Proceedings of the GCC Developers' Summit

June 22nd–24th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Eric Christopher, *Red Hat, Inc.*
David Edelsohn, *IBM*
Richard Henderson, *Red Hat, Inc.*
Andrew J. Hutton, *Steamballoon, Inc.*
Janis Johnson, *IBM*
Toshi Morita
Gerald Pfeifer, *Novell*
C. Craig Ross, *Linux Symposium*
Al Stone, *HP*
Zack Weinberg, *Codesourcery*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.

Performance Improvements for GCC Using Architecture Features on IA-64

Canqun Yang, Chunjiang Li, and Feng Wang
National University of Defense Technology, P.R. China
`canqun@nudt.edu.cn, chunjiangli@263.net, fengwang@nudt.edu.cn`

Abstract

The Intel IA-64 architecture provides a rich set of features to aid the compiler in exploiting instruction-level parallelism to achieve high performance. Currently, GCC is a widely used open source compiler on IA-64 platform, but its performance, especially floating-point performance is poor compared with commercial compilers because it has not fully utilized those features. We have been absorbed in improving performance of GCC on IA-64 architecture since late 2003. This paper reports some of our work on several algorithms concerning with architecture features of IA-64 for achieving higher floating-point performance of GCC, including FORTRAN alias analysis, induction variable optimizations, loop unrolling, and prefetch of loop arrays. These improvements have markedly optimized the floating-point performance of GCC on Itanium-2 systems.

1 Introduction

The Intel IA-64 architecture[1] provides many supports for instruction-level parallelism (ILP), such as Explicitly Parallel Instruction Computing (EPIC), control and data speculation, predication, register rotation, software pipelined loop branches, large register files, along with special instructions such as data prefetch, post-increment loads and stores, load-pair etc. The compiler for this architecture can achieve high performance no other than fully exploit the capability of instruction-level parallelism.

The open source compiler, GCC, is widely used in many areas for its features of multi-platform compatibility and multi-language supportability. But its performance (especially floating-point) is very low on IA-64 platform compared with commercial compilers. On IA-64 system, the performance of GCC is about 70% of the Intel compiler with test suite of SPEC CINT2000, and about 30% with SPEC CFP2000. Since 2001, projects aiming at improving GCC performance on IA-64 have been started[2]. But except for major changes on its infrastructure, its performance of floating-point nearly has not improved.

We have been concentrating on improving the performance of GCC on IA-64 platforms from late 2003. We did most of our work on the GCC of the version 3.5-tree-ssa branch on Dec. 21, 2003. As this version of GCC can only pass a small set of benchmarks, firstly we implemented some unsupported FORTRAN syntax to make it pass all the NAS benchmarks and most of SPEC CFP2000 benchmarks excluding 191.fma3d and 200.sixtrack. In this paper, without additional version notations, the term

GCC refers to this version of GCC.

In this paper, we grasp the main factors which affect the performance seriously, then improve the FORTRAN alias analysis, induction variable optimizations, loop unrolling, and prefetch of loop arrays. On Itanium-2 1.0GHz system, it has shown that our efforts improved the floating-point performance of GCC greatly, 42% improvements of SPEC CFP2000 benchmarks and 56% improvements for NAS benchmarks. By concluding our work, we present our ideas about the future optimization directions of GCC. It is hopefully that GCC could exploit the performance features of IA-64 Architecture.

This paper is organized as follows. The limitation of GCC on IA-64 Architecture is analyzed firstly in Section 2. In Section 3, our improvement and implementation done on some algorithms and techniques for GCC is presented. Performance comparison is given in Section 4. In Section 5, further optimization directions are discussed. And the conclusion is drawn in Section 6.

2 Limitation Analysis of GCC on IA-64

Compared with Intel compiler, the important optimizations which have not been fully implemented in GCC include software pipelining, inter-procedural optimizations and loop transformation[3]. So, we perform performance tests for Intel compiler to analyze how much the optimizations which have not been fully implemented in GCC contribute to the performance gain in Intel compiler. The test suite is SPEC CFP2000 benchmarks, and the version of the Intel compiler is V8.0.

