Developing High Performance Applications for HP-UX on IA-64

Carol Thompson
Hewlett-Packard Company
11000 Wolfe Road, M/S 42U5
Cupertino, CA 95014
carol_thompson@hp.com

Abstract

This paper presents an overview of the unique features of IA-64 from the perspective of application development tools, as well as the key considerations for developers of performance-sensitive applications for IA-64. It describes some of the features of the HP-UX development tools for IA-64, which are designed to provide enhanced performance as well as developer productivity. Other considerations for application development are presented, such as the data models supported on HP-UX/IA-64, new optimizations and language features that have been added to HP-UX compilers for IA-64, new application profiling tools, and special support for debugging on IA-64.

1. Introduction

The IA-64 architecture is the first commercial EPIC (Explicitly Parallel Instruction Computing) architecture. It includes many architectural features not found on today's RISC processors, such as PA-RISC. Developing high performance applications for IA-64 is much like developing for PA-RISC and other RISC processors, but understanding the architectural, language and compiler features that affect performance can help maximize the delivered performance. The design of the IA-64 architecture, undertaken jointly by HP and Intel, focused specifically on the requirements for maximizing performance of compiled code, and compiler architects from each company (including the author) participated in the architecture definition.

2. Overview of IA-64 Features

Although the focus of this paper is not to present details of the IA-64 architecture [1], it is important to begin with a basic understanding of the architectural features in order to describe their impact on application development.

The IA-64 architecture was designed to allow compilers explicit control over the execution resources of the processor, in order to maximize instruction level parallelism (ILP). Instruction level parallelism is the concurrent execution of multiple instructions. Maximizing ILP reduces execution time.

The three architectural features that are most relevant to application development are speculation, predication, and explicit parallelism. These features are designed to maximize the ability of the compiler to expose, enhance and exploit instruction level parallelism.

2.1 Speculation

Speculation is the execution of an instruction, or a dependent instruction stream, before it would normally be executed in the program order specified by the application developer. There are two main forms of speculation

2.1.1 Control Speculation

The first form of speculation is control speculation. This is the execution of an instruction before all of the conditions controlling its execution have been evaluated. Consider the following example (shown in C):

```
int a,b;
extern int *p;
extern int global;
if( condition ) {
   a = global;
   b = *p + 2;
}
```

Figure 1a: Control Speculation Example

The two assignment statements in the then clause are guarded by the evaluation of the condition. If we begin evaluation of these statements before the condition has

been evaluated, this would be considered control speculation. The benefit of control speculation is that the conditionally executed code can be executed concurrently with the evaluation of the guarding condition. If the condition and/or the guarded statements are costly to execute or have long latency, executing them concurrently can significantly reduce the overall latency of the code.

In the example of figure 1, the first assignment statement in the then clause involves a load through a global variable. Because the address of the global variable is known to be valid, this load can be safely executed before the guarding condition has been evaluated. This is called safe speculation.

The second assignment statement requires a load through a pointer. In general, the compiler cannot guarantee that it contains a valid address. Execution of this load before the condition has been evaluated may cause an unexpected exception if the condition is false. This is therefore considered unsafe speculation. However, it is often specifically this type of unsafe speculation that is most desirable to exploit. The speculation support in the IA-64 architecture allows the compiler to exploit this type of speculation safely, by separating the load mechanism from the exception reporting mechanism. First, a speculative load is provided which either loads the data, if the address is valid, or sets the speculation token (NaT) for the target register if the address is not valid. Second, the speculation token is propagated through most computational instructions, so that the compiler can execute not just the load, but a stream of dependent operations, before the condition has been evaluated. Figure 1b shows the code generated for the example in Figure 1a, when control speculation has been applied.

```
        ld              a = [global]
        ld.s            t1 = [p] ;;
        add             b = t1,2
        cmp.ne.unc      p1,p0 =
                        condition,0 ;;
(p1)    chk.s           b, L1
        ...
L1:     {recovery code}
```

Figure 1b: Control Speculation Example

Once the condition has been evaluated, we can execute a check instruction (chk.s) which will branch to recovery code if there may be load faults which need to be resolved.

2.1.2 Data Speculation

The second form of speculation is data speculation. This involves early execution of a load from memory, prior to one or more stores which:

- preceded the load in original program order, and
- may possibly write to the same memory location as is read by the load.

The IA-64 architecture provides a facility to dynamically identify address conflicts, and to allow the compiler to trigger the execution of a recovery code sequence.

