Preface

Lin Chao
Editor
Intel Technology Journal

With the close of the year 1999, it is appropriate that we look forward to Intel's next generation architecture--the Intel Architecture (IA)-64. The IA-64 represents a significant shift for Intel architecture: it moves from 32-bits to 64-bits. Targeted for production in mid-2000, the Itanium™ processor will be the first IA-64 processor. One of Intel's key aims is to facilitate a transition of this magnitude. To this end, Intel and key industry suppliers are working to ensure that a complete set of "ingredients" is available for the IA-64 architecture. This includes operating systems, compilers, and application development tools that are "64-bit capable." In this issue of the *Intel Technology Journal*, you will learn about Intel's efforts in IA-64 software technology.

In the first papers, Intel's own IA-64 compiler efforts are discussed. The first paper gives an overview of Intel's production compiler, code named "Electron." We then move to the second and third papers where software development tools for the IA-64 architecture are discussed. As is often the case with a brand new architecture, software companies start developing software in advance of actual hardware. SoftSDV (a software-based system development vehicle) is a tool that simulates IA-64 hardware platforms in lieu of actually having the hardware. This tool assists engineers in porting commercial operating systems and applications to the IA-64. Intel's IA-64 Assembler is described in the fourth paper. The Assembler can simplify IA-64 assembly language programming.

Validation is a critical function in the testing of new circuits such as those on IA-64 silicon. The fourth paper describes the porting of the Linux* and Mach* operating systems and how they run on software simulators to exercise operating system related functionality in the IA-64 architecture.

The final two papers discuss the floating-point functions on the IA-64 architecture. Fast and accurate computation of transcendental functions (e.g. sin, cos, and exp) and the implementation of floating-point operations are also discussed.

Copyright © Intel Corporation 1999. This publication was downloaded from http://www.intel.com/.
Legal notices at http://www.intel.com/sites/corporate/tradmarx.htm

The Challenges of New Software for a New Architecture

By Richard Wirt,
Intel Fellow and Director, MicroComputer Software Labs
Intel Corp.

In this Q4, 1999 issue of the *Intel Technology Journal*, we look at some of the software efforts that have gone into bringing out the new IA-64 architecture. Early in the development of the IA-64 architecture, we set very aggressive goals for the software compilers, floating point performance, and simulation environment.

Over the years, the compiler has become a very important factor in contributing to the utilization of a processor's architecture. The Intel IA-64 compiler, code named Electron, was considered part of the IA-64 architecture. We had compiler architects working side by side with CPU architects. This was very important since the IA-64 is a new architecture that exploits many new concepts, and the microarchitecture depends on the compiler to manage many of the resource dependencies. Not only has this compiler served to bring up many of the initial operating systems and applications, it has also served to help other third party compiler vendors to understand how to generate code for this new architecture. The compiler has met the challenge of pushing the boundaries and of achieving our initial architecture goals.

Being a new architecture, the IA-64 architecture provided Intel with the opportunity to take another look at the floating point on the IA-64 architecture. We set a goal of making it faster, making it fully IEEE compliant, and of achieving a near 0.5 units in the last place of precision for the transcendental functions libraries. The width of the architecture allowed us to take another look at the traditional algorithms. Since divide and square root are executed in software in the Itanium™ processor implementation of the IA-64 architecture, we have also formally proven the algorithms used.

Perhaps the biggest challenge was to simultaneously bring up and debug a new architecture, new chipsets, new compilers, and new versions of the operating systems. To do this, we set our sights on building a full system-level simulator that functionally behaved faithfully at the register and I/O ports levels as the first platforms were being built. Additionally, we wanted to be able to add performance simulators for the CPU, caches, and chipsets. The software simulator became known as the SoftSDV. It met its requirements to be functionally equivalent to the first software development vehicles manufactured for use by the ISVs and OS venders. Well before first silicon was available for the Itanium processor, we were running the firmware, all the major operating systems, and many major applications on the SoftSDV. This served its purpose very well, as we were able to run the same binaries on the real SDVs, using new silicon, within hours of their availability from Intel's manufacturing fabs.

With this issue of the *Intel Technology Journal*, we hope you will get a better insight into the software behind Intel's new 64-bit

architecture and gain an appreciation of the outstanding efforts of the many people on the software teams that contributed to the IA-64 software technologies.

Copyright © Intel Corporation 1999. This publication was downloaded from http://www.intel.com/.

Legal notices at http://www.intel.com/sites/corporate/tradmarx.htm

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.

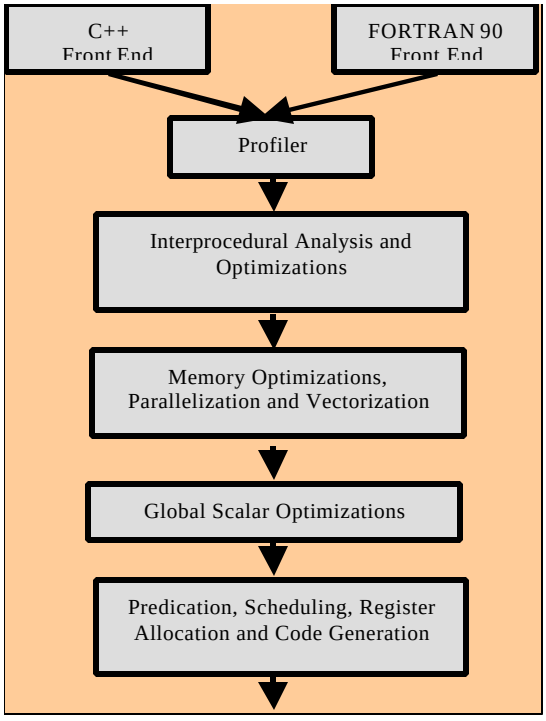


Figure 1: Organization of the Intel IA-64 compiler

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.

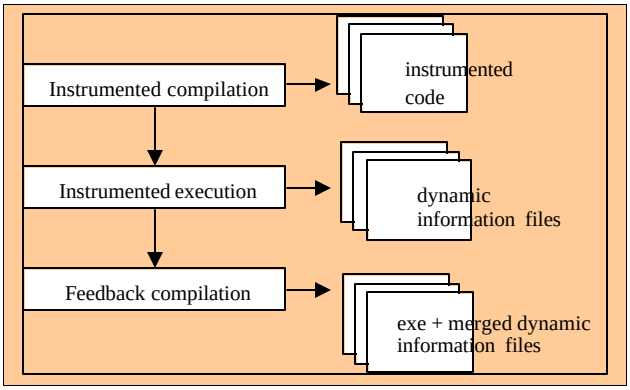


Figure 2: Steps in dynamic profile-guided compilation

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.

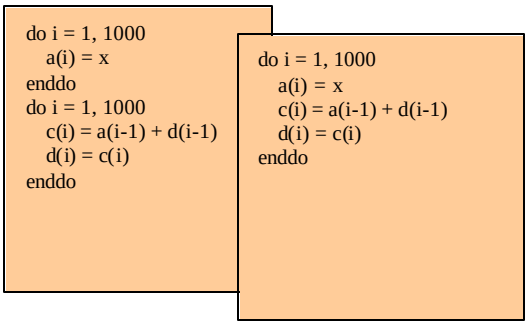


Figure 9: An example of a loop fusion

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.

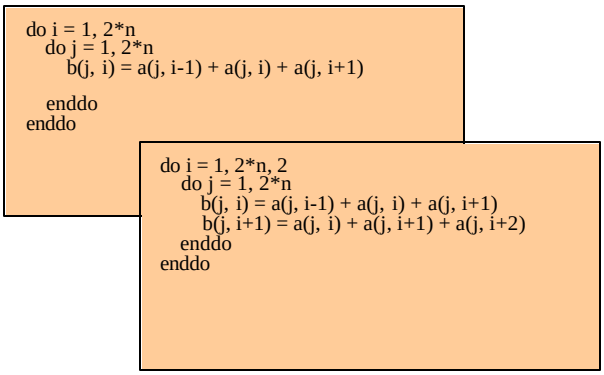


Figure 10: An example of a loop unroll and jam

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.

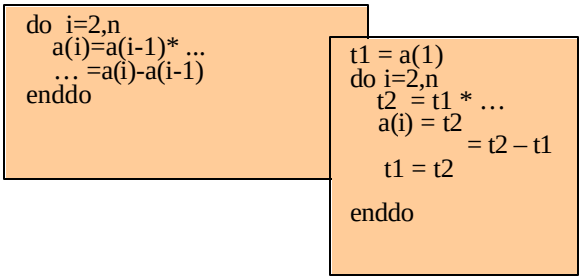


Figure 11: An example of a scalar variable replacement

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.

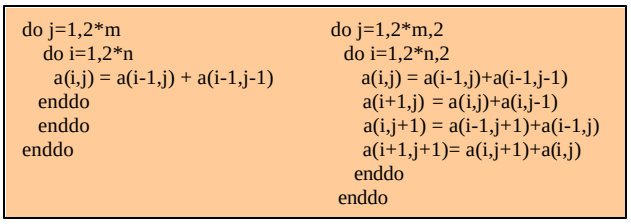


Figure 12: An example of register blocking

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.

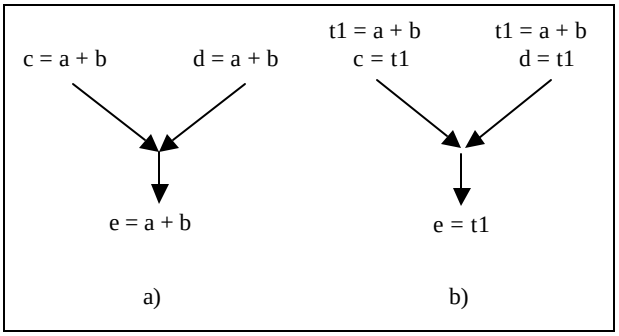


Figure 15: (a) expression a + b is fully available, (b) elimination of common subexpression

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.

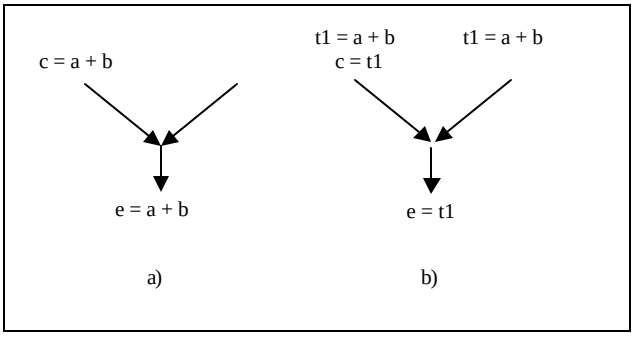


Figure 16: (a) expression a + b is partially available (b) elimination of partial redundancy

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.

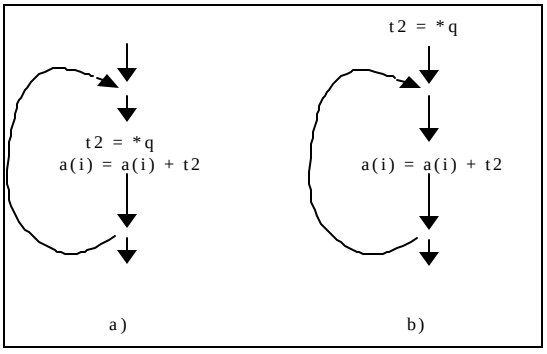


Figure 17: Example of loop-invariant code motion

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.

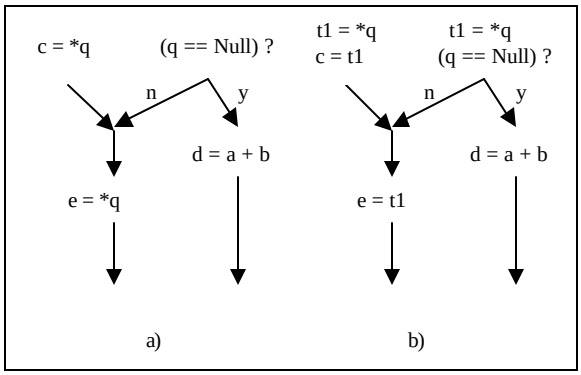


Figure 18: (a) expression *q is partially available, (b) violation of down-safety

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.

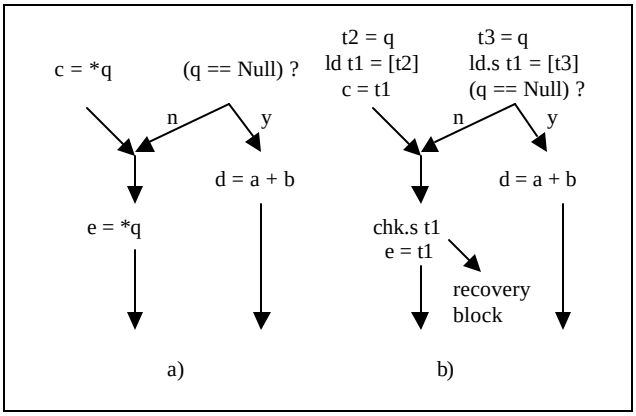


Figure 19: Redundancy elimination using control speculation

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.

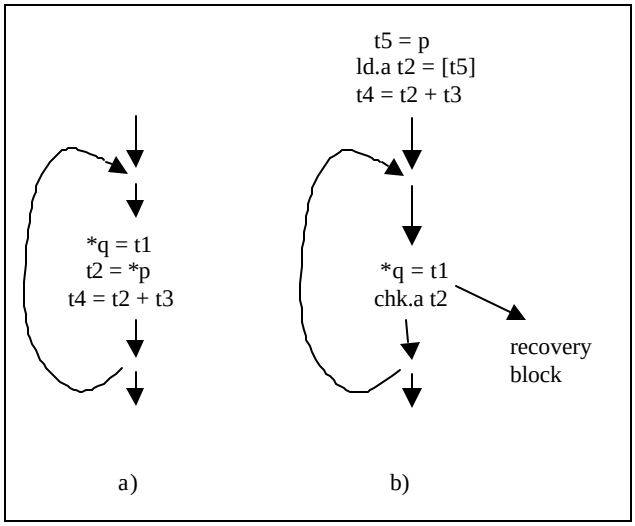


Figure 20: Removal of loop-invariant load using data speculation

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.

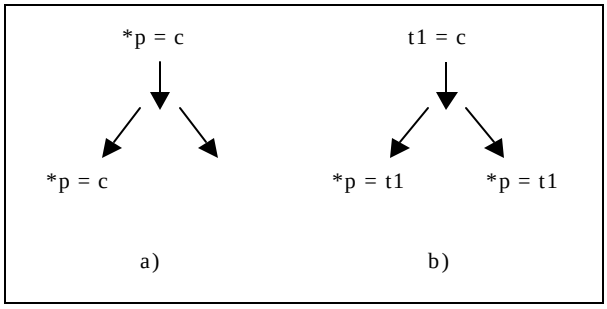


Figure 21: (a) store *p is partially dead, (b) elimination of partially dead store

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.

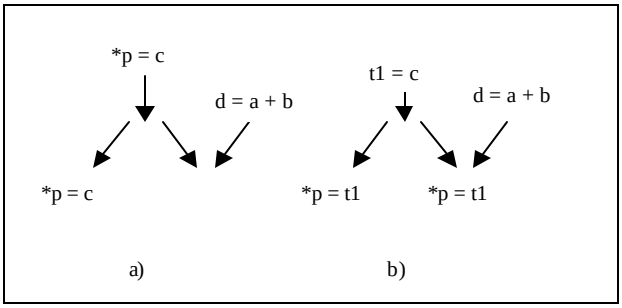


Figure 22: Example of incorrect insertion of a store

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.

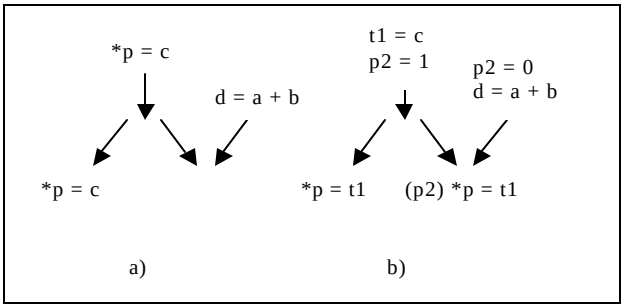


Figure 23: Elimination of partially dead store using predication

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.

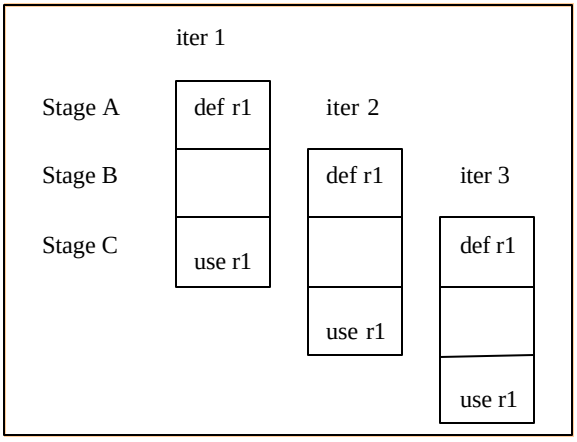


Figure 24: Pipelined loop iterations

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.

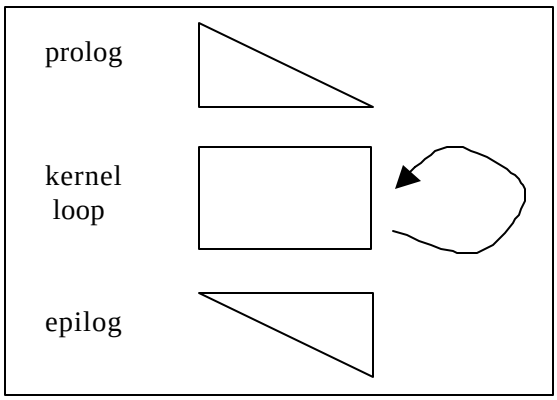


Figure 25: Execution phases in software-pipelined loops: IA-64 supports kernel only software-pipelined loops

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.

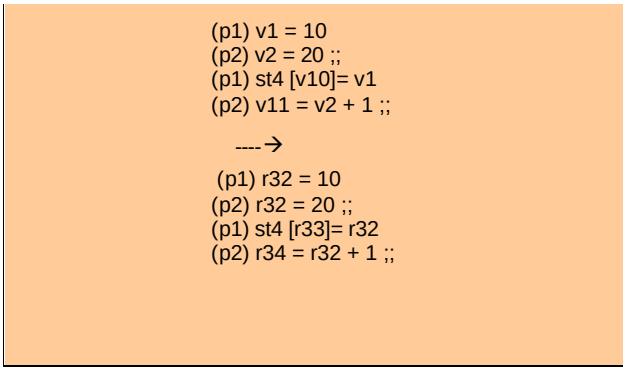


Figure 26: An example of register allocation

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.

SoftSDV: A Presilicon Software Development Environment for the IA-64 Architecture

Richard Uhlig, Microprocessor Research Labs, Oregon, Intel Corp.
Roman Fishtein, MicroComputer Products Lab, Haifa, Intel Corp.
Oren Gershon, MicroComputer Products Lab, Haifa, Intel Corp.
Israel Hirsh, MicroComputer Products Lab, Haifa, Intel Corp.
Hong Wang, Microprocessor Research Labs, Santa Clara, Intel Corp.

Index words: presilicon software development, dynamic binary translation, dynamic resource analysis, processor performance simulation, IO-device simulation

Abstract

New instruction-set architectures (ISAs) live or die depending on how quickly they develop a large software base. This paper describes SoftSDV, a presilicon software-development environment that has enabled at least eight commercial operating systems and numerous large applications to be ported and tuned to IA-64, well in advance of Itanium™ processor's first silicon. IA-64 versions of Microsoft Windows* 2000 and Trillian Linux* that were developed on SoftSDV booted within ten days of the availability of the Itanium processor.

SoftSDV incorporates several simulation innovations, including *dynamic binary translation* for fast IA-64 ISA emulation, *dynamic resource analysis* for rapid software performance tuning, and *IO-device proxying* to link a simulation to actual hardware IO devices for operating system (OS) and device-driver development. We describe how SoftSDV integrates these technologies into a complete system that supports the diverse requirements of software developers ranging from OS, firmware, and application vendors to compiler writers. We detail SoftSDV's methods and comment on its speed, accuracy, and completeness. We also describe aspects of the SoftSDV design that enhance its flexibility and maintainability as a large body of software.

* Other brands and names are the property of their respective owners.

INTRODUCTION

The traditional approach to fostering software development for a new ISA such as IA-64 is to supply programmers with a hardware platform that implements the new ISA. Such a platform is commonly known as a software-development vehicle (SDV) and suffers from a key dependency: it cannot be assembled until first silicon of the processor has been manufactured. This paper describes how Intel eliminated this dependency for IA-64 by building an SDV entirely in software through the simulation of all processor and IO-device resources present in an actual hardware IA-64 SDV. This simulation environment, which we call SoftSDV, has enabled substantial development of IA-64 software, well in advance of an Itanium processor's first silicon.

A principal design goal for SoftSDV is that it support development all along the software stack, from firmware and device drivers to operating systems and applications (see Figure 1). The performance of each of these layers of software is dependent upon optimizing compilers, which themselves must be carefully tuned to IA-64 [1, 2]. Each of these types of software development has a different set of requirements with respect to simulation *speed*, *accuracy*, and *completeness*.

Application developers, for example, are primarily interested in simulation speed, whereas optimizing-compiler writers value accuracy in processor-resource modeling so that they can evaluate the effectiveness of their code-generation algorithms. OS, firmware, and device-driver developers, on the other hand, require completeness in the modeling of platform IO devices and

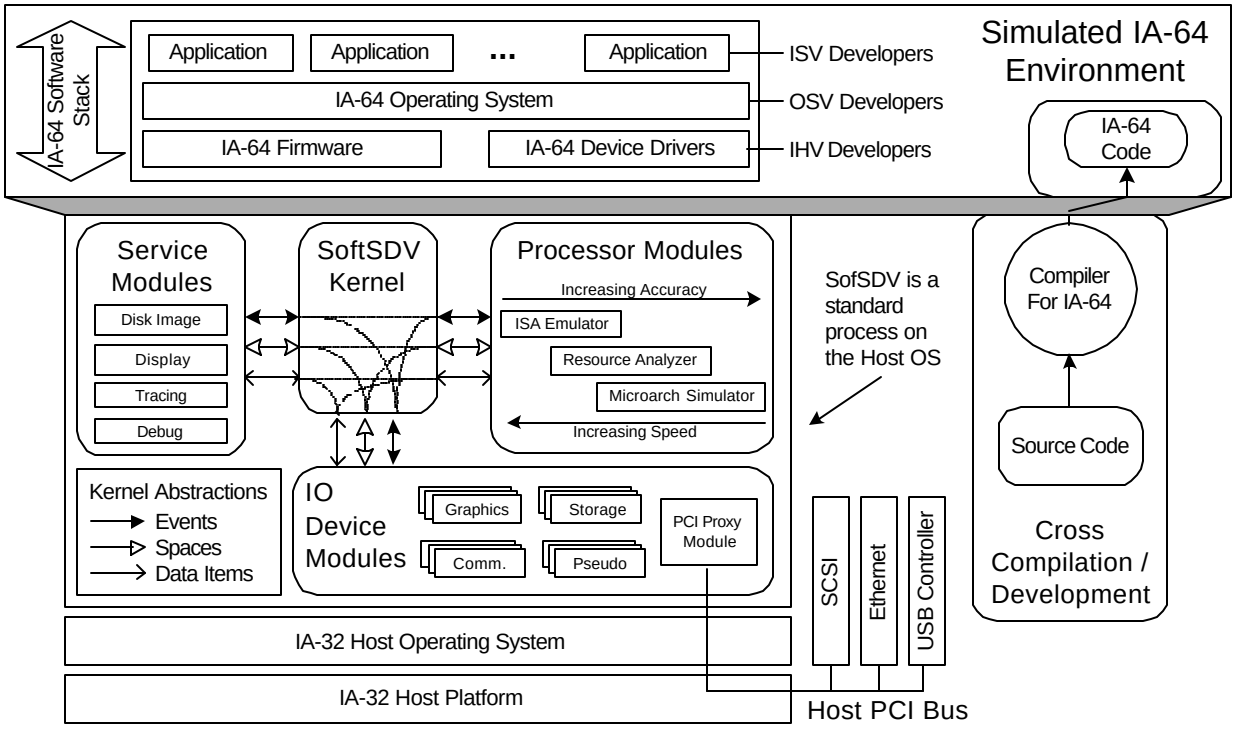


Figure 1: The SoftSDV software architecture

system-level processor functions (e.g., virtual-memory management, interrupt processing, etc.).

Not only do requirements vary based on the type of code, but their relative importance shifts depending on the stage of software development. Early efforts to port and debug code to IA-64 are best aided by very fast ISA emulation, whereas successive tuning of code for performance requires ever-increasing levels of simulation accuracy. Similarly, in the early stages of porting an OS to IA-64, it is sufficient to model a basic set of boot IO devices. However, once that OS is up and running, the meaning of simulation "completeness" expands to include arbitrary new IO devices as OS developers seek to port as many device drivers to IA-64 as possible.

Often overlooked, but equally important features of a simulation infrastructure are its *flexibility* and *maintainability*. SoftSDV has been under development for nearly as long as IA-64 and has had to track and rapidly adapt to improvements in the ISA definition; flexibility and maintainability of the simulation infrastructure were absolutely essential. But here too, the requirements change over time. Early in the development of a new ISA, design changes are frequent, and a simulation environment must adapt quickly. Later, as the hardware definition becomes more concrete, flexibility gradually becomes less important, and it can be traded for increased simulation speed or accuracy.

These diverse and shifting requirements underscore a fundamental truth of simulation: a single technique or tool cannot meet the needs of all possible types of software development at all times. SoftSDV acknowledges this fact through an extensible design that accommodates the best features of multiple innovative simulation technologies in a single, common infrastructure. The resulting system enables IA-64 software developers to select the combination of simulation speed, accuracy, and completeness that is most appropriate to their particular needs at a given time, while at the same time preserving the flexibility and maintainability of the overall simulation infrastructure.

In the next section, we review related work and then briefly overview the hardware components typically found in an IA-64 SDV. We then present the software architecture of SoftSDV and describe each of its component processor modules and IO-device modules in detail. We conclude with a discussion of results and a summary of our experiences and lessons learnt.

RELATED WORK

A survey of recent simulation research reveals a continuous tension between the conflicting goals of simulation speed, accuracy, completeness, and flexibility.

A good example of a technique that achieves very high simulation speeds at the expense of accuracy is *dynamic binary translation*. This method for fast ISA emulation works by translating instructions from a *target ISA* into equivalent sequences of *host ISA* instructions. By caching and reusing the translated code, an emulator can dramatically reduce the fetch-and-decode overhead of traditional instruction interpreters. Cmelik and Keppel describe an early implementation of this method in their Shade* simulator, and they provide an excellent survey of several related techniques [3]. The Embra simulator has shown that dynamic binary translation can be extended to support the simulated execution of a full OS by emulating privileged instructions, virtual-to-physical address translation, and exception and interrupt processing [4].

Fast ISA emulators such as Shade and Embra are unable to predict processor performance at a detailed clock-cycle level. When simulation accuracy and flexibility are of primary importance, a better approach is often that of a simulation tool set, such as SimpleScalar, which enables rapid construction and detailed simulation of a range of complex microarchitectures [5]. SimpleScalar has indeed exhibited a high degree of flexibility as evidenced by its extensive use in the research community for a variety of microarchitecture studies. But this flexibility and accuracy comes at a cost: detailed SimpleScalar simulations can be more than 1,000 times slower than native hardware, whereas fast ISA emulators like Shade and Embra exhibit slowdowns ranging from 10 to 40, depending on the type of workload that they simulate. These numbers illustrate the compromises that simulators must make between speed, accuracy and flexibility, with Shade and Embra representing one end of the spectrum and SimpleScalar situated near the other end.

Many other intermediate points along the speed-accuracy-flexibility spectrum are possible. FastCache, for example, extends dynamic binary translation techniques to simulate the performance of simple data-cache structures with slowdowns in the range of 2-7, but it is limited with respect to other forms of microarchitecture simulation [6]. At the expense of some flexibility, FastSim uses memoization (result caching) to model a full out-of-order processor microarchitecture with slowdowns ranging from 190 to 360, a speedup of roughly 8-15 relative to comparable SimpleScalar simulations [7]. Another way to trade speed for accuracy is *sampling*: running only certain portions of a target workload through a detailed performance simulator. If the samples are chosen carefully and are of sufficient length, they can predict the performance of the entire workload with far less simulation time, but at the expense of some increased error [8].

