Scientific Computing on the Itanium™ Processor*

Bruce Greer John Harrison Greg Henry Wei Li Peter Tang

Computational Software Lab

Intel Corporation

ABSTRACT

The 64-bit Intel® Itanium™ architecture is designed for high-performance scientific and enterprise computing, and the Itanium processor is its first silicon implementation. Features such as extensive arithmetic support, predication, speculation, and explicit parallelism can be used to provide a sound infrastructure for supercomputing. A large number of high-performance computer companies are offering Itanium™-based systems, some capable of peak performance exceeding 50 GFLOPS. In this paper we give an overview of the most relevant architectural features and provide illustrations of how these features are used in both low-level and high-level support for scientific and engineering computing, including transcendental functions and linear algebra kernels.

Keywords

EPIC, Itanium(TM) processor, fused multiply-add, linear algebra, transcendental functions

1. INTRODUCTION

The 64-bit Intel® Itanium™ architecture is designed for high-performance scientific and enterprise computing. Features such as extensive arithmetic support, predication, speculation, and explicit parallelism can be used to provide a sound infrastructure for supercomputing. The architecture has been carefully designed to allow efficient implementations and an effective combination of hardware and software, not only with current techniques but with a view to future trends in, for example, process technology and compiler optimization.

The Intel Itanium™ processor is the first in a line of high-performance implementations of this architecture. Later members of the Itanium Processor Family (IPF) are expected to be still more powerful, with higher clock speed, lower instruction latencies and greater memory bandwidth. Economies of scale and improvements in process technology are also expected to make them cheaper. Some preliminary facts about the second implementation of the Itanium architecture, currently codenamed McKinley, can be found in Section 6.

We maintain that the IPF will have a profound impact on supercomputing in the years to come. There are several reasons to believe this. Most importantly, all the major Supercomputing key players (IBM, HP, Compaq, NEC, SGI, Unisys, Hitachi, Fujitsu-Siemens, etc.), with the possible exception of Sun Microsystems, have plans to use Itanium processors in some of their high-end future products. This paper will go over some of the features that make this processor so attractive to them. It is our hope that by establishing a new high-end standard, the Itanium architecture will allow computer designers to consolidate research efforts that are currently fragmented over a number of competing proprietary RISC architectures. This may lead to the same efficiencies and economies of scale reaped as a result of the wide acceptance of Intel's 32-bit architecture (IA-32) currently represented by the Intel Pentium® 4 processor.

Some features of the Itanium architecture are completely new, whereas others are known from previous architectures — for example predicated execution from the Advanced Risc Machines Ltd. ARM† family [6] and the fused multiply-add from the IBM RS/6000† family [7]. Nevertheless the Itanium architecture is the first to bring together a unique combination of these features into a synergetic whole, intended to provide new scope for the implementation of still more powerful processors over the coming decades. In later sections, some of the key architectural features will be discussed in the context of their typical applications, moving from the simplest kernels to more complicated serial and parallel libraries. One should recognize that these features are not all hardware-related, and that certain architectural features imply software methodology changes. The ability to use the

SC2001 November 2001, Denver

†All other brands are the property of their respective owners.

rotating registers to get potentially extremely complicated code into short and simple loops is described in Section 5, for example.

This paper elaborates on the thesis that the Itanium processor is important to supercomputing. Section 2 surveys the key architectural features most relevant to scientific computing, and Section 3 explains how optimizing compilers can make use of these features to produce high quality numeric code. Section 4 discusses the construction of an accurate and fast run-time math library and Section 5 shows how the crucial kernels for linear algebra are optimized on the Itanium processor. These libraries are important in themselves as part of the software support for scientific computing, and also provide an excellent illustration on how architectural features can be exploited. Section 6 gives a brief overview of some significant improvements in the second implementation of the Itanium architecture, and finally Section 7 gives some concluding remarks.

