The Feasibility of 8-bit Load Value Prediction

Vincent Weaver
hppa13
ECE575

6 May 2004, Minor Revisions 12 January 2009

1 Project Summary

This project investigates the tradeoffs in using an 8-bit load value predictor on modern CPUs in cases where a full 64-bit load value predictor would not be feasible.

I used ATOM [3] to instrument all 8-bit loads for four different benchmarks (two from SPEC CPU 2000 [4]). I simulated eight different value predictors, including some that would not be implementable on 64-bit machines due to size constraints.

The results show that while the 8-bit values are indeed more predictable, 8-bit loads are only a small fraction of the total amount of loads. This unfortunately means that any increase in performance would most likely be negligible.

2 Project Description

On modern architectures most loads tend to be 64-bit or 32-bit values. Despite this, many CPUs still support 8-bit and 16-bit loads. A normal load value predictor treats these shorter loads as a full-sized load, but this is wasteful; it throws away the knowledge that the top 56 bits are zero.

8-bit values should be easier to predict than larger loads because there are only 256 possible values. A common use of 8-bit loads on UNIX based systems is the processing of ASCII text files, and since ASCII text has low entropy this should make these types of loads more predictable.

The goal of this project is to find an 8-bit predictor with a 90% or higher accuracy rate. A small effective predictor of this sort would greatly speed certain types of processor behavior, especially text and legacy applications. If such a simple and small predictor could be found, and if it was obvious that performance would greatly increase, then it might open the door to further value predictor use in industry.

3 Setup

3.1 Benchmarks

I used two of the SPEC CPU 2000 benchmarks: *equake*, *cc1*. Two other benchmarks were investigated, that while in SPEC CPU 2000, were not the official SPEC versions: *gcc*, and *gzip*.

The machine used for the test was dual-processor Alpha 21264 machine.

equake and *cc1* are the precompiled SPEC 2000 benchmarks provided for the ECE575 class by Professor Burtscher. The test inputs were used.

gcc is the Alpha cc1 binary from version 3.3 of gcc installed on our test machine. For input the *tbo.c* file is compiled, which is a short video game written in very dense and obfuscated C (http://deater.net/weave/vmwprod/tb1/tbo.html).

gzip is gzip version 1.2.4 compiled from source (using the "-arch host" compiler option) on the Alpha. For benchmarking the gzip-1.2.4.tar file is used as the input.

3.2 Instrumentation

Instrumentation was not a trivial task. The instrumentation files used (*freq.inst.c* and *proj6.inst.c*) are similar to standard load-value prediction instrumentation files from other projects.

Finding 8-bit loads is surprisingly hard. The Alpha instruction to load a byte is LDBU. This instruction is relatively new; the initial Alpha implementation did not support 8-bit reads. Originally 8-bit reads were implemented by a series of three instructions: LDQ_U, LDA, then EXTBL.

Experiments by DEC showed this overhead on 8-bit loads was a performance bottleneck, especially on legacy code (and especially when running Windows NT) [2]. The LDBU instruction was introduced on newer CPUs such as the Alpha 21264 and the EXTBL way of loading bytes is deprecated [1] in favor of LDBU.

Unfortunately, on closer inspection, the old EXTBL load method is still used in many of the binaries on our Alpha test system (including *cc1* and *equake*). These uses seem to be in the C library routines.

Proper detection of EXTBL based loads is not easy, it requires finding three instructions which may have been scattered all over by the compiler.

It would be nearly impossible to build 8-bit load-value prediction circuitry that used EXTBL-based 8-bit loads. All benefits from 8-bit prediction would be lost if you first had to load the 64-bit value from memory before the byte-extraction took place. It could be possible to treat all 64-bit loads as speculative and then predicting 8-bit values based on that, but that would add a lot of needless complexity.

For the purposes of this project an EXTBL instruction is treated the same as a LDBU. This is how a modern Alpha system would work, and it is similar to how other RISC designs with dedicated 8-bit load instructions work.

Both *equake* and *cc1* were disassembled and investigated to make sure this assumption was sane, and it does seem EXTBL is used almost exclusively to handle 8-bit loads. Unfortunately doing things this way limited some of the analysis that could be done on hybrid 64-bit/8-bit predictors.

gcc and *gzip* were atomized with the "-a" option which also atomized all relevant shared libraries, and the results from those libraries are included with the runs.