SimOS [9] and SimICS* [10] are both good examples of simulation systems that have attained a high level of completeness. Both extend their simulations beyond the processor to include IO-device models, and they are able to support the simulated execution of complete operating systems as a result.

SoftSDV uses many techniques similar to those described above. However, because of the unique capabilities of IA-64 and the diversity of software development that SoftSDV must support, we found we had to reexamine many of the techniques in a new context. To explain some of the issues, we briefly overview the components in a typical IA-64 SDV in the next section.

COMPONENTS OF AN IA-64 SDV

At the core of an actual hardware IA-64 SDV platform is one or more Itanium processors that implement the IA-64 ISA. For the purposes of this paper, the most relevant aspects of IA-64 are the following:[1]

- *Memory Management*: An IA-64 processor translates 64-bit virtual memory accesses through split instruction and data TLBs, which are refilled by software miss handlers with a hardware assist. The TLB enforces page-level read, write, and execute permissions for up to four privilege levels.
- *Predication*: Most IA-64 instructions can be executed conditionally, depending on the value of an optional predicate register associated with the instruction.
- *Data Speculation*: IA-64 supports an advanced-load operation, which enables a compiler to move loads ahead of other memory operations even when the compiler is unable to disambiguate neighboring memory references. The address from an advanced load is inserted into an Advanced Load Address Table (ALAT), which must be checked for data dependencies on subsequent memory operations. A load-check operation examines the ALAT and invokes fix-up code whenever necessary.

* Other brands and names are the property of their respective owners.

[1] More details regarding IA-64 are available from the Intel Developer's Web Site [2].

Module		Characteristics	Typical Usage
Processor Modules	Fast ISA Emulator	Highest speed, no cycle-accurate perf. data	Rapid IA-64 app. and OS development
	Resource Analyzer	Medium speed/accuracy, limited flexibility	Large application, OS and compiler tuning
	Microarch Simulator	Highest accuracy and flexibility	Advanced compiler and microarch co-design
IO-device Modules	Basic IO Devices	Provide platform-modeling completeness	Early OS porting with standard boot devices
	IO-Proxy Module	Links to arbitrary PCI and USB devices	Advanced device-driver development

- *Control Speculation*: An IA-64 compiler can move loads before branches that guard their execution. If a speculative load causes an exception, the exception is deferred, and a Not-a-Thing (NaT) bit associated with the destination register is set instead. NaT bits are propagated as a side effect of any uses of the speculative load value, and they are checked after the controlling branch finally executes.
- *Large Register Set:* IA-64 includes 128 general registers and 128 floating-point registers, along with numerous other special-purpose registers. IA-64 registers assist loop-pipelining scheduling through their support of *rotating registers*.

Predication, speculation, and abundant registers are all part of a key principle behind the design of IA-64: to enable a compiler to explicitly expose instruction-level parallelism (ILP) to the hardware microarchitecture [1, 2]. By helping to relieve the hardware of finding ILP, IA-64 makes possible higher-frequency processor implementations. The compiler expresses instruction parallelism to the hardware by collecting instructions into *instruction groups*, which are independent instructions that can be executed at the same time. As we will see, the above IA-64 features create new challenges and opportunities for processor-simulation techniques.

In addition to an Itanium processor, an IA-64 SDV typically contains a chipset (e.g., 460GX), with support for PCI and USB IO-device busses, and a basic collection of IO devices suitable for running an operating system. This includes an IO streamlined APIC interrupt controller and an assortment of keyboard, mouse, storage, network, and graphics devices. Since a hardware SDV typically contains a number of expansion PCI slots, and a USB host controller, the range of devices it supports is limited only by the availability of devices designed to these bus standards. Our goal with SoftSDV was to provide the same level of IO-device support.

SOFTWARE ARCHITECTURE

The software architecture of SoftSDV is based on a simulation kernel that is extended through the addition of *modules* (see Figure 1 and Table 1). A SoftSDV module either models a hardware platform component (such as a processor or IO device), or it implements a simulation service (such as a trace-analysis tool). We call these two types of modules *component modules* and *service modules*, respectively.

SoftSDV modules share data and communicate with one another through a set of abstractions provided by the kernel: *spaces*, *events*, *data items* and *time*.

- *SoftSDV spaces* are used to model how platform components communicate through physical-address space, IO-port space, PCI-configuration space, and other linearly addressable entities such as disk images and graphics framebuffers. Modules can create spaces and then register *access-handler* functions with the kernel for a certain address region in a space. When another module reads or writes an address in that range, the SoftSDV kernel routes the access to the registered handler. SoftSDV spaces enable an IO-device module, for example, to specify how its control registers behave by mapping them to specific addresses in IO-port space.
- *SoftSDV events* enable a module to request notification of some occurrence inside another module. Events can be used to model IO-device interrupts, or to collect event traces, such as a sequence of OS context switches, which might be analyzed by a trace-processing module.
- *SoftSDV data items* provide a mechanism for modules to name and share their state with other modules. Named data items enable a processor module, for example, to make its simulated register values available to a debugging tool. Some data items are managed by the SoftSDV kernel itself and they are sometimes used to specify configuration data that modules can query

to determine how they should function during a simulation.

- *SoftSDV time* enables modules to synchronize their interactions to a common clock and to schedule the execution of future events. A disk-controller model, for example, can register a callback function with the kernel to be called after some pre-computed delay that represents the time for a simulated seek latency.

SoftSDV has a standard set of tracing and debugging modules that are built upon the abstractions above. SoftSDV events enable the construction of tracing modules that specify, by event name, items to be traced. Traceable events include instructions, addresses, IO-device accesses, OS events, etc. SoftSDV traces are suitable for trace-driven cache and processor performance simulators, and they can also provide input to the Vtune™ performance analyzer [11]. The SoftSDV debugger uses SoftSDV events and data items for setting instruction and data breakpoints, reading memory values, examining register values for a specific simulated processor, etc.

SoftSDV also includes a standard set of component modules, including three IA-64 processor models, each offering different levels of simulation speed, accuracy, and flexibility. For completeness in platform modeling, SoftSDV also includes an assortment of IO-device models and a special IO-proxying module that links the simulation to actual PCI and USB devices installed in the simulation host machine. Table 1 summarizes these standard component modules and serves as an outline for the remainder of this paper, in which we detail the techniques used by each of these modules.

FAST IA-64 EMULATION

SoftSDV's fastest processor module is an emulator that implements the full IA-64 instruction set, including all privileged operations, address translation, and interrupt-handling functions required to support operating systems. The principal goal of the emulator is speed; it is unable to report cycle-accurate performance figures.

SoftSDV's emulator uses dynamic binary translation to convert IA-64 instructions into sequences of host (IA-32) instructions called *capsules*, which consist of a *prolog* and a *body* (see Figure 2). The capsule body emulates the IA-64 instruction in terms of IA-32 instructions executed on the host machine. In the example shown in Figure 2, a 64-bit *Add* instruction is emulated by a sequence of IA-32 instructions that loads the required source registers from a simulated IA-64 register file (held in host memory), performs the simulated operation (a 64-bit *Add* using the 32-bit *Add* operations of IA-32), and then stores the result back to the simulated register file.

The capsule prolog serves two purposes. First, it implements behavior that is common to all IA-64 instructions, such as predication and control speculation. A portion of the prolog code checks each instruction's qualifying predicate, and it jumps over the capsule to the next instruction if the predicate is false. Similarly, another portion of the prolog examines the NaT bits of all source operands, and it propagates any set values to destination registers, or it generates a fault as dictated by the semantics of the instruction. A second use for the prolog is to implement a variety of simulation services, such as execution tracing, instruction counting, and execution breakpoints, which we discuss later.

The emulator translates instructions only as needed in units of basic blocks, which it caches in a translation database. Since capsules require on average 25 IA-32 instructions for each IA-64 instruction that they emulate, a large simulated workload can quickly consume host memory, causing the host OS to begin paging to disk. The emulator limits the maximum size of its translation cache to prevent host-OS paging, choosing instead to retranslate IA-64 instructions, a far faster operation.

SoftSDV capsules eliminate the need for a full fetch-decode-execute loop. Instruction emulation begins with an indirect branch that goes directly to a capsule corresponding to the current simulated instruction pointer (IP). Since capsules are linked directly from one to another, execution proceeds from capsule to capsule until an untranslated instruction is encountered, and control

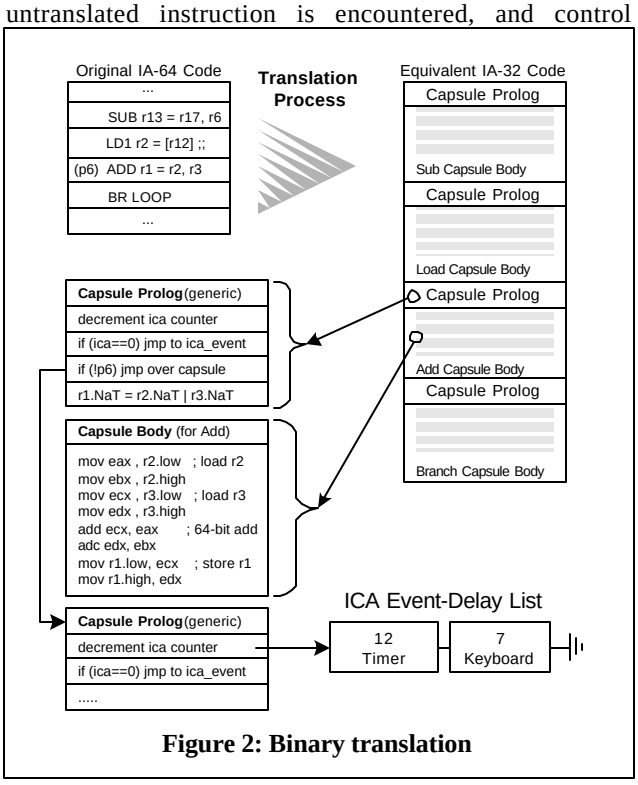


Figure 2: Binary translation

transfers back to the translator.

SoftSDV executes nearly all IA-64 instructions as capsules, but certain complex instructions are emulated with calls to C* functions. Throughout the development of SoftSDV, we considered moving to capsule-based implementations for various complex operations, but we frequently found that the resultant reduction in flexibility and maintainability of the emulator did not justify the optimization. We also considered more aggressive intercapsule optimizations, such as those used by Shade* and Embra (e.g., avoiding loads/stores to the simulated registers in host memory when neighboring instructions use the same register). Unfortunately, with the very large IA-64 register set, and the limited number of IA-32 host registers, we found such optimizations ineffective. Intercapsule optimizations suffer the further side effect of limiting the granularity at which debug breakpoints can conveniently be set.

Address Translation

SoftSDV models simulated physical memory by allocating pages of host virtual memory to hold the contents of simulated data. For any simulated memory reference, the emulator must first translate a simulated virtual address to a simulated physical address (V2P), and then further translate this physical address to its allocated host address (P2H). The composition of these two translation functions provides a full translation from virtual to host memory (V2H = P2H(V2P(Virtual))).

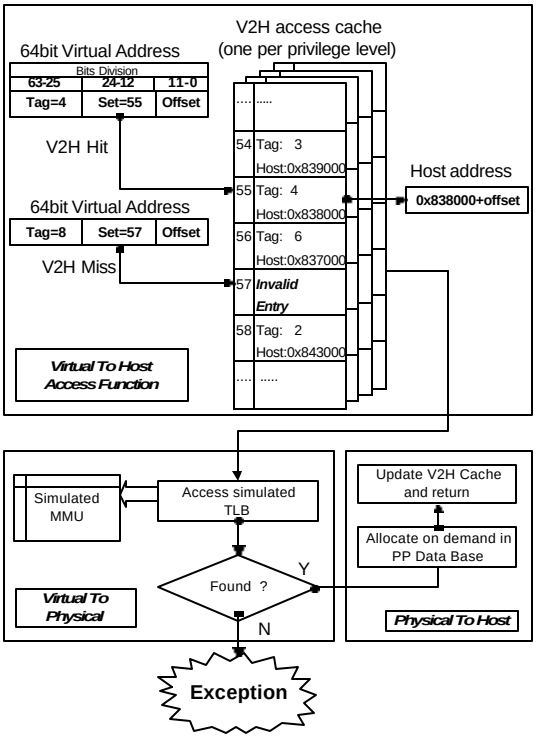


Figure 3: Emulating IA-64 address translation

The emulator uses a data structure called a V2H cache to accelerate address translation (see Figure 3). When an IA-64 load/store operation or branch/call instruction references a virtual address, the emulator first searches a V2H cache using highly efficient assembly code called from the capsule. In the common case, the translation is found, and the memory reference is quickly satisfied. In the case of a V2H miss, a full translation sequence (V2P and P2H) is activated by simulating an access to the TLB and an OS-managed page-table structure. If the translation succeeds, the V2H table is updated, and execution returns to the capsule. If the search fails to find a valid translation, then a simulated page fault or protection violation has occurred, and SoftSDV raises an exception for the simulated OS to handle.

The emulator implements V2H tables as variable-sized caches of valid address mappings. This is in contrast to Embra's *MMU relocation array*, which is a fixed-size 4-MB table that maps every possible page in a 32-bit virtual address space [4]. While the Embra approach guarantees a 100% hit rate for any valid mapping, we found this approach unusable for modeling a 64-bit virtual address space. Since V2H refills are a principal source of slowdowns, the emulator uses a number of optimization techniques to accelerate refills, and it supports configurable V2H table sizes to tune hit rates to the requirements of a given workload.

Page Protection

The emulator implements page protections by building a V2H cache for each type of memory access: *V2H-read*, *V2H-write*, and *V2H-execute*. A read-only page, for example, present in the V2H-read cache does not contain an entry in the V2H-write cache. By dividing the V2H caches in this way, the emulator avoids performing protection checks explicitly in each capsule; it merely generates code to access the V2H cache appropriate to the desired operation (e.g., load, store, branch), and a V2H miss enforces the protection. The emulator builds a set of read-write-execute V2H caches for each privilege level (i.e., 12 V2H caches in all) and simply switches between them when privilege levels change. The use of multiple V2H caches is a memory-speed tradeoff. At the expense of extra memory devoted to the V2H caches, the emulator simplifies the protection-checking code in each capsule, thus accelerating overall instruction-emulation times.

Speculative Data Accesses

Data speculation with advanced loads requires special treatment by the emulator. The capsule for an advanced

load includes code that inserts the load address into a simulated ALAT. The capsules for any subsequent store instructions include highly optimized code (11 host instructions) that searches the ALAT for an address match. Any entries with matching addresses are removed from the ALAT so that a subsequent load-check operation will invoke the appropriate fix-up code.

Instruction Fetching

Rather than point directly to translated code capsules, V2H-execute entries reference a page of pointers to code capsules. This level of indirection solves two problems. First, since the emulator lazily translates instructions a basic block at a time, many portions of a code page may be untranslated; these cases are handled by pointing to a *do-translate* function, rather than a capsule entry point. Second, since capsules are of varying size, their entry points are not simple to calculate given only a virtual address for an instruction; the pointers-to-capsules page greatly simplifies simulation of branch instructions and arbitrary jumps to code, such as from a debugger.

The emulator avoids V2H-execute lookups by directly chaining a branch-instruction capsule to its target-instruction capsule, provided they reside on the same page. For cross-page branches, the chain passes through the V2H-execute cache to ensure that the code page is accessible.

The emulator supports self-modifying code by preventing a page from residing in both the V2H-write and V2H-execute caches at the same time. Any attempt to write to a code page causes a V2H-write miss, which causes the page to be removed from the V2H-execute cache. Any subsequent attempt to execute instructions from the page causes a V2H-execute miss, which results in a lazy retranslation of the modified code page.

Interrupt Processing

The emulator can potentially run for long periods of time without ever leaving its capsules. This presents a problem because other SoftSDV modules (such as simulated IO devices) might need to interrupt processor execution. The emulator provides a solution to this through a mechanism called an *instruction-count action* (ICA). ICA is based on a sorted list of event-delay pairs that define a set of callback functions to be executed after some number of instructions have executed. The delay of the event at the head of the ICA list is decremented in the prolog of each capsule (see Figure 2). When the delay reaches 0, the event is triggered by calling its associated callback function. After completion of the callback function, the event-delay pair is deleted from the ICA list.

The ICA mechanism is integrated with the SoftSDV time abstraction, providing a coarse-grained notion of time based on number of instructions executed. It is also ideal for implementing certain IA-64 debugging facilities, such as single-step instruction execution, which would otherwise require retranslation of instruction capsules.

Multiprocessor Simulation

The emulator can simulate a platform with up to 32 processors in a symmetric shared-memory configuration. Only one processor is simulated at a time, with switches between simulated processors scheduled in a simple round-robin order with a configurable time slice. All processors share memory, translations, and all simulated platform devices. Capsules indirectly access simulated processor state and processor-specific emulator state (e.g., V2H caches, ICA lists, etc.) through a pointer, which enables rapid switching between simulated processors via a simple pointer change.

DYNAMIC RESOURCE ANALYSIS

The emulator described in the previous section is ideal for rapid porting of operating systems and applications to IA-64. Once the porting is complete, a software developer may be interested in tuning code for performance, which requires trading some simulation speed for increased accuracy. SoftSDV's second processor module offers just this: it aims to achieve a level of performance-prediction accuracy that is within 10% of a detailed Itanium microarchitecture simulator (which we describe in the next section), while retaining the highest possible simulation speed.

SoftSDV applies three principles to achieve this goal. First, it tightly integrates performance analysis with its fast IA-64 emulator, enabling it to overcome the bottlenecks of traditional trace-driven performance simulation. Second, it assumes an in-order processor pipeline and selectively models only those processor resources that have the greatest effect on overall performance (e.g., first-level caches, branch prediction, functional units, etc.). Third, it caches the results of its resource analysis to speed future performance simulation. We call this collection of methods *dynamic resource analysis*, and we refer to this SoftSDV module as the *resource analyzer*.

Processor and Memory Resources

Processor performance analysis ultimately boils down to accounting for the resource requirements of executing code. Take, for example, the code shown in Figure 4, which shows a *subtract* and a *load* in one instruction group and an *add* and a *branch* in a second instruction

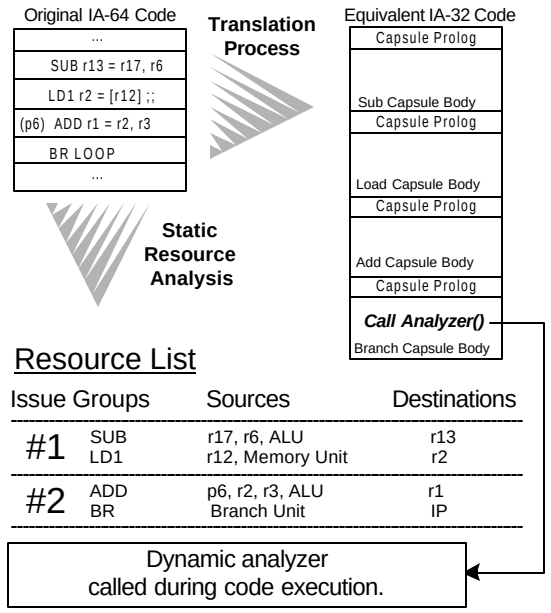


Figure 4: Resource analysis for two issue groups

group (the notation ";;" separates instruction groups). The *load* operation:

LD1 r2 = [r12] ;;

requires the resources of a source-address register (*r12*), a destination register (*r2*), and a data-cache line and read port. Similarly, the *add* instruction:

(p6) ADD r1 = r2, r3

requires the resources of an ALU functional unit, a predicate register (*p6*), two source registers (*r2* and *r3*), and a destination register (*r5*).

A processor examines instruction groups to find collections of instructions (called an *issue group*) that can be dispatched in parallel subject to the resource constraints of its execution pipeline. When all the required resources of all instructions in an issue group are available, the instructions can execute immediately; otherwise, they stall. In the example in Figure 4, the *load* feeds the *add* through register *r2*. If the *load* stalls because of a data cache miss, then the *add* will stall as well because it requires the *r2* resource. The resource analyzer identifies such resource dependencies by dividing its operation between two phases, each of which is tightly integrated with the IA-64 emulator. A *static resource-analysis* phase is invoked by the emulator's translator, and a *dynamic-analysis phase* is invoked by translated code capsules.

Static Resource Analysis

Like the IA-64 emulator, the resource analyzer examines code lazily, as it is first encountered. Whenever the IA-64 emulator completes translation of a new basic block of code, it calls the resource analyzer, which examines the basic block to find issue groups and then statically determines the microarchitectural resources required by each issue group. The analyzer caches this *resource list* to avoid the cost of repeating the analysis each time the issue group executes (see Figure 4). The resource list includes both *sources*, which are required for the issue group to execute, and *destinations*, which are resources that will be available after the issue group executes.

Dynamic Analysis Phase

The dynamic phase of the analyzer is invoked by emulator capsules after execution of each basic block. The analyzer examines instruction resource requirements by modeling a simplified Itanium microarchitecture consisting of a front end, a back end, and caches.

The analyzer's back end keeps track of the availability of core pipeline resources (e.g., register files, functional units, etc.) that could stall the execution of an issue group. Each resource that is required for the execution of the current issue group is checked for its availability in the current cycle. If it is not available, then the issue group stalls execution, and the cycle counter is advanced to the time when all required resources will be available and the issue group can enter the execution stage. Next, the resource state is updated to reflect the results of the current issue group. For each destination of the instructions in the issue group (excluding loads), the clock cycle in which these resources will be available for use by subsequent instructions is calculated by adding each instruction's latency to the cycle in which the issue group was dispatched.

The analyzer's front end models branch prediction and instruction-fetching mechanisms in accordance with the Itanium microarchitecture. Since all instructions traverse the front end several cycles before they reach the back end, the resource analyzer maintains two separate cycle counts, one for the back-end execution (as described previously) and one for the front end. The two cycle counters are synchronized through a model of a decoupling buffer, which specifies the cycle in which an issue group can be dispatched. If the issue group is not ready, the decoupling buffer stalls the back end. Conversely, if the decoupling buffer is full, it causes front-end stalls.

The analyzer models instruction and data caches to account for performance lost due to cache misses. These cache models are referenced by both the front end (for instruction fetches) and the back end (for data references).

The IA-64 emulator maintains its own copy of the first-level data cache and achieves speed through integration with its V2H translation tables in an approach similar to that of Lebeck's FastCache [6]. This copy contains a subset of the data-cache lines simulated by the resource analyzer. When the emulator detects a hit, it continues execution without calling the analyzer. In the rare case when the emulator suspects a cache miss, it calls the analyzer for verification. When a miss occurs in the analyzer cache, the analyzer updates its own cache contents and informs the emulator about the cache line it replaced.

The end results of the analysis described above are a set of performance metrics (e.g., branch-prediction accuracy, cache misses, stall cycles, etc.), which are mapped onto SoftSDV data items so that they can be read by a performance visualization tool, such as the Vtune™ performance analyzer.

DETAILED PROCESSOR MICROARCHITECTURE SIMULATION

The dynamic resource analyzer described previously is ideal for rapid tuning of compilers, operating systems, and large applications, where approximate performance of a fixed microarchitecture is acceptable. For other applications, such as compiler tuning for a new microarchitecture under development, a further shift along the speed-accuracy-flexibility spectrum is needed. To deal with such situations, SoftSDV works together with a simulation toolset for exploration of IA-64 microarchitectural designs. SoftSDV's third processor module, a detailed model of the Itanium microarchitecture, is written using this toolset, as are other future IA-64 microarchitectures currently under development by Intel.

IA-64 Microarchitecture Simulation Toolset

The toolset consists of two main components: an event-driven simulation engine, and a set of facilities for functional evaluation of IA-64 instruction semantics.

The event-driven simulation engine provides a flexible set of primitives for modeling microarchitectural resources, arbitration, and accounting of processor cycles. A processor pipeline is modeled as a set of communicating finite state machines through which *tokens* representing instructions or data are evaluated on a cycle-by-cycle basis. Whenever a token cannot acquire a certain resource (e.g., a latch or port), a stall condition is encountered and its cycle penalty is accounted for. The total execution time of a workload is derived from the accumulation of cycles spent by all tokens that traverse the pipeline.

The functional evaluation of IA-64 instruction semantics is provided by a set of four interfaces called by different stages of the event-driven microarchitecture model:

- *Fetch* provides a decoded instruction, given an IP.
- *Execute* computes the results of an instruction in the context of some microarchitectural state.
- *Retire* commits the microarchitectural state to the permanent architectural state.
- *Rewind* rolls back the unretired microarchitectural state to previously defined values.

By dividing the interfaces in this way, a processor model is able to express complex microarchitectural interactions involving speculative instruction execution.

Speed-Accuracy Modes and Sampling

Like the resource analyzer, the Itanium microarchitecture model is tightly integrated with the IA-64 emulator through an interface that enables the sharing of the architectural state (memory and processor registers). This interface makes it possible to dynamically change simulation speed and accuracy as a workload runs. It is possible, for example, to rapidly advance through the simulated boot of an OS with the fast IA-64 emulator. Then, prior to the execution of some workload of interest, the state of the processor's caches is initialized using memory traces produced by the emulator. When detailed simulation is desired, the microarchitectural simulator reads the current machine state from the emulator, and begins its simulation. After running in detailed mode for some time, execution can return to the fast mode to advance deeper into a workload's execution.

SoftSDV supports both *uniform sampling* at some regular, predefined period, and *event-based sampling*. For event-based sampling, special *markers* are compiled around regions of interest in a workload. SoftSDV dynamically recognizes such markers as a workload executes, and generates corresponding SoftSDV events, which are monitored by the processor modules to determine when they should switch between speed-accuracy modes.

IO-DEVICE MODULES

SoftSDV provides a standard collection of IO-device models suitable for supporting the simulated execution of a complete IA-64 operating system, including its basic device drivers. These include selected 460GX chipset and boardset functions (such as interrupt controllers, periodic timers, serial port, mouse, keyboard, etc.) as well as models for assorted storage and graphics controllers (such as IDE/ATA, ATAPI CD-ROM, VGA, etc.).

A SoftSDV device model typically consists of two halves: a front half that implements the device interface as expected by a simulated device driver, and a back half that simulates the device function. The implementation of many device models is a straightforward conversion between device commands and host-OS services. Models for the keyboard and mouse, for example, field host-OS window events (keypress up/down, pointer movement, etc.) and convert them into equivalent device events (a keyboard interrupt and scancode). Similarly, the serial-port model connects to the actual serial port of the host machine through host-OS calls, and it ferries data back and forth between the simulated and actual serial ports.

IO-Device Services

Since many devices offer a similar function (e.g., storage), but with differing interfaces (e.g., IDE/ATA, ATAPI, SCSI), we identified points in common among device models and structured them as reusable *IO-device services*, encapsulated in their own SoftSDV modules. A *disk-image service*, for example, provides functionality useful for any storage model. It holds the contents of a simulated disk in a file or a raw disk partition in the host machine, and has the ability to log all changes to the simulated disk's contents. An off-line utility can then either commit or discard those changes at the end of a simulation. The disk-image service maps itself onto a SoftSDV space, where its functionality can be accessed by any disk-interface model.

Similarly, a *display service* maps a generic flat framebuffer surface onto a SoftSDV space, and it reflects accesses to this surface in a host-OS window, or directly to a second dedicated display attached to the simulation host machine. This structuring relieves graphics-adapter models from the details of how a framebuffer is displayed so that they can focus instead on simulated processing of graphics commands. It also enhances SoftSDV maintainability, since it decouples graphics models from the host-OS windowing system; when porting to a new host OS, only the display service and not each graphics model must be rewritten.