2. KEY ARCHITECTURAL FEATURES

The Itanium architecture is based on the 'Explicitly Parallel Instruction Computer' (EPIC) philosophy. The basic EPIC principle is that the programmer or compiler should be able to indicate the inherent parallelism of programs *explicitly* in the instruction sequence, rather than obliging the processor to reconstruct it from a particular sequence of serial operations. In order to allow the programmer to expose more instruction-level parallelism (ILP), the architecture offers a unique combination of features including full predication, speculative and advance loads, and automatic register rotation for software-pipelined loops. In what follows we will focus mainly on the special features of the floating-point architecture; Dulong [2] discusses the rationale for other architectural features.

The Itanium architecture's floating-point instruction set has been carefully designed to combine high performance and good accuracy. A large floating-point register set (128 registers) is provided, and almost all operations can read their arguments from and write their results to arbitrary registers. Together with register rotation for software pipelined loops, this allows the encoding of common algorithms without running short of registers or needing to move data between them in elaborate ways. Registers can store floating-point numbers in a variety of formats, and the rounding of results is determined by flexible combinations of several selectable defaults and additional instruction completers. The basic arithmetic operation, the *fused multiply add*, allows higher accuracy and performance in many common algorithms. Several additional features are also present to support common programming idioms.

Floating-point multiply-add

The centerpiece of the Itanium architecture's floating-point instruction set is the `fma` (floating-point multiply add or fused multiply-accumulate) instruction, which computes $x \times y + z$ in a single floating-point operation with no intermediate rounding of the product. Related instructions perform the same operation with a sign change: `fms` computes $x \times y - z$ while `fnma` computes $z - x \times y$. Addition and multiplication are implemented as degenerate cases of the `fma`: $x \times 1 + y$ and $x \times y + 0$, which can be used without storing special constants, since registers `f0` and `f1` are hardwired to 0 and 1 respectively.[1]

Format	Precision	Exponent range
Single	24	$-126 \leq e \leq 127$
Double	53	$-1022 \leq e \leq 1023$
Double-extended	64	$-16382 \leq e \leq 16383$
Register	64	$-65534 \leq e \leq 65535$

Table 1: Some floating-point formats on the Itanium architecture

In implementations, there is almost no latency penalty for a `fma` operation over a direct implementation of a pure multiply. Thus, a single `fma` is significantly faster than a separate multiply and add. On code where adds and multiplies are heavily interleaved, this can lead to a significant performance increase. Obvious examples are the evaluation of polynomials and of vector dot products of the form $\mathbf{x} \times \mathbf{y}$. The latter computation

$$p = \Sigma_{i=0}^{N-1} x_i y_i$$

can be performed by a succession of `fma` operations of the form:

$$p = p + x_i \times y_i$$

Aside from its speed advantage, the fact that no intermediate rounding is performed on the product also tends to reduce rounding errors. In common cases this difference may be relatively unimportant, but can be crucial in special situations such as the implementation of division discussed later in this section or the run-time library discussed in Section 4.

Extended precision

The Itanium architecture supports a variety of floating-point formats, including those shown in Table 1 and various intermediate types such as a 'stack single' type with precision 24 but exponent range $-16382 \leq e \leq 16383$. Operations can be performed on arguments of different formats, making it easy to mix say, double-precision inputs with register-format intermediate results for better accuracy, or to store short constants in single precision to economize on memory. Results can also be rounded back into any of the supported formats, allowing narrowing casts without incurring double-rounding.

Thanks to extended precision, rounding errors are often less significant than they would be on an architecture where double is the highest available precision, yet the speed of operations is similar, far better than for a software implementation of quad arithmetic. Even when designing routines with a double-precision external interface, extended

[1]When the third argument is `f0`, the `fma` uses different rules for the determination of zero signs, in accordance with the IEEE Standard 754 [5].

precision can often be exploited for internal calculations. An important application can be found in double-precision polynomial evaluation (c.f. Section 4).

