Load Redundancy Elimination on Executable Code

Manel Fernández and Roger Espasa
Computer Architecture Department
Universitat Politècnica de Catalunya, Barcelona
{mfernand,roger}@ac.upc.es

Saumya Debray
Department of Computer Science
University of Arizona, Tucson AZ
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 this 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 [24, 9, 12, 21], due to two main reasons: 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. Thus, even if the compiler performs sophisticated inter-procedural analysis, the fact that it is only looking at a few files at a a time severely limits the usefulness of inter-procedural transformations. Furthermore, alias information and basic knowledge about variable allocation (for example, whether a variable is stored on the heap, stack or global area) is also lost when moving from one file (compilation unit) to the next. Vendors have tried to overcome this limitation by compiling separate files that contain intermediate representations rather than final object code [1]. Later, when linking, these "fake" intermediate object files are fully compiled and optimized together with the rest of the program units. The drawback of this approach is that it doesn't mix well with traditional Makefile-based software development environments. As a consequence, link time optimizations that are based solely on the final object representation have the attraction of being able to work on a full program basis and be fully integrated on a normal compile-build-test cycle.

A second reason for the recent interest in binary optimization has been the emergence of profile-directed feedback [22, 7, 10, 13]. As it has been shown in several studies [3, 4], the compiler can use to great advantage the profiling information. However, the same problem of separate compilation plagues the production use of profile feedback. If the profiling information has to be used when compiling, then a large project will be forced to re-build each and every file in order to take advantage of the profiling information. Furthermore, the profile-guided compilation needs to be specially coded into the Makefile environment. By contrast, it would be much better to be able to build the full application, instrument it to obtain profile data and then re-optimize the final binary without recompiling a single source file. This is the approach taken by Spike, for example [9] and is only possible if using binary optimization techniques.

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

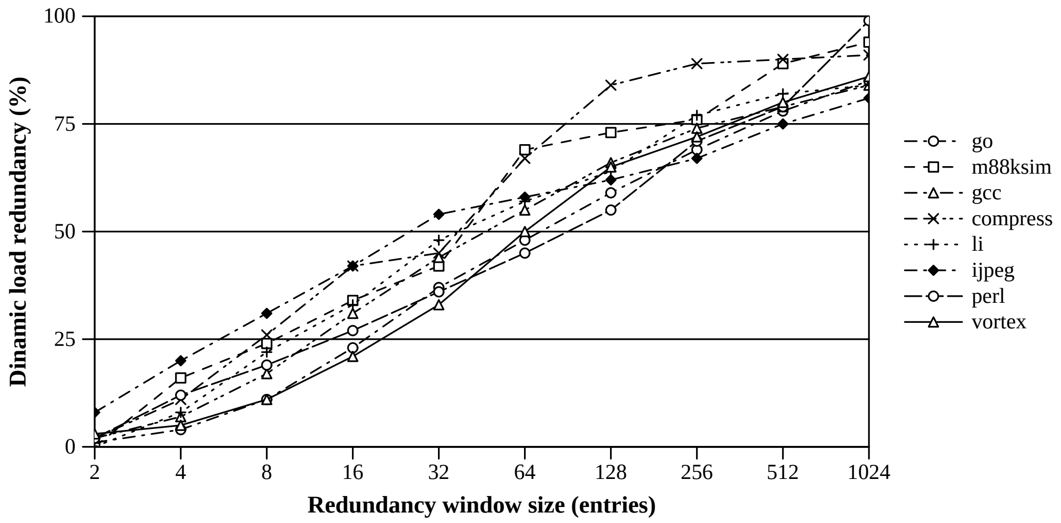

Figure 1: Dynamic amount of load redundancy for the whole SPECint95 (Compaq/Alpha executables compiled with full optimizations). X-axis is logarithmic.

aliasing adequately, or maybe there were not enough free registers available. We quantify these effects and 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 this 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.

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. Finally, we discuss what are the factors that prevent us from eliminating all the redundancy detected and perform experiments with different numbers of machine registers to show their impact on the amount of redundancy eliminated.

2 Dynamic amount of load redundancy

Before presenting our algorithms for removing redundant loads, we motivate 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 and also to quantify the typical distance (in memory instructions) between re-loads of the same data item.

To achieve this goal, 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 window is a simple FIFO queue, where new references coming into it displace the oldest memory 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 simulations are shown in Fig 1, where we present data for all the SPECint95 programs for various redundancy window sizes. As an example, the graph shows that, 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. That is, almost 75% of all load references were to memory locations that had been loaded recently and that, therefore, should be candidates to be optimized away by the compiler.

