Improving the Performance of GCC by Exploiting IA-64 Architectural Features[⋆]

Canqun Yang[1], Xuejun Yang[1] and Jingling Xue[2]

[1] School of Computer Science, National University of Defense Technology, Changsha, Hunan 410073, China
[2] Programming Languages and Compilers Group, School of Computer Science and Engineering, The University of New South Wales, Sydney, NSW 2052, Australia

Abstract. The 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 for IA-64, but its performance, especially its floating-point performance, is poor compared to that of commercial compilers because it has not fully utilized IA-64 architectural features. Since late 2003 we have been working on improving the performance of GCC on IA-64. This paper reports four improvements on enhancing its floating-point performance, namely alias analysis for FORTRAN (its part for COMMON variables already committed in GCC 4.0.0), general induction variable optimization, loop unrolling and prefetching arrays in loops. These improvements have significantly improved the floating-point performance of GCC on IA-64 as extensively validated using SPECfp2000 and NAS benchmarks.

1 Introduction

Based on EPIC (Explicitly Parallel Instruction Computing) technology, the IA-64 architecture[8, 9] was designed to allow the compiler explicit control over the execution resources of the processor in order to maximize instruction-level parallelism (ILP). To achieve this, the IA-64 architecture provides a rich set of architectural features to facilitate and maximize the ability of the compiler to expose, enhance and exploit instruction-level parallelism (ILP). These features include speculation, predication, register rotation, advanced branch architecture, special instructions such as data prefetching, load pre- increment, store pre-increment and many others. Thus, the compiler's ability to deliver many of the functionalities that are commonly realized in the processor hardware greatly impacts the performance of the processors in the IA-64 family.

GCC (GNU Compiler Collection) is an open-source, multi-language and multi-platform compiler. Being fairly portable and highly optimizing, GCC is widely used in research, business, industry and education. However, its performance, especially its floating-point performance, on the IA-64 architecture, is poor compared to that of commercial compilers such as Intel's icc [4]. We have measured the performance results of GCC (version 3.5-tree-ssa) and icc (version 8.0) using SPEC CPU2000 benchmarks

[⋆] This work was supported by the National High Technology 863 Program of China under Grant 2004AA1Z2210.

on a 1.0 GHz Itanium 2 system. GCC has attained 70% of the performance of icc for SPECint2000. In the case of SPECfp2000, however, the performance of GCC has dropped to 30% of that of icc. Since 2001 several projects have been underway on improving the performance of GCC on IA-64 [16]. While the overall architecture of GCC has undergone some major changes, its performance on IA-64 has not improved much.

This paper describes some progress we have made in our ongoing project on improving the performance of GCC on the IA-64 architecture. Commercial compilers such as Intel's icc and HP's compilers are proprietary. Research compilers such as ORC [15] and openIMPACT [14] are open-source but include only a few frontends (some of which are not extensively tested). GCC is attractive to us since it is an open-source, portable, multi-language and multi-platform compiler. We are interested in IA-64 partly because it is a challenging platform for compiler research and partly because of our desire in developing also an open-source compiler framework for VLIW embedded processors.

In late 2003, we initiated this project on improving the performance of GCC on IA-64. We have done most of our research in GCC 3.5-tree-ssa. As this version fails to compile many SPECfp2000 and NAS benchmarks, we have fixed the FORTRAN frontend so that all except the two SPECfp2000 benchmarks, `fma3d` and `sixtrack`, can compile successfully. We are currently porting our work to GCC 4.0.0.

In this paper, we report four improvements we have incorporated into GCC for improving its floating-point performance, namely, alias analysis for FORTRAN, general induction variable optimization, loop unrolling and prefetching arrays in loops. Our alias analysis for COMMON variables has already been committed in GCC 4.0.0. The four improvements were originally implemented in GCC 3.5 and have recently been ported to GCC 4.0.0 as well. In GCC 3.5, we have observed a performance increase of 41.8% for SPECfp2000 and 56.1% for the NAS benchmark suite on a 1.0 GHz Itanium 2 system. In GCC 4.0.0, its new loop unrolling has a performance bug: it does not (although it should have) split induction variables as it did in GCC 3.5. This affects the benefit of our loop unrolling negatively in some benchmarks. Our improvements incorporated into GCC 4.0.0 have resulted a performance increase of 14.7% for SPECfp2000 and 32.0% for NAS benchmark suite, respectively. Finally, GCC 3.5 (with our four improvements included) outperforms GCC 4.0.0 (the latest GCC release) by 32.5% for SPECfp2000 and 48.9% for NAS benchmark suite, respectively.

The plan of this paper is as follows. Section 2 reviews the overall structure of GCC. Section 3 discusses its limitations that we have identified and addressed in this work. In Section 4, we present our improvements for addressing these limitations. In Section 5, we present the performance benefits of all our improvements for SPECfp2000 and NAS benchmarks on an Itanium 2 system. Section 6 reviews the related work. Section 7 concludes the paper and discusses some future research directions.

