Integrating High-Level Optimizations in a Production Compiler: Design and Implementation Experience

Somnath Ghosh, Abhay Kanhere, Rakesh Krishnaiyer, Dattatraya Kulkarni,
Wei Li, Chu-Cheow Lim, and John Ng

Intel® Compiler Laboratory, Intel Corporation
2200 Mission College Blvd, Santa Clara, CA 95051
Telephone number: +1-408-765-0142
wei.li@intel.com

Abstract. The High-Level Optimizer (HLO) is a key part of the compiler technology that enabled Itanium™ and Itanium™2 processors deliver leading floating-point performance at their introduction. In this paper, we discuss the design and implementation experience in integrating diverse optimizations in the HLO module. In particular, we describe decisions made in the design of HLO targeting Itanium processor family. We provide empirical data to validate the design decisions. Since HLO was implemented in a production compiler, we made certain engineering trade-offs. We discuss these trade-offs and outline key learning derived from our experience.

1 Introduction

The Explicitly Parallel Instruction Computing (EPIC) technology behind the Itanium™ processor architecture provides a rich set of features [3,4], which allow the compiler to exploit instruction-level parallelism (ILP) and optimize applications in many new ways. Intel's compiler for Itanium processor family incorporates and extends the latest optimization techniques, and new techniques have been designed specifically for the Itanium architecture [3,4]. As a result, the Intel compiler helped deliver the world's best floating point performance during the introduction of the Itanium and Itanium2 processors. The High-Level Optimizer (HLO) has been a key component that helped achieve this performance. Broadly, HLO encompasses optimizations that operate on high-level program structures such as loops and arrays. In this paper, we discuss the design and implementation of HLO targeting Itanium processor family and describe key learning out of this experience.

Processor speed has been increasing much faster than memory speed over the past several generations of processor families. HLO component in the Intel compiler for the Itanium processor applies loop-based and region-based control and data transformations in order to: i) improve data access behavior with memory optimizations, ii) maximize resource usage in innermost loops, and iii) expose higher instruction-level parallelism.

G. Hedin (Ed.): CC 2003, LNCS 2622, pp. 303–319, 2003.

In HLO, we have implemented numerous well-known and new transformations, and more importantly, we combined and extended these transformations in special ways so as to exploit the Itanium™ processor architecture features for higher application performance. Fig. 1 shows the contribution of optimizations in HLO to the performance of the SPECfp2000 benchmark suite, consisting of 14 F77/F90/C programs. All experiments in this paper were conducted using version 6.0 Beta of the Intel Compiler for Microsoft Windows 2000/XP on an 800MHz Itanium processor based system with 4MB L3 cache. The graph shows the performance improvement or serial speedup over baseline. The baseline contains all optimizations excluding HLO used for SPEC base reporting. In particular, baseline includes inter-procedural optimizations and profile feedback.

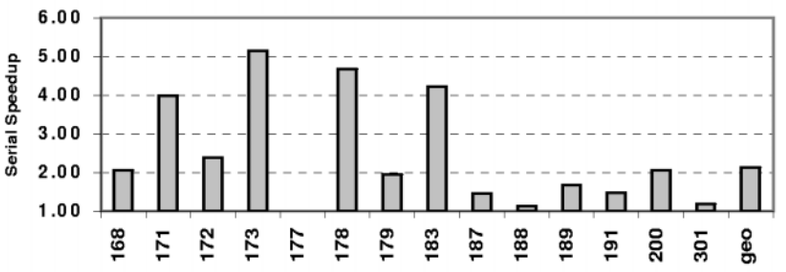

Fig. 1. Serial speedup due to optimizations in HLO. (`geo' is geomean)

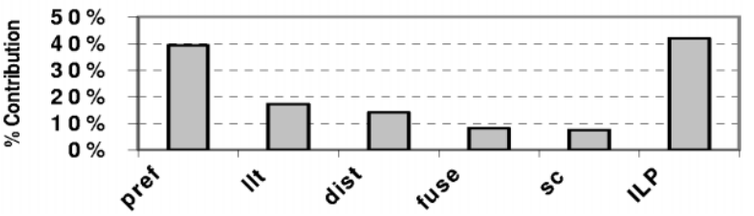

Fig. 2. Impact of individual optimizations in HLO on SPECfp2000 performance

Substantial performance gains from HLO are the result of selection of a large repertoire of transformations, design decisions that took into account the details of the Itanium architecture and careful and iterative phase-ordering decisions. This paper describes the experience in designing and implementing the HLO. The objective of this paper is to:

- Describe design decisions made during building of HLO targeting Itanium processor family.
- Present experimental results that validate the design decisions.

- Locality optimizations [1, 9]: linear loop transformations, loop fusion , loop distribution and strip-mining.
- Discuss the key learning and engineering trade-offs in a production compiler.

The rest of the paper is organized as follows. Section 2 provides design considerations for optimizations in the HLO targeting the Itanium processor family. Section 3 validates the design decisions with empirical data that show the impact of individual HLO transformations and how they interact with rest of the transformations. Key learning appears in Section 4 and concluding remarks in Section 5.