Clearly, a lot of redundancy exists even in these highly optimized binaries. As we can see, almost 50% of all loads are re-loading a data item that was read/written less than 100 memory instructions ago. Considering that in these streams, around 1/3 of instructions are memory references, it means that 50% of all loads are re-loading data that was already accessed

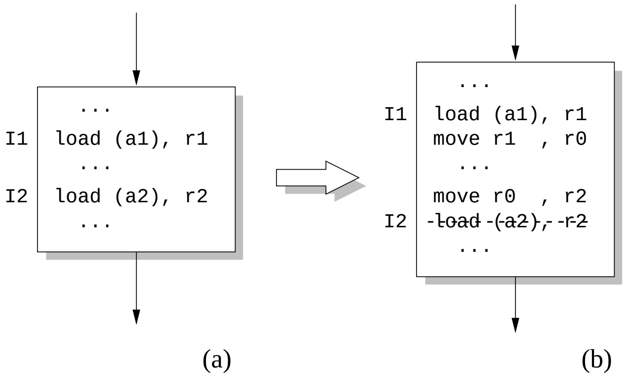

Figure 2: Elimination of redundant load inside a machine code basic block.

or stored less than 300 instructions ago. Furthermore, today's optimizing compilers are clearly able to deal with regions larger than this size and, thus, should be expected to optimize all this redundancy away.

3 LRE on executable code

The simplest example of Load Redundancy Elimination (LRE) is shown in Figure 2a. Suppose that an instruction I_1 loads a value into register r_1 from memory location pointed by a_1. Furthermore, this load is followed after some instructions by another instruction I_2 within the same basic bloc, which puts its value from location pointed by a_2 into register r_2. If it can be proved that both locations pointed by a_1 and a_2 are the same, and this location is not modified between these two instructions, then I_2 is *redundant* in front of I_1. Note that the redundancy is also present if the instruction I_1 is a store operation.

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. Bypassing the value is accomplished by inserting a couple of move operations that use a new *available* register (r_0 in the example; this register may or may not be the same as r_1 or r_2, depending on register lifetimes). The expectation is that, after running the LRE optimization, a copy propagator is also run to eliminate as many register moves introduced by LRE as possible.

Although this is but the most simple case of LRE, it already introduces the three fundamental problems that this optimization has to deal with: alias analysis, register liveness analysis and cost-benefit analysis. We now discuss each problem in more detail.

Alias analysis. The first problem is to decide if both loads (or any store/load pair) are really accessing the same memory location or not, and also to prove that there is no other store between them that *may* be in conflict with the memory location accessed by the redundant load. In our example in Figure 2, this amounts to proving that registers a_1 and a_2 do indeed point to the same memory location. Although there is an extensive work on pointer alias analysis [2, 25], such analyses are typically formulated in terms of source-level constructs, and do not handle features such as pointer arithmetic and out-of-bound array references, while these are precisely the features encountered in executable programs [11].

Register liveness analysis. The second problem is to find an available register to bypass the value from the redundancy source to the redundant instruction. This is not an easy task, due to the limited number of machine registers and also due to the constraints imposed by the calling convention. Register liveness analysis [12, 20] is a technique that computes which registers are live or dead at every point in the code. On executable code, control flow reconstruction is key to improve the accuracy of the register liveness analyzer, otherwise the analysis becomes too conservative to be useful.

Cost-Benefit Analysis Finally, the simple example presented shows that eliminating the load doesn't come without a cost: in fact, we have inserted two "move" instructions in the optimized 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. Of course, the cost can be reduced if we can use register r_1 as the bypassing register. This, however, will require that r_1 is not overwritten between instructions I_1 and I_2. In any case, LRE on executable code requires of a careful cost-benefit analysis, as Section 4 will discuss. If the cost-benefit analysis is too optimistic, performance degradation may appear.

Alias and register liveness analysis are well-known data-flow problems already described in the literature [19]. From now on, we assume that both of them have been computed before applying the LRE optimization. Then, the more accurate are these analysis, the more opportunities appear for LRE. A significant number of opportunities may be lost if the alias analyzer is not able to decide whether two references are in conflict. Also, discovered LRE opportunities are lost if the register liveness analyzer is not able to find an available register to effectively bypass the redundant value.

LRE on Intermediate Code vs. Executable Code

It is interesting to point out the difference between performing LRE on intermediate code (as done by compilers) and on executable code: our proposed optimization must deal with the limitations of a small register file. By contrast, most compilers will perform LRE by taking a new "pseudo-virtual" register from the infinite virtual register pool to bypass the value between the two loads. Interestingly, it may happen that at a later stage, when the register allocator runs, the compiler re-inserts the redundancy due to lack of machine registers (a problem that LRE on executable code does not suffer from).

Working on executable code also has the added difficulty that alias analysis becomes even more difficult, since no information on the original program variables is available. On the other side, the one advantage of working on executable code is that estimating the costs and benefits of inserting and removing instructions is rather more accurate than when working on intermediate level code.

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. Profile information consists of a frequency for each basic block and a probability for each branch in the program. We next outline the algorithms used and present the cost-benefit equations that use the basic block frequency information gathered in a profile run 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 either within the same basic block or in groups of small basic blocks.

LRE on Extended Basic Blocs

We have already seen the easiest form of LRE in the example given in Section 3, where we look for redundancy within a basic block. A natural extension of this scheme is to perform LRE on Extended Basic Blocks[1].