2 GCC Overview

GCC consists of language-specific frontends, a language-independent backend and architecture-specific machine descriptions [18, 20]. The frontend for a language translates a program in that language into an abstract syntax tree called *GIMPLE*. High-level optimizations, such as alias analysis, function inlining, loop transformations and partial redundancy elimination (PRE), are carried out on GIMPLE. However, high-level optimizations in GCC are limited and are thus topics of some recent projects [5].

Once all the tree-level optimizations have been performed, the syntax tree is converted into an intermediate representation called *RTL* (Register Transfer Language). Many classic optimizations are done at the RTL level, including strength reduction, induction variable optimization, loop unrolling, prefetching arrays in loops, instruction scheduling, register allocation and machine-dependent optimizations. In comparison with the tree-level optimizations (done on GIMPLE), the RTL-to-RTL passes are more comprehensive and more effective in boosting application performance.

Finally, the RTL representation is translated into assembly code. A *machine description* for a target machine contains all machine-specific information consulted by various compiler passes. Such a description consists of instruction patterns used for generating RTL instructions from GIMPLE and for generating assembly code after all RTL-to-RTL passes. Properly defined instruction patterns can help generate optimized code. In addition, a machine description also contains macro definitions for the target processor (e.g., processor architecture and function calling conventions).

3 Limitations of GCC on IA-64

The performance of GCC is quite far behind that of icc: GCC 3.5 reaches only 70% and 30% of icc 8.0 in SPECint2000 and SPECfp2000, respectively. Compared to icc, GCC 3.5 lacks loop transformations such as loop interchange, loop distribution, loop fusion and loop tiling, software pipelining and interprocedural optimizations. These three kinds of important optimizations are critical for icc's performance advantages. The importance of these optimizations for improving the performance of GCC was noted [16]. However, little progress has been made in GCC 4.0.0. The function inlining remains the only interprocedural optimization supported in GCC 4.0.0. The SWING modulo scheduler [7] was included in GCC 4.0.0 to support software pipelining. According to our experimental results, it cannot successfully schedule any loops in the SPECfp2000 and NAS benchmarks on IA-64. Loop interchange was added in GCC 4.0.0 but again it has not interchanged any loops in the SPECfp2000 and NAS benchmarks on IA-64.

One of our long-term goals is to develop and implement these three kinds of important optimizations in GCC to boost its performance on IA-64. At the same time, we will try to maintain the competitive edge of GCC as an open-source, multi-language and multi-platform compiler. In this early stage of our project, our strategy is to identify some limitations in the current GCC framework so that their refinements can lead to significant increase in the floating-point performance of GCC on IA-64. We have analyzed extensively the performance results of benchmarks compiled under different optimization levels and different (user-invisible) tunable optimization parameters in GCC using tools such as `gprof` and `pfmon`. We have also analyzed numerous assembly programs generated by GCC. The following two problem areas of GCC on IA-64 are identified:

– There is no alias analysis for FORTRAN programs in GCC. The lack of alias information reduces opportunities for many later RTL-level optimizations.

– The loop optimizations in GCC are weak. In particular, general induction variable optimization, loop unrolling and prefetching arrays in loops do not fully utilize IA-64 architectural features in exposing and exploiting instruction-level parallelism. Due to the lack of sophisticated high-level optimizations, the effectiveness of these RTL optimizations can be critical to the overall performance of GCC.

4 Improvements of GCC for IA-64

In this section, we present our improvements for the four components of GCC that we identified in Section 3 in order to boost its floating-point performance significantly. These four components are alias analysis for FORTRAN, general induction variable optimization, loop unrolling and prefetching arrays in loops. The alias analysis is performed on GIMPLE while the three optimizations are done at the RTL level.

We describe our improvements to the four components of GCC in separate subsections. In each case, we first describe the current status of GCC, then present our solution, and finally, evaluate its effectiveness using some selected benchmarks. Once having presented all the four improvements, we discuss the performance results of our improvements for the SPECfp2000 benchmark suite and the NAS benchmark suite. In this section, all benchmarks are compiled under GCC 3.5 at “-O3” on an Itanium 2 system, whose hardware details can be found in Section 5.

4.1 Alias Analysis

Alias analysis refers to the determination of storage locations that may be accessed in more than one way. Alias information is generally gathered by the front-end of the compiler and passed to the back-end to guide later compile optimizations. In GCC, alias analysis has been implemented for C/C++ but not for FORTRAN. This section introduces a simple alias analysis module we have added for FORTRAN in GCC.

GCC conducts alias analysis at the tree level (i.e., on GIMPLE) via an interface function, LANG_HOOKS_GET_ALIAS_SET, common to all programming languages. Each language-specific frontend provides its own implementation of this interface function. We have completed a simple implementation of an intraprocedural alias analysis for FORTRAN, by mainly detecting the aliases created due to EQUIVALENCE statements, pointers, objects with TARGET attributes and parameters.