Multiple status fields

Given the number of floating-point formats available in the Itanium architecture, it is important to have a flexible means of specifying the desired floating-point format for a particular result to be rounded into, as well as the direction of rounding (e.g. rounding to nearest or truncation). Having only a single status register would be inconvenient where there are several parallel threads of control, or where exceptions in some intermediate instructions need to be ignored. Therefore, the Itanium architecture features four different 'status fields' which can be specified by completers in the main floating-point instructions. An instruction with a given status field completer is then controlled by that status field.

Software conventions determine many of the appropriate applications for particular status fields. Typically `sf0` is the main 'user' status field used for most floating-point calculations, and `sf1`, with all exceptions disabled, is used for intermediate calculations in many standard numerical software kernels, e.g. those for division, square root and transcendental functions. However, the multiple status fields can be put to other uses. In particular, when implementing interval arithmetic one often wants to be able to switch repeatedly between rounding up and rounding down in a short sequence of calculations. On many existing architectures, changing the rounding mode is so costly that performance degrades dramatically. However, on the Itanium architecture, one can simply set up two status fields to have different rounding directions and use whichever is desired on each instruction.

Division and square root

There are no instructions specified in the Itanium architecture (except in IA-32 compatibility mode) for performing floating-point division or square root operations. Instead, the only instruction specifically intended to support division is the *floating-point reciprocal approximation* instruction, `frcpa`, which given a floating-point number a, returns an approximation to $1/a$ good to about 8 bits.[2] Similarly, the only instruction to support square root is the `frsqrta` (*floating-point reciprocal square root approximation*) instruction, which given a floating-point number a, returns an approximation to $1/\sqrt{a}$ good to about 8 bits. These initial approximations are intended to be refined to perfectly rounded quotients or square roots by software. The refinement calculations can be efficiently performed because of the `fma` instruction, as $r = b - aq$ can be calculated accurately (without intermediate rounding) for $q \approx b/a$.

The decomposition of division and square root into a number of simple, fast operations tends to increase throughput, since these operations inherit the high degree of pipelining in the basic `fma` operations. For example, in an implementation such as the Itanium processor with two fully pipelined `fma` units, double-extended precision division has an throughput of one operation every 7 cycles, far better than on most other architectures. In addition, more flexibility is afforded to the programmer or compiler to schedule the division in conjunction with other instructions. Finally, if an IEEE-correct result is not required (e.g. in graphics applications), much faster algorithms can be substituted. Two other novel uses of `frcpa` are given in Section 4.

[2]In special cases such as $a = 0$ the behavior is different, indicated by a predicate register setting.

Other features

There are a number of other floating-point instructions designed to support common programming idioms. For example `famax` and `famin` return whichever of the two argument values has the larger *absolute* value, which is often useful in a variety of computations. One interesting example is obtaining an exact sum by performing a floating-point addition and then subtracting off the summands, the larger one first, to recover the rounding error:

$$\begin{aligned} Hi &= x + y \\ max &= \texttt{famax}(x,\ y) \\ min &= \texttt{famin}(x,\ y) \\ tmp &= max - Hi \\ Lo &= tmp + min \end{aligned}$$

3. ITANIUM ARCHITECTURE OPTIMIZING COMPILERS

The features of the Itanium architecture provide new opportunities for the compiler to optimize applications. In May of 2001, the Itanium processor set the record-breaking floating-point performance of 711 Spec2000FP (base) using the Intel compiler.[3]

The Intel compiler for the Itanium architecture targets two major goals: minimizing the impact of memory accesses, and maximizing parallelism. The compilation techniques take advantage of the Itanium architectural features. 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. An overview of the Intel compiler can be found in [3].

The compiler has a comprehensive set of optimizations targeting scientific applications, including loop transformations, array dependence analysis, scalar replacement, data prefetching, data layout optimizations, software pipelining, locality analysis, and many more. Below is a sample list of these optimizations.