4 Results

4.1 Frequency of 8-bit Loads

The relative mix of 64-bit and 8-bit loads in the various benchmarks is shown in Table 1. The dynamic percentage of 8-bit loads varies widely, from below 3% up to 31%! This does not bode well for 8-bit values having an overwhelming impact in program execution speed with the possible exception of *gzip* (and we will find later that *gzip* has poor 8-bit predictability).

It makes sense that *equake*, a floating point program, has the least number of 8-bit loads whereas the programs that do a lot of text parsing and text disk I/O have much higher ratios.

Program	Static Loads	% 8-bit	Dynamic Loads	% 8-bit
equake	12889	4.52%	360250863	2.78%
cc1	93053	6.37%	376933000	8.00%
gcc	185425	6.33%	1771232	9.26%
gzip	55816	6.43%	11144290	31.05%

Table 1: Ratio of loads that are 8-bit

4.2 Frequent Value Results

Figure 1 shows the frequent load value results for the various benchmarks. This is simple for 8-bit values, as only 256 values are possible so a simple set of counters suffices.

We see that for *cc1* there are only a few spikes of frequent values: various lowercase letters and a few other symbols you would expect in a C file. The one unexpected result is 0x4 which appears often for unknown reasons.

In *equake* besides the NUL value, all of the 8-bit loads are related to reading the text input from disk. The ASCII values for 0-9 and the decimal point are prominent as would be expected for floating point values. The other values can be accounted for by %lf and such format specifiers in scanf and printf statements. This is hardly surprising as equake has no other need for 8-bit values.

The *gcc* results are what one would expect for something parsing C code. The distribution of characters looks similar to an English language distribution. Of curious note are the loads of 1, 2, 4 and 5 which would not normally be expected.

The *gzip* results unsurprisingly match more or less the distribution of characters in the tar file being processed.

All values above 128 were ignored for plotting purposes; in no case were there any significant results with the top bit set. This indicates the reads were largely ASCII values.

The files *freq.inst.c* and *freq.anal.c* were used to gather this data.

4.3 Predictor Results

We explored the use of various types of predictors to see which had the best performance.