In GCC, an alias set contains all memory references that are aliases to each other. Two memory references in different alias sets are not aliases. In our intraprocedural alias analysis, the alias sets are constructed based on the following simple facts:

– A COMMON variable that is contained in a COMMON block is its own alias set if there are not EQUIVALENCE objects within this COMMON block.
– There are no aliases for a parameter of a function (except the parameter itself) if the compiler switch “-fargument-noalias” is enabled by the user.
– A local variable is in its own alias set if it is not a pointer and does not have a TARGET attribute.

Figure 1 shows that such a simple alias analysis is already effective in removing redundant load instructions. These results for the four SPECfp2000 benchmarks compiled under GCC 3.5 at “-O3” are obtained using `pfmon` running with the train inputs. The percentage reductions for the four benchmarks `swim`, `mgrid`, `applu` and `apsi` are 31.25%, 42.15%, 8.00% and 21.13%, respectively.

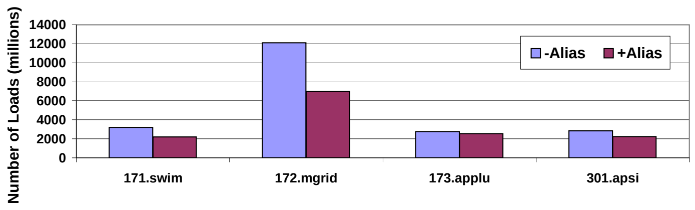


Fig. 1. Load instructions retired with (+Alias) and without (-Alias) alias analysis (in GCC 3.5).

4.2 General Induction Variable Optimizations

Induction variables are variables whose successive values form an arithmetic progression over some part (usually a loop) of a program. They are often divided into two categories: basic induction variables (BIVs) and general induction variables (GIVs). A BIV is modified (incremented or decremented) explicitly by the same constant amount during each iteration of a loop. A GIV may be modified in a more complex manner. There are two kinds of optimizations for induction variables: induction variable elimination and strength reduction. We improve the efficiency of the strength reduction of GIVs on IA-64 by utilizing some IA-64 architectural features.

GCC can identify the *GIV* of the form $b + c \times I$. If this is the address of an array, then b represents the base address of the array, I a loop variable and c the size of the array element. An address GIV can be strength reduced by replacing the multiplication $c \times I$ in the GIV with additions. This enables the array access operation and the address increment/decrement to be combined into one instruction. Such an optimization has been implemented in GCC. However, there are no great performance improvements on IA-64 since the legality test required for the optimization is too conservative.

In programs running on IA-64, the loop variable I (a BIV) is typically a 32-bit integer variable while the address of an array element (a GIV) is typically 64-bit long. The address $b + c \times I$ is normally computed as follows. First, the BIV I is evaluated and extended into 64 bits. Then $b + c \times I$ is evaluated to obtain the address GIV. Before performing the strength reduction for the address GIV, GCC first checks to see if the BIV may overflow or not (as a 32-bit integer) during loop execution. If the BIV may overflow, then whether the GIV can be legally reduced or not depends on whether I is unsigned or signed. In programming languages such as C, C++ and FORTRAN, unsigned types have the special property of never overflowing in arithmetic. Therefore, the strength reduction for the address GIV as discussed above may not be legal. For signed types, the semantics for an overflow in programming languages are usually undefined. In this case, GCC uses a compiler switch to determine if the strength reduction can be performed or not. If “-fwrapv” is turned on, then signed arithmetic overflow is well-defined. The strength reduction for the address GIV may not be legal if the BIV may overflow. If “-fwrapv” is turned off, then the strength reduction can be performed. In GCC, the function that performs the legality test for the GIV strength reduction is check_ext_dependent_givs. When compiling FORTRAN programs, the outcome of such a legality test is almost always negative. This is because the FORTRAN frontend introduces a temporary to replace the BIV I, causing the test to fail in general.

We have made two refinements for this optimization. First, the strength reduction for an address GIV is always performed if “-fwrapv” is turned off (which is the default case). The BIVs are signed in FORTRAN programs. This refinement yield good performance benefits for some benchmarks. Second, for unsigned BIVs (as in C benchmarks), we perform a limited form of symbolic analysis to check if these BIVs may overflow or not. If they do not overflow, then the GIV strength reduction can be performed.

Figure 2 illustrates the performance impact of the improved GIV optimization on four SPECfp2000 benchmarks. These benchmarks are compiled under GCC 3.5 at “-O3” with our alias analysis being enabled. The cycle distributions on the Itanium 2 processors are obtained as per [19]. Reducing the strength for an address GIV creates the opportunity for the array address access and address increment/decrement operations to be merged into one instruction. Therefore, unstalled cycles for the four benchmarks are significantly reduced. The percentage reductions for `wupwise`, `swim`, `mgrid` and `apsi` are 14.82%, 25.34%, 19.59% and 23.96%, respectively.