Software Pipelining

Software pipelining improves the performance of a loop by overlapping the execution of several iterations to increase

[3]Described as "phenomenal" by Stephen Shankland, *Itanium scores high in performance tests*, CNET News.com, May 30, 2001, 12:50 p.m. PT, `http://news.cnet.com/news/0-1003-200-6112206.html`.

instruction-level parallelism. The Intel compiler pipelines both counted loops and while loops. Control speculation is required to maximize the parallelism of while loops. Data speculation helps bypass the unlikely data dependencies often seen in sparse matrix applications. Loops with control flow are transformed, using predication, into single block loops suitable for pipelining.

In the Itanium architecture, rotating register support obviates the need for extensive unrolling, the traditional approach for RISC architectures. Rotating predicates are used to control the execution of the stages during the prologue and epilogue phases, so that only the kernel loop is required. In RISC architectures, these three execution phases are implemented using three distinct blocks of code.

Data Prefetching

The data prefetching implementation utilizes data-locality analysis to selectively prefetch only those data references that are likely to suffer cache misses. The predication support in the Itanium architecture provides an efficient way of adding prefetch instructions. The conditionals within the loop are converted to predicates through if-conversion, thus alleviating the need for unrolling, which would result in code expansion.

The rotating registers are used to reduce the number of prefetching instructions required. Multiple arrays accessed uniformly within a loop can be prefetched with a single `lfetch` instruction using a rotating register that rotates the addresses of the different arrays that have to be prefetched. This completely obviates the need for predicate calculations within the loop and saves memory slots that would otherwise be occupied by multiple `lfetch` instructions [1].

Loop Transformations

A large set of loop transformations have been implemented. Linear loop transformations are compound transformations representing sequences of loop reversal, loop interchange, loop skew, and loop scaling. These transformations can dramatically improve memory access locality, and improve the effectiveness of other optimizations, such as scalar replacement, invariant code motion, and software pipelining. Loop fusion improves cache performance, and reduces the cost of branches. Loop fusion in the Intel compiler for the Itanium architecture is more aggressive than that in compilers for RISC processors, since it takes advantage of the large number of available registers in the Itanium architecture. Loop unroll-and-jam unrolls the outer loops and fuses the unrolled copies together to enable more scalar replacement, which is very effective due to the large number of registers and rotating register support. Loop blocking is key to improving the cache performance of libraries and applications that manipulate large matrices of data items. Loop distribution splits a single nested loop into multiple adjacent nested loops that have a similar loop structure. 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.

Memory reference elimination

Scalar replacement replaces array memory references with registers. The Itanium 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 to eliminate the necessary moves introduced by scalar replacement. On traditional architectures, if one chooses to eliminate these moves, unrolling normally has to be used, with code expansion. Partial redundancy elimination is another technique to eliminate memory loads and stores for scalar references.

Parallelization

The Intel compiler supports OpenMP, 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. It also supports auto-parallelization, i.e. the compiler automatically detects parallelism and generates parallel code.

4. ACCURATE AND FAST RUN-TIME MATH LIBRARY

We must not lose sight of the fact that large-scale computing also depends on low-level support for fundamental scientific computing. A run-time library of mathematical functions is not only a well-accepted common set whose reliability and efficiency are crucial. It also reflects characteristics of low-level computational kernels that a specific large-scale computing problem may depend on critically. Thus, an accurate and fast run-time mathematical library gives two major benefits. First, the library itself is obviously valuable. Second, the ways in which architectural features are exploited to construct this run-time library are also likely to be applicable elsewhere. For this reason, we give an overview of the techniques employed in the construction of our IEEE double-precision run-time mathematical library of transcendental functions [4].

