equiv (i);
 1521:        reg_tick[i]++;
 1522:      }
 1523:
 1524:    if (in_table)
 1525:      for (hash = 0; hash < NBUCKETS; hash++)
 1526:        for (p = table[hash]; p; p = next)
 1527:  {
 1528:    next = p->next_same_hash;
 1529:
*1530:    if (GET_CODE (p->exp) != REG
 1531:        || REGNO (p->exp) >= FIRST_PSEUDO_REGISTER)
 1532:      continue;
 1533:
 1534:    tregno = REGNO (p->exp);
 1535:    tendregno
 1536:      = tregno + HARD_REGNO_NREGS (tregno, GET_MODE (p->exp));
 1537:    if (tendregno > regno && tregno < endregno)
 1538:    remove_from_table (p, hash);
 1539:  }
 1540:  }
 1541:
 1542:       return;


 **** #24 miss at 0050d368: cse.c:1949
   0050d328:                        : <canon_hash+498> addiu $v0[2],$v1[3],-47
   0050d330:                        : <canon_hash+4a0> sltiu $v0[2],$v0[2],2
```

```
   0050d338:  (        573,   0.00): <canon_hash+4a8> bne $v0[2],$zero[0],0050d360 <canon_hash+4d0>
   0050d340:                        : <canon_hash+4b0> lhu $v1[3],0($a0[4])
   0050d348:                        : <canon_hash+4b8> addiu $v0[2],$zero[0],50
   0050d350:  (       1014,   0.01): <canon_hash+4c0> beq $v1[3],$v0[2],0050d360 <canon_hash+4d0>
   0050d358:  (        113,   0.00): <canon_hash+4c8> bne $v1[3],$s6[22],0050d368 <canon_hash+4d8>
   cse.c:1944
   0050d360:                        : <canon_hash+4d0> addu $a1[5],$zero[0],$a0[4]
   cse.c:1949
** 0050d368:  (      56761,   0.36): <canon_hash+4d8> beq $s4[20],$s5[21],0050cf08 <canon_hash+78>
   cse.c:1954
   0050d370:                        : <canon_hash+4e0> addu $a0[4],$zero[0],$a1[5]
   0050d378:                        : <canon_hash+4e8> addu $a1[5],$zero[0],$zero[0]
   0050d380:                        : <canon_hash+4f0> jal 0050ce90 <canon_hash>
   0050d388:                        : <canon_hash+4f8> addu $s1[17],$s1[17],$v0[2]
   cse.c:1955
   0050d390:                        : <canon_hash+500> j 0050d500 <canon_hash+670>
   cse.c:1956
   0050d398:                        : <canon_hash+508> addiu $v0[2],$zero[0],69


---- Source Extract [cse.c] [1949]
---- (cse.c)
 1937:       as if we were hashing the constant, since we will be comparing
 1938:       that way.  */
 1939:    if (tem != 0 && GET_CODE (tem) == REG
 1940:        && REGNO_QTY_VALID_P (REGNO (tem))
 1941:        && qty_mode[reg_qty[REGNO (tem)]] == GET_MODE (tem)
 1942:        && (tem1 = qty_const[reg_qty[REGNO (tem)]]) != 0
 1943:        && CONSTANT_P (tem1))
 1944:      tem = tem1;
 1945:
 1946:    /* If we are about to do the last recursive call
 1947:       needed at this level, change it into iteration.
 1948:       This function  is called enough to be worth it.  */
*1949:    if (i == 0)
 1950:      {
 1951:        x = tem;
 1952:        goto repeat;
 1953:      }
 1954:    hash += canon_hash (tem, 0);
 1955:  }
 1956:       else if (fmt[i] == 'E')
 1957:  for (j = 0; j < XVECLEN (x, i); j++)
 1958:    hash += canon_hash (XVECEXP (x, i, j), 0);
 1959:       else if (fmt[i] == 's')
 1960:  {
 1961:    register char *p = XSTR (x, i);


 **** #25 miss at 0050d6b8: cse.c:2030
   cse.c:2024
   0050d688:  (      50742,   0.32): <exp_equiv_p+78> bne $s1[17],$s0[16],0050d698 <exp_equiv_p+88>
   0050d690:  (      27283,   0.17): <exp_equiv_p+80> beq $s7[23],$zero[0],0050dea0 <exp_equiv_p+890>
   cse.c:2026
   0050d698:  (         31,   0.00): <exp_equiv_p+88> beq $s1[17],$zero[0],0050d998 <exp_equiv_p+388>
   0050d6a0:  (        127,   0.00): <exp_equiv_p+90> beq $s0[16],$zero[0],0050d998 <exp_equiv_p+388>
   cse.c:2029
   0050d6a8:                        : <exp_equiv_p+98> lhu $s2[18],0($s1[17])
   cse.c:2030
   0050d6b0:                        : <exp_equiv_p+a0> lhu $v0[2],0($s0[16])
** 0050d6b8:  (      54935,   0.35): <exp_equiv_p+a8> beq $s2[18],$v0[2],0050d940 <exp_equiv_p+330>
   cse.c:2032
   0050d6c0:  (       8703,   0.05): <exp_equiv_p+b0> beq $s6[22],$zero[0],0050dd48 <exp_equiv_p+738>
   cse.c:2037
   0050d6c8:                        : <exp_equiv_p+b8> lhu $v1[3],0($s1[17])
   0050d6d0:                        : <exp_equiv_p+c0> addiu $v0[2],$v1[3],-58
   0050d6d8:                        : <exp_equiv_p+c8> sltiu $v0[2],$v0[2],2
   0050d6e0:                        : <exp_equiv_p+d0> bne $v0[2],$zero[0],0050d720 <exp_equiv_p+110>
   0050d6e8:                        : <exp_equiv_p+d8> addiu $v0[2],$v1[3],-47
   0050d6f0:                        : <exp_equiv_p+e0> sltiu $v0[2],$v0[2],2


---- Source Extract [cse.c] [2030]
---- (cse.c)
 2018:   register int i, j;
 2019:   register enum rtx_code code;
 2020:   register char *fmt;
 2021:
 2022:   /* Note: it is incorrect to assume an expression is equivalent to itself
 2023:      if VALIDATE is nonzero.  */
 2024:   if (x == y && !validate)
 2025:     return 1;
 2026:   if (x == 0 || y == 0)
 2027:     return x == y;
 2028:
 2029:   code = GET_CODE (x);
*2030:   if (code != GET_CODE (y))
 2031:     {
 2032:       if (!equal_values)
 2033:  return 0;
 2034:
 2035:       /* If X is a constant and Y is a register or vice versa, they may be
```

```
 2036:   equivalent.  We only have to validate if Y is a register.  */
 2037:       if (CONSTANT_P (x) && GET_CODE (y) == REG
 2038:    && REGNO_QTY_VALID_P (REGNO (y))
 2039:    && GET_MODE (y) == qty_mode[reg_qty[REGNO (y)]]
 2040:    && rtx_equal_p (x, qty_const[reg_qty[REGNO (y)]])
 2041:    && (! validate || reg_in_table[REGNO (y)] == reg_tick[REGNO (y)]))
 2042:  return 1;


 **** #26 miss at 005cfd18: recog.c:873
   005cfce0:  (        623,   0.00): <register_operand+88> bne $v1[3],$v0[2],005cfcf8 <register_operand+a0>
   recog.c:867
   005cfce8:                        : <register_operand+90> jal 005cf290 <general_operand>
   005cfcf0:                        : <register_operand+98> j 005cfd60 <register_operand+108>
   recog.c:868
   005cfcf8:                        : <register_operand+a0> lw $a0[4],4($a0[4])
   recog.c:873
   005cfd00:                        : <register_operand+a8> lhu $v1[3],0($a0[4])
   005cfd08:                        : <register_operand+b0> addu $a1[5],$zero[0],$zero[0]
   005cfd10:                        : <register_operand+b8> addiu $v0[2],$zero[0],52
** 005cfd18:  (      54522,   0.34): <register_operand+c0> bne $v1[3],$v0[2],005cfd58 <register_operand+100>
   005cfd20:                        : <register_operand+c8> lw $a0[4],4($a0[4])
   005cfd28:                        : <register_operand+d0> slti $v0[2],$a0[4],64
   005cfd30:  (      29185,   0.18): <register_operand+d8> beq $v0[2],$zero[0],005cfd50 <register_operand+f8>
   005cfd38:                        : <register_operand+e0> slti $v0[2],$a0[4],32
   005cfd40:  (       4679,   0.03): <register_operand+e8> beq $v0[2],$zero[0],005cfd50 <register_operand+f8>
   005cfd48:  (       4993,   0.03): <register_operand+f0> beq $a0[4],$zero[0],005cfd58 <register_operand+100>
   005cfd50:                        : <register_operand+f8> addiu $a1[5],$zero[0],1
   005cfd58:                        : <register_operand+100> addu $v0[2],$zero[0],$a1[5]
   005cfd60:                        : <register_operand+108> lw $ra[31],16($sp[29])


---- Source Extract [recog.c] [873]
---- (recog.c)
 861:   because it is guaranteed to be reloaded into one.
 862:   Just make sure the MEM is valid in itself.
 863:   (Ideally, (SUBREG (MEM)...) should not exist after reload,
 864:   but currently it does result from (SUBREG (REG)...) where the
 865:   reg went on the stack.)  */
 866:       if (! reload_completed && GET_CODE (SUBREG_REG (op)) == MEM)
 867:  return general_operand (op, mode);
 868:       op = SUBREG_REG (op);
 869:     }
 870:
 871:   /* We don't consider registers whose class is NO_REGS
 872:      to be a register operand.  */
*873:   return (GET_CODE (op) == REG
 874:    && (REGNO (op) >= FIRST_PSEUDO_REGISTER
 875:        || REGNO_REG_CLASS (REGNO (op)) != NO_REGS));
 876: }
 877:
 878: /* Return 1 if OP should match a MATCH_SCRATCH, i.e., if it is a SCRATCH
 879:    or a hard register.  */
 880:
 881: int
 882: scratch_operand (op, mode)
 883:      register rtx op;
 884:      enum machine_mode mode;
 885: {


 **** #27 miss at 00520fe8: cse.c:7177
   00520f98:  (        915,   0.01): <invalidate_from_clobbers+f0> beq $s4[20],$zero[0],00520fb0 <invalidate_from_clobbers+108>
   00520fa0:                        : <invalidate_from_clobbers+f8> lb $v0[2],37($s0[16])
   00520fa8:  (       1354,   0.01): <invalidate_from_clobbers+100> bne $v0[2],$zero[0],00520fc8 <invalidate_from_clobbers+120>
   00520fb0:                        : <invalidate_from_clobbers+108> lw $a0[4],0($s0[16])
   00520fb8:                        : <invalidate_from_clobbers+110> jal 0050e7d8 <cse_rtx_addr_varies_p>
   00520fc0:  (        351,   0.00): <invalidate_from_clobbers+118> beq $v0[2],$zero[0],00520fe0 <invalidate_from_clobbers+138>
   00520fc8:                        : <invalidate_from_clobbers+120> addu $a0[4],$zero[0],$s0[16]
   00520fd0:                        : <invalidate_from_clobbers+128> addu $a1[5],$zero[0],$s1[17]
   00520fd8:                        : <invalidate_from_clobbers+130> jal 0050a390 <remove_from_table>
   00520fe0:                        : <invalidate_from_clobbers+138> addu $s0[16],$zero[0],$s2[18]
** 00520fe8:  (      53907,   0.34): <invalidate_from_clobbers+140> bne $s0[16],$zero[0],00520f78 <invalidate_from_clobbers+d0>
   00520ff0:                        : <invalidate_from_clobbers+148> addiu $s3[19],$s3[19],4
   00520ff8:                        : <invalidate_from_clobbers+150> addiu $s1[17],$s1[17],1
   00521000:                        : <invalidate_from_clobbers+158> slti $v0[2],$s1[17],31
   00521008:  (       4196,   0.03): <invalidate_from_clobbers+160> bne $v0[2],$zero[0],00520f68 <invalidate_from_clobbers+c0>
   cse.c:7179
   00521010:                        : <invalidate_from_clobbers+168> lw $v0[2],0($s7[23])
   00521018:  (        154,   0.00): <invalidate_from_clobbers+170> bgez $v0[2],00521070 <invalidate_from_clobbers+1c8>
   cse.c:7181
   00521020:                        : <invalidate_from_clobbers+178> lw $v1[3],-22800($gp[28])


---- Source Extract [cse.c] [7177]
---- (cse.c)
 7165:    saying which kinds of memory references must be invalidated.
 7166:    X is the pattern of the insn.  */
 7167:
 7168: static void
 7169: invalidate_from_clobbers (w, x)
 7170:      struct write_data *w;
```

```
 7171:       rtx x;
 7172: {
 7173:   /* If W->var is not set, W specifies no action.
 7174:      If W->all is set, this step gets all memory refs
 7175:      so they can be ignored in the rest of this function.  */
 7176:   if (w->var)
*7177:     invalidate_memory (w);
 7178:
 7179:   if (w->sp)
 7180:     {
 7181:       if (reg_tick[STACK_POINTER_REGNUM] >= 0)
 7182:  reg_tick[STACK_POINTER_REGNUM]++;
 7183:
 7184:       /* This should be *very* rare.  */
 7185:       if (TEST_HARD_REG_BIT (hard_regs_in_table, STACK_POINTER_REGNUM))
 7186:  invalidate (stack_pointer_rtx);
 7187:     }
 7188:
 7189:   if (GET_CODE (x) == CLOBBER)


 **** #28 miss at 0050c818: cse.c:1692
   0050c7d0:                       : <invalidate_for_call+a0> addu $t2[10],$zero[0],$t5[13]
   0050c7d8:                       : <invalidate_for_call+a8> addu $t1[9],$zero[0],$t4[12]
   cse.c:1692
   0050c7e0:                       : <invalidate_for_call+b0> srl $v0[2],$a2[6],0x5
   0050c7e8:                       : <invalidate_for_call+b8> sll $v0[2],$v0[2],0x2
   0050c7f0:                       : <invalidate_for_call+c0> addu $v0[2],$v0[2],$s3[19]
   0050c7f8:                       : <invalidate_for_call+c8> lw $v0[2],0($v0[2])
   0050c800:                       : <invalidate_for_call+d0> andi $v1[3],$a2[6],31
   0050c808:                       : <invalidate_for_call+d8> srlv $v0[2],$v0[2],$v1[3]
   0050c810:                       : <invalidate_for_call+e0> andi $v0[2],$v0[2],1
** 0050c818:  (      51091,   0.32): <invalidate_for_call+e8> beq $v0[2],$zero[0],0050c918 <invalidate_for_call+1e8>
   cse.c:1694
   0050c820:                       : <invalidate_for_call+f0> lw $a0[4],0($a3[7])
   0050c828:                       : <invalidate_for_call+f8> lw $a1[5],0($t1[9])
   0050c830:                       : <invalidate_for_call+100> lw $v1[3],0($t2[10])
   0050c838:  (      17637,   0.11): <invalidate_for_call+108> beq $a0[4],$a2[6],0050c8b8 <invalidate_for_call+188>
   0050c840:  (        112,   0.00): <invalidate_for_call+110> beq $a1[5],$t6[14],0050c860 <invalidate_for_call+130>
   0050c848:                       : <invalidate_for_call+118> sll $v0[2],$a1[5],0x2
   0050c850:                       : <invalidate_for_call+120> addu $v0[2],$v0[2],$t5[13]
   0050c858:                       : <invalidate_for_call+128> j 0050c870 <invalidate_for_call+140>


---- Source Extract [cse.c] [1692]
---- (cse.c)
 1680:   int regno, endregno;
 1681:   int i;
 1682:   int hash;
 1683:   struct table_elt *p, *next;
 1684:   int in_table = 0;
 1685:
 1686:   /* Go through all the hard registers.  For each that is clobbered in
 1687:      a CALL_INSN, remove the register from quantity chains and update
 1688:      reg_tick if defined.  Also see if any of these registers is currently
 1689:      in the table.  */
 1690:
 1691:   for (regno = 0; regno < FIRST_PSEUDO_REGISTER; regno++)
*1692:     if (TEST_HARD_REG_BIT (regs_invalidated_by_call, regno))
 1693:       {
 1694:  delete_reg_equiv (regno);
 1695:  if (reg_tick[regno] >= 0)
 1696:    reg_tick[regno]++;
 1697:
 1698:  in_table |= TEST_HARD_REG_BIT (hard_regs_in_table, regno);
 1699:       }
 1700:
 1701:   /* In the case where we have no call-clobbered hard registers in the
 1702:      table, we are done.  Otherwise, scan the table and remove any
 1703:      entry that overlaps a call-clobbered register.  */
 1704:


 **** #29 miss at 0050d688: cse.c:2024
   0050d640:                       : <exp_equiv_p+30> addu $s7[23],$zero[0],$a2[6]
   0050d648:                       : <exp_equiv_p+38> sw $s6[22],48($sp[29])
   0050d650:                       : <exp_equiv_p+40> addu $s6[22],$zero[0],$a3[7]
   0050d658:                       : <exp_equiv_p+48> sw $ra[31],60($sp[29])
   0050d660:                       : <exp_equiv_p+50> sw $s8[30],56($sp[29])
   0050d668:                       : <exp_equiv_p+58> sw $s5[21],44($sp[29])
   0050d670:                       : <exp_equiv_p+60> sw $s4[20],40($sp[29])
   0050d678:                       : <exp_equiv_p+68> sw $s3[19],36($sp[29])
   0050d680:                       : <exp_equiv_p+70> sw $s2[18],32($sp[29])
   cse.c:2024
** 0050d688:  (      50742,   0.32): <exp_equiv_p+78> bne $s1[17],$s0[16],0050d698 <exp_equiv_p+88>
   0050d690:  (      27283,   0.17): <exp_equiv_p+80> beq $s7[23],$zero[0],0050dea0 <exp_equiv_p+890>
   cse.c:2026
   0050d698:  (         31,   0.00): <exp_equiv_p+88> beq $s1[17],$zero[0],0050d998 <exp_equiv_p+388>
   0050d6a0:  (        127,   0.00): <exp_equiv_p+90> beq $s0[16],$zero[0],0050d998 <exp_equiv_p+388>
   cse.c:2029
   0050d6a8:                       : <exp_equiv_p+98> lhu $s2[18],0($s1[17])
   cse.c:2030
```

```
   0050d6b0:                       : <exp_equiv_p+a0> lhu $v0[2],0($s0[16])
   0050d6b8:  (      54935,   0.35): <exp_equiv_p+a8> beq $s2[18],$v0[2],0050d940 <exp_equiv_p+330>


---- Source Extract [cse.c] [2024]
---- (cse.c)
 2012: static int
 2013: exp_equiv_p (x, y, validate, equal_values)
 2014:      rtx x, y;
 2015:      int validate;
 2016:      int equal_values;
 2017: {
 2018:   register int i, j;
 2019:   register enum rtx_code code;
 2020:   register char *fmt;
 2021:
 2022:   /* Note: it is incorrect to assume an expression is equivalent to itself
 2023:      if VALIDATE is nonzero.  */
*2024:   if (x == y && !validate)
 2025:     return 1;
 2026:   if (x == 0 || y == 0)
 2027:     return x == y;
 2028:
 2029:   code = GET_CODE (x);
 2030:   if (code != GET_CODE (y))
 2031:     {
 2032:       if (!equal_values)
 2033:  return 0;
 2034:
 2035:       /* If X is a constant and Y is a register or vice versa, they may be
 2036:   equivalent.  We only have to validate if Y is a register.  */


 **** #30 miss at 005094c0: cse.c:733
   00509488:                       : <rtx_cost+538> addiu $s0[16],$s0[16],1
   cse.c:738
   00509490:                       : <rtx_cost+540> addu $s2[18],$s2[18],$v0[2]
   cse.c:737
   00509498:                       : <rtx_cost+548> sltu $v1[3],$s0[16],$v1[3]
   005094a0:                       : <rtx_cost+550> bne $v1[3],$zero[0],00509448 <rtx_cost+4f8>
   cse.c:733
   005094a8:                       : <rtx_cost+558> addiu $s1[17],$s1[17],-4
   005094b0:                       : <rtx_cost+560> addiu $s4[20],$s4[20],-1
   005094b8:                       : <rtx_cost+568> addiu $s3[19],$s3[19],-1
** 005094c0:  (      50054,   0.32): <rtx_cost+570> bgez $s3[19],005093e8 <rtx_cost+498>
   cse.c:740
   005094c8:                       : <rtx_cost+578> addu $v0[2],$zero[0],$s2[18]
   cse.c:741
   005094d0:                       : <rtx_cost+580> lw $ra[31],56($sp[29])
   005094d8:                       : <rtx_cost+588> lw $s7[23],52($sp[29])
   005094e0:                       : <rtx_cost+590> lw $s6[22],48($sp[29])
   005094e8:                       : <rtx_cost+598> lw $s5[21],44($sp[29])
   005094f0:                       : <rtx_cost+5a0> lw $s4[20],40($sp[29])
   005094f8:                       : <rtx_cost+5a8> lw $s3[19],36($sp[29])


---- Source Extract [cse.c] [733]
---- (cse.c)
 721:       + GET_MODE_SIZE (GET_MODE (x)) / UNITS_PER_WORD);
 722:      return 2;
 723: #ifdef RTX_COSTS
 724:      RTX_COSTS (x, code, outer_code);
 725: #endif
 726:      CONST_COSTS (x, code, outer_code);
 727:    }
 728:
 729:   /* Sum the costs of the sub-rtx's, plus cost of this operation,
 730:      which is already in total.  */
 731:
 732:   fmt = GET_RTX_FORMAT (code);
*733:   for (i = GET_RTX_LENGTH (code) - 1; i >= 0; i--)
 734:     if (fmt[i] == 'e')
 735:       total += rtx_cost (XEXP (x, i), code);
 736:     else if (fmt[i] == 'E')
 737:       for (j = 0; j < XVECLEN (x, i); j++)
 738:  total += rtx_cost (XVECEXP (x, i, j), code);
 739:
 740:   return total;
 741: }
 742:
 743: /* Clear the hash table and initialize each register with its own quantity,
 744:    for a new basic block.  */
 745:
```

Appendix C

cc-knights Top 25 Misses

