Creating High Performance Embedded Applications Through Compiler Optimizations

White Paper

March 2005

Contents

Introduction

The role of the compiler in application development has continually gained importance as evolving processor architecture has increased the complexity of instructions, and the applications developed in the embedded industry. An optimal compiler can not only increase the performance of an application, but can also decrease development costs and engineering cycle time, and accelerate time to market.

Intel's knowledge of its processor architecture has made possible the in-house creation of a full set of software tools designed to extract the most performance out of Intel silicon. This document outlines a methodology for using these software tools to enhance the performance of an application. It will also detail the compiler optimizations available to help.

Performance Methodology

Application optimization is not an exact science. There is no magic formula, and not every compiler optimization behaves the same for every application. This makes it important to understand the general principles of optimization and the steps to take to optimize a particular application.

The ideal time to start thinking about optimization is during the application's design phase. This period typically has greater schedule flexibility than phases closer to release, and optimization activities can be more easily accommodated and budgeted for.

Having time available to concentrate on optimizations is only part of the battle. Code should be architected in optimization-friendly ways. The most important aspects of code design, from an optimization standpoint, are code that is:

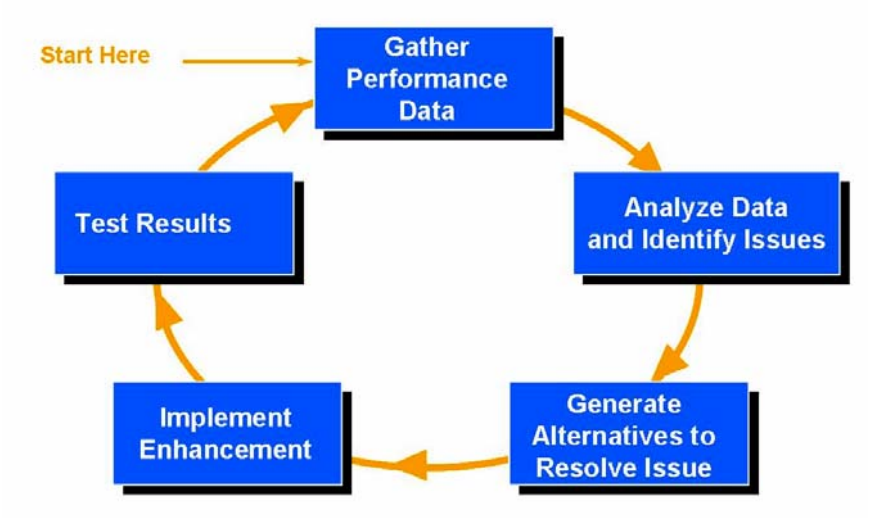
- Readable
- Reliable
- Portable
- Maintainable

A common misconception is that hand-written assembly code is required in the performance-intensive sections of an application. The benefits of allowing the compiler to do the optimizing are many, but they stem from the central fact that as processor architectures have exponentially increased in complexity, so have compiler optimization technologies.

For example, as new processors are introduced, the computing industry is inundated with information on how the newest processor instructions will benefit existing programs. This requires developers to study the new instructions, find out how they work, where they would be most beneficial in a particular application, and then re-code. Alternatively, they can simply use the latest compiler optimization flag. A good compiler can be a developer's best friend as long as it is correctly used.

The Performance Cycle

Once a program is stable and in correct working order, it is time to enter the sometimes long, sometimes arduous, but always very fulfilling Performance Cycle. This cycle (also known as the Closed Loop Cycle) consists of five steps, illustrated in the diagram below:



This is meant to be an iterative cycle. There is likely to be more than one performance-intensive area in an application and it is best to concentrate on one area at a time. Focusing on a large portion of the code will only muddle the data and make it more difficult to see how the applied optimizations directly affect the program.

Step 1: Gather Performance Data

Several different options exist for measuring application performance. Simply using a clock on the wall or a stop watch is an easy, but highly inaccurate way to time an application. Homegrown timing functions inserted into the code are a more effective way to gather performance data. Although these timers bring extra overhead, they provide accurate results. One of the most efficient and accurate ways to gather timing data is to purchase a good performance analyzer software package, such as the Intel® VTune™ Performance Analyzer. The VTune Analyzer will show the time spent in each function of the program and will also provide an analysis based on this data.