Parallelism and extra precision

One important consequence of a combination of extra precision (64 significant bits) and parallelism is that degree-n polynomials, for fairly large degree, can be evaluated in about $O(\log_2(n))$ latency, in contrast with $O(n)$ that a tradition Horner's recurrence offers. The method is simple recursive subdivision. The presence of 11 extra bits of accuracy allows for basically any algebraic decomposition of a polynomial and evaluation order. We have, by exhaustive enumeration, determined the optimal evaluation method (in terms of latency) of general and some special polynomials up to moderate degrees. Table 2 tabulates the results, for example, of general polynomials up to degree 15 on the Itanium processor. That we can evaluate long polynomials very quickly leads to interesting algorithms. For example, we no longer steadfastly shoot for very short polynomials by using large tables [8] but instead use smaller tables. In examples such as inverse tangents, or sine, we employ polynomials with as many as 22 terms.

Multiply accumulate

Polynomial	Latency (cycles)
$c_0 + c_1 x$	5
$c_0 + c_1 x + c_2 x^2$	10
$c_0 + c_1 x + \ldots + c_3 x^3$	11
$c_0 + c_1 x + \ldots + c_4 x^4$	15
$c_0 + c_1 x + \ldots + c_5 x^5$	16
$c_0 + c_1 x + \ldots + c_6 x^6$	16
$c_0 + c_1 x + \ldots + c_7 x^7$	17
$c_0 + c_1 x + \ldots + c_8 x^8$	20
$c_0 + c_1 x + \ldots + c_9 x^9$	21
$c_0 + c_1 x + \ldots + c_{10} x^{10}$	21
$c_0 + c_1 x + \ldots + c_{11} x^{11}$	22
$c_0 + c_1 x + \ldots + c_{12} x^{12}$	22
$c_0 + c_1 x + \ldots + c_{13} x^{13}$	23
$c_0 + c_1 x + \ldots + c_{14} x^{14}$	23
$c_0 + c_1 x + \ldots + c_{15} x^{15}$	24

Table 2: Optimal latency of polynomial evaluation on the Itanium processor

The utility of being able to compute $a \times b + c$ with just one rounding is tremendous. In transcendental function calculations, we often need to compute the form $X - N \times P$. Here, X is typically the input argument, and P an approximation to a "period" such as $\pi/2$, or $\log 2$. Let us elaborate on this somewhat subtle point. In a typical situation, one needs to compute $X - N\rho$, N being an integer value, to moderately more accuracy than the working precision in question. If we set P to be the machine representation of ρ, because by nature of this kind of calculation cancellation occurs in the subtraction, a simple `fma` allows us to obtain the result $X - N \times P$ *exactly*. We do incur a small error $N(\rho - P)$. But since ρ is fixed a priori, we can compensate easily for the lack of precision in P. Without `fma`, a rounding error will be incurred in the calculation of $N \times P$. This error cannot be compensated easily as it depends on the exact value of $N \times P$ and how it is rounded off. The workaround in the absence of `fma` is to reduce the precision of P so that a number of trailing bits are always zero. As long as this amount exceeds the number of bits in N, $N \times P$ is computed exactly. This workaround in essence reduces the accuracy we can have in P to such an extent that many extra steps are usually needed to compensate for it.

In one interesting instance of the calculation of the logarithm function log, `frcpa` is used together with `fma`. Here $P =$ `frcpa`(X) and $\log(X)$ is computed via

$$\log(X) = -\log(P) + \log(1 + (XP - 1)).$$

The value $\log(P)$ is obtained from a table of values calculated beforehand, and $\log(1 + (XP - 1))$ is computed by a short power series in the variable $t = XP - 1$ computed by a single `fma` instruction.