2.2 Predication

The next key feature for maximizing instruction level parallelism is predication. Predication is the conditional execution of an instruction based on the setting of a boolean (true or false) value contained in a predicate register. The IA-64 architecture provides 64 predicate registers which can be used to control the execution of nearly all instructions. Consider the code segment in figure 2a. When code is generated in a straightforward manner using branches, there are two branches and at least three cycles. Using predication, both assignments to x can execute in the same cycle (since both predicates will never be simultaneously true), saving two instructions and at least one execution cycle, and avoiding any risk of branch misprediction.

```
if ( a == 0 ) {
   x = 5;
} else {
   x = *p;
}
```

Figure 2a: Predication Example

```
        cmp.ne.unc      p1,p0 = a,0
(p1)    br              L1 ;;
        mov             x = 5
        br              L2 ;;
L1:     ld              x = [p]
L2:
```

Figure 2b: Generated code with branches

```
        cmp.ne.unc      p1,p2 = a,0 ;;
(p1)    mov             x = 5
(p2)    ld              x = [p]
```

Figure 2c: Generated code with predication

The value of predication is two-fold. First, predication enables the removal of branches. In a pipelined processor, a branch presents a potential disruption in the pipeline flow. The processor must predict whether the branch will be taken (if it is conditional), and where it will go (if it is indirect). If it guesses incorrectly, the pipeline must be flushed and restarted. With a deep pipeline and wide issue

bandwidth, this represents a significant loss of performance. For example, on Itanium™, a branch misprediction penalty is 9 cycles [2], representing 54 lost instruction issue opportunities. Even with sophisticated branch prediction techniques, a small percentage of mispredicted branches can translate into a significant performance cost.

2.3 Support for Software Pipelining

Software pipelining is a technique which allows the compiler to overlap the execution of multiple iterations of a loop, much as instruction pipelining in modern processors allows the overlapping of the execution of sequential instructions. On RISC processors, software pipelining generally requires significant code expansion, including setup code, unrolled loop iterations to handle register allocation, and finalization code at the end. Compilers for RISC processors generally pipeline only the simplest of loops (no control flow, single exit, counted loops) in order to keep the complexity manageable. Further, the benefits are greatly diminished for small loop counts, due to the large overhead.

The IA-64 architecture provides special branches, along with rotating registers (including predicates), which allow the compiler to generate software pipelined loops with little or no code expansion, even in the presence of control flow and non-counted loops.

2.4 Explicit Parallelism

Most modern processors utilize instruction level parallelism to maximize performance. PA-RISC processors, along with other RISC processors currently on the market, have been issuing multiple instructions per cycle for many years. In RISC processors, concurrent execution is achieved dynamically, through dependence analysis and instruction reordering. This approach has two significant disadvantages. First, the dependence analysis largely rediscovers information already known to the compiler at the time it generated the code, and utilizes precious processor resources to accomplish this. Second, the dependence analysis and reordering are limited in scope to a fairly small window of code.

The IA-64 architecture allows the compiler to communicate dependence information to the hardware, through explicit S bits (stops) between instruction groups. Where the compiler is unable to resolve dependence information that can only be known at execution time (such as whether two pointers actually point to the same memory location), it can utilize the control and data speculation features of the architecture to increase ILP.

2.5 Explicit Control of the Memory Hierarchy

As memory latencies continue to increase relative to the processor clock speed, memory optimizations play an increasingly critical role in maximizing application performance. Many RISC processors today offer instructions that allow applications more control of the memory hierarchy. For example, the PA-RISC 2.0 architecture provides data prefetch instructions to reduce memory latency effects. The IA-64 architecture expands upon this, providing prefetch, load and store instructions with the ability to specify hints about the expected locality and/or where in the memory hierarchy (i.e. what level of the cache) the data should reside.

In a multi-level cache hierarchy such as that in the IA-64 Itanium processor, it may be desirable to specify that certain data items (such as a very large array with little locality) remain at a cache level further from the processor. Facilities are provided to prefetch such data, so that the memory latency can be overlapped with previous computation, while not displacing the entire L1 and/or L2 cache.

3. Optimizations for IA-64

Compilers for IA-64 build on the optimization technology that has been developed for RISC architectures such as PA-RISC. One optimizing transformation which is most critical to RISC performance is instruction scheduling. This allows RISC compilers to exploit instruction level parallelism on RISC processors.

