Performance Modeling Using Compilers

Continuation on Practical Methods for Maximum Efficiency for Intel® Itanium® Architecture

Author: Max Alt
Intel® Solution Services
Channel Software Operations
May 2005

Abstract

With maturing compiler technologies, compile-time analysis can be a very powerful tool for optimizing code on any architecture. In combination with runtime performance analysis, the compiler becomes the most effective tuning tool. The methods described in this paper can be applied to any parallel architecture with multiple microprocessor execution units.

In the era of parallel and platform computing, we rely less on single execution unit performance, assuming that each component maximized by the compiler becomes important to evaluating performance of the platform as a whole and at the same time, to distribute the processing requests and to embrace techniques of compiler usage in maximizing single execution unit performance.

Knowing coding techniques, writing good code and having a good compiler take care of two latter issues; however, once the code is running, a developer always raises questions: what computing capacity does code implementation have, what hardware feature would increase that capacity, and how would the code's runtime performance be impacted if the hardware were different.

As there are compile-time optimization techniques, which enable a developer to estimate and improve performance without running the program (perhaps with slight guidance from runtime tools), there are also techniques which allow developers to estimate performance on similar architectures without using simulators.

This paper discusses the philosophy behind key performance parameters of the Intel® Itanium® architecture and their functions that factor in the performance modeling formulas.

This paper provides a slightly different angle on performance modeling and analysis of parallel architectures. It explores a compile-time based approach for predicting performance of compute intensive code blocks on Intel Itanium architecture. The author used simulator runs to validate the correctness and feasibility of proposed techniques. Keeping that in mind, there is a possibility of using discrete simulator runs for refining and guiding the result interpretation from compile-time based performance modeling approximation approaches.

Acknowledgements

The author would like to thank Allan Knies, Principle Engineer, Advanced Compiler Labs, for his contributions for the paper. . Allan kindly agreed to run a number of traces on the architecture simulator, which helped us to generate the data supporting the proposed model and approaches. Andrew Sun, Irma Esmer, and Harish Patel assisted with the architecture simulator support and trace runs when Allan was not available to help.

I would also like to thank the Intel Solution Services team (Kirk Hays, Victor Kores, Mike Downs, and Charles Emde) and LANL teams (Olaf Lubeck, Michael Lang, Greg Johnson, and Tuan Bui) for their support, heated discussions, and creative brainstorming meetings during validation and development of the ideas presented in this paper.

Table of Contents

Abstract iii

Acknowledgements iv

Table of Contents v

Introduction 6

Background to the performance modeling problem 10

Description of intuitive performance model 14

Description of performance prediction model not based on simulator 16

Approximation performance model 19

Performance modeling based on Simulation runs 23

Conclusion and issues 27

Appendix 29

References 36

Glossary 37

Introduction

This paper targets any software developer who strives to analyze, characterize, and tune the performance of a High Performance Computing (HPC) application, especially ones that are memory or floating–point intensive. Scientists who are interested in knowing how their algorithm implementation scales with technology improvements and what to expect from computing capacity of a solution in the future, may also find this paper relevant.

One of the greatest questions Intel consultants and engineers hear from Intel platform end-users or code developers are: "How do I know if I have a performance issue? Will I have a performance issue on a new platform?"

In this paper, the author attempts to address this question as well as the following frequently asked questions with regard to application performance:

1. Can the application be scaled linearly with frequency? If not, what would prevent that? How can I model a non-linear part of this?
2. Do we need to increase the L3 cache size in order to run the application faster?
3. Can I scale the application to more CPUs on my node?
4. Do I need a high-speed front-side bus and fast access to main memory in order to scale my application with frequency or number of processors?
5. What would be an optimal platform configuration with relation to ROI to get the maximum performance?
6. Do I need to wait for the next release of the processor or platform to maximize the performance? And if yes, what feature is the most critical?
7. How do I minimize stalls for overlapped memory accesses for future applications?

This paper also discusses specific issues related to Intel® Itanium® processors:

1. Platform features that maximize performance of a given application.
2. How future planned platforms are expected to affect application performance.
3. If an application performs well on the 2005-2007 Itanium architectures, will it still perform adequately when the architecture changes in 2008+.
4. How to create a function that models the non-linear part of scaling with the frequency.
5. How to converge the sequence CPI-peak, CPI-nops, CPI-reg into CPI-compile (see Glossary for definitions of CPI terms).

Scientists and developers are not usually exposed to architecture simulators because simulators are classified by the microprocessor companies even after the product has been released. Requests to perform simulator runs can be made to certain micro-

processor manufacturer' teams; however, the lengthy and computationally intense simulation process limits customer-driven simulation runs.

Therefore, the performance prediction approximation discussed below is extremely helpful in environments where access to microarchitecture simulators is limited or non-existent. These methods provide an alternative to scientists who want a rough projection of their application performance without heavy process overhead. The performance estimation can be further refined later if limited access to an architecture simulator is available.

Another important application of the proposed approach is the power control of the system. According to the model, knowing the application performance potential and performance factors may assist in configuring desired power consumption for production runs on current and future environments.

This paper addresses answers to the questions above in relation to a joint project between Intel and Los Alamos National Laboratories (LANL) that explored performance modeling opportunities for future Intel Itanium architectures. With the desire to develop a generic performance prediction formula for the architecture, the team understood that such a formula would depend on the many software and hardware performance parameters of a specific implementation. It implied primary focus on researching performance parameters and possible prediction models per single application. As a start for the discussion, the foundation assumptions given and variable parameters will be spelled out clearly.