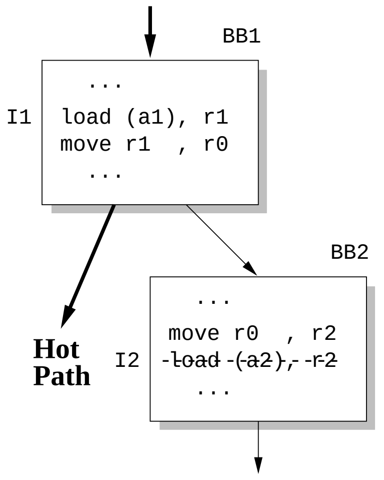

Figure 3: Elimination of redundant load within extended basic blocks; LRE must be only applied coupled to a cost-benefit analysis.

The implementation of LRE on an extended basic block is straight-forward using a bottom-up search to the EBB root. For every load in the EBB, we search bottom-up for other loads or stores that may be a source of redundancy, as shown in Figure 3. If we can prove, again, that addresses a_1 and a_2 point to the same memory location and that no intervening store has modified said location[2], then it is safe to remove I_2 and bypass the value from r_1 to r_2.

However, as already discussed, introducing a "move" instruction increases the cost of executing basic block BB1. What if, as in Figure 3, the hot path does not flow through BB2? In this case, a move instruction has been inserted in the critical path, although the bypassed value will be most often discarded. There is no benefit in applying LRE to I_2 in this example 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 cost and benefit of applying the optimization for each particular load.

The equations we use to compute the benefits (B) and costs (C) of removing a certain load are as follows:

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

[2]If an intervening store can be proved to write to the same location, then it becomes itself the source of redundancy and the algorithm works the same way. The problem is when an intervening store has an unknown address. In such case, bypassing is not safe and redundancy elimination can not proceed.

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

As it can be seen, the benefit computation includes the latency of the load being eliminated times the frequency of its basic block. On the other hand, the costs include the latencies of the *two* "move" instructions introduced weighted by the execution frequencies of the two basic blocks where they appear. Note that the costs are pessimistic, as they always include *both* "move" instructions, even though they might be later removed by the copy propagation phase. Our algorithm checks first whether either the source of redundancy register or the final destination register (r_1 and r_2 in our example, respectively) can be chosen to bypass the redundant value, avoiding some of the "move" insertions and keeping the cost C more realistic.

4.2 Eliminating Distant Redundancy

The LRE approach described in the previous Section was targeted at close redundancy. However, going back to Figure 1 in Section 2 there is still a lot of redundancy that can be caught if we can explore larger distances between instructions. Of course, in order to catch this distant redundancy, we need to apply LRE to regions of code that expand beyond an EBB and which, therefore, contain complicated control flow structures.

The major difference with the previous Section is that when working on a candidate load to be removed, we need to examine *all* the possible control flow paths that may reach the load in order to decide whether the load is truly redundant or not. Two situations may arise:

Full redundancy The candidate load is indeed redundant with respect to all the control flow paths that reach it.

Partial redundancy The candidate load is redundant on *some* paths, but not all, that reach it.

Fully Redundant Loads

The second algorithm we will evaluate is targeted at detecting fully redundant loads. For every candidate load, we scan all potential paths that may reach it looking for a source instruction that may render the load redundant. If redundancy is found on *all* paths and, again, all intervening stores have known addresses that do not alias with the load, then the load becomes a candidate for removal. An example is shown in Figure 4.

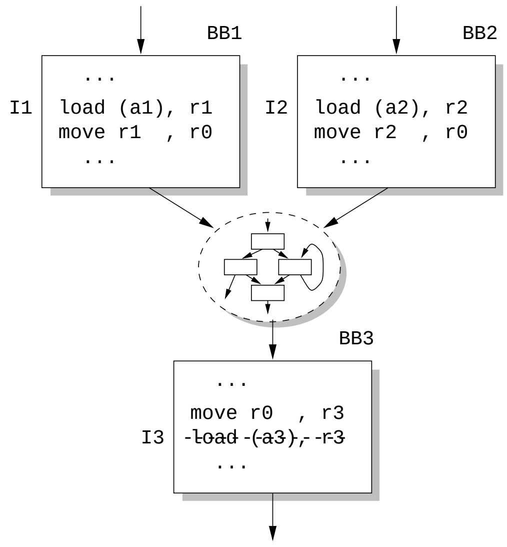

Figure 4: Elimination of a multi-path redundant load. That is, the redundant load is *fully redundant* for every path that reaches to it.

Once a load is a candidate we apply to it the cost-benefit equations already described. However, we have to extend the cost (C) equation to account for all the move instructions that must be inserted on each of the redundancy paths, as shown below:

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

As before, this cost is a pessimistic upper bound, since an appropriate choice of registers to bypass the value may avoid some of the "move" instructions.

