An Overview of the Intel® IA-64 Compiler

Carole Dulong, Microcomputer Software Laboratory, Intel Corporation
Rakesh Krishnaiyer, Microcomputer Software Laboratory, Intel Corporation
Dattatraya Kulkarni, Microcomputer Software Laboratory, Intel Corporation
Daniel Lavery, Microcomputer Software Laboratory, Intel Corporation
Wei Li, Microcomputer Software Laboratory, Intel Corporation
John Ng, Microcomputer Software Laboratory, Intel Corporation
David Sehr, Microcomputer Software Laboratory, Intel Corporation

Index words: IA-64, predication, speculation, compiler optimization, loop transformations, scalar optimizations, profiling, interprocedural optimizations

ABSTRACT

The IA-64 architecture is designed with a unique combination of rich features so that it overcomes the limitations of traditional architectures and provides performance scalability for the future. The IA-64 features expose new opportunities for the compiler to optimize applications. We have incorporated into the Intel IA-64 compiler the key technology necessary to exploit these new optimization opportunities and to boost the performance of applications on the IA-64 hardware. In this paper, we provide an overview of the Intel IA-64 compiler, discuss and illustrate several optimization techniques, and explain how these optimizations help harness the power of IA-64 for higher application performance.

INTRODUCTION

The IA-64 architecture has a rich set of features including control and data speculation, predication, large register files, and an advanced branch architecture [7, 13]. These features allow the compiler to optimize applications in new ways. To this end, the Intel IA-64 compiler incorporates the key technology necessary to exploit new optimization opportunities and to boost the performance of applications on IA-64 systems.

The Intel IA-64 compiler targets three main goals while compiling an application: i) to minimize the overhead of memory accesses, ii) to minimize the overhead of branches, and iii) to maximize instruction-level parallelism. The compilation techniques in the compiler take advantage of the IA-64 architectural features that are expressly designed to alleviate these very overheads. For instance, memory operations are eliminated by effectively using the large register file. Optimizations use rotating registers to reduce the overhead of software register renaming in loops. Predication is used in many situations, such as removing hard-to-predict branches and implementing an efficient prefetching policy. The compiler uses control and data speculation to eliminate redundant loads, stores, and computations.

In the first section of this paper, we present the high-level software architecture of the Intel IA-64 compiler. We then describe profile-guided and interprocedural optimizations, respectively. Memory disambiguation, a key analysis technique that enables several optimizations, is then discussed. We follow this with a description of memory optimizations. The design provisions for supporting parallelism at both coarse and fine granularity are discussed next followed by a section on scalar optimizations, which are aimed at eliminating redundant computations and expressions. Finally, we briefly describe code generation and scheduling techniques in the compiler.

THE ARCHITECTURE OF THE INTEL IA-64 COMPILER

The software architecture of the Intel IA-64 compiler is shown in Figure 1. The compiler incorporates i) state-of-the-art optimization techniques known in the compiler community, ii) optimization techniques that are extended to include the resources and features in the IA-64, and iii) new optimization techniques designed to fully leverage the IA-64 features for higher application performance.

Many of these techniques are described in subsequent sections of this paper.

The compiler has a common intermediate representation for C*, C++*, and FORTRAN90*, so that a majority of the optimization techniques are applicable irrespective of the source language (although certain optimization techniques take advantage of the special aspects of the source language).

Information about the program execution behavior, profile information, can be very useful in optimizing programs. The components in the Intel IA-64 compiler are designed to be aware of *profile information* [22], so that the compiler can select and tune optimizations for the target application when run-time profile information is available. *Interprocedural analysis* and optimization [16] have proven to be effective in optimizing applications by exposing opportunities across procedure call boundaries.

The optimizations in the Intel IA-64 compiler can be grouped into *high-level optimizations* including memory optimization, and parallelization and vectorization; *scalar optimizations;* and *scheduling and code generation*, which together achieve the three optimization goals mentioned in the introduction.

These *high-level optimizations* include loop-based and region-based control and data transformations to i) improve memory access locality, ii) expose coarse grain parallelism, iii) vectorize, and iv) expose higher instruction-level parallelism. The high-level optimization techniques are typically applied to program structures at a higher level of abstraction than those in many other optimizations. Therefore, the Intel IA-64 compiler elevates the common intermediate language while applying high-level optimizations, and it represents loop structures and array subscripts explicitly. This facilitates efficient access and update of program structures.

Some of the high-level optimizations in the Intel IA-64 compiler are *linear loop transformations* [17, 18], *loop fusion*, *loop tiling*, and *loop distribution* [16], which can improve the cache locality of array references. *Loop unroll and jam* [14] and *loop unrolling* exploit the large register file to eliminate redundant references to array elements and to expose more parallelism to the scheduler and code generator. *Scalar replacement* of memory references [14, 15] is a technique to replace memory references by compiler-generated temporary scalar variables, which are eventually mapped to registers. Finally, the compiler also inserts the appropriate type of *prefetches* [7, 19, 20] for data references so as to overlap the memory access latency with computation. These transformations are described in detail in later sections of this paper.

A primary objective of *scalar optimizations* is to minimize the number of computations and the number of references to memory. Scalar optimizations achieve this objective by a natural extension to a well known optimization, called *partial redundancy elimination* (PRE) [1,2,11], which minimizes the number of times an expression is evaluated. We have extended the PRE of the IA-64 compiler to eliminate both redundant computations and redundant loads of the same or known values. Moreover, the extended PRE uses control and data speculation to increase the number of loads that can be eliminated. The counterpart of PRE, called *partial dead store elimination* (PDSE), is used to remove redundant stores to memory. PDSE moves stores downward in the program's flow in order to expose and eliminate stores that have the same value.

Scheduling and code generation make effective use of predication, speculation, and rotating registers by if-conversion, global code scheduling, software pipelining, and rotating register allocation.

Optimizations in the IA-64 compiler are supported by state-of-the-art analysis techniques. *Memory disambiguation* determines whether two memory references potentially access the same memory location. This information is critical in hiding memory latency, because knowing that a store does not interfere with a later load is essential to scheduling memory references earlier. We also use data reuse and exact array data dependence information to guide certain optimizations.

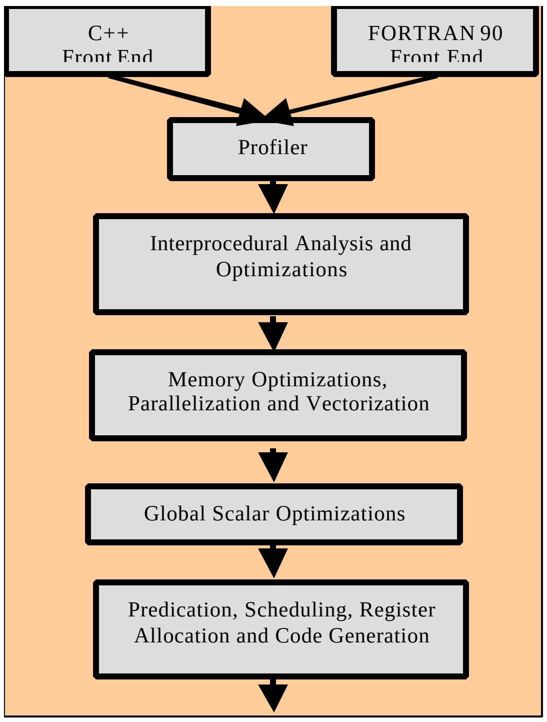
C++ Front End
FORTRAN 90 Front End
Profiler
Interprocedural Analysis and Optimizations
Memory Optimizations, Parallelization and Vectorization
Global Scalar Optimizations
Predication, Scheduling, Register Allocation and Code Generation

Figure 1: Organization of the Intel IA-64 compiler

PROFILE-GUIDED OPTIMIZATIONS

The compiler may be able to take the fullest advantage of the IA-64 architecture when accurate information about the program execution behavior, called *profile information*, is available. Profile information consists of a frequency for each basic block and a probability for each branch in the program.