LANL had proposed that our initial research focus on Sweep3d (3D Discrete Ordinates Neutron Transport). It is a publicly available, computation-intensive application widely used in national laboratories for simulation purposes, and it represents the heart of a real ASCII application. The project included the following aspects:

- Performance modeling of Sweep3d and other synthetic applications for EPIC architecture
- Characterization of performance modeling parameters
- Performance modeling approximation algorithm using Intel Compilers
- Verification of performance modeling algorithm using an Itanium architecture simulator
- Sweep3d performance analysis and optimization
- Analysis of advanced compiler features with the Intel Compiler engineering team

The goal of this performance modeling project was to develop a mathematical model for the performance behavior of Sweep3d applications on various existing and non-existing configurations of Itanium architecture deployments. The project went from optimization of the Sweep3d application to testing, measurement, and tuning. The joint project team investigated key influential performance parameters and established five common performance modeling parameters. Later on, a few discrete values for these parameters

were chosen to provide reference (or "golden") data points of performance obtained running the Sweep3d traces on the simulator. Proposed in this paper is that the approximation performance model can be extended to certain performance-critical loops. The mathematical formula for this model can be from a contiguous function obtained by interpolation of discrete values from both the simulator's reference point and values obtained from the approximation model.

This paper demonstrates general techniques, philosophy and approaches behind performance approximation models using Intel Compilers on an example of a Sweep3d application.

The optimization techniques used for Sweep3d, however, have been placed in the Appendix because the goal of this paper is to highlight the advanced performance modeling techniques and methodology to obtain performance modeling parameters. The optimization of Sweep3d is a tool, or pre-requisite, for engaging in the performance approximation process. The author also deferred to the Appendix the usage instructions and best known methods to operate the pfmon tool and open source profiling and tracing tools.

Through the entire course of this paper, the author emphasizes the importance of compile-time analysis, showing how to evaluate and analyze the Intel compiler's output for Itanium architecture, and how to manipulate the compiler's knobs to generate codes with various expected characteristics. These characteristics do not always give the best performing code; however, in combination with others, it may assist in solving the puzzle of understanding all performance knobs.

As an analogy for the compile-time manipulation with compiler knobs and the code, one can imagine a picture on a monitor visualized in the RGB color scheme. On paper, in the CMYK color scheme, this picture will look different, and in fact, may look very bad if color calibration tables are inaccurate. By adjusting the printer, color tables and original picture, one may get an approximation of what is visualized on the monitor. Then, if the paper were replaced with canvas, the entire process of color and printer recalibration would most likely need to be reiterated. These sets of adjustments can be approximated by manipulation of colors on the monitor only, in the original RGB scheme. (The word "approximation" is very essential here, as the author does not claim to have a precise mapping between the color schemes in different color spaces.)

Common color calibrations approaches resemble the described way of manipulating compiler knobs to approximate the performance. Manipulation with compiler knobs and source code can be considered like the manipulation of color on the monitor, and the final print is the runtime performance on the hardware. One can pre-adjust all the colors for different paper types, and the picture on the monitor will not look good, but on the final print and appropriate paper type, you will see a perfect picture. Going back to the compiler theme, it means that knowing the knobs and manipulating them intentionally in compile-time may produce a code that does not perform well, but has a desired characteristic. Or, it could be the code that gets the best performance on new architectures, and this code might assist in complete performance analysis of the code.

As additional benefit of this effort - it demonstrates an Intel customer the way to optimize their investment for the hardware in order to achieve best price/performance or watt/performance. It also allows opportunity to approximately predict performance of a compute intensive code for architectures that are not existent, but with known specifications. It becomes easier to suggest proven solution architecture to an HPC customer, which would allow Intel customer to maximize ROI. High customer satisfaction and matching customer's computing needs to the advised hardware capacity promotes quality of Intel solutions. In other words, this approach assists Intel customers and professional services to extend capacity planning methodology onto micro-architectural level (which currently is not a possible task with current capacity planning and TCO analysis tools).

Background to the performance modeling problem

A common question about an application is its ability to scale with the frequency. In the diminishing role of performance description via frequency, developers like to know the approximate scaling factors of their application performance in environments with new technologies. Currently, assuming the compiler is performing the best possible tuning for the architectures, the applications still do not scale linearly with frequency. The non-linear and negative factor of scaling is memory latency. Applications without memory constraints or memory bandwidth problems were observed to scale close to linearly. Most applications are memory dependent and introduce this non-linear factor when the frequency of the CPU is altered.

A lot of research has been devoted to the memory-related non-linear factor, which skids the application from linear scaling. Other non-linear factors which may affect application scaling with frequency are branch prediction mechanism parameters, instruction fetching, translation look-aside buffers parameters, and other CPU internal parameters that may change for different Itanium architectures. Experience showed that in HPC codes, the events related to these parameters are not often encountered when analyzing performance. However, the discussion of those factors may be a good subject for another paper. Here, the focus is mostly on memory-related performance parameters because they are considered the major obstacle to scaling HPC codes linearly with the frequency.

Assume the Itanium processor generations within a one-year timeframe would give the following spectrum of widely available configurations and parameters:

CPU frequency = {1.3; 1.4; 1.5; 1.6; 1.7; 2.0} GHz

FSB frequency = {533; 667; …} MHz

Cache size = {3M, 4M, 6M, 9M, 12M, 24M}

Latencies to caches L1/L2/L3 = {“McKinley,” “Madison,” future processors}

Latency to main memory = {depends on types of RAM, chipset design}

Given the CPI on current architecture, the information below shows that it will be possible to predict CPI on architecture with parameters taken from the five sets above.

Frequency, FSB, caches, latencies to memory, and latencies to caches are always a major concern for HPC architects and developers as performance factors. These five parameters are an intuitive start for performance modeling of HPC applications. However, it could be an issue to have all five parameters in the final formula, and problem reduction may be required. The number of parameters also may potentially create too many combinations even when access to an architecture simulator is available. For example, if each parameter has three possible values in the set, measuring across the entire product for all the 3 data points along 5 axis makes it 3^5 simulation runs. (Perhaps in the future, it might be possible to actually leave out the L3 latency since its impact is heavily

influenced by compiler knobs). Or in other words, about 300 instruction traces out of 100 million to 250 million instructions, depending on the application, make over 50 billion instructions to simulate. Some simulators may take 1 to 2 days to simulate 250 million instructions. It may take a few months to run 50 billion instructions.

When the prediction formula is not based on simulator runs, the complexity generated by using five parameters is even greater. The best way to avoid this is to simplify the problem and make some observations, assumptions, and notations.

First observation: on Itanium processors, the FSB frequency, cache size, latency to caches, and latency to main memory parameter set may be reduced and replaced by a memory latencies distribution vector, which represents equivalent metrics.