2 Design Considerations Targeting the Itanium™ Processor

In this section, we outline the design considerations for various optimizations in HLO while targeting the Itanium processor. The optimizations in HLO have been designed and implemented with a conscious effort to exploit the features in the Itanium processor architecture. In HLO, we have implemented many transformations that fall under these broad categories:

- Locality optimizations [1, 9]: linear loop transformations, loop fusion , loop distribution and strip-mining.
- ILP optimizations: unrolling, register blocking, affine-condition unswitching, and load-pair insertion.
- Maximize resource usage: Scalar replacement of memory references, affine-condition unswitching, and load-pair insertion.
- Data prefetching.
- State-of-the-art dependence and section analysis to support optimizations [1].

Fig. 2 shows the impact of individual optimizations in HLO on the performance of the SPECfp2000 benchmark suite. In the graph, x-axis shows the optimizations or groups of optimizations. Here *pref* stands for data-prefetching, *llt* for linear loop transformations, *dist* for loop distribution and strip-mining, *fuse* for fusion, *sc* for scalar replacement, and finally *ILP* stands for unroll-and-jam, affine-condition unswitching, and load-pair insertion. The y-axis is the percentage improvement because of the HLO optimizations over all other optimizations in our compiler.

This performance improvement is the result of 1) Itanium-architecture conscious design of optimizations and 2) careful orchestration of interaction between optimizations. Consequently, we are able to derive an efficient phase-ordering in our HLO which is shown in Fig. 3. (The figure does not include demand-driven calls to optimizations as a way of implementing certain other optimizations.) There exists no established technique to derive an optimal phase-ordering which is a computationally hard problem. In the subsections that follow, we describe design criteria for several optimizations targeting Itanium™ processor family. We also describe the rationale in positioning each optimization relative to other optimizations in the phase-order. These rationales together form the basis for a partial order of optimizations. The final phase order was derived from this partial order by considering other constraints such as minimizing compile time, say by minimizing updates to the dependence graph, and

ease of maintenance. In Section 3, we evaluate the current phase-order by providing empirical data on the interaction among HLO optimizations – a positive interaction validates the choices and a negative interaction shows opportunity for improvement.

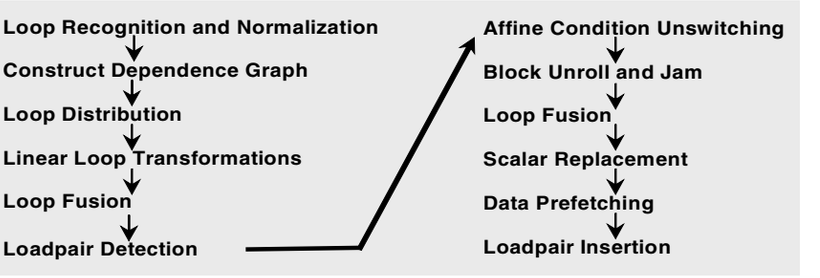

Fig. 3. Current phase-ordering of optimizations in HLO

2.1 Locality Optimizations

Caches are an important hardware means to bridge the gap between processor and memory access speeds. However, programs, as originally written, may not effectively utilize the available cache. Our design consideration was to:

- implement all loop transformations that are well-known in the community and industry to improve the locality of data reference [1,2,7,9].
- account for the fact that L2 is first level at which floating-point data may reside.
- account for the benefit of data prefetching in reuse models that trigger many loop transformations.

We have implemented *linear loop transformations*, *loop fusion*, and *loop distribution* in the Intel compiler to improve data locality. As a combined effect, linear loop transformations [6,9] can dramatically improve memory access locality. They can also improve the effectiveness of other optimizations, such as scalar replacement, invariant code motion, and software pipelining. For example, a loop interchange can make references to arrays to be inner-loop invariant, besides improving the access behavior of other references. Important design considerations are:

- Make sure phase order is such that linear loop transformations occur sufficiently early to enable other transformations.
- Software pipelining [5] is important on the Itanium processor family, so use linear loop transformations to improve effectiveness by enabling parallelism in inner loops, and by interchanging, whenever possible, so that innermost loops have sufficiently large counts.
- Linear loop transformations have to be aware of the benefits of exposing references with spatial locality for effective data prefetching.

Loop fusion [1,8] is effective in improving cache performance, since it combines the cache context of multiple loops into a single new loop. Thus, data reuse across nested loops is within the same new nested loop. Loop fusion increases opportunities for reducing the overhead of array references by replacing them with references to compiler-generated scalar variables. Loop fusion also improves the effectiveness of data prefetching. While implementing loop fusion for the Itanium processor family, one must consider:

- 128 floating-point and 128 general registers available in the Itanium processor family. This allows aggressive loop fusions without the risk of register pressure. The design allows for loop fusion across call boundaries, and code motion to enable loop fusion.
- Trade-offs between locality and instruction level parallelism. For example, a fused loop may not be software pipelined because of dependences. In this case, the benefits of locality must outweigh the benefits of exploiting ILP across the back-edge of the fused loop.

