Enabling Optimizations to Achieve Higher Performance on the HP PA-RISC Architecture

Version 1.0

HEWLETT PACKARD

California Language Laboratroy
11000 Wolfe Road
Cupertino, California 95014
Last Printing: October 8, 1997

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with furnishing, performance, or use of this material. Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

1.0 Introduction

The HP PA-RISC architecture is designed to deliver industry-leading performance on today's commercial and technical applications. Users can enhance the performance of their applications on the PA-RISC architecture by adopting strategies that go beyond merely compiling with the –O default optimization option. In this regard, it is important to understand the advanced performance enhancing features of HP compilers to achieve the highest possible performance. This document discusses factors that have traditionally limited application performance and motivates the use of advanced HP compiler functionality to address those limitations. Some guidelines on coding strategies to facilitate compiler optimizations are also presented.

Many of the performance-enhancing compiler features that will be discussed in this document are already available in HP's compilers for the PA-RISC architecture. Users are encouraged to use these features on current PA-RISC systems not only to achieve higher performance on PA-RISC systems, but also to be better positioned to exploit the full performance potential afforded by the next generation HP/Intel Enhanced Mode architecture (which will be referred to as the EM architecture in this document). The EM architecture is an advanced computer architecture that provides a number of unique features for achieving significantly higher performance. HP is developing advanced compiler optimization technology for hardware implementations of this architecture.

This document is organized into sections that discuss the following topics:

- The factors that affect application performance and the impact of compiler optimizations on performance.
- The optimization levels supported by HP's compilers and a brief description of the functionality provided at each optimization level.
- The factors that can limit the degree of optimization effectiveness and suggestions on how to improve optimization effectiveness.
- A summary of the recommendations on enabling compiler optimizations to maximize application performance on the PA-RISC architecture.

Much of the discussion in the current version of this document is focused on enabling optimizations to improve the performance of CPU-intensive *integer* applications.

2.0 Performance Factors and Compiler Optimizations

The execution time of any application can be characterized by the following equation:

Execution Time = (Number of Machine Instructions Executed) ×
(Average Number of Clock Cycles Required to
Execute a Machine Instruction) ×
(Clock Cycle Time)

Since performance is inversely proportional to execution time, higher performance is achieved by executing fewer instructions, spending fewer machines cycles to execute each instruction on average, and/or by increasing the clock frequency.

It is well known that algorithmic improvements are a primary means of enhancing application performance since they can often significantly reduce the number of instructions executed. In this document we'll assume that applicable algorithmic improvements have already been made. Instead, we will focus on further means to boost performance on a PA-RISC architecture system at any given clock frequency. In particular, we'll examine how compiler optimizations can help reduce both the number of machine instructions executed as well as the average number of cycles required per instruction to deliver higher performance.

In the past, optimizing compilers tended to focus principally on reducing the number of instructions executed by an application. However, for most current processors, and for the PA-RISC architecture in particular, the compiler plays a significant role in reducing the average number of cycles per instruction, or **CPI** for short. HP's compilers for the PA-RISC architecture implement a number of optimizations to reduce the overall CPI. It is important to understand the factors that determine the CPI and how compiler optimizations help improve this important performance metric.

2.1 Instruction-level Parallelism

The more instructions that are executed in parallel in each clock cycle on average, the lower the CPI. PA-RISC architecture systems support a very high degree of instruction-level parallelism. However, the compiler must determine the optimal machine instruction sequence to fully exploit the instruction-level parallelism provided by the hardware. Specifically, the hardware can execute multiple instructions in the same clock cycle only if there are no operand dependencies between them. It is the compiler's responsibility to identify and group independent instructions so that they may be executed concurrently and thus lower the overall CPI.

2.2 Pipeline stalls

Modern processors typically execute instructions in multiple pipelined stages. Each instruction's execution proceeds from one pipeline stage to the next at every clock cycle. Multiple instructions may be executing concurrently in different pipeline stages. However, if an instruction is unable to proceed from one pipeline stage to the next, instructions in earlier pipeline stages are also prevented from proceeding, resulting in a *pipeline stall*. Pipeline stalls typically occur if the machine resources or data values required by an instruction are not yet available. For instance, if instruction B uses the result computed by instruction A, and if instruction A has not yet finished computing its result, instruction B will be unable to start computing its result and hence cause a pipeline stall.