The memory latency distribution vector defines an offset from linear scaling with the frequency, i.e. (L1 latency, L2 latency, L3 latency, 0,…0) vector means that there are no places in the code that require access to the memory bus, which means it will scale linearly with frequency. The memory latency distribution vector on Itanium 2-based platforms can be obtained from DEAR_latency_gt_xxx performance counters (refer to the Appendix for details).

Second observation: if an application is not optimized, then the generated code has the entire spectrum of irregular events as a result of inefficient code runs. These events are irrelevant and a complicating factor in the performance model we are trying to build. Also, is there a benefit to analyzing the performance of an application that initially has high CPI? In a case like this where the performance is poor, it is easy to predict that performance on new generation architectures will be bad as well.

Third observation: the application has some software characteristics that may be considered as performance parameters, essential in defining a performance model. Examples of software performance parameters: memory access pattern and data dependencies for the analyzed code block.

Therefore, consider the following important assumption:

Assumption: The analyzed application is **optimized** on the current platform, and **CPI is minimized** for this platform.

CPI-peak: intended peak cycles per instruction

CPI-compile: cycles per instruction estimated by the compiler

CPI-nops: number of NOPs executed by cycles per instructions

CPI-stalls: number of stalls caused by cycles per instructions

CPI-reg: number of register dependencies caused by cycles per instructions

Prior to introducing the model, note the following points:

- The mathematical model of performance can be either an explicit function, which results in new CPI:

  CPI = (Frequency, mem latency, cache size, FSB frequency)

- Or it can be an implicit function of performance parameters which result in zero:

  F (Frequency, mem latency, cache size, FSB frequency, CPI) = 0

- Some performance parameters are functions of others, and may be redundant in attempting to create a mathematical model of application performance on Itanium processors. However, assuming the implicit function above has a solution, i.e. there are values for each performance parameter, that yields a function result of zero. One can see that changing a value of one parameter implies changing another parameter value if a function result of zero needs to be preserved. For example, an increase in frequency requires a decrease in memory latency in order to keep a balanced relationship; a decrease in memory latency implies an increase in cache size or decrease in FSB frequency to keep the function result of zero.

- The following points are examples of reducing performance parameters, when one parameter may be represented as a function of others:

  - The parameter "latency to memory" may reduce the following parameters: front-side bus frequency, platform design, and main memory type

  - "Latency to caches" may reduce these parameters: cache size and cache latencies (defined by the uarch)

  - "Average latency of memory operation" may incorporate front-side bus frequency, memory latency, cache capacity and cache latencies; Fast front-side bus can compensate higher cache latencies and low cache capacity

  - Data Events latencies are distributed based on the latency distribution vector. A full, contiguous model of memory latencies distribution: a function D which models a percent of entire memory accesses which will occur in *m* clock cycles.

    Example of *D(m1, m2, ..., mk)* which represents ALL memory access distribution functions:

    5% of all mem accesses are done in 256 cycles or more

    10% of all mem accesses are done in 128 cycles or more

    15% of all mem accesses are done in 64 cycles or more

30% of all mem accesses are done in 16 cycles or more

20% of all mem accesses are done in 8 cycles or more

20% of all mem accesses are done in less than 8 cycles.

Description of intuitive performance model

Consider a model that may depend on the following performance parameters:

CPU frequency (C)

IPC (INPUT, OUTPUT)

FSB frequency (F)

Cache size/capacity (S)

Latency to caches (L)

Memory latency (M)

Software – *Memory access pattern*

Software – *Data dependency*

The dependency between these parameters can be represented by the explicit function *f(C, F, S, L, M) = CPI*, or implicit function *F(C, F, S, L, M, CPI) = 0,* where CPI is the predicted CPI on the platform. Impact of software parameters on the performance model is the subject of additional research and will not be covered in this paper. Memory access pattern and data dependency are determined by the given code implementation. The data dependency metric usually is an essential part of the compiler's code generator decision. However, the memory access pattern metric is not well defined and, for instance, may be a theoretical estimate of the cache hit ratio.

Having CPI on a current platform provides an additional factor of performance and refines the model. It changes the formula to:

F(C, F, S, L, M, CPI_old) = CPI_new.

In addition, assume there is the set of CPIs which are derived from the runs on existing Itanium-based platforms: *CPI1, CPI2, …, CPIn.* These parameters provide an additional set of constraints which would add discrete solutions and restrict the final interpolated solution for the function F.

Some tools can model all sets of the parameters described above. The finest granularity of memory latency parameters (memory latency distribution vector) can be obtained with the Itanium processor simulator. The simulator can actually generate CPIs as an output triggered by modification of all the parameters discussed in the intuitive model.

If the Itanium processor simulator is not available, is it possible to get an IPC count on a target platform, or is there another way to model performance in order to validate the model and performance via the reference data point ("golden")?

Easiest performance prediction model (using the simulator):

$$F(C, F, S, L, M) = newCPI.$$

The end-function F will be a continuous function which is an interpolated function derived from the discrete values of running the Itanium processor simulator on the five changed performance parameters in the formula above.

Description of performance prediction model not based on simulator

This section demonstrates the thought process for creating an alternative performance prediction model without results from simulator runs. The suggestion is to use a compiler to produce an approximate performance prediction model for a given HPC application's hot code block. The way to use a compiler for this purpose is to alternate compiler knobs in order to transform scheduling characteristics of optimized code blocks.

The compiler knob file is an architecture-driven data file (list of knobs) which is fed to the compiler in order to generate scheduled code suitable to the specific microarchitecture. As an example, the knob file can provide latency to caches, average latency for memory access, and latency of specific ALU operations.