```
 **** #1 miss at 0050e660: cse.c:2337
   cse.c:2336
   0050e620:                       : <refers_to_mem_p+1e8> addiu $s3[19],$v1[3],-1
   0050e628:  (        175,   0.01): <refers_to_mem_p+1f0> bltz $s3[19],0050e770 <refers_to_mem_p+338>
   0050e630:                       : <refers_to_mem_p+1f8> sll $v0[2],$s3[19],0x2
   0050e638:                       : <refers_to_mem_p+200> addu $s1[17],$v0[2],$s5[21]
   0050e640:                       : <refers_to_mem_p+208> addu $s2[18],$s3[19],$a0[4]
   0050e648:                       : <refers_to_mem_p+210> addu $s8[30],$zero[0],$a0[4]
   cse.c:2337
   0050e650:                       : <refers_to_mem_p+218> lb $v0[2],0($s2[18])
   0050e658:                       : <refers_to_mem_p+220> addiu $t0[8],$zero[0],101
** 0050e660:  (      32826,   2.00): <refers_to_mem_p+228> bne $v0[2],$t0[8],0050e6b0 <refers_to_mem_p+278>
   cse.c:2339
   0050e668:  (       5273,   0.32): <refers_to_mem_p+230> beq $s2[18],$s8[30],0050e4f8 <refers_to_mem_p+c0>
   cse.c:2345
   0050e670:                       : <refers_to_mem_p+238> lw $a0[4],4($s1[17])
   0050e678:                       : <refers_to_mem_p+240> addu $a1[5],$zero[0],$s4[20]
   0050e680:                       : <refers_to_mem_p+248> addu $a2[6],$zero[0],$s6[22]
   0050e688:                       : <refers_to_mem_p+250> addu $a3[7],$zero[0],$s7[23]
   0050e690:                       : <refers_to_mem_p+258> jal 0050e438 <refers_to_mem_p>
   0050e698:  (          1,   0.00): <refers_to_mem_p+260> beq $v0[2],$zero[0],0050e750 <refers_to_mem_p+318>


---- Source Extract [cse.c] [2337]
---- (cse.c)
 2325:   If the base addresses are not equal, there is no chance
 2326:   of the memory addresses conflicting.  */
 2327:       if (! rtx_equal_p (mybase, base))
 2328:  return 0;
 2329:
 2330:       return myend > start && mystart < end;
 2331:     }
 2332:
 2333:   /* X does not match, so try its subexpressions.  */
 2334:
 2335:   fmt = GET_RTX_FORMAT (code);
 2336:   for (i = GET_RTX_LENGTH (code) - 1; i >= 0; i--)
*2337:     if (fmt[i] == 'e')
 2338:       {
 2339:  if (i == 0)
 2340:    {
 2341:      x = XEXP (x, 0);
 2342:      goto repeat;
 2343:    }
 2344:  else
 2345:    if (refers_to_mem_p (XEXP (x, i), base, start, end))
 2346:      return 1;
 2347:       }
 2348:     else if (fmt[i] == 'E')
 2349:       {


 **** #2 miss at 00520fe8: cse.c:7177
   00520f98:  (        322,   0.02): <invalidate_from_clobbers+f0> beq $s4[20],$zero[0],00520fb0 <invalidate_from_clobbers+108>
   00520fa0:                       : <invalidate_from_clobbers+f8> lb $v0[2],37($s0[16])
   00520fa8:  (        300,   0.02): <invalidate_from_clobbers+100> bne $v0[2],$zero[0],00520fc8 <invalidate_from_clobbers+120>
   00520fb0:                       : <invalidate_from_clobbers+108> lw $a0[4],0($s0[16])
   00520fb8:                       : <invalidate_from_clobbers+110> jal 0050e7d8 <cse_rtx_addr_varies_p>
   00520fc0:  (        393,   0.02): <invalidate_from_clobbers+118> beq $v0[2],$zero[0],00520fe0 <invalidate_from_clobbers+138>
   00520fc8:                       : <invalidate_from_clobbers+120> addu $a0[4],$zero[0],$s0[16]
   00520fd0:                       : <invalidate_from_clobbers+128> addu $a1[5],$zero[0],$s1[17]
   00520fd8:                       : <invalidate_from_clobbers+130> jal 0050a390 <remove_from_table>
   00520fe0:                       : <invalidate_from_clobbers+138> addu $s0[16],$zero[0],$s2[18]
** 00520fe8:  (      28570,   1.74): <invalidate_from_clobbers+140> bne $s0[16],$zero[0],00520f78 <invalidate_from_clobbers+d0>
   00520ff0:                       : <invalidate_from_clobbers+148> addiu $s3[19],$s3[19],4
```

```
   00520ff8:                       : <invalidate_from_clobbers+150> addiu $s1[17],$s1[17],1
   00521000:                       : <invalidate_from_clobbers+158> slti $v0[2],$s1[17],31
   00521008:  (       1681,   0.10): <invalidate_from_clobbers+160> bne $v0[2],$zero[0],00520f68 <invalidate_from_clobbers+c0>
   cse.c:7179
   00521010:                       : <invalidate_from_clobbers+168> lw $v0[2],0($s7[23])
   00521018:  (         13,   0.00): <invalidate_from_clobbers+170> bgez $v0[2],00521070 <invalidate_from_clobbers+1c8>
   cse.c:7181
   00521020:                       : <invalidate_from_clobbers+178> lw $v1[3],-22800($gp[28])


---- Source Extract [cse.c] [7177]
---- (cse.c)
 7165:    saying which kinds of memory references must be invalidated.
 7166:    X is the pattern of the insn.  */
 7167:
 7168: static void
 7169: invalidate_from_clobbers (w, x)
 7170:      struct write_data *w;
 7171:      rtx x;
 7172: {
 7173:   /* If W->var is not set, W specifies no action.
 7174:      If W->all is set, this step gets all memory refs
 7175:      so they can be ignored in the rest of this function.  */
 7176:   if (w->var)
*7177:     invalidate_memory (w);
 7178:
 7179:   if (w->sp)
 7180:     {
 7181:       if (reg_tick[STACK_POINTER_REGNUM] >= 0)
 7182:  reg_tick[STACK_POINTER_REGNUM]++;
 7183:
 7184:       /* This should be *very* rare.  */
 7185:       if (TEST_HARD_REG_BIT (hard_regs_in_table, STACK_POINTER_REGNUM))
 7186:  invalidate (stack_pointer_rtx);
 7187:     }
 7188:
 7189:   if (GET_CODE (x) == CLOBBER)


 **** #3 miss at 0050c228: cse.c:1566
   0050c1f0:                       : <invalidate+920> lw $s1[17],4($s0[16])
   cse.c:1569
   0050c1f8:                       : <invalidate+928> jal 0050e438 <refers_to_mem_p>
   0050c200:  (        148,   0.01): <invalidate+930> beq $v0[2],$zero[0],0050c220 <invalidate+950>
   cse.c:1570
   0050c208:                       : <invalidate+938> addu $a0[4],$zero[0],$s0[16]
   0050c210:                       : <invalidate+940> addu $a1[5],$zero[0],$s2[18]
   0050c218:                       : <invalidate+948> jal 0050a390 <remove_from_table>
   cse.c:1566
   0050c220:                       : <invalidate+950> addu $s0[16],$zero[0],$s1[17]
** 0050c228:  (      24015,   1.46): <invalidate+958> bne $s0[16],$zero[0],0050c1d0 <invalidate+900>
   cse.c:1563
   0050c230:                       : <invalidate+960> addiu $s3[19],$s3[19],4
   0050c238:                       : <invalidate+968> addiu $s2[18],$s2[18],1
   0050c240:                       : <invalidate+970> slti $v0[2],$s2[18],31
   0050c248:  (       1341,   0.08): <invalidate+978> bne $v0[2],$zero[0],0050c1c0 <invalidate+8f0>
   cse.c:1573
   0050c250:                       : <invalidate+980> lw $ra[31],68($sp[29])
   0050c258:                       : <invalidate+988> lw $s6[22],64($sp[29])
   0050c260:                       : <invalidate+990> lw $s5[21],60($sp[29])


---- Source Extract [cse.c] [1566]
---- (cse.c)
 1554:      a nonvarying address.  Remove all hash table elements
 1555:      that refer to overlapping pieces of memory.  */
 1556:
 1557:   if (GET_CODE (x) != MEM)
 1558:     abort ();
 1559:
 1560:   set_nonvarying_address_components (XEXP (x, 0), GET_MODE_SIZE (GET_MODE (x)),
 1561:       &base, &start, &end);
 1562:
 1563:   for (i = 0; i < NBUCKETS; i++)
 1564:     {
 1565:       register struct table_elt *next;
*1566:       for (p = table[i]; p; p = next)
 1567:  {
 1568:    next = p->next_same_hash;
 1569:    if (refers_to_mem_p (p->exp, base, start, end))
 1570:      remove_from_table (p, i);
 1571:  }
 1572:     }
 1573: }
 1574:
 1575: /* Remove all expressions that refer to register REGNO,
 1576:    since they are already invalid, and we are about to
 1577:    mark that register valid again and don't want the old
 1578:    expressions to reappear as valid.  */


 **** #4 miss at 0050c008: cse.c:1530
```

```
   0050bfd8:                        : <invalidate+708> addiu $s2[18],$s2[18],16768
   cse.c:1526
   0050bfe0:                        : <invalidate+710> lw $a0[4],0($s2[18])
   0050bfe8:  (       6460,   0.39): <invalidate+718> beq $a0[4],$zero[0],0050c0c0 <invalidate+7f0>
   cse.c:1530
   0050bff0:                        : <invalidate+720> lw $v1[3],0($a0[4])
   cse.c:1528
   0050bff8:                        : <invalidate+728> lw $s0[16],4($a0[4])
   cse.c:1530
   0050c000:                        : <invalidate+730> lhu $v0[2],0($v1[3])
** 0050c008:  (      14410,   0.88): <invalidate+738> bne $v0[2],$s6[22],0050c0b0 <invalidate+7e0>
   0050c010:                        : <invalidate+740> lw $a1[5],4($v1[3])
   0050c018:                        : <invalidate+748> slti $v0[2],$a1[5],64
   0050c020:  (       1965,   0.12): <invalidate+750> beq $v0[2],$zero[0],0050c0b0 <invalidate+7e0>
   cse.c:1535
   0050c028:                        : <invalidate+758> lw $v0[2],0($v1[3])
   0050c030:                        : <invalidate+760> sll $v0[2],$v0[2],0x10
   0050c038:                        : <invalidate+768> sra $v0[2],$v0[2],0x18
   0050c040:                        : <invalidate+770> sll $v0[2],$v0[2],0x2
   0050c048:                        : <invalidate+778> addu $v0[2],$v0[2],$s5[21]


---- Source Extract [cse.c] [1530]
---- (cse.c)
 1518:        in_table |= TEST_HARD_REG_BIT (hard_regs_in_table, i);
 1519:        CLEAR_HARD_REG_BIT (hard_regs_in_table, i);
 1520:        delete_reg_equiv (i);
 1521:        reg_tick[i]++;
 1522:      }
 1523:
 1524:    if (in_table)
 1525:      for (hash = 0; hash < NBUCKETS; hash++)
 1526:        for (p = table[hash]; p; p = next)
 1527:  {
 1528:    next = p->next_same_hash;
 1529:
*1530:    if (GET_CODE (p->exp) != REG
 1531:        || REGNO (p->exp) >= FIRST_PSEUDO_REGISTER)
 1532:      continue;
 1533:
 1534:    tregno = REGNO (p->exp);
 1535:    tendregno
 1536:      = tregno + HARD_REGNO_NREGS (tregno, GET_MODE (p->exp));
 1537:    if (tendregno > regno && tregno < endregno)
 1538:    remove_from_table (p, hash);
 1539:  }
 1540:  }
 1541:
 1542:       return;


 **** #5 miss at 0050c0b8: cse.c:1526
   cse.c:1537
   0050c080:                        : <invalidate+7b0> slt $v0[2],$s3[19],$v0[2]
   0050c088:  (        831,   0.05): <invalidate+7b8> beq $v0[2],$zero[0],0050c0b0 <invalidate+7e0>
   0050c090:                        : <invalidate+7c0> slt $v0[2],$a1[5],$s4[20]
   0050c098:  (        513,   0.03): <invalidate+7c8> beq $v0[2],$zero[0],0050c0b0 <invalidate+7e0>
   cse.c:1538
   0050c0a0:                        : <invalidate+7d0> addu $a1[5],$zero[0],$s1[17]
   0050c0a8:                        : <invalidate+7d8> jal 0050a390 <remove_from_table>
   cse.c:1526
   0050c0b0:                        : <invalidate+7e0> addu $a0[4],$zero[0],$s0[16]
** 0050c0b8:  (      12449,   0.76): <invalidate+7e8> bne $a0[4],$zero[0],0050bff0 <invalidate+720>
   cse.c:1525
   0050c0c0:                        : <invalidate+7f0> addiu $s2[18],$s2[18],4
   0050c0c8:                        : <invalidate+7f8> addiu $s1[17],$s1[17],1
   0050c0d0:                        : <invalidate+800> slti $v0[2],$s1[17],31
   0050c0d8:  (       1151,   0.07): <invalidate+808> bne $v0[2],$zero[0],0050bfe0 <invalidate+710>
   cse.c:1542
   0050c0e0:                        : <invalidate+810> j 0050c250 <invalidate+980>
   cse.c:1545
   0050c0e8:                        : <invalidate+818> addiu $v0[2],$zero[0],54


---- Source Extract [cse.c] [1526]
---- (cse.c)
 1514:    CLEAR_HARD_REG_BIT (hard_regs_in_table, regno);
 1515:
 1516:    for (i = regno + 1; i < endregno; i++)
 1517:      {
 1518:        in_table |= TEST_HARD_REG_BIT (hard_regs_in_table, i);
 1519:        CLEAR_HARD_REG_BIT (hard_regs_in_table, i);
 1520:        delete_reg_equiv (i);
 1521:        reg_tick[i]++;
 1522:      }
 1523:
 1524:    if (in_table)
 1525:      for (hash = 0; hash < NBUCKETS; hash++)
*1526:        for (p = table[hash]; p; p = next)
 1527:  {
 1528:    next = p->next_same_hash;
 1529:
```

```
 1530:    if (GET_CODE (p->exp) != REG
 1531:        || REGNO (p->exp) >= FIRST_PSEUDO_REGISTER)
 1532:      continue;
 1533:
 1534:    tregno = REGNO (p->exp);
 1535:    tendregno
 1536:      = tregno + HARD_REGNO_NREGS (tregno, GET_MODE (p->exp));
 1537:    if (tendregno > regno && tregno < endregno)
 1538:    remove_from_table (p, hash);


 **** #6 miss at 005d3140: recog.c:1630
   005d30f0:                      : <constrain_operands+d70> lw $t1[9],372($sp[29])
   005d30f8:                      : <constrain_operands+d78> lw $t0[8],380($sp[29])
   005d3100:                      : <constrain_operands+d80> addiu $s5[21],$s5[21],4
   005d3108:                      : <constrain_operands+d88> addiu $t1[9],$t1[9],4
   005d3110:                      : <constrain_operands+d90> sw $t1[9],372($sp[29])
   005d3118:                      : <constrain_operands+d98> lw $t1[9],268($sp[29])
   005d3120:                      : <constrain_operands+da0> addiu $s6[22],$s6[22],1
   005d3128:                      : <constrain_operands+da8> addiu $t0[8],$t0[8],4
   005d3130:                      : <constrain_operands+db0> sw $t0[8],380($sp[29])
   005d3138:                      : <constrain_operands+db8> slt $v0[2],$s6[22],$t1[9]
** 005d3140:  (      11584,   0.70): <constrain_operands+dc0> bne $v0[2],$zero[0],005d2560 <constrain_operands+1e0>
   recog.c:1881
   005d3148:                      : <constrain_operands+dc8> lw $t0[8],284($sp[29])
   005d3150:  (       3812,   0.23): <constrain_operands+dd0> bne $t0[8],$zero[0],005d33c0 <constrain_operands+1040>
   recog.c:1888
   005d3158:  (         74,   0.00): <constrain_operands+dd8> blez $s7[23],005d3310 <constrain_operands+f90>
   recog.c:1889
   005d3160:                      : <constrain_operands+de0> lw $t1[9],268($sp[29])
   005d3168:                      : <constrain_operands+de8> addu $s3[19],$zero[0],$zero[0]
   005d3170:                      : <constrain_operands+df0> slt $v0[2],$t0[8],$t1[9]


---- Source Extract [recog.c] [1630]
---- (recog.c)
 1618:       matching_operands[c] = -1;
 1619:       op_types[c] = OP_IN;
 1620:     }
 1621:
 1622:   which_alternative = 0;
 1623:
 1624:   while (which_alternative < nalternatives)
 1625:     {
 1626:       register int opno;
 1627:       int lose = 0;
 1628:       funny_match_index = 0;
 1629:
*1630:       for (opno = 0; opno < noperands; opno++)
 1631:  {
 1632:    register rtx op = recog_operand[opno];
 1633:    enum machine_mode mode = GET_MODE (op);
 1634:    register char *p = constraints[opno];
 1635:    int offset = 0;
 1636:    int win = 0;
 1637:    int val;
 1638:
 1639:    earlyclobber[opno] = 0;
 1640:
 1641:    if (GET_CODE (op) == SUBREG)
 1642:      {


 **** #7 miss at 0056d310: regclass.c:1165
   0056d2d8:                      : <record_reg_classes+c08> sw $v0[2],0($t7[15])
   regclass.c:1154
   0056d2e0:                      : <record_reg_classes+c10> lb $v0[2],0($s3[19])
   0056d2e8:                      : <record_reg_classes+c18> lbu $v1[3],0($s3[19])
   0056d2f0:  (       6683,   0.41): <record_reg_classes+c20> bne $v0[2],$zero[0],0056cc28 <record_reg_classes+558>
   regclass.c:1156
   0056d2f8:                      : <record_reg_classes+c28> sw $s3[19],0($s8[30])
   regclass.c:1165
   0056d300:                      : <record_reg_classes+c30> lhu $v0[2],0($s0[16])
   0056d308:                      : <record_reg_classes+c38> addiu $t6[14],$zero[0],52
** 0056d310:  (      10785,   0.66): <record_reg_classes+c40> bne $v0[2],$t6[14],0056d468 <record_reg_classes+d98>
   0056d318:                      : <record_reg_classes+c48> lw $v0[2],4($s0[16])
   0056d320:                      : <record_reg_classes+c50> slti $v0[2],$v0[2],64
   0056d328:  (       2287,   0.14): <record_reg_classes+c58> bne $v0[2],$zero[0],0056d468 <record_reg_classes+d98>
   regclass.c:1167
   0056d330:                      : <record_reg_classes+c60> lw $t7[15],444($sp[29])
   0056d338:                      : <record_reg_classes+c68> lw $t6[14],388($sp[29])
   0056d340:                      : <record_reg_classes+c70> addu $v0[2],$t7[15],$t6[14]
   0056d348:                      : <record_reg_classes+c78> lw $v0[2],240($v0[2])
   0056d350:  (       1929,   0.12): <record_reg_classes+c80> beq $v0[2],$zero[0],0056d5d8 <record_reg_classes+f08>


---- Source Extract [regclass.c] [1165]
---- (regclass.c)
 1153:      [(int) REG_CLASS_FROM_LETTER (c)];
 1154:        }
 1155:
 1156:    constraints[i] = p;
```

```
 1157:
 1158:    /* How we account for this operand now depends on whether it is  a
 1159:       pseudo register or not.  If it is, we first check if any
 1160:       register classes are valid.  If not, we ignore this alternative,
 1161:       since we want to assume that all pseudos get allocated for
 1162:       register preferencing.  If some register class is valid, compute
 1163:       the costs of moving the pseudo into that class.  */
 1164:
*1165:    if (GET_CODE (op) == REG && REGNO (op) >= FIRST_PSEUDO_REGISTER)
 1166:      {
 1167:        if (classes[i] == NO_REGS)
 1168:  alt_fail = 1;
 1169:        else
 1170:  {
 1171:    struct costs *pp = &this_op_costs[i];
 1172:
 1173:    for (class = 0; class < N_REG_CLASSES; class++)
 1174:      pp->cost[class] = may_move_cost[class][(int) classes[i]];
 1175:
 1176:    /* If the alternative actually allows memory, make things
 1177:       a bit cheaper since we won't need an extra insn to


 **** #8 miss at 005d26a0: recog.c:1654
   005d2650:  (        283,   0.02): <constrain_operands+2d0> beq $v0[2],$zero[0],005d30d0 <constrain_operands+d50>
   005d2658:                       : <constrain_operands+2d8> lw $t1[9],372($sp[29])
   005d2660:                       : <constrain_operands+2e0> lw $t0[8],380($sp[29])
   005d2668:                       : <constrain_operands+2e8> addu $s4[20],$zero[0],$s5[21]
   005d2670:                       : <constrain_operands+2f0> sw $t1[9],316($sp[29])
   005d2678:                       : <constrain_operands+2f8> sw $t0[8],324($sp[29])
   005d2680:                       : <constrain_operands+300> sll $v0[2],$v1[3],0x18
   005d2688:                       : <constrain_operands+308> sra $s0[16],$v0[2],0x18
   005d2690:                       : <constrain_operands+310> addiu $s3[19],$s3[19],1
   005d2698:                       : <constrain_operands+318> addiu $v0[2],$zero[0],44
** 005d26a0:  (      10725,   0.65): <constrain_operands+320> beq $s0[16],$v0[2],005d30d0 <constrain_operands+d50>
   recog.c:1655
   005d26a8:                       : <constrain_operands+328> addiu $v1[3],$s0[16],-33
   005d26b0:                       : <constrain_operands+330> sltiu $v0[2],$v1[3],83
   005d26b8:  (         56,   0.00): <constrain_operands+338> beq $v0[2],$zero[0],005d2ff0 <constrain_operands+c70>
   005d26c0:                       : <constrain_operands+340> sll $v0[2],$v1[3],0x2
   005d26c8:                       : <constrain_operands+348> lui $at[1],4097
   005d26d0:                       : <constrain_operands+350> addu $at[1],$at[1],$v0[2]
   005d26d8:                       : <constrain_operands+358> lw $v0[2],29608($at[1])
   005d26e0:                       : <constrain_operands+360> jr $v0[2]