Pipeline stalls increase the average number of cycles required to execute each instruction. The more often the processor pipeline is stalled due to data dependencies, the higher the CPI. For example, loading a data value into a register typically takes two cycles to execute; therefore, if an instruction that reads the contents of the loaded register immediately follows the load instruction, the processor is stalled for one cycle, resulting in a loss of performance. However, the compiler can try to place other instructions between the load operation and the instruction that uses the loaded value to avoid the pipeline stall.

2.3 Branch prediction

Source-code if-statements typically get translated into branch instructions. A branch instruction may alter the flow of control and cause a disruption in pipelined instruction processing. When a branch instruction enters the processor pipeline, the processor would ordinarily have to wait until the branch instruction progresses to a later pipeline stage, when the outcome of the branch is known, before it can determine which instructions to fetch into the pipeline next. Clearly, with this simplistic strategy, portions of the processor pipeline would remain idle during the execution of branch instructions. Moreover, the performance impact can be quite significant, since branch instructions tend to occur rather frequently in typical applications.

To accelerate branch execution, modern processors use special mechanisms to predict the outcomes of branches in advance. This allows instructions to be fetched and executed from the predicted branch destination *speculatively*. The predicted branch outcome is later compared against the actual outcome once it is known. If the prediction is correct, then the pipeline disruption is effectively avoided. Otherwise, the speculatively executed instructions have to be removed from the pipeline and execution is restarted at the correct branch destination.

Accurate branch prediction is becoming an increasingly important factor for high performance since mispredicted branches can significantly increase the CPI. Two types of branches account for most of the branch misprediction penalties:

- ❑ Conditional branch — because the result of the branch condition is not known until later, as discussed above.
- ❑ Indirect branch — because the target of the indirect branch may not be known until later. Primary sources of indirect branches are procedure returns, virtual function calls in C++, indirect calls using function pointers, and switch statements.

PA-RISC hardware systems incorporate sophisticated branch prediction mechanisms to enhance branch prediction accuracy. In addition, compilers for PA-RISC systems can significantly improve the percentage of branches that are correctly predicted through the following techniques:

- ❑ Giving hints to the hardware about the most likely outcome of a conditional branch using architected branch hinting mechanisms.
- ❑ Giving hints to the hardware about the likely target of an indirect branch instruction.
- ❑ Reducing the number of branches executed by eliminating branches. For example, compilers can eliminate branches resulting from procedure calls and returns through *procedure inlining*.

2.4 Instruction and data cache miss rate

Most modern computer systems use instruction and data caches to bridge the gap between processor and memory speeds. If a needed instruction or data item is available in the cache, the processor does not have to wait for it to be retrieved from main memory. However, if the next instruction is not in the instruction cache (i-cache) or if the data accessed by a currently executing instruction is not in the data cache (d-cache), the processor pipeline stalls until the required instruction or data item can be transferred from main memory to the cache. For high speed processors, this transfer can take many clock cycles. Therefore, the higher the i-cache and d-cache miss rates, the higher the CPI, and hence, the lower the performance.

Compilers can help improve the effectiveness of the caches on PA-RISC systems through the following techniques:

- ❑ Arrange instructions so that the most frequently used instructions are next to one another for more efficient use of the i-cache.
- ❑ Insert instructions into the code stream to *prefetch* instructions and data from memory to the caches well before they are needed and thus overlap the memory-to-cache transfer with other useful computations.
- ❑ Reorganize data intensive loop constructs to minimize cache misses.

2.5 TLB miss rate

Modern processors and operating systems support a *virtual memory* model, which allows each process to access memory independently. Each process accesses memory by specifying a *virtua*l address that needs to be converted into a *physical* memory address before the memory transfer is initiated. Processors use a structure known as a *Translation Look-aside Buffer,* or TLB for short, to cache frequently used virtual-to-physical memory address translations. The virtual-to-physical address translations are maintained on a *per-page* basis where pages are typically 4k bytes in size. The TLB is a fixed size table and therefore all mappings of virtual-to-physical addresses can not be kept in the TLB at the same time. A TLB miss occurs when the virtual-to-physical address mapping for a required instruction or data item is not in the TLB. When a TLB miss occurs, the relevant address translation is retrieved from the d-cache or main memory, which may take tens to hundreds of clock cycles. TLB misses are thus quite expensive and the higher the number of TLB misses, the higher the CPI, and hence, the lower the performance.

