Research of Register Pressure Aware Loop Unrolling Optimizations for Compiler

Xuehua Liu[1,2], Liping Ding[1,3] , Yanfeng Li[1,2] , Guangxuan Chen[1,4] , Jin Du[5]

[1]*Laboratory of Parallel Software and Computational Science, Institute of Software, Chinese Academy of Sciences, Beijing, China*
[2]*University of Chinese Academy of Sciences, Beijing, China*
[3]*Digital Forensics Lab, Institute of Software Application Technology, Guangzhou & Chinese Academy of Sciences, Guangzhou, China*
[4]*Zhejiang Police College, Hangzhou, China*
[5]*Yunnan Police College, Kunming, China*

Abstract. Register pressure problem has been a known problem for compiler because of the mismatch between the infinite number of pseudo registers and the finite number of hard registers. Too heavy register pressure may results in register spilling and then leads to performance degradation. There are a lot of optimizations, especially loop optimizations suffer from register spilling in compiler. In order to fight register pressure and therefore improve the effectiveness of compiler, this research takes the register pressure into account to improve loop unrolling optimization during the transformation process. In addition, a register pressure aware transformation is able to reduce the performance overhead of some fine-grained randomization transformations which can be used to defend against ROP attacks. Experiments showed a peak improvement of about 3.6% and an average improvement of about 1% for SPEC CPU 2006 benchmarks and a peak improvement of about 3% and an average improvement of about 1% for the LINPACK benchmark.

1 Introduction

Register pressure is the number of hard registers needed to store values in the pseudo registers at given program point [1] during the compilation process. Register pressure problem has been a known problem for compilers because of the mismatch between the infinite number of pseudo registers and finite hard registers. Too heavy register pressure may result in register spilling and then performance degradation to the target program [2]. There are a lot of transformations are able to increase register pressure. They are transformations who result in instruction movement like scalar replacement, loop invariant motion, Common sub-expression elimination and so on, and transformations that result in code redundancy, like inliner, loop unrolling, modulo scheduling [3] and so on. Meanwhile, some fine-grained randomizations[4,5], implemented during the compilation process, such as Atomic Instruction substitution, instruction reordering, register reassignment [6], random NOP insertion and so on, are adopted to defend ROP attacks. However, they are very likely to increase register pressure because they use instruction movement and code redundancy technologies a lot. As is known to all, loop optimizations are of crucial importance to source code optimization, especially for scientific programs. The effectiveness of loop optimizations is considered as one of important criteria to judge a compiler. And among all kinds of loop optimizations, loop unrolling is one of the fundamental transformations, because it is not only performed individually at least twice during the compilation process, it is also an important part of optimizations like vectorization, module scheduling and so on. Hence, it is crucial to improve the performance of loop unrolling transformation. Generally, loop unrolling is an approach known as the space-time tradeoff, who attempts to optimize a program's execution speed at the expense of its binary size. The goal of loop unrolling is to increase a program's speed by reducing or eliminating instructions that control the loop, such as pointer arithmetic and "end of loop" tests on each iteration, reducing branch penalties, as well as hiding latencies including the delay in reading data from memory [7]. To eliminate this computational overhead, loops can be re-written as a repeated sequence of similar independent statements. Since there is no effective method to control register pressure during unrolling process, loop unrolling transformation suffer from register spilling from time to time.

Effort has been paid to fight register pressure at some specific transformations, such as register pressure sensitive redundancy elimination [8], register pressure guided unroll-and-jam, a framework takes register pressure into account [9], register pressure aware inlining [10, 11], register pressure sensitive Rematerialization [12] and Prematerialization [13], register pressure aware register allocater [1,14], register pressure aware scheduling [3,15] and so on. However, among dozens of transformations in compiler, only a few of them have taken register pressure into account, and there is no general way to fight register pressure for loop optimizations. As a consequence, a lot of optimizations, especially loop optimizations suffer from register spilling in compiler. In this paper, the authors gave a try to fight against register pressure for loop unrolling optimization. And this approach is possible to benefit even more optimizations mentioned above.

