Partial Redundancy Elimination Considering Register Spills

Yasunobu Sumikawa
yas@cs.is.noda.tus.ac.jp
Tokyo University of Science
2641 Yamazaki, Noda, Chiba, Japan, 278–8510
Advisor: Prof. Munehiro Takimoto
ACM student member number: 8292527
category: graduate

1. Problem and Motivation

Partial redundancy elimination (PRE) is a code optimization technique that makes each partially redundant expression totally redundant by inserting the same expression at some program points and removing it from the original point [3, 7, 10]. The insertion and removal process moves the expression to some nodes of control flow graph closer to the start node, which tends to increase the register pressure, potentially leading to register spills [4, 9]. The resulting the extra execution cost of the objective code may exceed the improvement caused by the redundancy elimination. To decrease the impact of the register spills, costly redundant expressions should be removed preferentially and not-costly ones should be removed only if the removal does not lead to register spills.

We propose an effective PRE that selectively applies PRE to costly expressions including loop-invariant expressions to suppress register pressure. We call the approach Selective PRE (SPRE). SPRE is based on global value numbering (GVN) [1] in order to capture most *second-order effects*, are captured by iteratively applying PRE after copy propagation in the traditional PRE. As GVN globally assigns the same value number to equivalent expressions, SPRE recognize redundancies based on the value numbers. Furthermore, SPRE moves array references so that references to the same array are continuously executed by aggregating them, instead of removing them, and SPRE inserts new references to nodes where the register pressure is minimized without changing the referencing order. The reference aggregation increases the spatial locality on the memory, so that the references are likely to make a cache hit happen, because an array reference causes not only referred address but also its vicinity to be loaded to the cache memory.

Consider Figure 1(a). SPRE moves loop-invariant expression `z1+1` at Node 3 to Node 2 as shown in Figure 1(b). On the other hand, expression `z1+1` at Node 4 is not removed, although it is redundant. Furthermore, SPRE moves array reference `a[i]` at Node 2 immediate before `b[i]` at Node 1.

2. Background and Related Work

There are some techniques that remove some redundancies while suppressing register spill [2, 6]. These techniques do not consider the second-order effects and the good results are not reported. To the best of our knowledge, SPRE is the first practical technique.

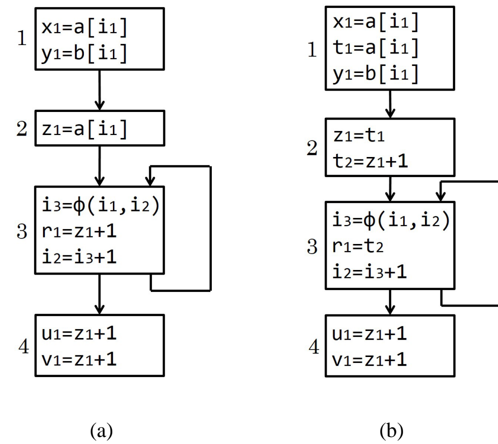

Figure 1. Code motion considering register spill. (a) Original code. (b) Result of applying our technique.

3. Approach and Uniqueness

SPRE is based on *effective demand-drive PRE* (EDDPRE) [8] that performs GVN and then applies query propagation to each expression one-by-one in *topological sort order*, so that the expressions whose queries have the same answer of $true$ are removed. The query propagation backwardly propagates a query from each expression e to check whether there are some expressions with the same value number as e. If all propagated queries have $true$ as answers, e is considered to be redundant. Otherwise, EDDPRE checks whether additional expressions can be safely inserted to make all queries $true$. If that is possible, e becomes redundant after insertions; otherwise, e is not redundant. In addition, once the query returns $true$ at a node, the answer of sub-query $isSelf$ is also given simultaneously, which checks whether the currently checked expression is the source expression. If $isSelf$ is $true$, the propagation path includes a cycle and the source expression is found to be loop-invariant.