Compilers can help reduce TLB misses, particularly instruction TLB misses. For many applications a significant amount of time is spent executing a small portion of the total code stream. If the compiler is able to identify which sections of the code are executed the most often, it can arrange to re-position those code fragments contiguously so that they fit into a smaller number of pages. This can reduce the number of TLB entries required during the execution of the program and potentially reduce the number of TLB misses.

3.0 Optimization Levels Provided by HP Compilers

The PA-RISC compilers provide the optimization levels listed below:

Optimization Level 1 — enabled by the +O1 compiler option.

This is the lowest optimization level, where a limited set of optimizations are performed. The optimizations performed at this level include common sub-expression elimination, redundant load elimination, and peephole optimizations which are each limited in scope to small sections of procedures.

Optimization Level 2 — enabled by the +O2 compiler option.

This is the optimization level that is selected by default when you specify the –O compiler option. Users should typically see a significant performance improvement compared to level 1 or no optimization. Most of the discussion in this document, however, is focused on how to achieve performance beyond what is attained at this level.

At optimization level 2, the compiler performs a number of transformations over the entire scope of individual procedures, such as common sub-expression elimination, loop invariant code motion, peephole optimizations, and elimination of unnecessary loads and stores, to reduce the number of instructions executed by the program. Additionally, the compiler performs instruction scheduling to reduce the CPI.

Optimization Level 3 — enabled by the +O3 compiler option.

At this level, the compiler performs procedure inlining. Only those procedures within the same source file are subject to procedure inlining at this optimization level.[1] Procedure inlining reduces the number of instructions by eliminating the overhead of procedure calls and also helps reduce the CPI by eliminating branches and increasing the scope of instruction scheduling. Procedure inlining is particularly effective for applications that have many small routines which are frequently executed.

Interprocedural alias analysis is also performed for file static variables at optimization level 3. In addition, the compiler also restructures certain types of data intensive loops to improve the effectiveness of the d-cache at this level.

[1] Note that for C++ programs, the compiler tries to inline procedures identified as inlining candidates in the source code even at lower optimization levels.

Optimization Level 4 — enabled by the +O4 compiler option.

At this level, the compiler performs procedure inlining across all files compiled at +O4. Since most applications have multiple source files, it is important to use +O4 to increase the effectiveness of procedure inlining. Level 4 optimization is most effective when used in conjunction with Profile Based Optimization (PBO), which is discussed later.

Interprocedural alias analysis is also performed for global variables when the +Owhole_program_mode option is specified at optimization level 4.

In general, to achieve the best possible performance on the PA-RISC architecture, applications should be compiled at the highest acceptable level of optimization. While maximum performance is typically achieved at optimization level 4, the time taken to compile large applications at this level can be significantly greater than at lower optimization levels. Nevertheless, users can be better positioned to exploit the full performance potential of the HP PA-RISC architecture by evaluating the effectiveness of optimization level 4 for their applications and adopting it in production builds and testing processes as appropriate.

4.0 Limiters to Effective Optimizations and Solutions

When compilers perform the kinds of optimizations discussed above, they typically face a number of challenges. These challenges and limiters are discussed next, along with strategies that HP compiler users can adopt to enhance optimization effectiveness and thus achieve significantly higher performance on their applications.

4.1 Knowledge of program execution profile

When a compiler compiles a source program it does not have precise knowledge of the *program execution profile*. The program execution profile encompasses information about the typical outcomes of conditional branches, how often each procedure is called, what portions of a program are executed the most often, etc. In the absence of such knowledge, the compiler is forced to make educated guesses about the execution profile using *heuristics*. These heuristics in some cases do a reasonable job of predicting the actual execution profile, while in other cases they are not very reliable or even applicable. Here are some specific examples where the lack of execution profile information can limit optimization effectiveness in HP compilers:

1. When the compiler performs procedure inlining within a file at +O3 or across files at +O4, it uses heuristics to guide which routines should be inlined and where inlining should be performed. However, the compiler may perform more inlining than needed and in some cases even hurt performance due to the increased code size.