By default, Intel compiler does not perform inter-procedural optimizations without the option `-ipo`. When at the optimization level `-O2`, Intel compiler performs software pipelining, but it can be turned off by adding directives: `!DIR$ NOSWP` for FORTRAN and `#pragma noswp` for C/C++. There are no options and directives related to loop transformation. At the optimization level `-O3`, Intel compiler conducts the following three high level loop optimizations: loop transformation, scalar replacement and data prefetch. So we can use the optimization level `-O2` to roughly estimate the effects of loop transformation.

In Figure 1, `Intel -O2 -NOSWP` means that Intel compiler turns off the high level loop optimization and software pipelining; `Intel -O2` means turning off the high level loop optimizations only; `Intel -O3 -NOSWP` means turning off software pipelining only; and `Intel -O3 -ipo` means turning on the inter-procedural optimizations. Compile options for GCC are `-O3 -ffast-math -funroll-loops -fprefetch-loop-arrays`.

Figure 1 depicts the performance of SPEC CFP2000 under different optimizations. Because there are execution errors with the program 187.facerec and 191.fma3d at `-O2` optimization level, results compared with `Intel -O2` and `Intel -O2 -NOSWP` are less precise.

1. With high level loop optimization (`Intel -O3` vs. `Intel -O2`), the performance increases 35%;

2. With software pipelining (`Intel -O3` vs. `Intel -O3 -NOSWP`), the performance increases 19%;

3. With inter-procedural optimizations (`Intel -O3 -ipo` vs. `Intel -O3`), the performance increases 36%;

4. Turning off these three optimizations (`Intel -O2 -NOSWP` vs. `GCC`), the overall performance is still 18% higher than GCC.

Obviously, the former three kinds of optimizations are the most important ones for improving the performance of GCC on IA-64 system. But those optimizations already implemented in GCC still need improvements. We conduct performance analysis for GCC with lots of means, including using performance analysis tools such as GPROF and PFMON, choosing GCC optimization options, tuning optimization parameters of GCC, and analyzing the assembly codes generated by GCC. We have found some major factors affecting the performance of GCC:

1. Without alias analysis for FORTRAN and without taking advantage of the load-pair instructions of IA-64, which result in lots of redundant LOADs.

2. The optimizations for loop are not well done. General induction variable optimizations, loop unrolling and prefetch of loop arrays does not fully utilize the features of IA-64.

3 Implementation and Improvement

Based on the analysis in the former section, we implement and improve several optimizations which affect the performance mostly, including alias analysis of FORTRAN, general induction variable optimizations, loop unrolling, and prefetch of loop arrays.

3.1 Alias Analysis

Alias analysis[4] refers to the determination of storage locations that may be accessed in two or more ways. Alias information is generally gathered by the front-end of the compiler and passed to the back-end to guide the compile optimization. In GCC, alias analysis for FORTRAN is not as adequate as for C/C++.

GCC implements alias analysis at the syntax tree level as a common function for all the programming languages. Each language-specific front-end can define the `LANG_HOOKS_GET_ALIAS_SET` as the alias analysis function of itself. In the FORTRAN front-end of GCC, we defined it as `gfc_get_alias_set`, then implemented alias analysis in this function. By now, we have finished a trial implementation of inner-procedural alias analysis, mainly consider the alias brought by EQUIVALENCE statement, pointers, objects with TARGET attributes, and parameters.

GCC uses alias set to express that the memory references contained in the same set alias each other. That is, two memory references in different alias set can not alias each other. Then if a object does not alias any other objects, we build a new alias set for it. In FORTRAN, objects satisfy the following conditions will not alias any other objects.

1. COMMON variables contained in a COMMON block without any EQUIVALENCE objects.

2. Parameters, if a programmer assumes that there are no aliases for parameters by turning on compile option `-fargument-noalias`.