SPRE selectively applies EDDPRE to each expression, depending on its cost. Furthermore, in SPRE, once a query about an array reference src is propagated to another array reference ar referring to the same array beyond some references to different arrays, src

Table 1. System parameters of machine

CPU	Intel Core i5-2320 3.00GHz
Integer registers	8
Floating registers	8
OS	Ubuntu 12.04LTS

Table 2. Execution time of objective code.

programs	A.PRE*2	B.EDDPRE	C.SPRE	(A-C)/A	(B-C)/B
equake	74.1 sec	72.0 sec	66.8 sec	9.9%	7.2%
art	32.9 sec	33.3 sec	33.3 sec	-1.2%	0.0%
mcf	33.6 sec	33.5 sec	33.4 sec	0.6%	0.3%
bzip2	74.4 sec	78.3 sec	73.3 sec	1.5%	6.4%
gzip	102 sec	103 sec	101 sec	1.0%	1.9%
gap	53 sec	47.7 sec	44.4 sec	16.2%	6.9%
ammp	120 sec	119 sec	118 sec	1.7%	0.8%
vpr	72 sec	70.4 sec	66.5 sec	7.6%	5.5%
parser	105 sec	107 sec	102 sec	2.9%	4.7%
twolf	113 sec	107 sec	107 sec	5.3%	0.0%

is moved at nodes where the total live-ranges of the indexes is the shortest around ar.

Uniqueness

Our approach has three characteristics that make it unique. First, SPRE removes only costly redundant expressions based on value numbers. Second, captures many second-order effects. Third, SPRE moves array references so that references to the same array appear continuously and decrease register pressure.

4. Results and Contributions

We have implemented our technique as a low-level intermediate representation converter in the COINS compiler [5]. To evaluate the benefits derived from our technique, we compared SPRE to EDDPRE and PRE*2 which applies PRE twice and applies copy propagation between them.

We applied the optimizations to three programs (equake, art, and ammp) of CFP2000, and seven programs (mcf, bzip2, gzip, gap, vpr, parser, and twolf) of CINT2000 in the SPEC benchmarks on x86 machine whose parameters are shown in Table 1.

Table 2 shows the execution times of PRE*2, EDDPRE, and SPRE. SPRE achieved better execution efficiency than EDDPRE and PRE*2 for most programs. Comparing SPRE with PRE*2, in art, the execution time was increased because register spill was not so serious problem in the program, while the removal of redundancies remarkably improves the efficiency. These results suggest that our technique is effective for many programs.

References

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 1–11, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: 10.1145/73560.73561. URL `http://doi.acm.org/10.1145/73560.73561`.

[2] G. Barany and A. Krall. Optimal and heuristic global code motion for minimal spilling. In *Proceedings of the 22nd international conference on Compiler Construction*, CC'13, pages 21–40, Berlin, Heidelberg, 2013. Springer-Verlag.

[3] R. Bodik, R. Gupta, and M. L. Soffa. Complete removal of redundant expressions. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 1–14, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. doi: 10.1145/277650.277653. URL `http://doi.acm.org/10.1145/277650.277653`.

[4] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, SIGPLAN '82, pages 98–105, New York, NY, USA, 1982. ACM. ISBN 0-89791-074-5. doi: 10.1145/800230.806984. URL `http://doi.acm.org/10.1145/800230.806984`.

[5] COINS. http://coins-compiler.sourceforge.jp/.

[6] R. Gupta and R. Bodik. Register pressure sensitive redundancy elimination. In *Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, CC '99, pages 107–121, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-65717-7. URL `http://dl.acm.org/citation.cfm?id=647475.727623`.

[7] J. Knoop, O. Ruthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 224–234, New York, NY, USA, 1992. ACM. ISBN 0-89791-475-9. doi: 10.1145/143095.143136. URL `http://doi.acm.org/10.1145/143095.143136`.

[8] Y. Sumikawa and M. Takimoto. Effective demand-driven partial redundancy elimination. *Information Processing Society of Japan Transactions on Programming*, 6(2):33–44, aug 2013.

[9] C. Wimmer and M. Franz. Linear scan register allocation on ssa form. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 170–179, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-635-9. doi: 10.1145/1772954.1772979. URL `http://doi.acm.org/10.1145/1772954.1772979`.

[10] H. Zhou, W. Chen, and F. Chow. An ssa-based algorithm for optimal speculative code motion under an execution profile. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 98–108, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993510. URL `http://doi.acm.org/10.1145/1993498.1993510`.