   If the compiler has a better idea of which routines are executed most frequently and how often a particular routine is called from a given call site, it can be more judicious in performing procedure inlining and the resulting code is likely to perform much better. For large applications (more than 100k lines), +O4 and procedure inlining are likely to be the most effective when the compiler has information about the program execution profile.

2. The compiler uses heuristics to decide which data values should be kept in registers to effectively use the available hardware registers. These heuristics work well in general. However, registers can be allocated more efficiently if the compiler knows which sections of a routine are executed most frequently. Knowledge of the execution profile can ensure that the most important variables are kept in registers and the minimum number of registers are used. Reducing the number of registers in use is quite important since it can help avoid the overhead of saving and restoring registers when the number of registers required exceeds the number registers actually available.
3. In most integer applications, typically four to ten instructions are executed before a branch instruction is encountered. This does not leave enough room for the compiler to schedule instructions to avoid pipeline stalls. The compiler needs to move instructions from the most likely destination of each branch and schedule them with instructions in the block of code preceding the branch. Without knowing the actual execution frequencies for each target of a conditional branch, the compiler has to use heuristics to predict the most likely branch destination. The compiler's branch prediction heuristics are not always accurate and can sometimes diminish the effectiveness of instruction scheduling.
4. The compiler tries to reorder the code so that the most frequently executed routines are placed together. This optimization is important for reducing instruction TLB misses as discussed earlier. However, without knowing which routines executed most frequently, it is very difficult for the compiler to lay out the code optimally.
5. The compiler tries to provide hints to the hardware for branches to reduce branch misprediction penalties. Such hints are useful only if the compiler itself can accurately predict the most likely target of a given branch.
6. When performing loop optimizations such as loop invariant code motion or loop unrolling, compilers typically assume that the loop body is executed many times. While this assumption is usually true, some loops are executed only one or two times. Loop optimizations can hurt performance if the transformed loops are not frequently executed. Armed with knowledge of how often loops are executed, the compiler can be more judicious in performing loop optimizations.

In summary, knowledge of the program execution profile would allow HP compilers to:

- ❑ Replace heuristics with more precise information to significantly improve the effectiveness of optimizations, particularly - procedure inlining, instruction scheduling, register allocation, and loop optimizations.
- ❑ Perform certain optimizations that can significantly improve the CPI. These optimizations include the generation of branch hints and code layout.

4.1.1 Profile Based Optimization (PBO)

HP's compilers for the PA-RISC architecture provide a framework called *PBO* to automatically gather and feed back information about the program execution profile to the compiler. Significant performance improvement has been achieved using PBO on PA-RISC systems.[2] PBO will be even more important for achieving the highest possible performance on the next generation HP/Intel EM architecture.

4.1.2 PBO Usage

The PBO functionality provided by HP's PA-RISC production compilers is described here. The compilers for the next generation HP/Intel EM architecture will provide an improved framework but the basic principles are likely to remain the same.

PBO can be used with any level of optimization. In other words, if profile information is available, it is used by optimizations performed at each level. A three-step procedure is used to optimize applications with PBO.

1. In the first step, the program is instrumented to collect an execution profile. In PA-RISC compilers, the +I option causes the code to be instrumented. (It is likely that this step will be optional for the HP/Intel EM architecture.)
2. The second step is to identify common input data sets for your application and to run the instrumented executable with the selected representative input data sets. A profile database is automatically generated when the program terminates and is updated for each subsequent application run.
3. The final step is to re-compile the program using the profile database as input with the appropriate level of optimization (preferably +O4). This is done on PA-RISC systems by specifying the +P compiler option. The resulting binary is highly optimized to achieve increased application performance for the representative input data sets.

PBO improves the effectiveness of many optimizations performed at +O2 and +O3 and is highly recommended for +O4 optimization. PBO also enables certain profile-dependent optimizations in the compiler that are performed only when the application is compiled with the +P option.

While PBO is very effective and safe to use, users may face the following challenges in using PBO:

- ❑ PBO generally involves some changes to the build process and *Makefiles*. However, these changes only need to be made once and the performance benefits can be reaped for years to come.
- ❑ PBO requires representative input data sets for gathering the profile information. Identification of representative data sets may not be easy for some applications. However, in practice, even PBO performed with input data sets that are not the most representative, still tends to be more effective than optimizations performed without any profile information.

