Load Redundancy Elimination on Executable Code*

Manel Fernández[1], Roger Espasa[1], and Saumya Debray[2]

[1] Computer Architecture Department, Universitat Politècnica de Catalunya
Barcelona, Spain
{mfernand,roger}@ac.upc.es
[2] Department of Computer Science, University of Arizona
Tucson AZ, USA
debray@cs.arizona.edu

Abstract. Optimizations performed at link time or directly applied to final program executables have received increased attention in recent years. This paper discuss the discovery and elimination of redundant load operations in the context of a link time optimizer, an optimization that we call *Load Redundancy Elimination (LRE)*. Our experiments show that between 50% and 75% of a program's memory references can be considered redundant because they are accessing memory locations that have been referenced less than 200–400 instructions away. We then present three profile-based LRE algorithms targeted at optimizing away these redundancies. Our results show that between 5% and 30% of the redundancy detected can indeed be eliminated, which translates into program speedups in the range of 3% to 8%. We also test our algorithm assuming different cache latencies, and show that, if latencies continue to grow, the load redundancy elimination will become more important.

1 Introduction

Optimizations performed at link time or directly applied to final program executables have received increased attention in recent years [13,3,11]. First, large programs tend to be compiled using separate compilation, that is, one or a few files at a time. Therefore, the compiler does not have the opportunity to optimize the full program as a whole, even performing sophisticated inter-procedural analysis. A second reason is the emergence of profile-directed compilation techniques [12,7]. However, the same problem of separate compilation plagues the use of profile feedback: large projects will be forced to re-build every file to take advantage of the profiling information. Link time optimizations are able to re-optimize the final binary using profile data without recompiling source code.

This paper presents an optimization to be applied in the context of link time optimizers. We discuss the discovery and elimination of load operations

* This work is being supported by the Spanish Ministry of Education under grants CYCIT TIC98-0511 and PN98 46057403-1. The research described in this paper has been developed using the resources of the CEPBA.

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 221–229, 2001.

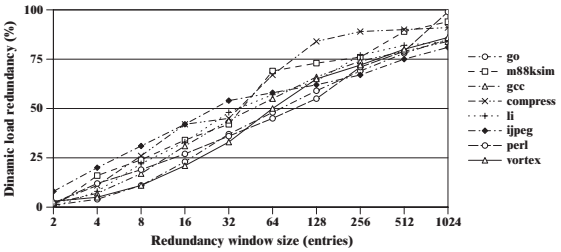

Fig. 1. Dynamic amount of load redundancy for the whole SPECint95 (Compaq/Alpha executables compiled with full optimizations). X-axis is logarithmic

that are redundant and can be safely removed in order to speed up a program, an optimization that we call *Load Redundancy Elimination (LRE)*. Unnecessary memory references appear in a binary due to a variety of reasons: a variable may not have been kept in a register by the compiler because it was a global, or maybe the compiler was unable to resolve aliasing adequately. We then present three profile-based LRE algorithms targeted at optimizing away these redundancies: a basic LRE algorithm for extended basic blocks, and two general algorithms that work over regions of arbitrary control flow complexity: one for removing fully redundant loads and the other for removing partially redundant loads.

2 Dynamic Amount of Load Redundancy

We start motivating our work by measuring a potential upper bound on how many loads could be removed from a program. Our goal is to measure how often a load is re-loading data that has already been loaded in the near past. Thus, we instrument the SPECint95 programs to catch all their memory references. Dynamic load redundancy is measured by recording the most recent n memory references into a *redundancy window*. This is a simple FIFO queue, where new references coming into it displace the oldest reference stored in the window. A dynamic instance of a load is then redundant if its effective address matches the address of any prior load or store that is still in the redundancy window.

The results of our measures are shown in Figure 1, for various redundancy window sizes. Clearly, a lot of redundancy exists even in these highly optimized binaries. As an example, for `m88ksim`, almost 75% of all load references were to memory locations that had been referenced by at least one of the most recent 256 memory instructions. In general, almost 50% of all loads are re-loading a data item that was read/written less than 100 memory instructions ago. Today's optimizing compilers are clearly able to deal with regions larger than this size, and should be expected to optimize all this redundancy away.