Parallelism

Parallelism is exploited not only in the evaluation of long expressions such as polynomials. In general, parallelism is exploited whenever a long critical path can be shortened significantly by an approximation whose correction can be computed in parallel. A notable, but by no means unique, situation involves division. Consider for example the calculation of the atan2 function. This function takes two argument X and Y and essentially calculates the phase angle of the complex number $X + iY$. The basic computation is of the form $atan(Y/X)$. On the surface, the division Y/X lies in the critical path. We exploited parallelism here by starting on the calculation of $\arctan(Z)$ where $Z = Y \times$ `frcpa`(X) immediately. The correction based on the formula

$$\arctan(Y/X) = \arctan(Z) + \arctan(\zeta), \quad \zeta = \frac{Y/X - Z}{1 + (Y/X)Z}$$

involves only a few terms of the Taylor series expansion in ζ and has a latency shorter than that of the main calculation. The latency of the division is thus essentially eliminated by use of parallelism. We note again that the correction term calculation is not nearly as convenient without the ever-useful multiply accumulate instruction. In another instance, we need to calculate $(1/X)^{25}$. Instead of calculating $1/X$ followed by exponentiation, we start the computation of W^{25} immediately where $W = \text{frcpa}(X)$. The correction needed is $(1 - \beta)^{-25}$ where $\beta = 1 - XW$. A polynomial approximation to the function $(1 - t)^{-25}$ is derived beforehand on the range $|t| \leq 2^{-8}$, and an evaluation of this polynomial at β (obtained via one multiply accumulate instruction) is carried out in parallel with the calculation of W^{25}.

Function	Latency (cycles)	Max. Error Observed (ulps)
exp	49	0.502
log	34	0.505
sin	62	0.502
atan	68	0.511
x^y	79	0.502

Table 3: Accuracy and speed of some run-time functions

Timing Results

We summarize in Table 3 the timing in cycles and accuracy in term of largest error observed in terms of units-of-last-place (ulps) of some key double-precision functions of the resulting run-time library.

5. LINEAR ALGEBRA

The same features used extensively in the scalar example earlier can be extended to scientific and engineering applications. We will discuss other features of the architecture that support pipelined loops.

While many efforts have been underway at Intel and other hardware and software vendors to port and optimize code for the Itanium processor, we will discuss our experiences with the creation of the Intel® Math Kernel Library[4] (MKL). The Itanium architecture variant has gone through several versions, with MKL 5.1 the current release. Many of the salient features of the architecture are exploited in DGEMM

[4] `http://developer.intel.com/products/software/mkl/index.htm`

from the level 3 BLAS.[5] We will use that as an example of how some of the architectural features are used. A set of vectorized transcendental functions, collectively known as the Vector Math Library (VML), is also part of MKL. We will briefly discuss how the Itanium architecture has been used for these functions and compare their performance on this processor with the same functions on 32-bit Intel processors.

Because of the demand for optimal performance, both the level 3 BLAS and the VML functions include extensive assembly code in them now. As the compiler becomes increasingly capable, we will place greater reliance on its optimization capabilities.

Level 3 BLAS

For obvious reasons a lot of effort has been expended on the level 3 BLAS, and DGEMM especially. For DGEMM, the theoretical peak performance approaches 3.2 GFLOPS on an 800 MHz processor. Getting that performance requires management of the multiple levels of cache, including effective use of the data prefetch instructions [10]. DGEMM performs the operation $\mathbf{C} = \alpha\mathbf{AB} + \beta\mathbf{C}$ where $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are matrices and α and β are scalars. A 4x4 block of $\mathbf{A}$ is multiplied by a 4x3 block of $\mathbf{B}$. During each clock, two multiply/accumulate operations are possible. The total number of cycles for this inner loop, fully unrolled, is 4x4x3/2, or 24 instruction groups and clocks.

Predication is often presented in the context of a means to eliminate branches within code. A pair of predicate registers are set by some condition that resolves to logically true or false. Afterwards these registers can be used to nullify or effect an operation by preceding the operation with the predicate. Predication can be used for a number of control operations within a loop, as in the case of this kernel, for such operations as:

- Reset registers to control loop execution
- Store the $\mathbf{C}$ block
- Load the next $\mathbf{C}$ block
- Load the next $\mathbf{A}$ and $\mathbf{B}$ blocks.