It is highly recommended that users start using the PBO framework on PA-RISC machines. They will not only get higher performance on PA-RISC systems but will also be better positioned to exploit the full performance potential on HP/Intel EM architecture systems.

For a more detailed description of how to use PBO on PA-RISC systems, please refer to the man pages and the on-line documentation provided with the language products.

[2] Recently, the performance of a large commercial CAD application on a PA-8000 based system was improved by almost 35% by using PBO and selective inlining performed at optimization level 4.

4.2 Aliasing

Aliasing significantly hinders the compiler's ability to extract maximum performance. Aliasing occurs when there are multiple ways to reference the same data location. The most common example of aliasing involves pointer dereferences.

In general, two different pointer variables can point to the same data location. The HP C compiler can keep track of local pointer variables that are assigned the addresses of discrete objects, but determining the actual addresses pointed to by a global pointer variable, a formal pointer parameter, or a pointer returned by a function call is almost impossible. For such pointers, the compiler makes very conservative assumptions and assumes that these pointers can point to all globals and that every pair of such pointers can potentially point to the same memory. Due to these conservative pointer aliasing assumptions, the compiler is very constrained when dealing with such inscrutable pointer dereferences.

Pointer aliasing hinders compiler optimizations in two ways:

- ❑ The compiler does not generate code to maintain aliased objects in registers and instead generates more instructions to access such objects from memory.
- ❑ The compiler is not free to move the use of an object ahead of a definition of an aliased object, thus curtailing the effectiveness of instruction scheduling.

In general, the compiler can not use the declared type of pointer variables to infer that pointers to two different types can not point to the same memory location. In the C language, for instance, it is possible to assign a pointer of one type to a pointer of another type. Pointer aliasing can be a significant performance inhibitor when optimizing C and C++ programs.

The PA-RISC architecture provides advanced features that allow native compilers to mitigate the performance impact of pointer aliasing. Nevertheless, the techniques mentioned below should be considered to improve application performance in the presence of pointer aliasing.

4.2.1 Using Compiler Options for improved aliasing information

Compiling programs with the +O3 or +O4 options improves the compiler's ability to analyze pointer variables for potential aliases. In particular, the compiler has better knowledge of global variables at optimization levels 3 and 4. At optimization level 3, the compiler can identify file static variables whose addresses are never taken and exclude them as potential aliases of pointer dereferences. When the +Owhole_program_mode option is used at optimization level 4, the compiler can identify globals whose addresses are never taken and exclude them as potential aliases of pointer deferences. Additionally, at optimization level 4, the compiler can analyze the potential side effects of procedure calls and can use this information to maintain global variables in registers across call sites.

4.2.2 Source Changes to Eliminate Aliasing

Sometimes it is possible to clearly expose the lack of pointer aliasing to the compiler through source code changes. The source code changes suggested below can be quite effective in facilitating compiler optimizations for performance-critical routines.

Example 1.

If a routine makes heavy use of the value of a global variable or a value loaded through pointer dereferencing, it is better to copy the value into a local variable at an infrequently executed location and then replace all of the original references with references to the local variable. This can be done safely only if one is certain that the global variable or the memory location pointed to by the dereferenced pointer variable is not modified by called functions or through some other pointer variable.

With this source change, the compiler can keep the value of the local variable in a register and avoid additional instructions that would otherwise be needed to access the value from its home memory location.

In the following source example, global variable g_var and the value, *q, are accessed inside the loop. The loop also contains an indirect store through another pointer variable, p. The compiler has to conservatively assume that p can point to g_var, or that p can point to the global pointer q, or that p and q can point to the same memory location. Hence, it has to load the values of g_var, p, and q from memory in each loop iteration.

```
int g_var;
int *q;
slow() {
    int i;
    int *p = foo();

    for (i = 0; i < 10; i++) {
        *p++ = g_var + *q + i;
    }
}
```

However, assume that the code is modified as below where the values of g_var and *q are copied into local variables which are then used inside the loop. Now the values assigned to the local variables will be kept in registers and the three loads inside the loop will be eliminated, thus speeding up the execution of the loop body. This is because the compiler is assured that the local variables val, and q_val can not be aliased to *p since their addresses have not been *captured.*

```
fast() {
    int i;
    int *p = foo();

    /* assign to local variables */
    int q_val = *q;
    int val = g_var;

    for (i = 0; i < 10; i++) {
        *p++ = val + q_val + i;
    }
}
```

In summary, better code is generated if references to global variables and pointer dereferences can be replaced by references to local variables. Also, global pointers should be avoided because they cause additional overhead: both the pointer variable and the object pointed to by the pointer variable are assumed to alias with all other pointers.