3. Local variables, which are not pointers and also without TARGET attributes.

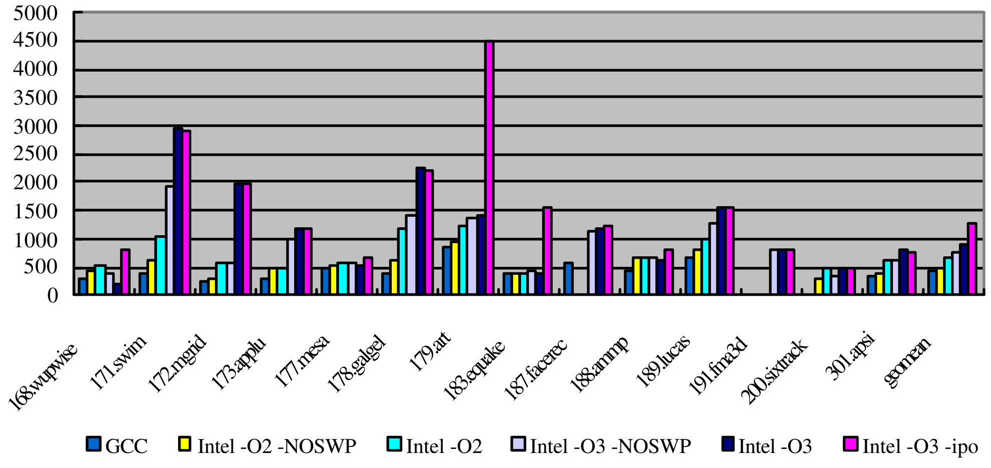

Figure 1: Performance comparison of Intel compiler with high level loop optimization, software pipelining and inter-procedural optimizations.

	171.swim	172.mgrid	173.applu	301.apsi
before	3.20×10^9	1.21×10^{10}	2.75×10^9	2.84×10^9
after	2.20×10^9	7.00×10^9	2.53×10^9	2.24×10^9

Table 1: The number of LOAD instructions retired before and after alias analysis

Through alias analysis, a lot of redundant LOADs can be removed, so the performance of GCC can be improved. An example is shown in Table 1. The result was gotten with the performance tool PFMON. The test programs are SPEC CFP2000 benchmarks in train mode.

3.2 Induction Variable Optimizations

Induction variables[4] are variables whose successive values form an arithmetic progression over a loop. Induction variables are often divided into basic induction variables (BIVs) and general induction variables (GIVs). BIVs are explicitly modified by the same constant amount each iteration of loop, e.g., the loop iterator. GIVs may be modified or computed by a linear equation of the form $GIV = b \times BIV + c$ (where b and c are constants). Optimizations for induction variables include elimination and strength reduction, and strength reduction of GIVs can utilize the architecture feature of IA-64.

IA-64 architecture provides post-increment load and store instructions which can combine the memory access instruction and the address modification instruction into one instruction. Using such kind of instructions, compilers can not only reduce the code size of the object code but also make the object code use less function units.

GCC can identify the address GIV (e.g., address of an array element) of the form: $GIV = b \times BIV + c$, where b represents the size of the array element, BIV represents the loop iterator, and c represents the base address of the array, then perform strength reduction which transforms multiply operation into add operation, and then combine the memory reference and the address increment into one instruction. Although GCC implemented such kind of optimization, it gains little performance improvements on IA-64 because it does not correctly analyze the legality for reducing GIVs.

Usually, the loop iterator (a BIV) is a 32-bit integer, and on 64-bit system, the address of an array element (a GIV) is 64-bit long. When calculating address through loop iterator, first the value of the BIV is calculated, then it is extended to 64-bit integer, and then use $b \times BIV + c$ to evaluate the GIV (the address). Before strength reduction of this type of GIV, GCC checks the BIV to see whether it may overflow or not within its mode (32-bit integer) during loop execution. If it may overflow, the GIV is illegal to be reduced. This check routine is defined by `check_ext_dependent_givs` in `loop.c`, but this checker almost always fails when compiling FORTRAN programs. The reason is that FORTRAN 95 front-end introduces a temporary to replace the iterator, and the checker can not deal with this case.

To make the strength reduction of GIV come into effect, we find a temporary way to fix this problem. When an address GIV is encountered, we suppose it is legal to be reduced.

3.3 Loop Unrolling

Loop unrolling[4] replicates the instructions in the loop body into multiple copies and adjusts the exit code of the loop to generate a new loop body. Loop unrolling not only can reduce the cost of loop execution but also can improve the effectiveness of other optimization such as common sub-expression elimination, induction variable optimizations, data prefetch, and software pipelining. Especially, on the architectures which support instruction-level parallelism, loop unrolling can increase the size of the loop body which provides more chances for improving instruction scheduling.