Step 2: Analyze Performance Data

Once performance data has been collected, it needs to be analyzed. Find the routines that are taking the majority of the application time and decide if the numbers make sense. Is there a loop that appears to be taking more time than it should? These areas are known as “hot spots.” This process is not always obvious, and the VTune Analyzer’s graphic data display will help immensely. This tool will also show the path in the code that is executed most frequently. However, in the absence of these tools, a strong knowledge of the application and some sweat and tears should identify the sections of code that are taking more time than they should.

Steps 3 and 4: Generate Alternatives and Implement Changes

The next two steps take time and knowledge of optimization techniques to accomplish. Enhancement modifications may be as simple as applying a compiler flag to targeted source files or may require redesigning sections of code. Perform a code inspection to diagnose why the application is slow and to see if any changes will increase speed. Subsequent sections of this document describe various compiler optimizations available in the Intel compilers and where they will be of most help. Success here delivers a faster application built on the combination of good code and a good compiler.

Step 5: Test Results

No matter how well-suited the compiler optimization or how efficient the new algorithm may be, it's very important to verify the results. Does the program still run correctly? If so, did it actually achieve a performance improvement? If either of the answers are "No," the head of the cycle has been reached and another iteration is in order.

Compiler Overview

Now that the steps to follow when optimizing code have been outlined, it is time to move on to the easiest way to get those optimizations for free.

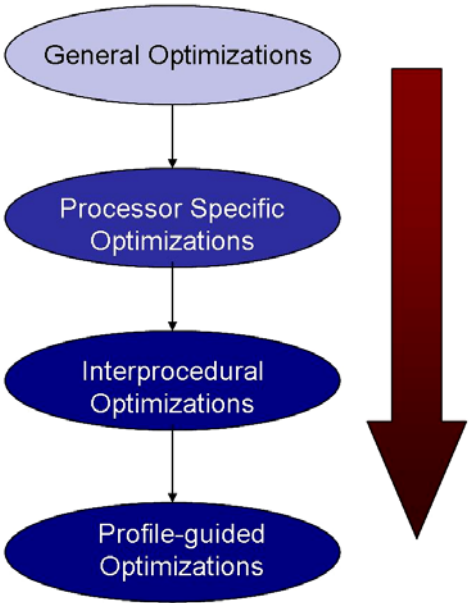
Deciding which compiler to use is the first step. If the target operating system has only one compiler available, then this step is easy. However, good compiler vendors put a great deal of time and energy into ensuring that their compiler is the best optimizing compiler out there, and for that reason they tend to want to make their products widely available. The Intel compilers, for example, are available on a variety of operating systems, including embedded-specific choices such as QNX Neutrino* RTOS.

As the number of compilers available for an operating system rises, the competition for the highest performing compiler also increases. This competition motivates compiler vendors to continue developing new optimization technologies, which can only benefit the developer.

Once the compiler selection is complete, it is important to read the documentation. While this may seem like a trivial step, typical compiler documentation will outline the technologies implemented in the compiler as well as provide examples on how to use those features.

It is important to show restraint when using compiler optimizations. It's tempting to find the most powerful optimization options and throw all of them at an application. However, as mentioned earlier, no single optimization technique will work the same for every application. It is best to apply one optimization at a time and measure any performance improvements before moving on.

The remainder of this paper discusses several types of optimizations. These are best applied in the order presented (see diagram below). Those described first are the easiest to use, followed by more complicated methods that have the potential for greater performance improvements.



General Optimizations

The Intel compilers have a series of options known as general optimizations. These consist of options that bundle together a variety of safe and easy optimization techniques. They will most often have a positive effect on the majority of programs. For example, the "optimize for speed" option is named –O2. While this option does not include every possible optimization that may increase the performance of a program, it incorporates optimizations like inlining of intrinsic functions as well as simple loop optimizations. This option is the best place to start when the optimization phase begins. If an application runs correctly in debug mode (i.e. with optimizations turned off, -O0), but not at –O2, it is likely that more complex optimizations will not work either.

Application incorrectness is not necessarily a problem with the compiler. Using aggressive optimizations will often reveal errors in code that may have otherwise gone undetected. Another common problem associated with higher optimizations is precision control. When the compiler is given free reign to fully optimize a program, it will sacrifice precision for speed. In applications where floating point consistency is heavily relied upon, it is important to tell the compiler to omit optimizations that affect these values. The Intel compilers have precision control options available.

Processor-Specific Optimizations

Processor-specific optimizations provide hints to the compiler about what processor the application is going to be run on. For example, if an application is going to target a Intel® Pentium® 4 processor capable of running SSE3 instructions, then the compiler needs to know it is free to generate these instructions where it feels they will be of most use. (One important word of caution: If this application is run on a processor that does not support the generated instructions, the application will crash.) The compiler is also able to vectorize operations in the program

that can be done in parallel. Vectorization occurs when one operation is applied to multiple data that are then processed in parallel. The vectorizer will automatically use the necessary instructions that convert sequential code into code that processes 2, 4, 8, or 16 elements in one operation.