On IA-64, this key optimization become even more significant, serving as the foundation for taking advantage of key architecture features such as predication, speculation, and rotating registers.

In the HP-UX compilers for IA-64, the code being compiled is divided into regions, which form the unit of operation for instruction scheduling. Speculation and predication are applied within these regions. Code for an entire region is scheduled as a unit, enabling code to be scheduled as efficiently as possible, increasing instruction level parallelism and reducing computational latency. Where possible, given reasonable constraints on the time consumed by the compiler, loop bodies are fully encompassed in a single region, allowing software pipelining of the loop. Judicious region selection is extremely important for generating optimal code.

Data prefetching is performed on loops. Where the compiler is able to discern an array reference pattern, it will emit appropriate data prefetch instructions, so that the data will be available for computation in the appropriate iteration.

4. What's different about developing for IA-64

From a high level, developing for IA-64 is no different than developing for any other architecture. However, the evolution of computer architectures from CISC to RISC has already influenced application development in the following ways:

- Optimization has become increasingly critical for application performance. Profile-based optimization [4], introduced for RISC, has an even larger performance impact on IA-64.
- Assembly code is diminishing in prevalence, due to increasing sophistication of RISC compilers, and increasing complexity of RISC processors.

IA-64 pushes these trends even further. Because the instruction level parallelism on IA-64 is explicit, the role of the compiler is critically important in delivering application performance.

Furthermore, assembly programming, already rendered complex by the introduction of RISC features such as delayed branching and exposed latencies, becomes even more challenging with the introduction of architectural features such as predication, speculation, and explicit parallelism.

In short, from an application development perspective, IA-64 merely continues the trends already in place for RISC processors.

5. Performance Tuning

Tuning an application for IA-64 is very much like tuning it for any other processor. The most important factor in application performance is the design and implementation of the core algorithms and data structures. Nearly any tuning exercise which improves the efficiency of these fundamental application components will provide benefits on IA-64 as well as RISC platforms.

Just as each implementation of a RISC architecture has unique performance characteristics, there are specific charateristics of the IA-64 Itanium processor which may or may not apply to future processors. For Itanium, data structure efficiency is a key consideration.

6. Application Profiling

The first step in performance tuning is measurement. HP provides the following tools to assist in the performance analysis phase of application development:

- HP Caliper is a suite of performance tools, newly developed for IA-64, which implement a number of different application profiling techniques.
- CXperf is a tool currently available on PA-RISC, and which will provide performance analysis on IA-64 as well.

With these tools, the application developer can characterize performance, and identify opportunities for tuning. For example, if the HP Caliper data indicates that data cache misses account for a significant percentage of application execution time, it may be profitable to spend some time tuning the application's data structures.

6.1 HP Caliper

The HP Caliper suite of performance analysis tools provide access to several types of performance data:

- HP Caliper/PMU provides access to information collected by the Performance Monitoring Unit (PMU) on IA-64 [ref]. This includes cache and TLB misses, branch misprediction rates, and pipeline stalls.
- HP Caliper/PBO provides profiling information indicating the execution frequencies for control transfers (branches and calls) in the application.
- HP Caliper/Gprof provides profiling information in the style of gprof, the standard Unix® profiling tool. This provides information about which functions in an application account for the most execution time. Unlike gprof, however, it supports multiple shared libraries and forks.

The HP Caliper tool suite provides for ease of use and low overhead, through the use of dynamic translation and sampling technologies. They operate on regular executable files, and do not require the use of special compiler options.

At first release, these tools will be invoked through a command line interface, and will provide a graphical viewing tool for a visual presentation of the profile data.

In a future release, an interactive graphical user interface will be added, along with additional features for detecting program correctness flaws as well as performance opportunities.

6.2 CXperf

CXperf is an interactive runtime performance analyzer, which supports both scalar and parallel application development. Metrics are collected on a per-thread basis for execution time, cache, TLB and paging data, process migration, call counts and call graphs. CXperf is able to report performance information relative to specific loops

in the program, and is especially well suited to loop-intensive applications. On IA-64, CXperf does not require that the application be built with special compiler options.

6.3 Profiling for the Compiler

As described in section 3, application profile data is extremely valuable to optimizing compilers. On PA-RISC, the HP-UX compilers provide the capability to generate an instrumented executable which, when run, will produce an execution profile. The application developer uses this instrumented executable to run the application using a set of representative input data. This profile information is subsequently used by the compiler to determine where and how to apply optimizing transformations.