Again, the load will maintain its "candidate" status only if the benefits of removing it out-weight the costs of adding the "move" instructions. If this is the case, then our algorithm starts looking for available registers to bypass the value. First of all, we start checking whether the destination register (r_3 in our example) can be used to bypass the value from *all source paths*. That is, if r_3 is not live on the paths that lead from BB1 and BB2 to BB3. Choosing r_3 as the bypass register results in avoiding the move instruction in BB3. If the destination register can not be used, the basic blocks that are the source of the redundancy (BB1 and BB2 in Figure 4) are sorted according to their execution frequency. Now, we start with the most executed basic block and check whether we can use the source redundancy register (r_1 in our example) to bypass the value on *all the other* paths. That is, if r_1 happens to be free on the path(s) that leads from BB2 to BB3, then by choosing r_1 as the bypass register we save a move instruction in BB1. We iterate for every source basic block until a free register is found (that is, if r_1 is not available on the path from BB2 to BB3, then we will try to use as bypass register r_2, again to save a "move" instruction in BB2). If after this process no register is found, then we simply look for *any* register that might be available on all paths simultaneously –note that this would match the pessimistic cost analysis outlined above. If still no register is found, then the redundant load can not be removed.

Clearly, this is an expensive optimization that should be applied only once. Furthermore, as we will see in Section 5, it may happen that after all the expensive alias analysis, we can not remove a load simply because we have no register available.

Partially Redundant Loads

So far, all the LRE algorithms discussed were only able to remove loads that are fully redundant. The upside of the previous algorithms is that the removal a of particular load is always a safe transformation, because there is always a static source of redundancy for the load removed. However, a high percentage of dynamic redundancy comes from loads that are redundant only on some control flow paths. We call these loads *partially redundant loads*.

We can see an example of a partially redundant load in Figure 5. Imagine that instruction I_2 is an invariant instructions inside the loop (suppose that neither a_0 nor location pointed by a_0 are changing in the loop). If we apply the algorithm of the previous Section, it will fail to remove the load because it is not fully redundant: instruction I_2 is redundant on the loop back-edge with the store I_1, but it is not on the entry point of the loop. A similar situation arises frequently even without considering loops.

The partial redundant load must be removed by inserting a copy of the instruction on the control flow paths where it is not available, thus making the load fully redundant. In the example, this means that a copy of the redundant load must be inserted in the loop preheader.

Partial LRE involves insertion of new instructions. As this insertions are usually done on a different extended basic block, the inserted instruction becomes *speculative*. In general, it is safe to perform speculation for instructions that cannot cause exceptions, but this is not the case for speculative loads. When speculating loads, the optimizer must be careful not to introduce side-effects into a program that did not

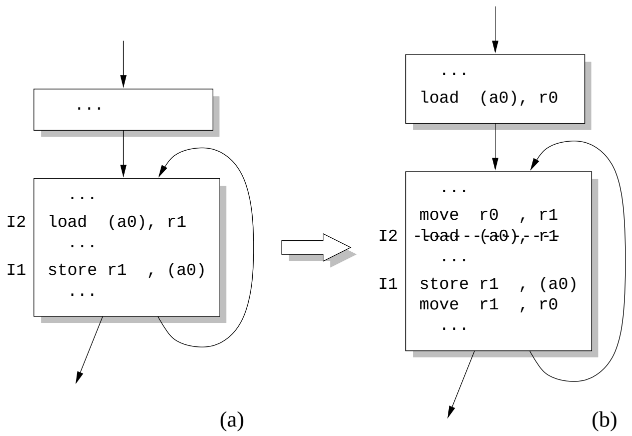

Figure 5: Elimination of a *partially redundant load*. Removing the redundant load requires to insert instances in less-frequent paths, in order to make the load fully redundant.

exhibit them before. In order to deal with safe load insertions only, our implementation of partial LRE is restricted to global and stack references. Access to global variables is always safe, because they are always live in the whole execution of a program. Local variables located in the stack are considered only if neither the stack pointer changes within the function (except the entry and exit points) nor other callee function store values in the stack out of its stack frame.

For the implementation of our partial LRE optimization, we have followed the approach described by Horspool and Ho [14]. They proposed a general profile driven PRE algorithm based upon edge profiles. The main idea is to insert copies on less frequently executed paths in favor of more frequently executed paths, as shown if Figure 5. We have adapted their algorithm to (a) only consider redundant load operations and (b) to deal with our cost-benefit analysis.

The cost of removing a redundant load on partial LRE is then as follows:

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

Being n the number of partial redundancies and m the number of load insertions needed. Cost involves not only bypassing the value, but inserting the new load operations that make the candidate load become fully redundant. To obtain a register to bypass the value, we use the same algorithm already described for the fully redundant loads case. The algorithm has to be extended, however, to also look for register availability at the new insertion points where we insert a load to make our candidate load fully redundant.

4.3 A Combined LRE algorithm

We have implemented the proposed LRE approaches within the `alto` link-time optimizer [21]. In order to maximize the optimization opportunities we use the following scheme.