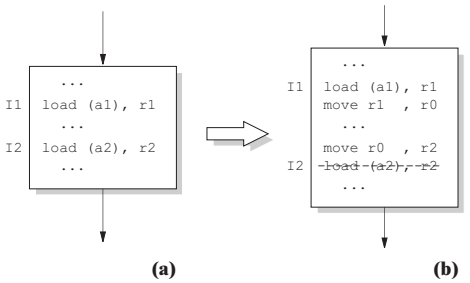

Fig. 2. Elimination of redundant load inside a machine code basic block

3 LRE on Executable Code

The simplest example of Load Redundancy Elimination (LRE) is shown in Figure 2. Suppose that an instruction I_1 loads a value into register r_1 from memory location pointed by a_1. This load is followed after some instructions by another instruction I_2 within the same basic block, which puts its value from location pointed by a_2 into register r_2. If it can be proved that both memory locations are the same, and this location is not modified between these two instructions, then I_2 is *redundant* in front of I_1[1]. Once a redundant load has been identified, we may try to eliminate it by *bypassing* the value from the first load to the redundant one, as shown in Figure 2b. This is accomplished by inserting a couple of move operations that use a new *available* register (r_0 in the example).

Although this is the most simple case of LRE, it already introduces the three fundamental problems that this optimization has to deal with. The first problem is to decide if both loads are really accessing the same memory location, and also that there is no store between them that *may* be in conflict with that location. There is an extensive work on *alias analysis* [1,9], but they are typically formulated in terms of source-level constructs that do not handle features encountered in executable programs [4]. The second problem is to find a register to bypass the source value to the redundant load. *Register liveness analysis* computes which registers are live at every point in the code [6,10]. Finally, the example shows that eliminating a load doesn't come without a cost: we have inserted "move" instructions in the code in the hope that (a) they can be removed by a copy propagator and (b) even if they are not, their cost will be lower than that of the original redundant load. In any case, a careful *cost-benefit* analysis is required.

Alias and register liveness analysis are well-known data-flow problems already described in the literature [9]. From now on, we assume that both of them have been computed before applying the LRE optimization. The more accurate are these analysis, the more opportunities appear for LRE.

[1] Note that the redundancy is also present if the instruction I_1 is a store operation.

4 Profile-Guided LRE

Information about the program execution behavior can be very useful in optimizing programs. Our proposal is to be aware of *profile information* to guide LRE. We next outline the algorithms used and present the cost-benefit equations that use the profile information to choose the candidates for removal.

4.1 Eliminating Close Redundancy

The results presented in Section 2 show that between 25% and 40% of all the redundancy detected can be captured using a redundancy window of just 16 entries. This indicates that the first source of redundancy that we should target our optimization at is located within small groups of basic blocks.

A natural extension of the example given in Figure 2 is to perform LRE on Extended Basic Blocks[2]. For every load in the EBB, we search bottom-up for other load or store that may be a source of redundancy, as shown in Figure 3a. But what if the hot path does not flow through BB2? A move instruction has been inserted in the critical path, although the bypassed value will be most often unused. There is no benefit in applying LRE and we might risk lowering performance. The lesson to learn is that it is not always beneficial to remove a redundant load, and it is necessary to apply LRE carefully. We need to be compute as precisely as possible the benefit (B) and cost (C) of applying the optimization for each particular load. The equations we use are as follows:

$$B = lat_{load} \times BB_2^{freq}$$
$$C = lat_{move} \times (BB_1^{freq} + BB_2^{freq})$$
$$LRE \Leftrightarrow C \leq B$$

The benefit includes the latency of the load being eliminated times the frequency of its basic block. The costs include the latencies of the *new* two "move" instructions weighted by the execution frequencies of their corresponding basic blocks. Note that the costs are pessimistic, as they always include *both* "move" instructions even though they might be later removed by a copy propagator.