---- Source Extract [recog.c] [1654]
---- (recog.c)
 1642:      {
 1643:        if (GET_CODE (SUBREG_REG (op)) == REG
 1644:    && REGNO (SUBREG_REG (op)) < FIRST_PSEUDO_REGISTER)
 1645:  offset = SUBREG_WORD (op);
 1646:        op = SUBREG_REG (op);
 1647:      }
 1648:
 1649:    /* An empty constraint or empty alternative
 1650:       allows anything which matched the pattern.  */
 1651:    if (*p == 0 || *p == ',')
 1652:      win = 1;
 1653:
*1654:    while (*p && (c = *p++) != ',')
 1655:      switch (c)
 1656:        {
 1657:        case '?':
 1658:        case '#':
 1659:        case '!':
 1660:        case '*':
 1661:        case '%':
 1662:  break;
 1663:
 1664:        case '=':
 1665:  op_types[opno] = OP_OUT;
 1666:  break;


 **** #9 miss at 005cfd18: recog.c:873
   005cfce0:                       : <register_operand+88> bne $v1[3],$v0[2],005cfcf8 <register_operand+a0>
   recog.c:867
   005cfce8:                       : <register_operand+90> jal 005cf290 <general_operand>
   005cfcf0:                       : <register_operand+98> j 005cfd60 <register_operand+108>
   recog.c:868
   005cfcf8:                       : <register_operand+a0> lw $a0[4],4($a0[4])
   recog.c:873
   005cfd00:                       : <register_operand+a8> lhu $v1[3],0($a0[4])
   005cfd08:                       : <register_operand+b0> addu $a1[5],$zero[0],$zero[0]
   005cfd10:                       : <register_operand+b8> addiu $v0[2],$zero[0],52
** 005cfd18:  (      10382,   0.63): <register_operand+c0> bne $v1[3],$v0[2],005cfd58 <register_operand+100>
   005cfd20:                       : <register_operand+c8> lw $a0[4],4($a0[4])
   005cfd28:                       : <register_operand+d0> slti $v0[2],$a0[4],64
   005cfd30:  (       1992,   0.12): <register_operand+d8> beq $v0[2],$zero[0],005cfd50 <register_operand+f8>
   005cfd38:                       : <register_operand+e0> slti $v0[2],$a0[4],32
   005cfd40:  (        392,   0.02): <register_operand+e8> beq $v0[2],$zero[0],005cfd50 <register_operand+f8>
```

```
   005cfd48:  (        229,   0.01): <register_operand+f0> beq $a0[4],$zero[0],005cfd58 <register_operand+100>
   005cfd50:                        : <register_operand+f8> addiu $a1[5],$zero[0],1
   005cfd58:                        : <register_operand+100> addu $v0[2],$zero[0],$a1[5]
   005cfd60:                        : <register_operand+108> lw $ra[31],16($sp[29])


---- Source Extract [recog.c] [873]
---- (recog.c)
 861:   because it is guaranteed to be reloaded into one.
 862:   Just make sure the MEM is valid in itself.
 863:   (Ideally, (SUBREG (MEM)...) should not exist after reload,
 864:   but currently it does result from (SUBREG (REG)...) where the
 865:   reg went on the stack.)  */
 866:       if (! reload_completed && GET_CODE (SUBREG_REG (op)) == MEM)
 867:  return general_operand (op, mode);
 868:       op = SUBREG_REG (op);
 869:     }
 870:
 871:   /* We don't consider registers whose class is NO_REGS
 872:      to be a register operand.  */
*873:   return (GET_CODE (op) == REG
 874:    && (REGNO (op) >= FIRST_PSEUDO_REGISTER
 875:        || REGNO_REG_CLASS (REGNO (op)) != NO_REGS));
 876: }
 877:
 878: /* Return 1 if OP should match a MATCH_SCRATCH, i.e., if it is a SCRATCH
 879:    or a hard register.  */
 880:
 881: int
 882: scratch_operand (op, mode)
 883:      register rtx op;
 884:      enum machine_mode mode;
 885: {


 **** #10 miss at 0050ea90: cse.c:2415
   cse.c:2411
   0050ea60:  (         35,   0.00): <canon_reg+68> bne $s5[21],$zero[0],0050ea78 <canon_reg+80>
   cse.c:2412
   0050ea68:                        : <canon_reg+70> addu $v0[2],$zero[0],$zero[0]
   0050ea70:                        : <canon_reg+78> j 0050ee30 <canon_reg+438>
   cse.c:2414
   0050ea78:                        : <canon_reg+80> lhu $a0[4],0($s5[21])
   cse.c:2415
   0050ea80:                        : <canon_reg+88> addiu $v1[3],$a0[4],-39
   0050ea88:                        : <canon_reg+90> sltiu $v0[2],$v1[3],22
** 0050ea90:  (       9637,   0.59): <canon_reg+98> beq $v0[2],$zero[0],0050ebd0 <canon_reg+1d8>
   0050ea98:                        : <canon_reg+a0> sll $v0[2],$v1[3],0x2
   0050eaa0:                        : <canon_reg+a8> lui $at[1],4097
   0050eaa8:                        : <canon_reg+b0> addu $at[1],$at[1],$v0[2]
   0050eab0:                        : <canon_reg+b8> lw $v0[2],9656($at[1])
   0050eab8:                        : <canon_reg+c0> jr $v0[2]
   cse.c:2437
   0050eac0:                        : <canon_reg+c8> lw $a0[4],4($s5[21])
   0050eac8:                        : <canon_reg+d0> slti $v0[2],$a0[4],64
   0050ead0:  (       3733,   0.23): <canon_reg+d8> bne $v0[2],$zero[0],0050ee28 <canon_reg+430>


---- Source Extract [cse.c] [2415]
---- (cse.c)
 2403: canon_reg (x, insn)
 2404:      rtx x;
 2405:      rtx insn;
 2406: {
 2407:   register int i;
 2408:   register enum rtx_code code;
 2409:   register char *fmt;
 2410:
 2411:   if (x == 0)
 2412:     return x;
 2413:
 2414:   code = GET_CODE (x);
*2415:   switch (code)
 2416:     {
 2417:     case PC:
 2418:     case CC0:
 2419:     case CONST:
 2420:     case CONST_INT:
 2421:     case CONST_DOUBLE:
 2422:     case SYMBOL_REF:
 2423:     case LABEL_REF:
 2424:     case ADDR_VEC:
 2425:     case ADDR_DIFF_VEC:
 2426:       return x;
 2427:


 **** #11 miss at 004d6f38: rtlanal.c:930
   004d6f08:                        : <rtx_equal_p+48> sw $s0[16],24($sp[29])
   rtlanal.c:923
   004d6f10:  (       6938,   0.42): <rtx_equal_p+50> beq $a0[4],$a1[5],004d7270 <rtx_equal_p+3b0>
   rtlanal.c:925
```

```
   004d6f18:  (         73,   0.00): <rtx_equal_p+58> beq $a0[4],$zero[0],004d7120 <rtx_equal_p+260>
   004d6f20:  (         63,   0.00): <rtx_equal_p+60> beq $a1[5],$zero[0],004d7120 <rtx_equal_p+260>
   rtlanal.c:928
   004d6f28:                       : <rtx_equal_p+68> lhu $a2[6],0($a0[4])
   rtlanal.c:930
   004d6f30:                       : <rtx_equal_p+70> lhu $v0[2],0($a1[5])
** 004d6f38:  (       8527,   0.52): <rtx_equal_p+78> bne $a2[6],$v0[2],004d7120 <rtx_equal_p+260>
   rtlanal.c:936
   004d6f40:                       : <rtx_equal_p+80> lb $v1[3],2($a0[4])
   004d6f48:                       : <rtx_equal_p+88> lb $v0[2],2($a1[5])
   004d6f50:  (        549,   0.03): <rtx_equal_p+90> bne $v1[3],$v0[2],004d7120 <rtx_equal_p+260>
   rtlanal.c:941
   004d6f58:                       : <rtx_equal_p+98> addiu $v0[2],$zero[0],52
   004d6f60:  (       2577,   0.16): <rtx_equal_p+a0> bne $a2[6],$v0[2],004d6fd8 <rtx_equal_p+118>
   rtlanal.c:946
   004d6f68:                       : <rtx_equal_p+a8> lw $v1[3],4($a0[4])
```

```
---- Source Extract [rtlanal.c] [930]
---- (rtlanal.c)
 918:   register int i;
 919:   register int j;
 920:   register enum rtx_code code;
 921:   register char *fmt;
 922:
 923:   if (x == y)
 924:     return 1;
 925:   if (x == 0 || y == 0)
 926:     return 0;
 927:
 928:   code = GET_CODE (x);
 929:   /* Rtx's of different codes cannot be equal.  */
*930:   if (code != GET_CODE (y))
 931:     return 0;
 932:
 933:   /* (MULT:SI x y) and (MULT:HI x y) are NOT equivalent.
 934:      (REG:SI x) and (REG:HI x) are NOT equivalent.  */
 935:
 936:   if (GET_MODE (x) != GET_MODE (y))
 937:     return 0;
 938:
 939:   /* REG, LABEL_REF, and SYMBOL_REF can be compared nonrecursively.  */
 940:
 941:   if (code == REG)
 942:     /* Until rtl generation is complete, don't consider a reference to the
```

```
 **** #12 miss at 005d2960: recog.c:1736
   005d2910:                       : <constrain_operands+590> lhu $v1[3],0($s1[17])
   005d2918:                       : <constrain_operands+598> addiu $t1[9],$zero[0],52
   005d2920:  (       3585,   0.22): <constrain_operands+5a0> bne $v1[3],$t1[9],005d2940 <constrain_operands+5c0>
   005d2928:                       : <constrain_operands+5a8> lw $v0[2],4($s1[17])
   005d2930:                       : <constrain_operands+5b0> slti $v0[2],$v0[2],64
   005d2938:  (        338,   0.02): <constrain_operands+5b8> beq $v0[2],$zero[0],005d30b0 <constrain_operands+d30>
   005d2940:                       : <constrain_operands+5c0> addiu $v0[2],$zero[0],53
   005d2948:  (         12,   0.00): <constrain_operands+5c8> beq $v1[3],$v0[2],005d30b0 <constrain_operands+d30>
   005d2950:                       : <constrain_operands+5d0> lhu $v0[2],0($s1[17])
   005d2958:                       : <constrain_operands+5d8> addiu $t0[8],$zero[0],52
** 005d2960:  (       8414,   0.51): <constrain_operands+5e0> bne $v0[2],$t0[8],005d30b8 <constrain_operands+d38>
   005d2968:                       : <constrain_operands+5e8> lw $a2[6],300($sp[29])
   005d2970:                       : <constrain_operands+5f0> lw $a3[7],292($sp[29])
   005d2978:                       : <constrain_operands+5f8> addu $a0[4],$zero[0],$s1[17]
   005d2980:                       : <constrain_operands+600> addiu $a1[5],$zero[0],1
   recog.c:1747
   005d2988:                       : <constrain_operands+608> j 005d30a0 <constrain_operands+d20>
   recog.c:1756
   005d2990:                       : <constrain_operands+610> lhu $a0[4],0($s1[17])
   005d2998:                       : <constrain_operands+618> addiu $t1[9],$zero[0],57
```

```
---- Source Extract [recog.c] [1736]
---- (recog.c)
 1724:  /* Anything goes unless it is a REG and really has a hard reg
 1725:     but the hard reg is not in the class GENERAL_REGS.  */
 1726:  if (strict < 0
 1727:      || GENERAL_REGS == ALL_REGS
 1728:      || GET_CODE (op) != REG
 1729:      || (reload_in_progress
 1730:  && REGNO (op) >= FIRST_PSEUDO_REGISTER)
 1731:      || reg_fits_class_p (op, GENERAL_REGS, offset, mode))
 1732:    win = 1;
 1733:  break;
 1734:
 1735:        case 'r':
*1736:  if (strict < 0
 1737:      || (strict == 0
 1738:  && GET_CODE (op) == REG
 1739:  && REGNO (op) >= FIRST_PSEUDO_REGISTER)
 1740:      || (strict == 0 && GET_CODE (op) == SCRATCH)
 1741:      || (GET_CODE (op) == REG
 1742:  && ((GENERAL_REGS == ALL_REGS
 1743:       && REGNO (op) < FIRST_PSEUDO_REGISTER)
```

```
 1744:      || reg_fits_class_p (op, GENERAL_REGS,
 1745:   offset, mode))))
 1746:    win = 1;
 1747:  break;
 1748:


 **** #13 miss at 004d7c98: rtlanal.c:1202
   find_reg_note():
   rtlanal.c:1202
   004d7c90:                        : <find_reg_note> lw $v1[3],28($a0[4])
** 004d7c98:  (       7984,   0.49): <find_reg_note+8> beq $v1[3],$zero[0],004d7cf8 <find_reg_note+68>
   rtlanal.c:1203
   004d7ca0:                        : <find_reg_note+10> lw $v0[2],0($v1[3])
   004d7ca8:                        : <find_reg_note+18> sll $v0[2],$v0[2],0x10
   004d7cb0:                        : <find_reg_note+20> sra $v0[2],$v0[2],0x18
   004d7cb8:  (       5648,   0.34): <find_reg_note+28> bne $v0[2],$a1[5],004d7ce8 <find_reg_note+58>
   004d7cc0:  (        697,   0.04): <find_reg_note+30> beq $a2[6],$zero[0],004d7cd8 <find_reg_note+48>
   004d7cc8:                        : <find_reg_note+38> lw $v0[2],4($v1[3])
   004d7cd0:  (        508,   0.03): <find_reg_note+40> bne $a2[6],$v0[2],004d7ce8 <find_reg_note+58>
   rtlanal.c:1205


---- Source Extract [rtlanal.c] [1202]
---- (rtlanal.c)
 1190:
 1191: /* Return the reg-note of kind KIND in insn INSN, if there is one.
 1192:    If DATUM is nonzero, look for one whose datum is DATUM.  */
 1193:
 1194: rtx
 1195: find_reg_note (insn, kind, datum)
 1196:      rtx insn;
 1197:      enum reg_note kind;
 1198:      rtx datum;
 1199: {
 1200:   register rtx link;
 1201:
*1202:   for (link = REG_NOTES (insn); link; link = XEXP (link, 1))
 1203:     if (REG_NOTE_KIND (link) == kind
 1204:  && (datum == 0 || datum == XEXP (link, 0)))
 1205:       return link;
 1206:   return 0;
 1207: }
 1208:
 1209: /* Return the reg-note of kind KIND in insn INSN which applies to register
 1210:    number REGNO, if any.  Return 0 if there is no such reg-note.  Note that
 1211:    the REGNO of this NOTE need not be REGNO if REGNO is a hard register;
 1212:    it might be the case that the note overlaps REGNO.  */
 1213:
 1214: rtx


 **** #14 miss at 00520f70: cse.c:7177
   00520f28:  (       1727,   0.10): <invalidate_from_clobbers+80> beq $v0[2],$zero[0],00521010 <invalidate_from_clobbers+168>
   cse.c:7177
   00520f30:                        : <invalidate_from_clobbers+88> sll $s5[21],$v1[3],0x3
   00520f38:                        : <invalidate_from_clobbers+90> sra $s5[21],$s5[21],0x1f
   00520f40:                        : <invalidate_from_clobbers+98> sll $s4[20],$v1[3],0x2
   00520f48:                        : <invalidate_from_clobbers+a0> sra $s4[20],$s4[20],0x1f
   00520f50:                        : <invalidate_from_clobbers+a8> addu $s1[17],$zero[0],$zero[0]
   00520f58:                        : <invalidate_from_clobbers+b0> lui $s3[19],4099
   00520f60:                        : <invalidate_from_clobbers+b8> addiu $s3[19],$s3[19],16768
   00520f68:                        : <invalidate_from_clobbers+c0> lw $s0[16],0($s3[19])
** 00520f70:  (       7922,   0.48): <invalidate_from_clobbers+c8> beq $s0[16],$zero[0],00520ff0 <invalidate_from_clobbers+148>
   00520f78:                        : <invalidate_from_clobbers+d0> lb $v0[2],36($s0[16])
   00520f80:                        : <invalidate_from_clobbers+d8> lw $s2[18],4($s0[16])
   00520f88:  (       7833,   0.48): <invalidate_from_clobbers+e0> beq $v0[2],$zero[0],00520fe0 <invalidate_from_clobbers+138>
   00520f90:  (         71,   0.00): <invalidate_from_clobbers+e8> bne $s5[21],$zero[0],00520fc8 <invalidate_from_clobbers+120>
   00520f98:  (        322,   0.02): <invalidate_from_clobbers+f0> beq $s4[20],$zero[0],00520fb0 <invalidate_from_clobbers+108>
   00520fa0:                        : <invalidate_from_clobbers+f8> lb $v0[2],37($s0[16])
   00520fa8:  (        300,   0.02): <invalidate_from_clobbers+100> bne $v0[2],$zero[0],00520fc8 <invalidate_from_clobbers+120>
   00520fb0:                        : <invalidate_from_clobbers+108> lw $a0[4],0($s0[16])
   00520fb8:                        : <invalidate_from_clobbers+110> jal 0050e7d8 <cse_rtx_addr_varies_p>


---- Source Extract [cse.c] [7177]
---- (cse.c)
 7165:    saying which kinds of memory references must be invalidated.
 7166:    X is the pattern of the insn.  */
 7167:
 7168: static void
 7169: invalidate_from_clobbers (w, x)
 7170:      struct write_data *w;
 7171:      rtx x;
 7172: {
 7173:   /* If W->var is not set, W specifies no action.
 7174:      If W->all is set, this step gets all memory refs
 7175:      so they can be ignored in the rest of this function.  */
 7176:   if (w->var)
*7177:     invalidate_memory (w);
 7178:
 7179:   if (w->sp)
```

```
 7180:     {
 7181:       if (reg_tick[STACK_POINTER_REGNUM] >= 0)
 7182:  reg_tick[STACK_POINTER_REGNUM]++;
 7183:
 7184:       /* This should be *very* rare.  */
 7185:       if (TEST_HARD_REG_BIT (hard_regs_in_table, STACK_POINTER_REGNUM))
 7186:  invalidate (stack_pointer_rtx);
 7187:     }
 7188:
 7189:   if (GET_CODE (x) == CLOBBER)


 **** #15 miss at 00520f88: cse.c:7177
   00520f38:                     : <invalidate_from_clobbers+90> sra $s5[21],$s5[21],0x1f
   00520f40:                     : <invalidate_from_clobbers+98> sll $s4[20],$v1[3],0x2
   00520f48:                     : <invalidate_from_clobbers+a0> sra $s4[20],$s4[20],0x1f
   00520f50:                     : <invalidate_from_clobbers+a8> addu $s1[17],$zero[0],$zero[0]
   00520f58:                     : <invalidate_from_clobbers+b0> lui $s3[19],4099
   00520f60:                     : <invalidate_from_clobbers+b8> addiu $s3[19],$s3[19],16768
   00520f68:                     : <invalidate_from_clobbers+c0> lw $s0[16],0($s3[19])
   00520f70:  (      7922,   0.48): <invalidate_from_clobbers+c8> beq $s0[16],$zero[0],00520ff0 <invalidate_from_clobbers+148>
   00520f78:                     : <invalidate_from_clobbers+d0> lb $v0[2],36($s0[16])
   00520f80:                     : <invalidate_from_clobbers+d8> lw $s2[18],4($s0[16])
** 00520f88:  (      7833,   0.48): <invalidate_from_clobbers+e0> beq $v0[2],$zero[0],00520fe0 <invalidate_from_clobbers+138>
   00520f90:  (        71,   0.00): <invalidate_from_clobbers+e8> bne $s5[21],$zero[0],00520fc8 <invalidate_from_clobbers+120>
   00520f98:  (       322,   0.02): <invalidate_from_clobbers+f0> beq $s4[20],$zero[0],00520fb0 <invalidate_from_clobbers+108>
   00520fa0:                     : <invalidate_from_clobbers+f8> lb $v0[2],37($s0[16])
   00520fa8:  (       300,   0.02): <invalidate_from_clobbers+100> bne $v0[2],$zero[0],00520fc8 <invalidate_from_clobbers+120>
   00520fb0:                     : <invalidate_from_clobbers+108> lw $a0[4],0($s0[16])
   00520fb8:                     : <invalidate_from_clobbers+110> jal 0050e7d8 <cse_rtx_addr_varies_p>
   00520fc0:  (       393,   0.02): <invalidate_from_clobbers+118> beq $v0[2],$zero[0],00520fe0 <invalidate_from_clobbers+138>
   00520fc8:                     : <invalidate_from_clobbers+120> addu $a0[4],$zero[0],$s0[16]
   00520fd0:                     : <invalidate_from_clobbers+128> addu $a1[5],$zero[0],$s1[17]


---- Source Extract [cse.c] [7177]
---- (cse.c)
 7165:    saying which kinds of memory references must be invalidated.
 7166:    X is the pattern of the insn.  */
 7167:
 7168: static void
 7169: invalidate_from_clobbers (w, x)
 7170:      struct write_data *w;
 7171:      rtx x;
 7172: {
 7173:   /* If W->var is not set, W specifies no action.
 7174:      If W->all is set, this step gets all memory refs
 7175:      so they can be ignored in the rest of this function.  */
 7176:   if (w->var)
*7177:     invalidate_memory (w);
 7178:
 7179:   if (w->sp)
 7180:     {
 7181:       if (reg_tick[STACK_POINTER_REGNUM] >= 0)
 7182:  reg_tick[STACK_POINTER_REGNUM]++;
 7183:
 7184:       /* This should be *very* rare.  */
 7185:       if (TEST_HARD_REG_BIT (hard_regs_in_table, STACK_POINTER_REGNUM))
 7186:  invalidate (stack_pointer_rtx);
 7187:     }
 7188:
 7189:   if (GET_CODE (x) == CLOBBER)