First we run an inlining pass using the already existing Alto framework. The reason is that our LRE algorithms are not inter-procedural and, yet, the calling conventions do introduce an important amount of redundancy at compile time. Thus, since we are working on the final binary, it is a good opportunity to remove this calling convention overhead.

Then we apply the close-distance LRE, that is, the LRE on extended basic blocks. Indeed, we re-run LRE on extended basic blocks several times during optimization, since computing the data-flow equations for an EBB and performing the load redundancy searches is relatively cost-effective.

Next, we run one of the long-distance algorithms, whether fully-redundant LRE or partially redundant LRE (Section 5 presents results for both cases)[3]. These are expensive optimizations and, therefore, we perform them only once, after performing also a space- and time-intensive data flow analysis. To maximize the benefits of the LRE optimization, we apply the analysis phase of LRE to every function and keep a list of all the candidates for removal sorted by net benefit. Then, the most executed loads ("hot" loads) are the ones that are tried to remove first, when the chance of finding available registers inside the function is higher.

After running the long-distance algorithms, we re-run the close-distance LRE (LRE on EBB), to catch any new opportunities opened up by the previous LRE phase. Since this is a cheap optimization, it doesn't contribute much to the overall running time of the algorithm.

Finally, to keep the running time of the LRE algorithm under control we use a parameter ϕ in the interval $(0, 1]$. Basic blocks are sorted according to their relative execution frequency and the LRE optimization is only applied to loads within basic blocks that have en execution frequency larger than $1 - \phi$. Thus, setting $\phi = 1.0$ will cause the algorithm to apply LRE to every single load in the code, but it will also cause a large increase in optimization time. The idea is to apply LRE only to the "hot loads" in the program. For all our experiments we have used a ϕ value of 0.75.

[3]Note that the partial-LRE algorithm, of course, subsumes the behavior of the fully-redundant LRE algorithm.

Benchmark	Input
`099.go`	50 10
`124.m88ksim`	dcrand.lit (train input)
`126.gcc`	gcc.i
`129.compress`	50000 e 2231
`130.li`	boyer.lsp (train input)
`132.ijpeg`	specmun (test input)
`134.perl`	primes.in (ref input, 51 lines)
`147.vortex`	persons.250 (train input)

Table 1: SPEC95 integer benchmark suite and their inputs.

5 Performance evaluation

5.1 Experimental framework

We have implemented the proposed LRE approaches within the `alto` link-time optimizer [21]. The benchmarks used were the eight programs in the SPEC95 integer benchmark suite. The eight programs and the inputs used for our experiments are listed in table 1. Note that we have used variants of the official SPEC input sets to keep simulation time down to a manageable value.

All programs were compiled with full optimizations, using the vendor-supplied C compiler on an AlphaServer 8400 equipped with an Alpha 21264 microprocessor. For processing by Alto, the compiler was also invoked with linker options to retain information and to produce statically linked executables[4]

The executables were instrumented using Pixie and executed on the SPEC training inputs to obtain an execution frequency profile. Finally, these binaries and their profiles were processed by Alto using different degrees of profile-guided LRE, for obtaining different measures about the effect of this optimization.

5.2 LRE Algorithm Effectiveness

We start evaluating the effectiveness of the three LRE algorithms under study (LRE on EBB for catching

[4]We used statically linked executables because Alto relies on the presence of relocation information for its control flow analysis. The Tru64 Unix linker refuses to retain information for non-statically linked executables.

Benchmark	EBB			Fully			Partial		
	Pot	**Ben**	**Rem**	**Pot**	**Ben**	**Rem**	**Pot**	**Ben**	**Rem**
`099.go`	542	436	430	584	470	454	754	532	489
`124.m88ksim`	572	481	462	705	607	585	858	664	615
`126.gcc`	2962	1886	1833	2996	1917	1854	3088	1960	1884
`129.compress`	246	191	185	253	201	195	287	215	209
`130.li`	917	389	382	1238	671	640	2027	958	913
`132.ijpeg`	369	309	303	386	324	316	491	378	363
`134.perl`	1096	894	872	1109	904	878	1152	928	891
`147.vortex`	1966	1759	1718	1970	1763	1722	2236	1918	1841

Table 2: Static LRE numbers for the SPEC95 integer benchmark suite. *Pot*: potential number of opportunities, *Ben*: opportunities that are beneficial using the cost/benefit analysis, *Rem*: loads actually removed (there was a register to bypass the redundant value).

close redundancy and fully-LRE and partial-LRE for catching distant redundancy) by comparing the number of dynamic loads executed against the program baseline. As a baseline, this section and all further sections use the fully-optimized benchmarks after being run through Alto with inlining turned on. That is, we are comparing the effectiveness of the algorithms implemented against what could be considered state-of-the-art optimized machine code.