4.2 Eliminating Distant Redundancy

Going back to Figure 1, there is still a lot of redundancy that can be caught if we can explore larger distances between instructions. To catch this redundancy, we need to apply LRE to regions of code that expand beyond an EBB.

The second algorithm we present is targeted at detecting *fully redundant loads*, that is, loads that are redundant with respect to *all* the control flow paths that reach them. For every load, we scan all potential paths looking for a source instruction that may render it. As shown in Figure 3b, if redundancy is found on all paths and all intervening stores do not alias with the load, the

[2] An EBB is a set of basic blocks with a single entry point but multiple exit points.

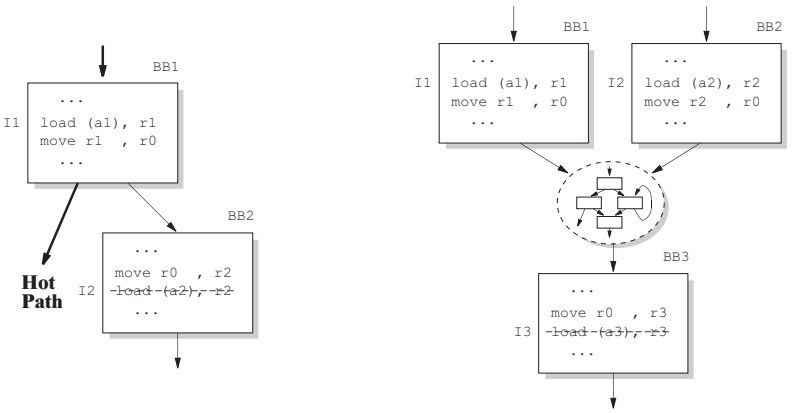

Fig. 3. Elimination of *fully redundant* loads: (a) within extended basic blocks, (b) for multi-path redundant loads. LRE should be applied coupled to a cost-benefit analysis

load becomes a candidate for removal. Then we apply the cost-benefit equations already described, although we have to extend the cost (C) to account for all the move instructions that must be inserted on each of the redundancy paths:

$$C = lat_{move} \times \left(BB_{red}^{freq} + \sum_{i=1}^{n} BB_{src_i}^{freq} \right)$$

If the benefit of removing the candidate out-weight the cost of adding the "move" instructions, the algorithm starts looking for an available register to bypass the value [5]. If no register is found, then the load can not be removed.

In the LRE algorithms discussed so far, the removal of a load is a safe transformation because there is always a static source of redundancy. However, a high percentage of dynamic redundancy comes from *partially redundant loads*, that is, loads that are redundant only on some control flow paths. Imagine that instruction I_2 in Figure 4 is an invariant inside the loop. The previous algorithm will fail to remove the load because it is not fully redundant: I_2 is redundant on the loop back-edge with I_1, but it is not on the entry point of the loop. This situation arises frequently, even without considering loops.

Partial LRE involves insertion of new instructions. As insertions are usually done on a different EBB, the inserted instructions become *speculative*. In general, it is safe to perform speculation for instructions that cannot cause exceptions, but this is not the case for speculative loads. In order to deal with safe insertions only, our implementation of partial LRE is restricted to global and stack references. We have followed the approach described by Horspool and Ho [8], that proposed a general profile driven PRE algorithm based upon edge profiles. The idea is to insert copies on less frequently executed paths in favor of more frequently executed paths. We have adapted their algorithm to (a) only consider redundant load operations, and (b) to deal with our cost-benefit analysis. Being n the