There are ways to change compiler knobs after the microarchitecture data was input into the compiler. It can be done per module, using compiler switches shown in the Glossary. Changes in the compiler knobs file can define or model the following parameters:

- *Load latency (FP, INT)*
- *Number of memory operations per cycle, which may change intensity of memory accesses*
- *Changes in code density, being able to create more sparse code with various pressures on functional units*

The compiler knobs used for performance prediction models assume floating point (FP) and integer memory latency (the average latency to obtain a value if memory access was not a cache miss). For example, the value is hard coded in the compiler knob file that software pipelines loops for Itanium 2 processors and assumes 11 cycles for average memory access if the memory reference is in one of the caches. It is an empirical number of cycles derived from latency to caches, which can be overridden by the compiler user. This means that the compiler scheduler assumes 11 cycles between the point when the FP memory reference is accessed and the point when the FP value was actually obtained. All software pipeline scheduling parameters depend on this value. Modifying this parameter yields different software pipeline scheduling, initiation intervals, and other characteristics of the code.

Assume in our inventory there are k existing platforms with different architectural performance parameters: different frequencies, caches, etc:

CPI1 = f(C1, F1, S1, L1, M1) on a current platform 1

CPI2 = f(C2, F2, S2, L2, M2) on a current platform 2

....

CPIk = f(Ck, Fk, Sk, Lk, Mk) on a current platform k

The N binaries will be generated by modifying the compiler's knob file. Each modification is done with "a reason" (which will be revealed later on). By running on the k platforms mentioned above, the following 2D matrix of CPI(i,j) is identified:

CPI11 = f(C1, F1, S1, L1, M1) on a current platform 1 with knob file 1

CPI12 = f(C1, F1, S1, L1, M1) on a current platform 1 with knob file 2

…..

CPIn = f(C1, F1, S1, L1, M1) on a current platform 1 with knob file n

…..

CPIkn = f(Ck, Fk, Sk, Lk, Mk) on a current platform k with knob file n

In order to interpolate all these functions and generate a continuous function that represents CPI, it is important to know how to model the memory access pattern of an application (or a hot code block). That is where the software performance prediction parameters start to impact the model. However, it is important to define a set of performance parameters in such a way that the software performance prediction characteristics (memory access patterns) of an application would become insignificant for the entire model.

As noted in the introduction to the problem, it is possible to reduce all the following parameters: FSB frequency, cache size capacity and latencies, memory latency, and memory access pattern to the memory latency distribution vector:

$$d(m1, m2, \ldots, mi) = G(F, S, L, M, \text{memory pattern}),$$

where function G should be constructed based on the application hot blocks, and modeled with the compiler by modification of the knob file.

The run j on current platform k with knob file change n will generate CPIj under a distribution memory latency vector dj. In other words, the memory latency distribution vector will define a run on a current platform, which will yield yet another CPI. The set of CPIs will converge to the new CPI on a target (predicting) platform.

Run 1 with mem distribution vector d1 will yield CPI1 and new memory distribution vector d1'

Run 2 with mem distribution vector d2 will yield CPI2 and new memory distribution vector d2'

…

*Run k*n with mem distribution vector d kn will yield CPIkn and new memory distribution vector d' kn*

All the vectors d' of new memory latencies will converge to ultimate memory latency distribution vector D', which provides an offset from linear scaling with the frequency on a target platform. Then, the sequence of derived CPI from those runs will interpolate into a contiguous function to result in the new CPI on the targeted platform:

$$F(CPI1, ..., CPI\ kn) = CPI.$$

Approximation performance model

As mentioned in the previous section, the memory distribution vector represents a "non-linear" component that prevents linear scaling with frequency, since this vector reflects all the irregular data access events which occur due to the architectural constraints.

Using parameter modifications in the compiler knob file, the compiler can generate n binaries which can be run on k – available Itanium-based platforms.

The function G described above defines required runs by guiding the modifications in the compiler knob file, and therefore imply changes in the compiler's scheduling in the intended way.

Those modifications in the knob file will yield differently scheduled code and will back-map CPIs to the memory distribution vectors. In this manner, the one-to-one correspondence will be established between the memory latency distribution vector and CPI. Since each run generates CPI and a memory latencies distribution vector, it is important to be able to generate a memory distribution vector by a given CPI, i.e., to build contiguous functions. The runs generate discrete functions, and therefore it would be necessary to generate enough binaries with CPIs to create a contiguous function on an interval.

The target architecture's performance parameters and the memory access pattern of the application guide generating the function G. Modeling memory latencies distribution vectors with the compiler knobs generates binaries; executing them on the current platform defines the series of CPIs. Therefore, memory latencies distribution vectors can be mapped to CPIs, creating one-to-one mapped pairs in the form:

$D1 \rightarrow CPI1$

$D2 \rightarrow CPI2$

....

$Dn \rightarrow CPIn$

The function G was defined in such a way that the sequence of *D1,...., Dn* will converge to D, the predicted memory distribution vector of the target platform, and in turn, the desired CPI on a target platform. In other words, *(d1, CPI1), (d2, CPI2), ..., (dn, CPIn)* converge to *(D, CPI*) where the sequence of pairs was created using function G.

The end-function F to model the performance could be described implicitly through function pairs:

F((d1, CPI1), (d2, CPI2),...., (dn, CPIn)) = 0, or in explicit form

F(d1,..., dn, CPI1,...., CPIn) = new CPI.

Performance prediction formula to compute CPI on a target platform

Assume l is the compiler constant (hard coded) parameter that signifies average memory access latency, i.e. approximate number of cycles to stall due to usage of a value obtained by memory access.

Assume the memory latencies distribution vector executing code with modified scheduling parameters on current platform is L8, L16, L32, and L64, where Ln is the number of times memory access had latency within range between A and B cycles, and A, B, C, D are given from the set 8, 16, 32, and 64 cycles respectively.

Assume memory latencies of less than 8 cycles are not causing stalls as the compiler scheduler assumes an average of g cycles per memory access (as l > 8 assumed).

This way the total memory accesses that caused the stalls are:

$$L8+L16+L32+L64 = n$$

And the total cycles spent on memory access stalls are:

$$A*L8 + B*L16+ C*L32 + D*L64 = Sc$$

Therefore, the average cycles spent on a stall during memory access is Sc/n during actual program execution.

The compiler assumed that the average cycles spent on a stall is l.

Thus, if the compiler scheduler optimizations were successful, the stalls on the usage of memory access instructions during actual runtime:

$$max\ \{0, (Sc/n) - l\}\ cycles\ on\ average.$$

The total amount of the memory access usages that caused the stall during runtime is the number specified above of total memory accesses that have caused the stall, i.e. “n.”