Figure 6 presents the reduction in number of dynamic loads for each benchmark with respect to the original baseline. As it can be seen, all programs do show improvements typically around 5%, with some rather better cases such as `m88ksim` and `compress`. Comparing our results to Table 2 in Lo *et al* [17], we can see that we achieve less benefits. However, we believe the reason for that is that we are working on final machine code while they were measuring reduction in dynamic loads *before* code generation and register allocation. Factoring this in, our results are very much in line with those presented in [17], yet we do not have the advantage of high quality alias analysis as they do.

The results show also that working only on EBBs is not enough to catch the close-redundancy we presented in Section 2. Except maybe for `perl` and `vortex`, LRE applied to EBBs yields a small reduction in dynamic loads. By contrast, fully-LRE improves the overall results for five programs and partial-LRE only yields extra improvements for `compress`.

In order to better understand this results, it is worth looking at the internals of our algorithm. Table 2 breaks down the opportunities for LRE for each of the three algorithms under evaluation. For each algorithm, three numbers are presented: Pot, Ben and Rem. Column "Pot" indicates the number of loads considered by the algorithm as candidates for removal.

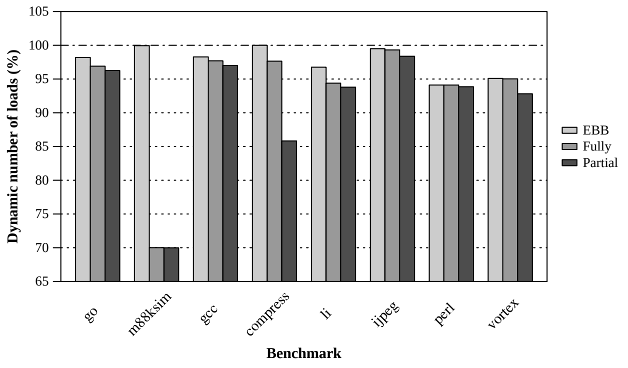

Figure 6: Effect of different LRE degrees in number of loads at run time. The baseline is optimized binaries without any LRE at all.

Note that this number is computed *after* memory disambiguation has determined that there are no conflicting aliases that prevent LRE. The second column, "Ben", are the number of candidates remaining *after* applying our cost benefit analysis based on BB frequencies. For example, the large drop in *gcc* indicates that the costs of removing those loads out-weight the benefits. Finally, column "Rem" indicates the number of static loads actually removed. The differences between column "Ben" and "Rem" are attributed to lack of registers to bypass the value from the source to the redundant load.

The lack of registers to bypass a value is *directly* responsible for only a minority of the "lost opportunities". The largest drop corresponds to column "Ben", where our cost-benefit analysis discards many opportunities for load removal. Since our cost equations are conservative, they always assume that a move instruction will be inserted, regardless of whether registers

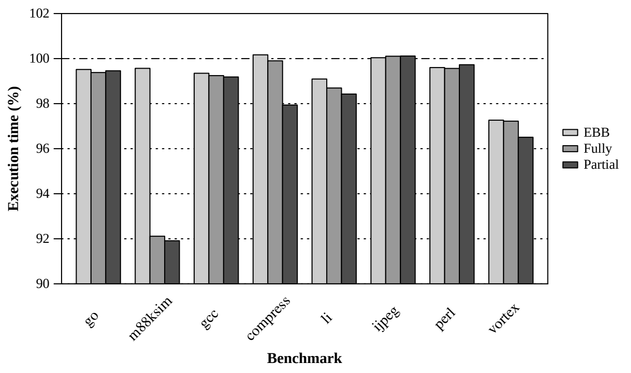

Figure 7: Effect of different LRE degrees in execution time. The baseline is optimized binaries without any LRE at all.

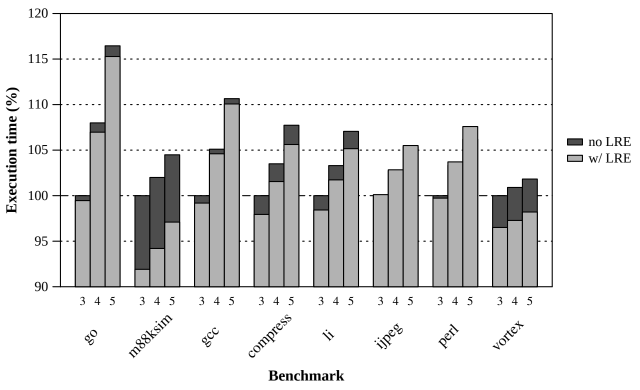

Figure 8: Effect of load latency (from 3-cycle to 5-cycle hit latency)

are really available or not. We are currently investigating other cost formulations that integrate the cost computation with the register availability computation.

5.3 Speedup using LRE

Counting the number of dynamic loads removed is certainly of interest to understand the effectiveness of each algorithm. However, the final measure of interest is whether execution time is reduced or not. To this end, we have decided to use the SimpleScalar 3.0 simulation toolset [6] to get an accurate measure of the differences between applying or not the LRE algorithms [5]. Our out-of-order simulator models a Compaq Alpha 21264 configuration [15]. Detailed simulation parameters are described in Table 3.