In GCC, the way of loop unrolling and the determination of the times of unrolling can be classified into following two conditions:

1. The number of iterations can be calculated statically. If the result that the number of iterations times the number of RTL instructions is less than `MAX_UNROLLED_INSNS`, then the loop is unrolled completely. Otherwise, try to unroll the loop a number of times, and ensure the times of unrolling is a factor of the number of iterations, so that only one exit test is needed. The times of unrolling approximately equal to the result that `MAX_UNROLLED_INSNS` divides the number of RTL instructions.

2. The number of iteration can be calculated exactly at runtime and the loop is always entered from top. In such condition, how many times the loop will execute should be calculated firstly, then execute the loop body for a few times so as to ensure that the remaining iterations will be multiple of 4 (or 2 if the loop is large). Then the loop is unrolled 8 (4 or 2) times with only one exit test at the end of the loop.

Obviously, in GCC, loop unrolling is mainly affected by the parameter `MAX_UNROLLED_INSNS`. GCC defines it as 200, which results in that loop unrolling can not succeed when the number of RTL instructions in the loop body exceeds 100. On IA-64 systems, it is not proper to define `MAX_UNROLLED_INSNS` as 200, which makes the loop unrolling ineffective. We compared the execution time of four programs from SPEC CFP2000 under different `MAX_UNROLLED_INSNS` values of 200, 400, 600, and 800. The result is shown in figure 2, programs run in train mode. It is obvious that when `MAX_UNROLLED_INSNS` equals 600, loop unrolling can achieve high effectiveness.

Increasing `MAX_UNROLLED_INSNS`, some big loop can be unrolled and the times of unrolling can also increase, which may lead to high register pressure. But IA-64 systems have

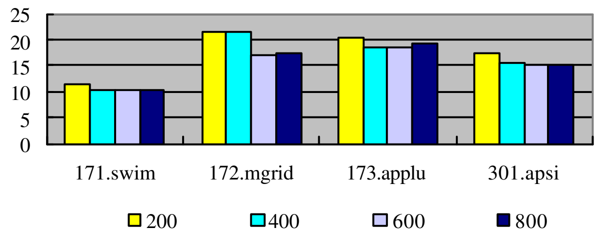

Figure 2: Effectiveness of loop unrolling in GCC under different `MAX_UNROLLED_INSNS` values.

large register files. It can endure higher register pressure. So, we can improve the loop unrolling algorithm in GCC by properly increasing the value of `MAX_UNROLLED_INSNS`.

3.4 Data Prefetch

Data prefetch is a technique for hiding memory access latency. It overlaps the memory access and computing or other memory access. Data prefetch need to properly insert prefetch instructions in the program to ensure that data can be fetched to the nearest level of memory hierarchy before being used. Data prefetch is a kind of complement for data locality optimization such as loop transformation and scalar replacement.

GCC implemented data prefetch at the RTL level, but it had no effectiveness on IA-64 platform. The machine description file did not present proper information to guide the data prefetch algorithm and the algorithm itself is not properly designed. The condition controlling data prefetch in GCC mainly includes:

1. `PREFETCH_NO_CALL`. Whether to insert prefetches into the loops containing function call is determined by this macro. If it is defined as 1, prefetch instructions can not be inserted in such loops.

2. `PREFETCH_DENSE_MEM`. Setting the value of memory access density for prefetches. It refers to the ratio of amount of bytes accessed by a prefetch to the amount of bytes prefetched. Memory access generally takes word or dual-word as an access unit, but the prefetch usually read a cache line containing 4 or more than 4 words. So, `PREFETCH_DENSE_MEM` reflects the effectiveness of a single prefetch. GCC sets this value as 220/256.

3. `PREFETCH_LOW_LOOPCNT`, least iteration times, is set as 32, means that prefetches can be inserted only if the number of iteration is more than 32.

4. `SIMULTANEOUS_PREFETCHES`. Setting of the maximum number of the real prefetch instructions which can be inserted into loop body. If the number is greater than `SIMULTANEOUS_PREFETCHES`, no prefetch instructions will be inserted into the loop body.