Besides enabling other transformations, loop distribution [1,9] spreads the potentially large cache context of the original loop into different new loops, so that the new loops have manageable cache contexts and higher cache hit rates. While designing loop distribution for the Itanium processor family, the design considerations included the following:

- As in other compilers, use loop distribution to create perfect nests, enable loop interchange and loop blocking.
- Use loop distribution to partition a loop into loops with calls or non-inlined intrinsic and loops without calls. This is because, loops with calls cannot be software pipelined in the current compiler.
- Large loops may be distributed to avoid running out of rotating registers in software pipelining. This requires tradeoff between loss of locality and benefit of software pipelining.
- Ability to expose ILP across loop back-edges has sufficiently higher benefit to tilt the balance towards expansion of scalar variables to enable loop distribution.

Phase-ordering constraints: Loop distribution, linear loop transformations, and loop fusion are run in that order. Loop distribution exposes perfect nests and thus opportunities for linear loop transformations. Together, they expose opportunities for loop fusion. All three rely on the same cost model for better synergy.

2.2 ILP Optimizations

In this section we describe unrolling while the other ILP optimizations are covered in Section 2.3. The design of the Intel compiler for the Itanium processor unifies loop blocking, unroll-and-jam [1,9], and inner loop unrolling. Loop unrolling exposes parallelism across instructions in adjacent loop iterations. The large number of registers in the Itanium processor architecture enables the compiler to unroll loops by significantly larger factors without register spills than compilers for other contemporary architectures. This feature can be used to expose outer loops as new inner loops for

software pipelining. While designing block-unroll-jam for the Itanium processor family, we had the following considerations:

- Unroll aggressively so as to extract greater ILP using large register file, and expose outer and larger loops to software pipelining and effective data prefetching.
- While blocking, consider the interaction of prefetching, need for a larger loop iteration count for efficient software pipelining, available bandwidth, and primarily the ability to issue 2 Fused-Multiply-Add (FMA) instructions in a cycle.
- Unroll and unroll-and-jam to maximally use machine resources and avoid *fractional II* loop body that under-utilizes machine resources.
- Use unrolling to expose more opportunities for loop fusion, insertion of load-pair instructions, and maximal resource usage.
- Pay attention to compile time since loop unrolling increases code size linearly, and the increase could adversely affect later optimizations that are quadratic or cubic in compile-time complexity.

Phase-ordering constraints: From the discussion above, it is clear that unroll-and-jam should be performed after loop distribution, interchange, and fusion, but before data prefetching and scalar replacement. We will find later that load-pair insertion has to be done after unrolling as well. However, note that there is an advantage to performing loop fusion again after loop unrolling as it exposes more conforming loop nests to loop fusion. Thus there is a second call to fusion after unroll-and-jam as shown in Fig. 3.

2.3 Maximizing Resource Usage

This category of optimizations includes scalar replacement of memory references, load-pair insertion, and affine-condition unswitching. Scalar replacement [2] is a technique to replace memory references with compiler-generated temporary scalar variables that are eventually mapped to registers. Most back-end optimization techniques map array references to registers when there is no loop-carried data dependence. However, the back-end optimizations do not have accurate dependence information to replace memory references with loop-carried dependence by scalar variables. Scalar replacement, as implemented in the Intel compiler for the Itanium™ processor, also replaces loop invariant memory references with scalar variables defined at appropriate levels of the loop nesting.

The design considerations for scalar replacement on an Itanium processor based platform are:

- Map the compiler-inserted scalars directly onto rotating registers supported by the Itanium architecture [3]. The scalar moves required for scalar replacement [2,7] to preserve values across iterations are marked as MCOPY statements. The code generator maps the scalars to appropriate rotating registers so that explicit move instructions are unnecessary.

- Ensure that exact dependence information is available for the common cases. This also implies that all earlier transformations such as unrolling will have to maintain accurate dependence information.

Itanium processor architecture provides instructions that load a pair of floating-point numbers at a time [3]. Such load-pair instructions take a single memory issue slot, thus possibly reducing the initiation interval of a software pipelined loop. For example, the loop in Fig. 4(a) has three memory operations per iteration. By using load-pair operations, the number of memory references can be reduced to two per iteration after unrolling (4(b) and 4(c)). The load-pair optimization has to take into account the following requirements on the Itanium processor.

- Load-pairs can be issued only at certain alignment boundaries, for example 16-byte boundary for double precision data elements. We therefore either need to generate code to peel off aligned portions of loops, or generate multi-version code for different alignment combinations.
- The load-pair results have to be loaded into an odd-even register pair. This requirement can only be enforced during register allocation in the code-generation phase of the compiler. We however chose to implement the load-pair optimization phase in HLO because high-level information is available to identify adjacent memory loads, and because we rely on loop unrolling to expose more load-pair opportunities than what would normally be found in user code.
- There must be a utility to determine the number of load-pairs that need to be inserted to balance memory operations and computations.

Affine-condition unswitching hoists conditions out of loops. This has been used in the compiler community to mostly expose perfect nests. However, we find that it is quite useful in improving the effectiveness of software pipelining as well. The initiation interval for loops with conditions tends to be much larger than what it should be considering that only some of the branches will be taken in any iteration of the loop.

We use affine conditions to partition the loops into many loops, where in each new loop there is code corresponding to one of the paths. As a result the initiation intervals of the new loops will only correspond to the instructions that are always executed. A key design consideration was to avoid code size bloat and un-switch only the critical conditions.