On PA-RISC, profile-based optimization delivers performance improvements in excess of 20% for real-world applications. However, achieving this performance requires a two-step build process, encompassing a special instrumented build, producing a special executable used only for profiling, followed by an optimizing build which utilizes the profile data.

On IA-64, profile data is even more important to the compiler. Many compiler decisions are enhanced by knowledge of the execution behavior of the application:

- Many optimizing transformations are performed on code regions. For best performance, it is important that these regions be selected to minimize region crossings within high frequency execution paths.
- Determining which instructions within a region to speculate or predicate is more effective when relative execution frequencies are know.
- The effectiveness of loop optimizations, such as unrolling and prefetching, can be enhanced by knowledge of average loop behavior.

In order to make the benefits of profiling accessible to more applications, HP has introduced significant usability improvements in the profile-based optimization support for IA-64. It is not necessary on IA-64 to do a separate instrumented (+I) build of the application in order to do profiling for feedback into the compiler. HP Caliper/PBO operates on an existing debuggable executable file, and produces a profile data file that can be utilized by the compiler in a subsequent profile-based compilation (using the +P option). That same profile data file can be viewed using the graphical presentation facilities of HP Caliper/PBO, providing useful feedback to the developer on application behavior.

6.3.1 Profile Options and Pragmas

Obtaining a fully representative profile data file is not always possible, for the following reasons:

- Representative input data sets may not be readily available
- Application or system configurations representative of all customer usage profiles may not be practical to duplicate

In these cases, the application developer may yet have specific knowledge of the branching behavior for the most critical execution paths. This information can be communicated to the compiler through special profiling options and pragmas:

*+Ofrequently_called=name[,name]**
+Ofrequently_called:filename
*+Orarely_called=name[,name]**
+Orarely_called:filename

These options indicate functions that are frequently or rarely called. They take as arguments either a list of function names, or the name of a file containing a list of function names. This information is useful to the compiler in making inlining decisions, and in reasoning about the execution frequency of code regions containing calls to these functions.

*#pragma frequently_called name[,name]**
*#pragma rarely_called name[,name]**

These pragmas are analogous to the options of the same name.

#pragma estimated_frequency f

This block-scoped pragma indicates the estimated relative execution frequency of the current block as compared with the immediately surrounding block. This may be used to indicate the average trip count in the body of a for loop, or to indicate the fraction of time a then clause is executed. The frequency, *f*, may be expressed as a floating point constant. The code in figure 4 illustrates the use of the estimated_frequency pragma:

```
if ( condition ) {
   #pragma estimated_frequency 0.8
   ...
   for ( ... ) {
     #pragma estimated_frequency 4.0
     ...
   }
} else {
   ...
}
```

Figure 4: Estimated_frequency pragma

In this example, the code in the then-clause of the if statement is expected to execute 80% of the time (implying that the else clause is executed only 20% of the time). The loop is expected to execute, on average 4 iterations. The compiler can utilize the information to guide its optimizations, such as giving precedence to speculating code from the then clause above the evaluation of the guarding condition. Knowledge of the average loop iteration count might cause the compiler, in this case, to determine that data prefetching would not be effective.

7. Platform-Specific Tuning

Itanium is the first of many IA-64 processors. Each will have unique characteristics for which code can be optimized. On PA-RISC, the HP-UX compilers offer two options to specify the target processor. One option, +DA, indicates the architecture version (1.0, 1.1 or 2.0) to be used. This option controls the processors on which the code will run correctly. The second option, +DS, specifies the processors for which the code should be optimized. This option affects only performance.

On IA-64, there is a single architecture version, and currently a single available processor model. However, HP's IA-64 compilers are already designed to generate code designed to run well on multiple target processors. This is the default code generation strategy. The +DSitanium option generates code specifically optimized for the Itanium processor, and future options will be provided as new processors are released.

8. Application-Specific Tuning

Certain application characteristics have a significant impact on performance. Some of these are covered in this section.

8.1 Memory Ordering Considerations

The C and C++ programming languages offer a fairly simplistic view of memory ordering constraints. Memory references are generally subject to optimizations such as dead or redundant code elimination, loop invariant code motion, coalescing of multiple loads or stores, etc. Some applications, however, have specific constraints for certain memory references:

- Multi-threaded applications must exercise care with regard to shared memory.
- Applications, such as device drivers writing to memory-mapped I/O, may require that those memory references remain untouched by optimization
- Some applications may rely on the value of certain memory locations in signal handlers, or after a return from a `longjmp`().