 **** #16 miss at 0050d688: cse.c:2024
   0050d640:                     : <exp_equiv_p+30> addu $s7[23],$zero[0],$a2[6]
   0050d648:                     : <exp_equiv_p+38> sw $s6[22],48($sp[29])
   0050d650:                     : <exp_equiv_p+40> addu $s6[22],$zero[0],$a3[7]
   0050d658:                     : <exp_equiv_p+48> sw $ra[31],60($sp[29])
   0050d660:                     : <exp_equiv_p+50> sw $s8[30],56($sp[29])
   0050d668:                     : <exp_equiv_p+58> sw $s5[21],44($sp[29])
   0050d670:                     : <exp_equiv_p+60> sw $s4[20],40($sp[29])
   0050d678:                     : <exp_equiv_p+68> sw $s3[19],36($sp[29])
   0050d680:                     : <exp_equiv_p+70> sw $s2[18],32($sp[29])
   cse.c:2024
** 0050d688:  (      7518,   0.46): <exp_equiv_p+78> bne $s1[17],$s0[16],0050d698 <exp_equiv_p+88>
   0050d690:  (      2716,   0.17): <exp_equiv_p+80> beq $s7[23],$zero[0],0050dea0 <exp_equiv_p+890>
   cse.c:2026
   0050d698:  (        18,   0.00): <exp_equiv_p+88> beq $s1[17],$zero[0],0050d998 <exp_equiv_p+388>
   0050d6a0:  (        10,   0.00): <exp_equiv_p+90> beq $s0[16],$zero[0],0050d998 <exp_equiv_p+388>
   cse.c:2029
   0050d6a8:                     : <exp_equiv_p+98> lhu $s2[18],0($s1[17])
   cse.c:2030
   0050d6b0:                     : <exp_equiv_p+a0> lhu $v0[2],0($s0[16])
   0050d6b8:  (      6668,   0.41): <exp_equiv_p+a8> beq $s2[18],$v0[2],0050d940 <exp_equiv_p+330>


---- Source Extract [cse.c] [2024]
---- (cse.c)
 2012: static int
 2013: exp_equiv_p (x, y, validate, equal_values)
 2014:      rtx x, y;
```

```
 2015:      int validate;
 2016:      int equal_values;
 2017: {
 2018:   register int i, j;
 2019:   register enum rtx_code code;
 2020:   register char *fmt;
 2021:
 2022:   /* Note: it is incorrect to assume an expression is equivalent to itself
 2023:      if VALIDATE is nonzero.  */
*2024:   if (x == y && !validate)
 2025:     return 1;
 2026:   if (x == 0 || y == 0)
 2027:     return x == y;
 2028:
 2029:   code = GET_CODE (x);
 2030:   if (code != GET_CODE (y))
 2031:     {
 2032:       if (!equal_values)
 2033:  return 0;
 2034:
 2035:       /* If X is a constant and Y is a register or vice versa, they may be
 2036:   equivalent.  We only have to validate if Y is a register.  */


 **** #17 miss at 00400560: bison.simple:355
   00400520:                       : <yyparse+330> addiu $a1[5],$a1[5],19064
   00400528:                       : <yyparse+338> addu $a2[6],$zero[0],$s3[19]
   00400530:                       : <yyparse+340> jal 005fe0c0 <fprintf>
   bison.simple:354
   00400538:                       : <yyparse+348> sll $v0[2],$s3[19],0x1
   00400540:                       : <yyparse+350> lui $s5[21],4096
   00400548:                       : <yyparse+358> addu $s5[21],$s5[21],$v0[2]
   00400550:                       : <yyparse+360> lh $s5[21],9412($s5[21])
   bison.simple:355
   00400558:                       : <yyparse+368> addiu $v0[2],$zero[0],-32768
** 00400560:  (       7358,   0.45): <yyparse+370> beq $s5[21],$v0[2],00400858 <yyparse+668>
   bison.simple:363
   00400568:                       : <yyparse+378> lw $v1[3],-21476($gp[28])
   00400570:                       : <yyparse+380> addiu $v0[2],$zero[0],-2
   00400578:  (       3211,   0.20): <yyparse+388> bne $v1[3],$v0[2],004005c0 <yyparse+3d0>
   bison.simple:366
   00400580:                       : <yyparse+390> lw $v0[2],-21484($gp[28])
   00400588:  (         48,   0.00): <yyparse+398> beq $v0[2],$zero[0],004005b0 <yyparse+3c0>
   bison.simple:367
   00400590:                       : <yyparse+3a0> lw $a0[4],-23992($gp[28])


---- Source Extract [bison.simple] [355]
---- (bison.simple)
!! Could not open /pong/usr3/s/smithz/research/bench/spec95-src/126.gcc/src/bison.simple !


 **** #18 miss at 004d6f10: rtlanal.c:923
   004d6ec8:                       : <rtx_equal_p+8>  sw $ra[31],56($sp[29])
   004d6ed0:                       : <rtx_equal_p+10> sw $s7[23],52($sp[29])
   004d6ed8:                       : <rtx_equal_p+18> sw $s6[22],48($sp[29])
   004d6ee0:                       : <rtx_equal_p+20> sw $s5[21],44($sp[29])
   004d6ee8:                       : <rtx_equal_p+28> sw $s4[20],40($sp[29])
   004d6ef0:                       : <rtx_equal_p+30> sw $s3[19],36($sp[29])
   004d6ef8:                       : <rtx_equal_p+38> sw $s2[18],32($sp[29])
   004d6f00:                       : <rtx_equal_p+40> sw $s1[17],28($sp[29])
   004d6f08:                       : <rtx_equal_p+48> sw $s0[16],24($sp[29])
   rtlanal.c:923
** 004d6f10:  (       6938,   0.42): <rtx_equal_p+50> beq $a0[4],$a1[5],004d7270 <rtx_equal_p+3b0>
   rtlanal.c:925
   004d6f18:  (         73,   0.00): <rtx_equal_p+58> beq $a0[4],$zero[0],004d7120 <rtx_equal_p+260>
   004d6f20:  (         63,   0.00): <rtx_equal_p+60> beq $a1[5],$zero[0],004d7120 <rtx_equal_p+260>
   rtlanal.c:928
   004d6f28:                       : <rtx_equal_p+68> lhu $a2[6],0($a0[4])
   rtlanal.c:930
   004d6f30:                       : <rtx_equal_p+70> lhu $v0[2],0($a1[5])
   004d6f38:  (       8527,   0.52): <rtx_equal_p+78> bne $a2[6],$v0[2],004d7120 <rtx_equal_p+260>
   rtlanal.c:936


---- Source Extract [rtlanal.c] [923]
---- (rtlanal.c)
 911: /* Return 1 if X and Y are identical-looking rtx's.
 912:    This is the Lisp function EQUAL for rtx arguments.  */
 913:
 914: int
 915: rtx_equal_p (x, y)
 916:      rtx x, y;
 917: {
 918:   register int i;
 919:   register int j;
 920:   register enum rtx_code code;
 921:   register char *fmt;
 922:
*923:   if (x == y)
 924:     return 1;
 925:   if (x == 0 || y == 0)
```

```
 926:     return 0;
 927:
 928:   code = GET_CODE (x);
 929:   /* Rtx's of different codes cannot be equal.  */
 930:   if (code != GET_CODE (y))
 931:     return 0;
 932:
 933:   /* (MULT:SI x y) and (MULT:HI x y) are NOT equivalent.
 934:      (REG:SI x) and (REG:HI x) are NOT equivalent.  */
 935:


 **** #19 miss at 0056d2f0: regclass.c:1154
   0056d2a8:                       : <record_reg_classes+bd8> bne $a0[4],$v0[2],0056d2c8 <record_reg_classes+bf8>
   0056d2b0:                       : <record_reg_classes+be0> addiu $v1[3],$v1[3],4
   0056d2b8:                       : <record_reg_classes+be8> j 0056d2c8 <record_reg_classes+bf8>
   0056d2c0:                       : <record_reg_classes+bf0> addiu $v1[3],$v1[3],8
   0056d2c8:                       : <record_reg_classes+bf8> lw $v0[2],0($v1[3])
   0056d2d0:                       : <record_reg_classes+c00> lw $t7[15],380($sp[29])
   0056d2d8:                       : <record_reg_classes+c08> sw $v0[2],0($t7[15])
   regclass.c:1154
   0056d2e0:                       : <record_reg_classes+c10> lb $v0[2],0($s3[19])
   0056d2e8:                       : <record_reg_classes+c18> lbu $v1[3],0($s3[19])
** 0056d2f0:  (       6683,   0.41): <record_reg_classes+c20> bne $v0[2],$zero[0],0056cc28 <record_reg_classes+558>
   regclass.c:1156
   0056d2f8:                       : <record_reg_classes+c28> sw $s3[19],0($s8[30])
   regclass.c:1165
   0056d300:                       : <record_reg_classes+c30> lhu $v0[2],0($s0[16])
   0056d308:                       : <record_reg_classes+c38> addiu $t6[14],$zero[0],52
   0056d310:  (      10785,   0.66): <record_reg_classes+c40> bne $v0[2],$t6[14],0056d468 <record_reg_classes+d98>
   0056d318:                       : <record_reg_classes+c48> lw $v0[2],4($s0[16])
   0056d320:                       : <record_reg_classes+c50> slti $v0[2],$v0[2],64
   0056d328:  (       2287,   0.14): <record_reg_classes+c58> bne $v0[2],$zero[0],0056d468 <record_reg_classes+d98>


---- Source Extract [regclass.c] [1154]
---- (regclass.c)
 1142:  ))
 1143:    win = 1;
 1144:  allows_mem = 1;
 1145:        case 'r':
 1146:  classes[i]
 1147:    = reg_class_subunion[(int) classes[i]][(int) GENERAL_REGS];
 1148:  break;
 1149:
 1150:        default:
 1151:  classes[i]
 1152:    = reg_class_subunion[(int) classes[i]]
 1153:      [(int) REG_CLASS_FROM_LETTER (c)];
*1154:        }
 1155:
 1156:    constraints[i] = p;
 1157:
 1158:    /* How we account for this operand now depends on whether it is  a
 1159:       pseudo register or not.  If it is, we first check if any
 1160:       register classes are valid.  If not, we ignore this alternative,
 1161:       since we want to assume that all pseudos get allocated for
 1162:       register preferencing.  If some register class is valid, compute
 1163:       the costs of moving the pseudo into that class.  */
 1164:
 1165:    if (GET_CODE (op) == REG && REGNO (op) >= FIRST_PSEUDO_REGISTER)
 1166:      {


 **** #20 miss at 0051fa88: cse.c:6852
   0051fa38:                       : <cse_insn+4b80> beq $s4[20],$zero[0],0051fa50 <cse_insn+4b98>
   0051fa40:                       : <cse_insn+4b88> lb $v0[2],37($s0[16])
   0051fa48:                       : <cse_insn+4b90> bne $v0[2],$zero[0],0051fa68 <cse_insn+4bb0>
   0051fa50:                       : <cse_insn+4b98> lw $a0[4],0($s0[16])
   0051fa58:                       : <cse_insn+4ba0> jal 0050e7d8 <cse_rtx_addr_varies_p>
   0051fa60:                       : <cse_insn+4ba8> beq $v0[2],$zero[0],0051fa80 <cse_insn+4bc8>
   0051fa68:                       : <cse_insn+4bb0> addu $a0[4],$zero[0],$s0[16]
   0051fa70:                       : <cse_insn+4bb8> addu $a1[5],$zero[0],$s1[17]
   0051fa78:                       : <cse_insn+4bc0> jal 0050a390 <remove_from_table>
   0051fa80:                       : <cse_insn+4bc8> addu $s0[16],$zero[0],$s2[18]
** 0051fa88:  (       6670,   0.41): <cse_insn+4bd0> bne $s0[16],$zero[0],0051fa18 <cse_insn+4b60>
   0051fa90:                       : <cse_insn+4bd8> addiu $s3[19],$s3[19],4
   0051fa98:                       : <cse_insn+4be0> addiu $s1[17],$s1[17],1
   0051faa0:                       : <cse_insn+4be8> slti $v0[2],$s1[17],31
   0051faa8:  (        745,   0.05): <cse_insn+4bf0> bne $v0[2],$zero[0],0051fa08 <cse_insn+4b50>
   cse.c:6853
   0051fab0:                       : <cse_insn+4bf8> jal 0050c730 <invalidate_for_call>
   cse.c:6861
   0051fab8:                       : <cse_insn+4c00> lw $t1[9],84($s8[30])
   0051fac0:                       : <cse_insn+4c08> sw $zero[0],76($s8[30])


---- Source Extract [cse.c] [6852]
---- (cse.c)
 6840:       sets[i].src_elt = src_eqv_elt;
 6841:
 6842:   invalidate_from_clobbers (&writes_memory, x);
```

```
 6843:
 6844:   /* Some registers are invalidated by subroutine calls.  Memory is
 6845:      invalidated by non-constant calls.  */
 6846:
 6847:   if (GET_CODE (insn) == CALL_INSN)
 6848:     {
 6849:       static struct write_data everything = {0, 1, 1, 1};
 6850:
 6851:       if (! CONST_CALL_P (insn))
*6852:  invalidate_memory (&everything);
 6853:       invalidate_for_call ();
 6854:     }
 6855:
 6856:   /* Now invalidate everything set by this instruction.
 6857:      If a SUBREG or other funny destination is being set,
 6858:      sets[i].rtl is still nonzero, so here we invalidate the reg
 6859:      a part of which is being set.  */
 6860:
 6861:   for (i = 0; i < n_sets; i++)
 6862:     if (sets[i].rtl)
 6863:       {
 6864:  register rtx dest = sets[i].inner_dest;


 **** #21 miss at 0050d6b8: cse.c:2030
   cse.c:2024
   0050d688:  (       7518,   0.46): <exp_equiv_p+78> bne $s1[17],$s0[16],0050d698 <exp_equiv_p+88>
   0050d690:  (       2716,   0.17): <exp_equiv_p+80> beq $s7[23],$zero[0],0050dea0 <exp_equiv_p+890>
   cse.c:2026
   0050d698:  (         18,   0.00): <exp_equiv_p+88> beq $s1[17],$zero[0],0050d998 <exp_equiv_p+388>
   0050d6a0:  (         10,   0.00): <exp_equiv_p+90> beq $s0[16],$zero[0],0050d998 <exp_equiv_p+388>
   cse.c:2029
   0050d6a8:                        : <exp_equiv_p+98> lhu $s2[18],0($s1[17])
   cse.c:2030
   0050d6b0:                        : <exp_equiv_p+a0> lhu $v0[2],0($s0[16])
** 0050d6b8:  (       6668,   0.41): <exp_equiv_p+a8> beq $s2[18],$v0[2],0050d940 <exp_equiv_p+330>
   cse.c:2032
   0050d6c0:  (        644,   0.04): <exp_equiv_p+b0> beq $s6[22],$zero[0],0050dd48 <exp_equiv_p+738>
   cse.c:2037
   0050d6c8:                        : <exp_equiv_p+b8> lhu $v1[3],0($s1[17])
   0050d6d0:                        : <exp_equiv_p+c0> addiu $v0[2],$v1[3],-58
   0050d6d8:                        : <exp_equiv_p+c8> sltiu $v0[2],$v0[2],2
   0050d6e0:                        : <exp_equiv_p+d0> bne $v0[2],$zero[0],0050d720 <exp_equiv_p+110>
   0050d6e8:                        : <exp_equiv_p+d8> addiu $v0[2],$v1[3],-47
   0050d6f0:                        : <exp_equiv_p+e0> sltiu $v0[2],$v0[2],2


---- Source Extract [cse.c] [2030]
---- (cse.c)
 2018:   register int i, j;
 2019:   register enum rtx_code code;
 2020:   register char *fmt;
 2021:
 2022:   /* Note: it is incorrect to assume an expression is equivalent to itself
 2023:      if VALIDATE is nonzero.  */
 2024:   if (x == y && !validate)
 2025:     return 1;
 2026:   if (x == 0 || y == 0)
 2027:     return x == y;
 2028:
 2029:   code = GET_CODE (x);
*2030:   if (code != GET_CODE (y))
 2031:     {
 2032:       if (!equal_values)
 2033:  return 0;
 2034:
 2035:       /* If X is a constant and Y is a register or vice versa, they may be
 2036:   equivalent.  We only have to validate if Y is a register.  */
 2037:       if (CONSTANT_P (x) && GET_CODE (y) == REG
 2038:    && REGNO_QTY_VALID_P (REGNO (y))
 2039:    && GET_MODE (y) == qty_mode[reg_qty[REGNO (y)]]
 2040:    && rtx_equal_p (x, qty_const[reg_qty[REGNO (y)]])
 2041:    && (! validate || reg_in_table[REGNO (y)] == reg_tick[REGNO (y)]))
 2042:  return 1;


 **** #22 miss at 0050e518: cse.c:2314
   cse.c:2311
   0050e4f0:                        : <refers_to_mem_p+b8> j 0050e770 <refers_to_mem_p+338>
   cse.c:2341
   0050e4f8:                        : <refers_to_mem_p+c0> lw $s5[21],4($s5[21])
   cse.c:2342
   0050e500:                        : <refers_to_mem_p+c8> j 0050e4e8 <refers_to_mem_p+b0>
   cse.c:2313
   0050e508:                        : <refers_to_mem_p+d0> lhu $v1[3],0($s5[21])
   cse.c:2314
   0050e510:                        : <refers_to_mem_p+d8> addiu $v0[2],$zero[0],57
** 0050e518:  (       6532,   0.40): <refers_to_mem_p+e0> bne $v1[3],$v0[2],0050e5e8 <refers_to_mem_p+1b0>
   cse.c:2320
   0050e520:                        : <refers_to_mem_p+e8> lw $v0[2],0($s5[21])
   cse.c:2316
   0050e528:                        : <refers_to_mem_p+f0> lw $a0[4],4($s5[21])
```

```
   cse.c:2320
   0050e530:                       : <refers_to_mem_p+f8> addiu $v1[3],$sp[29],32
   0050e538:                       : <refers_to_mem_p+100> sll $v0[2],$v0[2],0x10
   0050e540:                       : <refers_to_mem_p+108> sra $v0[2],$v0[2],0x18
   0050e548:                       : <refers_to_mem_p+110> sll $v0[2],$v0[2],0x2


---- Source Extract [cse.c] [2314]
---- (cse.c)
 2302:   if (GET_CODE (base) == CONST_INT)
 2303:     {
 2304:       start += INTVAL (base);
 2305:       end += INTVAL (base);
 2306:       base = const0_rtx;
 2307:     }
 2308:
 2309:  repeat:
 2310:   if (x == 0)
 2311:     return 0;
 2312:
 2313:   code = GET_CODE (x);
*2314:   if (code == MEM)
 2315:     {
 2316:       register rtx addr = XEXP (x, 0); /* Get the address.  */
 2317:       rtx mybase;
 2318:       HOST_WIDE_INT mystart, myend;
 2319:
 2320:       set_nonvarying_address_components (addr, GET_MODE_SIZE (GET_MODE (x)),
 2321:   &mybase, &mystart, &myend);
 2322:
 2323:
 2324:       /* refers_to_mem_p is never called with varying addresses.
 2325:   If the base addresses are not equal, there is no chance
 2326:   of the memory addresses conflicting.  */


 **** #23 miss at 0050bfe8: cse.c:1526
   0050bfa8:  (        828,   0.05): <invalidate+6d8> beq $t4[12],$zero[0],0050c250 <invalidate+980>
   cse.c:1525
   0050bfb0:                       : <invalidate+6e0> addu $s1[17],$zero[0],$zero[0]
   0050bfb8:                       : <invalidate+6e8> addiu $s6[22],$zero[0],52
   0050bfc0:                       : <invalidate+6f0> lui $s5[21],4099
   0050bfc8:                       : <invalidate+6f8> addiu $s5[21],$s5[21],-17788
   0050bfd0:                       : <invalidate+700> lui $s2[18],4099
   0050bfd8:                       : <invalidate+708> addiu $s2[18],$s2[18],16768
   cse.c:1526
   0050bfe0:                       : <invalidate+710> lw $a0[4],0($s2[18])
** 0050bfe8:  (       6460,   0.39): <invalidate+718> beq $a0[4],$zero[0],0050c0c0 <invalidate+7f0>
   cse.c:1530
   0050bff0:                       : <invalidate+720> lw $v1[3],0($a0[4])
   cse.c:1528
   0050bff8:                       : <invalidate+728> lw $s0[16],4($a0[4])
   cse.c:1530
   0050c000:                       : <invalidate+730> lhu $v0[2],0($v1[3])
   0050c008:  (      14410,   0.88): <invalidate+738> bne $v0[2],$s6[22],0050c0b0 <invalidate+7e0>
   0050c010:                       : <invalidate+740> lw $a1[5],4($v1[3])
   0050c018:                       : <invalidate+748> slti $v0[2],$a1[5],64


---- Source Extract [cse.c] [1526]
---- (cse.c)
 1514:    CLEAR_HARD_REG_BIT (hard_regs_in_table, regno);
 1515:
 1516:    for (i = regno + 1; i < endregno; i++)
 1517:      {
 1518:        in_table |= TEST_HARD_REG_BIT (hard_regs_in_table, i);
 1519:        CLEAR_HARD_REG_BIT (hard_regs_in_table, i);
 1520:        delete_reg_equiv (i);
 1521:        reg_tick[i]++;
 1522:      }
 1523:
 1524:    if (in_table)
 1525:      for (hash = 0; hash < NBUCKETS; hash++)
*1526:        for (p = table[hash]; p; p = next)
 1527:  {
 1528:    next = p->next_same_hash;
 1529:
 1530:    if (GET_CODE (p->exp) != REG
 1531:        || REGNO (p->exp) >= FIRST_PSEUDO_REGISTER)
 1532:      continue;
 1533:
 1534:    tregno = REGNO (p->exp);
 1535:    tendregno
 1536:      = tregno + HARD_REGNO_NREGS (tregno, GET_MODE (p->exp));
 1537:    if (tendregno > regno && tregno < endregno)
 1538:    remove_from_table (p, hash);


 **** #24 miss at 005b8438: sched.c:374
   005b83f8:  (         16,   0.00): <canon_rtx+68> bne $v0[2],$zero[0],005b8428 <canon_rtx+98>
   sched.c:373
   005b8400:                       : <canon_rtx+70> lw $v1[3],-21716($gp[28])
```