What looks like a single loop is in fact a triply nested loop. The innermost loop is fully unrolled. Predication controls the loop variables, moving the kernel multiply operation over a larger block structure. In addition, predication, along with other features of the architecture, allows the loop to do its own prologue and epilogue, thus holding down the size of the object file and making loops profitable even for small loop counts. A typical operation to set a predicate register is:

```
cmp.eq  p1, p0 = 1, count
```

Here, if `count` = 1, `p1` would get a 1 and `p0` a 0.

[5] `http://www.netlib.org/blas/index.html`

Matrix Size	MFLOPS
32	155.7
64	2191.1
100	1927.6
128	2555.0
200	2295.1
256	2568.8
300	2483.1
400	2475.2
500	2490.8
600	2437.9
700	2489.3
800	2515.4
900	2578.5
1000	2596.6

Table 4: `DGEMM` performance on a 800 MHz Itanium™ processor

The architecture of the processor is deeply pipelined. To support software pipelining [9] the registers (general, floating-point and predicate) can be rotated, i.e., a set of registers can be identified to the processor as belonging to a ring. Upon issuing certain branch or exit instructions the registers are incremented, modulo the number of registers in the set. This mechanism, along with the predicate registers, provides support for folding prologue and epilogue into the loop structure.

Data prefetch is used throughout the code to move data to registers from cache. Since the architecture of the processor is fully exposed, the programmer, or compiler, must make a decision about where the data is to determine what depth of memory pipelining should be used. In the case of DGEMM, the assumption is the data is in the level 3 cache with a 24 clock latency.

The use of these features permits the code to use both fused multiply-add units in every instruction for a total of 96 floating-point operations per iteration of the loop. On an 800 MHz system, the performance on DGEMM is shown in Table 4.

On the current implementation, efficiencies in excess of 80% of peak are achieved. We expect that level of efficiency to increase with the next member of the Itanium processor family for reasons discussed in Section 6.

VML

The second example is that of the vectorized transcendental functions. These functions represent vectorizations of most of the libm functions. MKL contains versions of these functions for all Intel processors. We will see how the arithmetic architecture of the Itanium processor affects performance *vis-à-vis* the other processors from Intel.

Vectorization of these functions (trigonometric, exponential, hyperbolic, etc) allows the machine to work on evaluating the function on several input values simultaneously, thus giving the opportunity to keep the arithmetic units productive in every clock cycle.

VML Function	Itanium Processor	Pentium III Processor
vdInv	4.3	11.5
vdSqrt	7.3	33.5
vdInvSqrt	6.2	31.8
vdExp	6.2	37.8
vdSin	9.3	49.9
vdTan	12.3	75.9

Table 5: Comparing VML performance (clocks/element) between the Itanium and Pentium III processors

Multiple arithmetic units already help ensure good performance on these functions. However, the extended precision and reduced rounding errors of the fused multiply-add units further enhance performance because full accuracy is much easier to maintain, reducing or eliminating costly steps needed on the other processors to maintain high accuracy over more rounding steps.

On the 32-bit processors the VML functions have 1 ulp (units in the last place) error limits. However, on the Itanium processor, those same functions have errors rivaling those of the scalar libm, or about $\frac{1}{2}$ ulp, because of the arithmetic behavior of the fused multiply-add units.

The VML functions assume data is in cache. The dominant task, as with the scalar functions, becomes scheduling the arithmetic units. Table 5 compares maximum performance between the Itanium and Pentium® III processors. (Pentium 4 processor optimizations, using SSE-2 instructions, are not complete so are not included here.)

6. FUTURE ITANIUM PROCESSOR GENERATIONS

The second member of the Itanium processor family, codename McKinley, was demonstrated at the Intel® Developer Forum in February 2001. We will briefly cover its enhancements relative to the Itanium processor, most of which will impact all of the functions discussed in this paper. In general, the architectural tunings improve performance by increasing clock speed, decreasing most latencies, improving memory bandwidth and adding more integer units which support additional memory accesses during each clock.