Phase-ordering constraints: Scalar replacement of memory references should be one of the last few optimizations in HLO since transformations such as loop interchange and unrolling expose new opportunities for scalar replacement of memory references. Array contraction is also performed as part of scalar replacement since they use similar logic. Since, scalar replacement and array contraction need accurate dependence analysis, the dependence graphs must be rebuilt before entering scalar replacement.

The interaction between load-pair insertion and loop unrolling influenced the design of load-pair insertion technique and the phase-ordering between unroll-jam and load-pair insertion.

- Position insertion of load-pairs after unroll-jam, because the latter exposes opportunities for inserting load-pair instructions.

- However, load-pair insertion cannot call unroll-jam on demand because that would also require other optimizations such as scalar replacement, that eliminate redundant loads, to be done after unroll-jam

These two constraints influenced us to divide load-pair insertion into two stages. The first stage of load-pair optimization is therefore executed immediately before loop unrolling, so that the loop is analyzed to estimate the number of load-pairs that may be generated if the loop were to be unrolled by a factor of 2. Loop unrolling will then factor the result into its resource usage model to determine the unroll factor.

The second stage is where the load-pairs are actually identified. Since it is run after unrolling, we are able to identify load-pairs that are either derived from user's original code or exposed by loop unrolling. Because we want to prevent load-pairs from being applied to redundant loads, this stage is placed after scalar replacement. Scalar replacement is also run after loop unrolling because the latter may expose redundant loads to be eliminated.

Affine-condition unswitching has to be run after linear loop transformations, because that is when we know that the innermost loop will be exposed to software pipelining. It is advantageous to perform affine-condition unswitching immediately after linear transformations. We made an engineering decision to move it later in phase-order and position it after load-pair detection so as to minimize updates to dependence graph.

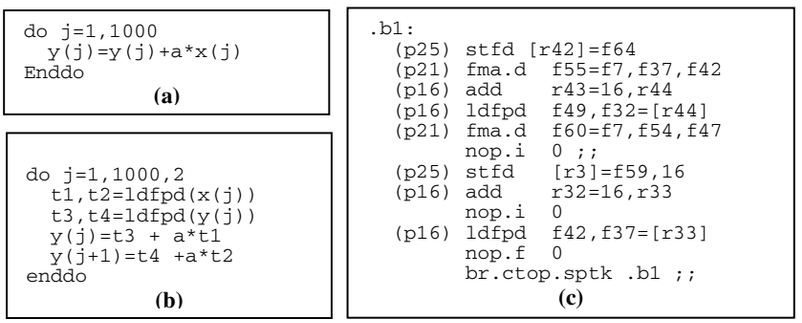

Fig. 4. An example of the use of load pairs

2.4 Data Prefetching

Data prefetching is an effective technique to hide memory access latency. Prefetch instructions (named *lfetch* in the Itanium processor architecture) have one argument: the address to be prefetched. The effect of the instruction is to move the cache line containing the address to a higher level of the memory hierarchy. The address itself has no cache alignment requirement.

In the example in Fig. 5, the compiler inserts prefetches for arrays a and b making use of the support for rotating registers in the Itanium processor architecture to mini-

mize the prefetch overheads. In this example, **incr** is a function of the cache line size, prefetch frequency, and the number of arrays that need to be prefetched within the loop. The addresses of the two arrays **a** and **b** that require prefetching are initialized before the loop (**r33** and **r34**). The design considerations for a prefetching algorithm on the Itanium processor are:

- Use data-locality analysis to selectively prefetch only those data references that are likely to suffer cache misses. References with spatial locality are selectively prefetched under a conditional of the form **(i mod L) == 0**, where **i** is the loop index and **L** denotes the cache line size. When multiple references access the same cache line, then only the leading reference needs to be prefetched.
- The cost incurred while prefetching data arises from the added overhead of executing prefetch instructions as well as instructions required for prefetch address calculation and predicate computation. The prefetch instructions will occupy memory slots, thereby increasing resource usage. *Compute-intensive* applications normally have sufficient free memory slots. Benefits from prefetching have to be weighed against the increase in resource usage in *memory-intensive* applications.
- The predication support in Itanium processor architecture provides an efficient way of adding prefetch instructions. The conditionals within the loop are converted to predicates through if-conversion, thus changing control dependency into data dependency. Indirect array references are prefetched making use of speculation support to load the index array speculatively.
- When multiple array references with spatial locality are accessed uniformly within a loop, prefetches can be issued with a single *lfetch* instruction that uses a rotating register to rotate the addresses of the different arrays that must be prefetched [4]. An example of this technique is illustrated in Fig. 5. This technique obviates the need for predicate calculations within the loop and saves memory slots that would otherwise be occupied by multiple *lfetch* instructions.
- Prefetch distance is estimated based on the memory latency, the resource requirements in the loop, and data dependence information.

The large number of registers available in the Itanium processor architecture enables prefetch addresses to be stored in registers obviating the need for register spill and fill within loops.

Phase-ordering constraints: Prefetching is run after most of the other optimizations within HLO. This is because prefetching can benefit from a lot of these other optimizations. Loop unroller will unroll all inner loops with small trip counts to expose any outer loops that may have a larger trip count. This makes the prefetches more effective. Fusion may reduce the total number of prefetches issued if the loops that are fused access the same data. Performing scalar replacement before prefetching ensures that a lot of memory references with group locality are replaced by temporary variables, thus reducing the compile time for prefetching.