Therefore, the total cycles wasted on memory access stalls during the execution is:

$$max\{0, Sc - l*n\},$$

which most likely will be just $Sc-l*n$ for most of the applications.

To compute CPI-stall, we need to compute the number of stalled instructions retired, and then it implies:

$$CPI\text{-}stall = (Sc - l*n) / Stalled\_inst\_retired$$

The total Stalled_inst_retired could be derived from the following assumptions:

a. If all the stalls are just memory stalls, then total stalls count is n, CPI-stall = Sc/n - l,

b. If all the stalls are considered as L3 misses, then CPI-stall = (Sc - l*n) / l3_misses_retired

c. If all the stalls are general stalls, then total stalls count is Si = BE_EXE_BUBBLE.ALL - flawless_exe_stalls, and it yields

$$CPI\text{-}stall = (Sc - l*n) / Si$$

And,

$$CPI = CPI\text{-}compiler + CPI\text{-}stall.$$

The formula deduced above is based on the assumption that the total number of stalls caused by memory accesses is Sc (a function of memory latency distribution vector). This assumption may not be accurate if two or more consecutive memory accesses would cause a stall with memory response time overlap. In other words, when the average stall for memory accesses is being computed by adding all the stalls, it does not account for the multiple stalls occurring at the same point. In the case of multiple stalls, only the longer stall should be counted while other memory stalls may be cleared.

The best way to compute overlapped multiple memory stalls is the memory bus request queue. There is a CPU event measuring the memory queue and the derived software event when the queue contains more than one outstanding request in it. At the point of memory stall interest (i.e., DEAR_latency_xxx) the memory queue event can provide a weighted factor to the stall when computing Sc and average stall for memory access. This factor would be an important update for the general-case formula. However, in Sweep3d, there are not many overlapped multiple memory stalls, as the compiler scheduler assumes the default of 11 cycles (counting very few L3 cache miss events, potentially causing a longer stall).

Validation of performance prediction formula

It is possible to make a hybrid model of approaches whether or not the simulator is available. Using the simulator may help to interpolate discrete values into function F, as much as it can potentially validate results on a non-existing platform. Validation of the model on an existing platform can be done by measuring actual CPI.

An application also has software performance parameters that may influence the modeling formula: Memory Access Pattern and Data Dependencies.

The Memory Access Pattern can provide a fine-grain understanding of cache hits and misses and guide us through the average FP load memory latency assumptions with better precision. The Memory Access Pattern is a mathematical function but often cannot be expressed as a simple formula.

For small applications, or loop kernels such as: $x = x + b(k)$ the formula can be obtained, but it is extremely complex.

Data dependency is a compiler-generated model. For each application, the compiler scheduler builds a dependency tree for memory accesses, registers and execution. The metric for data dependency is the number of cycles the compiler assumes it would take to get/set a value from/to the memory.

The next two sections describe the details of the performance modeling process using the microarchitecture simulator, and how it was used to validate the proposed approximated performance prediction model on a sample of Sweep3d applications.

Performance modeling based on Simulation runs

Based on the performance modeling parameters discussed above, the interpolation of function F can be performed with numerical analysis methods in 6-D (memory bandwidth, memory latency, L3 cache latency, L3 cache size, core CPU speed, and the resulting CPI at each point).

Three data points were chosen along each axis, since it is possible to extrapolate the values to a quadratic function along each dimension if necessary (currently, it is only using linear interpolation). If the simulation and prediction model will be reduced to reflect only 3 parameters, the problem would be simplified to 3-D.

Suppose there is some new point (call it P) in 5-D space for which we want to estimate CPI. Essentially, the closest point to the point to be projected can be found in this measured space (call it C). Then find the 5 nearest neighbors to that point (closest (C,dim) where dim is one of the input dimensions of interest). These 6 points define a plane in 6-D space, and then for determining CPI it would be sufficient to linearly interpolate by computing the plane equation.

In general, the main problem with this approach is the number of simulations required to fill the empirical part of the space. However, approximation can be done by reducing the space dimensions or the number of performance parameter values it is required to test.

In summary, according to the five parameters above, the steps to find target CPI are:

1. Select 3 data points against each axis.
2. Measure the 3 data points along the 5 parameters (3**5=243) which produces 243 data points.
3. Run the simulation on the source code traces.

   As a sample metric: a single threaded code base with 250 million instructions runs from 4 hours to 2 days in the simulator.
4. Use optimized Sweep3D code base to generate traces.
5. Use the PinPoint tool to generate traces.
6. Use Pfmon to validate cache misses, memory distribution vector, and CPI on existing platform.

Input parameters suggested for use in simulator runs:

- L3 cache sizes of 2 – 24 MB: 2, 8, 14
- L3 cache latencies of 12 – 40 clocks: 12, 16, 20
- Memory bandwidth of 6.4 – 32 GB per second per socket: 3.2, 6.4, 12.8
- Memory latency of 70 – 160 nanoseconds: 70, 110, 150
- Core CPU speed of 1.5 – 4.5 GHz: 1.5, 3.0, 4.5 GHz.

Input parameters can be reduced even further, assuming that simulations ran on one type of CPU: 2 GHz.

- Input Parameters: ($cpi = f(p_i)$)
  - Memory latency
  - L3 cache size
  - FSB frequency

- Output
  - Cpi
  - Memory histogram (memory latencies distribution vector)

On Sweep3d simulation traces (obtained either with default 12x12x12 or 100x100x100 input dimensions), the team observed that the simulation workload did not fit in a 12-MB cache. After 100 million instructions, only 30% of a 12-MB cache has data.

The L3 MPI (misses per instruction) goes down slightly from .000297 (3 MB) to .00285 (12 MB) for the traces with a 10x10x240 input trace. (A spreadsheet with the CPI/MPI breakdown is available in the Appendix.) The CPI variation on per trace per processor with the various cache and memory latencies varies by less than 10%.