Pseudo Devices

Some SoftSDV modules model a fictitious device, or only some aspect of an actual device. We have, for example, built a pseudo-device module that maps the resource requirements of multiple devices into PCI configuration space. Although not backed by actual PCI-device models, these headers present to a simulated OS the illusion of a platform with multiple PCI devices, and thus enable rapid testing of the OS's device configuration and plug-and-play algorithms. Such a test environment is, in fact, far more convenient than an actual hardware platform since

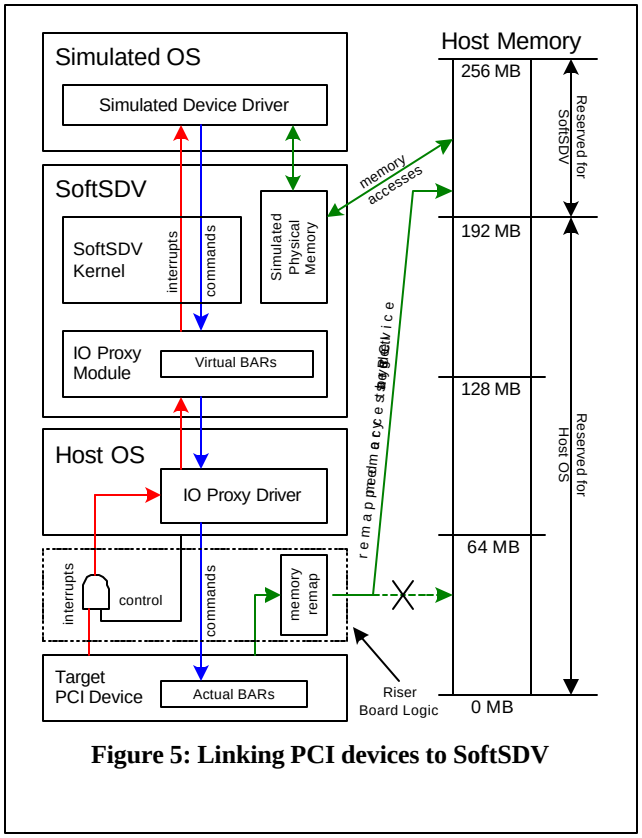


Figure 5: Linking PCI devices to SoftSDV

there is no need to physically populate actual PCI slots with various devices to create different test cases. We have experimented with other types of pseudo devices, such as an IO-monitoring service, which would enable the replaying of keyboard and mouse input to a graphical windowing system in a reproducible and OS-independent manner.

IO-PROXY MODULE

Due to the broad diversity and sheer number of IO devices in existence, we realized early in the development of SoftSDV that it would be impossible to write software models of all IO devices of interest. Indeed, some IO devices are so complex that modeling them in software would be considerably more work than the ultimate goal of porting their device drivers to IA-64.

An alternative solution is to link existing hardware devices directly to SoftSDV so that they can be accessed and programmed by simulated IA-64 device drivers under development. SoftSDV accomplishes this through a combination of hardware and software components, shown in Figure 5. The hardware components consist of an arbitrary target PCI device plugged into a custom-designed riser board that performs memory and interrupt-remapping functions. The software components include a SoftSDV module and a host OS device driver, which function as proxies for interactions between the actual

hardware PCI device and the simulated device driver. The proxy components support three forms of interaction between the target device and the simulated device driver running on SoftSDV:

- commands issued from the driver to the device via IO-port and memory-mapped IO accesses
- physical-memory accesses by the device
- interrupts from the device to the simulated driver

To support the largest possible set of devices, SoftSDV enables each of these types of interactions without requiring any specific knowledge of the target PCI device. The following sections detail how each of the three forms of interaction are supported.

Command Proxying

The key difficulty in proxying commands from the simulated device driver is determining the location of the control registers for arbitrary target devices. Fortunately, standard PCI configuration mechanisms provide a means for accomplishing this.

All PCI devices support a *configuration header* visible to software through well established locations in IO-port space. The PCI configuration header includes a set of *Base Address Registers* (BARs), which the host proxy device driver reads to determine the size and location of the device's control registers in IO-port space or physical-address space. The proxy driver passes this information up to the proxy SoftSDV module, which registers access handlers for those regions of address space with the SoftSDV kernel. The registered handler ensures that the proxy module is called whenever the simulated device driver sends commands to the device. The proxy module blindly passes such commands down to the proxy driver, which in turn issues them to the actual hardware device.

The actual solution is somewhat more complex because by the time the SoftSDV simulation starts running, the host OS will have already configured the target device's BARs to avoid conflicts with other host IO devices.[2] The problem is that SoftSDV models an entirely different (simulated) view of the platform, so the simulated OS may attempt to configure the target device's control registers to a different set of locations that might conflict with other host IO devices. The proxy code solves this problem by presenting a set of *virtual BAR* values to the simulated OS, and it remaps these values to the actual BAR values used by the target device.

Remapping Physical Memory Accesses

After the simulated driver sends the device commands, the target device will typically try to access physical memory to retrieve or deposit data associated with the command. This is a problem because the device is directed to perform the operations in simulated physical memory, but the device will operate on actual host memory belonging to the host OS, causing it to crash.

The solution is to partition the host physical memory between the host OS and the simulated OS. During boot, the host OS is configured not to use a certain amount of host physical memory (in the example in Figure 5, the reserved region is 64MB, starting at 192MB in host memory). The reserved region is then mapped to the simulated memory in the SoftSDV user process.

This alone is not sufficient to solve the problem since the device will still be programmed to use physical memory in the range of 0-64MB when it should instead be accessing memory in the range of 192-256MB. The proxy code could, in principle, interpret and modify the commands it intercepts from the simulated driver and remap addresses as appropriate before passing the commands to the actual device. Unfortunately, such a solution requires specific knowledge of the device and its commands. SoftSDV instead uses a small amount of hardware remapping logic located on the riser board between the target device and its host PCI slot. The remapping logic relocates all memory accesses made by the target device to the upper partition of memory, based on flexible configuration settings that specify the size of memory and the location of the partition.

Interrupt Proxying

When the target device generates an interrupt, it is fielded by the proxy code and sent to the simulated device driver. Two problems make it difficult to perform these steps in a device-independent manner. First, since interrupt lines are commonly shared between PCI devices, the proxy code must determine whether the interrupt is coming from the target device or from some other host device. Second, the interrupt line must be temporarily deactivated; otherwise, SoftSDV (which runs as an ordinary user-level process of the host OS) will be continually interrupted, and the simulated device driver will never have a chance to run.

Both of these operations are performed with the help of some additional logic on the riser board that enables the host proxy driver to sense and mask the interrupt from the device, before it is driven onto the shared PCI interrupt line. When an interrupt occurs, the proxy driver uses this

[2] BAR registers are programmable to enable plug-and-play software to assign conflict-free locations for device control registers.

logic to determine if the interrupt is in fact originating from the target device. If so, it temporarily masks the interrupt and passes control to the simulated device driver running in SoftSDV. The simulated driver, which understands the specifics of the device, then properly dismisses the interrupt at its source in the target device.

RESULTS AND EXPERIENCES

Table 2 summarizes the speed of SoftSDV's three processor modules when running selected SPEC95 benchmarks and when booting Microsoft 64-bit Windows* 2000. It also reports accuracy of the performance analyzer relative to the microarchitecture simulator.[3] All experiments were performed on a 500-MHz Intel® Pentium® III processor-based system with 512 MB of memory.

The emulator offers simulation speeds ranging from 10 to 25 million instructions per second (MIPS) for the SPEC benchmarks, and about 3 MIPS for the simulated Windows* boot. The lower execution rate for the OS boot is due to frequent address-space generation and process switching, which increases V2H translation overheads. Even so, at an average rate of 3 MIPS, SoftSDV is able to perform a simulated Windows boot in less than seven minutes, and is able to boot most other IA-64 operating systems in under ten minutes. The performance accuracy of the emulator, however, is limited to reporting the total number of instructions executed by a workload.

With error rates ranging from about 1% to 7% and averaging 3%, the dynamic resource analyzer exceeded our goal of predicting cycle-level processor performance to within 10% of the detailed Itanium microarchitecture simulator, and it does so at simulation rates ranging from about 160 to 250 thousand instructions per second (KIPS). These speeds are sufficient for tuning compilers and large applications, which often require millions if not billions of instructions to be executed. These results suggest that by explicitly exposing ILP, IA-64 compilers not only enable simpler, higher-frequency processor implementations, but they also make possible very fast processor performance analysis. The methods used by the analyzer do, however, have their limitations. The analyzer depends on the existence of a reference microarchitectural simulator against which it can be calibrated. Also, its flexibility is somewhat limited, since it assumes an in-order microarchitecture and is unable to model hardware-controlled speculative execution.

Workload	Emulator Speed (MIPS)	Analyzer		Microarch Simulator Speed (KIPS)
		Speed (KIPS)	Accuracy (% error)	
go	15.4	237	1.18%	15.0
m88ksim	14.8	252	1.04%	34.8
gcc	10.5	224	4.96%	17.2
compress	25.3	180	7.19%	18.4
li	13.5	227	4.06%	27.2
ijpeg	28.3	162	1.97%	27.8
perl	10.5	212	1.21%	18.6
vortex	11.8	158	2.79%	22.7
NT boot	3.00	211	—	—

When flexibility and highest accuracy are required, there is no substitute for detailed microarchitecture simulation. The IA-64 microarchitecture toolset has proven itself flexible enough to rapidly explore design options, both for Itanium processors and for future IA-64 microarchitectures currently under development at Intel. The tradeoff for this flexibility and accuracy is a much lower speed of simulation, in the range of 1 to 2 KIPS.

We found building state-sharing mechanisms between the three processor models to be a very powerful capability. Had each processor simulator only been able to work independently, methods such as sampling performance over extended regions of large workloads would never have been possible. With sampling, we are able to freely select the level of speed and accuracy required for a given simulation. For the SPEC benchmarks, our experience has been that uniform sampling ratios of between 15:1 and 80:1 yield simulation results that are statistically very close to full simulation. Table 2 reports effective KIPS rates for detailed microarchitecture simulation when a sampling ratio of 40:1 is used. The speedups relative to full simulation are not a full factor of 40 because simulation speeds are somewhat lower during the beginning of each detailed sample, and because the simulation still includes the overhead of running the emulator/analyzer during fast mode. Nevertheless, sampling effectively increases the speed of simulation by more than an order of magnitude.

Our original plans were to develop software models for all important IO devices, but we quickly realized that this approach was intractable, which led us to develop the IO-proxy module. The IO-proxying approach became

* Other brands and names are the property of their respective owners.

[3] Table 2 does not report the accuracy of the other two processor modules because the emulator does not provide performance results and because we use the microarchitecture simulator as the baseline for performance (i.e., for the purposes of this paper, we consider its error to be 0%).

particularly important for more complex device types, such as SCSI controllers, ethernet interfaces, graphics adapters, and USB host controllers, all of which successfully work with the technique. Not only did this approach save effort in writing IO-device models in software, but it also better supported a broad range of OS's. Had we selected particular SCSI or ethernet controllers to model, we would be limited to supporting OS's that had drivers for those particular devices. With the IO-proxy module, OS developers can select virtually any PCI or USB device for which their OS has a corresponding driver.

We achieved our goals of supporting IA-64 software development all along the software stack. SoftSDV successfully runs:

- several commercial operating systems, including Microsoft 64-bit Windows* 2000, Trillian Linux*, IBM Monterey-64*, Sun Solaris*, Compaq Tru64*, Novell Modesto*, and HP-UX*.
- device drivers for at least a dozen complex PCI devices ranging from graphics and ethernet adapters, to SCSI and USB host controllers.
- numerous large applications
- three well-tuned IA-64 optimizing compilers

These layers of code were all working together, all exercised before silicon, and all ready for bring-up.

The real test for SoftSDV came after the availability of actual hardware SDVs. IA-64 versions of Windows 2000 and Trillian Linux* that were developed on SoftSDV booted within ten days of the availability of Itanium processor's first silicon. Drivers for complex devices, such as SCSI disks, ethernet adapters, and graphics controllers, quickly followed over the next week, and the first MP operating systems were running a week after that. Many of the problems encountered were due to setup and configuration issues with the hardware SDV platform. Once resolved, other operating systems have been brought up even more quickly. IBM Monterey-64, for example, was up and running in under three hours on a qualified Itanium processor SDV.

For further discussion of the issues involved in porting operating systems to IA-64, please see "Porting Operating System Kernels to the IA-64 Architecture for Presilicon Validation Purposes" [12], which appears in this same issue of the *Intel Technology Journal*.

* Other brands and names are the property of their respective owners.

CONCLUSION

Central to the design of SoftSDV is extensibility. By building a core simulation kernel with a general set of abstractions, we were able to unify several different simulation technologies, each with their own unique capabilities with respect to speed, accuracy, completeness, and flexibility.

SoftSDV has proven that substantial amounts of complex software can be developed, presilicon, even for an entirely new ISA such as IA-64. As a general, extensible simulation infrastructure, SoftSDV is now being used in several other efforts throughout the company, including IA-32 presilicon software development, performance simulation of future IA-based microarchitectures, and processor design validation.

ACKNOWLEDGEMENTS

SoftSDV is the product of dozens of contributors spanning three of Intel's development sites. The fast IA-64 emulator and resource analyzer were developed in the Microprocessor Products Lab (MPL-Haifa) under the leadership of Tsvika Israeli. Methods for linking IA-64 microarchitecture models to SoftSDV were developed by the Microprocessor Research Lab (MRL) in Santa Clara, while the platform and IO-device modules were developed in MRL-Oregon. The design and implementation of the SoftSDV kernel and its extensible API was a joint effort of all three groups. The IA-64 microarchitecture simulation toolset was developed in the IA-64 Product Division (IPD) by Ralph Kling, Rumi Zahir, and their teams.

Many thanks to all those who contributed: Jonathan Beimal, Rahul Bhatt, Baruch Chaikin, Stephen Chou, Maxim Goldin, Tziporet Koren, Mike Kozuch, Ted Kubaska, Etai Lev-Ran, Larry McGlinchy, Amit Miller, Kalpana Ramakrishnan, Edward Rubinstein, Nadav Nesher, Paul Pierce, Rinat Rappoport, Shai Satt, Jeyasingh Stalinselvaraj, Gur Stavi, Jiming Sun, and Gadi Ziv.

As key users of SoftSDV, special thanks go to Daniel Aw, Alan Kay, Bernard Lint, Sunil Saxena, Stephen Skedzielewski, Fred Yang, and Wilfred Yu for their always helpful feedback, and especially for making OS-porting efforts to IA-64 an enormous success.

And finally, much credit goes to our managers, Shuky Erlich, Jack Mills, Wen-Hann Wang and Richard Wirt, who somehow kept us all working together productively!

REFERENCES

[1] C. Dulong, "The IA-64 Architecture at Work," *IEEE Computer*, Vol. 31, No. 7, pp. 24-32, July 1998.

[2] Intel Corporation, *IA-64 Application Developer's Architecture Guide*, Order Number: 245188-001, http://developer.intel.com/design/IA64/, May 1999.

[3] R. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," in *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pp. 128-137, Nashville, Tennessee, May, 1994.

[4] E. Witchel and M. Rosenblum, "Embra: Fast and Flexible Machine Simulation," in *Proceedings of ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pp. 68-79, Philadelphia, May, 1996.

[5] D. Burger and T. Austin, "The SimpleScalar Tool Set," Version 2.0, *University of Wisconsin-Madison Computer Sciences Tech Report #1342*, June, 1997.

[6] A. Lebeck and D. Wood, "Active Memory: A New Abstraction for Memory System Simulation," *ACM Transactions on Modeling and Computer Systems (TOMACS)*, Vol. 7, pp. 42-77, 1997.

[7] E. Schnarr and J. Larus, "Fast Out-of-Order Processor Simulation Using Memoization," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pp. 283-294, San Jose, CA, October, 1998.

[8] T. Conte, M. Hirsch, K. Menezes, "Reducing State Loss for Effective Trace Sampling of Superscalar Processors," in *Proceedings of the 1996 International Conference on Computer Design (ICCD)*, Austin, TX, October 1996.

[9] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod, "Using the SimOS Machine Simulator to Study Complex Computer Systems," *ACM Transactions on Modeling and Computer Systems (TOMACS)*, Vol. 7, pp. 78-103, 1997.

[10] P. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner, "SimICS/sun4m: A Virtual Workstation" in *Proceedings of the 1998 Usenix Annual Technical Conference*, New Orleans, Louisiana, June, 1998.

[11] J. Wolf, "Programming Methods for the Pentium® III Processor's Streaming SIMD Extensions Using the VTune™ Performance Enhancement Environment," *Intel Technology Journal*, Q2 1999.

[12] K. Carver, C. Fleckenstein, J. LeVasseur, S. Zeisset, "Porting Operating System Kernels to the IA-64 Architecture for Presilicon Validation Purposes," *Intel Technology Journal*, Q4 1999.

Authors' Biographies

Richard Uhlig is a senior staff researcher in Intel's Microprocessor Research Labs in Oregon. His research interests include exploring new ways to analyze and support the interfaces between platform hardware architectures and system software. Richard earned his Ph.D. in computer science and engineering from the University of Michigan in 1995. His e-mail is richard.a.uhlig@intel.com.

Roman Fishtein is a senior software engineer in the MicroComputer Products Lab in Haifa. He has been with Intel since 1995 and has been involved in many aspects of the SoftSDV project, which he is currently leading. Roman obtained B.S. and M.S. degrees in electrical engineering from the Tashkent Electrotechnical Institute of Communication. His e-mail is roman.fishtein@intel.com.

Oren Gershon joined the Intel Development Center in Haifa, Israel in 1990 and is currently a staff software engineer with MicroComputer Products Lab in Santa Clara. Oren pioneered the design and development of several key SoftSDV technologies and has been involved with the project since its inception in 1993. Oren earned the Intel Achievement Award for this work. During a recent stint in MRL Santa Clara, Oren has been researching advanced performance analysis and visualization tools for future IA-64 implementations. Oren obtained a B.S. degree in computer science from the Technion, Israel Institute of Technology in 1992. His e-mail is oren.gershon@intel.com.

Israel Hirsh is a senior software engineer in the MicroComputer Products Lab in Haifa, Israel. He has been with Intel since 1992 and has made major contributions to SoftSDV during the five years that he led the project. Israel obtained a B.S. degree in computer science from the Technion, Israel Institute of Technology in 1978. His e-mail is israel.hirsh@intel.com.

Hong Wang is an engineer working in the IA-64 Architecture and Microarchitecture Research Group of Intel MRL in California. He has been active in building and enhancing state-of-the-art simulation infrastructures for IA-64. Hong is an avid crank on entropy, FFT, and variational principles of physics. His current interests are in discovering unorthodox ideas and applying them to next-generation IA-64 processor designs. Hong earned his Ph.D. degree in electrical engineering in 1996 from the University of Rhode Island. His e-mail is hong.wang@intel.com.

Assembly Language Programming Tools for the IA-64 Architecture

Ady Tal, Microprocessor Products Group, Intel Corporation
Vadim Bassin, Microprocessor Products Group, Intel Corporation
Shay Gal-On, Microprocessor Products Group, Intel Corporation
Elena Demikhovsky, Microprocessor Products Group, Intel Corporation

Abstract

The IA-64 architecture, an implementation of Explicitly Parallel Instruction Computing (EPIC), enables the compiler to exercise an unprecedented level of control over the processor. IA-64 architecture features maximize code parallelism, enhance control over microarchitecture, permit large and unique register sets, and more. Explicit control over parallelism adds a new challenge to assembly writing, since the rules that determine valid instruction combinations are far from trivial, introducing new concepts such as bundling and instruction groups.

This paper describes Intel's IA-64 Assembler and IA-64 assembly assistant tools, which can simplify IA-64 assembly language programming. The descriptions of the tools are accompanied by examples that use advanced IA-64 features.

INTRODUCTION

The IA-64 architecture overcomes the performance limitations of traditional architectures and provides maximum headroom for future development. Intel's innovative 64-bit architecture allows greater instruction-level parallelism through speculation, predication, large register files, a register stack, advanced branch architecture, and more. 64-bit memory addressability meets the increasingly large memory footprint requirements of data warehousing, e-Business, and other high-performance server and workstation applications. Significant effort in the architectural definition maximizes IA-64 scalability, performance, and architectural longevity.

In the 64-bit architecture, the processor relies on the programmers or the compiler to set parallelism boundaries. Programmers can decide which instructions are executed in each cycle, taking data dependencies and availability of microarchitecture resources into account. Assembly can be the preferred programming language under the following situations: when learning new computer architectures in depth; when programming at a low level, such as that required for BIOS, operating systems, and device drivers; and when writing performance-sensitive critical code sections that power math libraries, multimedia kernels, and database engines.

Intel developed the Assembler and the Assembly Assistant in order to aid assembly programmers in rapidly writing efficient IA-64 assembly code, using the assembly language syntax jointly defined by Intel and Hewlett-Packard*.

The Intel® IA-64 Assembler is more than an assembly source code-to-binary translator. It can take care of many assembly language details such as templates and bundling; it can also determine parallelism boundaries or check for those given by assembly programmers. The assembler can also allocate *virtual registers* and so enable assembly programmers to write code with symbolic names, which are replaced automatically with physical registers.

The Assembly Assistant is an integrated development tool. It provides a visual guide to some IA-64 architecture features permitting assembly programmers to comprehend the workings of the processor. The Assembly Assistant has three main goals: to introduce the architecture to new assembly programmers; to make it easier to write assembly code and use the Assembler; and to help assembly programmers get maximum performance from their code. This last task is achieved through *static analysis*, a *drag-and-drop* interface for *manual optimization*, and through *automatic optimization* of code segments.

IA-64 ARCHITECTURE FEATURES FOR ASSEMBLY PROGRAMMING

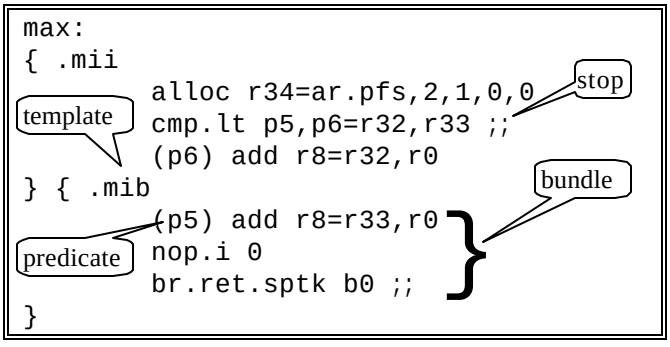
The IA-64 architecture incorporates many features that enable assembly programmers to optimize their code for efficient, high-sustained performance. To allow greater instruction-level parallelism, the architecture is based on

principles such as explicit parallelism, with many execution units, and large sets of general and floating-point registers. Predicate registers control instruction execution, and enable a reduction in the number of branches. Data and control speculation can be used to hide memory latency. Rotating registers allow low-overhead software pipelining, and branch prediction reduces the cost of branches.

The compiler takes advantage of these features to gain a significant performance speedup. Branch hints and cache hints enable compilers to communicate compile-time information to the processor. New compiler techniques are developed to use these features, and IA-64 compilers will continue gaining speedups utilizing these features in innovative ways.

This abundance of architecture features and resources makes assembly writing a challenging task for assembly programmers.

The IA-64 architecture requires that the instructions are packed in valid bundles. Bundles are constructs that hold instructions. Bundles come in several templates, restricting the valid combinations of instructions and defining the placement of boundaries (*stops*) for maximum parallelism. Each instruction can be placed into a specific slot in a bundle, according to the template and the instruction type. For example, the instruction *alloc r34=ar.pfs,2,1,0,0* appearing in the example below, can only be placed in an M slot of a bundle. The template defined in the example below for the first bundle is .mii meaning that the first slot can only be taken by an instruction that is valid for an M slot, and the following instructions must be valid for I slots. When no useful instruction can be placed in the bundle due to template restrictions, a special *nop* instruction, valid for the slot, must be used to fill the bundle (as observed in the case of the second slot in the second bundle; a nop.i was placed in an I slot).



Example 1: Code reflecting language syntax

Assembly programmers are expected to define groups of instructions that can execute simultaneously by inserting stops. If a stop is missing, then there is a chance that not all the instructions were meant to be executed in the same cycle. Such an instruction group may contain a dependent pair of instructions. For example, it may contain two instructions that write to the same register or a register write followed by a read of the same register.

The result of parallel execution of dependent instructions, even though not necessarily adjacent, is unpredictable and may vary in different IA-64 processor models. This situation is called a *dependency violation*. To avoid it, assembly programmers have to place the two instructions in different groups by inserting a stop.

In Example 1 we can see a *stop* after the *cmp* instruction. This *stop* will ensure that the *cmp* will not be executed in parallel with the following *add*, and it enables the *add* to use the predicate *p6* written by the *cmp* instruction without producing a *dependency violation*.

THE IA-64 ASSEMBLER

The IA-64 Assembler enables many capabilities beyond traditional assemblers. In addition to assembling, it implements full support of all architecture and assembly language features: bundles, templates, instruction groups, directives, symbols' aliases, and debug and unwind information.

Writing assembly code with bundles and templates is not trivial. Assembly programmers must know the type of execution unit for each instruction, be it memory, integer, branch, or floating-point. Another important element of the assembly language is the *stop* that separates the instruction stream into groups.

The IA-64 assembly language provides assembly writers with maximum control through the use of two modes for writing assembly code: *explicit* and *automatic*.

When writing in explicit mode, assembly programmers define bundle boundaries, specify the template for each bundle, and insert stops between instructions where necessary. The Assembler only checks that the assembly code is valid and has the right parallelism defined. This mode is recommended for expert assembly programmers or for writing performance-critical code.

The automatic mode significantly simplifies the task of assembly writing while placing the responsibility for bundling the code correctly on the Assembler. The Assembler analyzes the instruction sequence, builds bundles, and adds stops.

```
ld4   r4 = [r33]
add   r8 = 5, r8
mov r2 = r56
add   r32 = 5, r4
mov   r3 = r33
```

Example 2: Original user code

```
{ .mii
      ld4   r4 = [r33]
      add   r8 = 5, r8
      mov   r2 = r56 ;;
}
{ .mmi
      nop.m 0
      add   r32 = 5, r4
      mov   r3 = r33 ;;
}
```

Example 3: Code created after assembly with automatic mode

In the example above we can see how the user can write simple code, which the Assembler will then fit into bundles. The Assembler also adds *nop* instructions for valid template combinations and *stops* as needed to avoid any possibility of dependency violations. (A stop bit was added at the end of the first bundle to avoid a dependency violation between the first instruction and the next to last instruction on r4.)

Parallelism

One of the tasks of the Assembler is to help assembly programmers define the right parallelism boundaries. The Assembler analyzes the instruction stream, taking into consideration the architectural impact of each instruction and any implicit or explicit operands involved in the execution. However, it is hard to do the complete program analysis needed in order to detect all these conditions, statically. Consider a common case in IA-64 architecture where two instructions writing to the same register may be predicated by mutually exclusive predicates, as shown in the Example 4.

```
cmp.ne p2,p3 = r5,r0 ;;
      …
(p2) add r6 = 8, r5
(p3) add r6 = 12, r5
```

Example 4: Predicate relation

The Assembler can identify this case and ignore the apparent dependency violation between the two *add* instructions on *R6*. In this case, the compare instruction, which defines the pair of predicates, precedes their usage. However, more complicated cases may exist. For example, consider a case in which there is a function call between predicates set and usage. In this case, assembly programmers may know that the called function doesn't alter the predicates' values, but there is no way for the assembler to deduce this information, if the function is in a different file.

Another type of information known only at run-time is the program flow at conditional branches. The processor automatically begins a new instruction group when the branch is taken, and a dependency violation may occur only on the fall-through execution path.

When writing in explicit mode, assembly programmers are responsible for stops. The Assembler simply checks the code and reports errors, even when it finds only potential dependency violations. In this mode, to avoid false messages, assembly programmers can add annotations describing predicate relations at that point of the procedure.

```
.pred.rel "imply", p1, p2
(p1) mov r5 = r23
(p2) br.cond.dptk Label1
add r5 = 8, r15
```

Example 5: User annotation

The relation "p1 implies p2", in Example 5 means that if p1 is true then p2 is also true. Adding such a clue to the assembly code prevents a false dependency violation report between the second and fourth lines.

Automatic mode simplifies the programming tasks while delegating the responsibility for valid instruction grouping to the Assembler. In automatic mode, the source code contains no bundle boundaries. The Assembler ignores stops written by the assembly programmer; it builds bundles and adds stops according to the results of static analysis of instructions and program flow. In this mode, the code is guaranteed not to contain dependency violations.

The example below contains a dependency violation which is not immediately apparent. The first instruction writes to CFM, while the second instruction reads from CFM, resulting in a dependency violation. Using automatic mode, the dependency violation is automatically resolved.

```
br.ctop.dptk.many   l8
fpmin   f33=f1,f2
```

Example 6: Code containing dependency violation

```
{ .mib
      nop.m 0
      nop.i 0
      br.ctop.dptk.many   l8 ;;
}
{ .mfi
      nop.m 0
      fpmin   f33=f1,f2
      nop.i 0
}
```

Example 7: Code after automatic mode assembly

In the example above, observe how the Assembler detects a dependency violation on *CFM* between the instructions, and how it inserts a *stop* between them.