As a whole, prefetching also interacts a lot with other optimizations outside HLO. For example, strength reduction is run after HLO, making sure that the addresses that are inserted by prefetching are strength-reduced. Also, there is a handshake between prefetch and the software-pipeliner that is part of the code-generator. As part of HLO, the compiler estimates the likelihood of a loop being pipelined. If a loop is predicted

to be software-pipelined, an estimate of the initiation interval of the loop based on resource requirements is computed in advance. This estimate aids in the distance calculation for the prefetches in the loop. Also when prefetch relies on register rotation, the address copies are specially marked (shown as MCOPY in Fig. 5) for the software pipeliner. These special copies are turned into automatic copies using register rotation by the pipeliner.

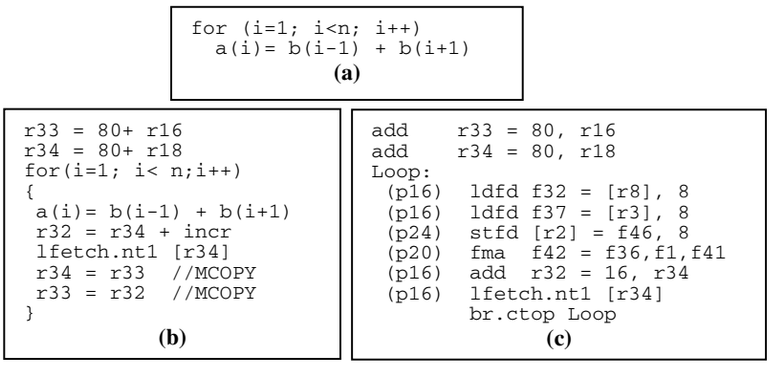

Fig. 5. Prefetch example illustrating the use of rotating registers: (a) original loop, (b) prefetch for a and b using a single lfetch instruction with rotation shown as explicit assignments, and (c) assembly code on Itanium™ processor with register rotation

3 Evaluation of Design Decisions

Certain compiler optimizations are independent in nature, in that their effect is independent of other optimizations. However, many compiler optimizations are highly inter-dependent. The interaction between the optimizations tends to be complex. An interaction could be positive in that an optimization enables several other optimizations or improves the effectiveness of other optimizations. An interaction could also be negative in that an optimization may disable, reduce, or mask the effectiveness of other optimizations. A chosen phase-ordering is most effective when all interactions are positive. In this section, we present experimental data for the interaction between optimizations in our HLO to show the effectiveness of our chosen phase ordering.

3.1 Experimental Framework

The data presented here are based on the performance of our compiler on the SPECfp2000 benchmark suite. We used version 6.0 Beta of the Intel Compiler for Microsoft Windows 2000/XP on an 800MHz Itanium processor based system with 4MB L3 cache. All the experiments done here are intended to show the interaction between

important optimizations in HLO. Later, we also present the interaction of some other important compiler modules with HLO.

We use the following notations in the discussion here:

- OPT: Set of all HLO optimizations under consideration.
- *opt*: one individual optimization in OPT.
- *P(X):* Represents the performance with the optimizations in *X* turned on.

In order to show the interaction among the optimizations in OPT, we measured the performance of all the benchmarks in SPECfp2000 with the following configurations:

- *P(BOTTOM):* Baseline performance where just the optimizations in OPT are disabled. BOTTOM represents all optimizations except HLO optimizations in OPT.
- *P(TOP):* Performance with all optimizations in BOTTOM and OPT turned on. This corresponds to the base compiler options for the reported SPECfp2000 performance numbers.
- *P(BOTTOM+opt):* Performance of BOTTOM with optimization *opt* turned on. This data is collected for each optimization in OPT.
- *P(TOP-opt):* Performance of TOP with optimization *opt* turned off. This data is collected for each optimization in OPT.

The intuition behind collecting the above data is to find the effect of each optimization when applied along with other optimizations as opposed to when applied on its own. This shows how an optimization performs in the absence and presence of other optimizations and thereby provides insight into interaction of this optimization with the other optimizations. We can make the following observations based on the above data:

1. *P(BOTTOM+opt) – P(BOTTOM),* say *gain_at_bottom*, gives the performance improvement or degradation when *opt* is the only HLO optimization turned on.
2. *P(TOP) – P(TOP-opt),* say *gain_at_top*, gives the performance improvement or degradation of *opt* when applied along with other optimizations.

Clearly, when the above quantities are same for an optimization *opt*, then *opt* does not interact with any other optimization in OPT. In other words, if *opt* improved (degraded) performance when applied on its own, then it would continue to improve (degrade) by the same extent when applied along with remaining optimizations in OPT. When *gain_at_top* is very large compared to *gain_at_bottom*, then most of the benefit from applying *opt* is the result of its positive interaction with other optimizations. This could happen, for instance, when another optimization in OPT that is applied earlier enables *opt* or improves its effectiveness. We can also get such a scenario if *opt* enabled a later optimization in OPT. This implies that a favorable phase-ordering was chosen.