We adjust the above four parameters repeatedly, then we find that the definitions of the first three parameters are reasonable, but the definition of `SIMULTANEOUS_PREFETCHES` is reasonless. In the machine description file of IA-64, `SIMULTANEOUS_PREFETCHES` is set as 6 which equals the maximal number of instruction that IA-64 can issued simultaneously. While computing intensive loops in the programs in SPEC CFP2000 need more than 6 prefetch instructions, but on the contrary, according to the conditions above, these loops do not be inserted any data prefetch instructions.

Figure 3 depicts the statistical results of prefetch requirements of some programs in SPEC CFP2000, x-axis denotes the number of prefetches required and y-axis denotes the accumulative distribution of prefetches in loops. The result is generated after adding following optimizations: alias analysis for FORTRAN, GIV optimization and loop unrolling. As the figure shows that the number of prefetches required by most of the loops is less than or equal to 6, but these loops account for small part of the execution time of the programs, while the loops which account for large part of the execution time require far more than 6 prefetches.

For example, four loops in 171.swim, loop 100 in procedure CALC1, loop 200 in procedure CALC2, loop 300 in procedure CACL3 and loop 400 in procedure CACL3Z, the number of prefetches required by them is 8, 6, 12, and 9 respectively. The two loops which account for nearly all the execution time of the program 172.mgrid, loop 600 in procedure PSINV and loop 800 in procedure RESID, the number of prefetches required by them is 28 and 29 respectively.

So, it is not proper to set a small upper bound for the number of prefetches which can be inserted into loops. But inserting too many prefetch instructions will result in performance degradation. Adopting rotating register allocation can dramatically reduce the number of prefetch instructions inserted to the loops[5], but GCC has not implemented rotating register allocation. We improved the prefetch algorithm for IA-64 as follows:

1. If a loop contains function call or its number of iteration is less than `PREFETCH_LOW_LOOPCNT`, then do not generate prefetch instructions.
2. Merge the prefetches which access the same cache line, and record the times of mergence.
3. Evaluate the memory access density of all prefetches, discard the prefetches whose memory access density are less than `PREFETCH_DENSE_MEM`.
4. Determine the upper bound of the number of the real prefetch instructions, *Pmax*, according to `SIMULTANEOUS_PREFETCHES` and the number of instructions in the loop (*num_insts*). For IA-64, our experiences shows defining `SIMULTANEOUS_PREFETCHES` as 18 and let the maxmum prefetches not to exceed one eighth of the *num_insts* will get good results. So, let *Pmax* = MIN (SIMULTANEOUS_PREFETCHES, *num_insts* $\div$ 8).
5. If the number of real prefetch instructions (*Pnum*) exceeds *Pmax*, sort all the prefetches according to the following two criteria in order:
   (a) Times of mergence. More times means this prefetch will satisfy more array accesses, so give it higher priority.
   (b) Memory access density. Higher density means higher efficiency, so higher priority.
6. Choose the first *K* prefetches to emit real prefetch instructions for them before and within the loop, where *K* satisfies the following condition: $\sum_{i=1}^{k} prefetch(i) \leq$ MIN(*Pmax*, *Pnum*), and *prefetch(i)* represents the number of the real prefetch instructions the *ith* prefetch needed within the loop.

4 Performance Evaluation

4.1 Platform

We performed performance evaluation on an IA-64 system for the implementation and improvement of all the optimizations described above. The platform is an Itanium-2

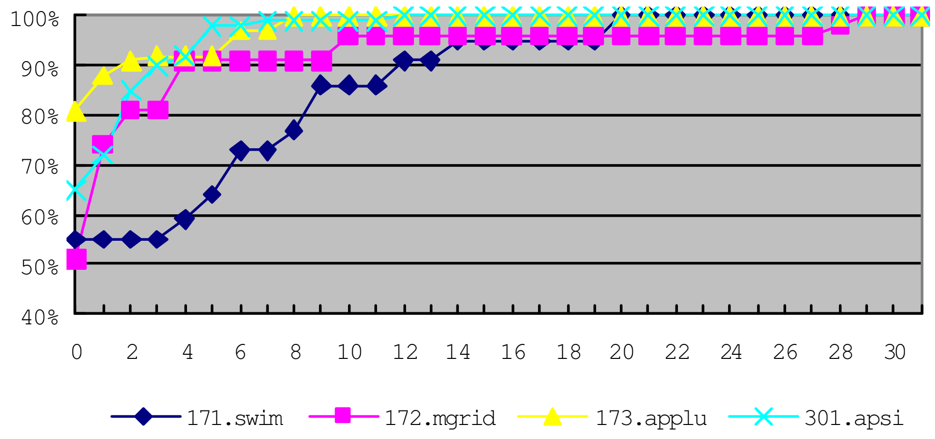

Figure 3: The statistical results of prefetch instruction requirements of some programs in SPEC CFP2000.