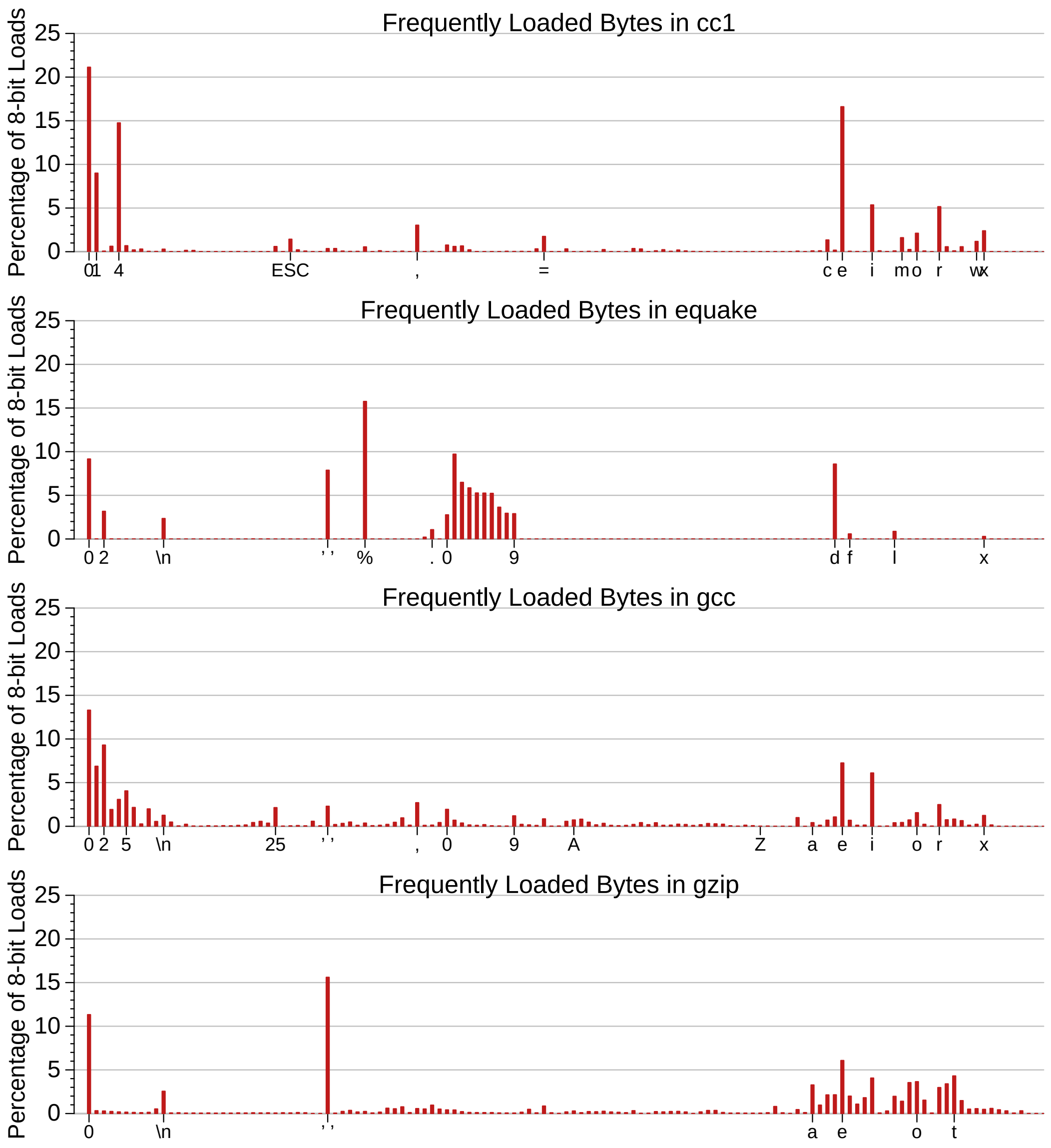

Figure 1: 8-bit load frequencies. For the C compilers this represents primarily the input C file. For equake it corresponds to the C scanf string ("%lf") as well as the ASCII representation of the floating point values in the input file.

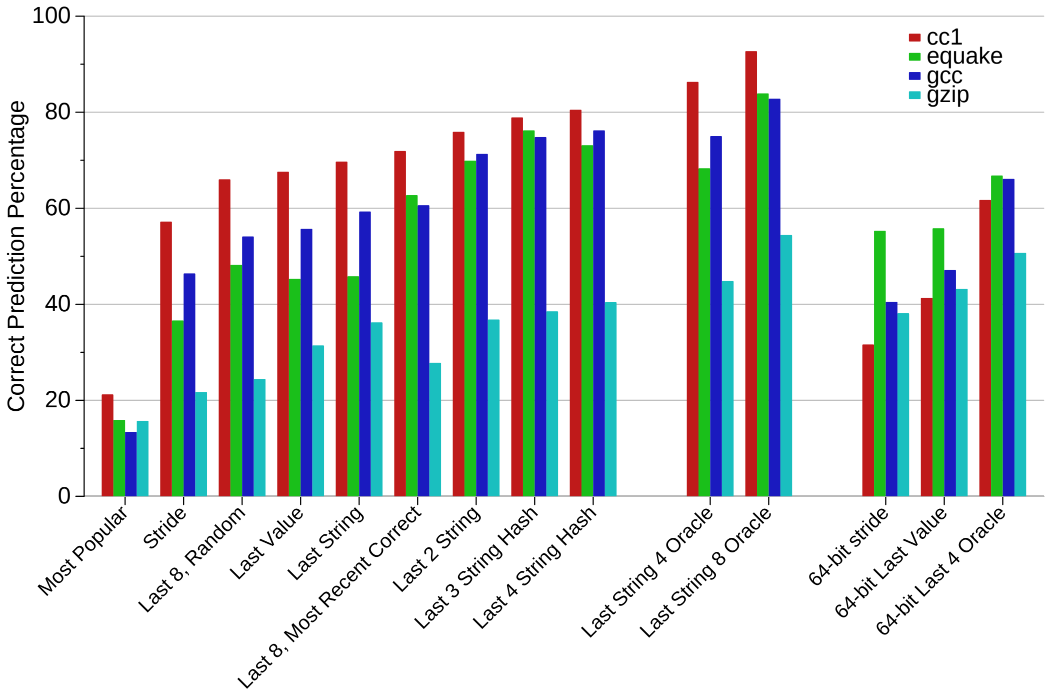

Figure 2: Results of various predictors

Popularity Predictor The popularity predictor has 256 counters and keeps track of the load popularity of each possible 8-bit value. The most popular value is tracked, and this value is predicted. This predictor did poorly, never much better than 20%. This is not surprising, as the frequency graphs in Figure 1 show that no one value ever appeared more than 22% of the time.