Similarly, if *opt* interacts negatively with the remaining optimizations in OPT, *gain_at_top* is less than *gain_at_bottom*. This usually suggests room for improvement either as tuning of an optimization or change in phase ordering. It may also be the case that two optimizations in OPT target the same performance issue, and the benefits obtained from the two optimizations are not additive in nature.

The graphs presented in this section show this interaction. We explain this with respect to the graph shown in Fig. 6. We normalize all the data with respect to *P(TOP)-P(BOTTOM)*. The actual speedup of *P(TOP)* over *P(BOTTOM)* was shown in Fig. 1

and Fig. 2. For a given optimization in OPT, we provide cumulative bar graphs for each benchmark performance. As shown in the legend, for a given benchmark we show the following sections on a bar in this order:

1. *P(BOTTOM+opt) - P(BOTTOM)*: Performance gain (loss) from applying only optimization *opt* from the set of HLO optimizations in OPT.
2. *P(Top-opt) - P(BOTTOM):* Performance gain (loss) from applying all optimizations in OPT except *opt*.
3. This is the additional gain or loss to reach *P(TOP)-P(BOTTOM)*. This is due to the interaction of *opt* with other optimizations in OPT. Positive and negative interactions are shaded differently.

Note that if *gain_at_bottom* is equal to *gain_at_top*, i.e. when there is no interaction, we can easily deduce that the first two sections in a bar should add up to *P(TOP)-P(BOTTOM)* which has the value one in our graphs. Similarly, if *gain_at_top* is more than *gain_at_bottom*, i.e. when there is positive interaction, then *P(TOP)-P(BOTTOM)* is more than the sum of the first two bar sections. On the contrary, it is less than the sum of the first two sections if the interaction is negative.

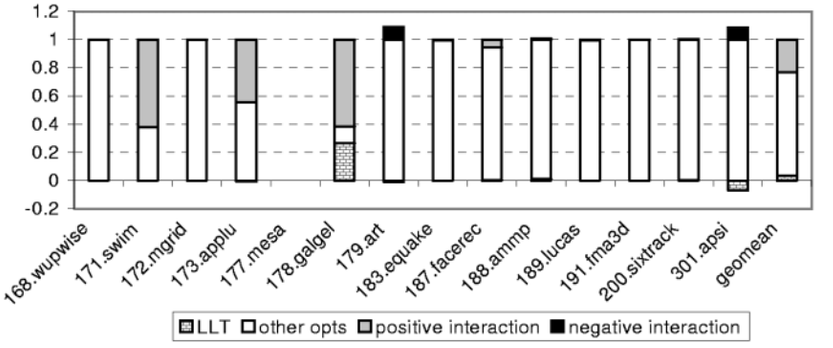

Fig. 6. Interaction graph for linear loop transformations

3.2 Analysis of Interaction Data

The graphs and analysis provided here show that HLO optimizations tend to have a high degree of interaction. They also validate our design consideration and phase-ordering and provide some useful insights to further opportunities. (The legend shown in Fig. 6 applies to all the graphs for all optimizations discussed in this section.)

3.2.1 Linear Loop Transformations

Fig. 6 shows the interactions for linear loop transformations. This shows that these transformations interact significantly with other optimizations in HLO for the benchmarks 171.swim, 173.applu, 178.galgel, and 301.apsi. In 171.swim, linear loop trans-

formations interact with data prefetching. Loop interchange exposes array accesses with spatial locality that make data prefetching more effective. For 173.applu, there is interaction between linear loop transformations and loop fusion. In this case, two adjacent loops could be fused only after loop reversal of the second loop. Interactions in 178.galgel are described in detail in Section 4. Note that 301.apsi shows negative interaction both at bottom (below y=0 in the graph) and top (above y=1 in the graph), which is a manifestation of small performance loss due to loop interchange. The performance loss is due to differences in distances computed for data prefetching for unit stride and non-unit stride array references.

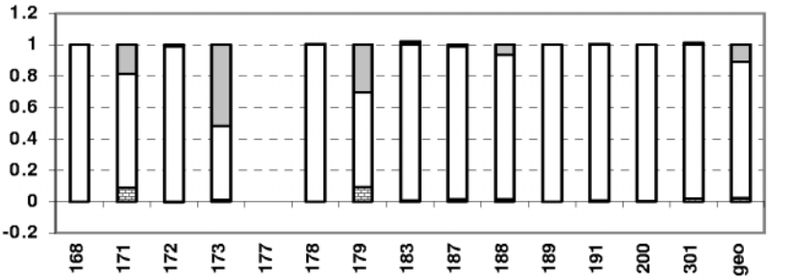

Fig. 7. Interaction graph for loop fusion

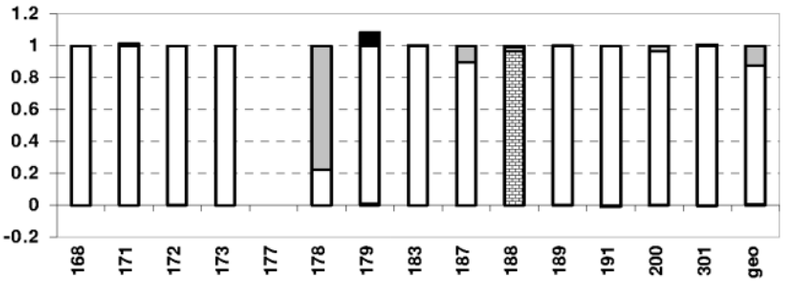

Fig. 8. Interaction graph for loop distribution

3.2.2 Loop Fusion

In order to enable effective loop fusion, transformations like code motion, loop peeling, loop reversal, and extensive array section analysis are required. Fig. 7 shows the interaction for loop fusion. We observe that interaction is high for 171.swim, 173.applu, and 179.art. There are two primary reasons for interactions in 173.applu – first, loop reversal enables more fusion as explained in the last subsection; second,

loop fusion enables many scalar replacements. Loop fusion enables scalar replacements and improves the effectiveness of data prefetching in 179.art.

