Reprinted from the

Proceedings of the GCC Developers' Summit

June 28th–30th, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon Incorporated*
C. Craig Ross, *Linux Symposium*

Review Committee

Ben Elliston, *IBM*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Red Hat*
Gerald Pfeifer, *Novell*
Ian Lance Taylor, *Google*
C. Craig Ross, *Linux Symposium*
Andrew J. Hutton, *Steamballoon Incorporated*

Proceedings Formatting Team

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

Improved Superblock Optimization in GCC

Robert Kidd and Wen-mei Hwu
Center for High Performance and Reliable Computing
University of Illinois at Urbana-Champaign
{rkidd,hwu}@crhc.uiuc.edu

Abstract

Superblock scheduling is a common technique to increase the level of instruction level parallelism (ILP) in generated code. Compared to a basic block, the Superblock gives an optimizer or scheduler a longer range over which instructions can be moved. The bookkeeping necessary to execute that move is less than would be necessary inside an arbitrary trace region. Additionally, the process of forming Superblocks generates more instructions that are eligible for movement. These factors combine to produce a significant increase in the ILP in a section of code.

By identifying the key feature of Superblock formation that allows this increase in ILP, we can generalize the concept to describe a class of similar optimizations. We refer to techniques in this class as *structural* techniques. Combining several optimizations in this class with aggressive classical optimization has been shown in the OpenIMPACT compiler to be particularly useful in developing ILP when compiling for the Itanium processor.

As a motivation for our work, we present an investigation into the value of structural compilation in the OpenIMPACT compiler. In this domain, structural techniques have been credited with a 10% to 13% increase in code performance over a compiler that implements only classical optimizations.

As a first step toward developing structural compilation techniques in GCC, we implemented Superblock formation at the Tree-SSA level. By performing structural transformations early, we give the compiler's high level optimizers an opportunity to specialize the transformed program, thereby cultivating higher levels of ILP. The early results of this modification are mixed, with some benchmarks improving and others slowing. In this paper, we present details on our implementation and study the effects of this structural transformation on later optimizations. Through this, we hope to motivate future work to implement and improve optimizations that can take advantage of the transformed control flow.

1 Introduction

As an EPIC (Explicitly Parallel Instruction Computing) processor, the Intel Itanium Processor Family relies heavily on the compiler to extract performance from a program. The architecture provides a large number of functional units, but the hardware does not dynamically schedule instructions to discover instruction level parallelism. Instead, the compiler

must find instructions that can execute in parallel during its code generation and scheduling stage. Within the context of a traditional optimizing compiler, control flow presents barriers that make it difficult to statically generate a parallel schedule. Superblock formation [3] is one technique to overcome the restrictions of control flow.

Originally, Superblock formation was developed solely to support instruction scheduling. In this traditional method, a Superblock is constructed using trace formation and tail duplication. The result of Superblock formation is a long single-entry, multiple-exit chain of basic blocks. This Superblock is then passed to a variant of a list scheduler that is capable of moving instructions across basic block boundaries. Compared to trace scheduling [1], the lack of side entrances into a Superblock simplifies the task of moving instructions between basic blocks.

Superblock formation is useful for more than instruction scheduling. Tail duplication eliminates all but one of the control flow paths into a basic block. As a result, much tighter bounds can be drawn on the state of variables at the block entrance. Optimizations such as constant propagation can be profitably applied to the duplicated tail to specialize the block, counteracting the code expansion inherent in Superblock formation.

In the OpenIMPACT compiler [6], Superblock formation is one of a class of structural compilation techniques. Other members of this class include function inlining and hyperblock formation. These optimizations use profile feedback to radically alter the control flow graph (CFG) to produce straight line sections for the typical course of execution. Traditional optimizations are then applied to this CFG to specialize blocks and discover ILP. While this technique does entail a large amount of code duplication, it packs the useful instructions into a tight schedule, resulting in little increase in instruction cache pressure.