Space-wise, the popularity predictor uses 256 8-bit counters and one 8-bit most-popular value for a total of 257 bytes.

Stride Predictor This predictor behaved poorly except for *gcc*. Stride predictors are best for loops and pointer traversing, and 8-bit values are rarely used in such cases.

This predictor uses num_entries 8-bit values (holding last value) and num_entries 8-bit values (assuming up to 8-bit strides) for a total of 1k of memory if num_entries is 512.

Last Value Predictor. This predictor does well for its small size. This type of predictor will only work if there are stretches where values repeat (such as spaces in a text file) or if the same value is repeatedly read from memory multiple times.

This predictor uses num_entries 8-bit values for a total of 512 bytes if num_entries is 512.

Last 8 Byte Value Predictors When operating on 8-bit values, you can hold 8 previous values in the same amount of memory that a 64-bit load would require. Thus you can hold 8 times as much history in the same amount of space.

The **Last 8 Oracle Predictor** has the best performance out of any tested, but of course it is not physically realizable. Two achievable selection methods are tried, **random** and **most-recently-correct**. Random was not optimal, though it performed better than the stride predictor. The "most recently correct" method of choosing from the last 8 loads did well except for the *gzip* case.

The random predictor uses num_entries*8 bytes to hold the table of last used values, for a total of 4096 bytes if 512 entries are used. The most recently correct predictor needs an additional 3bits per entry to hold the pointer to most recently correct, for a total of 4288 bytes.

String Predictors String predictors use the past history of characters sequences to predict the next byte.

First I use **last-string** where you look up the last value for that index in a lookup table to see what followed that character last time. This predictor takes num_entries last values plus a 256 byte table for a total of 768 bytes if 512 entries are used.

Next I look at **last2-string** which uses the last two characters in the history to see what the next one would predict. This is a 16bit value so it takes a 64kB table plus num_entries times two bytes for a total of 65kB. This predictor is good, although very large.

The **last3-hash-string** attempts to improve on last2-string without increasing the size much. It is the same as last2-string but an extra byte of history is used. In order to not make the tables bigger, the lookup is based on a hash where the third byte in the history is shifted left by 4 bits and xor'd on top of the other two bytes. This gives a small improvement over last2-string with an increase of only num_entries (to store the extra byte of history per index) of size.

A more complicated hash involving four bytes is used in **last4-hash-string**. Each of the four bytes is shifted left by 4 and then xor'd, with the top 4 bits being dropped. This increases performance on all but equake. The size is that of last3-hash-string with an increase of num_entries (again to store the extra byte of history per index).

Entry sizes (that the address of the load instruction is hashed into) of 256, 512, and 1024 were simulated. Unlike the 64-bit case, with 8-bit there is very little performance difference between the various history sizes. This is probably because there are so few 8-bit load locations there is not much aliasing even with a small working set. The results assume an entry size of 512 where applicable.

The results of the various predictors is shown in Figure 2. In addition to the 8-bit predictors, Figure 2 also has standard 64-bit load value predictor results for comparison. The 8-bit predictors obtain much better performance than the overall 64-bit predictor at a fraction of the size. One must remember that even at best only 30% of the loads are represented, and that's only with *gzip* which sadly does not predict well at all in the 8-bit case. The improvements in prediction are lost when you realize that the 8 bit loads are only a small fraction of total loads.

Figure 3 shows the results for *cc1* with bytes needed (roughly corresponding to needed on-chip real estate). The best part of the chart to be in is the upper left, which indicates the optimal size/performance ratio. Somewhat surprisingly the last-value predictor behaves best for this metric, although its prediction success rate is probably still not enough to justify the added complexity.

5 Conclusion and Future Work

More analysis can be done to determine if the 8-bit loads occur in performance critical code. If these reads are a bottleneck it might make sense to add an 8-bit load value predictor when a 64-bit predictor could not be justified. Especially as a very capable and fast 8-bit "last-value" predictor can be implemented with just 512 bytes of storage.

More advanced 8-bit predictors can also be imagined, especially when the input to the benchmarks is written-language. In that case a language-specific pre-built letter frequency table might be imagined that does better than dynamic methods.