```
   005b8408:                        : <canon_rtx+78> sll $v0[2],$a0[4],0x2
   005b8410:                        : <canon_rtx+80> addu $v0[2],$v0[2],$v1[3]
   005b8418:                        : <canon_rtx+88> lw $v0[2],0($v0[2])
   005b8420:                        : <canon_rtx+90> j 005b8538 <canon_rtx+1a8>
   sched.c:374
   005b8428:                        : <canon_rtx+98> lhu $v1[3],0($s0[16])
   005b8430:                        : <canon_rtx+a0> addiu $v0[2],$zero[0],65
** 005b8438:  (       6451,   0.39): <canon_rtx+a8> bne $v1[3],$v0[2],005b8530 <canon_rtx+1a0>
   sched.c:376
   005b8440:                        : <canon_rtx+b0> lw $a0[4],4($s0[16])
   005b8448:                        : <canon_rtx+b8> jal 005b8390 <canon_rtx>
   sched.c:377
   005b8450:                        : <canon_rtx+c0> lw $a0[4],8($s0[16])
   sched.c:376
   005b8458:                        : <canon_rtx+c8> addu $s1[17],$zero[0],$v0[2]
   sched.c:377
   005b8460:                        : <canon_rtx+d0> jal 005b8390 <canon_rtx>


---- Source Extract [sched.c] [374]
---- (sched.c)
 362: static char *reg_known_equiv_p;
 363:
 364: /* Indicates number of valid entries in reg_known_value.  */
 365: static int reg_known_value_size;
 366:
 367: static rtx
 368: canon_rtx (x)
 369:      rtx x;
 370: {
 371:   if (GET_CODE (x) == REG && REGNO (x) >= FIRST_PSEUDO_REGISTER
 372:       && REGNO (x) <= reg_known_value_size)
 373:     return reg_known_value[REGNO (x)];
*374:   else if (GET_CODE (x) == PLUS)
 375:     {
 376:       rtx x0 = canon_rtx (XEXP (x, 0));
 377:       rtx x1 = canon_rtx (XEXP (x, 1));
 378:
 379:       if (x0 != XEXP (x, 0) || x1 != XEXP (x, 1))
 380:  {
 381:    /* We can tolerate LO_SUMs being offset here; these
 382:       rtl are used for nothing other than comparisons.  */
 383:    if (GET_CODE (x0) == CONST_INT)
 384:      return plus_constant_for_output (x1, INTVAL (x0));
 385:    else if (GET_CODE (x1) == CONST_INT)
 386:      return plus_constant_for_output (x0, INTVAL (x1));


 **** #25 miss at 0050c9b8: cse.c:1711
   0050c988:                        : <invalidate_for_call+258> addiu $s2[18],$s2[18],16768
   cse.c:1707
   0050c990:                        : <invalidate_for_call+260> lw $a0[4],0($s2[18])
   0050c998:  (       2817,   0.17): <invalidate_for_call+268> beq $a0[4],$zero[0],0050cad0 <invalidate_for_call+3a0>
   cse.c:1711
   0050c9a0:                        : <invalidate_for_call+270> lw $a1[5],0($a0[4])
   cse.c:1709
   0050c9a8:                        : <invalidate_for_call+278> lw $s1[17],4($a0[4])
   cse.c:1711
   0050c9b0:                        : <invalidate_for_call+280> lhu $v0[2],0($a1[5])
** 0050c9b8:  (       5972,   0.36): <invalidate_for_call+288> bne $v0[2],$s5[21],0050cac0 <invalidate_for_call+390>
   0050c9c0:                        : <invalidate_for_call+290> lw $v1[3],4($a1[5])
   0050c9c8:                        : <invalidate_for_call+298> slti $v0[2],$v1[3],64
   0050c9d0:  (        911,   0.06): <invalidate_for_call+2a0> beq $v0[2],$zero[0],0050cac0 <invalidate_for_call+390>
   cse.c:1716
   0050c9d8:                        : <invalidate_for_call+2a8> lw $v0[2],0($a1[5])
   0050c9e0:                        : <invalidate_for_call+2b0> sll $v0[2],$v0[2],0x10
   0050c9e8:                        : <invalidate_for_call+2b8> sra $v0[2],$v0[2],0x18
   0050c9f0:                        : <invalidate_for_call+2c0> sll $v0[2],$v0[2],0x2
   0050c9f8:                        : <invalidate_for_call+2c8> addu $v0[2],$v0[2],$s4[20]


---- Source Extract [cse.c] [1711]
---- (cse.c)
 1699:       }
 1700:
 1701:   /* In the case where we have no call-clobbered hard registers in the
 1702:      table, we are done.  Otherwise, scan the table and remove any
 1703:      entry that overlaps a call-clobbered register.  */
 1704:
 1705:   if (in_table)
 1706:     for (hash = 0; hash < NBUCKETS; hash++)
 1707:       for (p = table[hash]; p; p = next)
 1708:  {
 1709:    next = p->next_same_hash;
 1710:
*1711:    if (GET_CODE (p->exp) != REG
 1712:        || REGNO (p->exp) >= FIRST_PSEUDO_REGISTER)
 1713:      continue;
 1714:
 1715:    regno = REGNO (p->exp);
 1716:    endregno = regno + HARD_REGNO_NREGS (regno, GET_MODE (p->exp));
 1717:
```

```
 1718:    for (i = regno; i < endregno; i++)
 1719:      if (TEST_HARD_REG_BIT (regs_invalidated_by_call, i))
 1720:        {
 1721:  remove_from_table (p, hash);
 1722:  break;
 1723:        }


 **** #26 miss at 0056d688: regclass.c:927
   0056d638:                     : <record_reg_classes+f68> sw $t6[14],460($sp[29])
   0056d640:                     : <record_reg_classes+f70> lw $t6[14],476($sp[29])
   0056d648:                     : <record_reg_classes+f78> addiu $s8[30],$s8[30],4
   0056d650:                     : <record_reg_classes+f80> addiu $t7[15],$t7[15],20
   0056d658:                     : <record_reg_classes+f88> sw $t7[15],468($sp[29])
   0056d660:                     : <record_reg_classes+f90> lw $t7[15],308($sp[29])
   0056d668:                     : <record_reg_classes+f98> addiu $s6[22],$s6[22],1
   0056d670:                     : <record_reg_classes+fa0> addiu $t6[14],$t6[14],4
   0056d678:                     : <record_reg_classes+fa8> slt $v0[2],$s6[22],$t7[15]
   0056d680:                     : <record_reg_classes+fb0> sw $t6[14],476($sp[29])
** 0056d688:  (       5910,   0.36): <record_reg_classes+fb8> bne $v0[2],$zero[0],0056c848 <record_reg_classes+178>
   regclass.c:1223
   0056d690:                     : <record_reg_classes+fc0> lw $t6[14],340($sp[29])
   0056d698:  (       3092,   0.19): <record_reg_classes+fc8> bne $t6[14],$zero[0],0056d850 <record_reg_classes+1180>
   regclass.c:1229
   0056d6a0:                     : <record_reg_classes+fd0> lw $t7[15],308($sp[29])
   0056d6a8:                     : <record_reg_classes+fd8> addu $s6[22],$zero[0],$zero[0]
   0056d6b0:  (         30,   0.00): <record_reg_classes+fe0> blez $t7[15],0056d850 <record_reg_classes+1180>
   0056d6b8:                     : <record_reg_classes+fe8> addiu $t5[13],$zero[0],2
   0056d6c0:                     : <record_reg_classes+ff0> lw $t4[12],388($sp[29])


---- Source Extract [regclass.c] [927]
---- (regclass.c)
 915:
 916:   /* Process each alternative, each time minimizing an operand's cost with
 917:      the cost for each operand in that alternative.  */
 918:
 919:   for (alt = 0; alt < n_alts; alt++)
 920:     {
 921:       struct costs this_op_costs[MAX_RECOG_OPERANDS];
 922:       int alt_fail = 0;
 923:       int alt_cost = 0;
 924:       enum reg_class classes[MAX_RECOG_OPERANDS];
 925:       int class;
 926:
*927:       for (i = 0; i < n_ops; i++)
 928:  {
 929:    char *p = constraints[i];
 930:    rtx op = ops[i];
 931:    enum machine_mode mode = modes[i];
 932:    int allows_mem = 0;
 933:    int win = 0;
 934:    char c;
 935:
 936:    /* If this operand has no constraints at all, we can conclude
 937:       nothing about it since anything is valid.  */
 938:
 939:    if (*p == 0)


 **** #27 miss at 0050ee20: cse.c:2449
   0050edd8:                     : <canon_reg+3e0> lw $v0[2],4($s2[18])
   0050ede0:                     : <canon_reg+3e8> lw $v0[2],0($v0[2])
   0050ede8:                     : <canon_reg+3f0> addiu $s1[17],$s1[17],1
   0050edf0:                     : <canon_reg+3f8> sltu $v0[2],$s1[17],$v0[2]
   0050edf8:                     : <canon_reg+400> bne $v0[2],$zero[0],0050ed90 <canon_reg+398>
   cse.c:2449
   0050ee00:                     : <canon_reg+408> addiu $s2[18],$s2[18],-4
   0050ee08:                     : <canon_reg+410> addiu $s7[23],$s7[23],-4
   0050ee10:                     : <canon_reg+418> addiu $s6[22],$s6[22],-1
   0050ee18:                     : <canon_reg+420> addiu $s4[20],$s4[20],-1
** 0050ee20:  (       5821,   0.35): <canon_reg+428> bgez $s4[20],0050ec48 <canon_reg+250>
   cse.c:2473
   0050ee28:                     : <canon_reg+430> addu $v0[2],$zero[0],$s5[21]
   cse.c:2474
   0050ee30:                     : <canon_reg+438> lw $ra[31],60($sp[29])
   0050ee38:                     : <canon_reg+440> lw $s8[30],56($sp[29])
   0050ee40:                     : <canon_reg+448> lw $s7[23],52($sp[29])
   0050ee48:                     : <canon_reg+450> lw $s6[22],48($sp[29])
   0050ee50:                     : <canon_reg+458> lw $s5[21],44($sp[29])
   0050ee58:                     : <canon_reg+460> lw $s4[20],40($sp[29])


---- Source Extract [cse.c] [2449]
---- (cse.c)
 2437:  if (REGNO (x) < FIRST_PSEUDO_REGISTER
 2438:      || ! REGNO_QTY_VALID_P (REGNO (x)))
 2439:    return x;
 2440:
 2441:  first = qty_first_reg[reg_qty[REGNO (x)]];
 2442:  return (first >= FIRST_PSEUDO_REGISTER ? regno_reg_rtx[first]
 2443:  : REGNO_REG_CLASS (first) == NO_REGS ? x
```

```
 2444:  : gen_rtx (REG, qty_mode[reg_qty[REGNO (x)]], first));
 2445:        }
 2446:      }
 2447:
 2448:    fmt = GET_RTX_FORMAT (code);
*2449:    for (i = GET_RTX_LENGTH (code) - 1; i >= 0; i--)
 2450:      {
 2451:        register int j;
 2452:
 2453:        if (fmt[i] == 'e')
 2454:  {
 2455:    rtx new = canon_reg (XEXP (x, i), insn);
 2456:
 2457:    /* If replacing pseudo with hard reg or vice versa, ensure the
 2458:       insn remains valid.  Likewise if the insn has MATCH_DUPs.  */
 2459:    if (insn != 0 && new != 0
 2460:        && GET_CODE (new) == REG && GET_CODE (XEXP (x, i)) == REG
 2461:        && (((REGNO (new) < FIRST_PSEUDO_REGISTER)


 **** #28 miss at 005b83c8: sched.c:371
   canon_rtx():
   sched.c:369
   005b8390:                        : <canon_rtx> addiu $sp[29],$sp[29],-32
   005b8398:                        : <canon_rtx+8> sw $s0[16],16($sp[29])
   005b83a0:                        : <canon_rtx+10> addu $s0[16],$zero[0],$a0[4]
   005b83a8:                        : <canon_rtx+18> sw $ra[31],24($sp[29])
   005b83b0:                        : <canon_rtx+20> sw $s1[17],20($sp[29])
   sched.c:371
   005b83b8:                        : <canon_rtx+28> lhu $v1[3],0($s0[16])
   005b83c0:                        : <canon_rtx+30> addiu $v0[2],$zero[0],52
** 005b83c8:  (       5741,   0.35): <canon_rtx+38> bne $v1[3],$v0[2],005b8428 <canon_rtx+98>
   005b83d0:                        : <canon_rtx+40> lw $a0[4],4($s0[16])
   005b83d8:                        : <canon_rtx+48> slti $v0[2],$a0[4],64
   005b83e0:  (       4383,   0.27): <canon_rtx+50> bne $v0[2],$zero[0],005b8428 <canon_rtx+98>
   005b83e8:                        : <canon_rtx+58> lw $v0[2],-21708($gp[28])
   005b83f0:                        : <canon_rtx+60> slt $v0[2],$v0[2],$a0[4]
   005b83f8:  (         16,   0.00): <canon_rtx+68> bne $v0[2],$zero[0],005b8428 <canon_rtx+98>
   sched.c:373
   005b8400:                        : <canon_rtx+70> lw $v1[3],-21716($gp[28])
   005b8408:                        : <canon_rtx+78> sll $v0[2],$a0[4],0x2


---- Source Extract [sched.c] [371]
---- (sched.c)
 359:    notes.  One could argue that the REG_EQUIV notes are wrong, but solving
 360:    the problem in the scheduler will likely give better code, so we do it
 361:    here.  */
 362: static char *reg_known_equiv_p;
 363:
 364: /* Indicates number of valid entries in reg_known_value.  */
 365: static int reg_known_value_size;
 366:
 367: static rtx
 368: canon_rtx (x)
 369:      rtx x;
 370: {
*371:   if (GET_CODE (x) == REG && REGNO (x) >= FIRST_PSEUDO_REGISTER
 372:       && REGNO (x) <= reg_known_value_size)
 373:     return reg_known_value[REGNO (x)];
 374:   else if (GET_CODE (x) == PLUS)
 375:     {
 376:       rtx x0 = canon_rtx (XEXP (x, 0));
 377:       rtx x1 = canon_rtx (XEXP (x, 1));
 378:
 379:       if (x0 != XEXP (x, 0) || x1 != XEXP (x, 1))
 380:  {
 381:    /* We can tolerate LO_SUMs being offset here; these
 382:       rtl are used for nothing other than comparisons.  */
 383:    if (GET_CODE (x0) == CONST_INT)


 **** #29 miss at 00600f38: ../sysdeps/generic/memset.c:62
   ../sysdeps/generic/memset.c:58
   00600f10:                        : <memset+c0> sw $a3[7],-4($v0[2])
   ../sysdeps/generic/memset.c:59
   00600f18:                        : <memset+c8> sw $a3[7],0($v0[2])
   ../sysdeps/generic/memset.c:60
   00600f20:                        : <memset+d0> addiu $v0[2],$v0[2],32
   00600f28:                        : <memset+d8> addiu $t0[8],$t0[8],32
   ../sysdeps/generic/memset.c:61
   00600f30:                        : <memset+e0> addiu $v1[3],$v1[3],-1
   ../sysdeps/generic/memset.c:62
** 00600f38:  (       5740,   0.35): <memset+e8> bne $v1[3],$zero[0],00600ee0 <memset+90>
   ../sysdeps/generic/memset.c:63
   00600f40:                        : <memset+f0> andi $a2[6],$a2[6],31
   ../sysdeps/generic/memset.c:66
   00600f48:                        : <memset+f8> srl $v1[3],$a2[6],0x2
   ../sysdeps/generic/memset.c:67
   00600f50:  (        897,   0.05): <memset+100> beq $v1[3],$zero[0],00600f78 <memset+128>
   ../sysdeps/generic/memset.c:69
   00600f58:                        : <memset+108> sw $a3[7],0($t0[8])
```

```
   ../sysdeps/generic/memset.c:70


---- Source Extract [../sysdeps/generic/memset.c] [62]
---- (/pong/usr3/s/smithz/research/tools/src/glibc-1.09/sysdeps/generic/memset.c)
 50:       while (xlen > 0)
 51:  {
 52:    ((op_t *) dstp)[0] = cccc;
 53:    ((op_t *) dstp)[1] = cccc;
 54:    ((op_t *) dstp)[2] = cccc;
 55:    ((op_t *) dstp)[3] = cccc;
 56:    ((op_t *) dstp)[4] = cccc;
 57:    ((op_t *) dstp)[5] = cccc;
 58:    ((op_t *) dstp)[6] = cccc;
 59:    ((op_t *) dstp)[7] = cccc;
 60:    dstp += 8 * OPSIZ;
 61:    xlen -= 1;
*62:  }
 63:       len %= OPSIZ * 8;
 64:
 65:       /* Write 1 `op_t' per iteration until less than OPSIZ bytes remain.  */
 66:       xlen = len / OPSIZ;
 67:       while (xlen > 0)
 68:  {
 69:    ((op_t *) dstp)[0] = cccc;
 70:    dstp += OPSIZ;
 71:    xlen -= 1;
 72:  }
 73:       len %= OPSIZ;
 74:     }


 **** #30 miss at 0050cac8: cse.c:1707
   0050ca90:                        : <invalidate_for_call+360> addu $a1[5],$zero[0],$s0[16]
   0050ca98:                        : <invalidate_for_call+368> jal 0050a390 <remove_from_table>
   cse.c:1722
   0050caa0:                        : <invalidate_for_call+370> j 0050cac0 <invalidate_for_call+390>
   cse.c:1718
   0050caa8:                        : <invalidate_for_call+378> addiu $a1[5],$a1[5],1
   0050cab0:                        : <invalidate_for_call+380> slt $v0[2],$a1[5],$a2[6]
   0050cab8:  (          4,   0.00): <invalidate_for_call+388> bne $v0[2],$zero[0],0050ca50 <invalidate_for_call+320>
   cse.c:1707
   0050cac0:                        : <invalidate_for_call+390> addu $a0[4],$zero[0],$s1[17]
** 0050cac8:  (       5659,   0.34): <invalidate_for_call+398> bne $a0[4],$zero[0],0050c9a0 <invalidate_for_call+270>
   cse.c:1706
   0050cad0:                        : <invalidate_for_call+3a0> addiu $s2[18],$s2[18],4
   0050cad8:                        : <invalidate_for_call+3a8> addiu $s0[16],$s0[16],1
   0050cae0:                        : <invalidate_for_call+3b0> slti $v0[2],$s0[16],31
   0050cae8:  (        483,   0.03): <invalidate_for_call+3b8> bne $v0[2],$zero[0],0050c990 <invalidate_for_call+260>
   cse.c:1725
   0050caf0:                        : <invalidate_for_call+3c0> lw $ra[31],40($sp[29])
   0050caf8:                        : <invalidate_for_call+3c8> lw $s5[21],36($sp[29])
   0050cb00:                        : <invalidate_for_call+3d0> lw $s4[20],32($sp[29])


---- Source Extract [cse.c] [1707]
---- (cse.c)
 1695:  if (reg_tick[regno] >= 0)
 1696:    reg_tick[regno]++;
 1697:
 1698:  in_table |= TEST_HARD_REG_BIT (hard_regs_in_table, regno);
 1699:       }
 1700:
 1701:   /* In the case where we have no call-clobbered hard registers in the
 1702:      table, we are done.  Otherwise, scan the table and remove any
 1703:      entry that overlaps a call-clobbered register.  */
 1704:
 1705:   if (in_table)
 1706:     for (hash = 0; hash < NBUCKETS; hash++)
*1707:       for (p = table[hash]; p; p = next)
 1708:  {
 1709:    next = p->next_same_hash;
 1710:
 1711:    if (GET_CODE (p->exp) != REG
 1712:        || REGNO (p->exp) >= FIRST_PSEUDO_REGISTER)
 1713:      continue;
 1714:
 1715:    regno = REGNO (p->exp);
 1716:    endregno = regno + HARD_REGNO_NREGS (regno, GET_MODE (p->exp));
 1717:
 1718:    for (i = regno; i < endregno; i++)
 1719:      if (TEST_HARD_REG_BIT (regs_invalidated_by_call, i))
```

Appendix D

li-train Top 25 Misses

```
 **** #1 miss at 00405a18: xldmem.c:284
   004059e8:  (      25720,   1.78): <mark+28> beq $a0[4],$zero[0],00405ba8 <mark+1e8>
   xldmem.c:274
   004059f0:                       : <mark+30> addu $s1[17],$zero[0],$zero[0]
   xldmem.c:275
   004059f8:                       : <mark+38> addu $s0[16],$zero[0],$a0[4]
   xldmem.c:278
   00405a00:                       : <mark+40> addiu $s2[18],$zero[0],-3
   xldmem.c:284
   00405a08:                       : <mark+48> lbu $v1[3],1($s0[16])
   00405a10:                       : <mark+50> andi $v0[2],$v1[3],1
** 00405a18:  (     292939,  20.28): <mark+58> bne $v0[2],$zero[0],00405ae0 <mark+120>
   xldmem.c:291
   00405a20:                       : <mark+60> ori $v0[2],$v1[3],1
   00405a28:                       : <mark+68> sb $v0[2],1($s0[16])
   xldmem.c:294
   00405a30:                       : <mark+70> addu $a0[4],$zero[0],$s0[16]
   00405a38:                       : <mark+78> jal 004060b8 <livecar>
   00405a40:  (      96966,   6.71): <mark+80> beq $v0[2],$zero[0],00405a88 <mark+c8>
   xldmem.c:295
   00405a48:                       : <mark+88> lbu $v0[2],1($s0[16])
```

```
---- Source Extract [xldmem.c] [284]
---- (xldmem.c)
 272:
 273:     /* initialize */
 274:     prev = NIL;
 275:     this = ptr;
 276:
 277:     /* mark this list */
 278:     while (TRUE) {
 279:
 280:  /* descend as far as we can */
 281:  while (TRUE) {
 282:
 283:      /* check for this node being marked */
*284:      if (this->n_flags & MARK)
 285:  break;
 286:
 287:      /* mark it and its descendants */
 288:      else {
 289:
 290:  /* mark the node */
 291:  this->n_flags |= MARK;
 292:
 293:  /* follow the left sublist if there is one */
 294:  if (livecar(this)) {
 295:      this->n_flags |= LEFT;
 296:      tmp = prev;
```

```
 **** #2 miss at 00405b10: xldmem.c:324
   00405ad8:                       : <mark+118> j 00405a08 <mark+48>
   xldmem.c:319
   00405ae0:  (      45646,   3.16): <mark+120> beq $s1[17],$zero[0],00405ba8 <mark+1e8>
   xldmem.c:323
   00405ae8:                       : <mark+128> lbu $v0[2],1($s1[17])
   00405af0:                       : <mark+130> andi $v0[2],$v0[2],2
   00405af8:  (     186921,  12.94): <mark+138> beq $v0[2],$zero[0],00405b80 <mark+1c0>
   xldmem.c:324
   00405b00:                       : <mark+140> addu $a0[4],$zero[0],$s1[17]
   00405b08:                       : <mark+148> jal 00406228 <livecdr>
** 00405b10:  (     280444,  19.41): <mark+150> beq $v0[2],$zero[0],00405b58 <mark+198>
```