The Intel IA-64 compiler gathers profile information about the specified program and annotates the intermediate language for the program with this information. The compiler supports two modes for determining profile information: *static* and *dynamic*. Static profiling, as the name suggests, is collected by the compiler without any trial runs of the program. The compiler uses a collection of heuristics to estimate the frequencies and probabilities, based on knowledge of "typical" program characteristics. Static information is necessarily approximate because it must be general enough to work with all programs. The compiler uses static profiling information whenever the optimizer is active, unless the developer selects dynamic profiling.

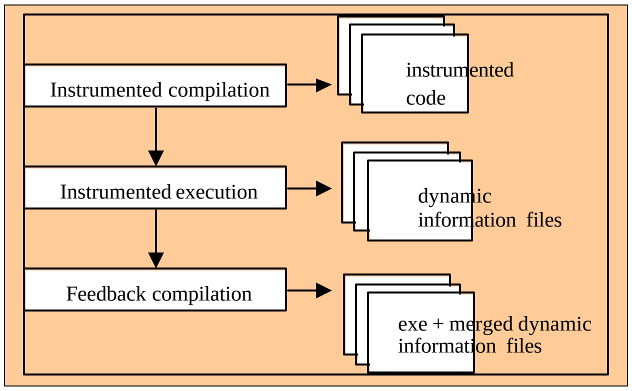

Figure 2: Steps in dynamic profile-guided compilation

Dynamic profiling information, or *profile feedback*, is gathered in a three-step process as shown in Figure 2. Instrumented compilation is the first step, where the application developer compiles all or part of the application with the *prof_gen* option, which produces executable code instrumented to collect profile information. The developer then runs the instrumented code one or more times with "typical" input sets to gather execution profiles. Finally, the developer compiles the application again, this time using the *prof_use* option, which combines the gathered profiles and annotates the internal representation of the program with the observed frequencies and probabilities. Many optimizations then read the information and use it to guide their behavior. The Intel IA-64 compiler uses profile information to guide several optimizations:

1. The compiler uses profile information to integrate procedures that are most frequently executed into their call sites, thereby providing the benefits of larger program scope while minimizing code growth.
2. Profile information is also used to guide the layout of procedures and blocks within procedures to reduce instruction cache and TLB misses.
3. Finally, the compiler uses profile information to make the best use of machine instruction width and speculation features. By knowing the program's execution behavior at scheduling time, the instruction scheduler is capable of selecting the right candidates for speculation.

INTERPROCEDURAL ANALYSIS AND OPTIMIZATION

IA-64's Explicitly Parallel Instruction Computing (EPIC) architecture makes it possible to execute a large number of instructions in a single clock cycle. Therefore, scheduling to fill instruction words is of vital importance to the compiler. As with other processors, effective use of instruction caches and branch prediction are also important. Traditionally, compilers have operated on one procedure of the program at a time. However, such intraprocedural analysis and optimization is no longer sufficient to fully exploit IA-64's architectural features. The interprocedural optimizer in the Intel IA-64 compiler is profile-guided and multifile capable, so that it can efficiently provide analysis and optimization for very large regions of application code.

The Intel IA-64 compiler provides extensive support for *interprocedural* analysis and optimization. One set of key features provided by the compiler is for points-to analysis, mod/ref analysis, side effect propagation, and constant propagation. The optimizer and scheduler for the IA-64 compiler may need to move instructions over large regions in order to fill scheduling slots. In order to move operations over large regions, the compiler frequently requires knowledge of memory references within the region. Points-to analysis aids this process by accurately determining which memory locations may be referenced by a memory reference. Figure 3 illustrates this with three memory references. If the store to an address in **r37** is known not to store to the same object as the object pointed to by **r33**, then the second load may be eliminated. Furthermore, because of IA-64's data speculation feature, it may be possible to eliminate the load even if the accesses might infrequently conflict. Similarly, moving memory references across function calls requires knowledge of what is modified or referenced by the function call. This is provided by mod/ref analysis.

Analysis and optimization for IA-64 also expose the need for larger program scope for the IA-64 compared to traditional optimizers. To give the optimizer and code generator larger scope, the interprocedural optimizer provides several forms of procedure integration: inlining, cloning, and partial inlining. Inlining replaces a call site by the body of the function that would be invoked, and it provides the fullest opportunity for optimization, albeit with potentially large increases in code size. Cloning and partial inlining are used to specialize functions to particular call sites, thereby providing many of the benefits of inlining while not increasing code size significantly.

```
ld4    r32=[r33]
…
st4    [r37]=r34
…
ld4     r35=[r33]
```

Figure 3: An example of a situation requiring point-to analysis information

The compiler attempts to produce the best performance without increasing code size, as large code size can cause poor use of instruction cache and TLBs. In order to reduce the impact of code size, while retaining as much optimization as possible, the compiler uses profile information and targets procedure integration to only those sites where it is most effective. Moreover, profile guidance with knowledge of the function call graph is used to lay out functions in an order that minimizes dynamic code size, which is especially important for TLB efficiency.

Memory Disambiguation

The effectiveness and legality of many compiler optimizations rely on the compiler's ability to accurately disambiguate memory references. For example, the compiler can eliminate a large number of loads and stores with accurate memory disambiguation. Accurate information about memory independence can help exploit more instruction-level parallelism. The code scheduler requires accurate memory disambiguation to aggressively reorder loads and stores. The legality and effectiveness of loop transformations rely on the availability of accurate and detailed data-dependence information. The remainder of this section illustrates the different kinds of analyses provided in the Intel IA-64 compiler for memory disambiguation.

The simplest disambiguation cases are direct scalar or structure references. Figure 4 shows a pair of direct structure references. The compiler may disambiguate these two memory references either by determining that **a** and **b** are different memory objects or that **field1** and **field2** are non-overlapping fields.

```
a.field1 = ..
.. = b.field2
```

Figure 4: Disambiguation of direct structure references

Figure 5 shows a pair of indirect references. In general, in order to disambiguate this pair of memory references, the compiler must perform points-to analysis [12], which determines the set of memory objects that each pointer could possibly point to. Because the pointer **p** or **q** could be a global variable or a function parameter, the points-to analysis performed by the Intel IA-64 compiler is interprocedural. In some cases, two indirect references can be disambiguated based on the pointer types. For example, in an ANSI C* conforming program, a pointer to a *float* and a pointer to an *int* cannot point to the same memory object.

```
*p = ..
.. = *q
```

Figure 5: Disambiguation of indirect references

Various other language rules and simple information are useful in providing disambiguation information, even when the more expensive analyses are turned off. For example, parameters in programs that conform to the FORTRAN* standard are independent of each other and of common block elements. Therefore, an indirect reference cannot access the same location as a direct access to a variable that has not had its address taken.

```
do i= 0, n
    a(i) = a(i-1) + a(i-2);
enddo
```

Figure 6: Disambiguation of array references

Figure 6 shows an example loop with loop-carried array dependencies. The value written to a(i) in one iteration is read as a(i-1) one iteration later, and as a(i-2) two iterations later. The Intel IA-64 compiler performs array data-dependence analysis using a series of dependence tests, and it determines accurate dependence direction and distance information.