Virtual Register Allocation

Large register sets in the IA-64 architecture complement the unique parallelism features. Maintaining assembly code becomes harder when there is a need to track the assignment of many variables to registers. Modifying a procedure code might lead to variable reallocation.

The virtual register allocation (VRAL) feature solves these problems. VRAL allows assembly programmers to use symbolic names instead of registers, and it performs the task of register allocation in procedures.

To employ VRAL, assembly programmers must use a set of VRAL directives in order to communicate some register-related information to the Assembler. Assembly programmers assign groups of physical registers for virtual allocation, and they define the usage policy; i.e., whether they should be scratch registers or registers that are preserved across calls. The Assembler assigns some default families that many assembly programmers are likely to use, including integer, floating point, branch, and predicate. Assembly programmers can also isolate registers of the same type in subfamilies. For example, a user-defined family may include all local registers of a procedure.

Each symbolic name used in a procedure, called a virtual register, belongs to one of the register families. The assembly language allows redefinition of virtual registers' names, which is convenient when used in preprocessor macros.

VRAL analyzes the *control flow graph* of the procedure, and it calculates the registers' live ranges. An accurate control flow graph is very significant for this analysis. The Assembler provides appropriate directives to specify the targets of indirect branches and additional entry points. In order to find a replacement for each symbolic name, VRAL applies standard graph-coloring techniques. The heuristic function used for allocation priorities considers both the results of the preceding analysis and the architecture constraints of registers' usage. Several physical registers may replace one symbolic name, and one physical register may be reallocated and utilized for several different symbolic names.

```
.proc foo
foo::
    alloc loc0=ar.pfs,2,8,0,0
.vreg.safe_across_calls r15-r21
.vreg.safe_across_calls loc3-@lastloc
.vreg.allocatable p6-p9
.vreg.family LocalRegs,loc3-@lastloc
.vreg.var LocalRegs,X,Y,Diff
      mov loc1=b0
      add X=in0,r0
      add Y=in1,r0 ;;
.vreg.var @pred,GT,LE
      cmp.gt GT,LE=X,Y ;;
      (GT) sub Diff=X,Y
      (LE) sub Diff=Y,X ;;
.vreg.redef GT,LE
      mov r8=Diff
      mov ar.pfs=loc0
      mov b0=loc1
      br.ret.dptk b0
.endp foo
```

Example 8: Code with virtual register

Consider the code in the Example 8 above. Starting with ***.vreg.safe_across_calls*** and ***.vreg.allocatable*** directives, we define the registers that are available for allocation. We then use the ***.vreg.family*** directive to define a family of virtual registers that will only be allocated from the local registers. We then define the virtual registers themselves and declare them to be part of the local registers' family defined earlier using the ***.vreg.var*** directive. The code itself then uses virtual registers X and Y instead of directly naming physical registers. The example also illustrates that virtual registers can be defined in the middle of the code, and then undefined with the ***.vreg.redef*** directive to allow reuse of symbolic names (used most frequently in macros).

All symbolic names defined with the directive ***.vreg.var*** are replaced with physical registers assigned for allocation by the directives ***.vreg.allocatable*** and ***.vreg.safe_across_calls***. For this simple example, in the current version of the Assembler, the registers chosen were X=R37, Y=R38, GT=P6, and LE=P7.

In order to effectively use VRAL, we plan to emit the allocation information to allow debugging using symbolic names. This enables the debugger to show the value of

the symbolic name, even if the value is represented by different registers in different parts of the code. Also, by emitting the allocation information, code optimizations without the allocation constraints will be enabled.

THE IA-64 ASSEMBLY ASSISTANT

In order to further assist IA-64 assembly developers, we designed and implemented a unique development environment to have the following:

1. a tool to reduce the steep learning curve for IA-64 assembly programming and to introduce IA-64 architecture using assembly programming
2. a user friendly environment for IA-64 assembly development
3. an environment for analyzing and improving the performance of assembly-code fragments

The Assembly Assistant delivers a comprehensive solution for assembly code developers, assembly language-directed editing, tools that aid the creation of new code, error reporting, static performance analysis, and manual and automatic optimizations.

Other needs such as debugging or run-time performance analysis will be addressed when the Assembly Assistant is integrated with other tools that supply these features.

The next sections describe the Assembly Assistant's editing, assembling, and analysis capabilities in detail and examine the unique features that the Assembly Assistant provides to IA-64 assembly programmers.

Editing and Assembling

The Assembly Assistant provides *syntax-sensitive coloring* that includes all components of assembly code: directives, instructions, comments, templates, and bundling. Every valid instruction format (including completers) is colored to mark it visually as a valid instruction.

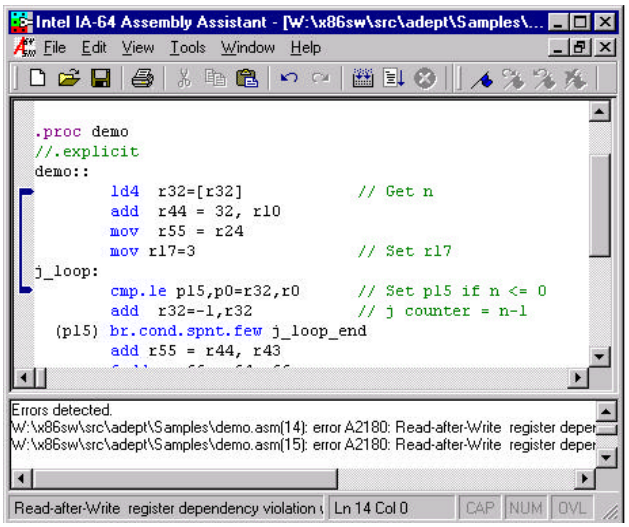
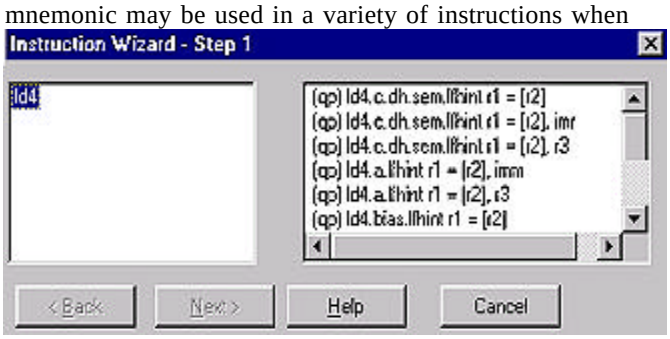


Figure 1: Source code window

The IA-64 instruction set is very rich, and the same mnemonic may be used in a variety of instructions when



combined with different types of operands. The Assembly Assistant provides an *Instruction Wizard* to help assembly programmers select the appropriate instruction with the right set of completers and operands. It allows assembly programmers to choose between instructions in different variations, and it provides a template to select the operands and activate on-line help about the instruction. The example in Figure 2 illustrates how the instruction wizard allows you to choose a specific ld4 form (step 1) and then easily apply the correct completers (step 2). The Help includes some information from ref [1], ref [5], and the *IA-64 Assembly Language Reference Guide*.

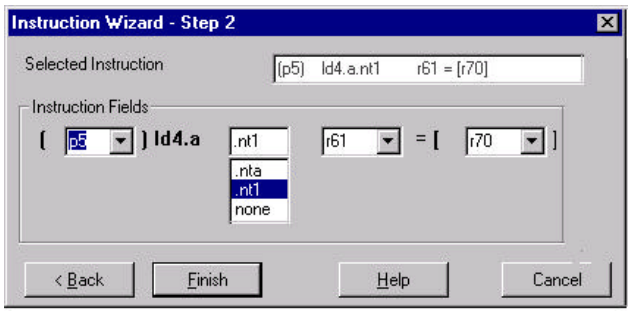


Figure 2: Instruction Wizard

Many assembly programmers need an easy way to interface assembly with other high-level languages such as C* and C++*. Procedures written in assembly must adhere to the *IA-64 Software Conventions* in order to execute correctly. The Assembly Assistant will generate the procedure linkage code given a high-level language-like function prototype. The *Procedure Wizard* generates a procedure template with information that assembly programmers can use to access input parameters and results (see Figure 3).

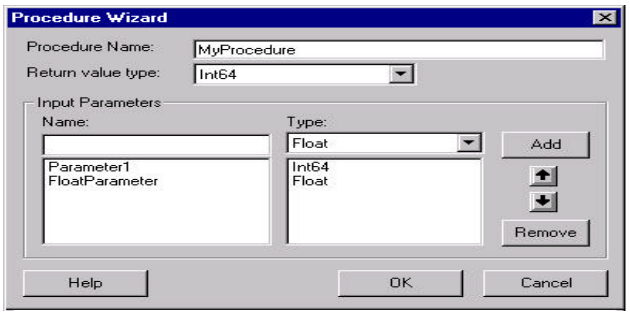


Figure 3: Procedure Wizard

The goal of the *Auto Code Wizard* is to provide an option to retain, customize, and reuse favorite code templates. An example of such a template is the code for integer multiplication. (It is provided as an example in the tool.) The IA-64 architecture does not have a multiplication instruction for general registers so the instruction for floating-point registers must be used. Assembly programmers could write an Auto Code template that moves values from general to floating-point registers, multiplies them, and moves the result back to a general register.

In general, using the source code editor together with wizards and the context-sensitive help provides a rich set of customizable tools to help both beginners and experienced IA-64 assembly programmers.

While browsing errors after compilation is a common task in all development environments, the Assembler identifies a special set of errors called dependency violations (see above). These errors can produce treacherous results, and special care is required while treating them. The difficulty is that these errors involve two instructions that may be distant from one another. When the error in the error view at the bottom of the screen is highlighted, it displays connected pointers pointing to the offending pair of instructions in the source code window (see Figure 1).

ANALYSIS WINDOW

The Assembly Assistant provides a static analysis as a guide to help assembly programmers improve performance. In this section, we discuss the *analysis window*. This window helps assembly programmers understand, browse, analyze, and optimize their assembly code.

The Assembly Assistant uses static performance analysis on a function-by-function basis, without any dynamic information on the program behavior (such as register values and memory access addresses). Fast performance simulation of instruction sequences is used in order to obtain the performance information.

The main performance information displayed in the *analysis view* is *cycle count* and *conflicts*. The cycle count is the estimated relative cycle number in which the instruction enters the execution pipe in the performance simulation. This number is relative to the beginning of the function or selected block. Usually the execution path doesn't utilize the full capacity made possible by the IA-64 architecture. Conflict indicators in the *stall* columns show the reasons for extra cycles in the execution path and processor stalls.

Assembly programmers analyze the conflicts and modify their code accordingly by manually moving an instruction earlier or later in the code sequence, selecting a different instruction sequence, etc. Automatic optimization attempts to find the best instruction scheduling. Optimizations are discussed in a later section.

As shown in Figure 4, the assembly source is presented along with line numbers, cycle counts, and conflicts, as discussed earlier. Conflicts are highlighted with different colors for each conflict, so assembly programmers can easily identify which instructions are involved in each conflict. In Figure 4, the cycle counts are based on a hypothetical machine model.

Another type of information is division of the instruction stream into several types of groups. The most interesting are bundles that are fetched together from memory and instruction groups, which assembly programmers define as candidates for parallel execution. Two additional groups for more advanced assembly programmers are issue groups (instructions that execute simultaneously) and basic blocks (for control flow analysis).

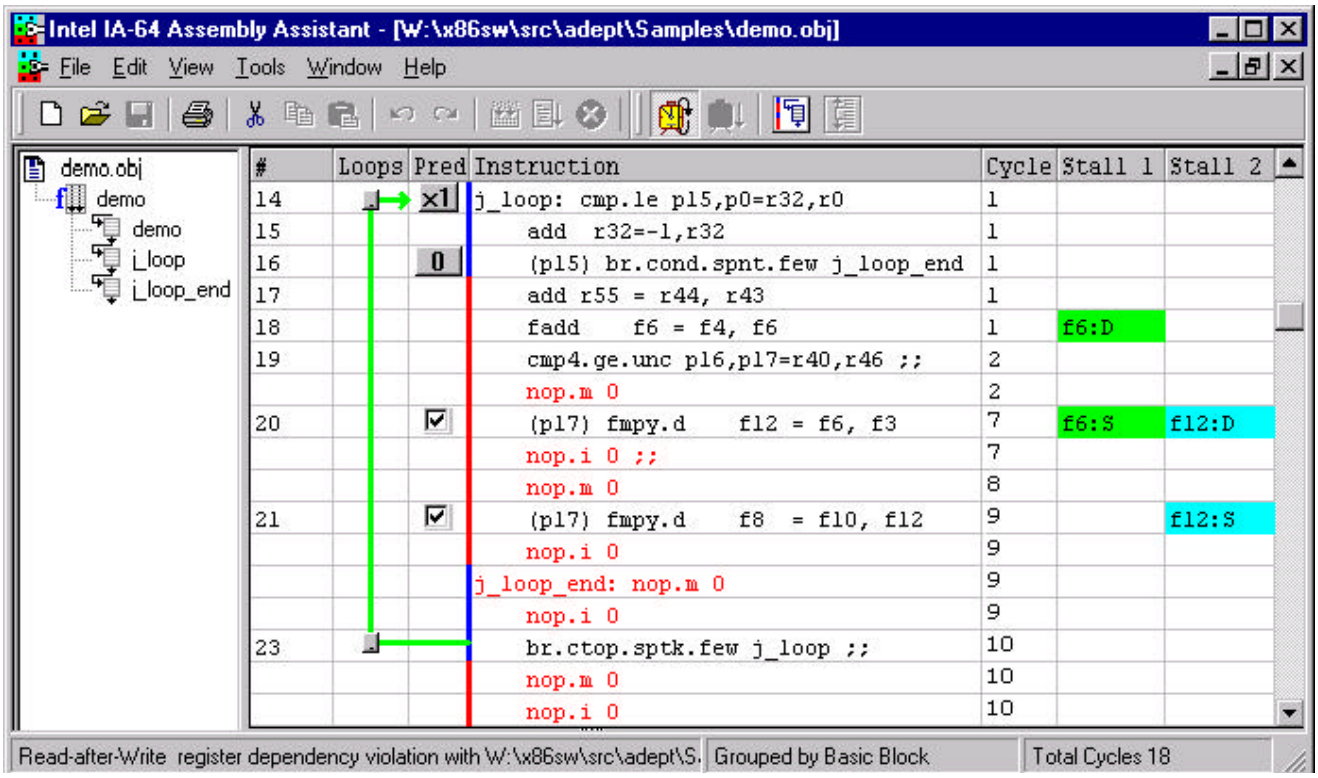


Figure 4: Analysis window

Run-time information is not available during static analysis. The Assembly Assistant provides information about execution flow and predicate values. These values control the simulated execution path and may activate or deactivate instructions. As illustsrated in Figure 4, each predicated instruction is preceded by a checkbox. Marking the checkbox signifies that the qualifying predicate is true, and the instruction executes. This interface eases analysis and optimization of predicated blocks. The Assembly Assistant provides the means to control predicate values. In the future, the Assembly Assistant might also calculate predicate relations for use in the static analysis, and marking a single predicate as true or false will automatically determine the values for all of the predicates that are related to it.

The Assembly Assistant gives assembly programmers more extensive control. Assembly programmers can specify whether or not the branch is taken, and they can set the probability of taking the branch in the simulation. The Assembly Assistant uses this probability when simulating loops: it provides assembly programmers with the approximate time of loop execution together with *a-priori* knowledge of possible execution paths.

The analysis window provides more than just loop visualization. Assembly programmers may select the number of iterations to simulate. Selecting a single iteration provides performance information and shows any conflicts between the main body of the loop and the prologue code. Selecting two iterations also displays conflicts between the head and tail of the loop code section. Selecting more than two iterations provides the approximate execution time of the loop calculated by branch probabilities, as described above.

The assembly module may contain more than one function. To help assembly programmers navigate in the analysis window, the window is split into two panes, just like the Microsoft Explorer* window. The left tree pane contains a list of all the functions in the module, while the right pane displays one function's analyzed code. Clicking on a function name or icon in the tree pane displays the analysis of the selected function. Assembly programmers work with one function at a time, viewing in the tree pane the list of labels in the active function. This allows easy navigation inside the function.

We have described above how assembly programmers can analyze assembly code. But analysis is useless if assembly programmers cannot apply their insights to

* Other brands and names are the property of their respective owners.

improve the code. The Assembly Assistant allows them to do this.

Assembly programmers may want to move stalling instructions and solve conflicts. The Assembly Assistant provides a simple drag-and-drop interface so that instructions can be moved manually. It displays the instruction's move boundaries as defined by data dependencies. Assembly programmers can drop the instructions into their new position. Assembly programmers can also apply automatic optimizations that reschedule the instruction stream to improve performance.

After completing code optimization in the analysis window, the Assembly Assistant generates new, improved code in a new source window.

OPTIMIZATION AIDES IN THE IA-64 ASSEMBLY ASSISTANT

When optimizing assembly code, a tool such as the Assembly Assistant can generally use conventional compiler techniques. However, a key challenge to optimization is code analysis, since some of the information visible to the compiler does not exist or is hard to infer from such low-level representations. Examples include branch targets for indirect branches, calling conventions, memory disambiguation, and aliasing.

This information is critical for code speedup and maintenance of correct code. An assembly optimizer also has no choice but to deal with every feature of the architecture, whereas a compiler might choose not to use certain features or instructions; for example, system instructions.

The first and simplest solution is to leave optimizations to the assembly programmers but still help them with available analysis data and IA-64 processor-specific information. An advanced and friendly user interface enables assembly programmers to easily perform the optimizations.

The next level of automation requires assembly programmers to provide missing information. A user-friendly interface allows assembly programmers to interact with the optimizer to define branch targets and more. Using this information, a control flow graph is created and analyzed. Assembly programmers can also provide program behavior information such as branch probabilities, which direct the optimizer to bias its optimizations accordingly.

Automatic optimization is also used, but as mentioned earlier, it is somewhat limited due to the conservative approach to assembly-level optimizations.

Manual Optimization

Analysis provides many hints for manual optimization. The analysis of register live ranges helps assembly programmers better use the registers. The analysis of data flow provides the assembly programmers with suspected dead code, and it detects the use of registers that were not initialized in the analyzed code. Data flow analysis is also helpful when trying to attain optimal scheduling. Height reduction (the process of reducing the control dependence, for example as in ref. [4]) and strength reduction (ref. [3] p.435) are easier for assembly programmers to handle when all the dependency chain is analyzed automatically.

For software pipelined loops, automatic tracking of the rotating registers used in the loop helps assembly programmers to write modulo-scheduled loops. This can also greatly simplify modification of the code.

It is difficult to keep track of other machine resources (such as control registers, functional units, and more). The Assembly Assistant can automatically keep track of machine resources, and it can warn assembly programmers when machine resources are insufficient for the code. The Assembly Assistant shows an assembly programmer the penalty incurred by the code, and it suggests methods to overcome the limitations inherent in the microarchitecture.

To aid in speculation, when moving a load beyond ambiguous memory references or control dependencies, the Assembly Assistant shows assembly programmers the probable costs and benefits of the speculation. Load instructions that inhibit scheduling can also be identified and they can be suggested to assembly programmers as likely candidates for speculation.

Automatic Optimization

While assembly programmers are certainly capable of performing most optimizations that can be done automatically, other optimizations are difficult. This is due either to complexity or tedium. For example, scheduling the instructions for optimal performance is mostly a problem of brute force. Experienced assembly programmers who are also familiar with all the microarchitecture details can tweak the scheduling to get the best performance, but sometimes the only way to get the best scheduling is to simply try out all of the combinations. The testing would have to be repeated after every source code change. This is clearly a mission for an automated tool.

In automatic optimization mode, the Assembly Assistant schedules the instruction stream in a top-to-bottom issue-group-scheduling approach. Instructions are scheduled according to internal heuristics, taking into account critical

path, code size, instruction priorities, utilization of machine resources, and more. Many possible templates are checked against the heuristics, and the best one is chosen for the issue group. The code in the example below will execute significantly slower than the same code after automatic optimization (~25%). Analyzing the example, we can observe that the instructions at lines 8 and 9 of the original code were moved up, and instructions on lines 3, 4, and 7 were moved down. This schedule was chosen considering the latencies of various functional units and in order to prevent unnecessary stalls.

Original code:

```
1  { .mii
2     ld4    r4 = [r33]
3     add    r8 = 5, r8
4     mov   r2 = r56 ;;
5  }
6  { .mii
7     add    r32 = 5, r4
8     mov    r3 = r33 ;;
9     add    r33 = 4, r3 ;;
10 }
11 { .mib
12    cmp4.lt p6,p7=r2,r33
13    nop.i 999
14    (p7)   br.cond.dpnt START
15 }
```

Code after automatic optimization:

```
1  { .mii
2     ld4           r4 = [r33]
3     mov           r3 = r33 ;;
4     add           r33 = 4, r3
5  }
6  { .mii
7     mov    r2 = r56
8     add           r8 = 5, r8 ;;
9     add           r32 = 5, r4
10 }
11 { .mib
12    cmp4.lt p6,p7=r2,r33
13    nop.i  999
14    (p7) br.cond.dpnt START
15 }
```

Example 9: A small code sample before and after automatic optimization

Optimal utilization of machine resources for parallel execution is very important, and actual results show that even code written by experienced assembly programmers can gain a speed-up of 6 – 8% from the Assembly Assistant's automatic scheduling. In the case of code written by inexperienced assembly programmers, the gain is likely to be much higher.

Assembly programmers can choose from various optimization schemes, actually changing the heuristics used. For example, scheduling for the smallest code size might incur significant penalties due to overloading of machine resources. By default, automatic optimization tries to address the issues most challenging to a human assembly programmer. It optimizes for better utilization of machine resources rather than concern itself with code size. However, this might result in inflated code and affect the instruction cache. Enabling various optimization schemes offers assembly programmers greater control over the automatic optimizations, and it allows expert assembly programmers to take advantage of automatic modes without losing flexibility.

Even for code that is not performance-critical, but still has to be written in assembly, automatic optimization can be valuable. It can be used either to speed up performance or to pack the code more tightly.

FUTURE ENHANCEMENTS FOR THE ASSEMBLY ASSISTANT

The Assembly Assistant is currently used by many IA-64 assembly programmers both beginners and experts to tune their code. We received many requests for more features and enhancements. The requests include a library of optimized special-purpose code (for example, integer divide and floating-point square root), more manual and automatic optimizations, visualization of registers' live ranges, etc. We are also looking into integrating the Assembly Assistant with other programming environments such as Microsoft Visual Studio* and the Intel® VTune™ performance analyzer.

CONCLUSION

It is strongly recommended that compilers be used in order to generate highly optimized code for the IA-64 architecture. The use of compilers also guarantees scalability and portability for future IA-64 implementations. However, we recognize the need of some developers to continue to use assembly code in their applications. We attempted to outline the difficulties faced by assembly programmers when writing for the IA-64 architecture, and we presented tools to alleviate or overcome these difficulties. The tools presented contribute to the following goals:

- Quickly familiarize assembly programmers with the new IA-64 architecture.

* Other brands and names are the property of their respective owners.

- Program in IA-64 assembly with relative ease.
- Provide a comprehensive development environment for assembly programming.
- Analyze and optimize assembly programs to utilize IA-64 unique features for optimal performance.

REFERENCES

[1] *IA-64 Application Developer's Architecture Guide*, Order number 245188.

[2] Analysis of Predicated Code, HPL-96-119.

[3] Steven S. Muchnick, *Advanced compiler design and implementation*, Morgan Kaufmann, San Francisco, California, 1997.

[4] "Control CPR: A Branch Height Reduction Optimization for EPIC Architectures, " in *Proceedings of the ACM SIGPLAN 99 Conference on PLDI*, Atlanta, Georgia 1999, pp.155-168.

[5] *IA-64 Assembler User's Guide*, Order number 712173.

AUTHORS' BIOGRAPHIES

Ady Tal received his M.Sc. degree from the Technion in 1990. He has been working for Intel Israel since 1996, and he leads the Optimizing Libraries development team. He was a member of the Assembler and Assembly Assistant development teams and has wide experience with all aspects of IA-64 architecture features and assembly optimization techniques. His e-mail is ady.tal@intel.com.

Vadim Bassin received an M.A. degree in computing from the Belorussian Radio-Engineering Institute in 1987. While he studied mostly hardware, he has worked only in software. He spent six years programming in real time and working on image processing in the Belorussian Science Academy and a small Israeli-American company before switching to network management. He finally found himself working at Intel on GUI tools for programmers. His e-mail is vadim.bassin@intel.com.

Shay Gal-On received his B.A. degree from the Technion in 1997. Since then he has lingered at Intel Israel, mainly working on assembly optimization techniques and bit manipulations' libraries. Professional interests run from optimizations to security. His e-mail is shay.gal-on@intel.com.

Elena Demikhovsky graduated from the Belorussian Radio-Engineering Institute with an M.Sc. degree in 1991. Since 1994, she has worked at Intel Israel as a software engineer. Elena has wide experience in the development of tools, especially the Assembler, for both the IA-32 and IA-64 architectures. Her e-mail is elena.demikhovsky@intel.com.

Porting Operating System Kernels to the IA-64 Architecture for Presilicon Validation Purposes

Kathy Carver, IA-64 Processor Division, Intel Corporation
Chuck Fleckenstein, IA-64 Processor Division, Intel Corporation
Joshua LeVasseur, IA-64 Processor Division, Intel Corporation
Stephan Zeisset, IA-64 Processor Division, Intel Corporation

Index words: presilicon, validation, COSIM, postsilicon, Mach, Linux

ABSTRACT

To provide additional vehicles for presilicon validation and postsilicon debug of the Intel Itanium™ processor, we ported two operating system kernels to the IA-64 architecture. The Mach3* microkernel was ported first, followed by the Linux* 2.2.0 kernel, and these have helped track the overall health of the Itanium™ processor's RTL model for the last two years. These operating system (OS) kernels also helped presilicon performance analysis and compiler-generated code analysis.

The Mach3 kernel (the IA-64 port was called Munster internally) was ported because it contained features similar to Microsoft Windows NT*, such as tasks, threads, interprocess communication (IPC), and symmetric multiprocessing (SMP). Mach3 allowed us to exercise parts of the Itanium processor's model in a similar way to Windows NT, but at a reduced scale and without device support.

Linux (the IA-64 port was called IPD-Linux) was ported because its source is readily available and 64-bit clean, it is highly configurable, and it would exercise the model in a different way than Mach3. We started with a released 2.2.0 version of the source and ported the kernel using a non-GNU C Compiler (GCC). The difficulty of porting the Linux kernel without GCC made the task more challenging.

Besides porting the architecture-specific portions of the kernels, modifications were necessary to both kernels to remove certain dependencies on external devices and BIOS initialization. Also, the OS initialization paths executed prior to user-level programs had to be shortened to accommodate the simulation speed of the RTL environment. The kernels had to be extremely configurable in order to run in diverse simulation environments.

Both kernels were tested from processor reset to user-mode code execution to validate the significant parts of the RTL that an operating system would exercise during the boot process. Kernel initialization, virtual memory management, context switching, trap handling, system call interfaces, and user-mode context paths were all exercised on the actual RTL model. This effort uncovered several errata in the RTL model and in the IA-64 tools (such as the compiler and linker). It also provided us a model regression sanity check for each new RTL release. We believe this presilicon effort was instrumental in allowing Windows NT to boot just days after first silicon. In this paper, we discuss the porting of kernels to the IA-64 architecture for presilicon operating system validation.

INTRODUCTION

One of the major goals for early silicon is to boot a commercial operating system (OS) shortly after the arrival of first silicon. In order to increase the probability of success we decided to use an operating system kernel to validate the processor in addition to using conventional presilicon testing methods. Traditional microprocessor validation includes feature validation, unit testing, and random instruction testing. The potential shortfall of these methods is that they often don't exercise the processor in the same environment in which it is later expected to run. In other words, an operating system programmer often thinks of a different,

* Other brands and names are the property of their respective owners.

but legal way, of exercising processor functionality that might not be covered by conventional methods of validation. Therefore, running an operating system kernel to exercise key OS-related features presilicon turned out to be a worthwhile effort. The following sections detail the issues we had to resolve during this effort.