```
   xldmem.c:326
   00405b18:                       : <mark+158> lw $v1[3],4($s1[17])
   xldmem.c:325
   00405b20:                       : <mark+160> lbu $v0[2],1($s1[17])
   xldmem.c:327
   00405b28:                       : <mark+168> sw $s0[16],4($s1[17])
   xldmem.c:328
   00405b30:                       : <mark+170> lw $s0[16],8($s1[17])
   xldmem.c:325


---- Source Extract [xldmem.c] [324]
---- (xldmem.c)
 312:      }
 313:  }
 314:
 315:  /* backup to a point where we can continue descending */
 316:  while (TRUE) {
 317:
 318:      /* check for termination condition */
 319:      if (prev == NIL)
 320:  return;
 321:
 322:      /* check for coming from the left side */
 323:      if (prev->n_flags & LEFT)
*324:  if (livecdr(prev)) {
 325:      prev->n_flags &= ~LEFT;
 326:      tmp = car(prev);
 327:      rplaca(prev,this);
 328:      this = cdr(prev);
 329:      rplacd(prev,tmp);
 330:      break;
 331:  }
 332:  else {
 333:      tmp = prev;
 334:      prev = car(tmp);
 335:      rplaca(tmp,this);
 336:      this = tmp;


 **** #3 miss at 00405af8: xldmem.c:323
   00405ac8:                       : <mark+108> lw $s0[16],8($s0[16])
   xldmem.c:308
   00405ad0:                       : <mark+110> sw $v1[3],8($s1[17])
   xldmem.c:313
   00405ad8:                       : <mark+118> j 00405a08 <mark+48>
   xldmem.c:319
   00405ae0:  (      45646,   3.16): <mark+120> beq $s1[17],$zero[0],00405ba8 <mark+1e8>
   xldmem.c:323
   00405ae8:                       : <mark+128> lbu $v0[2],1($s1[17])
   00405af0:                       : <mark+130> andi $v0[2],$v0[2],2
** 00405af8:  (     186921,  12.94): <mark+138> beq $v0[2],$zero[0],00405b80 <mark+1c0>
   xldmem.c:324
   00405b00:                       : <mark+140> addu $a0[4],$zero[0],$s1[17]
   00405b08:                       : <mark+148> jal 00406228 <livecdr>
   00405b10:  (     280444,  19.41): <mark+150> beq $v0[2],$zero[0],00405b58 <mark+198>
   xldmem.c:326
   00405b18:                       : <mark+158> lw $v1[3],4($s1[17])
   xldmem.c:325
   00405b20:                       : <mark+160> lbu $v0[2],1($s1[17])
   xldmem.c:327


---- Source Extract [xldmem.c] [323]
---- (xldmem.c)
 311:      break;
 312:      }
 313:  }
 314:
 315:  /* backup to a point where we can continue descending */
 316:  while (TRUE) {
 317:
 318:      /* check for termination condition */
 319:      if (prev == NIL)
 320:  return;
 321:
 322:      /* check for coming from the left side */
*323:      if (prev->n_flags & LEFT)
 324:  if (livecdr(prev)) {
 325:      prev->n_flags &= ~LEFT;
 326:      tmp = car(prev);
 327:      rplaca(prev,this);
 328:      this = cdr(prev);
 329:      rplacd(prev,tmp);
 330:      break;
 331:  }
 332:  else {
 333:      tmp = prev;
 334:      prev = car(tmp);
 335:      rplaca(tmp,this);
```

```
 **** #4 miss at 00405d28: xldmem.c:373
   00405cf0:                       : <sweep+68> lw $s3[19],0($s4[20])
   xldmem.c:371
   00405cf8:                       : <sweep+70> addiu $s2[18],$s4[20],8
   xldmem.c:372
   00405d00:                       : <sweep+78> addiu $s3[19],$s3[19],-1
   00405d08:  (          1,   0.00): <sweep+80> beq $s3[19],$s5[21],00405ec0 <sweep+238>
   00405d10:                       : <sweep+88> addiu $s1[17],$s4[20],9
   xldmem.c:373
   00405d18:                       : <sweep+90> lbu $v1[3],0($s1[17])
   00405d20:                       : <sweep+98> andi $v0[2],$v1[3],1
** 00405d28:  (     165423,  11.45): <sweep+a0> bne $v0[2],$zero[0],00405e90 <sweep+208>
   xldmem.c:374
   00405d30:                       : <sweep+a8> lb $v1[3],0($s2[18])
   00405d38:                       : <sweep+b0> addiu $v0[2],$zero[0],8
   00405d40:  (          4,   0.00): <sweep+b8> beq $v1[3],$v0[2],00405de0 <sweep+158>
   00405d48:                       : <sweep+c0> slti $v0[2],$v1[3],9
   00405d50:                       : <sweep+c8> beq $v0[2],$zero[0],00405d70 <sweep+e8>
   00405d58:                       : <sweep+d0> addiu $v0[2],$zero[0],6
   00405d60:                       : <sweep+d8> beq $v1[3],$v0[2],00405d88 <sweep+100>
   00405d68:                       : <sweep+e0> j 00405e40 <sweep+1b8>


---- Source Extract [xldmem.c] [373]
---- (xldmem.c)
 361:     struct segment *seg;
 362:     NODE *p;
 363:     int n;
 364:
 365:     /* empty the free list */
 366:     fnodes = NIL;
 367:     nfree = 0;
 368:
 369:     /* add all unmarked nodes */
 370:     for (seg = segs; seg != NULL; seg = seg->sg_next) {
 371:  p = &seg->sg_nodes[0];
 372:  for (n = seg->sg_size; n--; p++)
*373:      if (!(p->n_flags & MARK)) {
 374:  switch (ntype(p)) {
 375:  case STR:
 376:  if (p->n_strtype == DYNAMIC && p->n_str != NULL) {
 377:      total -= (long) (strlen(p->n_str)+1);
 378:      free(p->n_str);
 379:  }
 380:  break;
 381:  case FPTR:
 382:  if (p->n_fp)
 383:      fclose(p->n_fp);
 384:  break;
 385:  case VECT:


 **** #5 miss at 00405a40: xldmem.c:294
   xldmem.c:284
   00405a08:                       : <mark+48> lbu $v1[3],1($s0[16])
   00405a10:                       : <mark+50> andi $v0[2],$v1[3],1
   00405a18:  (     292939,  20.28): <mark+58> bne $v0[2],$zero[0],00405ae0 <mark+120>
   xldmem.c:291
   00405a20:                       : <mark+60> ori $v0[2],$v1[3],1
   00405a28:                       : <mark+68> sb $v0[2],1($s0[16])
   xldmem.c:294
   00405a30:                       : <mark+70> addu $a0[4],$zero[0],$s0[16]
   00405a38:                       : <mark+78> jal 004060b8 <livecar>
** 00405a40:  (      96966,   6.71): <mark+80> beq $v0[2],$zero[0],00405a88 <mark+c8>
   xldmem.c:295
   00405a48:                       : <mark+88> lbu $v0[2],1($s0[16])
   xldmem.c:296
   00405a50:                       : <mark+90> addu $v1[3],$zero[0],$s1[17]
   xldmem.c:297
   00405a58:                       : <mark+98> addu $s1[17],$zero[0],$s0[16]
   xldmem.c:295
   00405a60:                       : <mark+a0> ori $v0[2],$v0[2],2
   00405a68:                       : <mark+a8> sb $v0[2],1($s0[16])


---- Source Extract [xldmem.c] [294]
---- (xldmem.c)
 282:
 283:      /* check for this node being marked */
 284:      if (this->n_flags & MARK)
 285:  break;
 286:
 287:      /* mark it and its descendants */
 288:      else {
 289:
 290:  /* mark the node */
 291:  this->n_flags |= MARK;
 292:
 293:  /* follow the left sublist if there is one */
*294:  if (livecar(this)) {
 295:      this->n_flags |= LEFT;
 296:      tmp = prev;
```

```
 297:      prev = this;
 298:      this = car(prev);
 299:      rplaca(prev,tmp);
 300:  }
 301:
 302:  /* otherwise, follow the right sublist if there is one */
 303:  else if (livecdr(this)) {
 304:      this->n_flags &= ~LEFT;
 305:      tmp = prev;
 306:      prev = this;


 **** #6 miss at 004153f0: xlsym.c:97
   004153b8:                       : <xlxgetvalue+68> j 00415420 <xlxgetvalue+d0>
   xlsym.c:95
   004153c0:                       : <xlxgetvalue+70> lw $a1[5],-31408($gp[28])
   004153c8:  (          6,   0.00): <xlxgetvalue+78> beq $a1[5],$zero[0],00415418 <xlxgetvalue+c8>
   xlsym.c:96
   004153d0:                       : <xlxgetvalue+80> lw $v1[3],4($a1[5])
   004153d8:  (        107,   0.01): <xlxgetvalue+88> beq $v1[3],$zero[0],00415408 <xlxgetvalue+b8>
   xlsym.c:97
   004153e0:                       : <xlxgetvalue+90> lw $a0[4],4($v1[3])
   004153e8:                       : <xlxgetvalue+98> lw $v0[2],4($a0[4])
** 004153f0:  (      73301,   5.07): <xlxgetvalue+a0> beq $s0[16],$v0[2],004153b0 <xlxgetvalue+60>
   xlsym.c:96
   004153f8:                       : <xlxgetvalue+a8> lw $v1[3],8($v1[3])
   00415400:  (        172,   0.01): <xlxgetvalue+b0> bne $v1[3],$zero[0],004153e0 <xlxgetvalue+90>
   xlsym.c:95
   00415408:                       : <xlxgetvalue+b8> lw $a1[5],8($a1[5])
   00415410:  (          7,   0.00): <xlxgetvalue+c0> bne $a1[5],$zero[0],004153d0 <xlxgetvalue+80>
   xlsym.c:101
   00415418:                       : <xlxgetvalue+c8> lw $v0[2],8($s0[16])
   xlsym.c:102


---- Source Extract [xlsym.c] [97]
---- (xlsym.c)
 85: NODE *xlxgetvalue(NODE *sym)
 86: {
 87:     register NODE *fp,*ep;
 88:     NODE *val;
 89:
 90:     /* check for this being an instance variable */
 91:     if (getvalue(self) && xlobgetvalue(sym,&val))
 92:  return (val);
 93:
 94:     /* check the environment list */
 95:     for (fp = xlenv; fp; fp = cdr(fp))
 96:  for (ep = car(fp); ep; ep = cdr(ep))
*97:      if (sym == car(car(ep)))
 98:  return (cdr(car(ep)));
 99:
 100:     /* return the global value */
 101:     return (getvalue(sym));
 102: }
 103:
 104: /* xlygetvalue - get the value of a symbol (no instance variables) */
 105: NODE *xlygetvalue(NODE *sym)
 106: {
 107:     register NODE *fp,*ep;
 108:
 109:     /* check the environment list */


 **** #7 miss at 00407328: xleval.c:195
   004072d8:                       : <xlevlist+190> addiu $a0[4],$v0[2],1
   004072e0:                       : <xlevlist+198> slti $v0[2],$a0[4],500
   004072e8:                       : <xlevlist+1a0> sw $a0[4],-31392($gp[28])
   004072f0:                       : <xlevlist+1a8> beq $v0[2],$zero[0],00407318 <xlevlist+1d0>
   004072f8:                       : <xlevlist+1b0> lw $v1[3],-31388($gp[28])
   00407300:                       : <xlevlist+1b8> sll $v0[2],$a0[4],0x2
   00407308:                       : <xlevlist+1c0> addu $v0[2],$v0[2],$v1[3]
   00407310:                       : <xlevlist+1c8> sw $s0[16],0($v0[2])
   00407318:  (          3,   0.00): <xlevlist+1d0> beq $s0[16],$zero[0],00407370 <xlevlist+228>
   00407320:                       : <xlevlist+1d8> lb $v1[3],0($s0[16])
** 00407328:  (      47745,   3.31): <xlevlist+1e0> bne $v1[3],$s4[20],00407348 <xlevlist+200>
   00407330:                       : <xlevlist+1e8> addu $a0[4],$zero[0],$s0[16]
   00407338:                       : <xlevlist+1f0> jal 00406940 <evform>
   00407340:                       : <xlevlist+1f8> j 00407368 <xlevlist+220>
   00407348:                       : <xlevlist+200> addiu $v0[2],$zero[0],4
   00407350:                       : <xlevlist+208> bne $v1[3],$v0[2],00407370 <xlevlist+228>
   00407358:                       : <xlevlist+210> addu $a0[4],$zero[0],$s0[16]
   00407360:                       : <xlevlist+218> jal 004152d8 <xlgetvalue>
   00407368:                       : <xlevlist+220> addu $s0[16],$zero[0],$v0[2]
   00407370:                       : <xlevlist+228> lw $v0[2],-31392($gp[28])


---- Source Extract [xleval.c] [195]
---- (xleval.c)
 183:     for (val = NIL; src; src = cdr(src)) {
 184:
 185:  /* check this entry */
```

```
 186:  if (!consp(src))
 187:      xlfail("bad argument list");
 188:
 189:  /* allocate a new list entry */
 190:  new = consa(NIL);
 191:  if (val)
 192:      rplacd(last,new);
 193:  else
 194:      val = dst = new;
*195:  rplaca(new,xleval(car(src)));
 196:  last = new;
 197:     }
 198:
 199:     /* restore the previous stack frame */
 200:     xlstack = oldstk;
 201:
 202:     /* return the new list */
 203:     return (val);
 204: }
 205:
 206: /* xlunbound - signal an unbound variable error */
 207: void xlunbound(NODE *sym)


 **** #8 miss at 00405ae0: xldmem.c:319
   xldmem.c:304
   00405ab8:                        : <mark+f8> and $v0[2],$v0[2],$s2[18]
   00405ac0:                        : <mark+100> sb $v0[2],1($s0[16])
   xldmem.c:307
   00405ac8:                        : <mark+108> lw $s0[16],8($s0[16])
   xldmem.c:308
   00405ad0:                        : <mark+110> sw $v1[3],8($s1[17])
   xldmem.c:313
   00405ad8:                        : <mark+118> j 00405a08 <mark+48>
   xldmem.c:319
** 00405ae0:  (      45646,   3.16): <mark+120> beq $s1[17],$zero[0],00405ba8 <mark+1e8>
   xldmem.c:323
   00405ae8:                        : <mark+128> lbu $v0[2],1($s1[17])
   00405af0:                        : <mark+130> andi $v0[2],$v0[2],2
   00405af8:  (     186921,  12.94): <mark+138> beq $v0[2],$zero[0],00405b80 <mark+1c0>
   xldmem.c:324
   00405b00:                        : <mark+140> addu $a0[4],$zero[0],$s1[17]
   00405b08:                        : <mark+148> jal 00406228 <livecdr>
   00405b10:  (     280444,  19.41): <mark+150> beq $v0[2],$zero[0],00405b58 <mark+198>
   xldmem.c:326


---- Source Extract [xldmem.c] [319]
---- (xldmem.c)
 307:      this = cdr(prev);
 308:      rplacd(prev,tmp);
 309:  }
 310:  else
 311:      break;
 312:      }
 313:  }
 314:
 315:  /* backup to a point where we can continue descending */
 316:  while (TRUE) {
 317:
 318:      /* check for termination condition */
*319:      if (prev == NIL)
 320:  return;
 321:
 322:      /* check for coming from the left side */
 323:      if (prev->n_flags & LEFT)
 324:  if (livecdr(prev)) {
 325:      prev->n_flags &= ~LEFT;
 326:      tmp = car(prev);
 327:      rplaca(prev,this);
 328:      this = cdr(prev);
 329:      rplacd(prev,tmp);
 330:      break;
 331:  }


 **** #9 miss at 00406b78: xleval.c:104
   00406b30:                        : <evform+1f0> lw $a0[4],16($sp[29])
   00406b38:                        : <evform+1f8> lbu $v1[3],0($a0[4])
   00406b40:                        : <evform+200> addiu $v0[2],$v1[3],-1
   00406b48:                        : <evform+208> sltiu $v0[2],$v0[2],2
   00406b50:  (      24115,   1.67): <evform+210> beq $v0[2],$zero[0],00406bc8 <evform+288>
   xleval.c:104
   00406b58:  (          2,   0.00): <evform+218> beq $a0[4],$zero[0],00406b98 <evform+258>
   00406b60:                        : <evform+220> sll $v0[2],$v1[3],0x18
   00406b68:                        : <evform+228> sra $v0[2],$v0[2],0x18
   00406b70:                        : <evform+230> addiu $v1[3],$zero[0],1
** 00406b78:  (      35901,   2.49): <evform+238> bne $v0[2],$v1[3],00406b98 <evform+258>
   xleval.c:105
   00406b80:                        : <evform+240> lw $a0[4],20($sp[29])
   00406b88:                        : <evform+248> jal 00407148 <xlevlist>
   00406b90:                        : <evform+250> sw $v0[2],20($sp[29])
```

```
   xleval.c:106
   00406b98:                       : <evform+258> lw $v0[2],16($sp[29])
   00406ba0:                       : <evform+260> lw $a0[4],20($sp[29])
   00406ba8:                       : <evform+268> lw $v0[2],4($v0[2])
   00406bb0:                       : <evform+270> jalr $ra[31],$v0[2]


---- Source Extract [xleval.c] [104]
---- (xleval.c)
 92:     oldstk = xlsave(&fun,&args,(NODE **)NULL);
 93:
 94:     /* get the function and the argument list */
 95:     fun = car(expr);
 96:     args = cdr(expr);
 97:
 98:     /* evaluate the first expression */
 99:     if ((fun = xleval(fun)) == NIL)
 100:  xlfail("bad function");
 101:
 102:     /* evaluate the function */
 103:     if (subrp(fun) || fsubrp(fun)) {
*104:  if (subrp(fun))
 105:      args = xlevlist(args);
 106:  val = (*getsubr(fun))(args);
 107:     }
 108:     else if (consp(fun)) {
 109:  if (consp(car(fun))) {
 110:      env = cdr(fun);
 111:      fun = car(fun);
 112:  }
 113:  else
 114:      env = xlenv;
 115:  if ((type = car(fun)) == s_lambda) {
 116:      args = xlevlist(args);


 **** #10 miss at 004073b8: xleval.c:183
   xleval.c:183
   00407390:                       : <xlevlist+248> lw $v0[2],16($sp[29])
   xleval.c:195
   00407398:                       : <xlevlist+250> sw $a0[4],4($s1[17])
   xleval.c:183
   004073a0:                       : <xlevlist+258> lw $v0[2],8($v0[2])
   xleval.c:196
   004073a8:                       : <xlevlist+260> addu $s2[18],$zero[0],$s1[17]
   xleval.c:183
   004073b0:                       : <xlevlist+268> sw $v0[2],16($sp[29])
** 004073b8:  (      27180,   1.88): <xlevlist+270> bne $v0[2],$zero[0],004071e0 <xlevlist+98>
   xleval.c:200
   004073c0:                       : <xlevlist+278> sw $s5[21],-31420($gp[28])
   xleval.c:203
   004073c8:                       : <xlevlist+280> addu $v0[2],$zero[0],$s3[19]
   xleval.c:204
   004073d0:                       : <xlevlist+288> lw $ra[31],48($sp[29])
   004073d8:                       : <xlevlist+290> lw $s5[21],44($sp[29])
   004073e0:                       : <xlevlist+298> lw $s4[20],40($sp[29])
   004073e8:                       : <xlevlist+2a0> lw $s3[19],36($sp[29])


---- Source Extract [xleval.c] [183]
---- (xleval.c)
 171: NODE *xlevlist(NODE *args)
 172: {
 173:     NODE ***oldstk,*src,*dst,*new,*val;
 174:     NODE *last = NIL;
 175:
 176:     /* create a stack frame */
 177:     oldstk = xlsave(&src,&dst,(NODE **)NULL);
 178:
 179:     /* initialize */
 180:     src = args;
 181:
 182:     /* evaluate each argument */
*183:     for (val = NIL; src; src = cdr(src)) {
 184:
 185:  /* check this entry */
 186:  if (!consp(src))
 187:      xlfail("bad argument list");
 188:
 189:  /* allocate a new list entry */
 190:  new = consa(NIL);
 191:  if (val)
 192:      rplacd(last,new);
 193:  else
 194:      val = dst = new;
 195:  rplaca(new,xleval(car(src)));


 **** #11 miss at 004059e8: xldmem.c:270
   mark():
   xldmem.c:266
   004059c0:                       : <mark> addiu $sp[29],$sp[29],-32
```

```
   004059c8:                        : <mark+8> sw $ra[31],28($sp[29])
   004059d0:                        : <mark+10> sw $s2[18],24($sp[29])
   004059d8:                        : <mark+18> sw $s1[17],20($sp[29])
   004059e0:                        : <mark+20> sw $s0[16],16($sp[29])
   xldmem.c:270
** 004059e8:  (      25720,   1.78): <mark+28> beq $a0[4],$zero[0],00405ba8 <mark+1e8>
   xldmem.c:274
   004059f0:                        : <mark+30> addu $s1[17],$zero[0],$zero[0]
   xldmem.c:275
   004059f8:                        : <mark+38> addu $s0[16],$zero[0],$a0[4]
   xldmem.c:278
   00405a00:                        : <mark+40> addiu $s2[18],$zero[0],-3
   xldmem.c:284
   00405a08:                        : <mark+48> lbu $v1[3],1($s0[16])
   00405a10:                        : <mark+50> andi $v0[2],$v1[3],1


---- Source Extract [xldmem.c] [270]
---- (xldmem.c)
 258:     sweep();
 259:
 260:     /* count the gc call */
 261:     gccalls++;
 262: }
 263:
 264: /* mark - mark all accessible nodes */
 265: void mark(NODE *ptr)
 266: {
 267:     NODE *this,*prev,*tmp;
 268:
 269:     /* just return on nil */
*270:     if (ptr == NIL)
 271:  return;
 272:
 273:     /* initialize */
 274:     prev = NIL;
 275:     this = ptr;
 276:
 277:     /* mark this list */
 278:     while (TRUE) {
 279:
 280:  /* descend as far as we can */
 281:  while (TRUE) {
 282:


 **** #12 miss at 00406b50: xleval.c:103
   00406b08:                        : <evform+1c8> lui $a0[4],4096
   00406b10:                        : <evform+1d0> addiu $a0[4],$a0[4],784
   00406b18:                        : <evform+1d8> jal 00404360 <xlfail>
   xleval.c:103
   00406b20:                        : <evform+1e0> lw $v0[2],16($sp[29])
   00406b28:                        : <evform+1e8> beq $v0[2],$zero[0],00406e58 <evform+518>
   00406b30:                        : <evform+1f0> lw $a0[4],16($sp[29])
   00406b38:                        : <evform+1f8> lbu $v1[3],0($a0[4])
   00406b40:                        : <evform+200> addiu $v0[2],$v1[3],-1
   00406b48:                        : <evform+208> sltiu $v0[2],$v0[2],2
** 00406b50:  (      24115,   1.67): <evform+210> beq $v0[2],$zero[0],00406bc8 <evform+288>
   xleval.c:104
   00406b58:  (          2,   0.00): <evform+218> beq $a0[4],$zero[0],00406b98 <evform+258>
   00406b60:                        : <evform+220> sll $v0[2],$v1[3],0x18
   00406b68:                        : <evform+228> sra $v0[2],$v0[2],0x18
   00406b70:                        : <evform+230> addiu $v1[3],$zero[0],1
   00406b78:  (      35901,   2.49): <evform+238> bne $v0[2],$v1[3],00406b98 <evform+258>
   xleval.c:105
   00406b80:                        : <evform+240> lw $a0[4],20($sp[29])
   00406b88:                        : <evform+248> jal 00407148 <xlevlist>


---- Source Extract [xleval.c] [103]
---- (xleval.c)
 91:     /* create a stack frame */
 92:     oldstk = xlsave(&fun,&args,(NODE **)NULL);
 93:
 94:     /* get the function and the argument list */
 95:     fun = car(expr);
 96:     args = cdr(expr);
 97:
 98:     /* evaluate the first expression */
 99:     if ((fun = xleval(fun)) == NIL)
 100:  xlfail("bad function");
 101:
 102:     /* evaluate the function */
*103:     if (subrp(fun) || fsubrp(fun)) {
 104:  if (subrp(fun))
 105:      args = xlevlist(args);
 106:  val = (*getsubr(fun))(args);
 107:     }
 108:     else if (consp(fun)) {
 109:  if (consp(car(fun))) {
 110:      env = cdr(fun);
 111:      fun = car(fun);
```

```
 112:  }
 113:  else
 114:      env = xlenv;
 115:  if ((type = car(fun)) == s_lambda) {


 **** #13 miss at 00406698: xleval.c:31
   00406658:  (          2,   0.00): <xleval+a8> beq $v0[2],$zero[0],00406680 <xleval+d0>
   xleval.c:28
   00406660:                      : <xleval+b0> lw $v1[3],-31388($gp[28])
   00406668:                      : <xleval+b8> sll $v0[2],$a0[4],0x2
   00406670:                      : <xleval+c0> addu $v0[2],$v0[2],$v1[3]
   00406678:                      : <xleval+c8> sw $s0[16],0($v0[2])
   xleval.c:31
   00406680:  (       8671,   0.60): <xleval+d0> beq $s0[16],$zero[0],004066e0 <xleval+130>
   00406688:                      : <xleval+d8> lb $v1[3],0($s0[16])
   00406690:                      : <xleval+e0> addiu $v0[2],$zero[0],3
** 00406698:  (      22900,   1.59): <xleval+e8> bne $v1[3],$v0[2],004066b8 <xleval+108>
   xleval.c:32
   004066a0:                      : <xleval+f0> addu $a0[4],$zero[0],$s0[16]
   004066a8:                      : <xleval+f8> jal 00406940 <evform>
   004066b0:                      : <xleval+100> j 004066d8 <xleval+128>
   xleval.c:33
   004066b8:                      : <xleval+108> addiu $v0[2],$zero[0],4
   004066c0:                      : <xleval+110> bne $v1[3],$v0[2],004066e0 <xleval+130>
   xleval.c:34
   004066c8:                      : <xleval+118> addu $a0[4],$zero[0],$s0[16]


---- Source Extract [xleval.c] [31]
---- (xleval.c)
 19:  oscheck();
 20:     }
 21:
 22:     /* check for *evalhook* */
 23:     if (getvalue(s_evalhook))
 24:  return (evalhook(expr));
 25:
 26:     /* add trace entry */
 27:     if (++xltrace < TDEPTH)
 28:  trace_stack[xltrace] = expr;
 29:
 30:     /* check type of value */
*31:     if (consp(expr))
 32:  expr = evform(expr);
 33:     else if (symbolp(expr))
 34:  expr = xlgetvalue(expr);
 35:
 36:     /* remove trace entry */
 37:     --xltrace;
 38:
 39:     /* return the value */
 40:     return (expr);
 41: }
 42:
 43: /* xlxeval - evaluate an xlisp expression (bypassing *evalhook*) */


 **** #14 miss at 00407c08: xleval.c:327
   xleval.c:327
   00407bd8:                      : <xlsave+c8> addiu $v0[2],$s1[17],3
   00407be0:                      : <xlsave+d0> and $v0[2],$v0[2],$s3[19]
   xleval.c:330
   00407be8:                      : <xlsave+d8> sw $s0[16],-4($v1[3])
   xleval.c:331
   00407bf0:                      : <xlsave+e0> sw $zero[0],0($s0[16])
   xleval.c:327
   00407bf8:                      : <xlsave+e8> lw $s0[16],0($v0[2])
   00407c00:                      : <xlsave+f0> addiu $s1[17],$v0[2],4
** 00407c08:  (      19938,   1.38): <xlsave+f8> bne $s0[16],$zero[0],00407b88 <xlsave+78>
   xleval.c:334
   00407c10:                      : <xlsave+100> addu $v0[2],$zero[0],$s2[18]
   xleval.c:335
   00407c18:                      : <xlsave+108> lw $ra[31],32($sp[29])
   00407c20:                      : <xlsave+110> lw $s3[19],28($sp[29])
   00407c28:                      : <xlsave+118> lw $s2[18],24($sp[29])
   00407c30:                      : <xlsave+120> lw $s1[17],20($sp[29])
   00407c38:                      : <xlsave+128> lw $s0[16],16($sp[29])
   00407c40:                      : <xlsave+130> addiu $sp[29],$sp[29],40


---- Source Extract [xleval.c] [327]
---- (xleval.c)
 315: {
 316:     return (sym == k_optional || sym == k_rest || sym == k_aux);
 317: }
 318:
 319:
 320: /* xlsave - save nodes on the stack */
 321: NODE ***xlsave(NODE **nptr,...)
 322: {
 323:     va_list pvar;
```

```
 324:     NODE ***oldstk;
 325:     oldstk = xlstack;
 326:     va_start(pvar,nptr);
*327:     for (; nptr != (NODE **) NULL; nptr = va_arg(pvar, NODE **)) {
 328:  if (xlstack <= xlstkbase)
 329:      xlabort("evaluation stack overflow");
 330:  *--xlstack = nptr;
 331:  *nptr = NIL;
 332:     }
 333:     va_end(pvar);
 334:     return (oldstk);
 335: }


 **** #15 miss at 00415490: xlsym.c:111
   00415458:  (       4263,   0.30): <xlygetvalue+18> beq $v1[3],$zero[0],00415498 <xlygetvalue+58>
   xlsym.c:112
   00415460:                       : <xlygetvalue+20> lw $a1[5],4($v1[3])
   00415468:                       : <xlygetvalue+28> lw $v0[2],4($a1[5])
   00415470:  (          1,   0.00): <xlygetvalue+30> bne $a0[4],$v0[2],00415488 <xlygetvalue+48>
   xlsym.c:113
   00415478:                       : <xlygetvalue+38> lw $v0[2],8($a1[5])
   00415480:                       : <xlygetvalue+40> j 004154b0 <xlygetvalue+70>
   xlsym.c:111
   00415488:                       : <xlygetvalue+48> lw $v1[3],8($v1[3])
** 00415490:  (      19375,   1.34): <xlygetvalue+50> bne $v1[3],$zero[0],00415460 <xlygetvalue+20>
   xlsym.c:110
   00415498:                       : <xlygetvalue+58> lw $a2[6],8($a2[6])
   004154a0:  (         19,   0.00): <xlygetvalue+60> bne $a2[6],$zero[0],00415450 <xlygetvalue+10>
   xlsym.c:116
   004154a8:                       : <xlygetvalue+68> lw $v0[2],8($a0[4])
   xlsym.c:117
   004154b0:                       : <xlygetvalue+70> jr $ra[31]


---- Source Extract [xlsym.c] [111]
---- (xlsym.c)
 99:
 100:     /* return the global value */
 101:     return (getvalue(sym));
 102: }
 103:
 104: /* xlygetvalue - get the value of a symbol (no instance variables) */
 105: NODE *xlygetvalue(NODE *sym)
 106: {
 107:     register NODE *fp,*ep;
 108:
 109:     /* check the environment list */
 110:     for (fp = xlenv; fp; fp = cdr(fp))
*111:  for (ep = car(fp); ep; ep = cdr(ep))
 112:      if (sym == car(car(ep)))
 113:  return (cdr(car(ep)));
 114:
 115:     /* return the global value */
 116:     return (getvalue(sym));
 117: }
 118:
 119: /* xlsetvalue - set the value of a symbol */
 120: void xlsetvalue(NODE *sym,NODE *val)
 121: {
 122:     NODE *fp,*ep;
 123:


 **** #16 miss at 0040a7c8: xllist.c:62
   0040a780:                       : <cxr+70> beq $v0[2],$zero[0],0040a838 <cxr+128>
   0040a788:  (          3,   0.00): <cxr+78> beq $s0[16],$zero[0],0040a808 <cxr+f8>
   0040a790:                       : <cxr+80> addiu $a1[5],$zero[0],3
   0040a798:                       : <cxr+88> addiu $a0[4],$zero[0],97
   0040a7a0:                       : <cxr+90> lb $v0[2],0($s0[16])
   0040a7a8:  (          1,   0.00): <cxr+98> bne $v0[2],$a1[5],0040a808 <cxr+f8>
   xllist.c:62
   0040a7b0:                       : <cxr+a0> addiu $s1[17],$s1[17],1
   0040a7b8:                       : <cxr+a8> sll $v0[2],$v1[3],0x18
   0040a7c0:                       : <cxr+b0> sra $v0[2],$v0[2],0x18
** 0040a7c8:  (      16945,   1.17): <cxr+b8> bne $v0[2],$a0[4],0040a7e0 <cxr+d0>
   0040a7d0:                       : <cxr+c0> lw $s0[16],4($s0[16])
   0040a7d8:                       : <cxr+c8> j 0040a7e8 <cxr+d8>
   0040a7e0:                       : <cxr+d0> lw $s0[16],8($s0[16])
   0040a7e8:                       : <cxr+d8> lb $v0[2],0($s1[17])
   0040a7f0:                       : <cxr+e0> lbu $v1[3],0($s1[17])
   0040a7f8:  (       2108,   0.15): <cxr+e8> beq $v0[2],$zero[0],0040a838 <cxr+128>
   0040a800:  (          1,   0.00): <cxr+f0> bne $s0[16],$zero[0],0040a7a0 <cxr+90>
   xllist.c:65
   0040a808:                       : <cxr+f8> lb $v0[2],0($s1[17])


---- Source Extract [xllist.c] [62]
---- (xllist.c)
 50:
 51: /* cxr - common car/cdr routine */
 52: LOCAL NODE *cxr(NODE *args,char *adstr)
```

```
 53: {
 54:     NODE *list;
 55:
 56:     /* get the list */
 57:     list = xlmatch(LIST,&args);
 58:     xllastarg(args);
 59:
 60:     /* perform the car/cdr operations */
 61:     while (*adstr && consp(list))
*62:  list = (*adstr++ == 'a' ? car(list) : cdr(list));
 63:
 64:     /* make sure the operation succeeded */
 65:     if (*adstr && list)
 66:  xlfail("bad argument");
 67:
 68:     /* return the result */
 69:     return (list);
 70: }
 71:
 72: /* xcons - construct a new list cell */
 73: NODE *xcons(NODE *args)
 74: {


 **** #17 miss at 00406680: xleval.c:31
   00406640:                        : <xleval+90> addiu $a0[4],$v0[2],1
   00406648:                        : <xleval+98> slti $v0[2],$a0[4],500
   00406650:                        : <xleval+a0> sw $a0[4],-31392($gp[28])
   00406658:  (          2,   0.00): <xleval+a8> beq $v0[2],$zero[0],00406680 <xleval+d0>
   xleval.c:28
   00406660:                        : <xleval+b0> lw $v1[3],-31388($gp[28])
   00406668:                        : <xleval+b8> sll $v0[2],$a0[4],0x2
   00406670:                        : <xleval+c0> addu $v0[2],$v0[2],$v1[3]
   00406678:                        : <xleval+c8> sw $s0[16],0($v0[2])
   xleval.c:31
** 00406680:  (       8671,   0.60): <xleval+d0> beq $s0[16],$zero[0],004066e0 <xleval+130>
   00406688:                        : <xleval+d8> lb $v1[3],0($s0[16])
   00406690:                        : <xleval+e0> addiu $v0[2],$zero[0],3
   00406698:  (      22900,   1.59): <xleval+e8> bne $v1[3],$v0[2],004066b8 <xleval+108>
   xleval.c:32
   004066a0:                        : <xleval+f0> addu $a0[4],$zero[0],$s0[16]
   004066a8:                        : <xleval+f8> jal 00406940 <evform>
   004066b0:                        : <xleval+100> j 004066d8 <xleval+128>
   xleval.c:33
   004066b8:                        : <xleval+108> addiu $v0[2],$zero[0],4


---- Source Extract [xleval.c] [31]
---- (xleval.c)
 19:  oscheck();
 20:     }
 21:
 22:     /* check for *evalhook* */
 23:     if (getvalue(s_evalhook))
 24:  return (evalhook(expr));
 25:
 26:     /* add trace entry */
 27:     if (++xltrace < TDEPTH)
 28:  trace_stack[xltrace] = expr;
 29:
 30:     /* check type of value */
*31:     if (consp(expr))
 32:  expr = evform(expr);
 33:     else if (symbolp(expr))
 34:  expr = xlgetvalue(expr);
 35:
 36:     /* remove trace entry */
 37:     --xltrace;
 38:
 39:     /* return the value */
 40:     return (expr);
 41: }
 42:
 43: /* xlxeval - evaluate an xlisp expression (bypassing *evalhook*) */


 **** #18 miss at 0040cbb0: xllist.c:737
   xllist.c:735
   0040cb88:                        : <xnull+28> jal 004141c8 <xlarg>
   xllist.c:736
   0040cb90:                        : <xnull+30> lw $a0[4],24($sp[29])
   xllist.c:735
   0040cb98:                        : <xnull+38> addu $s0[16],$zero[0],$v0[2]
   xllist.c:736
   0040cba0:                        : <xnull+40> jal 004148e0 <xllastarg>
   xllist.c:737
   0040cba8:                        : <xnull+48> addu $v0[2],$zero[0],$zero[0]
** 0040cbb0:  (       5039,   0.35): <xnull+50> bne $s0[16],$zero[0],0040cbc0 <xnull+60>
   0040cbb8:                        : <xnull+58> lw $v0[2],-31600($gp[28])
   xllist.c:738
   0040cbc0:                        : <xnull+60> lw $ra[31],20($sp[29])
   0040cbc8:                        : <xnull+68> lw $s0[16],16($sp[29])
```

```
   0040cbd0:                       : <xnull+70> addiu $sp[29],$sp[29],24
   0040cbd8:                       : <xnull+78> jr $ra[31]


---- Source Extract [xllist.c] [737]
---- (xllist.c)
 725:     NODE *sym;
 726:     sym = xlmatch(SYM,&args);
 727:     xllastarg(args);
 728:     return (getvalue(sym) == s_unbound ? NIL : true);
 729: }
 730:
 731: /* xnull - is this null? */
 732: NODE *xnull(NODE *args)
 733: {
 734:     NODE *arg;
 735:     arg = xlarg(&args);
 736:     xllastarg(args);
*737:     return (null(arg) ? true : NIL);
 738: }
 739:
 740: /* xlistp - is this a list? */
 741: NODE *xlistp(NODE *args)
 742: {
 743:     NODE *arg;
 744:     arg = xlarg(&args);
 745:     xllastarg(args);
 746:     return (listp(arg) ? true : NIL);
 747: }
 748:
 749: /* xconsp - is this a cons? */


 **** #19 miss at 00407760: xleval.c:264
   xleval.c:259
   00407730:                       : <xlabind+b8> lw $a1[5],4($s3[19])
   00407738:                       : <xlabind+c0> addu $a0[4],$zero[0],$s1[17]
   00407740:                       : <xlabind+c8> addu $a2[6],$zero[0],$s4[20]
   00407748:                       : <xlabind+d0> jal 00415228 <xlbind>
   xleval.c:262
   00407750:                       : <xlabind+d8> lw $s0[16],8($s0[16])
   xleval.c:263
   00407758:                       : <xlabind+e0> lw $s3[19],8($s3[19])
   xleval.c:264
** 00407760:  (       4550,   0.31): <xlabind+e8> beq $s0[16],$zero[0],00407a40 <xlabind+3c8>
   00407768:                       : <xlabind+f0> lb $v0[2],0($s0[16])
   00407770:  (          2,   0.00): <xlabind+f8> beq $v0[2],$s2[18],004076f8 <xlabind+80>
   xleval.c:267
   00407778:                       : <xlabind+100> beq $s0[16],$zero[0],00407a40 <xlabind+3c8>
   00407780:                       : <xlabind+108> lb $a0[4],0($s0[16])
   00407788:                       : <xlabind+110> addiu $v0[2],$zero[0],3
   00407790:                       : <xlabind+118> bne $a0[4],$v0[2],004078e0 <xlabind+268>
   00407798:                       : <xlabind+120> lw $v1[3],4($s0[16])
   004077a0:                       : <xlabind+128> lw $v0[2],-31476($gp[28])


---- Source Extract [xleval.c] [264]
---- (xleval.c)
 252: {
 253:     NODE *arg;
 254:
 255:     /* evaluate and bind each required argument */
 256:     while (consp(fargs) && !iskeyword(arg = car(fargs)) && consp(aargs)) {
 257:
 258:  /* bind the formal variable to the argument value */
 259:  xlbind(arg,car(aargs),env);
 260:
 261:  /* move the argument list pointers ahead */
 262:  fargs = cdr(fargs);
 263:  aargs = cdr(aargs);
*264:     }
 265:
 266:     /* check for the '&optional' keyword */
 267:     if (consp(fargs) && car(fargs) == k_optional) {
 268:  fargs = cdr(fargs);
 269:
 270:  /* bind the arguments that were supplied */
 271:  while (consp(fargs) && !iskeyword(arg = car(fargs)) && consp(aargs)) {
 272:
 273:      /* bind the formal variable to the argument value */
 274:      xlbind(arg,car(aargs),env);
 275:
 276:      /* move the argument list pointers ahead */


 **** #20 miss at 00415458: xlsym.c:111
   xlygetvalue():
   xlsym.c:110
   00415440:                       : <xlygetvalue> lw $a2[6],-31408($gp[28])
   00415448:  (          8,   0.00): <xlygetvalue+8> beq $a2[6],$zero[0],004154a8 <xlygetvalue+68>
   xlsym.c:111
   00415450:                       : <xlygetvalue+10> lw $v1[3],4($a2[6])
```

```
** 00415458:  (       4263,   0.30): <xlygetvalue+18> beq $v1[3],$zero[0],00415498 <xlygetvalue+58>
   xlsym.c:112
   00415460:                        : <xlygetvalue+20> lw $a1[5],4($v1[3])
   00415468:                        : <xlygetvalue+28> lw $v0[2],4($a1[5])
   00415470:  (          1,   0.00): <xlygetvalue+30> bne $a0[4],$v0[2],00415488 <xlygetvalue+48>
   xlsym.c:113
   00415478:                        : <xlygetvalue+38> lw $v0[2],8($a1[5])
   00415480:                        : <xlygetvalue+40> j 004154b0 <xlygetvalue+70>
   xlsym.c:111
   00415488:                        : <xlygetvalue+48> lw $v1[3],8($v1[3])


---- Source Extract [xlsym.c] [111]
---- (xlsym.c)
 99:
 100:     /* return the global value */
 101:     return (getvalue(sym));
 102: }
 103:
 104: /* xlygetvalue - get the value of a symbol (no instance variables) */
 105: NODE *xlygetvalue(NODE *sym)
 106: {
 107:     register NODE *fp,*ep;
 108:
 109:     /* check the environment list */
 110:     for (fp = xlenv; fp; fp = cdr(fp))
*111:  for (ep = car(fp); ep; ep = cdr(ep))
 112:      if (sym == car(car(ep)))
 113:  return (cdr(car(ep)));
 114:
 115:     /* return the global value */
 116:     return (getvalue(sym));
 117: }
 118:
 119: /* xlsetvalue - set the value of a symbol */
 120: void xlsetvalue(NODE *sym,NODE *val)
 121: {
 122:     NODE *fp,*ep;
 123:


 **** #21 miss at 00415930: xlsym.c:190
   xlsym.c:190
   00415908:                        : <hash+40> lbu $a3[7],0($a2[6])
   xlsym.c:191
   00415910:                        : <hash+48> sll $v1[3],$v1[3],0x2
   xlsym.c:190
   00415918:                        : <hash+50> lb $a0[4],0($a2[6])
   xlsym.c:191
   00415920:                        : <hash+58> sra $v0[2],$v0[2],0x18
   00415928:                        : <hash+60> xor $v1[3],$v1[3],$v0[2]
   xlsym.c:190
** 00415930:  (       2666,   0.18): <hash+68> bne $a0[4],$zero[0],004158f8 <hash+30>
   xlsym.c:192
   00415938:                        : <hash+70> div $zero[0],$v1[3],$a1[5]
   00415940:  (          5,   0.00): <hash+78> bne $a1[5],$zero[0],00415950 <hash+88>
   00415948:                        : <hash+80> break
   00415950:                        : <hash+88> addiu $at[1],$zero[0],-1
   00415958:  (          2,   0.00): <hash+90> bne $a1[5],$at[1],00415978 <hash+b0>
   00415960:                        : <hash+98> lui $at[1],32768
   00415968:                        : <hash+a0> bne $v1[3],$at[1],00415978 <hash+b0>
   00415970:                        : <hash+a8> break