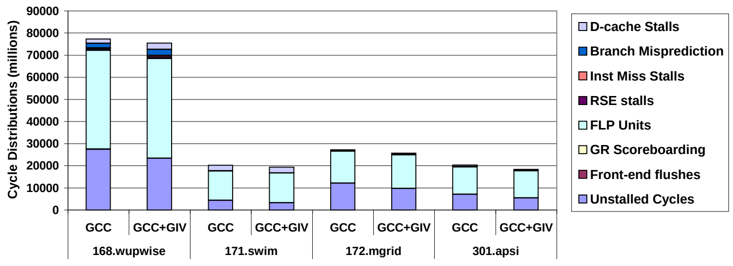


Fig. 2. Effects of the improved GIV optimization on Itanium cycle categories (in GCC 3.5).

4.3 Loop Unrolling

Loop unrolling for a loop replicates the instructions in its loop body into multiple copies. In addition to reduce the loop overhead, loop unrolling can also improve the effectiveness of other optimizations such as common subexpression elimination, induction variable optimizations, instruction scheduling and software pipelining. Loop unrolling is particularly effective in exposing and enhancing instruction-level parallelism.

In GCC, the effectiveness of loop unrolling is crucially dependent on a tunable parameter called MAX_UNROLLED_INSNS. This parameter specifies the maximum number of (RTL) instructions that is allowed in an unrolled loop. The default is 200.

The existing loop unrolling algorithm in GCC works as follows. Let LOOP_CNT be the number of iterations in a loop. Let NUM_INSNS be the number of instructions in a loop. Let UNROLL_FACTOR be the number of times that the loop is unrolled. UNROLL_FACTOR is chosen so that the following condition always holds:

$$\text{NUM\_INSNS} \times \text{UNROLL\_FACTOR} < \text{MAX\_UNROLLED\_INSNS} \quad (1)$$

The situation when the exact value of LOOP_CNT can be calculated statically (i.e., at compile time) is handled specially. The loop will be fully unrolled when

$$\text{NUM\_INSNS} \times \text{LOOP\_CNT} < \text{MAX\_UNROLLED\_INSNS} \quad (2)$$

Otherwise, UNROLL_FACTOR is set as the largest divisible factor of LOOP_CNT such that (1) holds. If UNROLL_FACTOR has not been determined so far or LOOP_CNT can only be calculated exactly at run time, UNROLL_FACTOR is set as the largest in $\{2, 4, 8\}$ such that (1) holds. Finally, the so-called preconditioning code is generated for a loop whenever possible so that only one exit test is needed in the unrolled loop.

Loop unrolling on IA-64 is not effective since the default 200 for MAX_UNROLLED_INSNS is inappropriate for this architecture. We have done extensive benchmarking by trying different values. We found that loop unrolling is the most effective on IA-64 if MAX_UNROLLED_INSNS is set to be 600. Figure 3 gives the cycle distributions of four SPECfp2000 benchmarks compiled under GCC 3.5 at "-O3 -funroll-loops" with improved alias analysis and GIV optimization and run under the train inputs when MAX_UNROLLED_INSNS takes four different values.

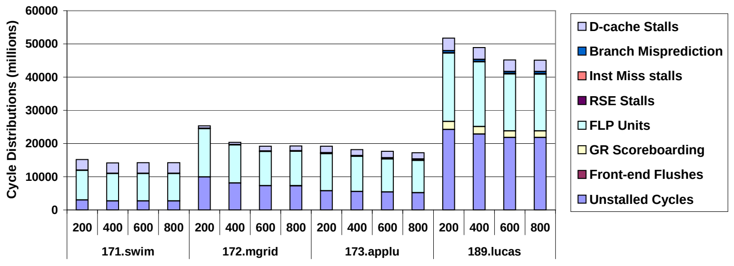


Fig. 3. Effects of the improved loop unrolling on Itanium cycle categories (in GCC 3.5).

As shown in the experimental results, loop unrolling becomes more effective when performed more aggressively on IA-64. Unrolling more iterations in a loop tends to increase the amount of instruction-level parallelism in the loop. As a result, the number of unstalled cycles and the number of cycles spent on the FLP units are both reduced. The best performance results for the four benchmarks are attained when MAX_UNROLLED_INSNS = 600. However, loop unrolling may increase register pressure and code size. As MAX_UNROLLED_INSNS increases, more loops and larger loops may be unrolled, leading to potentially higher register pressure and larger code size. Fortunately, the IA-64 architecture possesses large register files and can sustain higher register pressure than other architectures. So we recommend

MAX_UNROLLED_INSNS to be set as 600. In future work, we will investigate a more sophisticated strategy that can also take register pressure into account.

4.4 Prefetching Arrays in Loops

Data prefetching techniques anticipates cache misses and issue fetches to the memory system in advance of the actual memory accesses. To provide a overlap between processing and memory accesses, computation continues while the prefetched data are being brought into the cache. Data prefetching is therefore complementary to data locality optimizations such as loop tiling and scalar replacement.