RUNNING AN OPERATING SYSTEM IN THE PRESILICON ENVIRONMENT

The two main constraints the presilicon environment imposes on an operating system are as follows:

1. The simulation speed of the RTL simulator effectively restricts test runs to a few million cycles of the simulated processor clock, and it causes turnaround times in the order of multiple days.
2. The simulated environment lacks devices.

The constraint in simulation speed had two major consequences for porting. First we had to reduce the number of instructions executed by the kernel during its initialization sequence. Second, we had to test the kernel thoroughly on functional simulators before committing it to a run on the RTL model.

To cope with the slow simulation speed, we wrote a tool that allowed us to run our kernel up to an arbitrary point in the functional simulator and then to continue simulation on the RTL model from that point on. To accomplish this, our tool read the saved architectural state from the functional simulator and used it to generate a sequence of IA-64 instructions that restored the architecture to this state. Then it read the memory image saved from the functional simulator and used it to generate a new binary, with the state restoration sequence placed at the processor reset vector. When we ran this new binary on the RTL model, the processor went through the state restoration sequence and then continued at the point where the state was saved on the functional simulator. Our two primary uses of this tool were (1) to skip the kernel initialization sequence and have the RTL simulation start directly with the execution of user-mode programs, and (2) to improve the latency for running the kernel initialization sequence on the RTL model by subdividing it into multiple parts and running the parts in parallel. To minimize the danger of processor errata being obscured by cold caches, we allowed for heavy overlap between the parts.

Reducing the Instruction Count

Our initial profiles of the kernel startup sequence for both Munster and IPD-Linux* showed that a large portion of time was spent in the routines for zeroing and copying memory, and in the initialization of a few key data structures, the most prominent being the structures used for virtual memory management. Our solution, therefore, included the following:

- Optimize the routines for zeroing and copying memory (bzero/memset, bcopy/memcopy).
- Reduce the amount of physical memory presented to the kernel. This reduced the time spent initializing page management information.
- Add delayed initialization for some kernel data structures.

These changes, however, did not reduce the functionality of the kernels.

One example of how we modified the Mach kernel to reduce the instruction count during kernel initialization was through changes to the zone allocation code. Most memory allocation for kernel data structures is done through zones, which act as buckets for fixed-size blocks of memory (zone entries) whose typical size ranges from a few bytes to a few hundred bytes. When an entry was allocated from a zone that had no entries in its free list, the free list was replenished by allocating one page of memory and splitting it up into zone entries, all linked together in a free list. The number of instructions required for doing this initialization for dozens of zones was quite high when keeping the speed of RTL simulation in mind. Therefore we changed the mechanism for replenishing a zone. Instead of immediately entering a whole page into the free list, a “free space” pointer was kept. The pointer was initially set to the newly allocated page, and it was used to carve out new zone entries one by one at the time they were actually needed.

Testing in Different Environments

Figure 1 shows our available simulation environments. Except for device support, matching environments were available on the functional and the RTL simulator. Even though no device support was available on RTL, we still needed to test our kernels with devices on the functional level to prepare for postsilicon.

* Other brands and names are the property of their respective owners.

Our kernels had to run in several different simulation environments. We ported the kernels so they could be configured with or without devices and run with or without external interrupts, etc. Our kernels were flexible enough to run in simulation environments that ranged from just one processor with memory to a full simulation of a multiprocessor platform with devices.

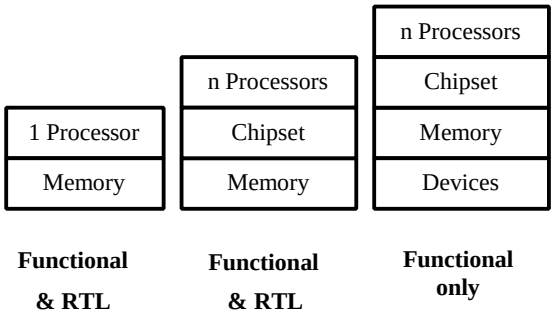


Figure 1: Available simulation environments

We used two functional simulators, Giza and SoftSDV [8], to test and debug the presilicon operating system kernels before running in RTL. Both simulators were utilized in our development process in order to debug code quickly. Giza was also designed to be used as a checker against the RTL model, so we always ran our code through it before running in RTL. Since the SoftSDV simulator is already described in "SoftSDV: A Presilicon Software Development Environment for the IA-64 Architecture" in this issue of the *Intel Technology Journal*, we only describe the Giza simulator.

Giza is built around an instruction accurate software simulator for Itanium processor's ISA (Sphinx). It supports critical implementation specific registers, SAPIC, a non-blocking memory hierarchy (TLB+caches) that handles both synchronous and asynchronous traffic between the CPU and the external sub-system, and multiple CPU instances (multiprocessor). Implementation-specific registers are modeled to support firmware execution. SAPIC, non-blocking memory hierarchy, and multiprocessor (MP) are modeled to support characteristic subsystem traffic for typical IA-64 platforms. A functional accurate software model that mimics the Itanium processor's front-side-bus (FSB) is designed to schedule CPU events and dispatch the resulting transactions to and from memory and I/O subsystems. Software models for the Itanium processor's chipset and Itanium processor's standard devices represent the latter.

By using functional simulators, we avoided wasting precious RTL cycles that could be used by conventional tests. We began with uniprocessor (UP) versions of the functional simulators and OS kernels. Once we passed the UP functional simulator test, the code would run on the RTL. These jobs often took over a million cycles to complete so the ramifications of simple code mistakes were great and had to be eliminated before being run on the RTL models. Once the kernels passed a UP functional and RTL simulator run, they were moved onto the multiprocessor path. Each symmetric multiprocessing (SMP) version of the kernel was debugged via a functional simulator. The MP RTL environment, known as COSIM, allowed modeling of multiple IA-64 RTL processor models, chipset models, PCI busses, and external interrupt controllers. This environment allowed us to exercise SMP kernels on many of the platform components before silicon was available, which taught us valuable lessons and uncovered errata that were not uncovered during conventional methods of testing.

Since operating system code is not "self checking," the Munster and IPD Linux kernels were run in RTL with an RTL checker running at the same time. The RTL checker is a functional simulator that runs in conjunction with the RTL simulation and compares the architectural state after the retirement of each bundle. If a state mismatch occurs, then an error condition is flagged, and further analysis can be done to isolate the root of the problem.

Porting Challenges

There were many challenges in porting the kernels to run presilicon in RTL. We encountered a number of tool problems since we were on the leading edge as far as running code with the actual RTL model is concerned. The early tool sets often worked for running code in the functional simulator, but had problems with generating correct code for running in the RTL simulator. We had to write a utility called the AfterBurner to post-process compiler-generated assembly code and to fix problems that were preventing the code from running in RTL.

During the project, the compiler-generated code quality (correctness and performance) improved, as did the modeling of the architecture by the functional simulators. However, for some sequences of legal C code, the compiler produced semantically incorrect as well as architecturally incorrect code. In certain cases, due to the sequential nature of the functional simulator, architecturally incorrect code would appear to function correctly. In other cases, architecturally incorrect hand-written code would appear to execute correctly within the functional simulator (e.g., missing serialization instructions required by the architecture went undetected).

Benefits of Using Two Different Kernels

The benefit of porting multiple kernels to The Itanium processor was the ability to share some of the low-level start-up, trap handling (TLB faults, etc.), bcopy, port IO usage, and other code between the two kernels. It took us roughly two weeks to obtain a linkable Linux IA-64 kernel, and much of that time was spent on accommodating a non-GNU [11] C compiler. Much of the low-level code was already done from the Mach* port and just had to be merged into the Linux source tree. Another benefit of a second port was that it allowed us to redesign some of the code to make it cleaner and more efficient.

Porting two kernels allowed us to test some of the IA-64 Instruction Set Architecture in a slightly different way. We achieved a broader validation of some features such as Instruction Level Parallelism (ILP), speculation, predication, use of the large register files, the Register Stack Engine (RSE), and advanced branch architecture. Our "common trap handler," the common path for saving and restoring state when entering/exiting the kernel, for the IPD Linux was very different from the Mach* version in both the design of the operating system and in the area of performance. As a result, the processor was exercised in an alternate way.

Both kernels supported Seamless mode, which is the ability to run IA-32 binaries on top of an IA-64 operating system kernel. We ran IA-32 user-mode programs on the kernels in presilicon RTL as another validation test.

ISSUES SPECIFIC TO PORTING MUNSTER

Mach3 [6] was the first kernel to be ported so the architecture-dependent code had to be written from scratch. We received some example code from other Intel groups, but some of it didn't fit very well into the Mach3 architecture. One of the biggest issues that we encountered was Mach3's ability to come into kernel mode on one stack and leave on another [15]. This added complexity to the trap handler due to the fact that all of the required IA-64 state had to be saved on to the Process Control Block (PCB) and restored into the new stack state. In Mach*, instead of a static assignment between threads and kernel stacks, the assignment is dynamic, and a thread that blocks in kernel context while waiting for some event can hand off its stack to the thread that is next in line. This is beneficial in terms of cache locality, but it complicates handling of the register stack engine because the kernel backing store is part of the kernel stack. As such, it does not persist between the time a thread enters the kernel and the time it returns. When entering the kernel from user mode, we first had to flush the dirty RSE registers into a dedicated area in the process control block before switching to the kernel backing store. Then, when we returned to user mode, we had to load the flushed RSE registers from the process control block into the physical register file before switching to the user backing store.

There were also LP-64 issues in the Mach3 source code where assumptions were made that ints, longs, and pointers were all the same size. This caused pointers to be truncated in some cases.

The Munster kernel port involved IA-64 start-up code, fault handling, TLB handling, context switching, system calls, interrupt handling, interprocess communication, LP 64-bit clean efforts, and user-mode libraries. We also had to port the Mach* build tools to UnixWare* before the Mach3 kernel could be built.

ISSUES SPECIFIC TO PORTING IPD-LINUX

Linux* was ported as another presilicon operating system validation kernel. (The source for the Trillian* kernel, which was demonstrated during the 1999 Intel Developer's Forum was not available when this port began.) The main issue with porting Linux to IA-64 presilicon was the lack of a complete IA-64 GNU C compiler [13] at the time we began the port. We used the Intel Electron C compiler to compile the kernel. This required us to conditionally compile around the heavy usage of GNU C extensions [12] within the Linux kernel. Extensions such as inline C and inline assembly functions are not supported by the Electron compiler so this made the port more difficult than if we had a GNU C compiler available. The majority of our work in this port was in the two architecture-dependent directories that we added (include/asm-ia64 and arch/ia64). We also added an inline directory under arch/ia64 as a substitute for those routines that are normally inlined by the GNU C compiler. Since we didn't have access to a GNU C compiler early in our development cycle, a basic user-mode shell, Josh, was written and used to launch Linux tests for validation purposes. Without a full GNU C compiler it proved very difficult to port the GNU C Library (GLIB C), which forms the basis for the full set of user-level shells and commands. Attempts were made to port GLIBC without the GNU C compiler, but they were unsuccessful in presilicon.

* Other brands and names are the property of their respective owners.

ISSUES SPECIFIC TO SUPPORTING PRESILICON PERFORMANCE ANALYSIS

The Linux port was also chosen as a vehicle to facilitate architectural performance research. The study of performance phenomena related to the microarchitecture, architecture, operating system, software, and tools required an open source workload. Besides offering a complete source, Linux offers SMP support, 64-bit clean code, runtime C library, and a kernel designed to work with multiple architectures. All of these features were integral to quickly satisfy the group's goals.

To automate data collection and analysis, the Linux kernel was augmented with many hooks to communicate with the simulation infrastructure. The hooks reported information about the internal state of the kernel, which were recorded within an event trace. The simulator and post processing tools correlate the information with architecture events, debug information, and compiler annotations. The kernel was also instrumented to support efficient branch and data trace collection when run on silicon.

To integrate the Linux kernel with the trace environment, and to support the silicon trace collection, a common feature set was added to the kernel. The kernel additions collect information about context switches, process creation and termination, and modifications to the address space (such as through mmap() or munmap()). This type of data collection has proven useful for other domains such as checkpoint/restore (the EPCKPT project), and kernel profiling (Intel Vtune™ performance analyzer [14]).

PRESILICON RESULTS

By running the operating systems presilicon, we found several unique RTL errata that would have affected commercial operating systems postsilicon. Errata were found in operating system-specific code, compiler-generated code sequences, speculative execution, platform interrupt paths, and tools. Other testing methods did not find these errata. Therefore, since this was a new architecture, it was worthwhile to incorporate an operating system kernel test presilicon to rule out major issues and to provide an indicator of the overall RTL model health.

Operating System-Specific Errata

The first errata we uncovered was related to the return from interrupt (rfi) instruction that is used by operating systems to return from an interruption/fault. The error occurred when the operating system start-up code used this instruction in the process of switching from physical mode to virtual mode, and the new instruction address was only valid in the new addressing mode. In checking the validity of the target address of the rfi instruction, the processor was using the previous (physical) addressing mode instead of the new (virtual) one, generating an exception. This prevented our kernels from booting and could have affected other operating system kernels like Trillian Linux and Windows NT*.

Errata Uncovered by Compiler-Generated Code Sequences

Presilicon OS runs uncovered an error in the Itanium processor's RTL where certain combinations of floating-point instructions produced an incorrect result because sequences of multiply-accumulate instructions with register dependencies were not properly stalled. The significance of this error is that this exact instruction sequence is used by the C compiler to implement the integer modulo operation. Since C is the primary language for software development, this error would have been encountered by most applications running on the processor.

Code Example:

```
int i, x, y ;

i=x%y ;
```

where the generated code would contain a sequence like the following (note the pseudo registers for the example):

```
fma fz=fa,fb,fc
;;
fma fw=fz,fk,fj
```

The result of the first fma is not available for several cycles, so the second fma instruction should have been stalled until fz was available.

Speculative Execution Errata

The Itanium processor does extensive branch prediction and speculatively executes instructions at the predicted branch target long before it is known if the branch will be taken. We found a problem with a conditional call to a subroutine, where the subroutine was short enough to execute a return instruction before it was known if the conditional call should have been executed. This caused the processor to permanently commit some of the state changes caused by the return instruction, even if the conditional call was incorrectly predicted and was not supposed to be executed.

---

* Other brands and names are the property of their respective owners.

Example:

// where p3 is false (p3) br.call b0 = foo	**foo:** alloc … mov … ;; br.ret b0

In this pseudo code example the predicate p3 was false, but the code at **foo** was speculatively executed, and its results were erroneously committed.

Platform Interrupt Errata

Several interrupt-related errata were uncovered by writing small tests that exercised the interrupt paths utilized by an operating system on a typical Itanium platform. This testing was accomplished in a simulation environment (COSIM) that combined multiple Itanium processor models, the chipset model, and platform component models such as the external interrupt controller model. This allowed exercising the path from a simulated device to the processor.

Unique errata were uncovered by generating interrupts in the modeled environment, causing the execution of specific interrupt flow paths. One of the interrupt errors was uncovered by redirecting interrupts to a particular processor based upon the priority of the processor in a multiprocessor simulation.

Tools Errata

During the development and testing of Munster and IPD Linux, many bugs were found in software development tools such as compilers and linkers, and also in various simulators used to run the kernels. Munster and IPD Linux were run in every simulator that was available to us and were crucial in detecting multiprocessor functional simulator errors.

POST-SILICON RESULTS

The great advantage of using a kernel for postsilicon debug that has been validated in the presilicon environment is that it removes potential software bugs from the list of unknowns during initial bring-up.

The extensive presilicon testing allowed the bring-up team to concentrate on mechanical, electrical, and silicon issues and provided them with a metric for assessing bring-up progress.

First Itanium processor's silicon was very healthy, so our kernels were able to run without modification as soon as initial platform issues were resolved and a stable operating range for the processor was found.

CONCLUSION

Testing an RTL model using an operating system kernel consumes many RTL cycles. The tests typically run for over one million cycles and take up cycles that could be used by shorter focus tests. This is the reason that our code was debugged on a functional simulator before launching the tests on the RTL model. We expected the kernels to boot if the model was healthy and if there were no infrastructure problems with the testing environment. The advantages of using an operating system far outweigh the disadvantages. We were able to find errata presilicon that normally would not have been detected until hardware was available. At that point, the problems can be difficult to isolate and expensive to fix. The kernels were also valuable during the first few days of Itanium processor's postsilicon bring-up. Our kernels were able to run without modification as soon as the silicon and platform hardware were stable. This allowed us to use our kernels as an indicator of hardware health during the first few days.

ACKNOWLEDGMENTS

These people were always there to support this unique project: Piyush Desai, Andy Pfiffer, Rahul Bhatt, Stan Smith, Rumi Zahir, Drew Hess, Tony Porterfield, Moenes Iskarous, Dennis Marer, Ken Arora, and Randy Anderson.

REFERENCES

[1] Carver, Fleckenstein, et al., "A System Centric Reference Model and an OS Microkernel for Pre-Silicon CPU Platform Validation," *Intel DTTC 1998*, (internal document).

[2] Alessandro Rubini, "Linux Device Drivers," O'Reilly & Associates, February 1998, ISBN: 1-56592-292-1.

[3] Linux Kernel Source Archives (best porting reference), ftp://ftp.kernel.org/pub

[4] M. Beck, H. Bohme, et al., Linux Kernel Internals (Second Edition), Addison-Wesley, 1998, ISBN: 0-201-33143.

[5] *IA-64 Application Developer's Architecture Guide*, Order Number: 245188-001, May 1999. http://developer.intel.com/design/IA64

[6] CMU CS Project Mach* Home Page, http://www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/mach.html

[7] OSF Kernel and Server Programming Manuals, http://www.cs.cmu.edu/afs/cs/project/mach/public/www/doc/osf.html

[8] R. Uhlig, R. Fishtein, O. Gershon, H. Wang, "SoftSDV: A Presilicon Software Development Environment for the IA-64 Architecture," *Intel Technology Journal*, Q4 1999.

[9] Gollakota, Naga, "COSIM," *Intel DTTC* 1998.

[10] H.P. Messmer, *The Indispensable PC Hardware Book*, Addison-Wesley, 1997.

[11] GNU's Not Unix !, http://www.gnu.ai.mit.edu

[12] "Using and Porting the GNU Compiler Collection (GCC), Extensions to the C Language Family",

http://www.gnu.org/software/gcc/onlinedocs/gcc_4.html#SEC62

[13] "GCC Home Page",

http://www.gnu.org/software/gcc/gcc.html

[14] "Vtune™ Performance Analyzer," http://developer.intel.com/vtune/analyzer/index.htm

[15] Draves, R.P., Bershad, B., Rashid, R. F., and Dean, R.W., "Using Continuations to Implement Thread Management and Communication in Operating Systems," in *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, 1991, ftp://ftp.cs.cmu.edu//afs/cs/project/mach/public/doc/unpublished/internals_slides/Continuations/sosptr.ps

[16] Wheeler, Bob, "Porting and Modifying the Mach* 3.0 Microkernel," *Third USENIX MACH* Symposium*, 1993, ftp://ftp.cs.cmu.edu/afs/cs/project/mach/public/doc/published/porting.tutorial.slides.ps

[17] Rusling, David, "The Linux Kernel," http://www.redhat.com/mirrors/LDP/LDP/tlk/tlk.html

AUTHORS' BIOGRAPHIES

Kathy Carver received a B.S. degree in computer science and mathematics from Western Kentucky University in 1981. She joined Intel Corporation in 1992 and worked on compiler validation and integration for Intel's IPSC860, Paragon, and TFLOPS systems. She is currently part of the Itanium processor Architecture Validation group in IPD. Her e-mail is kat@co.intel.com

Chuck Fleckenstein is currently working for the IPD Itanium processor validation group. He received an M.S. degree in computer science from Wright State University in 1988. His interests include distributed programming languages and operating systems. His e-mail is cfleck@co.intel.com.

Joshua LeVasseur works for the IPD IA-64 Architecture group. He currently focuses on system architecture performance, IA-64 optimizations, and simulator/analysis infrastructure. His e-mail is joshua.levasseur@intel.com

Stephan Zeisset received an M.S. degree in computer science from the Munich University of Technology in 1994. Since then he has been working on the operating system of the TFLOPS system and its predecessors, with a specialization in distributed memory management. Stephan is currently working on Itanium processor validation for IPD. His e-mail is sz@co.intel.com

The Computation of Transcendental Functions on the IA-64 Architecture

John Harrison, Microprocessor Software Labs, Intel Corporation
Ted Kubaska, Microprocessor Software Labs, Intel Corporation
Shane Story, Microprocessor Software Labs, Intel Corporation
Ping Tak Peter Tang, Microprocessor Software Labs, Intel Corporation

Index words: floating point, mathematical software, transcendental functions

ABSTRACT

The fast and accurate evaluation of transcendental functions (e.g. exp, log, sin, and atan) is vitally important in many fields of scientific computing. Intel provides a software library of these functions that can be called from both the C* and FORTRAN* programming languages. By exploiting some of the key features of the IA-64 floating-point architecture, we have been able to provide double-precision transcendental functions that are highly accurate yet can typically be evaluated in between 50 and 70 clock cycles. In this paper, we discuss some of the design principles and implementation details of these functions.

INTRODUCTION

Transcendental functions can be computed in software by a variety of algorithms. The algorithms that are most suitable for implementation on modern computer architectures usually comprise three steps: reduction, approximation, and reconstruction.

These steps are best illustrated by an example. Consider the calculation of the exponential function exp(x). One may first attempt an evaluation using the familiar Maclaurin series expansion:

$$\exp(x) = 1 + x + x^2/2! + x^3/3! + \ldots + x^k/k! + \ldots .$$

When x is small, computing a few terms of this series gives a reasonably good approximation to exp(x) up to, for example, IEEE double precision (which is approximately 17 significant decimal digits). However, when x is large, many more terms of the series are needed to satisfy the same accuracy requirement. Increasing the number of terms not only lengthens the calculation, but it also introduces more accumulated rounding errors that may degrade the accuracy of the answer.

To solve this problem, we express x as

$$x = N \ln(2) / 2^K + r$$

for some integer K chosen beforehand (more about how to choose later). If $N \ln(2)/2^K$ is made as close to x as possible, then $|r|$ never exceeds $\ln(2)/2^{K+1}$. The mathematical identity

$$\exp(x) = \exp(N \ln(2) / 2^K + r) = 2^{N/2^K} \exp(r)$$

shows that the problem is transformed to that of calculating the exp function at an argument whose magnitude is confined. The transformation to r from x is called the reduction step; the calculation of exp(r), usually performed by computing an approximating polynomial, is called the approximation step; and the composition of the final result based on exp(r) and the constant related to N and K is called the reconstruction step.

In a more traditional approach [1], K is chosen to be 1 and thus the approximation step requires a polynomial with accuracy good to IEEE double precision for the range of $|r| \le \ln(2)/2$. This choice of K leads to reconstruction via multiplication of exp(r) by 2^N, which is easily implementable, for example, by scaling the exponent field of a floating-point number. One drawback of this approach is that when $|r|$ is near $\ln(2)/2$, a large number of terms of the Maclaurin series expansion is still needed.

More recently, a framework known as table-driven algorithms [8] suggested the use of $K > 1$. When for example $K=5$, the argument r after the reduction step would satisfy $|r| \le \log(2)/64$. As a result, a much shorter polynomial can satisfy the same accuracy requirement. The tradeoff is a more complex reconstruction step, requiring the multiplication with a constant of the form

$$2^{N/32} = 2^M 2^{d/32}, \quad d = 0, 1, \ldots, 31,$$

*All other brands and names are the property of their respective owners.

where $N = 32M + d$. This constant can be obtained rather easily, provided all the *32* possible values of the second factor are computed beforehand and stored in a table (hence the name table-driven). This framework works well for modern machines not only because tables (even large ones) can be accommodated, but also because parallelism, such as the presence of pipelined arithmetic units, allow most of the extra work in the reconstruction step to be carried out while the approximating polynomial is being evaluated. This extra work includes, for example, the calculation of *d*, the indexing into and the fetching from the table of constants, and the multiplication to form $2^{N/32}$. Consequently, the performance gain due to a shorter polynomial is fully realized.

In practice, however, we do not use the Maclaurin series. Rather, we use the lowest-degree polynomial *p(r)* whose worst deviation $|p(r) - \exp(r)|$ within the reduced range in question is minimized and stays below an acceptable threshold. This polynomial is called the *minimax* polynomial. Its coefficients can be determined numerically, most commonly by the Remez algorithm [5].

Another fine point is that we may also want to retain some convenient properties of the Maclaurin series, such as the leading coefficient being exactly 1. It is possible to find minimax polynomials even subject to such constraints; some examples using the commercial computer algebra system Maple are given in reference [3].

DESIGN PRINCIPLES ON THE IA-64 ARCHITECTURE

There are tradeoffs to designing an algorithm following the table-driven approach:

- Different argument reduction methods lead to tradeoffs between the complexity of the reduction and the reconstruction computation.
- Different table sizes lead to tradeoffs between memory requirements (size and latency characteristics) and the complexity of polynomial computation.

Several key architectural features of IA-64 have a bearing on which choices are made:

- Short floating-point latency: On IA-64, the generic floating-point operation is a "fused multiply add" that calculates $A \acute{} B + C$ per instruction. Not only is the latency of this floating-point operation much shorter than memory references, but this floating-point operation consists of two basic arithmetic operations.
- Extended precision: Because our target is IEEE double-precision with 53 significant bits, the native 64-bit precision on IA-64 delivers 11 extra bits of accuracy on basic arithmetic operations.
- Parallelism: Each operation is fully pipelined, and multiple floating-point units are present.

As stated, these architectural features affect our choices of design tradeoffs. We enumerate several key points:

- Argument reduction usually involves a number of serial computation steps that cannot take advantage of parallelism. In contrast, the approximation and reconstruction steps can naturally exploit parallelism. Consequently, the reduction step is often a bottleneck. We should, therefore, favor a simple reduction method even at the price of a more complex reconstruction step.
- Argument reduction usually requires the use of some constants. The short floating-point latency can make the memory latency incurred in loading such constants a significant portion of the total latency. Consequently, any novel reduction techniques that do away with memory latency are welcome.
- Long memory latency has two implications for table size. First, large tables that exceed even the lowest-level cache size should be avoided. Second, even if a table fits in cache, it still takes a number of repeated calls to a transcendental function at different arguments to bring the whole table into cache. Thus, small tables are favored.
- Extended precision and parallelism together have an important implication for the approximation step. Traditionally, the polynomial terms used in core approximations are evaluated in some well specified order so as to minimize the undesirable effect of rounding error accumulation. The availability of extended precision implies that the order of evaluation of a polynomial becomes unimportant. When a polynomial can be evaluated in an arbitrary order, parallelism can be fully utilized. The consequence is that even long polynomials can be evaluated in short latency.

Roughly speaking, latency grows logarithmically in the degree of polynomials. The permissive environment that allows for functions that return accurate 53-bit results should be contrasted with that which is required for functions that return accurate 64-bit results. Some functions returning accurate 64-bit results are provided in a special double-extended `libm` as well as in IA-32 compatibility operations [6]. In both, considerable effort was taken to minimize rounding error. Often, computations were carefully choreographed into a dominant part that was calculated exactly and a smaller part that was subject to rounding error. We frequently stored precomputed values in two pieces to maintain

intermediate accuracy beyond the underlying precision of 64 bits. All these costly implementation techniques are unnecessary in our present double-precision context.

We summarize the above as four simple principles:

1. Use a simple reduction scheme, even if such a scheme only works for a subset of the argument domain, provided this subset represents the most common situations.
2. Consider novel reduction methods that avoid memory latency.
3. Use tables of moderate size.
4. Do not fear long polynomials. Instead, work hard at using parallelism to minimize latency.

In the next sections, we show these four principles in action on Merced, the first implementation of the IA-64 architecture.

SIMPLE AND FAST RANGE REDUCTION

A common reduction step involves the calculation of the form

$$r = x - N\,\mathbf{r}.$$

This includes the forward trigonometric functions sin, cos, tan, and the exponential function exp, where $\mathbf{r}$ is of the form $\mathbf{p}/2^K$ for the trigonometric functions and of the form $\ln(2)/2^K$ for the exponential function. We exploit the fact that the overwhelming majority of arguments will be in a limited range. For example, the evaluation of trigonometric functions like sin for very large arguments is known to be costly. This is because to perform a range reduction accurately by subtracting a multiple of $\mathbf{p}/2^K$, we need to implicitly have a huge number of bits of $\mathbf{p}$ available. But for inputs of less than 2^{10} in magnitude, the reduction can be performed accurately and efficiently. The overwhelming majority of cases will fall within this limited range. Other more time-consuming procedures are well known and are required when arguments exceed 2^{10} in magnitude (see [3] and [6]).