Function calls can inhibit optimization. Figure 7 shows an example where a function call may inhibit dead store elimination. If the function foo() reads ***p**, then the first store to ***p** is not dead. Interprocedural mod/ref information [10] is used to determine the set of memory locations written/read as a result of a function call.

```
*p = ..

foo();

*p = ..
```

Figure 7: Disambiguation of a memory reference and a function call

MEMORY OPTIMIZATIONS

Processor speed has been increasing much faster than memory speed over the past several generations of processor families. This phenomenon is true for the IA-64 processor family as well. Indeed, the speed differential is expected to be even larger for the IA-64 processors, since IA-64 is a high-performance architecture. As a result, the compiler must be very aggressive in memory optimizations in order to bridge the gap. The Intel IA-64 compiler applies loop-based and region-based control and data transformations in order to i) improve data access behavior with memory optimizations, ii) expose coarse grain parallelism, iii) vectorize, and iv) expose higher instruction-level parallelism. In the compiler, we implemented numerous well known and new transformations, and more importantly, we combined and tuned these transformations in special ways so as to exploit the IA-64 features for higher application performance.

In this section, we illustrate a chosen few memory optimization techniques in the compiler, and we explain how these transformations help harness the power of the IA-64 processor implementations for higher application performance. Memory optimization techniques in the Intel IA-64 compiler include, but are not limited to, i) cache optimizations, ii) elimination of loads and stores, and iii) data prefetching. All these transformations are supported by exact data dependence and temporal and spatial data reuse analyses algorithms. The compiler also applies several other well known optimization techniques such as secondary induction variable elimination, constant propagation, copy propagation, and dead code elimination.

Cache 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 available cache. Hence, we have implemented several loop transformations to improve the locality of data reference in applications. With improved locality of data reference, the majority of data references will be to higher and faster levels of memory hierarchy, so that data references incur much smaller overheads. The *linear loop transformations*, *loop fusion*, *loop distribution*, and *loop block-unroll-and-jam* are some of the transformations implemented in the compiler.

```
do i = 1, 1000
  do j = 1, 1000
     c(j) = c(j) + a(i, j)  *  b(j)
  enddo
enddo
```

```
do j = 1, 1000
  do i = 1, 1000
     c(j) = c(j) + a(i, j)  *  b(j)
  enddo
enddo
```

Figure 8: An example of a linear loop transformation

Linear Loop Transformations

Linear loop transformations are compound transformations representing sequences of loop reversal, loop interchange, loop skew, and loop scaling [17,18]. Loop reversal reverses the execution order of loop iterations, whereas loop interchange interchanges the order of loop levels in a nested loop. Loop skew modifies the shape of the loop iteration space by a compiler-determined skew factor. Loop scaling modifies a loop to have non-unit strides. As a combined effect, linear loop transformations 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, the loop interchange in Figure 8 makes references to arrays **b** and **c** both inner loop invariants, besides improving the access behavior of array **a**.

Loop Fusion

Loop fusion combines adjacent *conforming* nested loops into a single nested loop [16]. Loop fusion 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. It also 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. Loop fusion in the Intel IA-64 compiler is more aggressive than that in compilers for IA-32 or RISC processors, for example, since loop fusion in the IA-64 takes advantage of a large number of available registers. In the loop on the right-hand side of Figure 9, cache locality is improved because the accesses to array **a** are reused within the same loop. Further, it enables the compiler to replace references to arrays **a** and **d** with references to compiler-generated scalar variables.

```
do i = 1, 1000
   a(i) = x
enddo
do i = 1, 1000
   c(i) = a(i-1) + d(i-1)
   d(i) = c(i)
enddo
```

```
do i = 1, 1000
   a(i) = x
   c(i) = a(i-1) + d(i-1)
   d(i) = c(i)
enddo
```

Figure 9: An example of a loop fusion

Loop Block-Unroll-Jam

Loop unroll and jam unrolls the outer loops and fuses the unrolled copies together [14]. As a result, several outer loop iterations are merged into a single iteration in the new loop nest. For example, the **i** loop in the two-dimensional loop on the left-hand side of Figure 10 is unrolled by a factor of two. The two resulting loop nests (one for the even values of **i** and one for the odd values of **i**) are jammed together to obtain the loop on the right-hand side of Figure 10.

```
do i = 1, 2*n
   do j = 1, 2*n
      b(j, i) = a(j, i-1) + a(j, i) + a(j, i+1)

   enddo
enddo
```

```
do i = 1, 2*n, 2
   do j = 1, 2*n
      b(j, i) = a(j, i-1) + a(j, i) + a(j, i+1)
      b(j, i+1) = a(j, i) + a(j, i+1) + a(j, i+2)
   enddo
enddo
```

Figure 10: An example of a loop unroll and jam

When all loops in a loop nest are blocked, loop blocking or tiling transforms an n-dimensional loop nest into a 2n-dimensional loop nest, where the inner n-loops together scan the iterations in a block or tile of the original iteration space. Loop blocking is key to improving the cache performance of libraries and applications that manipulate large matrices of data items.

The design of the Intel IA-64 compiler unifies loop blocking, unroll and jam, and inner loop unrolling. Traditionally, compilers implement loop blocking, loop unroll and jam, and (inner) loop unrolling separately. In the process, such compilers use more than one cost model and multiple code-generation mechanisms. Whereas in fact, the three transformations are closely related. Loop blocking is a unification of strip-mining and interchange transformations. Outer loop unrolling and jamming can be viewed as blocking of the outer loops with block sizes equal to corresponding unroll factors, followed by unrolling the local iteration spaces corresponding to a block or a tile. Inner loop unrolling is a special case of blocking, where only the innermost loop is strip-mined and unrolled. All of the three transformations focus on bringing as many "related" array accesses and associated computations as possible into inner loops. In the process of doing so the outer loop unroll and jam and the inner loop unroll increase the size of the loop body.

Loop Distribution

The effect of loop distribution on loop structure is the opposite of loop fusion [16]. Loop distribution splits a single nested loop into multiple adjacent nested loops that have a similar loop structure. The computation and array accesses in the original loop are distributed across newly formed nested loops. Besides enabling other transformations, loop distribution 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.

LOAD AND STORE ELIMINATION

The IA-64 architecture has a much larger register file than traditional architectures. The IA-64 compiler takes advantage of this to eliminate loads and stores by effectively registering the memory references. In this section, we describe two optimization techniques that eliminate loads and stores: *scalar replacement* and *register blocking*.

Scalar Replacement

Scalar replacement [14,15] is a technique to replace memory references with compiler-generated temporary scalar variables, which 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 IA-64 compiler, also replaces loop invariant memory references with scalar variables defined at the appropriate levels of loop nesting.