Within GCC, we modified the pre-existing RTL Superblock formation pass to work at the Tree-SSA level. The overall performance change due to this modification is minimal. Certain benchmarks run faster, while others run slower. However, this patch gives us a starting point to investigate the possibility of applying structural optimization techniques within GCC. In the following sections, we will discuss the effects, good and bad, of the patch and suggest a future direction based on our experience with OpenIMPACT.

The primary goal of our work is to improve the performance of code compiled by GCC for the Itanium processor. However, we believe that the structural compilation model will also work well on traditional superscalar processors. The code specialization derived from traditional optimizers should apply to any architecture.

2 The Superblock and structural compilation

Although it has been throughly covered in literature, it is useful to review here the process through which a Superblock is constructed. This process is illustrated in Figure 1. Starting with a control flow graph annotated with profile weights (part (a)), the *trace formation* pass determines the typical path of execution. This pass constructs the hot trace by following the typical path of execution until execution frequency falls below some threshold or a loop back edge is encountered. The resulting trace is highlighted with a dotted outline in Figure 1(b). After constructing the trace, the *tail duplication* pass eliminates side entrances. If a basic block on the trace has more than one predecessor, as is the case for block z, that block is

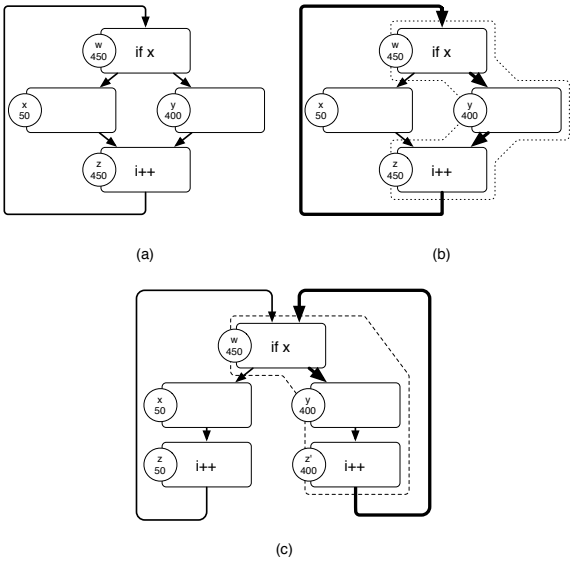

Figure 1: Superblock formation process

copied and the duplicate inserted into the trace. The sole predecessor of the duplicate will be on the trace, as seen in part (c). The result of this process is a Superblock, a series of basic blocks that has a single entrance at the top and one or more side exits. This is indicated by the dashed line in Figure 1(c).

Structural compilation [5] is a generalization of Superblock-based optimization. Structural optimizations share a common attribute in that they target side entrances (join points) of a frequently executed trace. Inside the join block and below, program context is blurred as the compiler must assume that any predecessor may have executed before the join block. Duplicating the join block and its children onto the hot trace has the effect of extending the context of the trace into the duplicated block. Specialization of the duplicated code through classical optimizations reduces the number of instructions on the trace, and use of features such as speculation extend the range over which instructions can move. The end result for the typical path of execution is a shorter schedule with higher levels of ILP.

In the structural model, code duplication is performed early in the compilation path. Code expansion limits are raised to allow more code duplication. The expansion must be done somewhat speculatively, as it is difficult to predict the precise effect later optimizations will have, given a certain level of duplication. Therefore, a large amount of code duplication is inherent to the structural model. The code generated using this model is bigger than that generated with a traditional model, so one might expect a degradation in instruction cache efficiency. However, the expansion is mitigated by two factors. First, the Itanium processor provides large caches, which help offset the increase in code size. The second factor, which applies to all architectures, is that the code duplicated by these techniques should appear outside the path of typical execution. By specializing for the typical path, we reduce the number of redundant instructions and pack the useful ones more closely. This reduces the number of cache lines that need to be fetched and increases the number of useful instructions per line.