Example 2.

It is common to have multiple levels of pointer indirection in C programs. The following construct

```
p->x_ptr->y_ptr->z_ptr
```

is an example of an expression involving multiple levels of pointer indirection. When the same complex pointer expression involving one or more levels of pointer indirection is repeatedly executed, it is better to copy the pointer expression into a local pointer and then use the local pointer for subsequent references, as illustrated below.

For example, instead of writing,

```
p->x_ptr->y_ptr->z_ptr = g_p;
p->x_ptr->y_ptr->d1 = 10;
```

considerably less code would be generated if the code is modified as follows:

```
q = p->x_ptr->y_ptr;
q->z_ptr = g_p;
q->d1 = 10;
```

where q is a local pointer. While this should be safe to do most of the time, one should make sure that no other pointer assignments can modify any of the pointers in the expression p->x_ptr->y_ptr.

4.3 Specialized Optimizations

There are certain optimizations that may not universally result in performance improvements and therefore the compiler does not perform these specialized optimizations by default. The relevant compiler options need to be explicitly specified to enable them. One important optimization that falls in this category is *explicit data-cache prefetching***.** Explicit data cache prefetching is an advanced compiler optimization that is performed by the HP compilers for PA-8000 based systems.

To prefetch data into the d-cache, the compiler needs to insert explicit prefetch instructions into the code stream. If the prefetched data is already present in the d-cache, explicit cache prefetching can actually hurt performance due to the overhead of extra instructions. Unfortunately, it is generally very difficult for a compiler to accurately predict whether the data accessed by a program will be present in the d-cache.

Due to the unpredictable nature of d-cache usage, explicit data cache prefetching is performed by the PA-8000 HP compilers only when a special compiler option is specified, namely the +Odataprefetch command line option.

Explicit data cache prefetching can significantly improve the performance of loops with high d-cache miss rates. When the +Odataprefetch option is specified, the compiler automatically inserts instructions inside loops to prefetch data several loop iterations before they are needed. The cache data transfer is thus overlapped with the execution of the intervening loop iterations.

Optimizations to reduce cache miss rates will continue to be very important for achieving high performance. HP compilers for the next generation HP/Intel EM architecture are likely to provide compiler options and pragmas to annotate memory references that are likely to incur d-cache misses, as well as an improved PBO framework to guide the compiler's cache prefetching decisions and other cache-related optimizations. More details will be provided on compiler features for improving cache performance in a later version of this document.

4.4 Conservative Assumptions

Compilers must make very safe conservative assumptions about program behavior to ensure that applications continue to work with optimizations as expected. For example, when the compiler encounters a call to the C library routine, strcpy(), it must assume that the program may have defined its own version of strcpy() and therefore it can not replace the strcpy() invocation with a fast sequence of inline instructions.

In many cases, these assumptions are too conservative, and the compiler could have made more aggressive assumptions without changing the behavior of the application. Therefore, HP compilers provide a number of options which allow the compiler to make more aggressive assumptions to improve the performance of the generated code. These options should be used with care and after making sure that the aggressive optimization assumptions will not break the application.

The PA-RISC compiler provides the following useful options that should be seriously considered by users interested in improving application performance. These options are supported by the compilers for the PA-RISC architecture and will be supported for the next generation HP/Intel EM architecture as well.

+Olibcalls

This option allows the compiler to assume that the program being compiled has not defined its own versions of library routines such as strcpy(), memset(), memcpy(), alloca(), and fabs(). With this assumption, the compiler can replace calls to these routines with very fast inline instruction sequences. This option can yield significant performance gains for applications that spend a lot of their execution time in these library routines.

+ESlit

If your program does not modify literal strings, this option can direct the compiler to allocate string literals in the code area. This option can usually provide a moderate performance improvement.