For an example of scalar replacement of memory references, consider the loop on the left-hand side of Figure 11. In the transformed loop, all the read references to array **a** are replaced by compiler-inserted temporary scalar variables. In particular, note the replacement of loop-carried data reuse of **a(i-1)**, which is replaced by a scalar variable saved from a previous iteration. In other words, the technique is capable of scalar replacing for loop independent as well as for loop-carried (either by an input or flow dependence) data reuses.

```
do  i=2,n
   a(i)=a(i-1)* ...
   ... =a(i)-a(i-1)
enddo
```

```
t1 = a(1)
do i=2,n
   t2  = t1 * ...
   a(i) = t2
             = t2 – t1
    t1 = t2

enddo
```

Figure 11: An example of a scalar variable replacement

The IA-64 architecture provides rotating registers, which are rotated one register position each time a special loop branch instruction is executed. This hardware feature enables the compiler to map the compiler-inserted scalars directly onto the rotating registers. In particular, assignment statements of the form **t1=t2** in the example above do not have any computational overhead at all because the assignment is implicitly affected by the rotation of registers.

Scalar replacement of memory references uses the direction vectors and dependence types in the *data dependence graph* to determine the memory references that should be replaced by scalars and to determine how to perform the book-keeping required for the replacement. The compiler examines the data dependence graph for each loop and partitions the memory references based on whether the corresponding data dependencies are *input*, *flow*, or *output* dependencies. Memory references within each group are sorted by *dependence distance* and *topological order*. Memory references with loop-independent and loop-carried flow dependence are processed first, followed by memory references with loop-carried output dependence.

Register Blocking

Register blocking turns loop-carried data reuse into loop-independent data reuse. Register blocking transforms a loop into a new loop where the loop body contains iterations from several adjacent original loop iterations. Register blocking is similar to loop blocking or tiling, with relatively smaller tile sizes, followed by an unrolling of the iterations in the tile. Register blocking is demonstrated in the example in Figure 12. Register blocking takes advantage of the large register file to map the references to many of the common array elements in adjacent loop iterations onto registers.

```
do j=1,2*m                                do j=1,2*m,2
   do i=1,2*n                               do i=1,2*n,2
      a(i,j) = a(i-1,j) + a(i-1,j-1)          a(i,j) = a(i-1,j)+a(i-1,j-1)
   enddo                                      a(i+1,j) = a(i,j)+a(i,j-1)
   enddo                                      a(i,j+1) = a(i-1,j+1)+a(i-1,j)
enddo                                         a(i+1,j+1)= a(i,j+1)+a(i,j)
                                           enddo
                                          enddo
```

Figure 12: An example of register blocking

The original loop on the left-hand side of this figure has two distinct array read references in every iteration. The register blocked loop on the right-hand side of the figure has only six distinct array read references for every four iterations in the original loop. Note that two of the six references are loop independent reuses. In the Intel IA-64 compiler design, register blocking is followed by scalar replacement of memory references, since register blocking exposes new opportunities for scalar replacement of memory references.

DATA PREFETCHING

Data prefetching is an effective technique to hide memory access latency. It works by overlapping time to access a memory location with time to compute as well as time to access other memory locations [7, 19, 20]. Data prefetching inserts prefetch instructions for selected data references at carefully chosen points in the program, so that referenced data items are moved as close to the processor as possible before the data items are actually used. Note that the data prefetch instructions do not normally block the instruction stream and do not raise exceptions. Prefetching is complementary to techniques that optimize memory accesses such as loop transformations, scalar replacement of memory references, and other locality optimizations. The data prefetching algorithm implemented in the Intel IA-64 compiler makes use of data prefetch instructions and other data prefetching support features available on the IA-64.

The cost incurred while prefetching data arises from the added overhead of executing prefetch instructions as well as instructions that generate the addresses for prefetched data items. The prefetch instructions will occupy memory slots, thereby increasing resource usage. *Compute-intensive* applications normally have sufficient free memory slots. However, the benefits from prefetching have to be weighed against the increase in resource usage in *memory-intensive* applications. One must avoid prefetching for data already in the cache, because such prefetches result in an overhead and are of no benefit. Data prefetches should be issued at the right time: they should be sufficiently early so that the prefetched data item is available in cache before its use; they should be sufficiently late so that the prefetched data item is not evicted from the cache before its use. Prefetch distance denotes how far ahead a prefetch is issued for an array reference. This distance is estimated based on the memory latency, the resource requirements in the loop, and data-dependence information.

We implemented a data prefetching technique that utilizes data-locality analysis to selectively prefetch only those data references that are likely to suffer cache misses. For example, if a data reference within a loop exhibits *spatial locality* by accessing locations that fall within the same cache line, then only the first access to the cache line will incur a miss. Thus this reference can be 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. Similarly, if a data reference exhibits *temporal locality*, then only the first access must be prefetched.

```
do j = 1,n
  do i = 1,m
     a(i,j) = a(i,j) + b(0,i) + b(0,i+1)
  enddo
enddo
```

```
do j = 1,n
  do i = 1,m
     a(i,j) = a(i,j) + b(0,i) + b(0,i+1)
     if (mod(i,8) == 0)
         call prefetch(a(i+k, j))
     if (j == 1)
         call prefetch(b(0, i+k+1))
  enddo
enddo
```

Figure 13: An example of data prefetching

In the example in Figure 13, the compiler inserts prefetches for arrays **a** and **b**. The references to array **a** have spatial locality, whereas the references to array **b** have temporal locality with respect to the **j** loop iterations. Note that the calls to the prefetch intrinsic function finally map to the prefetch instructions in IA-64. In this example, **k** is the prefetch distance computed by the compiler.

The conditional statements used to control the data prefetching policy can be removed by loop unrolling, strip-mining, and peeling. However, this may result in code expansion, which can cause increased instruction cache misses. The predication support in IA-64 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. The large number of registers available in IA-64 enables prefetch addresses to be stored in registers obviating the need for register spill and fill within loops.

The IA-64 architecture provides support for memory access hints that enable the compiler to orchestrate data movement between memory hierarchies efficiently [7]. Data can be prefetched into different levels of cache depending on the access patterns. For example, if a data reference does not exhibit any kind of reuse, then it can be prefetched using a special **nta** hint to reduce cache pollution. This kind of architectural support for data movement enables the compiler to perform better data reuse analysis across loop bodies so that unnecessary prefetches are avoided.

PARALLELIZATION AND VECTORIZATION

Support for *OpenMP*[*], automatic parallelization, vectorization, and load-pair optimization are all included in the design of the IA-64 compiler. The design takes advantage of native support for parallelism on the IA-64, which includes semaphore instructions such as exchange, compare-and-exchange, and fetch-and-add, in addition to the fused multiply accumulate instruction (fma). The support for parallelism on IA-64 also includes SIMD, i.e., parallel arithmetic operations on 1, 2, and 4 bytes of data. In order to exploit the fine grain locality of data access in applications, IA-64 provides load instructions that simultaneously load a pair of double floating-point precision data items.

Parallelization

OpenMP is an industry standard to specify shared memory parallelism. It consists of a set of compiler directives, library routines, and environment variables that provide a model for parallel programming aimed at portability across shared memory systems from different vendors.

An alternative approach to parallelization is to let the compiler automatically detect parallelism and generate parallel code. The Intel IA-64 compiler has accurate data-dependence information to determine loops that can be parallelized.