Frequency

The frequency of the processor increases from 800 MHz to the GHz range. Most of the software discussed in this paper should see an almost linear boost in performance from this frequency increase.

Bus

The system bus increases from 64 bits to 128 bits, and bus bandwidth increases from 2.1 GBytes/s to 6.4 GBytes/s. This added bandwidth will improve performance on vector operations (level 1 and level 2 BLAS, for instance) and improve scaling on multithreaded code.

Caches

The L1 cache does not change in size, but the latencies of L1, L2 and L3 caches are all reduced compared with the Itanium processor. The L3 cache moves on-die, is 12-way set-associative (versus 4-way now), and is non-blocking, so it supports out-of-order reads/writes. Bandwidth increases from 11.7 GBytes/s to 32 GBytes/s.

Execution units

The number of integer execution units increases from 4 to 6. These additional units support the increased memory access capability, supporting two reads *and* two writes per clock.

Other

There are numerous other enhancements to the architecture including: revamped branch prediction, reduced branch prediction penalties, enhanced prefetching, including streaming prefetch, improved TLB (translation look-aside buffer) and hardware page walker which will also improve performance on scientific and engineering applications.

7. CONCLUSIONS

We have presented some features of the Itanium architecture that impact on technical computation. The arithmetic behavior of the fused multiply-add units improves performance by permitting substantial freedom in order of operations for both libm and VML. Features such as predication and rotating registers contribute to improved performance by reducing code size and making loops profitable even for small trip counts. Loop overhead is minimized by predication, as multiply nested loops are incorporated into a single loop with predication affecting when registers are reset. Compiler development has been underway in concert with the development of the processor to provide highly effective compilation. The features of the architecture provide the resources needed to build powerful, highly optimizing compilers. We expect to rely increasingly on the compiler to provide performance approaching that which we have accomplished in the examples presented in the paper. The McKinley processor has many improvements in memory performance, cache size, structure and bandwidth, memory accesses per instruction, clock frequency, and latencies that will, in both obvious more subtle ways, improve performance.

8. REFERENCES

[1] G. Doshi, R. Krishnaiyer, and K. Muthukumar. Optimizing software data prefetches with rotating registers. In *Proceedings of PACT 2001*, 2001.

[2] C. Dulong. The IA-64 architecture at work. *IEEE Computer*, 64(7):24–32, July 1998.

[3] C. Dulong, R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, J. Ng, and D. Sehr. An overview of the Intel IA-64 compiler. *Intel Technology Journal*, 1999-Q4:1–15, 1999. Available on the Web as `http://developer.intel.com/technology/itj/q41999/articles/art_1.htm`.

[4] J. Harrison, T. Kubaska, S. Story, and P. Tang. The computation of transcendental functions on the IA-64 architecture. *Intel Technology Journal*, 1999-Q4:1–7, 1999. Available on the Web as `http://developer.intel.com/technology/itj/q41999/articles/art_5.htm`.

[5] IEEE. Standard for binary floating point arithmetic. ANSI/IEEE Standard 754-1985, The Institute of Electrical and Electronic Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, 1985.

[6] D. Jagger and D. Seal, editors. *ARM Architecture Reference Manual*. Addison-Wesley, 2nd edition, 2000.

[7] R. Montoye, E. Hokenek, and S. Runyon. Design of the IBM RISC System/6000 floating-point execution unit. *IBM Journal of research and development*, 34:59–70, 1990.

[8] P. T. P. Tang. Table-lookup algorithms for elementary functions and their error analysis. In P. Kornerup and D. W. Matula, editors, *Proceedings of the* 10^{th} *Symposium on Computer Arithemtic*, pages 232–236, 1991.

[9] W. Triebel. *IA-64 Architecture for Software Developers*. Intel Press, 2000.

[10] S. VanderWiel and D. Lilja. When caches aren't enough: Data prefetching techniques. *Computer*, 30(7):23–30, 1997.