3.2.3 Loop Distribution

Fig. 8 shows the interaction for loop distribution. 178.galgel has a very high interaction. As we will explain in Section 4, a combination of loop distribution, interchange, and unroll helps to considerably improve the performance of 178.galgel. But, loop distribution when applied alone, has no effect on performance. In 188.ammp, loop distribution helps software pipelining by partitioning loops into pipelineable and non-pipelineable sections. 179.art has a short run-time and the negative interaction in the graph is within experimental error.

3.2.4 Scalar Replacement

Interaction for scalar replacement of memory references (which also includes array contraction) is shown in Fig. 9. Scalar replacement of memory references, when applied alone, improves performance for 172.mgrid and 301.apsi.. In 172.mgrid, 173.applu, 178.galgel, and 200.sixtrack, other transformations help scalar replacement to be more effective. In 173.applu, a large number of scalar replacements are enabled by loop fusion. In 200.sixtrack, loop unrolling enables more opportunities for scalar replacement in key loops.

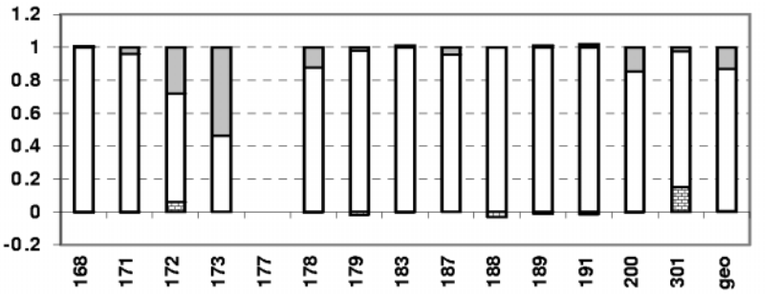

Fig. 9. Interaction graph for scalar replacement

3.2.5 ILP Enhancing Techniques

Interaction of loop unrolling, load-pair insertion and affine-condition unswitching is shown in Fig. 10. In 173.applu, loop unrolling enables loop fusion across large regions. In 183.equake loop unrolling enables loop fusion and prefetching. 200.sixtrack is a floating-point intensive code with many opportunities for extracting ILP. In this application, loop unrolling enables larger loops to be pipelined. Larger loops also help prefetching to be more effective.

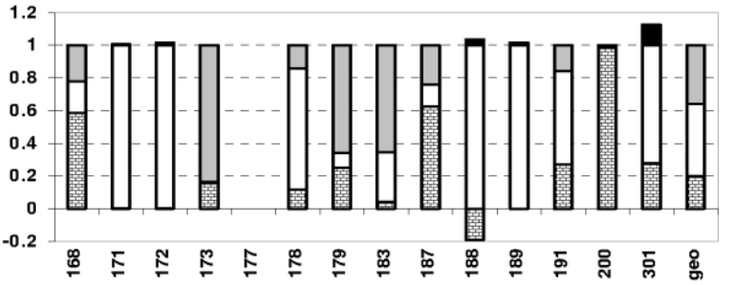

Fig. 10. Interaction graph for loop unrolling and load-pair insertion

3.2.6 Data Prefetching

Interaction for data prefetching is shown in Fig. 11. It is interesting to note that data prefetching on its own benefits nearly all applications in the graph. Data prefetching also shows significant positive interaction with other optimizations. For example, bars for 183.equake, 171.swim and 173.applu show large improvement in performance due to positive interactions with other transformations. Loop unrolling, fusion, and blocking are key helpers. For example, unrolling of loops exposes larger inner loop to data prefetching. Note that the gains from prefetching for 173.applu are present only in the presence of other optimizations. Bar for 189.lucas is dominated by the gain from prefetch alone and does not show interactions with other HLO optimizations. 200.sixtrack does not benefit from prefetching, and there is a small degradation in performance at the bottom and at the top. This is because the data accessed fits in the cache, and prefetching only adds to the resource requirements without any noticeable benefit. The geomean for benefits from prefetching alone is close to 20%, and this increases to about 40% at the top (as shown in Fig. 2) due to significant positive interactions.