Vectorization

The IA-64 floating-point SIMD operations can further improve the performance of floating-point applications. IA-64 provides the capability of doing multiple floating-point operations at the same time. The traditional loop vectorization techniques can be used to exploit this feature.

```
do j = 1, 1000
  y(j) = y(j) + a*x(j)
enddo
```

```
do j = 1, 1000, 2
  t1,t2 = ldfpd(x(j),x(j+1))
  t3,t4 = ldfpd(y(j),y(j+1))
  y(j) = t3 + a*t1
  y(j+1) = t4 + a*t2
enddo
```

Figure 14: An example of the use of load-pairs

Load-Pairs

IA-64 provides high bandwidth instructions that load a pair of floating-point numbers at a time [7]. Such load-pair instructions take a single memory issue slot, thus possibly reducing the initiation interval of the software pipelined loop. Data alignment is required to make this work. Special instructions in IA-64 can be used to avoid possible code expansion. For example, the loop in Figure 14 has three memory operations per iteration. By using load-pair operations, the number of memory references can be reduced to two per iteration.

SCALAR OPTIMIZATIONS

A primary objective of *scalar optimizations* is to minimize the number of computations and the number of references to memory. Partial redundancy elimination (PRE) [1, 2, 11] is a well known scalar optimization technique that subsumes global common subexpression elimination (CSE) and loop invariant code motion. CSE removes expressions that are always redundant (redundant on all control flow paths). PRE goes beyond CSE by attempting to remove redundancies that occur only on some control flow paths. In this paper, we highlight the use of scalar optimizations to eliminate loads and stores.

Traditional PRE

An expression at program point **p** in the program control flow graph (CFG) is fully redundant if the same expression is already **available**. An expression **e** is said to be **available** at a point **p** if along every control flow path from the program entry to **p** there is an instance of **e** that is not subsequently **killed** by a redefinition of its operands. Figure 15 shows an example of a fully redundant expression and its elimination by CSE. The redundancy is removed by saving the value of the redundant expression in a temporary variable and then later reusing that value instead of reevaluating the expression.

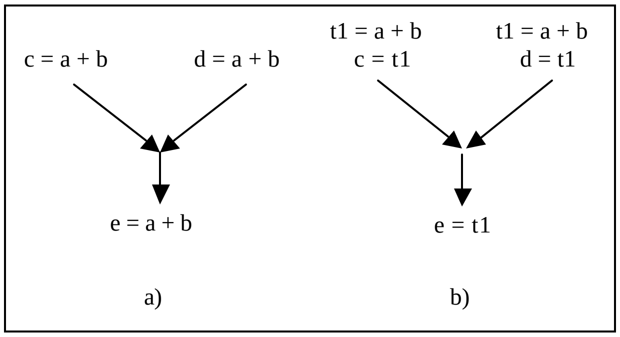

Figure 15: (a) expression a + b is fully available, (b) elimination of common subexpression

An expression **e** is **partially available** at a point **p** if there is an instance of **e** along only some of the control flow paths from the program entry to **p**. Figure 16 shows an example of a partially redundant expression and PRE. The partial redundancy is removed by inserting a copy of the redundant expression on the control flow paths where it is not available, making it fully redundant.

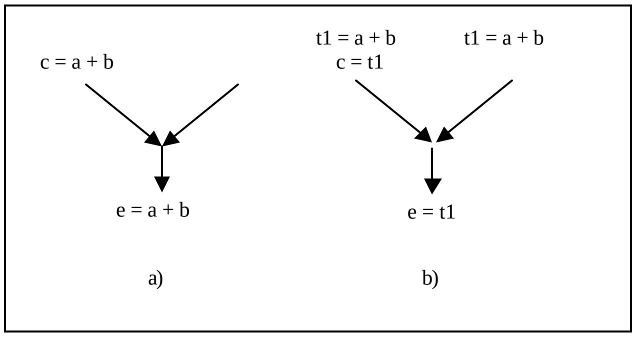

Figure 16: (a) expression a + b is partially available (b) elimination of partial redundancy

PRE can move the loop invariant to outside the loop as shown in Figure 17. The expression **q* is available on the loop back-edge, but not on entry to the loop. After inserting *t2 = *q* in the loop preheader, **q* is fully available and can be removed from the loop.

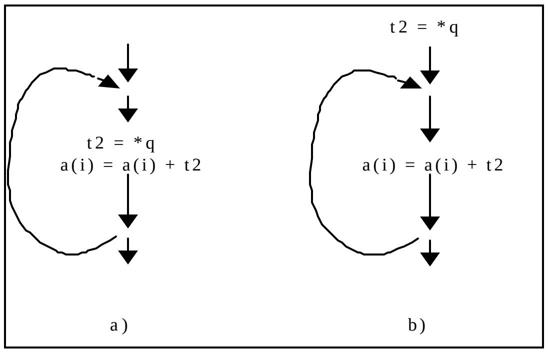

Figure 17: Example of loop-invariant code motion

Note, however, that the optimizer must be careful not to insert a copy of an expression at a point that would cause the expression to be evaluated when it was not evaluated in the original source code. Figure 18 shows such an example. The insertion of an expression *e* at a program point *p* is said to be *down-safe* if along every control flow path from *p* to the program exit there is an instance of *e* such that the inserted expression is available at each later instance. In Figure 18, the insertion of *t1 = *q* is not down-safe. There are two aspects to down-safety. The first is that an unsafe insertion may create an incorrect program. For example, in Figure 18, the expression **q* is executed before checking if *q* is a null pointer. Second, an unsafe insertion reduces the number of instructions along one path at the expense of another path. In Figure 18, the redundancy is eliminated for the left-most path, but an extra instruction, *t1 = *q*, is executed on the right-most path.

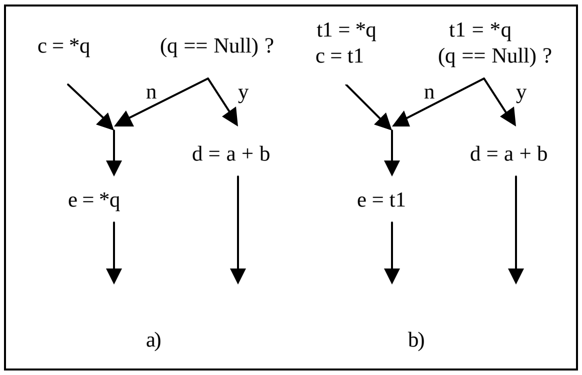

Figure 18: (a) expression *q is partially available, (b) violation of down-safety

Extended PRE for IA-64

The standard PRE algorithm removes all the redundancies possible with safe insertions. We have extended PRE to use control speculation to remove redundancies on one control flow path, perhaps at the expense of another, less important control flow path. In the example in Figure 18, assume that the left-most control flow path is executed much more frequently than the right-most path. If the redundancy on the left-most path could be removed without producing an incorrect program, overall performance would be improved even though an extra instruction is executed on the right-most path.