The general difficulty of range reduction implementation is that $\mathbf{r}$ is not a machine number. If we compute:

$$r = x - N\,P$$

where the machine number P approximates $\mathbf{p}/2^K$, then if x is close to a root of the specific trigonometric function, the small error, $\mathbf{e} = |P - \mathbf{p}/2^K|$, scaled up by N, constitutes a large relative error in the final result. However, by using number-theoretic arguments, one can see that when reduction is really required for double-precision numbers in the specified range, the result of any of the trigonometric functions, sin, cos, and tan, cannot be smaller in magnitude than about 2^{-60} (see [7]).

The worst relative error (which occurs when the result of the trigonometric function is its smallest, $2^{60,}$ and N is close to 2^{10+K}) is about $2^{70+K}\,\varepsilon$. If we store P as two double-extended precision numbers, $P\_1 + P\_2$, then we can make $\mathbf{e} < 2^{-130-K}$ sufficient to make the relative error in the final result negligible.

One technique to provide an accurate reduced argument on IA-64 is to apply two successive `fma` operations

$$r_0 = x - N\,P\_1; \quad r = r_0 - N\,P\_2.$$

The first operation introduces no rounding error because of the well known phenomenon of cancellation.

For sin and cos, we pick K to be 4, so the reconstruction has the form

$$\sin(x) = \sin(N\mathbf{p}/16)\cos(r) + \cos(N\mathbf{p}/16)\sin(r)$$

and

$$\cos(x) = \cos(N\mathbf{p}/16)\cos(r) - \sin(N\mathbf{p}/16)\sin(r).$$

Periodicity implies that we need only tabulate $\sin(N\mathbf{p}/16)$ and $\cos(N\mathbf{p}/16)$ for $N = 0, 1, \ldots, 31$.

The case for the exponential function is similar. Here $\ln(2)/2^K$ (K is chosen to be 7 in this case) is approximated by two machine numbers $P\_1 + P\_2$, and the argument is reduced in a similar fashion.

NOVEL REDUCTION

Some mathematical functions f have the property that

$$f(u\,v) = g(f(u), f(v))$$

where g is a simple function such as the sum or product operator. For example, for the logarithm, we have (for positive u and v)

$$\ln(u\,v) = \ln(u) + \ln(v) \quad (g \text{ is the sum operator})$$

while for the cube root, we have

$$(u\,v)^{1/3} = u^{1/3}\,v^{1/3} \quad (g \text{ is the product operator}).$$

In such situations, we can perform an argument reduction very quickly using IA-64's basic floating-point reciprocal approximation (`frcpa`) instruction, which is primarily intended to support floating-point division. According to its definition, `frcpa`(a) is a floating-point with 11 significant bits that approximates $1/a$ using a lookup on the top 8 bits of the (normalized) input number a. This 11-bit floating-point number approximates $1/a$ to about 8 significant bits of accuracy. The exact values returned are specified in the IA-64 architecture definition. By enumeration of the approximate reciprocal values, one can show that for all input values a,

$$\texttt{frcpa}(a) = (1/a)\,(1 - \mathbf{b}), \quad |\mathbf{b}| \le 2^{-8.86}.$$

We can write $f(x)$ as

$$f(x) = f(x \cdot \texttt{frcpa}(x) / \texttt{frcpa}(x))$$
$$= g(\ f(x \cdot \texttt{frcpa}(x)),\ f(1/\texttt{frcpa}(x))\ ).$$

The $f(1/\texttt{frcpa}(x))$ terms can be stored in precomputed tables, and they can be obtained by an index based on the top 8 bits of x (which uniquely identifies the corresponding frcpa (x)).

Because the *f's* we are considering here have a natural expansion around 1,

$$f(x \cdot \texttt{frcpa}(x))$$

is most naturally approximated by a polynomial evaluated at the argument $r = x \cdot \texttt{frcpa}(x) - 1$. Hence, a single fma constitutes our argument reduction computation, and the value frcpa(x) is obtained without any memory latency.

We apply this strategy to $f(x) = \ln(x)$.

$$\ln(x) = \ln(1/\texttt{frcpa}(x)) + \ln(\texttt{frcpa}(x)\, x)$$
$$= \ln(1/\texttt{frcpa}(x)) + \ln(1 + r)$$

The first value on the right-hand side is obtained from a table, and the second value is computed by a minimax polynomial approximating $\ln(1+r)$ on $|r| \le 2^{-8.8}$. The quantity $2^{-8.8}$ is characteristic of the accuracy of the IA-64 frcpa instruction.

The case for the cube root function cbrt is similar.

$$(x)^{1/3} = (1/\texttt{frcpa}(x))^{1/3} (\texttt{frcpa}(x)\, x)^{1/3}$$
$$= (1/\texttt{frcpa}(x))^{1/3} (1 + r)^{1/3}.$$

The first value on the right-hand side is obtained from a table, and the second value is computed by a minimax polynomial approximating $(1+r)^{1/3}$ on $|r| \le 2^{-8.8}$.

MODERATE TABLE SIZES

We tabulate here the number of double-extended table entries used in each function. The trigonometric functions sin and cos share the same table, and the functions tan and atan do not use a table at all.

Function	Number of Double-Extended Entries
cbrt	256 (3072 bytes)
exp	24 (288 bytes)
Ln	256 (3072 bytes)
sin, cos	64 (768 bytes)
tan	None
atan	None

Table 1: Table sizes used in the algorithms

Table 1 does not include the number of constants for argument reduction nor does it include the number of coefficients needed for evaluating the polynomial.

OPTIMAL EVALUATION OF POLYNOMIALS

The traditional Horner's rule of evaluation of a polynomial is efficient on serial machines. Nevertheless, a general degree-n polynomial requires a latency of n fma's. When more parallelism is available, it is possible to be more efficient by splitting the polynomial into parts, evaluating the parts in parallel, and then combining them. We employ this technique to the polynomial approximation steps for all the functions. The enhanced performance is crucial to the cases of tan and atan where the polynomials involved are of degrees 15 and 22. Even for the other functions where the polynomials are varying in degree from 4 to 8, our technique also contributes to a noticeable gain over the straightforward Horner's method. We now describe this technique in more detail.

Merced has two floating-point execution units, so there is certainly some parallelism to be exploited. Even more important, both floating-point units are fully pipelined in five stages. Thus, two new operations can be issued *every* cycle, even though the results are then not available for a further five cycles. This gives much of the same benefit as more parallel execution units. Therefore, as noted by the author in reference [3], one can use more sophisticated techniques for polynomial evaluation intended for highly parallel machines. For example, Estrin's method [2] breaks the evaluation down into a balanced binary tree.

We can easily place a lower bound on the latency with which a polynomial can be computed: if we start with x and the coefficients c_i, then by induction, in n serial fma operations, we cannot create a polynomial that is a degree higher than 2^n, and we can only equal 2^n if the term of the highest degree is simply x^{2^n} with unity as its coefficient. For example, in one operation we can reach $c_0 + c_1 x$ or $x + x^2$ but not $x + c_0 x^2$. Our goal is to find an actual scheduling that comes as close as possible to this lower bound.

Simple heuristics based on binary chopping normally give a good evaluation strategy, but it is not always easy to visualize all the possibilities. When the polynomial can be split asymmetrically, or where certain coefficients are special, such as 1 or 0, there are often ways of doing slightly better than one might expect in overall latency or at least in the number of instructions required to attain that latency (and hence in throughput). Besides, doing the scheduling by hand is tedious. We search automatically for the best scheduling using a program that exhaustively examines all essentially different scheduling. One simply

enters a polynomial, and the program returns the best latency and throughput attainable, and it lists the main ways of scheduling the operations to attain this.

Even with various intelligent pruning approaches and heuristics, the search space is large. We restrict it somewhat by considering only `fma` combinations of the form $p_1(x) + x^k p_2(x)$. That is, we do not consider multiplying two polynomials with nontrivial coefficients. Effectively, we allow only solutions that work for arbitrary coefficients, without considering special factorization properties. However, for polynomials where all the coefficients are 1, these results may not be optimal because of the availability of nontrivial factorizations that we have ruled out. For example, we can calculate:

$$1 + x + x^2 + x^3 + x^4 + x^5 + x^6$$

as

$$1 + (1 + (x + x^2))\,(x + (x^2)\,(x^2))$$

which can be scheduled in 15 cycles. However, if the restriction on `fma` operations is observed, then 16 cycles is the best attainable.

The optimization program works in two stages. First, all possible evaluation orders using these restricted `fma` operations are computed. These evaluation orders ignore scheduling, being just "abstract syntax" tree structures indicating the dependencies of subexpressions, with interior nodes representing `fma` operations of the form $p_1(x) + x^k\, p_2(x)$:

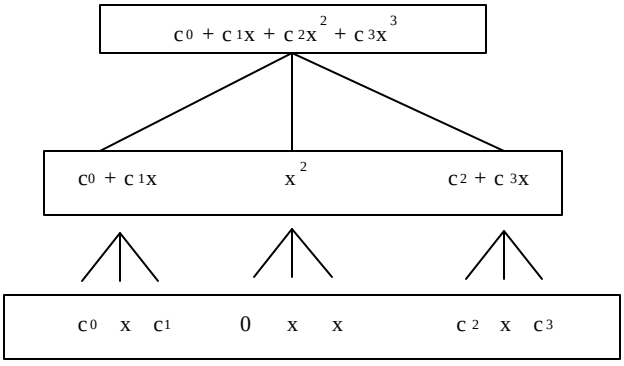


Figure 1: A dependency tree

However, because of the enormous explosion in possibilities, we limit the search to the smallest possible tree depth. This tree depth corresponds to the minimum number of serial operations than can possibly be used to evaluate the expression using the order denoted by that particular tree. Consequently, if the tree depth is d then we cannot possibly do better than $5d$ cycles for that particular tree. Now, assuming that we can in fact do at least as well as $5d + 4$, we are justified in ignoring trees of a depth greater than or equal to $d + 1$, which could not possibly be scheduled in as few cycles. This turns out to be the case for all our examples.

The next stage is to take each tree (in some of the examples below there are as many as 10000 of them) and calculate the optimal scheduling. The optimal scheduling is computed backwards by a fairly naive greedy algorithm, but with a few simple refinements based on stratifying the nodes from the top as well as from the bottom.

The following table gives the evaluation strategy found by the program for the polynomial:

$$x + c_2x^2 + c_3x^3 + \ldots + c_9x^9$$

Table 2 shows that it can be scheduled in 20 cycles, and we have attained the lower bound. However, if the first term were c_1x we would need 21.

Cycle	FMA Unit 1	FMA Unit 2
0	$v_1 = c_2 + x\ c_3$	$v_2 = x\ x$
3	$v_3 = c_6 + x\ c_7$	$v_4 = c_8 + x\ c_9$
4	$v_5 = c_4 + x\ c_5$	
5	$v_6 = x + v_2\ v_1$	$v_7 = v_2\ v_2$
9	$v_8 = v_3 + v_2\ v_4$	
10	$v_9 = v_6 + v_7\ v_5$	$v_{10} = v_2\ v_7$
15	$v_{11} = v_9 + v_{10}\ v_8$	

Table 2: An optimal scheduling

OUTLINE OF ALGORITHMS

We outline each of the seven algorithms discussed here. We concentrate only on the numeric cases and ignore situations such as when the input is out of the range of the functions' domains or non-numeric (NaN for example).

Cbrt

1. *Reduction:* Given x, compute $r = x$ `frcpa`$(x) - 1$.
2. *Approximation*: Compute a polynomial p(r) of the form $p(r)=p_1r + p_2r^2 + \ldots + p_6r^6$ that approximates $(1+r)^{1/3}-1$.
3. *Reconstruction:* Compute the result $T + Tp(r)$ where the T is $(1/$`frcpa`$(x))^{1/3}$. This value T is obtained via a tabulation of $(1/$`frcpa`$(y))^{1/3}$, where $y=1+k/256$, k ranges from 0 to 255 and a tabulation of $2^{-j/3}$, and j ranges from 0 to 2.

Exp

1. *Reduction:* Given x, compute N, the closest integer to the value $x\ (128/\ln(2))$. Then compute $r = (x - N\ P_1) - N\ P_2$. Here P_1+P_2 approximates $\ln(2)/128$ (see previous discussions).

2. *Approximation:* Compute a polynomial $p(r)$ of the form $p(r) = r + p_1 r^2 + \ldots + p_4 r^5$ that approximates $\exp(r) - 1$.

3. *Reconstruction:* Compute the result $T + Tp(r)$ where T is $2^{N/128}$. This value T is obtained as follows. First, N is expressed as $N = 128\,M + 16\,K + J$, where I_1 ranges from 0 to 15, and I_2 ranges from 0 to 7. Clearly $2^{N/128} = 2^M\ 2^{K/8}\ 2^{J/128}$. The first of the three factors can be obtained by scaling the exponent; the remaining two factors are fetched from tables with 8 entries and 16 entries, respectively.

Ln

1. *Reduction:* Given x, compute $r = x \cdot \texttt{frcpa}(x) - 1$.

2. *Approximation:* Compute a polynomial $p(r)$ of the form $p(r) = p_1 r^2 + \ldots + p_5 r^6$ that approximates $\ln(1+r) - r$.

3. *Reconstruction:* Compute the result $T + r + p(r)$ where the T is $\ln(1/\texttt{frcpa}(x))$. This value T is obtained via a tabulation of $\ln(1/\texttt{frcpa}(y))$, where $y = 1 + k/256$, k ranges from 0 to 255, and a calculation of the form $N \ln(2)$.

Sin and Cos

We first consider the case of $\sin(x)$.

1. *Reduction:* Given x, compute N, the closest integer to the value $x \cdot (16/\pi)$. Then compute $r = (x - N \cdot P_1) - N \cdot P_2$. Here $P_1 + P_2$ approximates $\pi/16$ (see previous discussions).

2. *Approximation:* Compute two polynomials: $p(r)$ of the form $r + p_1 r^3 + \ldots + p_4 r^9$ that approximates $\sin(r)$ and $q(r)$ of the form $q_1 r^2 + q_2 r^4 + \ldots + q_4 r^8$ that approximates $\cos(r) - 1$.

3. *Reconstruction:* Return the result as $Cp(r) + (S + Sq(r))$ where C is $\cos(N \cdot \pi/16)$ and S is $\sin(N \cdot \pi/16)$ obtained from a table.

The case of $\cos(x)$ is almost identical. Add 8 to N just after it is first obtained. This works because of the identity $\cos(x) = \sin(x + \pi/2)$.

Tan

1. *Reduction:* Given x, compute N, the closest integer to the value $x \cdot (2/\pi)$. Then compute $r = (x - N \cdot P_1) - N \cdot P_2$. Here $P_1 + P_2$ approximates $\pi/2$ (see previous discussions).

2. *Approximation:* When N is even, compute a polynomial $p(r) = r + r\,t\,(p_0 + p_1 t + \ldots + p_{15} t^{15})$ that approximates $\tan(r)$. When N is odd, compute a polynomial $q(r) = (-r)^{-1} + r(q_0 + q_1 t + \ldots + q_{10} t^{10})$ that approximates $-\cot(r)$. The term t is r^2. We emphasize the fact that parallelism is fully utilized.

3. *Reconstruction:* If N is even, return p. If N is odd, return q.

Atan

1. *Reduction:* No reduction is needed.

2. *Approximation:* If $|x|$ is less than 1, compute a polynomial $p(x) = x + x^3(p_0 + p_1 y + \ldots + p_{22} y^{22})$ that approximates $\text{atan}(x)$, y is x^2. If $|x| > 1$, compute several quantities, fully utilizing parallelism. First, compute $q(x) = q_0 + q_1 y + \ldots + q_{22} y^{22}$, $y = x^2$, that approximates $x^{45} \text{atan}(1/x)$. Second, compute c^{45} where $c = \texttt{frcpa}(x)$. Third, compute another polynomial $r(\beta) = 1 + r_1 \beta + \ldots + r_{10} \beta^{10}$, where β is the quantity $x \cdot \texttt{frcpa}(x) - 1$ and $r(\beta)$ approximates the value $(1 - \beta)^{-45}$.

3. *Reconstruction:* If $|x|$ is less than 1, return $p(x)$. Otherwise, return $\text{sign}(x)\pi/2 - c^{45} r(\beta) q(x)$. This is due to the identity $\text{atan}(x) = \text{sign}(x)\pi/2 - \text{atan}(1/x)$.

SPEED AND ACCURACY

These new double-precision elementary functions are designed to be both fast and accurate. We present the speed of the functions in terms of latency for arguments that fall through the implementation in a path that is deemed most likely. As far as accuracy is concerned, we report the largest observed error after extensive testing in terms of units of last place (ulps). This error measure is standard in this field. Let f be the mathematical function to be implemented and F be the actual implementation in double precision. When $2^L < |f(x)| \le 2^{L+1}$, the error in ulps is defined as $|f(x) - F(x)| / (2^{L-52})$. Note that the smallest worst-case error that one can possibly attain is *0.5* ulps. Table 3 tabulates the latency and maximum error observed.

Function	Latency (cycles)	Max. Error (ulps)
cbrt	60	0.51
exp	60	0.51
ln	52	0.53
sin	70	0.51
cos	70	0.51
tan	72	0.51
atan	66	0.51

Table 3: Speed and accuracy of functions

CONCLUSIONS

We have shown how certain key features of the IA-64 architecture can be exploited to design transcendental functions featuring an excellent combination of speed and accuracy. All of these functions performed over twice as fast as the ones based on the simple conversion of a library tailored for double-extended precision. In one instance, the ln function described here contributed to a two point increment of SpecFp benchmark run under simulation.

The features of the IA-64 architecture that are exploited include parallelism and the fused multiply add as well as less obvious features such as the reciprocal approximation instruction. When abundant resources for parallelism are available, it is not always easy to visualize how to take full advantage of them. We have searched for optimal instruction schedules. Although our search method is sufficient to handle the situations we have faced so far, more sophisticated techniques are needed to handle more complex situations. First, polynomials of a higher degree may be needed in more advanced algorithms. Second, more general expressions that can be considered as multivariate polynomials are also anticipated. Finally, our current method does not handle the full generality of microarchitectural constraints, which also vary in future implementations on the IA-64 roadmap. We believe this optimal scheduling problem to be important not only because it yields high-performance implementation, but also because it may offer a quantitative analysis on the balance of microarchitectural parameters. Currently we are considering an integer programming framework to tackle this problem. We welcome other suggestions as well.

REFERENCES

[1] Cody Jr., William J. and Waite, William, *Software Manual for the Elementary Functions*, Prentice Hall, 1980.

[2] Knuth, D.E., *The Art of Computer Programming vol. 2: Seminumerical Algorithms*, Addison-Welsey, 1969.

[3] Muller, J. M., *Elementary functions: algorithms and implementation*, Birkhaüser, 1997.

[4] Payne, M., "An Argument Reduction Scheme on the DEC VAX," *Signum Newsletter*, January 1983.

[5] Powell, M.J.D., *Approximation Theory and Methods*, Cambridge University Press, 1981.

[6] Story, S. and Tang, P.T.P., "New algorithms for improved transcendental functions on IA-64," in *Proceedings of 14th IEEE symposium on computer arithmetic*, IEEE Computer Society Press, 1999.

[7] Smith, Roger A., "A Continued-Fraction Analysis of Trigonometric Argument Reduction," *IEEE Transactions on Computers*, pp. 1348-1351, Vol. 44, No. 11, November 1995.

[8] Tang, P.T.P., "Table-driven implementation of the exponential function in IEEE floating-point arithmetic," *ACM Transactions on Mathematical Software*, vol. 15, pp. 144-157, 1989.

AUTHORS' BIOGRAPHIES

John Harrison has been with Intel for just over one year. He obtained his Ph.D. degree from Cambridge University in England and is a specialist in formal validation and theorem proving. His e-mail is johnh@ichips.intel.com.

Ted Kubaska is a senior software engineer with Intel Corporation in Hillsboro, Oregon. He has a M.S. degree in physics from the University of Maine at Orono and a M.S. degree in computer science from the Oregon Graduate Institute. He works in the MSL Numerics Group where he implements and tests floating-point algorithms. His e-mail is Theodore.E.Kubaska@intel.com.

Shane Story has worked on numerical and floating-point related issues since he began working for Intel eight years ago. His e-mail is Shane.Story@intel.com.

Ping Tak Peter Tang (his friends call him Peter) joined Intel very recently as an applied mathematician working in the Computational Software Lab of MSL. Peter received his Ph.D. degree in mathematics from the University of California at Berkeley. His interest is in floating-point issues as well as fast and accurate numerical computation methods. Peter has consulted for Intel in the past on such issues as the design of the transcendental algorithms on the Pentium, and he contributed a software solution to the Pentium division problem. His e-mail is Peter.Tang@intel.com.

IA-64 Floating-Point Operations and the IEEE Standard for Binary Floating-Point Arithmetic

Marius Cornea-Hasegan, Microprocessor Products Group, Intel Corporation
Bob Norin, Microprocessor Products Group, Intel Corporation

Index words: IA-64 architecture, floating-point, IEEE Standard 754-1985

ABSTRACT

This paper examines the implementation of floating-point operations in the IA-64 architecture from the perspective of the IEEE Standard for Binary Floating-Point Arithmetic [1]. The floating-point data formats, operations, and special values are compared with the mandatory or recommended ones from the IEEE Standard, showing the potential gains in performance that result from specific choices.

Two subsections are dedicated to the floating-point divide, remainder, and square root operations, which are implemented in software. It is shown how IEEE compliance was achieved using new IA-64 features such as fused multiply-add operations, predication, and multiple status fields for IEEE status flags. Derived integer operations (the integer divide and remainder) are also illustrated.

IA-64 floating-point exceptions and traps are described, including the Software Assistance faults and traps that can lead to further IEEE-defined exceptions. The software extensions to the hardware needed to comply with the IEEE Standard's recommendations in handling floating-point exceptions are specified. The special case of the Single Instruction Multiple Data (SIMD) instructions is described. Finally, a subsection is dedicated to speculation, a new feature in IA processors.

INTRODUCTION

The IA-64 floating-point architecture was designed with three objectives in mind. First, it was meant to allow high-performance computations. This was achieved through a number of architectural features. Pipelined floating-point units allow several operations to take place in parallel. Special instructions were added, such as fused floating-point multiply-add, or SIMD instructions, which allow the processing of two subsets of floating-point operands in parallel. Predication allows skipping operations without taking a branch. Speculation allows speculative execution chains whose results are committed only if needed. In addition, a large floating-point register file (including a rotating subset) reduces the number of save/restore operations involving memory. The rotating subset of the floating-point register file enables software pipelining of loops, leading to significant gains in performance.

Second, the architecture aims to provide high floating-point accuracy. For this, several floating-point data types were provided, and instructions new to the Intel architecture, such as the fused floating-point multiply-add, were introduced.

Third, compliance with the IEEE Standard for Binary Floating-Point Arithmetic was sought. The environment that a numeric software programmer sees complies with the IEEE Standard and most of its recommendations as a combination of hardware and software, as explained further in this paper.

Floating-Point Numbers

Floating-point numbers are represented as a concatenation of a sign bit, an M-bit exponent field, and an N-bit significand field. In some floating-point formats, the most significant bit (integer bit) of the significand is not represented. Its assumed value is 1, except for denormal numbers, whose most significant bit of the significand is 0. Mathematically

$$f = \sigma \cdot s \cdot 2^e$$

where $\sigma = \pm 1$, $s \in [1,2)$, $s = 1 + k / 2^{N-1}$, $k \in \{0, 1, 2, \ldots, 2^{N-1}-1\}$, $e \in [e_{min}, e_{max}] \cap \mathbf{Z}$ ($\mathbf{Z}$ is the set of integers), $e_{min} = -2^{M-1} + 2$, and $e_{max} = 2^{M-1} - 1$.

The IA-64 architecture provides 128 82-bit floating-point registers that can hold floating-point values in various formats, and which can be addressed in any order.

Floating-point numbers can also be stored into or loaded from memory.

IA-64 FORMATS, CONTROL, AND STATUS

Formats

Three floating-point formats described in the IEEE Standard are implemented as required: single precision (M=8, N=24), double precision (M=11, N=53), and double-extended precision (M=15, N=64). These are the formats usually accessible to a high-level language numeric programmer. The architecture provides for several more formats, listed in Table 1, that can be used by compilers or assembly code writers, some of which employ the 17-bit exponent range and 64-bit significands allowed by the floating-point register format.

Format	Format Parameters
Single precision	M=8, N=24
Double precision	M=11, N=53
Double-extended precision	M=15, N=64
Pair of single precision floating-point numbers	M=8, N=24
IA-32 register stack single precision	M=15, N=24
IA-32 register stack double precision	M=15, N=53
IA-32 double-extended precision	M=15, N=64
Full register file single precision	M=17, N=24
Full register file double precision	M=17, N=53
Full register file double-extended precision	M=17, N=64

Table 1: IA-64 floating-point formats

The floating-point format used in a given computation is determined by the floating-point instruction (some instructions have a precision control completer *pc* specifying a static precision) or by the precision control field (pc), and by the widest-range exponent (wre) bit in the Floating-Point Status Register (FPSR). In memory, floating-point numbers can only be stored in single precision, double precision, double-extended precision, and register file format ('spilled' as a 128-bit entity, containing the value of the floating-point register in the lower 82 bits).

Rounding

The four IEEE rounding modes are supported: rounding to nearest, rounding to negative infinity, rounding to positive infinity, and rounding to zero. Some instructions have the option of using a static rounding mode. For example, fcvt.fx.trunc performs conversion of a floating-point number to integer using rounding to zero.

Some of the basic operations specified by the IEEE Standard (divide, remainder, and square root) as well as other derived operations are implemented using sequences of add, subtract, multiply, or fused multiply-add and multiply-subtract operations.

In order to determine whether a given computation yields the correctly rounded result in any rounding mode, as specified by the standard, the error that occurs due to rounding has to be evaluated. Two measures are commonly used for this purpose. The first is the error of an approximation with respect to the exact result, expressed in fractions of an ulp, or unit in the last place. Let $\mathbf{F}_N$ be the set of floating-point numbers with N-bit significands and unlimited exponent range. For the floating-point number $f = \sigma \cdot s \cdot 2^e \in \mathbf{F}_N$, one ulp has the magnitude

$$1 \text{ ulp} = 2^{e-N+1.}$$

An alternative is to use the relative error. If the real number x is approximated by the floating-point number a, then the relative error ε is determined by

$$a = x \cdot (1 + \varepsilon)$$

The Floating-Point Status Register

Several characteristics of the floating-point computations are determined by the contents of the 64-bit FPSR.

A set of six trap mask bits (bits 0 through 5) control enabling or disabling the five IEEE traps (invalid operation, divide-by-zero, overflow, underflow, and inexact result) and the IA-defined denormal trap [2]. In addition, four 13-bit subsets of control and status bits are provided: status fields sf0, sf1, sf2, and sf3. Multiple status fields allow different computations to be performed simultaneously with different precisions and/or rounding modes. Status field 0 is the user status field, specifying rounding-to-nearest and 64-bit precision by default. Status field 1 is reserved by software conventions for special operations, such as divide and square root. It uses rounding-to-nearest, the 64-bit precision, and the widest-range exponent (17 bits). Status fields 2 and 3 can be used in speculative

operations, or for implementing special numeric algorithms, e.g., the transcendental functions.

Each status field contains a 2-bit rounding mode control field (00 for rounding to nearest, 01 to negative infinity, 10 to positive infinity, and 11 toward zero), a 2-bit precision control field (00 for 24 bits, 10 for 53 bits, and 11 for 64 bits), a widest-range exponent bit (use the 17-bit exponent if wre = 1), a flush-to-zero bit (causes flushing to zero of tiny results if ftz = 1), and a traps disabled bit (overrides the individual trap masks and disables all traps if td = 1, except for status field 0, where this bit is reserved). Each status field also contains status flags for the five IEEE exceptions and for the denormal exception.

The register file floating-point format uses a 17-bit exponent range, which has two more bits than the double-extended precision format, for at least three reasons. The first is related to the implementation in software of the divide and square root operations in the IA-64 architecture. Short sequences of assembly language instructions carry out these computations iteratively. If the exponent range of the intermediate computation steps is equal to that of the final result, then some of the intermediate steps might overflow, underflow, or lose precision, preventing the final result from being IEEE correct. Software Assistance (SWA) will be necessary in these cases to generate the correct results, as explained in [4]. The two (or more) extra bits in the exponent range (17 versus 15 or less) prevent the SWA requests from occurring. The second reason for having a 17-bit exponent range is that it allows the common computation of $x^2 + y^2$ to be performed without overflow or underflow, even for the largest or smallest double-extended precision numbers. Third, the 17-bit exponent range is necessary in order to be able to represent the product of all double-extended denormal numbers.