server with 2 CPUs of 1GHz, 2GB memory, 16KB L1 instruction cache and 16KB data cache, 256KB L2 cache, 3MB L3 cache, and Red Hat Enterprise Linux AS 2.1 as operating system. The test suites are SPEC CFP2000 benchmarks (excluding 191.fma3d and 200.sixtrack) and NAS benchmarks. The compile option for all the version of GCC is `-O3 -ffast-math -funroll-loops -fprefetch-loop-arrays`.

4.2 Compared with the Original GCC

Figure 4 describes the result under different optimizations. `+alias` means alias analysis of FORTRAN. `+giv` means adding general induction variable optimizations to the former optimizations. `+unroll` means adding loop unrolling to the former optimizations. `+prefetch` means adding prefetch of loop arrays to all the former optimizations. `geo-mean` is the geometric mean of the SPEC ratio.

The ratio of SPEC CFP2000 with GCC is 420. After adding alias analysis, the ratio reaches 455; after adding general induction variable optimization, the ratio reaches 470; after adding loop unrolling, the ratio reaches 540; after adding prefetch of loop arrays, the ratio reaches 596. So, after adding all the above optimizations into GCC, the ratio of SPEC CFP2000 increases from 420 to 596, increasing 42%.

Figure 5 depicts the performance of NAS programs after adding the optimizations described above incrementally. After adding alias analysis, the speedup of NAS programs reaches 1.1; after adding general induction variable optimization, it reaches 1.14; after adding loop unrolling, it reaches 1.52; after adding prefetch, it reaches 1.56.

4.3 Compared with GCC-4.0.0

We try to apply our optimizations to GCC-4.0.0. For COMMON variables, we create `RECORD_TYPE` declarations for COMMON blocks without EQUVALENCE objects, this makes the alias analyzer in GCC backend work well. This work was already committed in GCC-4.0.0. We also modify `loop.c` to improve general induction optimizations, prefetch of loop arrays, and adjust the `MAX_UNROLLED_INSNS` defined in `params.def` or to use the compile option

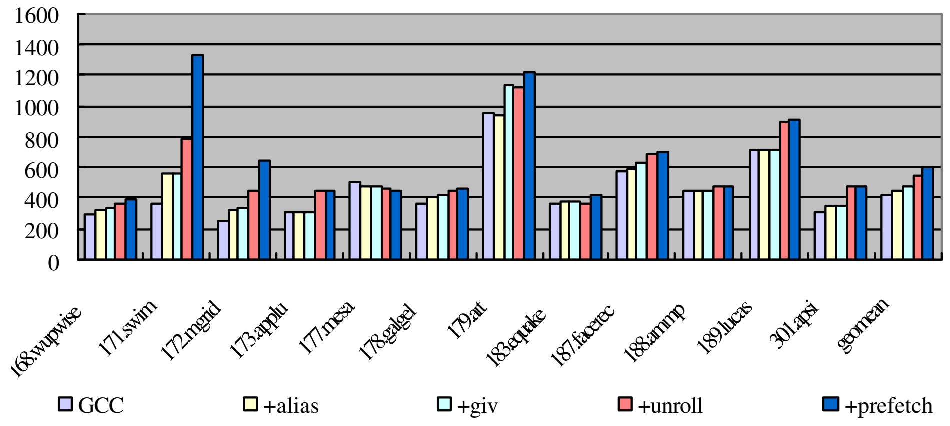

Figure 4: The performance of SPEC CFP2000 benchmarks after incrementally adding the optimizations presented in this paper.