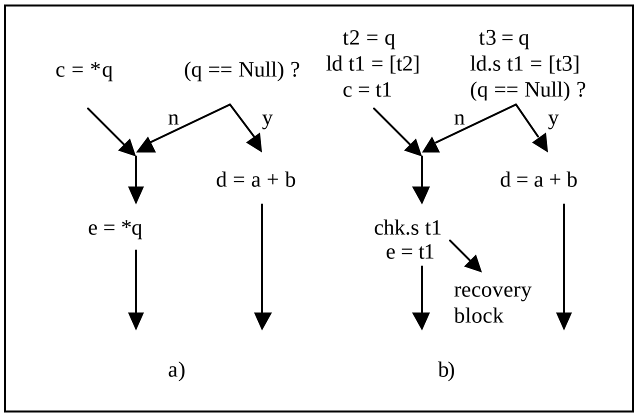

Figure 19: Redundancy elimination using control speculation

Figure 19 shows how the redundancy in Figure 18 could be removed using the IA-64 support for control speculation. The insertion of *q is done using a speculative load, and a check instruction is added in place of the redundant load. Executing the check is preferable to executing the redundant load because the check does not use memory system resources and because the latency of the load is hidden by executing it earlier. Also, elimination of the redundant load may expose further opportunities for redundancy elimination in the instructions that depend on the load.

Removal of redundant loads can sometimes be inhibited by intervening stores. In Figure 20 (a), the loop-invariant load *p cannot be removed unless the compiler can prove that the store *q does not access the same memory location. The process of determining whether or not two memory references access the same location is called *memory disambiguation* and was described earlier in this paper.

If the compiler can determine that there is an unknown, but small probability that **p* and **q* access the same memory location, the loop invariant load and the add that depends on it can be removed using the IA-64 support for data speculation as shown in Figure 20 (b). The insertion of **p* in the preheader is done using an advanced load, and a check instruction is added in place of the original redundant load. If the store **q* accesses the same memory location as the load **p*, a branch to a recovery code block will be taken at the check instruction. The recovery block contains code to reload **p* and re-execute *t4=t2 + t3*. If the store **q* and load **p* access different memory locations, then only the check is executed instead of the redundant load and add.

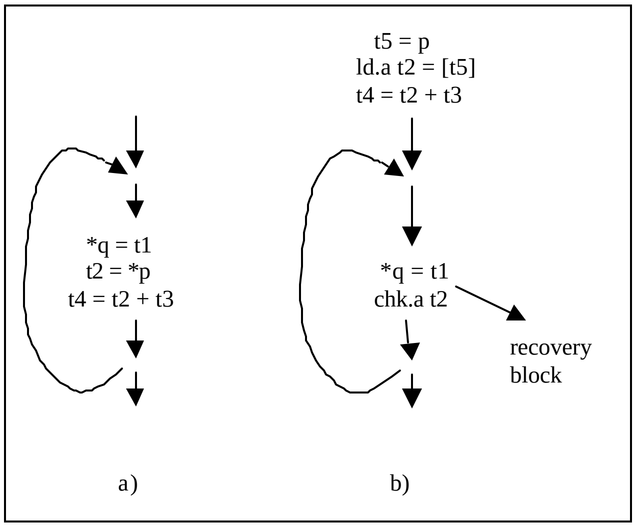

Figure 20: Removal of loop-invariant load using data speculation

Partial Dead Store Elimination

In contrast to PRE which removes redundant loads, Partial Dead Store Elimination (PDSE) removes redundant stores in the program. Figure 21 shows an example of PDSE. The partial redundancy is removed by inserting a copy of the partially dead store into the control flow paths where it is not dead, making it fully redundant.

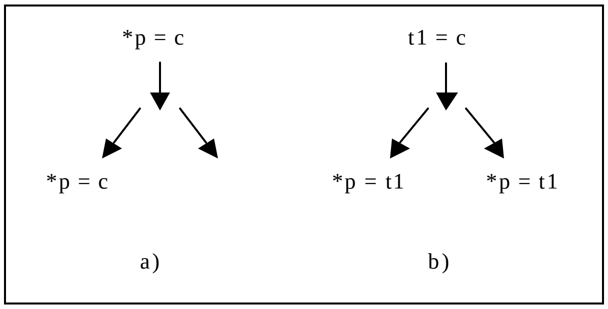

Figure 21: (a) store *p is partially dead, (b) elimination of partially dead store

As with PRE, the compiler must be careful when inserting stores to avoid executing a store when it should not be executed. Figure 22 shows an example of an incorrect insertion of a store. In Figure 22b, the store *p = t1 on the right is executed even if the path containing d=a+b is executed. In the original program in Figure 22a, no store to *p is executed when the path containing d=a+b is executed.

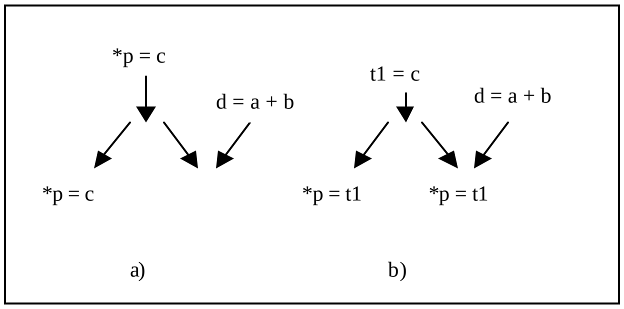

Figure 22: Example of incorrect insertion of a store

Figure 23 shows how the redundancy in Figure 22 could be removed using a predicated store. In Figure 23b, the redundancy on the left-most path is removed by inserting a predicated store. Instructions are required to set the predicate p2 to 1 when the store should be executed, and to 0 when it should not be executed. In Figure 23b, suppose that the left-most path is executed much more frequently than the right-most path. On the left-most path, executing p2=1 is preferable to executing the store, because the p2=1 does not use memory system resources. In some cases, an appropriate instruction to set p2 may already exist as a result of the if-conversion or another optimization, thereby reducing the cost of predicating the store.

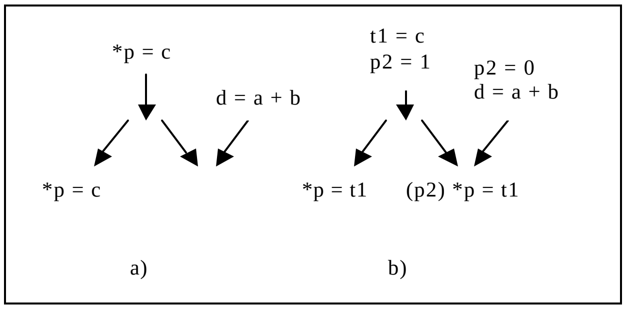

Figure 23: Elimination of partially dead store using predication

THE SCHEDULER AND CODE GENERATOR

The *scheduler and code generator* in the compiler make effective use of predication, speculation, and rotating registers by global code scheduling, software pipelining, and rotating-register allocation. In this section, we provide an overview of predication techniques, software-pipelining, global code scheduling, and register allocation.

Predication

Branches can decrease application performance by consuming hardware resources at execution time and by restricting instruction-level parallelism. Predication is one of several features IA-64 provides to improve the efficiency of branches [7]. Predication is the conditional execution of an instruction that is based on a qualifying predicate, where the qualifying predicate is a predicate register whose value determines whether or not the instruction must execute normally. If the predicate is true, the instruction updates the computation state; otherwise, it generally behaves like a *nop*. The execution of most IA-64 instructions is gated by a qualifying predicate. The values of predicate registers can be set with a variety of compare and test bit instructions. Predicated execution avoids branches, and it simplifies compiler optimizations by converting a control dependence to a data dependence.