The results of our simulations are presented in Figure 7. Of course, 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 overall execution time. Overall, however, the decrease in execution time shows that we have hit (removed) some loads that indeed were on the program's critical path and, therefore, contributed heavily to overall execution time.

5.4 Effects of load latency

Another interesting measure to gauge the importance of the LRE transformation 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. In this section we re-simulated all the benchmarks changing the L1-cache latency from its value to 3 cycles up to 4 and 5 cycles. Furthermore, for each experiment we also re-compiled every benchmark, since all our cost/benefit analysis are dependent on the latency of the loads. At larger latencies, it is more likely that the cost/benefit equations tend to favor the substitution of a load by one or more "move" instructions.

The results can be seen in Figure 8. The first thing to observe, as expected, is that the longer the latency the worse the execution time of all programs. However, as latency increases, the importance of performing LRE also grows. Consider, for example, the case of `vortex`. After applying partial-LRE, the execution time at a 5-cycle latency is *better* than the original execution time using a 3 cycle latency.

5.5 Effects of register file size

Another interesting experiment that our simulation environment allows is to vary the number of logical registers available to the compiler. While we clearly do not expect in the near future that microprocessor vendors increase the number of registers in their instruction sets, it is a very interesting experiment from the compiler's point of view. By increasing the logical register set, we can get insight into how many opportunities for load removal could be re-gained by better register allocation or, in our case in executable code, by better register re-shuffling.

We modified both Alto and the SimpleScalar toolset to be able to optimize and simulate Alpha bi-

[5]Variations in real execution time due to interactions with other processes and OS variations were too large to allow effective measurement of speedups

Parameter	**Value**
Fetch width	4 instructions.
L1 I-cache	64Kb, 4-way set-associative, 64-byte line, LRU, 1-cycle hit latency.
Branch predictor	Combined predictor of a 2-level adaptative predictor with 8K 2-bit counters, 16-bit global history, and a bimodal predictor of 8K entries with 2-bit counters. 3-cycle extra mispredict latency.
Decode/Commit width	4 instructions.
Issue mechanism	4-way out-of-order issue, wrong-path issue included.
Instruction window size	128 entries.
Functional units	4 simple integer, 1 integer mult/div, 4 simple FP, 1 mult/div.
Load/store queue (LSQ)	64 entries, stores in LSQ may bypass values to later loads.
L1 D-cache	64Kb, 2-way set-associative, 64-byte line, LRU, 3-cycle hit latency.
L2 unified I/D-cache	1Mb, 4-way set-associative, 64-byte line, LRU, 12-cycle hit latency. 16 bytes to main memory, 16 cycles first chunk, 2 cycles interchunk.
Instruction TLB	1Mb, 4-way set-associative, 4Kb page, LRU, 30-cycle miss penalty.
Data TLB	1Mb, 4-way set-associative, 4Kb page, LRU, 30-cycle miss penalty.

Table 3: Simulation parameters for an Alpha 21264 configuration.

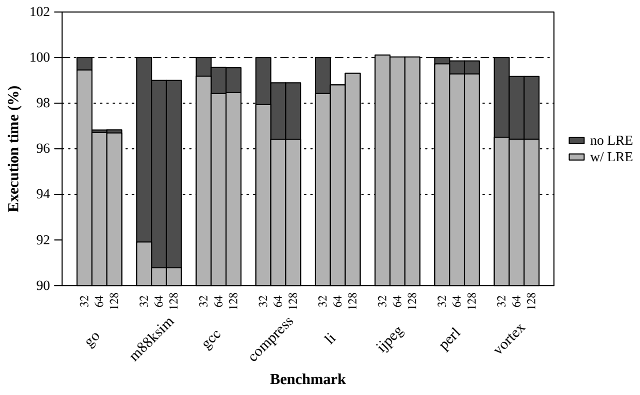

Figure 9: Effect of register file size. Number of logical integer registers varies from 32 to 128 registers.

naries containing 64 and 128 integer registers (the default being 32). Of course, we re-optimized all the binaries with the new integer register file sizes and re-simulated them on SimpleScalar. The results can be seen in Figure 9.

First of all, having more registers improves execution time whether we apply LRE or not, as the black bars in Figure 9 show. When we turn to the results for the LRE algorithm, we see that only the go program really takes advantage of the extra registers. This suggests that register pressure may be less of a problem than originally anticipated and that, probably, we should focus future work on improving the number of candidates that our LRE algorithm targets (i.e., the "Pot" column in Table 2).

6 Related work

While a number of systems have been described for optimization of executable code [24, 9, 12, 21], to the best of our knowledge, any elimination of redundant loads carried out by these systems is limited to fairly simple load removal.

Load redundancy elimination can be seen as a particular case of *Partial Redundancy Elimination* (PRE), where the expressions to be considered for removing are only load operations. PRE [16, 8] is a well known scalar optimization that subsumes various *ad hoc* code motion optimizations (as common subexpression elimination and loop invariant code motion) by attempting to remove redundancies that occur only on some control flow paths. Horspool and Ho [14] described a new formulation of PRE based on a cost-benefit of the flowgraph, by using edge profiles (our current implementation of partial LRE optimization is based in their equations). Gupta, Berson and Fang [13] extended this profile driven PRE algorithm by using path profiles.