Special Values

The various floating-point formats support the IEEE mandated representations for denormals, zero, infinities, quiet NaNs (QNaNs), and signaling NaNs (SNaNs). In addition, the formats that have an explicit integer bit in the significand can also hold other types of values. These formats are double-extended, with 15-bit exponents biased by 16383 (0x3fff), and all the register file formats, with 17-bit exponents biased by 65535 (0xffff). The exponents of these additional types of values are specified below for the register file format:

> *unnormalized numbers*: non-zero significand beginning with 0 and exponent from 0 to 0x1fffe, or pseudo-zeroes with a significand of 0, and exponent from 0x1 to 0x1fffe
>
> *pseudo-NaNs*: non-zero significand and exponent of 0x1ffff (unsupported by the architecture); the pseudo-QNaNs have the second most significant bit of the significand equal to 1; this bit is 0 for pseudo-SNaNs
>
> *pseudo-infinities*: significand of zero and exponent of 0x1ffff (unsupported by the architecture)

Note that one of the pseudo-zero values, encoded on 82 bits as 0x1fffe0000000000000000, is denoted as NaTVal ('not a value') and is generated by unsuccessful speculative load from memory operations (e.g. a speculative load, in the presence of a deferred floating-point exception). It is then propagated through the speculative chain to indicate in the end that no useful result is available.

Two special categories that overload other floating-point numbers in register file format are the SIMD floating-point pairs, and the canonical non-zero integers. Both have an exponent of 0x1003e (unbiased 63). The value of the canonical non-zero integers is equal to that of the unnormal or normal floating-point numbers that they overlap with. The exponent of 63 moves the binary point beyond the least significant bit, the resulting value being the integer stored in the significand. The SIMD floating-point numbers consist of two single-precision floating-point values encoded in the two halves of the 64-bit significand of a floating-point register, with the biased exponent set to 0x1003e. For example, the 82-bit value of 0x1003e 3f800000 3f800000 represents the pair (+1.0, +1.0). Note that all the arithmetic scalar floating-point instructions have SIMD counterparts that operate on two single-precision floating-point values in parallel.

IA-64 FLOATING-POINT OPERATIONS

All the floating-point operations mandated or recommended by the IEEE Standard are or can be implemented in IA-64 [2]. Note that most IA-64 instructions [2] are predicated by a 1-bit predicate (*qp*) from the 64-bit predicate register (predicate p0 is fixed, containing always the logical value 1). For example, the fused multiply-add operation is

> (*qp*) fma.*pc.sf f1 = f3, f4, f2*

The fma instruction is executed if *qp* = 1; otherwise, it is skipped. Two instruction completers select the precision control (*pc*) and the status field (*sf*) to be used. When the qualifying predicate is not represented, it is either not necessary, or it is assumed to be p0. When *qp* = 1, fma calculates $f3 \cdot f4 + f2$, where '*pc*' can be 's', 'd', or none. If the instruction completer '*pc*' is 's', fma.s.*sf* generates a result with a 24-bit significand. Similarly, fma.d.*sf*

generates a result with a 53-bit significand. The exponent in the two cases is in the 8-bit or 11-bit range respectively if *sf*.wre = 0, and in the 17-bit range if *sf*.wre = 1. If *'pc'* is none, the precision of the computation fma.*sf f1 = f3, f4, f2* is specified by the pc field of the status field being used, *sf*.pc. The exponent size is 15 bits if *sf*.wre = 0, and 17 bits if *sf*.wre = 1.

Addition and multiplication are implemented as pseudo-ops of the floating-point multiply-add operation. The pseudo-op for addition is fadd.*pc.sf f1 = f3, f2* obtained by replacing *f4* with register f1 that contains +1.0. The pseudo-op for multiplication is fmpy.*pc.sf f1 = f3, f4*, obtained by replacing *f2* with f0 that contains +0.0.

The reason for having a fused multiply-add operation is that it allows computation of $a \cdot b + c$ with only one rounding error. Assuming rounding to nearest, fma computes

$$(a \cdot b + c)_{rn} = (a \cdot b + c) \cdot (1 + \varepsilon)$$

where $|\varepsilon| < 2^{-N}$, and N is the number of bits in the significand. The relative error above (ε) is smaller in general than that obtained with pure add and multiply operations:

$$((a \cdot b)_{rn} + c)_{rn} = (a \cdot b\,(1 + \varepsilon_1) + c) \cdot (1 + \varepsilon_2)$$

where $|\varepsilon_1| < 2^{-N}$ and $|\varepsilon_2| < 2^{-N}$.

The benefit that arises from this property is that it enables the implementation of a whole new category of numerical algorithms, relying on the possibility of performing this combined operation with only one rounding error (see the subsections on divide and square root below).

Subtraction (fsub.*pc.sf f1 = f3, f2*) is implemented as a pseudo-op of the floating-point multiply-subtract, fms.*pc.sf f1 = f3, f4, f2* (which calculates $f3 \cdot f4 - f2$) where *f4* is replaced by f1. In addition to fma and fms, a similar operation is available for the floating-point negative multiply-add operation, fnma.*pc.sf f1 = f3, f4, f2*, which calculates $-f3 \cdot f4 + f2$.

A deviation from one of the IEEE Standard's recommendations is to allow higher precision operands to lead to lower precision results. However, this is a useful feature when implementing the divide, remainder, and square root operations in software.

For parallel computations, counterparts of fma, fms, and fnma are provided. For example, fpma.*pc.sf f1 = f3, f4, f2* calculates $f3 \cdot f4 + f2$. A pair of ones (1.0, 1.0) has to be loaded explicitly in a floating-point register to emulate the SIMD floating-point add.

Divide, square root, and remainder operations are not available directly in hardware. Instead, they have to be implemented in software as sequences of instructions corresponding to iterative algorithms (described below).

Rounding of a floating-point number to a 64-bit signed integer in floating-point format is achieved by the fcvt.fx.*sf f1 = f2* instruction followed by fcvt.xf *f2 = f1*. For 64-bit unsigned integers, the similar instructions are fcvt.fxu.*sf f1 = f2* and fcvt.xuf.*pc.sf f2 = f1*. Two variations of the instructions that convert floating-point numbers to integer use the rounding-to-zero mode regardless of the rounding control bits used in the FPSR status field (fcvt.fx.trunc.*sf f1 = f2* and fcvt.fxu.trunc.*sf f1 = f2*). They are useful in implementing integer divide and remainder operations using floating-point instructions. For example, the following instructions convert a single precision floating-point number from memory (whose address is in the general register r30) to a 64-bit signed integer in r8:

```
ldfs f6=[r30];; // load single precision fp number
fcvt.fx.trunc.s0 f7=f6;; // convert to integer
getf.sig r8=f7;;
```

(Note that stop bits (;;) delimit the instruction groups.) The biased exponent of the value in f7 is set by fcvt.fx.trunc.s0 to 0x1003e (unbiased 63) and the significand to the signed integer that is the result of the conversion. (If the conversion is invalid, the significand is set to the value of Integer Indefinite, which is -2^{63}.) Since rounding to zero is used by fcvt.fx.trunc, specifying the status field only tells which status flags to set if an invalid operation, denormal, or inexact result exception occurs (Exceptions and Traps are covered later in the paper.) For the conversion from a floating-point number to a 64-bit unsigned integer, fcvt.fx.trunc above has to be replaced by fcvt.fxu.trunc.

The opposite conversion, from a 64-bit signed integer in r32 to a register-file format floating-point number in f7, is performed by

```
setf.sig f6 = r32;; //sign=0 exp=0x1003e signif.=r32
fcvt.xf f7 = f6;; // sign=sign(r32); no fp exceptions
```

where the result is an integer-valued normal floating-point number. To convert further, for example to a single precision floating-point number, one more instruction is needed

```
fma.s.s0 f8=f7,f1,f0;;
```

where the single precision format is specified statically, and status field s0 is assumed to have wre = 0.

For 64-bit unsigned integers, the similar conversion is

setf.sig f6 = r32;; // sign=0 exp=0x1003e signif.=r32

fcvt.xuf.s0 f7 = f6

where fcvt.xuf.*pc*.*sf* f7 = f6 is actually a pseudo-op for fma.*pc*.*sf* f7 = f6, f1, f0, and a synonym of fnorm.*pc*.*sf* f7 = f6 (it is assumed that status field s0 has pc = 0x3). The result is thus a normalized integer-valued floating-point number. This is important to know, since floating-point operations on unnormalized numbers lead to Software Assistance faults (as explained further in the paper), thereby slowing down performance unnecessarily.

Conversions between the different floating-point formats are achieved using floating-point load, store, or other operations. For example, the following sequence converts a single precision value from memory to double precision format, also in memory (r29 contains the address of the single precision source, and r30 that of the double precision destination):

ldfs f6 = [r29];;

fma.d.s0 f7=f6,f1,f0;;

stfd [r30] = f7

This conversion could trigger the invalid exception (for a signaling NaN operand) or the denormal operand exception. These can happen on the fma instruction, but the conversion will be correct numerically even without this instruction, as all the single precision values can be represented in the double precision format.

The opposite conversion is shown below (it is assumed that status field s0 has wre = 0):

ldfd f6=[r29];;

fma.s.s0 f7=f6,f1,f0;;

stfs [r30]=f7;;

The role of the fma.s.s0 is to trigger possible invalid, denormal, underflow, overflow, or inexact exceptions on this conversion.

Other conversions between floating-point and integer formats can be achieved with short sequences of instructions. For example, the following sequence converts a single precision floating-point value in memory to a 32-bit signed integer (correct only if the result fits on 32 bits):

ldfs f6 = [r30];; // load f6 with fp value from memory

fcvt.fx.trunc.s0 f7=f6;; // convert to signed integer

getf.sig r29 = f7;; // move the 64-bit integer to r29

st4 [r28] = r29;; // store as 32-bit integer in memory

The opposite conversion, from a 32-bit integer in memory to a single precision floating-point number in memory, is performed by

ld4 r29 = [r30];; // load r29 with 32-bit int from mem

sxt4 r28=r29;; // sign-extend

setf.sig f6 = r28;; // 32-bit integer in f6; exp=0x1003e

fcvt.xf f7=f6;; // convert to normal floating-point

fma.s.s0 f8 = f7,f1,f0;; // trigger I exceptions if any

stfs [r27] = f8;; // store single prec. value in memory.

Floating-point compare operations can be performed directly between numbers in floating-point register file format, using the fcmp instruction. For other memory formats, a conversion to register format is required prior to applying the floating-point compare instruction. From the 26 functionally distinct relations specified by the IEEE Standard, only the six mandatory ones are implemented (four directly, and two as pseudo-ops):

fcmp.eq.*sf p1, p2 = f2, f3* (test for '=')

fcmp.lt.*sf p1, p2 = f2, f3* (test for '<')

fcmp.le.*sf p1, p2 = f2, f3* (test for '<=')

fcmp.gt.*sf p1, p2 = f2, f3* (test for '>')

fcmp.ge.*sf p1, p2 = f2, f3* (test for '>=')

fcmp.unord.*sf p1, p2 = f2, f3* (test for '?')

The result of a compare operation is written to two 1-bit predicates in the 64-bit predicate register. Predicate *p1* shows the result of the comparison, while *p2* is its opposite. An exception is the case when at least one input value is NaTVal, when *p1* = *p2* = 0. A variant of the fcmp instruction is called 'unconditional' (with respect to the qualifying predicate). The difference is that if *qp* = 0, the unconditional compare

(*qp*) fcmp.eq.unc.*sf p1, p2 = f2, f3*

clears both output predicates, while

(*qp*) fcmp.eq.*sf p1, p2 = f2, f3*

leaves them unchanged.

Six more compare relations are implemented, as pseudo-ops of the above, to test for the opposite situations (neq, nlt, nle, ngt, nge, and ord). The remaining 14 comparison relations specified by the IEEE Standard can be performed based on the above.

A special type of compare instruction is fclass.*fcrel.fctype p1,p2=f2,fclass9*, that allows classification of the contents of *f2* according to the class specifier *fclass9*. The *fcrel* instruction completer can be

'm' (if *f2* has to agree with the pattern specified by *fclass9*), or 'nm' (*f2* has to disagree). The *fctype* completer can be *none* or 'unc' (as for fcmp). *fclass9* can specify one of {NaTVal, QNaN, SNaN} OR none, one or both of {positive, negative} AND none, one or several of {zero, unnormal, normal, infinity} (nine bits correspond to the nine classes that can be selected, with the restrictions specified on the possible combinations).

IA-64 FLOATING-POINT OPERATIONS DEFERRED TO SOFTWARE

A number of floating-point operations defined by the IEEE Standard are deferred to software by the IA-64 architecture in all its implementations:

- floating-point divide (integer divide, which is based on the floating-point divide operation, is also deferred to software)
- floating-point square root
- floating-point remainder (integer remainder, based on the floating-point divide operation, is also deferred to software)
- binary to decimal and decimal to binary conversions
- floating-point to integer-valued floating-point conversion
- correct wrapping of the exponent for single, double, and double-extended precision results of floating-point operations that overflow or underflow, as described by the IEEE Standard

In addition, the IA-64 architecture allows virtually any floating-point operation to be deferred to software through the mechanism of Software Assistance (SWA) requests, which are treated as floating-point exceptions. Software Assistance is discussed in detail in the sections describing the divide operation, the square root operation, and the exceptions and traps.

IA-64 FLOATING-POINT DIVIDE AND REMAINDER

The floating-point divide algorithms for the IA-64 architecture are based on the Newton-Raphson iterative method and on polynomial evaluation. If a/b needs to be computed and the Newton-Raphson method is used, a number of iterations first calculate an approximation of 1/b, using the function

$$f(y) = b - 1/y$$

The iteration step is

$$e_n = (1 - b \cdot y_n)_{rn} \approx 0$$

$$y_{n+1} = (y_n + e_n \cdot y_n)_{rn} \approx 1/b$$

where the subscript *rn* denotes the IEEE rounding to the nearest mode.

Once a sufficiently good approximation y of 1/b is determined, $q = a \cdot y$ approximates a/b. In some cases, this might need further refinement, which requires only a few more computational steps.

In order to show that the final result generated by the floating-point divide algorithm represents the correctly rounded value of the infinitely precise result a/b in any rounding mode, it was proved (by methods described in [3] and [4]) that the exact value of a/b and the final result q* of the algorithm before rounding belong to the same interval of width 1/2 ulp, adjacent to a floating-point number. Then

$$(a/b)_{rnd} = (q^*)_{rnd}$$

where *rnd* is any IEEE rounding mode.

The algorithms proposed for floating-point divide (as well as for square root) are designed, as seen from the Newton-Raphson iteration step shown above, based on the availability of the floating-point multiply-add operation, fma, that performs both the multiply and add operations with only one rounding error.

Two variants of floating-point divide algorithms are provided for single precision, double precision, double-extended and full register file format precision, and SIMD single precision. One achieves minimum latency, and one maximizes throughput.

The minimum latency variant minimizes the execution time to complete one operation. The maximum throughput variant performs the operation using a minimum number of floating-point instructions. This variant allows the best utilization of the parallel resources of the IA-64, yielding the minimum time per operation when performing the operation on multiple sets of operands.

Double Precision Floating-Point Divide Algorithm

The double precision floating-point divide algorithm that minimizes latency was chosen to illustrate the implementation of the mathematical algorithm in IA-64 assembly language. The input values are the double precision operands a and b, and the output is a/b.

All the computational steps are performed in full register file double-extended precision, except for steps (11) and (12), which are performed in full register file double precision, and step (13), performed in double-precision.

The approximate values are shown on the right-hand side.

(1) $y_0 = 1/b \cdot (1 + \varepsilon_0)$, $|\varepsilon_0| \le 2^{-m}$, m=8.886

(2) $q_0 = (a \cdot y_0)_m = a/b \cdot (1 + \varepsilon_0)$

(3) $e_0 = (1 - b \cdot y_0)_{rn} \approx -\varepsilon_0$

(4) $y_1 = (y_0 + e_0 \cdot y_0)_{rn} \approx 1/b \cdot (1 - \varepsilon_0^2)$

(5) $q_1 = (q_0 + e_0 \cdot q_0)_{rn} \approx a/b \cdot (1 - \varepsilon_0^2)$

(6) $e_1 = (e_0^2)_{rn} \approx \varepsilon_0^2$

(7) $y_2 = (y_1 + e_1 \cdot y_1)_{rn} \approx 1/b \cdot (1 - \varepsilon_0^4)$

(8) $q_2 = (q_1 + e_1 \cdot q_1)_{rn} \approx a/b \cdot (1 - \varepsilon_0^4)$

(9) $e_2 = (e_1^2)_{rn} \approx \varepsilon_0^4$

(10) $y_3 = (y_2 + e_2 \cdot y_2)_{rn} \approx 1/b \cdot (1 - \varepsilon_0^8)$

(11) $q_3 = (q_2 + e_2 \cdot q_2)_{rn} \approx a/b \cdot (1 - \varepsilon_0^8)$

(12) $r_0 = (a - b \cdot q_3)_{rn} \approx a \cdot \varepsilon_0^8$

(13) $q_4 = (q_3 + r_0 \cdot y_3)_{rnd} \approx a/b \cdot (1 - \varepsilon_0^{16})$

The first step is a table lookup performed by frcpa, which gives an initial approximation y_0 of 1/b, with known relative error determined by m = 8.886. Steps (3) and (4), (6) and (7), and (9) and (10) represent three iterations that generate increasingly better approximations of 1/b in y_1, y_2, and y_3. Note that step (2) above is exact: y0 has 11 bits (read from a table), and a has 53 bits in the significand, and thus the result of the multiplication has at most 64 bits that fit in the significand. Steps (5), (8), and (11) calculate three increasingly better approximations q_1, q_2 and q_3 of a/b. Evaluating their relative errors and applying other theoretical properties [4], it was shown that $q_4 = (a/b)_{rnd}$ in any IEEE rounding mode *rnd*, and that the status flag settings and exception behavior are IEEE compliant. Assuming that the latency of all floating-point operations is the same, the algorithm takes seven fma latencies: steps (2) and (3) can be executed in parallel, as can steps (4), (5), (6); then (7), (8), (9) and also (10) and (11).

The implementation of this algorithm in assembly language is shown next.

```
(1)  frcpa.s0 f8,p6=f6,f7;;  // y0=1/b in f8
(2)  (p6) fma.s1 f9=f6,f8,f0  // q0=a*y0 in f9
(3)  (p6) fnma.s1 f10=f7,f8,f1;;  // e0=1-b*y0 in f10
(4)  (p6) fma.s1 f8=f10,f8,f8  // y1=y0+e0*y0 in f8
(5)  (p6) fma.s1 f9=f10,f9,f9  // q1=q0+e0*q0 in f9
(6)  (p6) fma.s1 f11=f10,f10,f0;; // e1=e0*e0 in f11
(7)  (p6) fma.s1 f8=f11,f8,f8  // y2=y1+e1*y1 in f8
(8)  (p6) fma.s1 f9=f11,f9,f9  // q2=q1+e1*q1 in f9
(9)  (p6) fma.s1 f10=f11,f11,f0;; // e2=e1*e1 in f10
(10) (p6) fma.s1 f8=f10,f8,f8  // y3=y2+e2*y2 in f8
(11) (p6) fma.d.s1 f9=f10,f9,f9;;//q3=q2+e2*q2in f9
(12) (p6) fnma.d.s1 f6=f7,f9,f6;;  // r0=a-b*q3 in f6
(13) (p6) fma.d.s0 f8=f6,f8,f9;;// q4=q3+r0*y3 in f8
```

Note that the output predicate p6 of instruction (1) (frcpa) predicates all the subsequent instructions. Also, the output register of frcpa (f8) is the same as the output register of the last operation (in step (13)). If the frcpa instruction encounters an exceptional situation such as unmasked division by 0, and an exception handler provides the result of the divide, p6 is cleared and no other instruction from the sequence is executed. The result is still provided where it is expected. Another observation is that the first and last instructions in the sequence use the user status field (sf0), which will reflect exceptions that might occur, while the intermediate computations use status field 1 (sf1, with wre = 1). This implementation behaves like an atomic double precision divide, as prescribed by the IEEE Standard. It sets correctly all the IEEE status flags (plus the denormal status flag), and it signals correctly all the possible floating-point exceptions if unmasked (invalid operation, denormal operand, divide by zero, overflow, underflow, or inexact result).

Floating-Point Remainder

The floating-point divide algorithms are the basis for the implementation of floating-point remainder operations. Their correctness is a direct consequence of the correctness of the floating-point divide algorithms. The remainder is calculated as $r = a - n \cdot b$, where n is the integer closest to the infinitely precise a/b. The problem is that n might require more bits to represent than available in the significand for the format of a and b. The solution is to implement an iterative algorithm, as explained in [5] for FPREM1 (all iterations but the last are called 'incomplete'). The implementation (not shown here) is quite straightforward. The rounding to zero mode for divide can be set in status field sf2 (otherwise identical to the user status field sf0). Status field sf0 will only be used by the first frcpa (which may signal the invalid, divide by zero, or denormal exceptions) and by the last fnma (computing the remainder). The last fnma may also signal the underflow exception.

Software Assistance Conditions for Scalar Floating-Point Divide

The main issue identified in the process of proving the IEEE correctness of the divide algorithms [4] is that there are cases of input operands for *a/b* that can cause overflow, underflow, or loss of precision of an intermediate result. Such operands might prevent the sequence from generating correct results, and they require alternate algorithms implemented in software in order to avoid this. These special situations are identified by the following conditions that define the necessity for IA-64 Architecturally Mandated Software Assistance for the scalar floating-point divide operations:

(a) $e_b \le e_{min} - 2$ (y_i might become huge)

(b) $e_b \ge e_{max} - 2$ (y_i might become tiny)

(c) $e_a - e_b \ge e_{max}$ (q_i might become huge)

(d) $e_a - e_b \le e_{min} + 1$ (q_i might become tiny)

(e) $e_a \le e_{min} + N - 1$ (r_i might lose precision)

where e_a is the (unbiased) exponent of a; e_b is the exponent of b; e_{min} is the minimum value of the exponent in the given format; e_{max} is its maximum possible value; and N is the number of bits in the significand. When any of these conditions is met, frcpa issues a Software Assistance (SWA) request in the form of a floating-point exception instead of providing a reciprocal approximation for *1/b*, and clears its output predicate. An SWA handler provides the result of the floating-point divide, and the rest of the iterative sequence for calculating *a/b* is predicated off. The five conditions above can be represented to show how the two-dimensional space containing pairs (e_a, e_b) is partitioned into regions (Figure 4 of [4]). Alternate software algorithms had to be devised to compute the IEEE correct quotients for pairs of numbers whose exponents fall in regions satisfying any of the five conditions above. Note though that due to the extended internal exponent range (17 bits), the single precision, double precision, and double-extended precision calculations will never require architecturally mandated software assistance. This type of software assistance might be required only for floating-point register file format computations with floating-point numbers having 17-bit exponents.

When an architecturally mandated software assistance request occurs for the divide operation, the result is provided by the IA-64 Floating-Point Emulation Library, which has the role of an SWA handler, as described further.

The parallel reciprocal approximation instruction, fprcpa, does not signal any SWA requests. When any of the five conditions shown above is met, fprcpa merely clears its output predicate, in which case the result of the parallel divide operation has to be computed by alternate algorithms (typically by unpacking the parallel operands, performing two single precision divide operations, and packing the results into a SIMD result).

IA-64 FLOATING-POINT SQUARE ROOT

The IA-64 floating-point square root algorithms are also based on Newton-Raphson or similar iterative computations. If $\sqrt{a}$ needs to be computed and the Newton-Raphson method is used, a number of Newton-Raphson iterations first calculate an approximation of $1/\sqrt{a}$, using the function

$$f(y) = 1/y^2 - a$$

The general iteration step is

$$e_n = (1/2 - 1/2 \cdot a \cdot y_n^2)_{rn}$$

$$y_{n+1} = (y_n + e_n \cdot y_n)_{rn}$$

where the subscript *rn* denotes the IEEE rounding to the nearest mode. The first computation above is rearranged in the real algorithm in order to take advantage of the fma instruction capability.

Once a sufficiently good approximation y of $1/\sqrt{a}$ is determined, $S = a \cdot y$ approximates $\sqrt{a}$. In certain cases, this too might need further refinement.

In order to show that the final result generated by a floating-point square root algorithm represents the correctly rounded value of the infinitely precise result $\sqrt{a}$ in any rounding mode, it was proved (by methods described in [3] and [4]), that the exact value of $\sqrt{a}$ and the final result R* of the algorithm before rounding belong to the same interval of width 1/2 ulp, adjacent to a floating-point number. Then, just as for the divide operation

$$(\sqrt{a})_{rnd} = (R^*)_{rnd}$$

where *rnd* is any IEEE rounding mode.

Floating-point square root algorithms are provided for single precision, double precision, double-extended and full register file format precision, and for SIMD single precision, in two variants. One achieves minimum latency, and one maximizes throughput.

SIMD Floating-Point Square Root Algorithm

We next present as an example the algorithm that allows computing the SIMD single precision square root, and which is optimized for throughput, having a minimum

number of floating-point instructions. The input operand is a pair of single precision numbers (a_1, a_2). The output is the pair ($\sqrt{a_1}$, $\sqrt{a_2}$). All the computational steps are performed in single precision. The algorithm is shown below as a scalar computation. The approximate values shown on the right-hand side are computed assuming no rounding errors occur, and they neglect some high order terms that are very small.

(1) $y_0 = 1/\sqrt{a} \cdot (1 + \varepsilon_0)$, $|\varepsilon_0| \le 2^{-m}$, m=8.831

(2) $h = (1/2 \cdot y_0)_{rn} \approx (1 / (2 \cdot \sqrt{a})) \cdot (1 + \varepsilon_0)$

(3) $t_1 = (a \cdot y_0)_{rn} \approx \sqrt{a} \cdot (1 + \varepsilon_0)$

(4) $t_2 = (1/2 - t_1 \cdot h)_{rn} \approx -\varepsilon_0 - 1/2 \cdot \varepsilon_0^2$

(5) $y_1 = (y_0 + t_2 \cdot y_0)_{rn} \approx 1/\sqrt{a} \cdot (1 - 3/2 \cdot \varepsilon_0^2)$

(6) $S = (a \cdot y_1)_{rn} \approx \sqrt{a} \cdot (1 - 3/2 \cdot \varepsilon_0^2)$

(7) $H = (1/2 \cdot y_1)_{rn} \approx (1 / (2 \cdot \sqrt{a})) \cdot (1 - 3/2 \cdot \varepsilon_0^2)$

(8) $d = (a - S \cdot S)_{rn} \approx a \cdot (3 \cdot \varepsilon_0^2 - 9/4 \cdot \varepsilon_0^4)$

(9) $t_4 = (1/2 - S \cdot H)_{rn} \approx 3/2 \cdot \varepsilon_0^2 - 9/8 \cdot \varepsilon_0^4$

(10) $S_1 = (S + d \cdot H)_{rn} \approx \sqrt{a} \cdot (1 - 27/8 \cdot \varepsilon_0^4)$

(11) $H_1 = (H + t_4 \cdot H)_{rn} \approx (1 / (2 \cdot \sqrt{a})) \cdot (1 - 27/8 \cdot \varepsilon_0^4)$

(12) $d_1 = (a - S_1 \cdot S_1)_{rn} \approx a \cdot (27/4 \cdot \varepsilon_0^4 - 729/64 \cdot \varepsilon_0^8)$

(13) $R = (S_1 + d_1 \cdot H_1)_{rnd} \approx \sqrt{a} \cdot (1 - 2187/128 \cdot \varepsilon_0^8)$

The first step is a table lookup performed by fprsqrta, which gives an initial approximation of ($1/\sqrt{a_1}$,$1/\sqrt{a_2}$) with known relative error determined by m = 8.831. The following steps implement a Newton-Raphson iterative algorithm. Specifically, step (5) improves on the approximation of ($1/\sqrt{a_1}$,$1/\sqrt{a_2}$). Steps (3), (6), (10) and (13) calculate increasingly better approximations of ($\sqrt{a_1}$,$\sqrt{a_2}$). The algorithm was proved correct as outlined in [3] and [4]. The final result (R_1,R_2) equals (($\sqrt{a_1}$)$_{rnd}$,($\sqrt{a_2}$)$_{rnd}$) for any IEEE rounding mode *rnd*, and the status flag settings and exception behavior are IEEE compliant.