2 Design of register pressure aware loop unrolling

In loop unrolling optimization, the unroll factor, the times of the loop body to be copied, determines the register pressure of the loop body after unrolling. At machine-dependent level intermediate representation stage in GCC, loop unroll factor is figured out with heuristic approach as follows:

• Initialize unroll factor as the quotient of the maximum number of instructions over the sum of instructions of the loop body, the maximum value of instructions is 200 by default and can be set manually.

• Force the unroll factor to be the power of 2 to satisfy better alignments, since it is unable to cope with overflows in computation of number iterations.

• Adjust the unroll factor to no bigger than the max factor, which is 8 by default and can be set manually.

However, this method is too rough to handle register pressure problem, because the unroll factor is determined manually without taking hard register resource into account. As a result, on the one side, it cannot guarantee that register pressure is under control in all cases, on the other hand, in some cases, it may be too conservative to reach their full potential. Actually, each instruction need different amount of registers, for instance, instruction DIV asks for 4 registers while instruction NOP asks for nothing. Hence, the instructions should not be treated equally. In sum, determining the unroll factor merely by the number of instructions is not universally applicable.

In order to address the issue mentioned above, the authors propose a register pressure aware approach to calculate loop unroll factor during the loop unrolling transformation at the machine dependent level intermediate representation stage. The authors achieve this goal as follows:

2.1 The definition of loop invariable, loop induction variable and loop variable

Since variables' attributes are missing at the machine dependent level intermediate representation stage, the authors reclassify the variables contained in registers of a loop body and divide them into three categories according to their impacts on the register pressure. They are loop invariable, loop induction variable and loop variable.

• Loop invariable

The value of loop invariable is constant no matter how many times the loop body is copied, such as the base address of array, the pointer, the constant and so on. This kind of variables takes only one register for each no matter how many times the loop body is copied and it is unlikely to increase the register pressure.

• Loop induction variable

The value of loop induction variable is increased or decreased by a fixed step or it is linear function of other induction variables in each iteration of a loop, such as the loop iterators. This kind of variables takes only one register no matter how many times the loop body is copied and it is unlikely to increase the register pressure.

• Loop variable

Loop variables are other variables beyond loop invariables and loop induction variables. Their values usually change without obvious rule in each iteration of a loop. They may take as many registers as the times the loop body is copied at worst. And they are most likely to increase the register pressure.

2.2 The authors' approach

The authors take advantage of some existing methods and extend them to the machine dependent level intermediate representation stage to analyze the variables in registers and identify their variable types, and then figure out a reasonable unroll factor.

2.2.1 Find out the number of loop invariable

According to classical data flow analysis infrastructure [16], a data flow problem can be formalized as a 4-tuples (D, V, $\wedge$, F).

• D is the direction of the data flow, which is either FORWARDS or BACKWARDS.

• (V, $\wedge$) is a semilattice includes a domain of values V and a meet operator $\wedge$.

• F is a family of transfer functions from V to V. this family must include functions suitable for the boundary conditions, which are constant transfer functions for the special nodes ENTRY and EXIT in any flow graph.

In order to find out all the loop invariables, the authors take advantages of existing live variable problem [16] and do the data flow analysis inside a loop. The 4-tuples (D, V, $\wedge$, F) for this problem can be described as follows:

D: backwards

(V, $\wedge$): (active variables, $\cup$)

$$\text{F: IN}[\mathit{EXIT}]=\emptyset \tag{1}$$

$\text{IN}[B]= \mathit{use}_B \cup (\text{OUT}[B]- \mathit{def}_B)$

$\text{OUT}[B]= \cup_{S \text{ a successor of } B} \text{IN}[S]$

• def_B as the set of variables defined (i.e., definitely assigned values) in B prior to any use of that variable in B.

• use_B as the set of variables whose values may be used in B prior to any definition of the variable.

• IN[B] as the set of live variables of loop L at the point of the start of basic block B.

• OUT[B] as the set of live variables of loop L at the point of the end of basic block B.