The Intel IA-64 compiler eliminates branches through predication and thus improves the quality of the code's schedule. The benefits are particularly pronounced for branches that are hard to predict. The compiler uses a transformation called *if-conversion*, where conditional execution is replaced with predicated instructions. For a simple example, look at the following code sequence:

```
if (a <b)
    s = s + a
else
    s = s + b
```

can be rewritten in IA-64 without branches as

```
cmp.lt p1, p2 = a,b
(p1) s = s + a
(p2) s = s + b
```

Since instructions from opposite sides of the conditional are predicated with complementary predicates, they are guaranteed not to conflict, and the compiler has more freedom when scheduling to make the best use of hardware resources.

Predication enables the compiler to perform upward and downward code motion with the aim of reducing the dependence height. This is possible because predicating an instruction replaces a control dependence with a data dependence. If the data dependence is less constraining than the control dependence, such a transformation may improve the instruction schedule. The compiler also uses predication to efficiently implement *software pipelining* discussed in the next section.

Note that predication may increase the critical path length because of unbalanced dependence heights or over-usage of particular resources, such as those associated with memory operations. The compiler has to weigh this cost against the profitability of predication by considering various factors such as the branch misprediction probabilities, miss cost, and parallelism.

The IA-64 supports special parallel compare instructions that allow compound expressions using the relational operators *and* and *or* to be computed in a single cycle. These instructions can be used to reduce the control path by reducing the total number of branches. IA-64 also has the support of multiway branches, where different predicates can be used to branch to different targets within an instruction group.

Software Pipelining

Software pipelining [3,4] in the Intel IA-64 compiler improves the performance of a loop by overlapping the execution of several iterations. This improves the utilization of available hardware resources by increasing the instruction-level parallelism. Figure 24 shows several overlapped loop iterations.

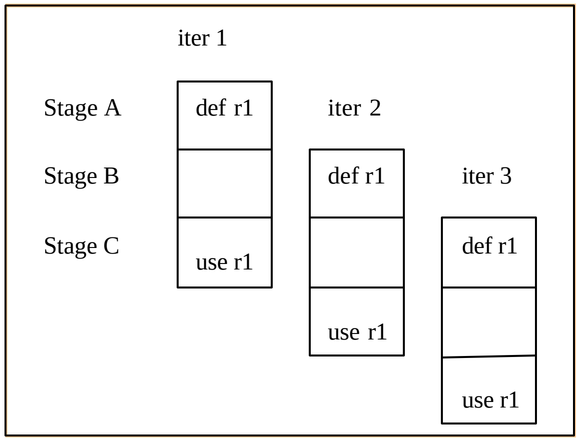

Figure 24: Pipelined loop iterations

Analogous to hardware pipelining, each iteration is divided into stages. In this example, each iteration is divided into three stages, and up to three iterations are executed simultaneously. The number of cycles between the start of successive iterations in a software-pipelined loop is called the Initiation Interval (II), and each stage is II cycles in length.

Software-pipelined loops have three execution phases: the prolog phase, in which the software pipeline is filled; the steady-state kernel phase, in which the pipeline is full; and the epilog phase, in which the pipeline is drained. In RISC architectures, these three execution phases are implemented using three distinct blocks of code as shown in Figure 25.

In IA-64, rotating predicates [5, 6, 7] are used to control the execution of the stages during the prolog and epilog phases, so that only the kernel loop is required. This reduces code size. During the first iteration of the kernel loop, stage A of source iteration 1 is enabled. During kernel iteration 2, stage A of source iteration 2 and stage B of source iteration 1 are enabled, and so on. During the epilog phase, the hardware support sequentially disables stages.

In order to illustrate another advantage of IA-64 support for software-pipelining, consider register r1 defined early in each iteration and used late in each iteration, as shown in Figure 24. When the iterations overlap, the separate lifetimes also overlap. The definitions of r1 from iterations two and three overwrite the value of r1 that is needed by the first iteration. In RISC architectures, the kernel loop must be unrolled three times so that each of the three overlapped lifetimes can be assigned to different registers to avoid clobbering of values [4, 6]. In IA-64, on the other hand, unrolling of the kernel loop is unnecessary because rotating registers can be used to perform renaming of the registers, thus reducing the code size [5, 6, 7].

The Intel IA-64 compiler uses a software pipelining algorithm called modulo scheduling [8]. In modulo scheduling, a minimum candidate II is computed prior to scheduling. This candidate II is the maximum of the resource-constrained minimum II and the recurrence-constrained (dependence cycle constrained) minimum II.

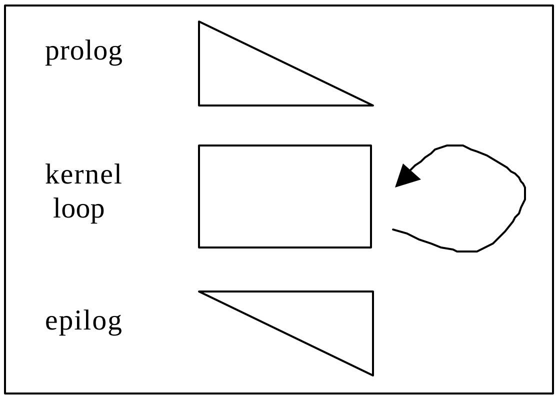

Figure 25: Execution phases in software-pipelined loops: IA-64 supports kernel only software-pipelined loops

The Intel compiler pipelines both counted loops and while loops. Loops with control flow or with early exits are transformed, using if-conversion, into single block loops suitable for pipelining. Outer loops can also be pipelined, and several optimizations are done to reduce the recurrence-constrained II.

Global Code Scheduling

The Intel IA-64 compiler contains both a global code scheduler (GCS) [9] and a fast local code scheduler. The *GCS* is the primary scheduler, and it schedules code over acyclic regions of control flow. The local code scheduler rearranges code within a basic block and is run after register allocation to schedule the spill code.

The GCS allows arbitrary acyclic control flow within the scheduling scope referred to as a scheduling region. There is no restriction placed on the number of entries into or exits from the scheduling region. The GCS also enables code scheduling across inner loops by abstracting them away through nesting. The GCS employs a new path-based data dependence representation that combines control flow and data-dependence information to make data analysis easy and accurate.

Most scheduling techniques find it difficult to make good decisions on the generation and scheduling of compensation code. This problem is addressed by the *GCS* using *wavefront scheduling* and *deferred compensation*. The GCS schedules along all the paths in a region simultaneously. The *wavefront* is a set of blocks that represents a strongly independent cut set of the region. Instructions are only scheduled into blocks on the wavefront. The wavefront can be thought of as the boundary between scheduled and yet to be scheduled code in the scheduling region.

Control flow in program code can make the task of code motion difficult and complicated. In the GCS, tail duplication is done at the instruction level and is referred to as *P-ready code motion*. An instruction is duplicated based on a cost and profitability analysis.

Register Allocation

Register allocation refers to the task of assigning the available registers to variables such that if two variables have overlapping live ranges, they are assigned separate registers. In doing so, the register allocator attempts to maximally utilize all the available registers. The large number of architectural registers in IA-64 enables multiple computations to be performed without having to frequently spill and copy intermediate data to memory. Register allocation can be formulated as a graph coloring problem where nodes in the graph represent live ranges of variables and edges represent a temporal overlap of live ranges. Nodes sharing an edge must be assigned different colors or registers.