Comparison between data obtained from running traces on the simulator and from the compiler-based performance prediction approximation model

The figure below shows the initial run of the optimized version of Sweep3d on the default input file. Then 14 binary files were made with various FP load latency values for the compiler's scheduler. Each of the 14 Sweep3d executables had different code generation scheduling based on an assumption of how long it takes to obtain a value from the accessed memory location. In this table of memory distribution vectors, IPC per each of 14 executables was run on the existing Itanium-based platform.

SWEEP	X	y	z		RAM	
INPUT	12	12	410		8M	
FP_LD_LATENCY		IPC	LAT64	LAT32	LAT16	LAT8
33		2.38	387	2832	9306	88589
27		2.47	339	2634	9384	89401
25		2.45	325	2516	9607	86118

19		2.43	330	2323	8025	86998
15		2.43	384	2377	9506	88304
13		2.42	418	1937	7187	86984
12		2.26	345	2103	8214	88716
11 - Default		2.41	343	2103	9000	78446
10		2.28	321	1641	7943	88311
9		2.43	368	1692	7212	92114
8		2.28	236	1337	6536	88800
7		2.41	309	1536	6071	87563
5		2.36	312	982	6195	79508
3		2.16	231	1047	6158	72103

Legend:
LAT8 – number of times memory access took more than 8 cycles
LAT16 – number of times memory access took more than 16 cycles
LAT32 – number of times memory access took more than 32 cycles
LAT64 – number of times memory access took more than 64 cycles

Figure 1. Memory latencies distribution and IPC, 30M-instruction block profiled with pfmon.

Figure data interpretation. The IPC is well-optimized: default of 11 cycles for FP load latency makes IPC = 2.41. IPC does not deviate much from its default when the FP load latency is being varied. The best deviation is 2.47 when the FP load latency is as high as 27 cycles. In this case, the number of memory accesses that caused large stalls is minimal. With lower FP load latency, one notices smaller stall cycles, but more memory stall overlaps since 3 to 5 cycles are not enough to retrieve a value from the memory if caches were missed. The memory requests queue is not reflected in the figure.

In summary, the Sweep3d does not miss L3 cache significantly, and memory latency over 16, 32 and 64 cycles may be neglected with relation to the number of instructions profiled. The total number of memory latencies greater than L2 latency varies from 80,000 to 100,000, and greater than L3 latency varies from 7,000 to 10,000, out of 30 million instructions executed. This aligns with performance simulator results where the microarchitecture simulator showed MPI = 0.00285 for a 100-million instruction block trace.

Indeed, below it is easy to show that on the example of Sweep3d, the performance prediction formula (based on values from Figure 1) also matches simulator run results.

Based on the definitions of L8, L16, L32, and L64 above and the numbers in the table:

L8 = 78446 – (9000 + 2103 + 343) = 67,000

L16 = 9000 – (2103 + 343) = 6,554

L32 = 2103 – 343 = 1,760

L64 = 343

The total stalls caused by memory accesses are 75,657 with total stalls of:

Sc = 343*64 + 1760*32 + 6554*16 + 67000*8 = 719,136 cycles related to memory accesses

The average memory access stall takes 9.5 cycles based on the 10x10x240 input file. With the compiler scheduler assuming the default average of 11 cycles per memory stall, it means that software pipelining loops do not stall on memory, and perhaps can be scheduled for 9 to 10 cycles of floating-point memory latency.

For the 100x100x100 input file, a similar induction can be made, resulting in an average of 13 cycles per memory stall. The CPI formula applied on Sweep3d shows that CPI is almost equal to CPI-compiler, which is CPI predicted by a compiler on a hot spot. Currently the compiler-predicted CPI is not based on computed hot spots or their weight in the executed code. The simplest way to compute theoretical CPI-compiler on a hot spot is to extract the number of cycles in the initiation interval from the optimization report and count the number of instructions required to implement the loop iteration extracted from assembly. The CPI-compiler obtained on Sweep3d code is 0.4 (IPC = 2.5/6), which does not depend on the input. The only parameter in the equation that does depend on the input is the CPI-stall, which grows from calculated zero (for 10x10x240 input) to 0.1 (for 100x100x100) on the target platform.

Conclusion and issues

This section briefly captures conclusions and outstanding issues.

Conclusions reached by this effort:

- The compiler can be a convenient tool to predict and model performance of a single application, while an architecture simulator may not be available for the majority of developers and HPC architects due to runtime and run environment restrictions.
- The simulator results may still be used to refine the compiler-based performance prediction function.
- The performance prediction model is based on the memory latency distribution vector (or DEAR_latency_xxx on Itanium processors) for specially generated N binaries with modified compiler scheduler parameters. The value for this parameter is being computed from the average number of cycles per memory stall for an optimized hot block.
- The 5-D space of performance parameters used are:
    - Memory bandwidth
    - Memory latency
    - Latency to caches
    - CPU core frequency
    - Cache sizes, which can be represented by a memory latency distribution vector
- Pre-requisite for performance prediction model is that the given application is optimized and the compiler has performed optimal scheduling for hot blocks.
- Sweep3d performance was suggested by LANL to validate approaches and techniques for developing a performance prediction. Sweep3d was modeled using memory bandwidth, latency to cache, and memory latency, with fixed CPU core frequency (to 2.0 GHz).
- The Sweep3d application was optimized about 2x, with optimized IPC = 2.41.
- Even on a large input, Sweep3d does not show a large number of L3 cache misses and memory bus originated accesses, and therefore performance is not affected by memory bandwidth, memory latencies, and other parameters. The model was partially validated on Sweep3d. Overlapped multiple memory stalls and other parts of the model were not validated on Sweep3d.
- Tools used for profiling:
    - VTune™ Performance Analyzer was used to perform hot spot analysis and optimize CPI for Sweep3d.
    - Pfmon was used to detect actual CPI and cache misses.
    - Pfmon was also used for memory distribution vectors per N-binaries run for the approximation model.