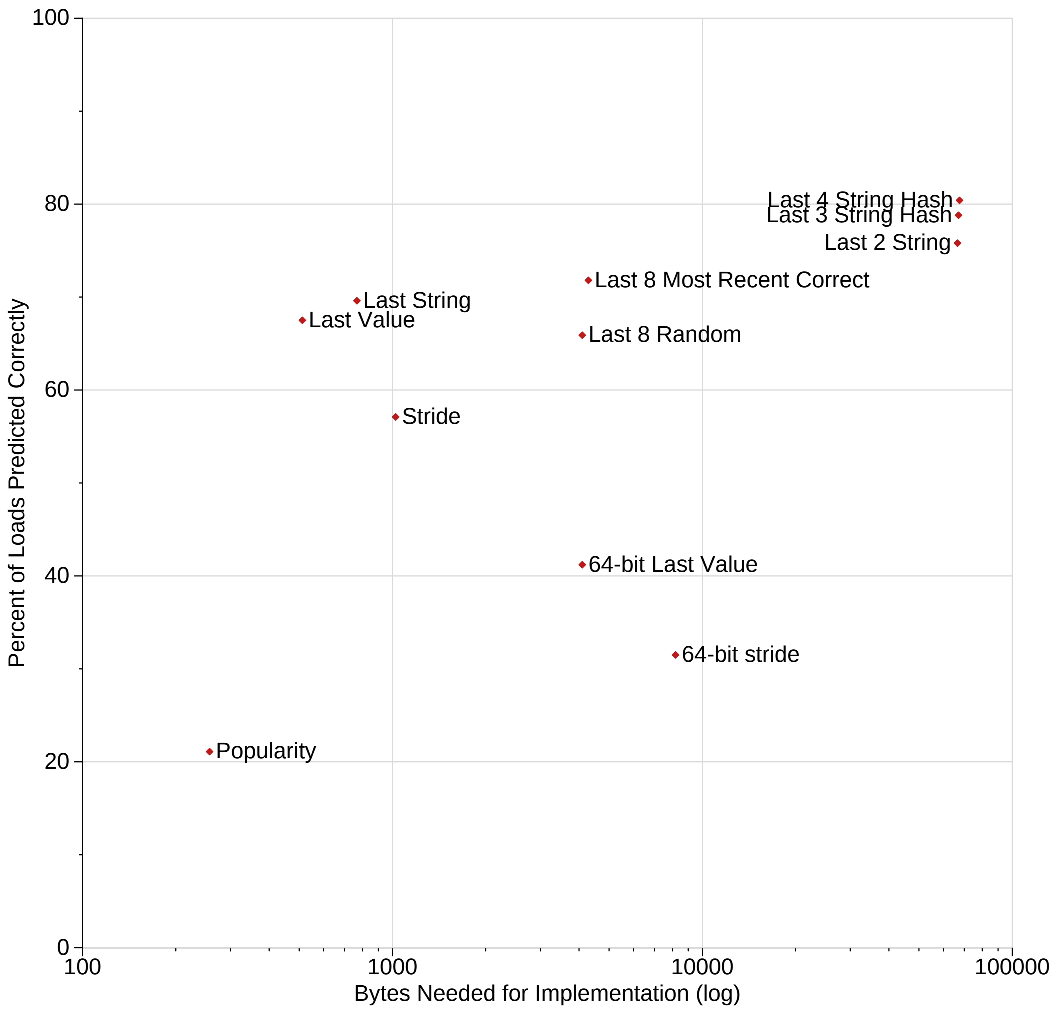

Figure 3: Size vs Performance Tradeoff cc1

A wider variety of benchmarks needs to be investigated, to determine if these results are indicative of a normal workload, or if they overstate the cause of 8-bit value prediction.

Other architecture should be investigated, such as the x86 architecture which is much more friendly to 8-bit operations.

While one can predict 8-bit loads with much better accuracy than 64-bit loads, in most cases the performance gain would go almost unnoticed because 8-bit loads do not occur frequently. Therefore the added complexity and die area needed for 8-bit value prediction is not worth the performance gain, at least on the Alpha architecture.

References

[1] Compaq Computer Corporation. *Alpha Architecture Handbook*, 1998.

[2] D. P. Hunter and E. B. Betts. Measured effects of adding byte and word instructions to the Alpha architecture. *Digital Technical Journal*, 8(4):89–106, 1996.

[3] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[4] Standard Performance Evaluation Corporation. SPEC CPU benchmark suite. `http://www.specbench.org/osg/cpu2000/`, 2000.