The authors do the data flow analysis according to the formalized description inside the loop body as follows:

• Suppose loop L consists of basic blocks B_1, B_2, B_3, B_n, *EXIT*, *EXIT* is a fake ending block of loop L without instructions, hence IN[*EXIT*]=∅ .

• Walk the basic block chain of Loop body forward and figure out all the def and use sets for each block.

• Walk the basic block chain of Loop body backward and figure out all the IN and OUT sets for each block based on the results of (1) and (2).

• IN[B_1] consists of all the active variables of loop L. The size of IN[B_1] is the number of hard registers needed for all the loop invariables.

2.2.2 Find out the number of loop induction variable

There are two kinds of induction variables: basic induction variable and general induction variable. Variable i is a basic induction variable only if the assignments to i in the loop body are of the form i=i±c (c is a constant). Variable j is a general induction variable only if the assignment to j in the loop body is of the form $j=a_j \times i+ b_j$ (i is a basic induction variable and a_j and b_j are loop invariant expressions). The authors take advantage of existing affine induction variables analysis method at the machine dependent intermediate stage to figure out the number of iterations of a loop by walking the use-def chains. The number of loop induction variable is the size of the union of basic induction variable and general induction variable in the loop.

2.2.3 Find out the number of loop variable

Loop variables are difficult to identify, for there is no one-size-fits-all rule to describe it. Other than find out all the loop variables directly, the author deduce the amount of loop variables based on previous results. The authors take advantage of an existing loop register pressure estimation algorithm in GCC to find out the amount of hard registers needed for pseudo registers in loop body before unrolling. Loop variable number is result of the amount of hard registers minus loop invariable number and loop induction variable number.

2.2.4 Find out the number of available registers for loop variable

Since each loop invariable and loop induction variable take only one register no matter how many times the loop body is copied, and each loop variable take as many registers as the times the loop body is copied, loop variable are most likely to increase register pressure. Loop variable should be taken carefully during loop unrolling. The number of available registers for loop variables means all the available hard registers except those that saved for loop invariable and loop induction variable. The size of this register set and the number of loop variable are the determining factors for calculating the loop unroll factor.

2.2.5 Calculate unroll factor

According to all the previous results, unroll factor can be calculated as follows:

• Initialize unroll factor

For most CPU infrastructures, hard registers are divided into several subclasses, such as general register, SSE register, MMX register and so on. For each kind of hard registers, unroll factor is initialized as the quotient of the number of available registers over the number of loop variable, and then take the minimum.

• Force unroll factor to be the power of 2 to satisfy better alignments, since it is unable to cope with overflows in computation of number iterations.

• Adjust unroll factor to no bigger than the max factor given by user.

Other than the original heuristic approach, on the one side the authors make the best of the hard registers, on the other side the authors prevent spilling from happening in the first place. Hence, the authors' approach is of better performance than the original heuristic approach.

3 Experimental evaluation

The authors implemented their design on GCC5.3. All the experiments were performed on a 2.67GHz Intel Core i7 CPU and 4GiB of memory, running Ubuntu16.04 with Linux kernel version 4.4.0 as the operating system. The authors evaluated the performance of their design using the CPU-intensive SPEC CPU2006 benchmark suite and Linpack benchmark.

SPEC CPU2006 test suite is a worst-case, CPU-bound performance test which is wildly used in all kinds of performance testing, especially compiler' performance. The authors compiled all becnchmarks of this test suite with O2 optimization level and loop unrolling enabled, set the max unroll factor to 8, took the ref input set of SPEC CPU2006, run each version three times and averaged the results. Table 1 and Figure 1 show a compare result of performance of benchmarks with the original approach and new approach. The bigger the value is, the better the performance is. Overall, with the new approach, there are obvious improvements for 410.bwaves and 437.leslie3d, and slight improvements for 401.bzip2, 403.gcc, 429.mcf, 471.cmnetpp, 473.astar, 444.namd, 453.povray, 454.calculix and 482.sphinx3 compared with the original algorithm. A peak improvement of about 3.6% reported across all the benchmarks was found for 437.leslie3d and the average performance was increased by about 1% for the overall SPEC 2006 test-suits. However, there is performance degradation for 462.libquantum because of the inaccuracy of the authors' new approach.

