A Transparent Method for Correlating Profiles with Source Programs

Eugene Albert
Gem Compiler Group
Compaq Computer Corporation
110 Spitbrook Road
Nashua, NH 03062
Gene.Albert@compaq.com

1 Introduction

Feedback directed optimizations can improve the performance of a wide range of applications. However, using feedback to build large applications is cumbersome. Applications must be built twice, once to generate the executable from which the profile is to be generated and once to make use of the profile. Furthermore, the executable to be profiled must be built in a manner that allows the compiler to relate the profile to the basic block structure of the source program. For example, the executable to be profiled may have to be built without optimization. This alone can be enough to discourage the use of feedback altogether in a production environment.

This paper describes a transparent technique used in Compaq's GEM compiler back-end [1], which greatly simplifies profile correlation. GEM is the common optimizer and code generator used by Compaq's C, C++, and Fortran Alpha [2] compilers. With this method, GEM can make use of profiles generated from fully optimized, production quality applications. This eliminates the need for a special build, making feedback optimization easier to use. The technique is transparent in the sense that it does not perturb the code generated by the compiler.

Section 2 of this paper explores the various methods by which compilers correlate profiles with source programs. It focuses on the method that was in place in GEM compilers before the implementation of the new method. Section 3 describes the new method. Section 4 evaluates the efficacy of this method on the Spec 95 integer benchmarks [3]. Section 5 is a summary, placing this method among a family of efforts designed to make feedback directed optimization easier to use and more effective on Compaq's Alpha platforms.

2 Correlating profiles with source programs

The GEM feedback system uses pixie-format profiles for its data. The pixie format partitions the training executable into an array of basic blocks and associates counts with each basic block. The compiler examines the profile data and matches the basic blocks described by that data with the basic blocks of the initial internal representation (IR) of the source program. This correlation is critical. No matter how representative the profile data is of a typical workload, feedback directed optimizations will be ineffective if the correlation is seriously flawed.

As first implemented, profile correlation in GEM compilers required that the training executable be built without optimization. With this method, the goal during the training build is to reproduce the basic block structure of the initial IR in the training executable. Source line information embedded in the executable for debugging is used to assist in the match process. This method gives good results, but is fragile. The slightest rearrangement of blocks by the compiler can cause the correlation to degrade, significantly reducing the benefit of the feedback directed optimizations. Because of this, this simple correlation method requires that the training run turn off optimization.

Another commonly used method is for the compiler to place counters in the initial IR during the build of the training executable. In this case the profile is derived from these counters, and no match is needed (the profile is a direct profile of the initial IR). However, this too requires a special purpose build of the application. The counters will greatly slow down the executable, perhaps even perturbing its behavior if there are time-critical aspects to the application.

In contrast to either of these methods, the new method, which makes use of a compilation database, can make use of a profile generated from a fully optimized, untooled (from the point of view of the compiler) executable. This process is transparent in the sense that it does not affect the optimization and code generation decisions made by the compiler in building the training executable.

3 The basic block compilation database

With the new source correlation method, GEM generates information about the relationship between the basic blocks in the executable and the basic blocks of the source program. This information is created during the training build, preserved by the compiler, and then used in the match during the feedback build.

Other than the creation of the database, compilation is unaffected by the intent to profile the output. In particular, the executable created during the training build can be fully optimized and need not contain any compiler-introduced counters. The actual profiling step may include inserting counters in the executable, e.g., by pixie, or a sampling tool might be used that does not require changes to the executable, such as DCPI [4]. (In the latter case, the data is transformed from DCPI's internal format to pixie's format.) However, the profiling process is transparent to the compiler.

The database identifies the basic blocks in the training executable and the basic blocks in the source program, but does not contain any information about the graph structure among the source blocks or object blocks. The intent is not to remember the graph transformations performed by the compiler, as has been done in efforts to debug optimized code [5]. In fact, predecessor/successor relationships and information derived from these relationships (e.g., loop structures) are not present in the database. Instead, a simple relation is maintained: basic blocks in the executable correspond to basic blocks in the source program.