Obstacles encountered during the effort:

- Technical issues:
  - It took four hours to capture a trace.
  - Some traces ran for multiple days.
  - L3 cache misses were insignificant on Sweep3d, i.e., 1 miss per 1000 instructions (=1/1000 MPI). Although it increased with a larger input file of 100x100x100 (taking 2 GB of memory), the Sweep3d did not produce meaningful data for L3 cache misses, FSB activity, or memory accesses.
- Sweep3d is an important part of the workload at LANL, but the results might be better with another application (Sweep3d may not be the best fit to validate the performance model).
- Compiler-estimated CPI is not relevant for estimation. The correct estimated CPI per block (loop) could be obtained directly from assembly of the loop kernels and optimization reports.
- During the process, some compiler issues surfaced related to loadpair generation that related to versioning.
- The validation of the approximation performance model for the case of overlapped memory access instructions is still an open issue.

Appendix

Sweep3d Workload, Environment and Metrics

The input file is located in the main directory and by default processes a 12x12x12 dimension block. Increasing the block size in the input file yields a cubic increase in memory consumption and usage. The team at one point had used 128-cubed blocks, consuming nearly 2 GB of memory.

The metrics of measuring Sweep3d performance was time to execute ./sweep3d executable.

Sweep3d Analysis Hardware and Software Environment

Table 1 lists the specifications of the Itanium 2-based system used in this project.

System Name	Tiger 4
Processor Type	4 x Itanium® 2 processors
Processor Speed	1.5 GHz
Memory	8 GB
L3 Cache	6 MB

Table 2. Specifications of Itanium 2-based system

The Itanium 2-based system was built with the following Linux§ OS:

- Red Hat Enterprise Linux Advanced Server 2.1, Kernel 2.4.18-e.31smp

In addition, the following Intel tools were installed on the Itanium 2-based system:

- Intel C++ Compiler 8.1 for Linux
- Intel Fortran Compiler 8.1 for Linux
- VTune Performance Analyzer for Linux 3.0
- Dante's simulator for the Itanium processor
- PinPoint tool for capturing traces
- Pfmon (hardware counters performance monitoring tool)

Sweep3d Analysis, Optimizations and Findings

Baseline

On 12-cubed input, the time to completion of Sweep3d was 0.40 seconds per iteration.

The number of iterations in the input file was increased to 100, so the application could be profiled with VTune analyzer or other performance tools. The time per iteration is the metric that does not depend on number of iterations or how long the application is running in total. Therefore, time per iteration was chosen as the primary metric.

Compiler flags used were –O2 and –O3 to establish the baselines:

With –O2 the time per iteration was 0.40 sec

With –O3 the time per iteration was 0.92 sec

Using pfmon on the original binary, the average actual IPC was 2 with –O2.

Vtune™ analyzer results

The function sweep_ is performing main computations. The following four loops were identified as major hot spots:

```
do n = 2, nm
 do i = 1, it
   phi(i) = phi(i) + pn(m,n,iq)*src(i,j,k,n)
 end do
end do
```

Figure 2 Loop 1 – Sweep3d

```
          DO i = i0, i1, i2
c           dl(i) = 1.0 / dl
            ql = ( phi(i) +
     &           aci(i)*phiir + cj*phijb(i,lk,mi) + ck*phikb(i,j,mi) )
c Loop cannot be distributed - phiir!!! !DIR distribute point
            temp_phi = ql * adl(i)
            phi(i) = temp_phi
c           phi(i) = ql * dl
c auxiliary equations (diamond)
            phiir         = 2.0d+0*temp_phi - phiir
            phii(i)       = phiir
            phijb(i,lk,mi) = 2.0d+0*temp_phi - phijb(i,lk,mi)
            phikb(i,j,mi)  = 2.0d+0*temp_phi - phikb(i,j,mi)
          END DO ! i
```

Figure 3. Loop 2 – Sweep3d

```
            do i = 1, it
              flux(i,j,k,1) = flux(i,j,k,1) + w(m)*phi(i)
            end do
```

Figure 4. Loop 3 –

```
c
            do n = 2, nm
              do i = 1, it
                flux(i,j,k,n) = flux(i,j,k,n)
     &                + pn(m,n,iq)*w(m)*phi(i)
              end do
            end do
```

Figure 5. Loop 4 – Sweep3d

The hot spot above presented the following timing distribution (approximate, assuming ONE to be the total execution time per iteration):

Loop 1 – 1/5

Loop 4 – 1/7

Loop 2 – 1/9

Loop 3 – 1/9

Other blocks – very small percentage

Software pipelining

The initial software pipelining report showed that none of the loops were software pipelined. That was originally due to the Intel Fortran Compiler 8.1 Beta bug in loadpair generation. With –O2 this phenomenon did not occur, since loadpair is part of HLO optimizations. While the version 8.1 Beta was being fixed, the team disabled the loadpair generation with the compiler flag -mP2OPT_hlo_loadpair=F

After that step, all of the loops (except loop 2) were software pipelined. The II was not very good, but it varied between –O2 and –O3 compilations.

Loop 2 does have data dependency that cannot be resolved. The data dependency on the phiir variable yields the II recurrence. This recurrence may not be solved with loop distribution and temporary array due to the algorithm.

Loadpair generation and versioning

The main reason loops 1, 3, and 4 failed to pipeline was the issue related to loadpair generation and versioning of the loops when the compiler optimizer creates multi-version code for memory array alignment to support correct loadpair-based code.

The performance difference between –O2 and –O3 (that –O3 was worse than –O2) was caused by incorrect loadpair generation and versioning code.

By setting the flag -mP2OPT_hlo_loadpair=F to disable loadpair generation, the performance of –O3 improved, being slightly better than –O2.

Scalar replacement

In loops 1 and 4, one can observe that arrays *pn(m,n,iq)* or *pn(m,n,iq)*w(m)* can be substituted by scalars respectively, as their values are not changing within the loop body. By assisting the compiler in loops 1 and 4 and taking it out of the loop manually, the software pipeliner schedules the code better (based on II and estimated CPI).

However, after this was done, performance did not improve. Separating the changes and performing scalar replacement for Loop 1 and Loop 4 separately shows that for Loop 1, scalar replacement boosts the performance of –O3 by an additional 50% (making total time per iteration with –O3 to be 0.18 sec vs. baseline of 0.40 sec). However, the Loop 4 scalar replacement degraded the performance.