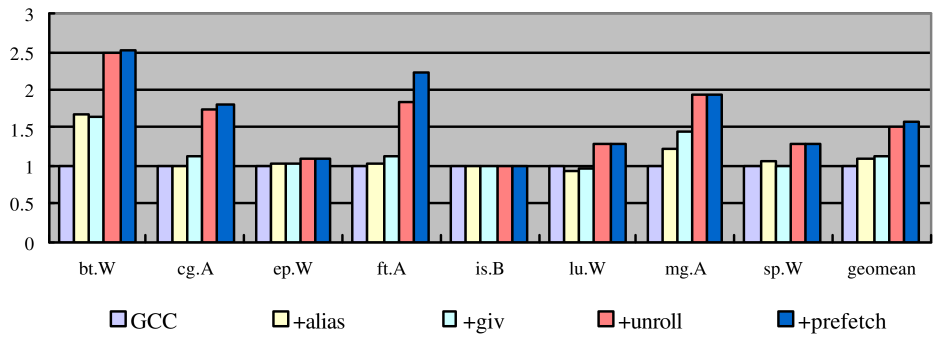

Figure 5: The speedup of NAS benchmarks after incrementally adding the optimizations presented in this paper.

`-param max-unrolled-insns` to improve loop unrolling. As the loop.c will be removed and the new RTL loop optimizer does not support address GIV splitting. And this defect in GCC-4.0.0 greatly degrades the effectiveness of applying optimizations discussed in this paper, as shown in Figure 6 and Figure 7. Our patches for these optimizations have not been committed yet.

Figure 6 depicts the test result for SPEC CFP2000 benchmarks after incrementally adding optimizations presented in this paper to GCC-4.0.0. The ratio of SPEC CFP2000 with GCC-4.0.0 is 450. After adding alias analysis, the ratio reaches 485; after adding general induction variable optimizations, the ratio is 486, nearly no increase; after adding loop unrolling, the ratio reaches 500; after adding prefetch of loop arrays, the ratio reaches 516. So, after adding all the above optimizations into GCC-4.0.0, the ratio of SPEC CFP2000 increases from 450 to 516, increasing 15%. The spec ratio of our optimized GCC reaches 596, increasing 33% compared with gcc-4.0.0.

Figure 7 depicts the performance increase of NAS programs after adding the optimizations described above incrementally to GCC-4.0.0. After adding alias analysis, the speedup of NAS programs reaches 1.09; after adding general induction variable optimization, it reaches 1.14; after adding loop unrolling, it reaches 1.27; after adding prefetch of loop arrays, it reaches 1.32. The speedup of our optimized GCC reaches 1.49 compared with gcc-4.0.0.

5 Future Optimization Directions

For IA-64 platform, the optimizations of GCC should emphasize on the following: loop transformation, software pipelining, and interprocedural optimization.

Loop transformation [6] is aiming at achieving higher cache locality. GCC implemented the optimization framework for nesting loops [7], based on this framework, the problem should be solved including powerful data dependence analysis, and fully implementing all kinds of loop transformation algorithms.

GCC has implemented software pipelining preliminarily [8]. But the SWING modulo scheduler can only make successful schedule for simple loops on IA-64. The work for consummating software pipelining includes: precise dependence analysis, improvement of scheduling algorithm and fully take advantage of software pipelining supports provided by IA-64.

Inter-procedural optimizations play an important role in Intel compiler on IA-64 systems. GCC partially implemented inter-procedural optimizations in a single file, such as inline and constant propagation, but much effort is needed to make it take full effectiveness.

6 Conclusion

In this paper, we find the main factors which affect GCC performance mostly on IA-64 platform using comparative test and analysis. A few optimization algorithms are improved and implemented which result in significant performance improvements. Although compared with Intel commercial compiler, there still exists a vast performance gap. It is hopeful that the performance of GCC can reach or approach the performance of commercial compilers.