Register Promotion allows scalar values to be allocated to registers for regions or their lifetime, where the compiler can prove that there are no aliases for the value. Promotion carries out elimination of both redundant loads and dead stores. Cooper and Lu [18] examined promotion over loop regions. Their results indicate that the main benefit of promotion comes from removing store operations. Lo *et al* use a variant of SSA-PRE to remove unnecessary loads and stores over any program region. However, they not consider the effect of spilling because it simulate with an infinite symbolic register set before register allocation. Both previous works only counted the improvement compared to the total number of load and store instructions. Postiff, Greene and Mudge [23] presented recently a register promotion algorithm at link-time, although their algorithm does not use any PRE approach at all. They also present numbers for long register files, but the gain in this case seems to come from several *ad hoc* techniques for promoting global and constant values into a dedicated subset of the long register file.

Bodík, Gupta and Soffa [5] developed a load redundancy analysis, and design a method for evaluating its precision. However, their paper is only focused in the analysis, they do not perform any elimination of redundant loads at all.

7 Summary and future directions

This paper has presented three algorithms to perform load redundancy elimination on executable files. We have instrumented the SPECint95 programs and shown that between 50% and 75% of all memory references can be considered "redundant"', since they access memory locations that had already been referenced by another load or store instruction within a close dynamic distance (less than 256 references away in our dynamic window experiments).

We have presented an algorithm targeted at catching the close-distance redundancy by looking at redundancy within an extended basic block. This first algorithm is able to remove less than 5% of all loads and, therefore, yields speedups below 4% in execution time. The results seem to indicate that an extended basic block is too small of a region to catch the redundancy measured in our redundancy experiments. The second algorithm presented, LRE for fully redundant loads found on arbitrary control flow regions, yields an average increase of a 10% in candidate loads considered 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 discussed, LRE for partially redundant loads, significantly increases the number of static loads removed (a 30% over the EBB algorithm). However, the extra cost of the algorithm only shows its strengths on `compress`, where an extra 12% of dynamic loads are removed over the fully-LRE algortithm.

In the third part of the paper we have explored the effects of our algorithms on future architectures. Our simulations have shown that, as the L1 cache latency increases, the importance of the LRE optimization also increases. Furthermore, we have explored how well our optimization would do under a scenario with more machine registers available (such as in the IA64 architecture). Results indicate that lack of register does not seem to be the gating factor to achieve better speedups. On the contrary, we believe that we need to explore better alias analysis algorithms to fully obtain the potential of the LRE optimization.

Acknowledgments

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.

References

[1] *MIPSpro Compiling and Performance Tunning Guide.* Number 007-2360-008. Silicon Graphics, Inc., Mountain View, CA, 1999.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers principles, techniques, and tools.* Addison-Wesley, Reading, MA, 1986.

[3] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 46–57, Paris, France, Dec, 2–4 1992. ACM Press.

[4] T. Ball, P. Mataga, and M. Sagiv. Edge profiling versus path profiling: The showdown. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 46–57, Orlando, Florida, January 19–21 1998.

[5] R. Bodik, R. Gupta, and M. L. Soffa. Load-reuse analysis: Design and evaluation. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 64–76, Atlanta, Georgia, May 1–4 1999.

[6] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, Computer Sciences Department, University of Wisconsin-Madison, 1997.

[7] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W.-M. W. Hwu. Profile-guided automatic inline expansion for C programs. *Software Practice and Experience*, 22(5):349–369, May 1992.

[8] F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on SSA from. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 273–286, Las Vegas, Nevada, June 15–18 1997.

[9] 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.

[10] R. Cohn and P. G. Lowney. Hot cold optimization of large Windows/NT applications. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 80–89, Paris, France, Dec, 2–4 1992. ACM Press.

[11] S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–24, Orlando, Florida, January 19–21 1998.

[12] 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.

[13] 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. IEEE Computer Society Press.

[14] 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.

[15] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[16] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.

[17] R. Lo, F. Chow, R. Keneddy, S.-M. Liu, and P. Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 26–37, Montreal, Canada, June 17–19 1998.

[18] J. Lu and K. Cooper. Register promotion in C programs. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 308–319, Las Vegas, Nevada, June 15–18 1997.

[19] S. S. Muchnick. *Building an Optimizing Compiler*. Morgan Kaufman, 1997.

[20] R. Muth. *Alto: A Platform for Object Code Modification*. PhD thesis, Department of Computer Science, University of Arizona, 1999.

[21] 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.

[22] 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.

[23] M. Postiff, D. Greene, and T. Mudge. The need for large register files in integer codes. Technical Report CSE-TR-434-00, EECS/CSE University of Michigan, 2000.

[24] 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.

[25] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, La Jolla, California, June 18–21 1995.