In GCC, the array elements in loops are prefetched at the RTL level. However, its prefetching algorithm is not effective on IA-64. The prefetching algorithm relies on a number of tunable parameters. Those relevant to this work are summarised below.

1. PREFETCH_BLOCK specifies the cache block size in bytes for the cache at a particular level. The default value is 32 bytes for the caches at all levels.
2. SIMULTANEOUS_PREFETCHES specifies the maximum number of prefetch instructions that can be inserted into an innermost loop. If more prefetch instructions are needed in an innermost loop, then no prefetch instructions will be issued at all for the loop. The default value on IA-64 is 6, which is equal to the maximum number of instructions that can be issued simultaneously on IA-64.
3. PREFETCH_BLOCKS_BEFORE_LOOP_MAX specifies the maximum of prefetch instructions inserted before a loop (to prefetch the cache blocks for the first few iterations of the loop). The default value is also 6.
4. PREFETCH_DENSE_MEM represents the so-called memory access density for a prefetch instruction. It refers to the ratio of the number of bytes actually accessed to the number of bytes prefetched in a prefetch instruction. In Itanium 2, the cache line sizes of its L1, L2 and L3 caches are 64 bytes, 128 bytes and 128 bytes, respectively. It is therefore possible that some data prefetched by a prefetch instruction may not be accessed. Thus, PREFETCH_DENSE_MEM reflects the effectiveness of a single prefetch instruction. The default value for this parameter is 220/256.

Our experimental evaluations show that the default values for the first three parameters are not reasonable. Figure 4 plots some statistics about prefetch instructions required inside loops in four SPECfp2000 benchmarks. A data point $(x, y\%)$ for a benchmark means that the percentage number of loops requiring x or fewer prefetch instructions in that benchmark is $y\%$. The statistics are obtained using GCC 3.5 at the optimization level "-O3 -funroll-loops -fprefetch-loop-arrays" with the alias analysis for FORTRAN, GIV optimization and loop unrolling incorporated. In `swim`, the number of prefetch instructions required by 81.00% of the loops is less than or equal to 6, but these loops account for a small portion of the execution time of the program. The loops that are responsible for the most of the execution time may require 7 or more prefetch instructions. For example, loop 100 in function CALC1, loop 200 in function CALC2 loop 300 in function CACL3 and loop 400 in function CACL3Z require 14, 20, 9 and 9 prefetch instructions, respectively. In `mgrid`, loop 600 in function PSINV and loop 800 in function RESID account for nearly all the execution time. They require 10 and 12 prefetch instructions, respectively. In all these loops requiring more than 6 prefetch instructions, no instructions will be actually inserted according to the semantics of SIMULTANEOUS_PREFETCHES.

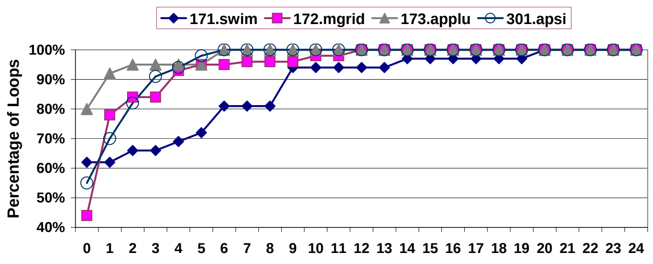


Fig. 4. Prefetch instructions required in four SPECfp2000 benchmarks (in GCC 3.5).

Therefore, it is too simplistic to use a uniform upper bound to limit the number of prefetch instructions issued in all loops. Some loops may need more prefetch instructions than others. Inserting too many prefetch instructions into a loop may result in performance degradation. However, such a situation can be alleviated by adopting rotating register allocation [3]. Unfortunately, such a scheme is not supported in GCC.

We have refined the prefetching algorithm in GCC for IA-64 as follows: All the default values are chosen by extensive benchmarking on an Itanium 2 system.

– First, we should allow more prefetch instructions to be issued on IA-64:

  PREFETCH_BLOCKS_BEFORE_LOOP_MAX = 12
  SIMULTANEOUS_PREFETCHES = 12

– Second, we introduce a new parameter, PMAX, which is used to determine the maximum number of prefetch instructions, that can be inserted inside a loop:

  $$\text{PMAX} = \text{MIN (SIMULTANEOUS\_PREFETCHES, NUM\_INSNS} \div 6).$$

  where NUM_INSNS is the number of instructions in the loop. If the number of prefetch instructions calculated are no large than PMAX, then all will be issued. Otherwise, the PMAX most effective prefetch instructions will be issued. Prefetch instruction F_1 is more effective than prefetch instruction F_2 if more array accesses can use the data prefetched by F_1 than that by F_2. That being equal, F_1 is more effective than F_2 if F_1 has a higher memory access density than F_2.
– Third, there are three levels of cache in the Itanium 2 processors. The L1 cache is only for integer values. The cache line sizes for L1, L2 and L3 caches are 64, 128 and 128 bytes, respectively. We set PREFETCH_BLOCK=64 for integer values and PREFETCH_BLOCK=128 for floating-point values. In addition, we use the prefetch instruction `lfetch` to cache integer values at the L1 cache and `lfetch.nt1` to cache floating-point values at the L2 cache. Our experimental results show that this third refinement leads to only slight performance improvements in a few benchmarks. Note that the statistical results shown in Figure 4 remain

nearly the same when the two different values for PREFETCH_BLOCK are used. This is because in GCC, the prefetch instructions required for an array access inside a loop is calculated as (STRIDE + PREFETCH_BLOCK - 1)/PREFETCH_BLOCK, where STRIDE is 8 bytes for almost all array accesses.