The compilation method used in GCC to this point more closely resembles what we term the *incremental* approach. In this method, traditional global scheduling techniques are evolved and refined to deliver higher levels of ILP. Control flow may be altered, but it is done in response to the needs of the optimizer or scheduler. The overall CFG tends to remain the same, and can restrict optimization opportunities. An analogy with simulated annealing is apt. The incremental model is a well tested and reliable method to arrive at an optimal point in the range of possible schedules. However, in many cases, this optimal schedule is a local minimum. By applying structural techniques, we hope to move the optimizer's starting point to a place where a more optimal schedule can be found.

Benchmark	g-no-spec	I-CL	I-NS	I-CS
164.gzip	506	602	677	752
175.vpr	498	607	644	719
181.mcf	257	332	330	341
186.crafty	591	646	677	704
197.parser	494	520	523	541
252.eon	517	364	428	429
254.gap	421	558	573	599
256.bzip2	426	652	658	698
300.twolf	553	724	830	921
geomean	462	540	575	609
geomean2	456	567	596	637

Heading	Compiler version	options
g-no-spec	GCC as of 3/6/2006	-O3, profile feedback enabled, no speculation
I-CL	OpenIMPACT	classical optimizations only
I-NS	OpenIMPACT	structural opti, no speculation, pointer analysis and modulo scheduling disabled for 252.eon
I-CS	OpenIMPACT	control speculation, otherwise like I-NS
Scores generated on an Itanium 2 1.0 GHz, 1.5M L3 cache		

Table 1: Comparison of classical and structural optimizing compilers

Table 1 presents estimated[1] SPECint2000 scores to illustrate the benefit that a structural optimization pass can have on code performance. In this table, all benchmarks have been compiled using profile feedback.

The g-no-spec column is our baseline GCC configuration. This is a recent version of GCC mainline that lacks speculation support. This configuration includes the RTL level Superblock formation pass[2]. I-CL is our baseline OpenIMPACT configuration. Classical optimizations are run, but structural techniques are disabled. This configuration is roughly equivalent to g-no-spec. I-NS turns on structural transformation, and I-CS turns on control speculation.

The geomean row shows the geometric mean for all nine benchmarks. Geomean2 excludes 252.eon from the comparison, as GCC's score is inflated relative to OpenIMPACT for the purposes of this paper. OpenIMPACT has a long history as a C compiler, but C++ support has only been added recently. Even now, C++ support is implemented by lowering the incoming code to C and compiling. This is functional, but inefficient. As a proper C++ compiler, GCC has a distinct advantage over OpenIMPACT on 252.eon that is outside the scope of this paper.

A comparison of the I-CL and g-no-spec columns shows that OpenIMPACT's classical optimization path improves performance approximately 24% over GCC. This can be attributed to better alias analysis in OpenIMPACT and more aggressive settings on the clas-

[1]These numbers are the result of runs on real hardware, but do not reflect a rigorous SPEC run. We have followed SPEC's run rules with respect to training/reference inputs and consistency of optimization settings, but we have been unable to complete a full run of the suite. We therefore label these results "estimated" as per SPEC's rules for research use.

[2]Results from the Tree-SSA level Superblock formation pass are presented in Section 3.

sical optimizers. Comparing I-CL and I-NS, we see that turning on structural transforms in OpenIMPACT gives a 5% improvement in performance. We believe 5% to be a reasonable estimate for the performance benefit possible in GCC with the addition of structural transformation.

The I-CS column shows the effect of combining control speculation and structural transforms in OpenIMPACT. We see a 12% to 13% increase in performance over the classical configuration when these two features are combined. Our Superblock work should therefore fit in nicely with other work in progress to add speculation support to the compiler.

Although it is not useful as a comparison between GCC and OpenIMPACT, it is interesting to note that within OpenIMPACT, the structural technique is particularly useful for 252.eon. In this case, indirect call profiling and function inlining combine to reduce the cost of virtual method calls. Although OpenIMPACT does not support static C++ optimizations such as class hierarchy analysis, it is able to approximate them in some cases through structural transformation. Even with proper C++ support, structural transformation will likely still be beneficial for heavily object-oriented code.