Table 1. SPEC 2006 cases score.

SPEC 2006 cases	original method	new method	improvement (%)
401.bzip2	19	19.1	1%
403.gcc	21.1	21.2	-
429.mcf	17.6	18	2.30%

471.cmnetpp	12.9	13.1	1.60%
473.astar	14.2	14.6	2.80%
410.bwaves	25.7	26.5	3%
437.leslie3d	16.7	17.3	3.60%
444.namd	18.8	19.3	2.70%
453.povray	27.6	27.7	-
454.calculix	11.5	11.6	1%
482.sphinx3	24	24.5	2%

LINPACK is a software library for performing numerical linear algebra, which is wildly used to test the performances of loop optimizations. The authors compiled all tests using the –O2 optimization level with loop unrolling enabled, set the max factor to be 8, set test array size setting from 1K to 10K, run each version three times and averaged the results. The results show that a peak improvement of about 3% with 2K array size and an average improvement of about 1% to the performance of LINPACK, which are almost consistent with SPEC 2006 tests. And the test result is more stable, since the benchmarks of LINPACK are much simple than SPEC2006's.

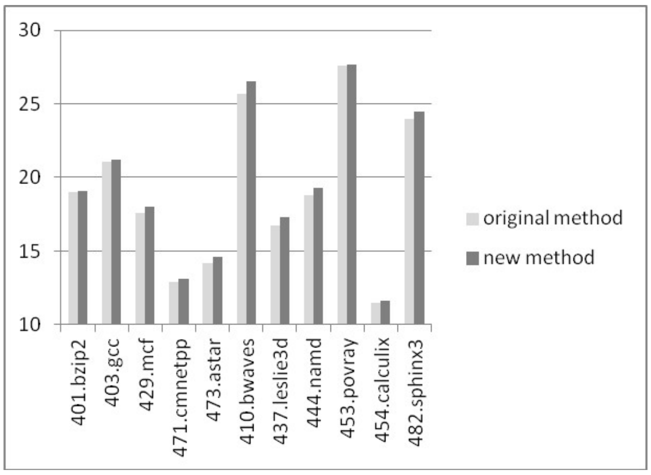

Figure 1. Contrast between original method and new method for SPEC 2006.

4 Conclusion

In this paper, the authors focus on improving the effectiveness of loop unrolling optimization at the machine dependent intermediate stage of compiler by taking register pressure into account. Firstly, the authors reclassify the variables in registers of a loop body and divide them into three categories according to their impact on the register pressure. Secondly, in order to find out the type of the variables contained in registers of a specific loop, the authors adopted several classical data flow analysis and induction variable analysis technologies. Then, in order to make fair use of pseudo registers, the authors propose a new approach to calculate unroll factor. In order to verify the effectiveness of this method, the authors implemented their design on GCC5.3 and evaluated the performance of GCC using the SPEC CPU2006 test suite and Linpack benchmark. Experiment results showed a peak improvement of about 3.6% and an average improvement of about 1% for SPEC CPU 2006 benchmarks and a peak improvement of about 3% and an average improvement of about 1% for the Linpack benchmark.

However, since some of the meta data are inaccurate, the authors' approach not always works. In very rare instances, the inaccuracy may result in performance regression instead. In the future, the authors are going to increase the accuracy by taking advantage of the loop versions or profile guided optimization technologies. With the real time profiling data, the authors are able to adjust their approach to make an even better choice. In addition, the authors' approach not only works for loop optimization but also works for swing modulo scheduling. Besides, a register pressure aware transformation is able to reduce the performance overhead of fine-grained randomization during the compilation process and therefore to increase the usability of this kind of method to defend ROP attacks.