When using predication, it is particularly common for sequences of instructions to be predicated with complementary predicates. In such cases, it is possible to use the same registers for two separate variables, even when their live ranges seem to overlap. This is because the compiler can figure out that only one of the variables will be updated depending on the predicate values. For example, in the code sequence of Figure 26, the same register is allocated for both *v1* and *v2* since *p1* and *p2* are complementary predicates.

```
(p1) v1 = 10
(p2) v2 = 20 ;;
(p1) st4 [v10]= v1
(p2) v11 = v2 + 1 ;;

    ----→

 (p1) r32 = 10
(p2) r32 = 20 ;;
(p1) st4 [r33]= r32
(p2) r34 = r32 + 1 ;;
```

Figure 26: An example of register allocation

CONCLUSION

In this paper, we provided an overview of the Intel IA-64 compiler. We described the organization of the compiler, as well as the features and functionality of several optimization techniques. The compiler applies region and loop-level control and data transformations, as well as global optimizations, to programs. All the optimization techniques in the compiler are aware of profile information and effectively use interprocedural analysis information. The optimizations effectively target three main goals while compiling an application: i) to minimize the overhead of memory accesses, ii) to minimize the overhead of branches, and iii) to maximize instruction-level parallelism. We described how the optimization techniques in the Intel IA-64 compiler take advantage of the IA-64 architectural features for improved application performance. We illustrated the techniques with example codes, and we highlighted the benefits as a result of specific optimizations. The Intel IA-64 compiler incorporates all the infrastructure and technology necessary to leverage the IA-64 architecture for improved integer and floating-point performance.

ACKNOWLEDGMENTS

We thank the members of the IA-64 compiler team for their contributions to the compiler technology described in this paper. We also thank the reviewers for their excellent and useful suggestions for improvement.

REFERENCES

[1] E. Morel and C. Renvoise, “Global optimization by suppression of partial redundancies,” *Comm. ACM*, 22(2), February 1979, pp. 96-103.

[2] F. Chow, S. Chan, R. Kennedy, S. Liu, R. Lo, and P. Tu, “A new algorithm for partial redundancy elimination based on SSA form,” in *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, June 1997, pp. 273-286.

[3] B. R. Rau and C. D. Glaeser, “Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High-Performance Scientific Computing,” in *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, October 1981, pp. 183-198.

[4] 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, pp. 318-328.

[5] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt, “Overlapped Loop Support in the Cydra 5,” in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 26-38.

[6] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai, “Code Generation Schema for Modulo-Scheduled Loops,” in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992, pp. 158-169.

[7] *IA-64 Application Developer’s Architecture Guide*, Order Number 245188-001, May 1999.

[8] B. R. Rau, “Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops,” in *Proceedings of the 27th International Symposium on Microarchitecture*, December 1994, pp. 63-74.

[9] J. Bharadwaj, K.N. Menezes, and C. McKinsey, “Wavefront Scheduling: Path-Based Data Representation and Scheduling of Subgraphs,” to appear in *Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO32)*, (Haifa, Israel), December 1999.

[10] K. D. Cooper and K. Kennedy, “Interprocedural Side-Effect Analysis in Linear Time,” in *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988, pp. 57-66.

[11] J. Knoop, O. Ruthing, and B. Steffen, “Lazy code motion” in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 224-234, June 1992.

[12] B. Steensgaard, “Points-to Analysis in Almost Linear Time” in *Proceedings of the Twenty-Third Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 32-41, January 1996.

[13] C. Dulong, "The IA-64 Architecture at Work," *IEEE Computer*, July 1998.

[14] S. Carr, "Memory-Hierarchy Management" Ph.D. Thesis, Rice University, July 1994.

[15] S. Carr and K. Kennedy, "Scalar Replacement in the Presence of Conditional Control Flow," *Technical Report CRPC-TR92283*, Rice University, November 1992.

[16] S. Muchnik, *Advanced Compiler Design Implementation*, Morgan Kaufman, 1997.

[17] M. Wolf and M. Lam, *A Loop Transformation Theory and an Algorithm to Maximize Parallelism*, Parallel Distributed Systems, Volume 2 (4), pp. 452-471, October, 1991.

[18] W. Li and K. Pingali, "A Singular Loop Transformation Framework Based on Non-Singular Matrices," *International Journal of Parallel Programming*, Volume 22 (2), 1994.

[19] T. Mowry, "Tolerating Latency Through Software-Controlled Data Prefetching," Ph.D. Thesis, Stanford University, March 1994, Technical Report CSL-TR-94-626.

[20] V. Santhanam, E. Gornish, and W. Hsu, "Data Prefetching on the HP PA-8000," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997, pp. 264-273.

AUTHORS' BIOGRAPHIES

Carole Dulong has been with Intel for over nine years. She is the co-manager of the IA-64 compiler group. Prior to joining Intel's Microcomputer Software Laboratory, she was with the IA-64 architecture group, where she headed the IA-64 multimedia architecture definition and the IA-64 experimental compiler development. Her e-mail is carole.dulong@intel.com.

Rakesh Krishnaiyer received a B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Madras in 1993, and an M.S. and Ph.D. degree from Syracuse University in 1995 and 1998, respectively. He currently works on high-level optimizations for the IA-64 compiler at Intel. His research interests include compiler optimizations, high-performance parallel and distributed computing systems, and computer architecture. His e-mail address is rakesh.krishnaiyer@intel.com.

Dattatraya Kulkarni received his Ph.D. degree in computer science from the University of Toronto, Toronto, Canada in 1997. He has been working on compiler optimization techniques for uniprocessor and multiprocessor systems for the past nine years. He is currently with the Intel IA-64 compiler team. His e-mail is dattatraya.kulkarni@intel.com.

Daniel Lavery received a Ph.D. degree in electrical engineering from the University of Illinois in 1997. As a member of the IMPACT research group under Professor Wen-mei Hwu, he developed new software pipelining techniques. Since joining Intel in 1997, he has worked on the architecture and compilers for IA-64. He is currently an IA-64 compiler developer in Intel's Microcomputer Software Laboratory. His e-mail address is daniel.m.lavery@intel.com.

Wei Li leads and manages the high-level optimizer group for IA-64. He has published many research papers in the areas of compiler optimizations, parallel and distributed computing, and scalable data mining. He served on the program committees for parallel and distributed computing conferences. He received his Ph.D. degree in computer science from Cornell University, and he was on the faculty at the University of Rochester before joining Intel. His e-mail address is wei.li@intel.com.

John Ng received an M.S. degree in computer science from Rutgers University in 1975 and a B.Sc. degree in mathematics from Illinois State University in 1973. He joined the Intel IA-64 compiler team three years ago. Prior to that he was a Senior Programmer at IBM Corporation. He has been working on compiler optimizations, vectorization, and parallelization since 1982. His e-mail is john.ng@intel.com.

David Sehr received his B.S. degree in physics and mathematics from Butler University in 1985. He received his M.S. and Ph.D. degrees from the University of Illinois working under the direction of David Padua and Laxmikant Kale. His thesis work was in the area of restructuring compilation of logic languages. He joined Intel in 1992 and since that time has worked on loop optimizations, scalar optimizations, and interprocedural and profile-guided optimizations for IA-32 and IA-64. He is currently the group leader for the IA-64 compiler scalar optimizer. His e-mail address is david.sehr@intel.com.