Generally, in these cases, the application developer must declare such variables using the `volatile` type specifier. This indicates to the compiler that references to these variables must not be subject to optimization. However, the semantics of this specifier are by necessity overloaded to handle all of the above situations. As a result, the compiler must inhibit all optimizations to volatile memory locations, even if the application doesn't require all of the constraints.

In order to minimize the performance impact of volatile variables, the HP-UX C compiler has introduced new type qualifier extensions to enable more efficient compiler support for `volatile` data types. They are:

__unordered
__synchronous
__non_sequential
__side_effect_free

One or more of these type qualifiers may be used with a volatile keyword in a type declaration. Their semantics are as follows:

__unordered	References to this variable need not be explicitly ordered relative to other memory references, either within the same or different threads.
__synchronous	All references to this variable are synchronous with respect to the current thread of execution (i.e. the location will not be read or written by signal handlers or other threads).
__non_sequential	Memory references to this variable may be re-ordered relative to other non-sequential memory references.
__side_effect_free	Loads of this variable do not have side effects (such as memory mapped I/O). The compiler may issue prefetches or speculative loads of these variables.

These type specifiers were designed with the needs of the HP-UX operating system in mind, and they can be useful for optimizing the performance of any similar code with memory ordering constraints.

8.2 C99 Language Extensions

The HP-UX C compiler supports several features that are part of the new C99 language definition [3]:

- Complex and imaginary data types, in <complex.h>

- Support for C99 floating point hexadecimal constants, including printf/scanf support using %A and %a.
- C99 math function specialization.
- Floating Point Pragmas:
  - STDC FP_CONTRACT: enables or disables contraction of floating point expressions. Contraction can reduce rounding error, and can improve efficiency, as when the combined multiply and add instruction (fma) is used. Contraction is enabled by default.
  - STDC FENV_ACCESS: Informs the compiler whether or not the application will not access the floating point status flags, or modify the default floating point evaluation modes. It is off by default.
  - STDC CX_LIMITED_RANGE: When enabled, allows the compiler to use the usual algorithms for complex multiply, divide and absolute value, which may compromise treatment of infinities, overflow and underflow.
- A limited implementation of the `restrict` keyword.

8.2.1 Restrict keyword

The `restrict` keyword can be used to indicate pointers which do not alias with other pointers. The HP-UX C compiler on IA-64 provides support for the use of this keyword on parameter declarations. Figure 5 shows an example of the use of the `restrict` keyword, indicating that the `s1` and `s2` pointers do not alias with each other, or with other pointers, but that no assertion is made about `other_pointer`.

```
foo ( restrict char *s1,
      restrict char *s2,
      char  *other_pointer )
{
  ...
}
```

Figure 5: C99 restrict keyword

The restrict keyword is similar in use to the +Onoparmsoverlap option, but is more powerful, in that use of the latter requires that all pointer parameters be distinct.

8.3 Floating Point Applications

The HP-UX compilers for IA-64 have a number of features designed for fine-tuning the performance of floating point applications.

8.3.1 Floating Point Evaluation Mode

The HP-UX compilers provide an option to specify the width of evaluation for floating point computation:

-fpeval=[float|double|float80]

This option indicates the minimum precision under which floating point expression evaluation will occur. This option also affects the evaluation width for C99 complex and imaginary types. The default for C++ and for C with -Aa or -Ae is -fpeval=float. The default for C with -Ac is -fpeval=double.

8.3.2 Accuracy, Precision and Exception Behavior

The HP-UX C and C++ compilers support options which give the user control over the accuracy, precision and exception behavior of floating point computations.

+O[no]cxlimitedrange	This option provides equivalent functionality to the STDC CX_LIMITED_RANGE pragma (C99), but applies to the compilation unit. Default is +Onocxlimitedrange.
+Ofltacc=strict	Disallows any floating point optimization that may change result values.
+Ofltacc=default	Allows contractions (e.g. fused multiply and add), as with the C99 pragma FP_CONTRACT ON, but disallows any other floating point optimization that may change results. As implied, this is the default.
+Ofltacc=limited	Like default, but also allows floating point optimizations (such as substitution of 0.0 for x*0.0) which may affect the generation and propagation of infinities, NaNs, and the sign of zero. Also implies +Ocxlimitedrange.
+Ofltacc=relaxed	In addition to “limited” behavior, also allows floating point optimizations (such as reordering of expressions, even if parenthesized) that may change rounding error. Also implies +Ocxlimitedrange.