Acknowledgement

This work has been supported by the National Key Research and Development Program of China under grant 2016QY01W0200, Science and Technology Planning Project of Guangdong Province, China (2017B050506002), Collaborative Innovation Center for Economics crime investigation and prevention technology, Jiangxi Province (No.JXJZXTCX-007, No.JXJZXTCX-009), Science and Technology Planning Project of Guangzhou Municipality, China (201802020015), Support Scheme of Guangzhou for Leading Talents in Innovation and Entrepreneurship (No. 2016008).

References

1. Vladimir N. Makarov, Fighting register pressure in GCC, Proceedings of the GCC Developers'Summit, 2004, pp.85-104, form https://gcc.gnu.org/wiki/.
2. Guohui Li, Yonghua Hu, Yaqiong Qiu, Wenti Huang, Investigation on the Optimization for Storage Space in Register-Spilling. CollaborateCom, 2016, pp.627-633, Berlin Heidelberg, Germany: Springer-Verlag.
3. Josep Llosa, Eduard Ayguadé, Antonio González, Mateo Valero, Jason Eckhardt, Lifetime-Sensitive Modulo Scheduling in a Production Environment, IEEE Transactions on Computers, 50 (3): 234-249, 2001.
4. Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, Michael Franz, Profile-guided automated software diversity, Code Generation and Optimization, 2013, Piscataway, NJ: IEEE Press.
5. Mark Murphy, Per Larsen, Stefan Brunthaler, Michael Franz, Software Profiling Options and Their Effects on Security Based Diversification, Proceedings of the First ACM Workshop on Moving Target Defense, pp.87-96, New York, NY: ACM Press, 2014.
6. Vasilis Pappas, Michalis Polychronakis, Angelos D.

Keromytis, Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. IEEE Symposium on Security and Privacy, pp.601-615, Piscataway, NJ: IEEE Press, 2012.

7. Petersen, W.P., Arbenz, P., Introduction to Parallel Computing. Oxford, England: Oxford University Press, 2004.
8. Gupta R., Bodík R., Register Pressure Sensitive Redundancy Elimination. In: Jähnichen S. (eds) Compiler Construction. CC 1999. Lecture Notes in Computer Science, vol.1575, pp.107-121, Berlin Heidelberg, Germany: Springer-Verlag, 1999.
9. Min Zhao, Bruce R. Childers, Mary Lou Soffa, Model-Based Framework: An Approach for Profit-Driven Optimization, Code Generation and Optimization, 2005, pp.317-327, Piscataway, NJ: IEEE Press.
10. Zhao P., Amaral J.N., To Inline or Not to Inline? Enhanced Inlining Decisions. In: Rauchwerger L. (eds), Languages and Compilers for Parallel Computing. LCPC, 2003, vol.2958, Lecture Notes in Computer Science, pp.405-419, Berlin Heidelberg, Germany: Springer-Verlag, 2004.
11. Dhruva R. Chakrabarti, Shin-Ming Liu, Inline Analysis: Beyond Selection Heuristics, Code Generation and Optimization, 2006, pp.221-232, Piscataway, NJ: IEEE Press.
12. Preston Briggs, Keith D. Cooper, Linda Torczon , Rematerialization. Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation, volume 27 (7), pp.311-321, New York, NY: ACM Press, 1992.
13. Ivan D. Baev, Richard E. Hank, David H. Gross, Prematerialization: reducing register pressure for free, Proceedings of the 15th international conference on Parallel architectures and compilation techniques, pp.285-294, New York, NY: ACM Press, 2006.
14. Ivan D. Baev, Techniques for Region-Based Register Allocation, Code Generation and Optimization, 2009 pp.147-156, Piscataway, NJ: IEEE Press.
15. Rami Beidas, Wai Sum Mong, Jianwen Zhu, Register pressure aware scheduling for high level synthesis. Design Automation Conference, 2011 pp.461-466, Piscataway, NJ: IEEE Press.
16. A. Aho, M. Lam, R. Sethi and J. Ullman, Compilers: Principles, Techniques and Tools (2nd edition), Boston, Mass: Addison-Wesley Pub. Co, 2007.