3 Technical details and performance results

At the RTL level, GCC has included a Superblock formation pass for some time [2]. This pass prepares Superblocks for the Haifa interblock scheduler, but runs after most of GCC's optimization passes. Structural compilation requires this transformation to happen early in the compiler; to achieve this, we implemented a Superblock formation pass at the Tree-SSA level.

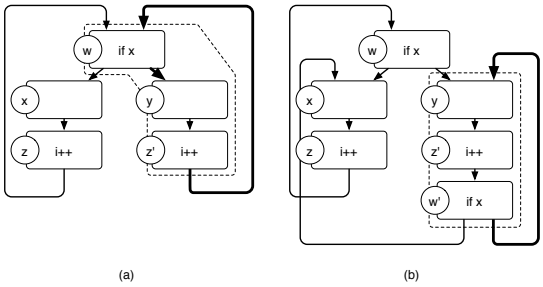

Figure 2: Loop header duplication to form a simple loop from a Superblock

Because the CFG manipulation API is shared between RTL and Tree-SSA, the modifications necessary to run Superblock formation at the Tree level were minor. The primary change necessary was to add SSA variables generated in the duplicated tail to the ϕ-nodes for its successor blocks. We also adjusted the trace formation routine so that it generates Superblock loops that are of a simple form that can be processed by the loop optimizers. This is illustrated in Figure 2.

Figure 2(a) shows the problem that can occur when forming a Superblock inside a loop. By tail duplicating block z to form z', we created a Superblock for the common case inside the loop. However, because tail duplication stopped when it found the loop backedge, we were forced to add a second control flow arc back to the loop header. The loop is no longer in a simple form that can be processed by GCC's loop optimizers. If trace formation instead follows the backedge one time and stops after duplicating the loop header, our Superblock forms a simple loop, as in Figure 2(b). The form of this simple loop increases the effectiveness of later optimizations, as will be seen in Section 4.2.

The final change we made was to remove the Superblock layout pass. We leave code layout to the basic block reordering stage later in RTL.

The Tree-SSA Superblock formation executes immediately after SSA form is constructed and before any optimizers. Because tail duplication has the potential to create new pointer loads and stores, it is possible that formation will interfere with alias analysis. We have tested building Superblocks before and after alias analysis, but neither configuration is noticeably better than the other.

Our comparison baseline is GCC mainline revision 112576, which supports Superblock formation at the RTL level. Against this baseline, we compare a build of the ia64-improvements branch. This branch includes the changes to mainline through revision 112576, but uses the Tree-SSA level Superblock formation pass. We run both versions of GCC with the -O3 flag and with profile feedback enabled. Both versions support speculation on Itanium. For the ia64-improvements branch, we set an aggressive Superblock expansion limit of 300% as opposed to the 100% limit used with the RTL level pass. We ran performance numbers on three architectures: x86, x86_64, and Itanium. Machine specifications are listed in Table 2.

Tables 3 and 4 show the estimated change in performance for selected SPEC2000 benchmarks. For each architecture, we present the score for GCC mainline (stock), the score for the ia64-improvements branch (SB), and the percent difference between the two. We have not yet fully analysed the cause of the slight performance degradation on x86_64, but we believe this processor to be more sensitive to code scheduling than x86. Forming Superblocks at the Tree-SSA level is beneficial for x86 and on Itanium for the floating point benchmarks. Superblock formation tends to produce simpler, straighter loops that are more palatable to loop optimizers. This helps explain the performance improvement in loop-intensive floating point code.

For 191.fma3d on x86, Superblock formation gives a significant performance improvement. This improvement is due to a drastic reduction in the time spent executing one function, `platq_stress_integration`. The prologue of this function sets up a number of variables using `MIN` and `MAX` statements. These statements are biased, so they become simple assignments inside the Superblock. Constant propagation can then simplify the body of the function.