A basic block database consists of a list of routine entries, one for each routine in the source program. Each routine entry can be related unambiguously to a routine in the source program, and a routine in the executable. Each routine entry consists of a table of source basic blocks (SBB table), a table of object basic blocks (OBB table), and a mapping from the OBB entries to the SBB entries. Routines which have been eliminated from the executable by the compiler will still have an entry in the database. For these routines, the OBB table will be empty.

The SBB table corresponds to the basic block structure at the beginning of optimization. As long as the source code for the routine is unchanged, the IL for this routine will always have the same structure at this point in the compilation process.

The OBB table corresponds to the basic block structure of the executable. This will be different from compilation to compilation, depending upon optimization level, other optimization switches, and the profile used for the feedback optimizations. What is important is that the database correspond to the executable that is profiled.

The database relates (maps) each OBB to zero or more SBBs. OBBs can be mapped to SBBs in the same routine or to SBBs from other routines (e.g., due to inlining). Each mapping has a scale factor, which is used in deriving the source block's execution count from a profile. The scale factor is used, for example, in representing the effect of loop unrolling.

3.1 Correlation using the basic block database

In order to match the profile to the source program (i.e., the initial IL), the compiler first matches the OBB table with the profile data and the SBB table with the source files. Both of these steps are straightforward. It then uses the OBB->SBB mappings to propagate execution counts from the profile to the SBBs. The propagation of counts to SBBs must be completed for all routines at the same time, since inlining makes computation of the SBB counts a global task. Once computed, the SBB counts are associated with the initial IL and feedback directed optimization can proceed.

More than one OBB can map to the same SBB. When multiple mappings refer to the same basic block they are added together. It can also be the case that no OBBs are mapped to a particular SBB. This can happen if the compiler detects that a block is unreachable and eliminates it from the program, in which case the SBB is given an "orphan" count of zero. If an SBB is orphaned for some other reason, its count is invalid, and it does not participate in feedback directed optimizations. The following example illustrates these points. Given the following OBBs, mappings, and execution counts:

OBB	OBB->SBB map	Count
1	SBB 1, scale=1; SBB 6, scale=2	40
2	SBB 3, scale=8	100
3	SBB 3, scale=1	4
4	SBB 4, scale=2	5
5		10
6	SBB 2, scale=1	20

the compiler derives these SBB counts:

SBB	Count	computation
1	40	Count(OBB1)
2	20	Count(OBB6)
3	804	Count(OBB2) * 8 + Count(OBB3)
4	10	Count(OBB4) * 2
5	invalid	No orphan count
6	80	Count(OBB1) * 2

3.2 Transformations to the basic block database

When optimization begins, the source basic blocks and the object basic blocks are initially isomorphic. Each OBB is mapped to its corresponding SBB. As the compiler transforms the IL, the OBBs and mappings are updated to reflect the new state of the IL. This update preserves the ability to reproduce the counts for the initial IL from the transformed IL. Transformations such as creating new blocks, reordering blocks, copying and splitting blocks, merging blocks, and the deleting blocks all require substantive change to the database.

The creation of new blocks, i.e., blocks with no direct relationship to blocks in the initial source, does not affect the mappings in the database. For example, new loop tops and bottoms have no direct relationship to blocks in the initial source, and therefor are not mapped back to any SBBs. Basic block reordering is also simple to model. The permutation of the IL is identical to the permutation of the OBBs. The mappings from the OBBs to the SBBs are unchanged.

Copying and splitting both create new blocks from old blocks. They differ in how they relate to the SBBs. With copying, both the original and the copy still fully represent the original. The mappings are duplicated in the copy. Sometimes the copies are located near each other (e.g., loop unrolling), and sometimes they are not connected at all (e.g., trace building and procedure inlining).