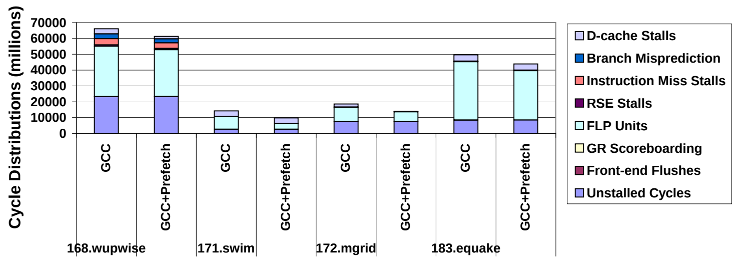


Fig. 5. Effects of the improved prefetching on Itanium cycle categories(in GCC 3.5).

Figure 5 illustrates the effectiveness of our improved prefetching algorithm. All four benchmarks are compiled under GCC at "-O3 -funroll-loops -fprefetch-loop-arrays" with alias analysis, GIV optimization and loop unrolling included. The percentage reductions in the D-cache (L1 cache) category for `wupwise`, `swim`, `mgrid` and `equake` are 51.66%, -5.356%, 84.59% and 0.99%, respectively. The percentage reductions in the FLP units for the same four benchmarks are 7.28%, 57.38%, 31.77% and 15.45%, respectively. This category includes the stalls caused by the register-register dependences and the stalls when instructions are waiting for the source operands from the memory subsystem. By prefetching array data more aggressively in computation-intensive loops, the memory stalls in these benchmarks have been reduced more significantly.

5 Experimental Results

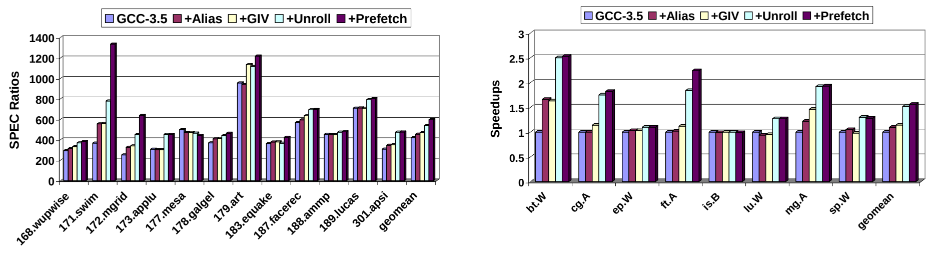
We have implemented our techniques in GCC 3.5 and GCC 4.0.0. We evaluate this work using SPECfp2000 and NAS benchmarks on a 1.0 GHz Itanium 2 system running Redhat Linux AS 2.1 with 2GB RAM. The system has a 16KB L1 instruction cache, a 16KB data cache, a 256KB L2 (unified) cache and a 3MB L3 (unified) cache. We have excluded the two SPECfp2000 benchmarks, `fma3d` and `sixtrack`, in our experiments since they cannot compile and run successfully under GCC 3.5. We present and discuss our results under GCC 3.5 and GCC 4.0.0 in two separate subsections.

5.1 GCC 3.5

Figure 6(a) illustrates the cumulative effects of our four techniques on improving the performance of SPECfp2000. "GCC-3.5" refers to the configuration under which all benchmarks are compiled using GCC 3.5 at "-O3 -funroll-loops -fprefetch-loop-arrays". "+Alias" stands for GCC 3.5 with the alias analysis for FORTRAN being included. In "+GIV", the GIV optimization is also enabled. In "+Unroll", our loop unrolling is also turned on. Finally, "+Prefetch" means that the optimization for prefetching arrays in loops is also turned on. Therefore, "+Prefetch" refers to GCC 3.5 with all our four techniques being enabled.

The performance of each benchmark is the (normalized) ratio of the run time of the benchmark to a SPEC-determined reference time. The ratio for SPECfp2000 is calculated as the geometric mean of the normalized ratios for all the benchmarks.