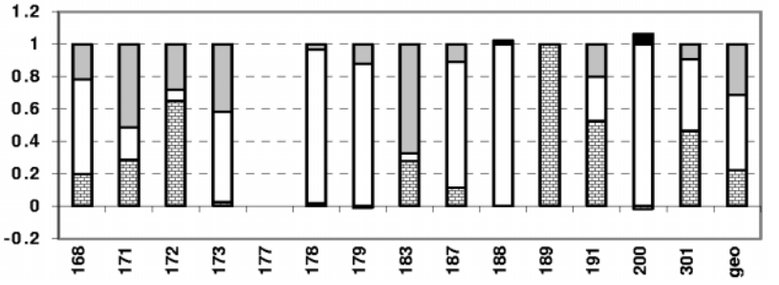

Fig. 11. Interaction graph for data prefetching

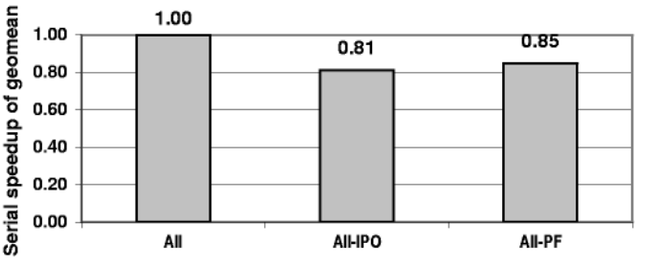

Fig. 12. Influence of IPO and Profile feedback (PF) on HLO. (All=HLO+IPO+PF)

4 Key Learning

The design decisions discussed in previous sections were all related to the fact that we were targeting HLO for Itanium processor family. However, some of our other decisions were made because we implemented HLO in a production compiler. In a production compiler, minimizing compile time, memory usage, and maintenance costs is very important. In fact, while designing a production compiler, designers tend to forego an opportunity to improve performance in order to improve compile time, memory usage or maintenance efforts. For example, positioning affine-condition unswitching early in phase-order can enable more transformations. However, unswitching causes an update on the dependence graph that is expensive. We chose to position unswitching later in the phase-order, because we viewed the compile time increase a higher penalty than potential gains of moving it earlier in the phase-order. In contrast, we chose to rebuild rather than incrementally update dependence graph after certain sequence of transformations. This decision slightly increased the compile time. However, we estimated that the increase in compile time was better than the engineering cost of maintaining an incremental dependence update mechanism.

Early in the design and implementation phase of HLO, we decided that the transformations needed only the high-level resource estimates – such as number of basic blocks. However, we learned that the transformations can be more effective with low-level resource estimates. We redesigned resource estimation to include an estimation of initiation interval of loops, number of registers, and whether a loop is likely to be software pipelined. This redesign proved to be key to many transformations including loop fusion, distribution, unrolling and insertion of load-pair instructions.

Traditional compilers tend to do a maximal loop distribution followed by a loop fusion. We learnt that for certain engineering applications this can result in sub-optimal performance. We had to overlay the loop distribution heuristics to control distributions by distributing only at heuristically determined points.

Load-pair instructions proved to be very important for certain engineering applications. However, for certain other applications, load-pair instructions did not yield significant improvements as we expected. We learnt that for these applications saving

memory resources did not matter as much because the memory operations were incurring a higher latency than what was assumed at the time of scheduling.

We believe that HLO communicates more information to the code generator compared to other high-level optimizers in the industry. This helped us tightly integrate the two components for higher overall performance. We made several decisions that helped communicate only the information that would be needed to minimize memory usage and compile time. For example, dependence information that can be easily deduced from a symbolic memory disambiguator was not explicitly communicated via a dependence graph.

In this paper, we did not discuss the impact of analysis beyond HLO. Effectiveness of optimizations in HLO is enhanced by inter-procedural optimization (IPO) and profile feedback [4], that are included in the SPEC base options. Their interactions with HLO are shown in Fig. 12.

5 Concluding Remarks

In this paper, we described design decisions made while designing and implementing HLO targeting Itanium processor family. We presented experimental results that validate the design decisions. The results showed a well designed high-level optimizer can have significant impact on overall performance. Such a design must consider at least the architecture-driven design considerations discussed in this paper. From the evaluation of the design choices we made, we can conclude that implementing the entire repertoire of transformations is disproportionately more effective than a subset. Since HLO was implemented in a production compiler, we made certain engineering trade-offs. We discussed these trade-offs and outlined key learning derived from our experience.

References

[1] R. Allen and K. Kennedy, Optimizing Compilers for Modern Architectures, Morgan Kaufmann Publishers, 2002.
[2] S.Carr, “Memory-Hierarchy Management” Ph.D. Thesis, Rice University, July 1994.
[3] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir, “Introducing the IA-64 Architecture,” IEEE Micro, Sept-Oct 2000, 12–23.
[4] R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, C. Lim, J. Ng, D. Sehr, “An Advanced Optimizer for the IA-64 Architecture,” IEEE Micro, Nov-Dec 2000.
[5] M.S. Lam, “Software Pipelining: An Effective Scheduling Technique for {VLIW} Machines,” in Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, June 1988, 318–328.
[6] W. Li and K. Pingali, “A Singular Loop Transformation Framework Based on Non-Singular Matrices,” *International Journal of Parallel Programming*, volume 22 (2), 1994.
[7] S. Muchnik, Advanced Compiler Design Implementation, Morgan Kaufman, 1997.
[8] Singhai & McKinley, “A parameterized Loop Fusion Algorithm for Improving Parallelism and Cache Locality”, by The Computer Journal, 1997"
[9] M. Wolfe, High Performance Compilers for Parallel Computing, Addison-Wesley, 1996.