For integer codes, the results are more mixed. These control-intensive benchmarks are where we expect to see a significant performance improvement on Itanium, and yet we see a negligible change. This is not entirely unexpected. As we have observed in OpenIMPACT, the combined efforts of several structural techniques are often necessary to get a significant performance improvement. These passes must push the program's CFG to a critical point where optimizations can radically specialize the typical path of execution. It is encouraging to note that 181.mcf, 186.crafty, and 256.bzip2 do improve. 186.crafty is an extremely control-intensive chess simulator, and the improvement here replicates a result from OpenIMPACT detailed in [5]. The improvement in 181.mcf, a memory-intensive benchmark, can be attributed to the combined effect of Superblock formation and data speculation. This benefit of combining speculation and structural transformation has also been observed in OpenIMPACT. These results strongly suggest that results from OpenIMPACT will apply to GCC as development progresses. We will expand on effect of Superblock formation on 256.bzip2 in Section 4.2.

Finally, Table 5 compares the size of the benchmark executable across the two compiler configurations. Even with an aggressive Superblock expansion limit, executable size does not significantly increase. By duplicating blocks, Superblock formation gives the op-

Configuration	Processor	Operating System
ia64	Itanium 2, 1.6 GHz, 6 MB L3	Linux 2.6.8, LP64
x86_64	Athlon 2800+, 1.8 GHz, 512 KB L2	Linux 2.6.12, LP64
x86	Xeon, 2.4 GHz, 512 KB L2	Linux 2.4.18, LP32

Table 2: Machine configurations

	ia64			x86_64			x86		
Benchmark	stock	SB	%	stock	SB	%	stock	SB	%
164.gzip	810	810	0.00%	914	850	-7.00%	676	696	2.96%
175.vpr	929	923	-0.65%	771	760	-1.43%	484	472	-2.48%
175.gcc				1032	1033	0.10%	922	947	2.71%
181.mcf	682	706	3.52%	587	587	0.00%	535	544	1.68%
186.crafty	942	983	4.35%	1531	1586	3.59%	460	477	3.70%
197.parser	901	901	0.00%	859	864	0.58%	731	763	4.38%
252.eon	792	785	-0.88%				831	790	-4.93%
254.gap	651	651	0.00%	999	1004	0.50%			
255.vortex				1454	1532	5.36%			
256.bzip2	798	825	3.38%	897	889	-0.89%	580	585	0.8%
300.twolf	1039	944	-9.14%	792	793	0.13%	606	611	0.83%
geomean	830	830	-0.01%	947	947	0.05%	631	638	1.04%

Table 3: Estimated change in SPECint2000 performance

	ia64			x86_64			x86		
Benchmark	stock	SB	%	stock	SB	%	stock	SB	%
168.wupwise	605	578	-4.46%	1186	1140	-3.88%	903	918	1.66%
171.swim	773	804	4.01%	1142	1138	-0.35%	634	633	-0.16%
172.mgrid	346	347	0.29%	755	751	-0.53%	479	478	-0.21%
173.applu	519	538	3.66%	900	895	-0.56%	666	665	-0.15%
177.mesa	976	986	1.02%	1394	1383	-0.79%	483	487	0.83%
179.art	2716	2730	0.52%	777	785	-1.03%	272	268	-1.47%
183.equake	511	500	-2.15%	1085	1097	1.11%	960	967	0.73%
187.facerec	583	602	3.26%	791	772	-2.40%	477	479	0.42%
188.ammp	847	868	2.48%	882	884	0.23%	400	391	-2.25%
189.lucas	815	864	6.01%	1220	1200	-1.64%	546	538	-1.47%
191.fma3d	294	294	0.00%	870	874	0.46%	464	525	13.15%
200.sixtrack	371	368	-0.81%	451	449	-0.44%	429	431	0.47%
301.apsi	580	579	-0.17%	839	834	-0.60%	502	497	-1.00%
geomean	638	644	1.01%	912	906	-0.65%	527	531	0.75%