---- Source Extract [xlsym.c] [190]
---- (xlsym.c)
 178: {
 179:     NODE *p;
 180:     for (p = getplist(sym); consp(p) && consp(cdr(p)); p = cdr(cdr(p)))
 181:  if (car(p) == prp)
 182:      return (cdr(p));
 183:     return (NIL);
 184: }
 185:
 186: /* hash - hash a symbol name string */
 187: int hash(char *str,int len)
 188: {
 189:     int i;
*190:     for (i = 0; *str; )
 191:  i = (i << 2) ^ *str++;
 192:     i %= len;
 193:     return (abs(i));
 194: }
 195:
 196: /* xlsinit - symbol initialization routine */
 197: void xlsinit(void)
 198: {
 199:     NODE *array,*p;
 200:
 201:     /* initialize the obarray */
 202:     obarray = xlmakesym("*OBARRAY*",STATIC);
```

```
 **** #22 miss at 004071d0: xleval.c:183
   004071a8:                       : <xlevlist+60> sw $s1[17],28($sp[29])
   xleval.c:177
   004071b0:                       : <xlevlist+68> jal 00407b10 <xlsave>
   xleval.c:183
   004071b8:                       : <xlevlist+70> addu $s3[19],$zero[0],$zero[0]
   xleval.c:177
   004071c0:                       : <xlevlist+78> addu $s5[21],$zero[0],$v0[2]
   xleval.c:180
   004071c8:                       : <xlevlist+80> sw $s0[16],16($sp[29])
   xleval.c:183
** 004071d0:  (       2600,   0.18): <xlevlist+88> beq $s0[16],$zero[0],004073c0 <xlevlist+278>
   004071d8:                       : <xlevlist+90> addiu $s4[20],$zero[0],3
   xleval.c:186
   004071e0:                       : <xlevlist+98> lw $v0[2],16($sp[29])
   004071e8:                       : <xlevlist+a0> lb $v0[2],0($v0[2])
   004071f0:  (          1,   0.00): <xlevlist+a8> beq $v0[2],$s4[20],00407210 <xlevlist+c8>
   xleval.c:187
   004071f8:                       : <xlevlist+b0> lui $a0[4],4096
   00407200:                       : <xlevlist+b8> addiu $a0[4],$a0[4],820
   00407208:                       : <xlevlist+c0> jal 00404360 <xlfail>


---- Source Extract [xleval.c] [183]
---- (xleval.c)
 171: NODE *xlevlist(NODE *args)
 172: {
 173:     NODE ***oldstk,*src,*dst,*new,*val;
 174:     NODE *last = NIL;
 175:
 176:     /* create a stack frame */
 177:     oldstk = xlsave(&src,&dst,(NODE **)NULL);
 178:
 179:     /* initialize */
 180:     src = args;
 181:
 182:     /* evaluate each argument */
*183:     for (val = NIL; src; src = cdr(src)) {
 184:
 185:  /* check this entry */
 186:  if (!consp(src))
 187:      xlfail("bad argument list");
 188:
 189:  /* allocate a new list entry */
 190:  new = consa(NIL);
 191:  if (val)
 192:      rplacd(last,new);
 193:  else
 194:      val = dst = new;
 195:  rplaca(new,xleval(car(src)));


 **** #23 miss at 00413000: xlread.c:558
   xlread.c:558
   00412fb8:                       : <nextch+30> addu $a0[4],$zero[0],$s0[16]
   00412fc0:                       : <nextch+38> jal 00409350 <xlpeek>
   00412fc8:                       : <nextch+40> addu $a0[4],$zero[0],$v0[2]
   00412fd0:                       : <nextch+48> beq $a0[4],$s1[17],00413020 <nextch+98>
   00412fd8:                       : <nextch+50> lw $v0[2],-30528($gp[28])
   00412fe0:                       : <nextch+58> sll $v1[3],$a0[4],0x1
   00412fe8:                       : <nextch+60> addu $v1[3],$v1[3],$v0[2]
   00412ff0:                       : <nextch+68> lhu $v0[2],0($v1[3])
   00412ff8:                       : <nextch+70> andi $v0[2],$v0[2],16
** 00413000:  (       2475,   0.17): <nextch+78> beq $v0[2],$zero[0],00413020 <nextch+98>
   xlread.c:559
   00413008:                       : <nextch+80> addu $a0[4],$zero[0],$s0[16]
   00413010:                       : <nextch+88> jal 004090e0 <xlgetc>
   00413018:                       : <nextch+90> j 00412fb8 <nextch+30>
   xlread.c:560
   00413020:                       : <nextch+98> addu $v0[2],$zero[0],$a0[4]
   xlread.c:561
   00413028:                       : <nextch+a0> lw $ra[31],24($sp[29])
   00413030:                       : <nextch+a8> lw $s1[17],20($sp[29])


---- Source Extract [xlread.c] [558]
---- (xlread.c)
 546:     rtable = getvalue(s_rtable);
 547:     if (!vectorp(rtable) || ch < 0 || ch >= getsize(rtable))
 548:  return (NIL);
 549:     return (getelement(rtable,ch));
 550: }
 551:
 552: /* nextch - look at the next non-blank character */
 553: LOCAL int nextch(NODE *fptr)
 554: {
 555:     int ch;
 556:
 557:     /* return and save the next non-blank character */
*558:     while ((ch = xlpeek(fptr)) != EOF && isspace(ch))
 559:  xlgetc(fptr);
```

```
 560:     return (ch);
 561: }
 562:
 563: /* checkeof - get a character and check for end of file */
 564: LOCAL int checkeof(NODE *fptr)
 565: {
 566:     int ch;
 567:
 568:     if ((ch = xlgetc(fptr)) == EOF)
 569:  badeof(fptr);
 570:     return (ch);


 **** #24 miss at 0040c930: xllist.c:701
   xllist.c:700
   0040c900:                       : <xatom+30> lw $a0[4],24($sp[29])
   xllist.c:699
   0040c908:                       : <xatom+38> addu $s0[16],$zero[0],$v0[2]
   xllist.c:700
   0040c910:                       : <xatom+40> jal 004148e0 <xllastarg>
   xllist.c:701
   0040c918:  (          1,   0.00): <xatom+48> beq $s0[16],$zero[0],0040c938 <xatom+68>
   0040c920:                       : <xatom+50> lb $v1[3],0($s0[16])
   0040c928:                       : <xatom+58> addiu $v0[2],$zero[0],3
** 0040c930:  (       2321,   0.16): <xatom+60> beq $v1[3],$v0[2],0040c948 <xatom+78>
   0040c938:                       : <xatom+68> lw $v0[2],-31600($gp[28])
   0040c940:                       : <xatom+70> j 0040c950 <xatom+80>
   0040c948:                       : <xatom+78> addu $v0[2],$zero[0],$zero[0]
   xllist.c:702
   0040c950:                       : <xatom+80> lw $ra[31],20($sp[29])
   0040c958:                       : <xatom+88> lw $s0[16],16($sp[29])
   0040c960:                       : <xatom+90> addiu $sp[29],$sp[29],24
   0040c968:                       : <xatom+98> jr $ra[31]


---- Source Extract [xllist.c] [701]
---- (xllist.c)
 689:     xlstack = oldstk;
 690:
 691:     /* return the updated list */
 692:     return (val);
 693: }
 694:
 695: /* xatom - is this an atom? */
 696: NODE *xatom(NODE *args)
 697: {
 698:     NODE *arg;
 699:     arg = xlarg(&args);
 700:     xllastarg(args);
*701:     return (atom(arg) ? true : NIL);
 702: }
 703:
 704: /* xsymbolp - is this an symbol? */
 705: NODE *xsymbolp(NODE *args)
 706: {
 707:     NODE *arg;
 708:     arg = xlarg(&args);
 709:     xllastarg(args);
 710:     return (arg == NIL || symbolp(arg) ? true : NIL);
 711: }
 712:
 713: /* xnumberp - is this a number? */


 **** #25 miss at 00412df8: xlread.c:531
   00412db0:                       : <pname+f0> sb $v0[2],0($a0[4])
   xlread.c:531
   00412db8:                       : <pname+f8> addu $a0[4],$zero[0],$s2[18]
   00412dc0:                       : <pname+100> jal 00409350 <xlpeek>
   00412dc8:                       : <pname+108> addu $s0[16],$zero[0],$v0[2]
   00412dd0:                       : <pname+110> beq $s0[16],$s4[20],00412e48 <pname+188>
   00412dd8:                       : <pname+118> addu $a0[4],$zero[0],$s0[16]
   00412de0:                       : <pname+120> jal 00412f00 <tentry>
   00412de8:                       : <pname+128> lw $v1[3],-31488($gp[28])
   00412df0:                       : <pname+130> addu $a0[4],$zero[0],$v0[2]
** 00412df8:  (       2254,   0.16): <pname+138> beq $a0[4],$v1[3],00412e30 <pname+170>
   00412e00:                       : <pname+140> beq $a0[4],$zero[0],00412e48 <pname+188>
   00412e08:                       : <pname+148> lb $v0[2],0($a0[4])
   00412e10:  (        681,   0.05): <pname+150> bne $v0[2],$s3[19],00412e48 <pname+188>
   00412e18:                       : <pname+158> lw $v1[3],4($a0[4])
   00412e20:                       : <pname+160> lw $v0[2],-31484($gp[28])
   00412e28:                       : <pname+168> bne $v1[3],$v0[2],00412e48 <pname+188>
   xlread.c:528
   00412e30:                       : <pname+170> addu $a0[4],$zero[0],$s2[18]
   00412e38:                       : <pname+178> jal 004090e0 <xlgetc>


---- Source Extract [xlread.c] [531]
---- (xlread.c)
 519: }
 520:
 521: /* pname - parse a symbol name */
```

```
 522: LOCAL NODE *pname(NODE *fptr,int ch)
 523: {
 524:     NODE *val,*type;
 525:     int i;
 526:
 527:     /* get symbol name */
 528:     for (i = 0; ; xlgetc(fptr)) {
 529:  if (i < STRMAX)
 530:      buf[i++] = (islower(ch) ? toupper(ch) : ch);
*531:  if ((ch = xlpeek(fptr)) == EOF ||
 532:      ((type = tentry(ch)) != k_const &&
 533:              !(consp(type) && car(type) == k_nmacro)))
 534:      break;
 535:     }
 536:     buf[i] = 0;
 537:
 538:     /* check for a number or enter the symbol into the oblist */
 539:     return (isnumber(buf,&val) ? val : xlenter(buf,DYNAMIC));
 540: }
 541:
 542: /* tentry - get a readtable entry */
 543: LOCAL NODE *tentry(int ch)


 **** #26 miss at 0040a7f8: xllist.c:62
   xllist.c:62
   0040a7b0:                        : <cxr+a0> addiu $s1[17],$s1[17],1
   0040a7b8:                        : <cxr+a8> sll $v0[2],$v1[3],0x18
   0040a7c0:                        : <cxr+b0> sra $v0[2],$v0[2],0x18
   0040a7c8:  (      16945,   1.17): <cxr+b8> bne $v0[2],$a0[4],0040a7e0 <cxr+d0>
   0040a7d0:                        : <cxr+c0> lw $s0[16],4($s0[16])
   0040a7d8:                        : <cxr+c8> j 0040a7e8 <cxr+d8>
   0040a7e0:                        : <cxr+d0> lw $s0[16],8($s0[16])
   0040a7e8:                        : <cxr+d8> lb $v0[2],0($s1[17])
   0040a7f0:                        : <cxr+e0> lbu $v1[3],0($s1[17])
** 0040a7f8:  (       2108,   0.15): <cxr+e8> beq $v0[2],$zero[0],0040a838 <cxr+128>
   0040a800:  (          1,   0.00): <cxr+f0> bne $s0[16],$zero[0],0040a7a0 <cxr+90>
   xllist.c:65
   0040a808:                        : <cxr+f8> lb $v0[2],0($s1[17])
   0040a810:                        : <cxr+100> beq $v0[2],$zero[0],0040a838 <cxr+128>
   0040a818:                        : <cxr+108> beq $s0[16],$zero[0],0040a838 <cxr+128>
   xllist.c:66
   0040a820:                        : <cxr+110> lui $a0[4],4096
   0040a828:                        : <cxr+118> addiu $a0[4],$a0[4],1888
   0040a830:                        : <cxr+120> jal 00404360 <xlfail>


---- Source Extract [xllist.c] [62]
---- (xllist.c)
 50:
 51: /* cxr - common car/cdr routine */
 52: LOCAL NODE *cxr(NODE *args,char *adstr)
 53: {
 54:     NODE *list;
 55:
 56:     /* get the list */
 57:     list = xlmatch(LIST,&args);
 58:     xllastarg(args);
 59:
 60:     /* perform the car/cdr operations */
 61:     while (*adstr && consp(list))
*62:  list = (*adstr++ == 'a' ? car(list) : cdr(list));
 63:
 64:     /* make sure the operation succeeded */
 65:     if (*adstr && list)
 66:  xlfail("bad argument");
 67:
 68:     /* return the result */
 69:     return (list);
 70: }
 71:
 72: /* xcons - construct a new list cell */
 73: NODE *xcons(NODE *args)
 74: {


 **** #27 miss at 004091d8: xlio.c:39
   004091a8:                        : <xlgetc+c8> sw $v0[2],4($s0[16])
   004091b0:                        : <xlgetc+d0> bne $v0[2],$zero[0],004091c0 <xlgetc+e0>
   xlio.c:33
   004091b8:                        : <xlgetc+d8> sw $zero[0],8($s0[16])
   xlio.c:34
   004091c0:                        : <xlgetc+e0> lw $v1[3],4($s2[18])
   xlio.c:36
   004091c8:                        : <xlgetc+e8> j 00409318 <xlgetc+238>
   xlio.c:39
   004091d0:                        : <xlgetc+f0> lw $v1[3],8($s0[16])
** 004091d8:  (       1791,   0.12): <xlgetc+f8> beq $v1[3],$zero[0],004091f0 <xlgetc+110>
   xlio.c:40
   004091e0:                        : <xlgetc+100> sw $zero[0],8($s0[16])
   004091e8:                        : <xlgetc+108> j 00409318 <xlgetc+238>
   xlio.c:49
```

```
   004091f0:                        : <xlgetc+110> lw $v0[2],-31376($gp[28])
   xlio.c:46
   004091f8:                        : <xlgetc+118> lw $s1[17],4($s0[16])
   xlio.c:49
   00409200:                        : <xlgetc+120> beq $v0[2],$zero[0],004092c0 <xlgetc+1e0>


---- Source Extract [xlio.c] [39]
---- (xlio.c)
 27:      ch = EOF;
 28:  else {
 29:      if (!consp(lptr) ||
 30:  (cptr = car(lptr)) == NIL || !fixp(cptr))
 31:  xlfail("bad stream");
 32:      if (rplaca(fptr,cdr(lptr)) == NIL)
 33:  rplacd(fptr,NIL);
 34:      ch = getfixnum(cptr);
 35:  }
 36:     }
 37:
 38:     /* otherwise, check for a buffered file character */
*39:     else if (ch = getsavech(fptr))
 40:  setsavech(fptr,0);
 41:
 42:     /* otherwise, get a new character */
 43:     else {
 44:
 45:  /* get the file pointer */
 46:  fp = getfile(fptr);
 47:
 48:  /* prompt if necessary */
 49:  if (prompt && fp == stdin) {
 50:
 51:      /* print the debug level */


 **** #28 miss at 0041a248: ../sysdeps/generic/strcmp.c:37
   ../sysdeps/generic/strcmp.c:35
   0041a220:                        : <strcmp> lbu $v0[2],0($a0[4])
   ../sysdeps/generic/strcmp.c:36
   0041a228:                        : <strcmp+8> lbu $v1[3],0($a1[5])
   ../sysdeps/generic/strcmp.c:35
   0041a230:                        : <strcmp+10> addiu $a0[4],$a0[4],1
   ../sysdeps/generic/strcmp.c:36
   0041a238:                        : <strcmp+18> addiu $a1[5],$a1[5],1
   ../sysdeps/generic/strcmp.c:37
   0041a240:                        : <strcmp+20> andi $v0[2],$v0[2],255
** 0041a248:  (       1573,   0.11): <strcmp+28> beq $v0[2],$zero[0],0041a270 <strcmp+50>
   ../sysdeps/generic/strcmp.c:40
   0041a250:                        : <strcmp+30> andi $v1[3],$v1[3],255
   0041a258:  (       1118,   0.08): <strcmp+38> beq $v0[2],$v1[3],0041a220 <strcmp>
   ../sysdeps/generic/strcmp.c:42
   0041a260:                        : <strcmp+40> subu $v0[2],$v0[2],$v1[3]
   0041a268:                        : <strcmp+48> j 0041a278 <strcmp+58>
   ../sysdeps/generic/strcmp.c:38
   0041a270:                        : <strcmp+50> subu $v0[2],$zero[0],$v1[3]
   ../sysdeps/generic/strcmp.c:43


---- Source Extract [../sysdeps/generic/strcmp.c] [37]
---- (/pong/usr3/s/smithz/research/tools/src/glibc-1.09/sysdeps/generic/strcmp.c)
 25:    equal to or greater than S2.  */
 26: int
 27: DEFUN(strcmp, (p1, p2), CONST char *p1 AND CONST char *p2)
 28: {
 29:   register CONST unsigned char *s1 = (CONST unsigned char *) p1;
 30:   register CONST unsigned char *s2 = (CONST unsigned char *) p2;
 31:   unsigned reg_char c1, c2;
 32:
 33:   do
 34:     {
 35:       c1 = (unsigned char) *s1++;
 36:       c2 = (unsigned char) *s2++;
*37:       if (c1 == '\0')
 38:  return c1 - c2;
 39:     }
 40:   while (c1 == c2);
 41:
 42:   return c1 - c2;
 43: }


 **** #29 miss at 00407608: xleval.c:237
   004075d8:                        : <evfun+168> sw $v1[3],24($sp[29])
   004075e0:                        : <evfun+170> beq $v1[3],$zero[0],00407610 <evfun+1a0>
   xleval.c:238
   004075e8:                        : <evfun+178> addiu $a0[4],$sp[29],24
   004075f0:                        : <evfun+180> jal 00414388 <xlevarg>
   xleval.c:237
   004075f8:                        : <evfun+188> lw $v1[3],24($sp[29])
   xleval.c:238
   00407600:                        : <evfun+190> addu $s4[20],$zero[0],$v0[2]
```

```
   xleval.c:237
** 00407608:  (       1526,   0.11): <evfun+198> bne $v1[3],$zero[0],004075e8 <evfun+178>
   xleval.c:241
   00407610:                        : <evfun+1a0> lw $v1[3],16($sp[29])
   xleval.c:247
   00407618:                        : <evfun+1a8> addu $v0[2],$zero[0],$s4[20]
   xleval.c:244
   00407620:                        : <evfun+1b0> sw $s5[21],-31420($gp[28])
   xleval.c:241
   00407628:                        : <evfun+1b8> sw $v1[3],-31408($gp[28])
   xleval.c:248


---- Source Extract [xleval.c] [237]
---- (xleval.c)
 225:     if ((fargs = car(fun)) && !consp(fargs))
 226:  xlfail("bad formal argument list");
 227:
 228:     /* create a new environment frame */
 229:     newenv = xlframe(env);
 230:     oldenv = xlenv;
 231:
 232:     /* bind the formal parameters */
 233:     xlabind(fargs,args,newenv);
 234:     xlenv = newenv;
 235:
 236:     /* execute the code */
*237:     for (cptr = cdr(fun); cptr; )
 238:  val = xlevarg(&cptr);
 239:
 240:     /* restore the environment */
 241:     xlenv = oldenv;
 242:
 243:     /* restore the previous stack frame */
 244:     xlstack = oldstk;
 245:
 246:     /* return the result value */
 247:     return (val);
 248: }
 249:


 **** #30 miss at 0040ce78: xllist.c:787
   0040ce40:                        : <cequal+60> lw $a0[4],32($sp[29])
   xllist.c:783
   0040ce48:                        : <cequal+68> addu $s0[16],$zero[0],$v0[2]
   xllist.c:784
   0040ce50:                        : <cequal+70> jal 004148e0 <xllastarg>
   xllist.c:787
   0040ce58:                        : <cequal+78> addu $a0[4],$zero[0],$s1[17]
   0040ce60:                        : <cequal+80> addu $a1[5],$zero[0],$s0[16]
   0040ce68:                        : <cequal+88> jalr $ra[31],$s2[18]
   0040ce70:                        : <cequal+90> addu $v1[3],$zero[0],$zero[0]
** 0040ce78:  (       1384,   0.10): <cequal+98> beq $v0[2],$zero[0],0040ce88 <cequal+a8>
   0040ce80:                        : <cequal+a0> lw $v1[3],-31600($gp[28])
   0040ce88:                        : <cequal+a8> addu $v0[2],$zero[0],$v1[3]
   xllist.c:788
   0040ce90:                        : <cequal+b0> lw $ra[31],28($sp[29])
   0040ce98:                        : <cequal+b8> lw $s2[18],24($sp[29])
   0040cea0:                        : <cequal+c0> lw $s1[17],20($sp[29])
   0040cea8:                        : <cequal+c8> lw $s0[16],16($sp[29])
   0040ceb0:                        : <cequal+d0> addiu $sp[29],$sp[29],32
   0040ceb8:                        : <cequal+d8> jr $ra[31]


---- Source Extract [xllist.c] [787]
---- (xllist.c)
 775:
 776: /* cequal - common eq/eql/equal function */
 777: LOCAL NODE *cequal(NODE *args, int (*fcn)())
 778: {
 779:     NODE *arg1,*arg2;
 780:
 781:     /* get the two arguments */
 782:     arg1 = xlarg(&args);
 783:     arg2 = xlarg(&args);
 784:     xllastarg(args);
 785:
 786:     /* compare the arguments */
*787:     return ((*fcn)(arg1,arg2) ? true : NIL);
 788: }
 789:
```