Basic block splitting is modeled as the creation of a new, unrelated block. For example, when a routine is inlined, the block containing the call site is split at the call site. The mappings remain with the original block (the code leading to the call). The new block has no mappings. If the original block is later deleted, the mappings will be lost.

Basic block merging happens when related blocks are combined into a single block such that execution of the merged block represents execution of both of the original blocks. Both sets of mappings are copied to the new block. Mappings to the same SBB can be combined by adding their scale factors.

When blocks are deleted the OBB and all of its mappings are removed from the database. This can result in a loss of information in the basic block database. If the SBB named by the deleted mapping is no longer referenced in any mapping, it may not be possible to compute a valid count for this SBB. We adopt two strategies for mitigating this loss of information:

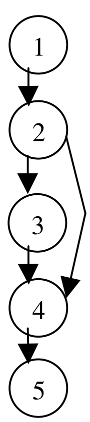
- If the block was eliminated because it was unreachable, a count of zero is given to the SBB.
- If it was eliminated for some other reason (e.g., because it is empty), the compiler attempts to treat the deletion as a merge of the deleted block with one of its predecessors or successors.

Although deleted blocks are frequently unimportant, the loss of information can sometimes cause problems. This will be discussed further in Section 4.

3.3 Compiler optimizations and basic block transformations

This section explores in more detail the effect on the basic block database of two classic optimizations performed by GEM, procedure inlining and loop unrolling.

Inlining splits a block with a call and inserts the flow graph for the inlined routine in between the two fragments:



The first part of the original block

The entry block for the inlined routine

The rest of the inlined routine (a single, conditionally executed block in this example)

The exit block for the inlined routine

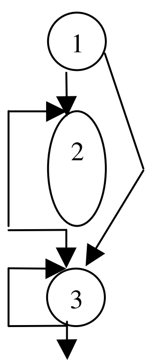
The second part of the original block

The inlining routine now includes copies of the basic blocks from the inlined routine. Assuming that the original block maps to SBB 4, and that the inlined blocks map to their corresponding SBBs, the database entries for these blocks will look like this after inlining:

OBB	Mappings	OBB	Mappings
1	SBB 4	4	SBB 3 of inlined routine
2	SBB 1 of inlined routine	5	<no mapping>
3	SBB 2 of inlined routine		

At this point, the original body of the inlined routine still exists and has its OBBs with mappings to the same SBBs. The counts for the original SBBs are computed by adding the counts for the two sets of OBBs. If the compiler later eliminates the inlined routine entirely (e.g., if it is a local routine with all call sites inlined), there will still be an entry for the eliminated routine in the basic block database. The compiler will still be able to calculate meaningful execution counts for this routine from its inlined blocks.

Loop unrolling takes a simple loop and duplicates the body of the loop `N` times to create a new loop. If the original loop meets various conditions, the new loop will be the `N` copies merged into a single block. The resultant graph looks like this:



A new block

The `N` copies of the original loop body
(where `N` is the loop unroll factor)

The original loop (now a cleanup loop, if necessary).

The block guarding the unrolled loop has no mappings, since it is not directly related to any of the blocks of the original source program. The scale factor for the unrolled loop is the result of merging the copies of the loop. Assuming that the original block mapped to SBB 5, the database entries for these blocks will now be:

OBB	Mappings
1	<no mapping>
2	SBB 5, scale=N
3	SBB 5

If the unrolled loop bodies cannot be combined into a single basic block, each of the `N` copies of the loop will map back to the original SBB with no scale factor.

Other optimizations such as trace building, loop peeling[1] and software pipelining are similar to these optimizations in their effect on the basic block database: blocks are copied, blocks are split, and new blocks are created. For example, the fill and drain blocks created for software pipelining are copies of the original loop that have been split. In this case we arbitrarily copy the mappings to the fill blocks and leave the drain blocks unmapped.