Table 4: Estimated change in SPECfp2000 performance

	int				fp		
Benchmark	stock	SB	%	Benchmark	stock	SB	%
164.gzip	1069787	1071587	0.17%	168.wupwise	1629360	1633480	0.25%
175.vpr	1310609	1307185	-0.26%	171.swim	1590538	1591482	0.06%
176.gcc	5836321	5749873	-1.48%	172.mgrid	1588244	1488644	0.03%
181.mcf	942326	942566	0.03%	173.applu	1691129	1691929	0.05%
186.crafty	1413018	1505658	-0.52%	177.mesa	2444067	2435987	-0.33%
252.eon	6984346	6986010	0.02%	179.arg	1016630	1020022	0.33%
254.gap	2253387	2247859	-0.25%	183.equake	1020942	1015102	-0.57%
255.vortex	2398115	2366931	-1.30%	187.facerec	1804178	1806962	0.15%
256.bzip2	2398115	2366931	-0.12%	198.ammp	1387201	1388161	0.07%
300.twolf	1519425	1516401	-0.20%	189.lucas	1677890	1680458	0.15%
				191.fma3d	4125512	4124912	-0.01%
				200.sixtrack	3890478	3887366	-0.08%
				301.apsi	1897353	1900977	0.19%

Table 5: Effect of Superblock formation on executable size (in bytes) for ia64

timizers more opportunity to eliminate instructions, controlling code expansion and increasing performance.

4 Analysis of performance change

On Itanium, 256.bzip2 and 300.twolf showed a significant change in performance. This section delves deeper into these two benchmarks to determine why performance improved or diminished.

We analysed the benchmarks using the i2prof.pl and q-tools packages. These packages use HP's libpfm library and pfmon utility to retrieve values from the Itanium performance counters to analyse.

i2prof.pl is a Perl script that wraps the pfmon utility to record raw counter values for the entire run of a program. Various performance metrics are determined from these counters, but they apply to the entire program and cannot be attributed to individual functions or lines.

```
for ( node = list; node;
      node = node->next ) {
   a = ...
   if (condition)
      b = ...
   else
      b = a
   *arg += b ...
}
```

Figure 3: Kernel from `new_dbox_a`

Q-tools provides the q-syscollect utility, which performs statistical sampling using the Itanium performance monitoring unit. At a specified interval, the program is interrupted and the program counter (PC) of the instruction triggering the event being monitored is recorded. Q-tools also includes the qprof program, which generates a gprof-style per-function report from the per-PC data.

4.1 300.twolf

300.twolf slowed by 9% when Superblock formation was performed at the Tree-SSA level. A

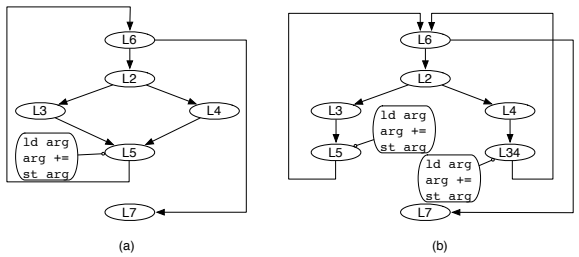

Figure 4: Structure of `new_dbox_a` before and after Superblock formation

comparison of execution profiles showed that much of the extra execution time could be attributed to the function `new_dbox_a`. This function is called approximately 120 million times during execution. Figure 3 illustrates the kernel of this function.

In this loop, `arg` is an integer pointer argument used to return a value from the function. The integer `arg` points to is updated, but the value of `arg` is not changed inside this function. Because of this, GCC's partial redundancy elimination (PRE) stage is able to move the load and store of `arg` out of the for loop.