This phenomenon can be explained by versioning, and the "good" software pipelined version with loadpair generation was not triggered by runtime. In contrast, in case of manual scalar replacement, loop distribution did not occur and versioning invoked an incorrect optimized version. Turning off loop distribution or loadpair generation by using –mP2OPT_hlo_loop_distribute=F or –mP2OPT_hlo_loadpair=F for respective functionality, makes the versioning generate an appropriate performance optimization.

Tools usage

PinPoint

This tool generates a trace file for the architecture simulator (based on given application, input file, and number of instructions). The PinPoint tool generates a trace through profiling the code and identifying marked regions (hot block of executed instructions).

PinPoint tool is an open source tool that includes a profiling component. However, the profiling is done according to the instruction execution number and not by cycles spent on execution. This profiling method is architecture independent, sets irrelevancy for execution time, and in fact is measuring hot execution count blocks rather than hot spots. In turn, measuring hot spots bundles the time spent in the block with current architecture

and models that spot behavior by assuming different architectural parameters will not be a correct task.

Perform Step1.sh and Step2.sh as guided by the script. Make sure the PinPoint binary directory is set in the environment path.

Step1.sh creates a profile and hot spot information based on given inputs.

Step2.sh parses the profiled information to determine what regions the trace should extract for simulation to represent the window of the 250-million instructions hot spot. This step generates a text PinPoint file with extention .pp.

Step3.sh generates the trace according to the PinPoint file information and hotspot regions detemined during the profiling step.

Usage in Step3.sh:

pingtrace -pp swip.I.0.pp -ppnum N -- [application where N is the number of the region you want to trace].

The regions are described in the .pp file using a 'marker' scheme. Each marker is a pair (address, count). PinPoint emits up to 4 'start' and 'end' markers (the following example uses only 1 'start' and 1 'end' marker).

The .pp file looks something like this:

#Pinpoint= 1 Slice= 226 Icount= 56250000000 Len= 250000000

region 1 21.359 4 4 250000000 56250000000

start 1 ***19423700*** *63*

end 2 ***10424438*** *75*

...

markedInstrs 46

mark 1 ***0x4000000000115b92*** *1 0 1*

mark 2 ***0x400000000011d1a2*** *0 1 1*

So the start of region 1 happens when address **0x4000000000115b92 (mark 1)** is executed **19,423,700** times. The region ends when address **0x400000000011d1a2 (mark 2)** is reached **10,424,438** times.

The region is described by:

(Start_marker_address, start_count) (end_address_marker, end_count).

This statement says the region begins when “Start_marker_address” is executed “start_count” times, and it ends when “end_address_marker” is executed “end_count” times.

So, to trace region 1, you need to invoke pingtrace as follows:

% pingtrace -ppnum 1 -pp sweep.I.0.pp

Note: Do not use -length/-start_address once you use -pp. Flag –single is not used.

There is a comment line with "#" in the following example:

#Pinpoint= 1 Slice= 226 Icount= 56250000000 Len= 250000000

It identifies the same region with Icount and Len, so, in theory, one could trace the same region (region 1) as follows:

% pingtrace -skip 56250000000 -length 250000000

But there is no need to use this since the regions are better identified with the marker scheme described above. On Itanium processors, we saw that the instruction counts can vary a lot from run to run, so '-skip/-length' is not very reliable.

Pfmon

Pfmon is very easy to use and is a simplified version of hardware profilers for advanced VTune analyzer users. If there is an identified hot spot function and/or start address where the hot block is located, it is extremely easy to obtain the actual IPC and hardware counter values per block.

Pfmon is HP’s performance tool , available with Red Hat’s Linux Advance Server distribution (for which man page is available).

Usage:

pfmon –e cpu_cycles, l3_misses -- application

References

Coding Considerations. Practical methods on maximizing efficiency on Intel® Itanium® architectures, Max Alt; Intel, 2004.

Optimizing compilers for modern architectures, Randy Allen and Ken Kennedy; Morgan Kaufmann Publishers: San Francisco, 2002.

Software Optimizations for High Performance Computing: Creating Faster Applications, Isom Crawford and Kevin Wadleigh; Hewlett-Packard Company: New Jersey, 2000.

Theory of scheduling, Richard W. Conway, William L. Maxwell, Louis W. Miller; Addison Wesley Publishing Company: Reading, Mass., 2003.

Cache blocking and memory bandwidth discussions:

- The Cache Performance and Optimizations of Blocked Algorithms. *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV),* April 1991. S. Lam, E. E. Rothberg and M. E. Wolf.

Intel® Itanium® 2 processor optimization guides:

- Introduction to microarchitectural optimization for Itanium® 2 processors reference manual

- Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization

ASCI Purple benchmarks: http://www.llnl.gov/asci/purple/benchmarks

Glossary

Compiler knobs – Microarchitecture-driven data which is input to the compiler abstracts in order to generate scheduled code suitable to the specific microarchitecture

CPI – Cycles per instruction

CPI-peak – Intended peak number of cycles per instruction

CPI-compile – CPI estimated by the compiler

CPI-nops – Number of CPI-executed NOPs

CPI-stalls – Number of CPI-caused stalls

CPI-reg – Number of CPI-caused register dependencies

DEAR_latency_gt_8 – CPU event that registers an instruction pointer for which memory latency was greater than 8 cycles.

DEAR_latency_gt_xxx – CPU event that registers an instruction pointer for which memory latency was greater than xxx cycles

EAR – Event address register

F/M ratio – The ratio between the number of floating point operations and the number of memory operations per loop kernel.

FP – Floating point

HLO – High-level optimizer

HPC – High-performance computing

II (Initiation Interval) – The number of cycles between the start of successive iterations in the loop (if the II is *n* cycles, a new loop iteration will be completed every *n* cycles at steady state). These cycles are also called software pipeline loop kernels.

ILP – Instruction level parallelism

IPC – Instructions per cycle

ITLB – Instruction translation look-aside buffer

-mP2OPT_ecg_mm_fp_ld_latency – Intel compiler flag to override scheduler's parameters on assumed average memory access stall

#pragma mem_control – Intel C compiler directive to redefine scheduler's parameter on assumed average memory access required to load floating point value (affects only a scope of loop following pragma directive).

SWP – Software pipelining