4 Evaluation of the basic block database

The effectiveness of the basic block database was evaluated with the SPEC 95 integer tests on an Alpha 21264 running version 4.0E of Compaq's TRU64 Unix operating system. Each of the eight integer tests were built and run in four modes:

Old – the old matching algorithm, unoptimized training build
Zero – the new matching algorithm, unoptimized training build
Opt – the new matching algorithm, optimized training build
Nofb – no use of feedback directed optimizations

[1] Loop peeling is a special case of trace building. If a loop is expected to execute only one or two times, one or two copies of the loop body are inserted into the graph before the loop [6].

All training runs were performed with the Spec-provided training data and gathered using pixie. All timing runs were performed with the Spec-provided reference data. Results are presented with two adjacent tables. The first table gives execution times; the second table shows the percent improvement from the no-feedback (nofb) run. The nofb category is presented to help judge the relative worth of differences among the first three.

Two separate sets of results are presented. The second set of results was generated with a compiler that was modified in response to the first results, attempting to address anomalies. Here is the first set of results:

Runtime	compress	go	perl	ijpeg	li	vortex	gcc	m88ksim
old	69.62	185.14	70.84	78.85	69.47	89.48	71.86	61.67
zero	68.73	181.95	68.26	78.53	67.99	95.31	71.12	61.00
opt	69.24	188.19	72.48	78.77	67.86	96.17	73.12	84.25
nofb	76.65	179.32	72.34	82.17	81.13	114.45	79.67	95.86

%	compress	go	perl	ijpeg	li	vortex	gcc	m88ksim
old	10.1	-3.1	2.1	4.2	16.8	27.9	10.9	55.4
zero	11.5	-1.4	6.0	4.6	19.3	20.1	12.0	57.1
opt	10.7	-4.7	-0.2	4.3	19.6	19.0	9.0	13.8

In general, the first set of results is encouraging:

- Zero vs. old - The new correlation method with an unoptimized training run outperforms the old method on 7 out of 8 of the tests. The only exception is `vortex`.
- Opt vs. old - The new method outperforms the old method on 3 tests (`compress`, `ijpeg`, and `li`), is marginally worse on 3 others (`go`, `perl`, and `gcc`), and significantly worse on 2 of the tests (`vortex` and `m88ksim`).
- Opt vs. zero - Profile data from a fully optimized training run is nearly as effective as profile data from an unoptimized training run for all but `perl` and `m88ksim`. `M88ksim` is the big exception, where the optimized training run is much less effective.

Two anomalies jump out of these results:

- For `vortex`, the new correlation method is significanly less effective than the old method.
- For `m88ksim`, an optimized training run is much less effective than an unoptimized training run.

In examining `vortex`, it was discovered that the new method produced a less accurate profile of the SBBs for one of the critical routines, `oaGetObject`. This was due to an oversight in the heuristics dealing with OBB deletion in the peephole optimizer. (Part of the GEM peephole optimizer is still run with optimization turned off, mostly to reduce nests of branches and labels.) This oversight caused some OBB->SBB mappings to be reassociated with the wrong blocks. Fixing this led the SBBs for `oaGetObject` to have identical counts for both unoptimized and optimized training runs.

Another problematic loss of mapping information was found in examining the routine `execute` in `m88ksim`. This routine consists of a large `switch` statement, where most of the cases end with a conditional. In GEM's initial IR, this idiom is represented with an empty block for the `break` (i.e., a block consisting of only a label and a branch). The optimizer eliminates this block. The heuristic for this case was to reassociate the OBB->SBB mappings of the empty block with the target of the branch. While this heuristic was frequently useful, in this circumstance it produced numerous irregular SBB counts of the form:

```
              Case DIVU:
0                    ...
                     if (ir->op == DIVU)
0                           ...
                     else
0                           ...
36490                break;
```

The block following the `switch` statement is executed 36490 times, and reassociating the mappings for the `break` with this block cause it to be treated as if it too is executed 36490 times. One alternative heuristic is to delete the mapping rather than reassociating it. This yielded mixed results: some tests improved, but others degraded. Fortunately, reassociating under some circumstances (if the target has no more than two predecessors) and deleting it otherwise yielded a uniform improvement. In fact, this heuristic helped `vortex` even more than `m88ksim`.