Figure 4 shows the control flow graph of this kernel before and after Superblock formation. Tail duplication moves a copy of the update of `arg` onto both sides of the biased `if` statement, shown in Figure 4(b). In this case, the Superblock formation pass did not duplicate the loop header as we would expect. The cause of this is being investigated. If Superblock formation had duplicated the loop header, we would expect the PRE pass to move the load out of the more frequently executed side of this loop. It is not yet clear whether GCC's alias analysis framework is strong enough to completely remove the load from the Superblock loop. We plan to investigate further the interaction between Superblock formation and alias analysis.

```
L6:  j = 0;
     tmp = yy[j];
L8:  while (ll_i != tmp) {
L7:   j++;
      tmp2 = tmp;
      tmp = yy[j];
      yy[j] = tmp2;
     }
L9:  yy[0] = tmp;

     if (j == 0)
L10:  /* do something */
      /* yy is not touched */
     else
L11:  /* do something else */
      /* yy is not touched */
      /* L12 and L20 appear here */
L21: ...
```

Figure 5: The core of `generateMTFValues` from 256.bzip2

4.2 256.bzip2

Performance of 256.bzip2 increased by approximately 3% with the addition of the Tree-SSA Superblock pass. The execution profile did not show a significant difference in the run time of any one function, so we used i2prof.pl to collect overall performance statistics for the program. These statistics showed that the Superblock version experienced fewer L1D cache misses than the standard version. Sampling the `L1D_READ_MISSES_ALL` counter with q-syscollect, we were able to locate a loop in `generateMTFValues` that experienced a decreased number of cache stalls.

The loop in question is shown in Figure 5. This loop searches for a specific character in the array `y`, determines the index `j` of that character, rotates elements 0 through `j - 1` to positions 1 through `j`, and writes the desired character to element 0. The important feature to note is that this loop does a number of single

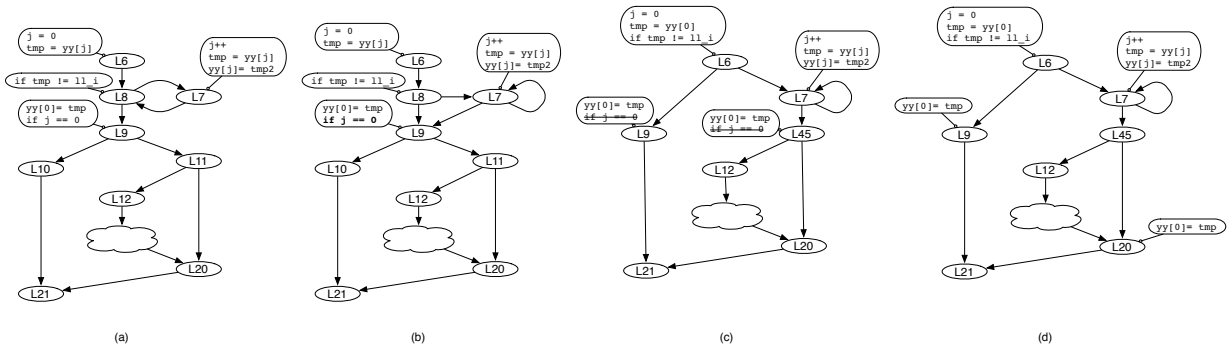

Figure 6: Interaction of structural techniques and traditional optimizations

byte loads and stores to the character array. In certain circumstances, scheduling two stores to nearby addresses within a couple cycles of one another can trigger an L1D stall on the Itanium 2.

This stall is due to a quirk in the design of the L1D cache on the Itanium 2. Although the processor advertises two store ports, in reality, the L1D cache is only pseudo-dual ported for stores [4]. Cache lines are split into eight single ported banks. Store coalescing hardware helps mitigate the penalty that would otherwise be associated with scheduling multiple stores to sequential addresses. However, if two stores that cannot be coalesced attempt to access the same bank, the younger one will be forced to stall. This appears in the benchmark compiled by GCC mainline. The store to `yy[0]` in L9 is scheduled one cycle after the store to `yy[j]` inside the loop. When the loop exits, the store in L9 may stall if `yy[0]` uses the same bank as `yy[j]`. Structural compilation transforms the CFG enough that the store can be moved many cycles later in the schedule, eliminating this stall.