The assembly language implementation is as follows (only the floating-point operations are numbered):

```
      movl r3 = 0x3f0000003f000000;; // +1/2,+1/2
      setf.sig f7=r3 // +1/2,+1/2 in f7
(1)   fprsqrta.s0 f8,p6=f6;;          // y0=1/sqrt(a) in f8
(2)   (p6) fpma.s1 f9=f7,f8,f0        // h=1/2*y0 in f9
(3)   (p6) fpma.s1 f10=f6,f8,f0;; // t1=a*y0 in f10
(4)   (p6) fpnma.s1 f9=f10,f9,f7;;// t2=1/2-t1*h in f9
(5)   (p6) fpma.s1 f8=f9,f8,f8;;  // y1=y0+t2*y0 in f8
(6)   (p6) fpma.s1 f9=f6,f8,f0    // S=a*y1 in f9
(7)   (p6) fpma.s1 f8=f7,f8,f0;;  // H =1/2*y1 in f8
(8)   (p6) fpnma.s1 f10=f9,f9,f6 // d=a-S*S in f10
(9)   (p6) fpnma.s1 f7=f9,f8,f7;; // t4=1/2-S*H in f7
(10)  (p6) fpma.s1 f10=f10,f8,f9// S1=S+d*H in f10
(11)  (p6) fpma.s1 f7=f7,f8,f8;;  // H1=H+t4*H in f7
(12)  (p6) fpnma.s1 f9=f10,f10,f6;;// d1=a-S1^2 in f9
(13)  (p6) fpma.s0 f8=f9,f7,f10;;//R=S1+d1*H1 in f8
```

Software Assistance Conditions for Scalar Floating-Point Square Root

Just as for divide, cases of special input operands were identified in the process of proving the IEEE correctness of the square root algorithms [4]. The difference with respect to divide is that only loss of precision of an intermediate result can occur in an iterative algorithm calculating the floating-point square root. Such operands might prevent the sequence from generating correct results, and they require alternate algorithms implemented in software in order to avoid this. These special situations are identified by the following condition that defines the necessity for IA-64 Architecturally Mandated Software Assistance for the scalar floating-point square root operation:

$e_a \le e_{min} + N - 1$ (d_i might lose precision)

where e_a is the (unbiased) exponent of a, e_{min} is the minimum value of the exponent in the given format, and N is the number of bits in the significand. When this condition is met, frsqrta issues a Software Assistance request in the form of a floating-point exception, instead of providing a reciprocal approximation for $1/\sqrt{a}$, and it clears its output predicate. The result of the floating-point square root operation is provided by an SWA handler, and the rest of the iterative sequence for calculating $\sqrt{a}$ is predicated off. Due to the extended internal exponent range (17 bits), the single precision, double precision, and double-extended precision calculations will never require architecturally mandated software assistance. This type of software assistance might be required only for floating-point register file format computations with floating-point numbers having 17-bit exponents.

When an architecturally mandated software assistance request occurs for the square root operation, the result is provided by the IA-64 Floating-Point Emulation Library.

Just as for the parallel divide, the parallel reciprocal square root approximation instruction, fprsqrta, does not signal any SWA requests. When the condition shown

above is met, fprsqrta merely clears its output predicate, in which case the result of the parallel square root operation has to be computed by alternate algorithms (typically by unpacking the parallel operands, performing two single precision square root operations, and packing the results into a SIMD result).

DERIVED OPERATIONS: INTEGER DIVIDE AND REMAINDER

The integer divide and remainder operations are based on floating-point operations. They are not specified in the IEEE Standard [1], but their implementation is so close to that of the floating-point operations mandated by the standard, that it is worthwhile mentioning them here.

A 64-bit integer divide algorithm can be implemented based on the double-extended precision floating-point divide. A 32-bit integer divide algorithm can use the double precision divide. The 16-bit and 8-bit integer divide can use the single precision divide. But the desired computation can be performed in each case by shorter instruction sequences. For example, 24 bits of precision are not needed to implement the 16-bit integer divide.

As examples, the signed and then unsigned 16-bit integer divide algorithms are presented. They are both based on the same core (all four steps below are performed in full register file double-extended precision):

(1) $y_0 = 1/b \cdot (1 + \varepsilon_0)$, $|\varepsilon_0| \le 2^{-m}$, $m=8.886$

(2) $q_0 = (a \cdot y_0)_{rn} = a/b \cdot (1 + \varepsilon_0)$

(3) $e_0 = (1 + 2^{-17} - b \cdot y_0)_{rn} \approx -\varepsilon_0$ (adding 2^{-17} ensures correctness of the final result)

(4) $q_1 = (q_0 + e_0 \cdot q_0)_{rn} \approx a/b \cdot (1 - \varepsilon_0^2)$

The assembly language implementation of the 16-bit signed integer divide algorithm follows. It is assumed that the 16-bit operands are received in r32 and r33, and the result is returned in r8.

```
sxt2 r2=r32    // sign-extend dividend
sxt2 r3=r33;;  // sign-extend divisor
setf.sig f8=r2  // integer dividend in f8
setf.sig f9=r3  // integer divisor in f9
movl r9=0x8000400000000000;;// 1 + 2^-17 in r9
setf.sig f10=r9  // (1 + 2^-17) · 2^63 in f10
fcvt.xf f6=f8   // normal fp dividend in f6
fcvt.xf f7=f9;; // normal fp divisor in f7
fmerge.se f10=f1,f10 // 1 + 2^-17 in f10
(1) frcpa.s1 f8,p6=f6, f7;; // y0 in f8
(2) (p6) fma.s1 f9=f6, f8, f0 // q0 = a * y0 in f9
(3) (p6) fnma.s1 f10=f8,f7,f10;; //e0=(1+2^-17)-b*y0
(4) (p6)  fma.s1 f8=f9,f10,f9;;// q1=q0+e0*q0 in f8
    fcvt.fx.trunc.s1 f8=f8;; // integer quotient in f8
    getf.sig r8=f8;;  // integer quotient in r8
```

The 16-bit unsigned integer divide is similar, but uses the zero-extend instead of the sign-extend instruction from 2 bytes to 8 bytes (zxt2 instead of sxt2), conversion from unsigned integer to floating-point for the operands (fcvt.xuf instead of fcvt.xf), and conversion from floating-point to unsigned integer for the result (fcvt.fxu.trunc instead of fcvt.fx.trunc).

The integer remainder algorithms are implemented as extensions of the corresponding integer divide algorithms. The 16-bit signed integer remainder algorithm is almost identical to the 16-bit signed integer divide, with the last instruction replaced by the following sequence that is needed to calculate $r = a - (a/b) \cdot b$:

```
fcvt.xf f8=f8;; // convert to fp and normalize
fnma.s1 f8=f8, f7, f6;; // r = a - (a/b) · b in f8
fcvt.fx.trunc.s1 f8=f8;;// integer remainder in f8
getf.sig r8=f8;;  // integer remainder in r8
```

EXCEPTIONS AND TRAPS

IA-64 arithmetic floating-point instructions [2] may signal all of the five IEEE-specified exceptions and also the Intel Architecture defined exception for denormal operands. Invalid operation, denormal operand, and divide-by-zero are pre-computation exceptions (floating-point faults). Overflow, underflow, and inexact result are post-computation exceptions (floating-point traps).

In addition to these user visible exceptions, Software Assistance (SWA) faults and traps can be signaled. They do not surface to the user level, and cannot be disabled (masked). The SWA requests are handled by a system SWA handler, the IA-64 Floating-Point Emulation Library.

The status flags in a given status field can be cleared using the fclrf.*sf* instruction. Control bits may be set using the fsetc.*sf amask7, omask7* instruction, which initializes the seven control bits of the specified status field to the value obtained by logically AND-ing the sf0.controls (seven bits) and *amask7,* and logically OR-ing with *omask7.* Alternatively, a 64-bit unsigned

integer value can be moved to or from the FPSR (application register 40): mov ar40 = *r1*, or mov *r1* = ar40.

Exception handlers can be registered, disabled, saved, or restored with software support (from the operating system and/or compiler) as specified by the IEEE Standard.

IA-64 Software Assistance Faults and Traps

The IA-64 architecture allows virtually any floating-point operation to be deferred to software through the mechanism of Software Assistance requests, which are treated as floating-point exceptions, always unmasked, and resolved without reaching a user handler. On Itanium™, the first implementation of the IA-64 architecture, SWA requests may be signaled in three forms:

- IA-64 architecturally mandated SWA faults. These occur for certain combinations of operands of the floating-point divide and square root operations, and only for frcpa and frsqrta (scalar reciprocal approximation instructions).
- Itanium-specific SWA faults. They occur whenever a floating-point instruction has a denormal input. All the arithmetic floating-point instructions on Itanium signal this exception, except for fma.*pc*.*sf f1 = f2*, f1, f0 (fnorm.*pc*.*sf f1=f2*), fms.*pc*.*sf f1 = f2*, f1, f0, and fnma.*pc*.*sf f1 = f2*, f1, f0. They signal Itanium-specific SWA faults only when the input is a canonical double-extended denormal value (i.e., when the input has a biased exponent of 0x00000 and a most significant bit of the non-zero significand equal to 0).
- Itanium-specific SWA traps. They occur whenever a floating-point instruction has a tiny result (smaller in magnitude than the smallest normal floating-point number that can be represented in the destination format). These exceptions only occur for fma, fms, fnma, fpma, fpms, and fpnma.

The IA-64 Floating-Point Emulation Library

When an unmasked floating-point exception occurs, the hardware causes a branch to the interruption vector (Floating-Point Fault or Trap Vector) and then to a low-level OS handler. From here, handling of the floating-point exception is propagated higher in the operating system, and an exception handler is invoked that decides whether to provide a result for the excepting instruction and allow execution of the application to continue.

SWA requests are treated like regular floating-point exceptions, but they are always ‘unmasked’ and handled by an SWA handler represented by the IA-64 Floating-Point Emulation Library. The library is able to calculate the result for any IA-64 arithmetic floating-point instruction. When an SWA fault or trap occurs, it is processed and the result is provided to the operating system kernel. The execution continues in a transparent manner for the user. In addition to satisfying the SWA requests, the SWA handler filters all other unmasked floating-point exceptions that occur, passing them to the operating system kernel that will continue to search for an appropriate user-provided exception handler.

Figure 1 depicts the control flow that occurs when an application running on an IA-64 processor signals an unmasked floating-point exception. The IA-64 Floating-Point Emulation Library is shown as part of the operating system kernel, but this is implementation dependent. If an unmasked floating-point exception other than an SWA fault or trap occurs, a user handler must have already been registered in order to resolve it. The user handler can be called directly by the operating system, receiving ‘raw’ information about the exception, or through an optional IEEE filter (as shown in Figure 1) that processes the information about the exception, thereby allowing a less complex handler to resolve the situation.

An example of an SWA trap is illustrated in Figure 2. The computation generates a result that is tiny and inexact (sufficient to trigger an underflow or an inexact trap if any was unmasked). As traps are masked, an Itanium™-specific SWA trap occurs, propagated from the application code to the floating-point emulation library via the OS kernel trap handler in steps 1 and 2. The result generated by the emulation library is then passed back in steps 3 and 4.

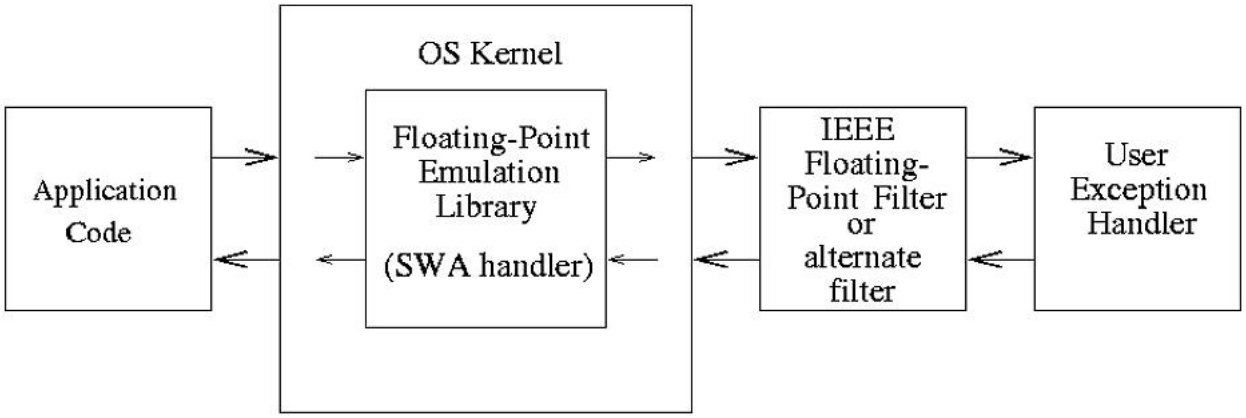


Figure 1: Flow of control for IA-64 floating-point exceptions

DIVIDE EXAMPLE: (1.0...0100 * 2^emin) / (1.0 * 2^4) = 0.0001 * 2^emin
(U traps disabled; SWA trap)

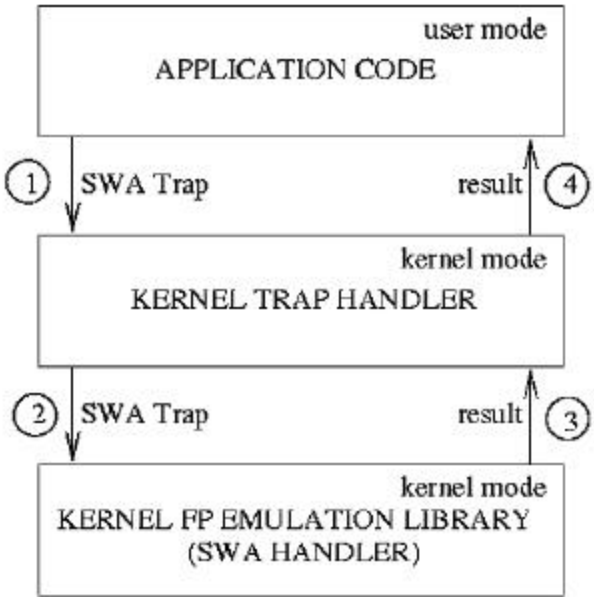


Figure 2: Flow of control for handling an SWA trap signaled by an IA-64 floating-point instruction

FLOATING-POINT EXCEPTION HANDLING

The floating-point exception priority is documented in [2], but for a given implementation of the architecture (Itanium™ in this case), a distinction can be made regarding the source of an exception. This can be signaled by the hardware, or from software, by the IA-64 Floating-Point Emulation Library.

For example, on Itanium, denormal faults are signaled by software (the IA-64 Floating-Point Emulation Library) after they are reported initially by the hardware as Itanium-specific SWA faults. SWA faults that are not converted to denormal faults (because denormal faults are masked) cause the result to be calculated by software. Whether the result of a floating-point instruction is calculated in hardware or in software, it can further signal other floating-point exceptions (traps).

For example, architecturally mandated SWA faults might lead to overflow, underflow, or inexact exceptions signaled from the IA-64 Floating-Point Emulation Library.

Another example is that of the SWA traps, that are always raised from hardware. They have to be resolved in software, but this computation might further lead to inexact exceptions signaled from the IA-64 Floating-Point Emulation Library.

The information that is relevant to a floating-point user exception handler is passed to it through a register file save area, the excepting instruction pointer and opcode, the Floating-Point Status Register, and a set of specialized registers.

The IA-64 IEEE Floating-Point Filter

The floating-point exception handling mechanism of an operating system raises portability issues, as exception handling is almost always implemented using proprietary data structures and procedures. A solution that can be adopted is to implement an IEEE Floating-Point Filter that preprocesses the exception information provided by the operating system kernel before passing it on to the user handler (see Figure 1). The filter, which can be viewed as part of the user handler, helps in the processing of all the IEEE floating-point exceptions (invalid operation, divide-by-zero, overflow, underflow, and inexact result) and also in the processing of the denormal exceptions that are IA specific. The interface between the operating system and the IEEE filter should be almost identical to that of the IA-64 Floating-Point Emulation Library, as they both process exception information. The IEEE filter also accomplishes the correct wrapping of the exponents when overflow or underflow traps are taken, as required by the IEEE Standard [1] (operation deferred to software by the IA-64 architecture).

An important advantage is that the IEEE Floating-Point Filter simplifies greatly the task of the user handler. All the complexities of reading operating system-specific information, decoding operation codes, and reading and writing floating-point or predicate registers are abstracted away by the filter. Also, exceptions generated by parallel (SIMD) instructions will appear to the user handler as originating in scalar instructions. The following two examples illustrate some of these benefits.

The example in Figure 3 illustrates the case of a scalar divide operation that signals an SWA fault, and then an underflow trap (underflow traps are assumed to be unmasked). The SWA fault is signaled by an frcpa instruction that jumpstarts the iterative computation calculating the quotient. The sequence of steps performed in handling the exception is numbered from 1 to 10 in the figure. As the result is provided by the user exception handler for underflow exceptions, the output predicate of frpca has to be clear when execution of the application program containing it is resumed (clearing the output predicate is the task of the user handler or of the IEEE Floating-Point Exception Filter if present). The clear output predicate disables the iterative computation following frcpa, as the result is already in the correct floating-point register (the iterative computation is assumed to be automatically inlined by the compiler).

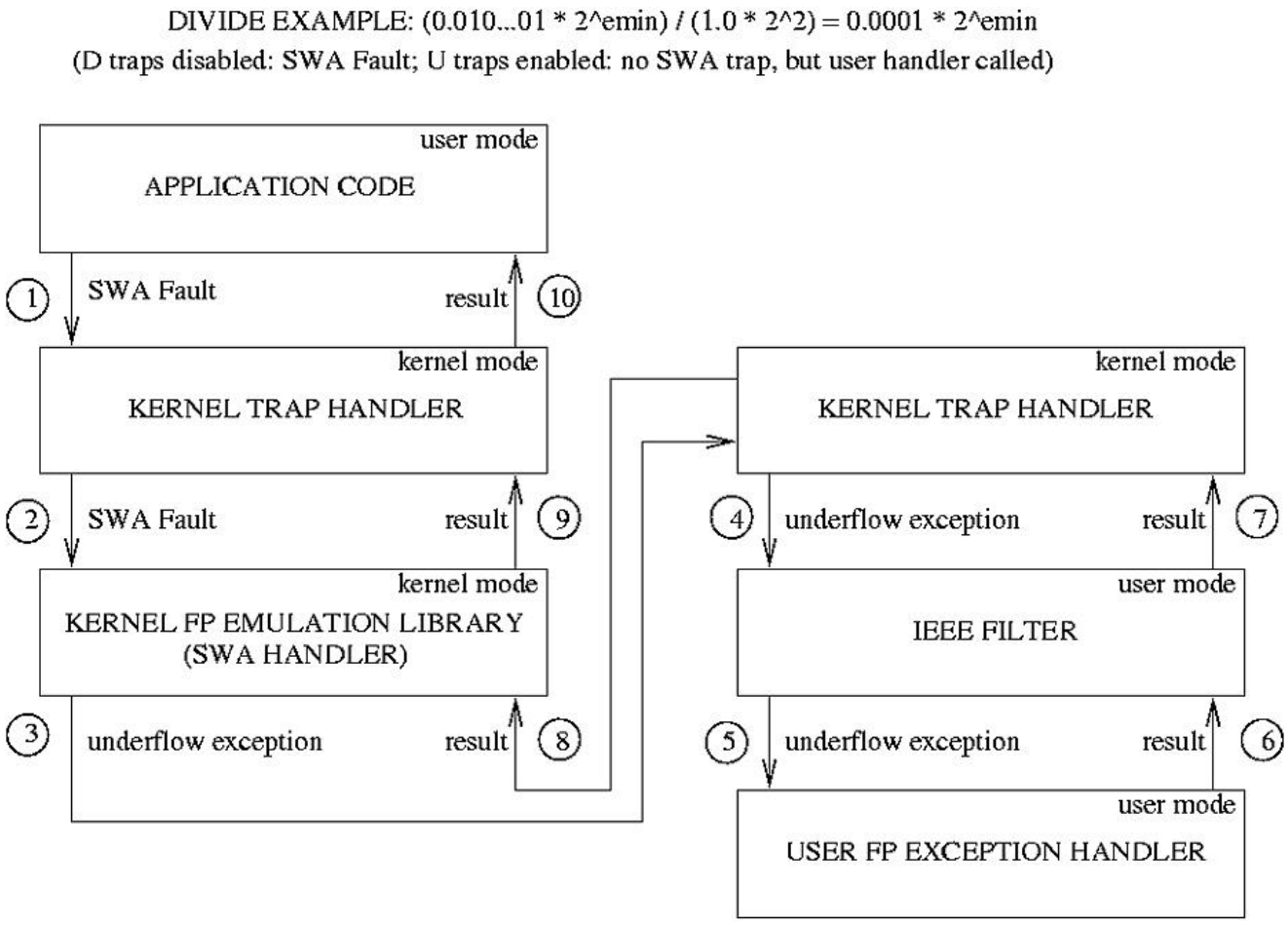


Figure 3: Flow of control for handling an SWA fault signaled by a divide operation, followed by an underflow trap

The example in Figure 4 illustrates the case of a parallel instruction that signals an invalid fault in the high half, and an underflow trap in the low half, with no SWA requests involved. Both invalid and underflow exceptions are assumed to be unmasked (enabled). As only the fault is detected first, the IEEE filter tries to re-execute the low half of the instruction, generating a new exception (underflow trap). The sequence of steps executed while handling these exceptions is numbered from 1 to 12 in the figure.

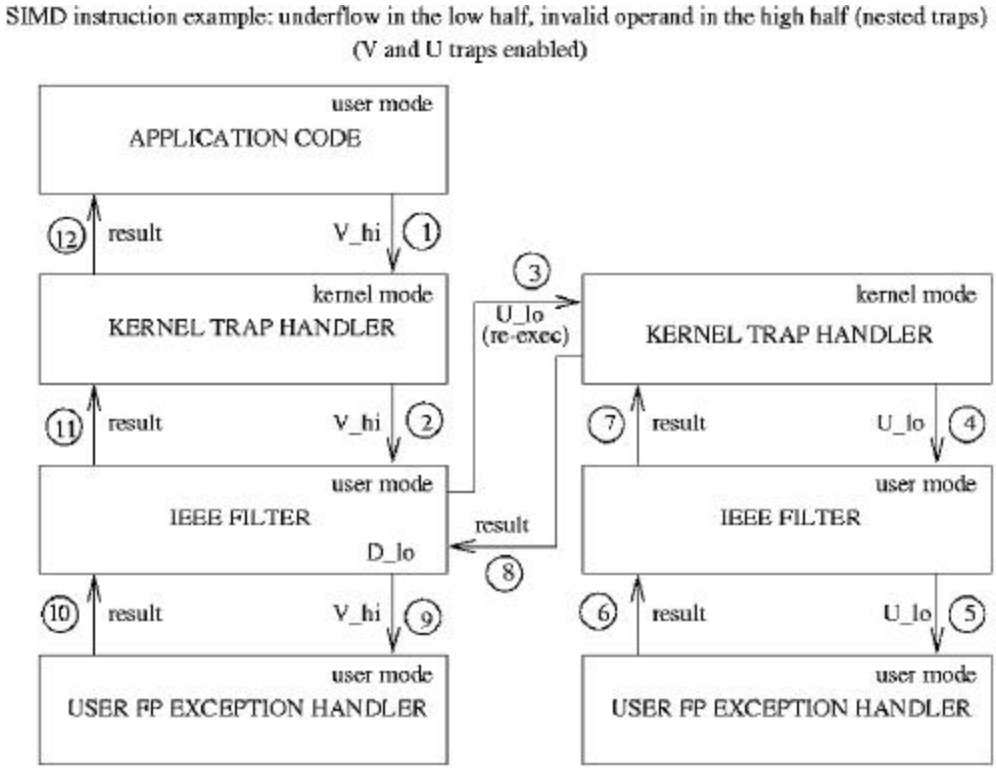


Figure 4: Flow of control for handling an invalid fault in the high half (V high) and an underflow trap in the low half (U low) of a parallel IA-64 instruction

SPECULATION FOR FLOATING-POINT COMPUTATIONS

Control speculation refers to a performance optimization technique where a sequence of instructions is executed before it is known that the dynamic control flow of the program will actually reach the point in the program where the sequence of instructions is needed. Control speculation in floating-point computations on IA-64 processors is possible, as loads to general or floating-point registers have both non-speculative (e.g., ldf, ldfp), and speculative (e.g., ldf.s, ldfp.s) variants. All instructions that write their results to general or floating-point registers are speculative.

A speculative floating-point computation uses status fields sf2 or sf3. The computation is considered to have failed if it signals a floating-point exception that is unmasked in the user status field sf0, or if it sets a status flag that is clear in sf0. This is checked for with the floating-point check flags instruction, fchkf.*sf target25*: the status flags in *sf* are compared with the status flags in sf0. If any flags in *sf* are set and the corresponding traps are enabled, or if any flags are set in *sf* that are not set in sf0, then a branch is taken to *target25*, which should be the address of the recovery code for the failed speculative floating-point computation. The compliance with the IEEE Standard is thus preserved even for speculative chains of computation.

The following example shows original code without control speculation. It is assumed that the contents of f9 are not used at the destination of the branch.

```
        (p6) br.cond some_label ;;
        fma.s0 f9=f8,f7,f6  // Do f9=f8*f7+f6
continue_label:
```

This code sequence can be rewritten using control speculation with sf2 to move the fma ahead of the branch as follows:

```
        fma.s2 f9=f8,f7,f6  // Speculative f9=f8*f7+f6
        // other instructions
        (p6) br.cond some_label ;;
        fchkf.s2 recovery_label   // Check speculation
continue_label:
```

If sf0 and sf2 do not agree, then the recovery code must be executed to cause the actual exception with sf0.

```
recovery_label:
        fma.s0 f9=f8,f7,f6  // Do real f9=f8*f7+f6
        br continue_label
```

CONCLUSION

Compliance with the IEEE Standard for Binary Floating-Point Arithmetic [1] is important for any modern processor. In this paper, we have shown how various facets of the standard are implemented or reflected in the IA-64 architecture, which is fully compliant with the IEEE Standard. In addition, we have highlighted features of the floating-point architecture that allow high-accuracy and high-performance computations, while abiding by the IEEE Standard.

ACKNOWLEDGMENTS

The authors thank Roger Golliver, Gautam Doshi, John Harrison, Shane Story, Ted Kubaska, and Cristina Iordache from Intel Corporation, and Peter Markstein from Hewlett-Packard* Company for their contributions, support, ideas, and/or feedback regarding various parts of this paper.

REFERENCES

[1] ANSI/IEEE Std 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE, New York, 1985.

[2] *IA-64 Application Developer's Architecture Guide*, Intel Corporation, 1999.

[3] Cornea-Hasegan, M., "Proving IEEE Correctness of Iterative Floating-Point Square Root, Divide, and Remainder Algorithms," *Intel Technology Journal*, Q2, 1998 at http://developer.intel.com/technology/itj/q21998.htm

[4] Cornea-Hasegan, M. and Golliver, R., "Correctness Proofs Outline for Newton-Raphson Based Floating-Point Divide and Square Root Algorithms," *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, 1999, IEEE Computer Society, Los Alamitos, CA, pp. 96-105.

[5] *Intel Architecture Software Developer's Manual*, Intel Corporation, 1997.

AUTHORS' BIOGRAPHIES

Marius Cornea-Hasegan is a senior staff software engineer with Intel Corporation in Hillsboro, Oregon. He holds an M.Sc. degree in electrical engineering from the Polytechnic Institute of Cluj, Romania, and a Ph.D. degree in computer science from Purdue University, in West Lafayette, IN. His interests include floating-point architecture and algorithms as parts of a computing system. His e-mail is marius.cornea@intel.com.

Bob Norin joined Intel in 1994. Currently he is manager of the MSL Numerics Group in MPG. Bob received his Ph.D. degree in electrical engineering from Cornell University. He has over 25 years experience developing software for high-performance computers. His technical interests include optimizing the performance of floating-point applications and developing mathematical library software. His e-mail is bob.norin@intel.com.