+O[no]fenvaccess	+Ofenvaccess disallows any optimizations which might affect behavior under non default floating point modes (e.g. alternate rounding directions or trap enables) or where floating point exception flags are queried. It is equivalent to placing a C99 FENV_ACCESS ON pragma at the beginning of the file. Default is +Onofenvaccess

8.4 Parallel Programming

HP-UX Fortran compilers provide multiple means for specifying parallel constructs.

8.4.1 Fortran 95

The HP-UX Fortran compiler for IA-64 provides full support for the Fortran 95 programming language standard.

8.4.2 OpenMP

HP's Fortran compiler provides full support for the OpenMP programming model.

8.4.3 HP Parallel Directives

The HP parallel directives continue to be supported on IA-64, as they were on PA-RISC.

8.5 Inline Assembly

The IA-64 architecture has been explicitly designed to support high-level language compilers. However, there are a number of instructions, such as memory hierarchy management, synchronization, and specialized instructions such as popcnt, which cannot easily be specified through standard high-level language constructs.

Many compilers support an inline assembly directive to provide access to target instructions. This is often implemented using call syntax to an `asm` function that accepts a string argument. This string is parsed as assembly language. The limitations of this approach are that the programmer must have knowledge of the available registers, and optimization must be quite conservative.

The HP-UX C compiler provides inline assembly support that is fully integrated into the optimizing compiler. Many IA-64 instructions are supported, and the programmer can use regular C expressions for the operands. For example, the following internally defined function generates a popcnt instruction with the given 64-bit unsigned integer argument, and returns the result in a 64-bit unsigned integer:

```
uint64_t _Asm_popcnt (uint64_t r3);
```

9. Data models on HP-UX/IA-64

Like PA-RISC, the IA-64 architecture provides full support for both 32-bit and 64-bit addressing and arithmetic. In addition, the HP-UX operating system provides support for both data models. Therefore, as on PA-RISC, the application developer has a choice of data model. The default data model is 32-bit. The data model may be specified using the +DD32 or +DD64 option, to select the 32 or 64 bit addressing model, respectively.

In addition to selecting the size of data addresses, the data model also affects the size of other fundamental data types. Table 1 shows the data sizes for the two available data models in HP C and C++. These data sizes were selected for ease of portability from the 32-bit to the 64-bit data model, and for compatibility among vendors.

	+DD32	+DD64
char	8	8
short	16	16
int	32	32
long	**32**	**64**
long long	64	64
pointer	**32**	**64**

Table 1: Size in bits of fundamental data types in the two available data models

The considerations in selecting which data model to use include:

- Data size requirements of the application. Does the application need access to more data than can be addressed with a 32-bit pointer?
- Performance considerations. The use of 64-bit pointers can significantly increase the data working set of the application, resulting in an increase in data cache misses, and reduced overall performance.
- External dependencies. If the application depends on libraries or other in-process components developed elsewhere, they must share a common data model.

In general, if the application's data requirements and external dependencies do not compel it to move to the 64-bit addressing model, it is most beneficial to continue to use 32-bit addressing.

10. Application Structure and Procedure Linkage

The structure of an application has a major impact on its performance. Structural boundaries, such as between procedures, compilation units, and shared libraries, impose limits on the scope, type and quality of optimizations that can be performed. Furthermore, procedure linkage costs themselves can impact performance.

10.1 Performance Implications of Program Structure

Within a process, a procedure call may or may not cross load module boundaries (a load module being either an executable file or shared library). If the call crosses a load module boundary, it must go indirect through a linkage table. In addition, the global data pointer (gp) must be saved and restored around the call, as it is unique for each shared library. However, even within the same load module, the linkage may be indirect, if the reference cannot be resolved at link time. For a call-intensive application, this overhead may be significant.

Similarly, when a global data item is referenced, if it is not defined in the same compilation unit as the reference, it may be accessed indirectly through the linkage table.

The HP-UX development environment is designed to provide full support for shared libraries. When a source file is compiled, the compiler will assume that the resulting object file may be included in a shared library, unless told otherwise. Furthermore, any references to symbols not defined in the same compilation unit, will be assumed to potentially reside in another load module. This results in code generation that is less than optimal for the case in which the symbols both reside in the same load module.