Before our optimizations are used, the ratio of SPECfp2000 is 420.7. Alias analysis helps lift the ratio to 455.1, resulting in a 8.2% performance increase for SPECfp2000. The GIV optimization pushes the ratio further to 470.1, which represents a net performance increase of 3.3%. Loop unrolling is the most effective. Once this optimization is turned on, the ratio of SPECfp2000 reaches 540.1. This optimization alone improves the SPECfp2000 performance by 14.9%. Finally, by prefetching arrays in loops, the ratio of SPECfp2000 climaxes to 596.4. This optimization is also effective since a net performance increase of 10.4% is observed. Our four optimizations have increased the ratio of SPECfp2000 from 420.7 to 596.4, resulting a performance increase of 41.8%. For SPECfp2000, loop unrolling and prefetching are the two most effective optimizations.



(a) SPECfp2000 (b) NAS

Fig. 6. Performance results of SPECfp and NAS benchmarks

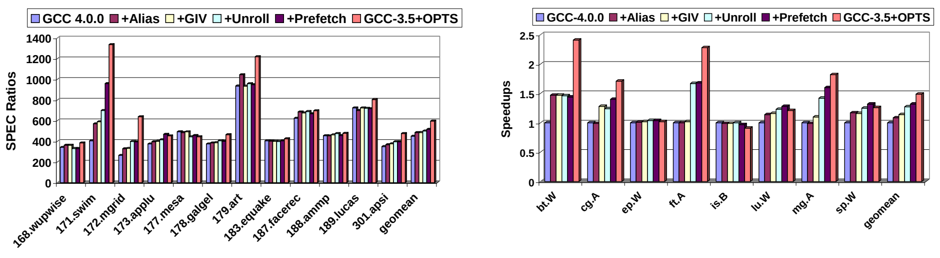
Figure 6(b) shows the performance improvements for the NAS benchmarks. The execution times of a benchmark under all configurations are normalized to that obtained under "GCC-3.5" (with our techniques turned off). Therefore, the Y-axis represents the performance speedups of our optimization configurations over "GCC-3.5". The performance increase for the entire benchmark suite under each configuration is taken as the geometric mean of the speedups of all the benchmarks under that configuration. The speedups for "+Alias", "+GIV", "+Unroll" and "+Prefetch" over "GCC-3.5" are 9.6%, 14.3%, 51.7% and 56.1%. Therefore, our four optimizations have resulted a 56.1% performance increase for the NAS benchmark suite. For these benchmarks, alias analysis and loop unrolling are the two most effective optimizations.

5.2 GCC 4.0.0

GCC 4.0.0 is the latest release of GCC, which includes (among others) the SWING modulo scheduler for software pipelining [7] and some loop transformations such as loop interchange. As we mentioned earlier, both are not applied on IA-64 for any SPECfp2000 or NAS benchmark. Therefore, they are not used in our experiments.

We have also implemented our techniques in GCC 4.0.0. Note that the part of this analysis for COMMON variables has already been committed in GCC 4.0.0. In GCC 4.0.0, loop unrolling does not split induction variables as it did in GCC 3.5. The lack of such a useful optimization has made our loop unrolling optimization less effective in GCC 4.0.0. (This "performance" bug may be fixed in future GCC releases.)

Figure 7(a) gives the performance results for SPECfp2000. With all our techniques in place, a performance increase of 14.7% is obtained. This result is less impressive compared to our performance achievements in GCC 3.5. The major reason is that loop unrolling that is the most effective in GCC 3.5 has not achieved its full potential due to the performance bug regarding the induction variable splitting we mentioned earlier. However, GCC 3.5 with our improvements incorporated, represented by the "GCC-3.5+OPTS" configuration, outperforms GCC 4.0.0 by 32.5%.



(a) SPECfp2000 (b) NAS

Fig. 7. Performance results of SPECfp2000 and NAS benchmarks.

Figure 7(b) shows the performance results for NAS benchmarks. Again, loop unrolling is not as effective as it was in GCC 3.5 for the reason explained earlier. However, our techniques have resulted in a performance increase of 32.0%. Alias analysis and loop unrolling are still the two most effective techniques. Finally, GCC 3.5 with our improvements incorporated, represented by the "GCC-3.5+OPTS" configuration, outperforms GCC 4.0.0 by 48.9%.

6 Related Work

There are a number of open-source compilers for the IA-64 family processors. The Open Research Compiler (ORC) [15] targets only the IA-64 family processors. There are frontends for C, C++ and FORTRAN. The openIMPACT compiler [14] is also designed for the IA-64 architecture alone. Its frontends for C and C++ are not completed yet. One of its design goals is to make openIMPACT fully compatible with GCC.

We have adopted GCC as a compiler platform for this ongoing research because GCC is a multi-language and multi-platform compiler. In addition, GCC is very portable and highly optimizing for a number of architectures. It is more mature than ORC and openIMPACT since ORC and openIMPACT can often fail to compile programs.

There are GNU projects on improving the performance of GCC on IA-64 [16]. However, little results have been included in the latest GCC 4.0.0 version. The SWING modulo scheduler for software pipelining [7] is included in GCC 4.0.0 but does not schedule any loops successfully on IA-64 according to our experimental evaluations.