This was not the key to the performance degradation with `m88ksim`. For this application, the critical section of code is a loop in the routine `alignd`, which aligns floating point operands:

```
for (*s = 0; expdiff > 0; expdiff--) {
    *s |= *amantlo & 1;
    *amantlo >>= 1;
    *amantlo |= *amanthi << 31;
    *amanthi >>= 1;
}
```

The dramatic gain in performance for m88ksim is due to inlining `alignd` into `fadd64`, disambiguating the pointers, and then software pipelining the loop. The inlining happens whether the training run is done with or without optimization. However, the pointer disambiguation only happens when the training run is done without optimization. With the optimized training run, the trace builder [6] is much more aggressive in the region containing this loop. This has the unintended consequence of interfering with the pointer disambiguation for the inner loop, and ultimately keeps the loop from being software pipelined. Performing the pointer disambiguation before the traces are built leads to the loop being pipelined, and erases the performance difference.

The reason why the tracer is more aggressive with the unoptimized profile is itself interesting. The trace builder uses heuristics intended to make sure that the code duplication inherent in building traces is profitable. One of these heuristics is to only duplicate a block if it is among the most frequently executed blocks in the program. Although the counts of the duplicated blocks are identical in the optimized and unoptimized profiles, their relative frequency changes. This is because other frequently executed routines (`killtime` and `check_scoreboard` in particular) end up having very different characteristics in the two profiles, as shown in this table:

Number of highly executed blocks	optimized profile	unoptimized profile
check_scoreboard	2 blocks	40 blocks
killtime	5 blocks	26 blocks

Because of this, the optimized training data leads the trace builder to use a lower threshold in deciding if the blocks in `alignd` are highly executed.

The second set of results was generated with a compiler with these three changes: fixing the reassociations performed in the peephole optimizer, improving the heuristic used in reassociating mappings of empty blocks, and reordering optimization phases to detect value propagations before the trace is built. These results show significant change from the first results:

Runtime	compress	go	perl	ijpeg	li	vortex	gcc	m88ksim
old	70.78	189.00	70.04	78.78	67.98	89.59	72.45	61.70
zero	69.25	187.10	67.33	78.45	66.89	92.73	69.71	63.52
opt	68.94	193.25	67.86	78.63	66.66	92.57	72.38	63.23
nofb	77.13	183.52	73.74	82.06	80.59	115.40	79.28	94.65

%	compress	go	perl	ijpeg	li	vortex	gcc	m88ksim
old	9.0	-2.9	5.3	4.2	18.5	28.8	9.4	53.4
zero	11.4	-1.9	9.5	4.6	20.5	24.4	13.7	49.0
opt	11.9	-5.0	8.7	4.4	20.9	24.7	9.5	49.7

The second set of results is even more encouraging than the first set of results:

- Zero vs. old - The new correlation method with an unoptimized training run outperforms the old method on 6 of the tests (down from 7, `vortex` and `m88ksim` are the exceptions).
- Opt vs. old - The new method outperforms the old method on 5 tests (up from 3), and is only marginally worse on the other 3 (`go`, `gcc`, and `m88ksim`).
- Opt vs. zero - Profile data from a fully optimized training run is nearly as effective as profile data from an unoptimized training run for all 8 tests (up from 6).

In particular, there are improvements in both of the anomalies found in the first set of results:

- For `vortex`, there is significant improvement in the difference between the old and new methods for vortex.
- For `m88ksim`, there is a complete reduction of the difference between the optimized and unoptimized profiles.

There is still more work that could be done on these benchmarks:

- For vortex, there is still a gap between the old and new methods.
- For m88ksim, part of the closing of the gap is due to a performance degradation with the unoptimized profile.