Figure 6 shows the evolution of `generateMTFValues` as it is processed by the optimizers. We have annotated significant lines of code onto their corresponding CFG nodes. Figure 6(a) shows the core of `generateMTFValues` immediately after SSA form is constructed. Blocks L7 and L8 form the rotate loop, which cycles zero or more times. As the loop exits, L7 writes the `j`th element of the array. One cycle later, L9 writes into element 0 of the array and branches based on the value of `j`. If `j` is 0, L10 falls through to the rest of the function. Otherwise, L11 leads into a complex block of code.

Figure 6(b) shows the function after Superblock formation. Tail duplication copied the header of the L7-L8 loop so that L7 now forms a single block loop. We now have two flows into L9. Along the L8-L9 arc, `j` will always be 0. Along L7-L9, `j` will be non-zero. The original purpose of L9 was to determine whether the loop iterates, and by duplicating L8, we have made L9 redundant.

Figure 6(c) shows the kernel after constant- and value range propagation (VRP). VRP duplicates block L9 to make L45. VRP then propagates the value of `j` from L6 to L9 and from L7 to L45. This, in turn, allows for the elimination of the `if` statements in L9 and L45. None of the code in the subgraph headed by L12 uses the array `yy`, so in part (d), store sinking is able to move the write to `yy[0]` to L20. When this code is finally scheduled, the write to `yy[0]` doesn't occur until at least 12 cycles after the loop body, eliminating the stall.

5 Future work

Fully implementing the structural compilation model in GCC will be more a matter of tuning pieces already written than writing new code. At this point, Tree-SSA optimizers should generally be capable of moving instructions across basic blocks. There are already several structural-style passes written at the Tree-SSA level, such as the loop unroller. Study into moving these types of passes forward in the compilation order would be worthwhile.

Parameter and heuristic tuning is a matter that still needs to be addressed. Many of GCC's parameters governing code expansion are set conservatively relative to OpenIMPACT. If code expansion is done early enough, these can be set quite aggressively without adversely affecting the instruction cache performance of the generated code. Heuristics in the loop unroller refuse to unroll loops containing control flow for fear of increasing the number of branch mispredicts. Early unrolling combined with predication support may make unrolling such loops profitable, and so heuristics like these should be revisited.

Other code expanding transforms, such as branch target expansion, could be easily implemented within GCC. It would also be useful to investigate running multiple rounds of expansion. A Superblock-unroll-Superblock sequence could potentially give a very nice straight code sequence with high levels of ILP.

6 Conclusion

Although it has not yet demonstrated an overall performance improvement for Itanium, the Tree-SSA Superblock formation pass holds promise. We can already see an improvement in certain benchmarks, such as 186.crafty and 256.bzip2 due to the structural compilation model. Implementing additional early structural transformation passes will give optimizers more freedom to move and simplify program code. At the same time, we must ensure that the optimization and analysis passes can accept and use the modified control flow. When the transformation and optimization stages are fully compatible, we expect to replicate the consistent, positive performance results from the OpenIMPACT compiler.

7 Acknowledgments

We would like to thank the members of the OpenIMPACT research group and the GCC community for their help in this work. We would also like to thank the Gelato federation for its support of this project.

References

[1] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[2] Jan Hubička. Profile driven optimizations in GCC. *GCC Summit*, 2005.

[3] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.

[4] Intel Corporation. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization, Document Number 251110-003*, May 2004.

[5] J. W. Sias, S.-Z. Ueng, G. A. Kent, I. M. Steiner, E. M. Nystrom, and W. W. Hwu. Field testing IMPACT EPIC research results in Itanium 2. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 26–39, June 2004.

[6] UIUC OpenIMPACT Effort. The OpenIMPACT IA-64 Compiler. `http://gelato.uiuc.edu/`.