This work describes four improvements in the current GCC framework for improving the performance of GCC on IA-64. Our improvements are simple but effective in boosting the performance of GCC significantly on IA-64.

Loop unrolling and induction variable optimizations are standard techniques employed in modern compilers [13]. Alias analysis is an important component of an optimizing compiler [2]. In GCC, alias analysis should be carried out not only at both its intermediate representations [6] but also at the frontends for specific programming languages. However, the alias analysis component for FORTRAN programs in GCC is weak. This work demonstrates that a simple intraprocedural alias analysis can improve the performance of FORTRAN programs quite significantly.

Software data prefetching [1, 12, 17] works by bringing in data from memory well before it is needed by a memory operation. This hides the cache miss latencies for data accesses and thus improves performance. This optimization works the best for programs that have array accesses in which data access patterns are regular. In such cases, it is possible to predict ahead of time the cache lines that need to be brought from memory. Some work has been done on data prefetching for non-array accesses as well [10, 11]. Data prefetching represents one of the most effective optimizations in commercial compilers [4, 3] for IA-64. For IA-64, adopting rotating register allocation to aid data prefetching in GCC is attractive.

7 Conclusion

In this paper, we describe some progress we have made on improving the floating-point performance of GCC on the IA-64 architecture. Our four improvements are simple but effective. We have implemented our improvements in both GCC 3.5 and GCC 4.0.0. Our experimental results show significant performance increases for both SPECfp2000 and NAS benchmark programs. The part of our alias analysis regarding COMMON variables has been committed in GCC 4.0.0.

Compared to Intel's icc, GCC still falls behind in terms of its performance on IA-64. The following three kinds of optimizations are critical for icc's performance advantages: loop transformations such as loop interchange, loop distribution, loop fusion and loop tiling, software pipelining and interprocedural optimizations. A preliminary implementation for optimizing nested loops for GCC has been developed [5]. However, its loop interchange transformation cannot even successfully interchange any loops on IA-64. The SWING modulo scheduler for software pipelining has also been incorporated in GCC 4.0.0 [7]. Due to the imprecision of the dependence analysis in GCC 4.0.0, the SWING modulo scheduler can hardly make a successful schedule on IA-64. In GCC 4.0.0, the function inlining remains to be the only interprocedural optimization supported. We plan to make contributions in these areas in future work. We strike, as our long-term goal, to achieve performance on IA-64 comparable to that by commercial compilers while retaining the improved GCC as an open-source, portable, multi-language and multi-platform compiler.

References

1. D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. *In Proceedings of the Fourth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 40–52, 1991.
2. S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. *In The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–24, 1998.
3. G. Doshi, R. Krishnaiyer, and K. Muthukumar. Optimizing software data pre-fetches with rotating registers. *In Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 257–267, September 2001.
4. C. Dulong, R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, J. Ng, and D. Sehr. An overview of the intel ia-64 compiler. *Intel Technology Journal*, Q4 1999.
5. David Edelsohn. High-level loop optimizations for gcc. *In Proceedings of the 2004 GCC Developers' Summit*, pages 37–54, 2004.
6. Sanjiv K. Gupta and Naveen Sharma. Alias analysis for intermediate code. *In Proceedings of the GCC Developers' Summit 2003*, pages 71–78, May 2003.
7. Mostafa Hagog. Swing modulo scheduling for gcc. *In Proceedings of the 2004 GCC Developers' Summit*, pages 55–64, 2004.
8. J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the ia-64 architecture. *IEEE Mirco*, pages 12–23, Sept-Oct 2000.
9. Intel. *Intel IA-64 Architecture Software Developer's Manual*, volume 1. October 2002.
10. M.H. Lipasti, W.J. Schmidt, S.R. Kunkel, and R.R. Roediger. Spaid:software prefetching in pointer and call intensive environments. *In Proc 28th International Symposium on Microarchitecture*, pages 231–236, Nov 1995.
11. C.K. Luk and T.C. Mowry. Compiler-based prefetching for recursive data structures. *In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, September 1996.
12. T.C. Mowry, M.S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. *In Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.
13. Steven S. Muchnick. *Advanced Compiler Design Implementation*. Academic Press, 1997.
14. openIMPACT. http://www.gelato.uiuc.edu.
15. ORC, 2003. http://ipf-orc.sourceforge.net.
16. The GCC Summit Participants. Projects to improve performance on ia-64. *http://www.ia64-linux.org/compilers/gcc_summit.html*, June 2001.
17. V. Santhanam, E. Gornish, and W. Hsu. Data prefetching on the hp pa-8000. *In Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 264–273, June 1997.
18. Richard M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, Inc., 1994.
19. Sverre Jarp. A methodology for using the Itanium 2 performance counters for bottleneck analysis, 2002. http://www.gelato.org/pdf/Performance_counters_final.pdf.
20. Kejia Zhao, Canqun Yang, Lifang Zeng, and Hongbing Luo. *Analyzing and Porting GNU GCC.* Technical Report, National University of Defense Technology, P. R. China, 1997.