+Onofltacc

This option is primarily intended to improve the performance of floating-point applications. By default, the compiler has to assume that preserving a high degree of floating-point accuracy is important and therefore it does not perform any optimizations that can affect the accuracy of floating-point computation results. However, if an application can tolerate small variations in floating-point results, this option allows the compiler to perform more aggressive optimizations and can provide significant performance improvements. One such optimization is the replacement of floating-point division operations occurring in loops with a less expensive floating-point reciprocal multiplication operation, where the reciprocal value of the divisor is computed outside the loop.

4.5 Source changes can help improve performance

In general, there are many source code changes that can help improve performance. There is little that the optimizer can do that compares with the implementation of a significantly better algorithm. However, there are a few simple changes that should be considered to help improve performance.

4.5.1 Data Types

Choices of data types can impact performance. Depending on the data types of variables, more or less code can be generated by the compiler. The following examples can help guide better choices for data types:

Example 1.

For computations involving different C integer data types (short, int, long, char), the compiler generates extra instructions to convert a value of the shorter type into the longer type. In the ILP32 data model (the default mode for 32-bit applications), both int and long data type objects are 32-bits in length but in the LP64 data model (the default mode for 64-bit applications), int objects are 32-bits while long objects are 64 bits in length. Therefore, while mixing int and long variables in arithmetic expressions in the ILP32 data model does not result in extra instructions, it will result in extra instructions under the LP64 data model. Thus, it is recommended that, when possible, one should avoid mixing data types in computations.

Example 2.

In the LP64 data model, pointers are 64 bits in length. Therefore, whenever a 32-bit signed integer is added to a pointer, the compiler needs to generate an extra instruction to sign-extend the 32-bit integer value into a 64-bit value prior to the pointer addition. These extra instructions may be avoided by using unsigned integer or long data types in address arithmetic.

In the following example,

```
int i, *p, *q, x[100], y[100];
unsigned int j;

main() {
    * (p + i) = * (q + j);
    x[i] = y [j];
}
```

more code will be generated for (p+i) compared to (q+j) under the LP64 data model because i is a signed int variable while j is an unsigned int variable. Similarly, more code will be generated for the store to x[i] compared to the load of y[i].

4.5.2 Malloc Optimization

If the malloc() C library routine is high in the execution profile, it may be prudent to change the memory allocation algorithm of malloc() by calling the mallopt() C library routine in order to improve performance. For example:

```
#include <malloc.h>
mallopt(M_MXFAST, 128)
```

This tells the malloc() library routine to use an alternative algorithm for all blocks smaller than 128 bytes in size. Although this algorithm may potentially waste more space, it is usually much faster than the default algorithm. This can help improve performance of applications that allocate and free small blocks of memory frequently.

5.0 Summary

To achieve the best possible performance on HP PA-RISC architecture systems, users must go beyond the mere use of the –O command-line option when compiling their applications. Users should start using PBO and higher levels of optimizations (+O3 and +O4) on PA-RISC systems today to better position themselves to exploit the full performance potential of the HP/Intel EM architecture. In particular, PBO is a key enabler for achieving the best performance on the HP/ Intel EM architecture and users should consider early adoption of PBO in their production build processes.

Various source changes suggested in this document may be considered for performance-critical routines and should help improve application performance on PA-RISC systems as well.

Optimizations aimed at increasing cache efficiency will also be increasingly important for high frequency HP/Intel EM hardware systems. Future versions of this document will provide more information on how users can guide compilers in reducing the overhead of cache misses.

6.0 More Information

The following documentations will provide additional information on compiler optimization and performance tuning with PA-RISC compilers:

- ❑ The HP PA-RISC Compiler Optimization Technology White Paper: On your 10.x system at /opt/langtools/newconfig/white_papers/optimize.ps
- ❑ “Performance Tuning with PA-RISC Compilers”, Carl Burch, InterWorks’97, Tutorial #58, April 12-17. 1997
- ❑ “Optimization for a Superscalar Out-of-Order machine”, Anne Holler, Proceedings of the 29th Annual International Symposium on Microarchitecture, 1996
- ❑ “Advanced Performance Features of the 64-bit PA-8000”, Doug Hunt, Proceedings of the Spring 1995 COMPCON.
- ❑ “PA-8500: The Continuing Evolution of the PA-8000 Family”, Greg Lesarte and Doug Hunt, Proceedings of the Spring 1997 COMPCON