There are four binding modes for data and code symbols:

- By default, a symbol is presumed to be probably, but not definitely, defined in the same load module. These references incur an additional cost over direct local references.
- A protected symbol is one which will be defined in the current load module, and will not be preempted by another symbol of the same name in a different load module. References to protected symbols are more efficient than to default or external symbols.
- A hidden symbol is one which will be defined in the current load module, and which will not be visible to other load modules. References to hidden symbols are more efficient than to default or external symbols.
- An external symbol is one that is presumed to be defined in a different load module. The linkage table reference is generated directly by the compiler, incurring further overhead over the default case if the symbol is not actually external to the load module, but reducing the overhead over the default case if it is indeed external.

The HP-UX compilers provide options to allow the developer to specify information about the symbol binding behavior of the application. Each option takes an argument which is either a list of symbols, or a file containing a symbol list. When specified with no argument, they apply to all symbols:

-Bprotected[=*symbol*[,*symbol*]*]
-Bprotected:*filename*
-Bprotected_def
    -Bprotected_def implies that only the symbols defined within the current translation unit should have protected binding mode.
-Bhidden[=*symbol*[,*symbol*]*]
-Bhidden:*filename*
-Bdefault=*symbol*[,*symbol*]*
-Bdefault:*filename*
    The -Bdefault options are useful to specify exceptions to blanket -B options, such as -Bhidden with no symbols specified.
-Bextern[=*symbol*[,*symbol*]*]
-Bextern:*filename*

These options can be useful for optimizing function call and data reference overhead.

10.2 Optimization and Program Structure

Ideally, high frequency execution paths should be contained within a single procedure. This makes it possible for the compiler to optimize the entire execution path at the default level of optimization (-O; equivalent to +O2).

At optimization levels of +O3 and higher, the compiler optimizes across procedure boundaries, within the scope of a compilation unit (source file). At this level, the compiler can optimize entire execution paths that are fully contained within a source file. Inlining across procedure boundaries can eliminate procedure linkage cost, and more importantly expose larger code sequences for optimization, resulting in additional optimization opportunities and higher instruction level parallelism.

With the +O4 option, the compiler can optimize across an entire load module. This maximizes the scope

available for optimization, resulting in the highest level of performance.

11. Global Data

Global data is referenced from a global pointer (gp). Global data items of size 8 bytes and smaller are allocated next to gp, with larger data objects allocated next. The compiler assumes that there will be no more than 4 megabytes of small data items, and will use a shorter code sequence to reference them. Larger objects are referenced using a slightly more costly code sequence. The +Oshortdata=n option can be used to indicate that data items of size n bytes and smaller (n greater than 8) should be placed in the short data area, and referenced with the more efficient code sequence. If no value of n is given (+Oshortdata), all data items are allocated in the short data area.

12. Debugging

The IA-64 architecture is highly dependent on the compiler for runtime performance. For that reason, the HP-UX IA-64 compilers perform limited code optimizations even in the absence of optimization options. This level of optimization is fully compatible with the -g (debugging) option, with some minimal limitations on modifiability of user variables. Variables can generally be modified at procedure boundaries, and immediately after they have been set. The debugger will issue a warning if the user attempts to modify a variable at a code location where it is not supported. Aside from variable modification, there is no other impact on debugging, as all user-visible state is updated in original program order.

When optimization is enabled (-O), the compiler provides source location information to enable the developer to do rough navigation at the source level. Future releases will include additional support for debugging of optimized code.

13. Conclusion

While the IA-64 architecture provides many new features for enhancing application performance, developing and porting applications to IA-64 is not a great deal different from developing for today's RISC processors. However, the importance of tuning for performance, and taking full advantage of the optimization features available, is increasing. Performance can be maximized through the use of performance analysis tools, as well as language features and options that address the specific requirements and characteristics of the application.

References

[1] Intel Corp., "IA-64 Application Developer's Architecture Guide," http://developer.intel.com/design/ia64/downloads/adag.htm, 1999.

[2] Intel Corp., "Itanium Processor Microarchitecture Reference for Software Optimization", Order number 245473-001, http://developer.intel.com/design/ia-64/downloads/245473.htm, March 2000.

[3] Programming Language C, ISO/IEC 9899:1999.

[4] Pettis, K. and Hansen, R.C., "Profile Guided Code Positioning," *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, Vol 25, No. 6, June 1990.

[5] HP Developer's Resource website: http://devresource.hp.com.