However, the bottom line is that the new correlation method, properly tuned, is as effective as the old method (old vs. opt), and even more effective when both use unoptimized profiles (old vs. zero):

Spec ratios	old	zero	opt
First set of results	27.23	27.55	26.01
Second set of results	27.33	27.56	27.33

5 Summary and Conclusions

Feedback directed optimization can offer a significant performance boost to a variety of applications. Among other things, feedback can be used to direct optimizations that increase code size (e.g., loop unrolling, inlining and trace building) to those portions of the application which are most heavily executed [6, 7, 8, 9]. However, feedback is often seen as difficult to use. The training application must be built. Representative training data must be chosen. The training must be done. The application must be rebuilt using the training data. Any one of these tasks can be extremely time consuming and error prone.

This paper describes a method which makes the first of these steps, building the application to profile, less onerous. As the compiler optimizes an application, it saves information about the relationship between the optimized program and the original program. In subsequent compilations, the compiler uses this information to correlate a profile generated from a highly optimized executable with the original source. This makes it possible to guide feedback directed optimizations with a profile generated from a fully optimized program. This can help to make the process of using feedback significantly easier: the application built by today's production build can be profiled, and that profile can be used in tomorrow's build.

One direction for the future is to continue to streamline the profiling process by more fully integrating the basic block database with Compaq's continuous profiling tool (DCPI [4]). Once this is done we will be able to support the use of profile data gathered from fully optimized, totally untooled executables. Interestingly, one of the main problems to address in this effort is similar to the main problem addressed in this paper: since DCPI is a sampling tool, there will be inconsistencies in the profile data. The compiler must be able to deal with these inconsistencies, as it must be able to deal with the inconsistencies introduced by loss of information in the basic block database.

The correlation method supported by the basic block database is not perfect. Loss of information can be introduced into the database during the compilation process. Data presented in this paper suggests that this loss can be significant, although it also suggests that vigilant tuning of the heuristics used by the compiler in maintaining the database can keep this loss at negligible levels. Profile to source correlation using a basic block database is clearly a useful approach with advantages over other approaches.

Acknowledgements

Robert Cohn developed the original implementation of feedback in the GEM compiler. Robert also reviewed the design for the basic block database. Many others contributed to this work. Most notable are Ron Brender, who pointed me to Max Copperman's work on debugging optimized code, the other members of the GEM project, and Eric Boyd, Don Courtney, Randy Lowell, and John P. Williams, who helped integrate the basic block database with other development environment tools on Compaq TRU64 Unix.

References

1. Dave Blickstein, et al., “The GEM Optimizing Compiler System,” Digital Technical Journal, vol. 4, no. 4 (1992): 121-136.
2. Richard Sites, ed., Alpha Architecture Reference Manual, Digital Press, Burlington, MA, 1992.
3. Information about the SPEC benchmarks is available from the Standard Performance Evaluation Corporation at http://www.specbench.org/.
4. Jennifer Anderson, et al., "Continuous Profiling: Where Have All the Cycles Gone?" Proceedings of the Sixteenth ACM Symposium on Operating System Principles, Saint-Malo, France (October 1997): 1-14.
5. Max Copperman, “Debugging Optimized Code Without Being Mislead.” uscs-crl-93-21, University of California at Santa Cruz, June 11, 1993.
6. Robert Cohn, and P. Geoffrey Lowney, “Feedback directed optimization in Compaq's compilation tools for Alpha,” paper presented at the 2nd workshop on Feedback-Directed Optimization (FDO).
7. Pohua Chang, et al., "Using Profile Information to Assist Classic Code Optimizations," Software-Practice and Experience, vol. 21, no. 12 (1991): 1301-1321.
8. Wen-mai Hwu, et al., “The Superblock: An Effective Technique for VLIW and Superscalar Compilation ,” The Journal of Supercomputing, Kluwer Academic Publishers, 1993, pp. 229-248.
9. Alan Dean Samples, “Profile Driven Compilation,” Phd. Thesis, University of California at Berkeley, 1991.