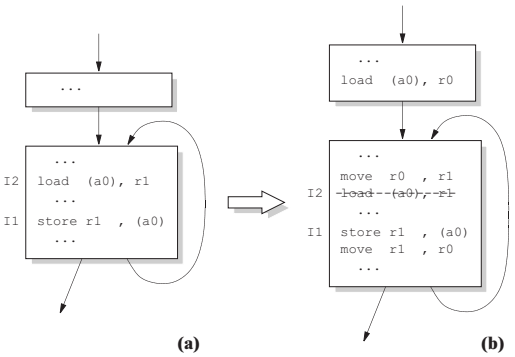

Fig. 4. Elimination of a *partially redundant load*. Removing the redundant load requires to insert instances in less-frequent paths, in order to make the load fully redundant

number of partial redundancies and m the number of load insertions needed, the cost of removing a load is then:

$$C_{bypass} = lat_{move} \times \left(BB_{red}^{freq} + \sum_{i=1}^{n} BB_{src_i}^{freq} \right)$$
$$C_{insert} = lat_{load} \times \sum_{i=1}^{m} EDG_i^{freq},\ C = C_{bypass} + C_{insert}$$

Cost involves not only bypassing the value, but inserting the new load operations that make the candidate become fully redundant. We use the same algorithm as before for obtaining available registers, but it has been extended to also look for register availability at the new load insertion points.

5 Performance Evaluation

We have implemented the proposed LRE approaches within the `alto` link-time optimizer [11,5]. The SPEC95 integer benchmarks were compiled with full optimizations using the vendor-supplied C compiler, on an AlphaServer equipped with an Alpha 21264 microprocessor. The programs were instrumented using Pixie and executed on the SPEC training inputs to obtain an execution frequency profile. Finally, these binaries and were processed by Alto with/without using different degrees of profile-guided LRE: LRE on EBB for catching close redundancy, and fully- and partial-LRE for catching distant redundancy.

We start evaluating the effectiveness of the three LRE algorithms under study by comparing the number of dynamic loads executed, for each benchmark. As it can be seen in Figure 5a, all programs show improvements around 5%, with some rather better cases such as `m88ksim` and `compress`. The results also show that working only on EBBs is not enough to catch the close-redundancy we presented in Section 2. Except for `perl` and `vortex`, LRE applied to EBBs yields a small reduction in dynamic loads. By contrast, fully-LRE improves the overall results

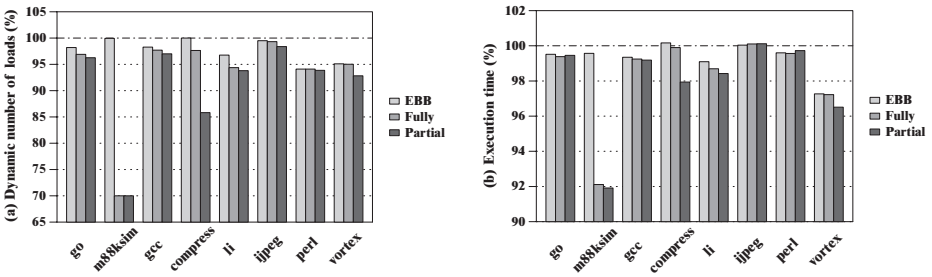

Fig. 5. Effect of different LRE degrees in (a) number of loads at run time, and (b) execution time. The baseline is optimized binaries by Alto without any LRE at all

for five programs and partial-LRE only yields extra improvements for `compress`. Additional number to better understand this results are presented in [5].

We are also interested in quantifying the percentage of reduction in execution time. We have decided to use the SimpleScalar toolset [2] that models a Compaq Alpha 21264 to get an accurate measure of the differences between the LRE algorithms. Results are presented in Figure 5b. Since loads are only a fraction of all instructions executed in a program, reduction in execution time is smaller than the corresponding reduction in number of dynamic loads. Thus, for example, the 30% reduction in dynamic loads in `m88ksim` only translates into an 8% reduction in execution time. However, the decrease in execution time shows that we have removed some loads that indeed were on the program's critical path.