Processors are utilized more efficiently when instructions are optimally scheduled for the architecture. There is a separate optimization flag for instruction scheduling, which takes into account specific processor instruction latencies as well as cache line sizes. For example, given the following code:

```
for (int i = 0; i < length; i++)
{
 p[i] = q[i] * 32;
}
```

Depending on the target processor specified, the compiler could produce the following instruction sequence:

```
movl (%ebx,%edx,4),%esi
addl % esi, % esi
addl % esi, % esi
addl % esi, % esi
addl % esi, % esi
addl % esi, % esi
movl %esi, (%ebp,%edx,4)
```

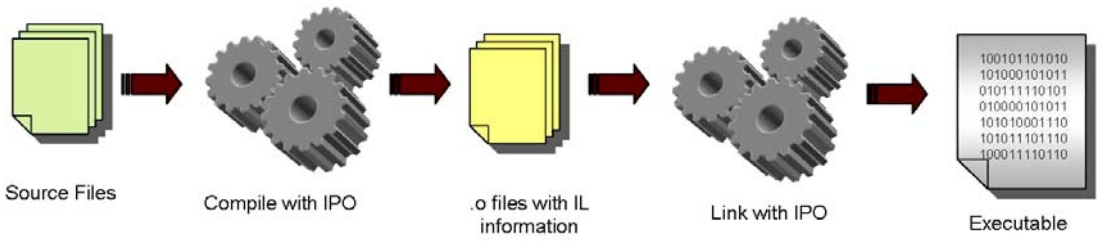
Here the compiler changed the multiply by 32 into a series of adds. The resulting code will execute faster on the targeted processor.

With a technology called Automatic Processor Dispatch, it is even possible to target more than one processor. With this option, the compiler inserts processor ID checks and creates multiple code paths in the executable. At run time, the processor's ID is fed into the program and the stream of code associated with that processor is executed. This results in slightly larger code due to the multiple code paths, but is a good option if the end user is planning to run the application on more than one target system.

Interprocedural Optimizations

The options mentioned to this point only affect the span of one function. Programs can often benefit from optimizations occurring across multiple functions or even across the entire program. This technology is known as Interprocedural Optimization (IPO). This option allows the compiler to look at the whole program as though it were one file. In this way, actions and optimizations taken in one routine can affect those in another routine.

The IPO process is depicted below:



The first step consists of compiling the source files with the IPO option. The compiler creates object files containing the intermediate language (IL) used by the compiler. Upon linking, the compiler combines all of the IL information and analyzes it for optimization opportunities. Since the compiler has the whole program to analyze in this step, the linking phase will take much longer than normal. This is yet another reason why it is important to schedule time dedicated to optimizing. The more involved optimizations take longer to accomplish and often require more analysis by both the compiler and the developer.

One of the main optimization techniques enabled with IPO is inline function expansion. Inline function expansion occurs when the compiler finds a function call that is frequently executed. Using internal heuristics, the compiler can decide to replace the function call with the actual code of the function. By minimizing jumps through the code, this creates a higher performing application.

Another example of how IPO can benefit a program is through pointer disambiguation. The compiler's main goal is to ensure a correctly running program. If the compiler cannot predict the full impact of an optimization, it will take the safe route and not perform it. Take the following function:

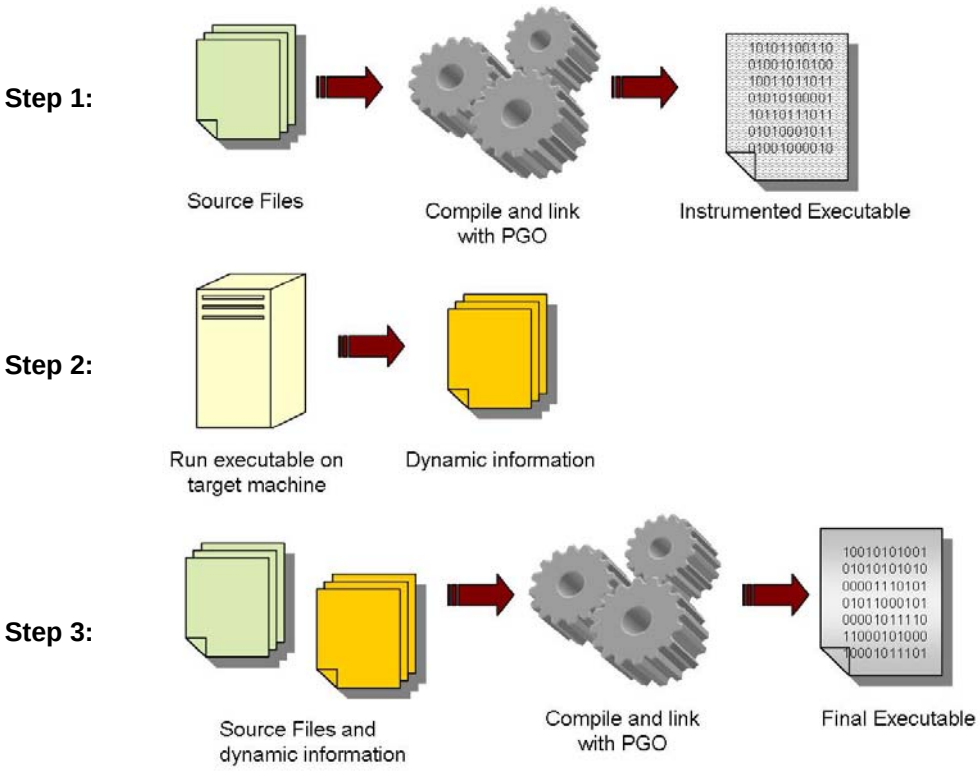
```
void func1(int *a, int *b)
{
 for (int i = 0; i < MAX; i++)
       a[i] = b[i];
}
```

In this situation, the compiler will not optimize this routine without the help of IPO. The compiler does not know what the variables a and b are pointing to. To ensure a correctly running program, the compiler will assume these pointers are aliased. (Aliasing occurs when two pointers are assigned the same location in memory.) If, in fact, these pointers are not aliased, IPO will feed this information to the compiler, which will then optimize the routine.

Profile-Guided Optimizations

A compiler is limited to optimizing based on the data available at compile time. The actual execution of a program could behave in a way that is not intuitive from simply analyzing the available source code. Perhaps one of the most evolved optimization techniques is known as Profile-Guided Optimization (PGO). This optimization allows the compiler to use data collected during program execution to aid in the optimization analysis. Knowing which areas of code are executed most frequently, the compiler can be more selective about the optimizations it performs and can also make improved decisions about how to perform them.

PGO is a three step process, as illustrated in the following diagram.



In the first step, the compiler creates an instrumented executable. This executable contains additional statements that write the execution information into a file. The next step is to run the instrumented application. This executable will perform quite slowly due to the added overhead of writing the dynamic information to a file. This information includes basic block numbers with associated execution counts and other information for the compiler to use. The final step consists of recompiling the application. During this compilation, the compiler reads in the dynamic information and optimizes the code paths accordingly.

It is important when using PGO that the instrumented executable runs on a dataset that is representative of the typical workload processed by the application. If the dataset used only hits the “corner cases” in the application, the compiler spends its time optimizing cases that the end user will likely never encounter.

Profile-guided optimization can affect a program in many ways. Since the compiler is aware of the order in which the code is executed, it can make more intelligent decisions about basic block ordering, register allocation, and vectorization. PGO will also perform enhanced switch statement optimizations. For example, given the following code:

```
for (i=0; i < NUM_BLOCKS; i++) {
       switch (check3(i))  {
         case 3:           /* 25% */
             x[i] = 3; break;
         case 10:          /* 75% */
             x[i] = 10; break;
         default:          /* 0% */
             x[i] = 99; break; }
    }
```

PGO will gather data on the percentage of times each case statement is executed (presented in comments above). To enable the highest performance, the compiler will order the code with the common case occurring first. In this example, case 10 is executed most frequently and having that case evaluated first will eliminate the overhead associated with evaluating case 3 first.

Profile-guided optimization is not suited for every application. It benefits most when an application has a consistent execution path and performs similarly for multiple datasets. If the application has a relatively flat profile, with code execution evenly distributed, then PGO will likely not provide a performance improvement. Before embarking on a PGO build, it is important to consider that it requires changing the build structure. Adding the three step process to some build systems can be difficult. This technique should generally be used at the end of the optimization process.

Conclusion

Compiler optimization technologies are continually advancing. As the demand placed on embedded applications increases and the required time to market decreases, the compiler plays an integral part in all application development. Armed with a good compiler and some knowledge about optimization techniques, developers can much more easily create high performance embedded applications.