7 Acknowledgment

We would like to thank the other CCRG (Creative Compiler Research Group) members for

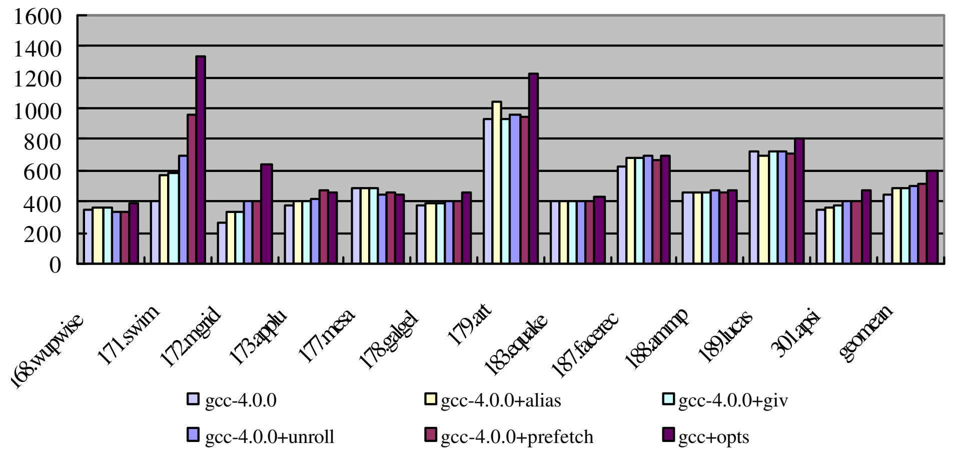

Figure 6: The performance of SPEC CFP2000 benchmarks after incrementally adding the optimizations presented in this paper to GCC-4.0.0.

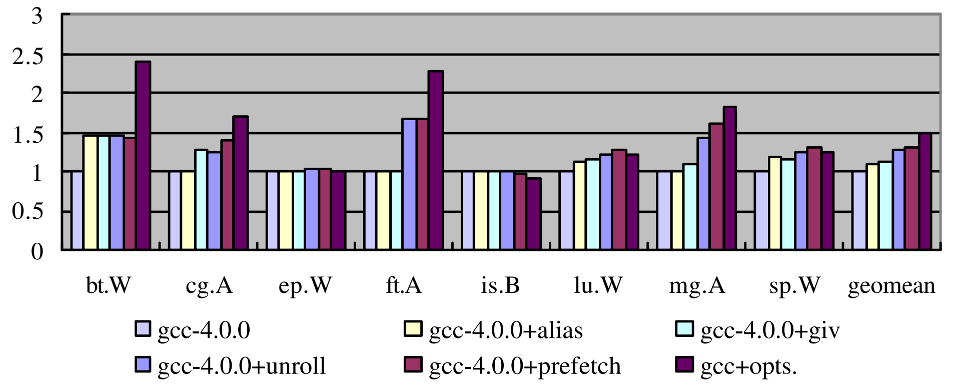

Figure 7: The speedup of NAS benchmarks after incrementally adding the optimizations presented in this paper to GCC-4.0.0.

their cooperative and creative work. They are Kejia Zhao, Chun Huang, Xiaoqiang Zhang, and Lin Peng. Also, we would like to thank all those GCC developers who gave us a lot of valuable advice and help.

References

[1] Intel, *Intel IA-64 Architecture Software Developer's Manual*, Vol. 1, October 2002.

[2] the GCC Summit Participants, *Projects to Improve Performance on IA-64*, `http://www.ia64-linux.org/compilers/gcc_summit.html`. the GCC IA-64 Summit, June 6, 2001.

[3] R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, C. Lim, J. Ng, D. Sehr, *An Advanced Optimizer for the IA-64 Architecture*, IEEE Micro, Nov–Dec. 2000.

[4] Steven S. Muchnick, *Advanced Compiler Design Implementation*, Academic Press, 1997.

[5] G. Doshi, R. Krishnaiyer, K. Muthukumar, *Optimizing Software Data Prefetches with Rotating Registers*, Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques, September 2001. Page(s): 257–267.

[6] U. Banerjee, *Loop Transformations for Restructuring Compilers*, Kluwer Academic Publishers, Boston, 1994.

[7] David Edelsohn, *High-Level Loop Optimizations for GCC*, In Proceedings of the 2004 GCC Developers' Summit, pages 37–54, June 2004.

[8] Mostafa Hagog, *Swing Modulo Scheduling for GCC*, In Proceedings of the 2004 GCC Developers' Summit, pages 55–64, June 2004.

[9] Jan Hubička, *The GCC Call Graph Module: a Framework for Inter-procedural Optimization*, In Proceedings of the 2004 GCC Developers' Summit, pages 65–78, June 2004.