Another interesting measure is to see what will happen in the future, as L1-cache latency continues to increase. Current CPUs are typically at a 2-cycle or 3-cycle latency and the trend is towards hyper-pipelining and, therefore, longer latencies. We re-simulated all the benchmarks changing the L1-cache latency to 3, 4 and 5 cycles. We also re-compiled every benchmark, since our cost/benefit analysis is dependent on the latency of the loads. Results can be seen in Figure 6. As expected, the longer the latency the worse the execution time of all programs. However, as latency increases, the importance of performing LRE also grows. For example, after applying partial-LRE in `vortex`, the execution time at a 5-cycle latency is *better* than the original execution time using a 3-cycle latency.

6 Summary and Future Directions

This paper has presented three algorithms to perform load redundancy elimination on executable files. We have shown that between 50% and 75% of all memory references in the SPECint95 programs can be considered "redundant"', since they access memory locations that had already been referenced by another load or store within a close dynamic distance. The first algorithm, LRE within extended basic blocks, is able to remove less than 5% of all loads, and yields speedups below 4% in execution time. The results indicate that an extended

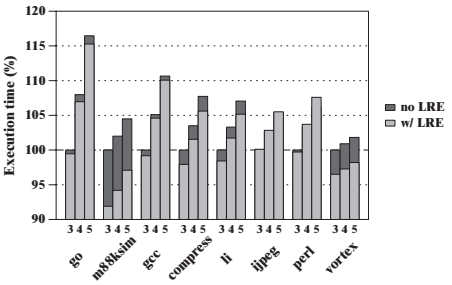

Fig. 6. Effect of load latency in execution time (from 3-cycle to 5-cycle hit latency)

basic block is too small to catch the redundancy measured in our experiments. The second algorithm, LRE for fully redundant loads on arbitrary control flow regions, yields an average increase of a 10% in loads to be removed. Despite this small increase, fully-LRE does detect some of the critical loads and thus increases speedups up to an 8%. The third algorithm, LRE for partially redundant loads, significantly increases the number of static loads removed (a 30% over the EBB algorithm). However, the algorithm only shows its strengths on `compress`, where an extra 12% of dynamic loads are removed over the fully-LRE algorithm. We believe that we need to explore better alias analysis algorithms to fully obtain the potential of the LRE optimization. We also test our algorithms assuming different cache latencies, and show that, if latencies continue to grow, LRE will become more important.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers principles, techniques, and tools.* Addison-Wesley, Reading, MA, 1986. 223
2. D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, CS Department, University of Wisconsin-Madison, 1997. 227
3. R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin. Spike: An optimizer for Alpha/NT executables. In USENIX, editor, *The USENIX Windows NT Workshop 1997*, pages 17–23, Seattle, Washington, August 11–13 1997. 221
4. S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–24, Orlando, Florida, January 19–21 1998. 223
5. M. Fernández, R. Espasa, and S. Debray. Load redundancy elimination on executable code. Technical Report UPC-DAC-2001-3, Computer Architecture Department, Universitat Politècnica de Catalunya-Barcelona, 2001. 225, 226, 227
6. D. W. Goodwin. Interprocedural dataflow analysis in an executable optimizer. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 122–133, Las Vegas, Nevada, June 15–18 1997. 223

7. R. Gupta, D. A. Berson, and J. Z. Fang. Path profile guided partial redundancy elimination using speculation. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 230–239, Chicago, May 14–16 1998. 221
8. R. N. Horspool and H. C. Ho. Partial redundancy elimination driven by a cost-benefit analysis. In *8th Israeli Conference on Computer System and Software Engineering*, pages 111–118, Herzliya, Israel, June 1997. 225
9. S. S. Muchnick. *Building an Optimizing Compiler*. Morgan Kaufman, 1997. 223
10. R. Muth. *Alto: A Platform for Object Code Modification*. PhD thesis, Department of Computer Science, University of Arizona, 1999. 223
11. R. Muth, S. Debray, S. Watterson, and K. de Bosschere. alto: A link-time optimizer for the DEC Alpha. Technical Report TR98-14, Department of Computer Science, University of Arizona, 1998. 221, 226
12. K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990. 221
13. A. Srivastava and D. W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, Dec. 1992. 221