Improve Computing Performance, without any Hardware Update

Carlos Jorge Feijó Lopes

Departamento de Informática, Universidade do Minho
4710 - 057 Braga, Portugal
clopes@proxima-si.pt

Abstract: Computer performance is an important issue that must be taken into account by the programmers when they develop robustness and consistent solutions to apply in professional applications. Usually the performance problem is "solved" with a hardware upgrade, instead of using optimisation techniques, during the development process, which helps the compiler optimisation task. This communication focus on some optimisation techniques that can be applied on most common problems faced on computer performance, and also attempts to draw some conclusions on the more adequate method to meet the performance goals, either by the programmer or to leave it to the compiler.

1 Introduction

Computer performance is an important issue that must be taken into account by the programmers when they develop robustness and consistent solutions to apply in professional applications. Usually the performance problem is tackled with a hardware upgrade, rather than using optimisation techniques, during the development process, which helps the compiler on is own optimisation. Programmers must know how computer compilers work, so they could make such kind of tasks. As well, cache memory plays a considerable role on this matter.

This communication will focus all his information on IA32 architecture.

When software developers face a performance problem, that can't be solved through usual optimisation techniques, it's necessary to look inside processor features and get to work, with some optimisation based on those processors properties.

The first part of this communication covers the most relevant optimisation techniques that can be found on modern processors. Then, some tips are given on how programmers can help the compiler to apply its optimisation techniques. Finally, the conclusion tries to give some solid ideas on how performance can be achieved, either by leaving optimisation to the compiler, or to the programmers.

2 Modern Processors Optimisation Techniques

Compilers have some limitations when generating machine-level code. To face them, some optimisation techniques could be applied. Let us see some usual (most relevant), compiler optimisation techniques.

Modern processors are pipelined, which means that elaborated mechanisms are employed to make sure that the behaviour of this parallel execution exactly captures the sequential semantic model required by the machine-level program. These superscalar processors can perform multiple and out-of-order operations on every clock cycle.

Out-of-order means that the order in which instructions execute, may not correspond to their execution order in the assembly code. Generally, three types of constraints limit processor performance: (i) the data dependencies in the program, which force some operations to delay until their operands have been computed; (ii) the resource constraints that limit how many operations can be performed at any given time; and (iii) the success of the branch prediction logic, which constrains the degree to which the processor can work far enough ahead in the instruction stream, to keep the execution unit busy.

There are some simple methods to help understand how a processor can take advantage of parallelism and branch prediction in executing a program. These will be further developed along this communication.

2.1 Reducing Loop Overhead

The compiler can reduce overhead effects, by performing more data operations at each iteration, through *loop unrolling*. The idea is to access and combine multiple array elements within a single iteration. The resulting program requires less iteration, leading to reduce loop overhead. However loop unrolling has some disadvantages too. Loop unrolling increases the amount of code generated, and decreases the code readability. It can also introduce new source of overhead, because it is necessary to take care with the number of available registers in the instruction set.

2.2 Operation Splitting (Parallelism)

Modern compilers are pipelined [3], as stated earlier, meaning that they can start on a new operation before the previous one is completed (this is one form of parallelism). Compilers take advantage of this fact, by trying to split a complex operation in several simple operations. A complex operation means that we are accumulating the value as a single variable, while splitting in a set of more simple operations creates several variables that can be combined to get the final value. For a combining operation that is associative and commutative, such as integer addition or multiplication, the processor compiler can improve performance by splitting the set of combining operations into two or more parts and combining the results at the end. Like loop unrolling, parallelism may create some processor overhead. The benefits of parallelism are limited by the ability to express the computation in assembly code. The IA32 instruction set has a small number of registers to hold the values being accumulated. If the degree of parallelism exceeds the number of available registers (*registers spilling*), the compiler will store some of the temporary values on the stack. Once this happens, the performance dramatically drops.

This limitation affects the compiler performance and must be taken in account when using parallelism optimisation techniques. When looking for performance and registers spilling occurs, it might be preferable to rewrite the code so that fewer temporary values are required. That could be done by explicitly declare local variables as well intermediate

results being saved to avoid recomputation. With floating-point data a programmer must try to keep all the local variables in the floating-point register stack.

When working with integers the maximum performance could be achieved by simply unrolling the loop many times. With floating-point data type the maximum performance could be achieved by introducing some, but not too much, parallelism.

2.3 Branch Prediction and Misprediction Penalties

Modern processors work well ahead of the currently executing instructions, reading new instructions from memory, and decoding them to determine what operations to perform on what operands. This *instruction pipelining* works well as long as the instructions follow on a simple sequence. When a branch is encountered, the processor must guess which way the branch will go. In a processor that employs *speculative execution*, the processor begins executing the instructions at the predicted branch target [1]. It does this in a way that avoids modifying any actual register or memory locations until the actual outcome has been determined. If the prediction is correct, the processor commits the result of the speculatively executed instructions by storing them in registers or memory. If the prediction is incorrect, the processor must discard all of the speculatively executed results, and restart the instruction fetch process at the correct location. This leaves to a significant *branch penalty*. Branches to lower addresses are used to close loops. On the other hand forward branches are used for conditional operations.

2.4 Cache Memory

This communication has not addressed yet a relevant aspect related to the performance issue: the *cache* memory.

Cache memory is a small, fast, temporary storage area for instructions and data. It improves the processor's ability to access recently used instructions or data. Cache memory hold blocks of the most recently referenced instructions and data items. An important cache property that software developers must pay attention is *locality* [3]. A program has good locality if when the compiler requires some instruction or data multiple times, the required information is still in the cache (*cache hit*) ready to be referenced again. If the required data or instruction is not in the cache it is a *cache miss*. Because of that, locality has an enormous impact on the design and performance of hardware and software systems [1]. The processor works on a Fetch-Decode-Execute cycle [3]. Instructions are fetched from memory, decoded, and then executed. Often, instructions are executed one following the other, or in loops, where, the same instructions are executed many times. It is possible for the processor to retrieve the next instruction from the cache memory more than 90% of the time [1], avoiding the relatively long delay when accessing main memory. The use of caches significantly improves processor performance due to the faster access to frequently used data.

Given that cache memory improves the processor ability to access recently used instructions or data, it will be natural to think that the larger the cache, the faster the processor performance; however, this is not always true. A larger cache will tend to increase the hit rate, and it is always harder to make big memories run faster [1].

3 Programmers Optimisation Techniques

Has seen above, compilers use optimisation techniques whenever possible. However, to produce optimal code the compiler may need some assistance. This section focus on some points where the programmer can and must help the compiler.

3.1 Improving the Performance of Floating-Point Applications

Improving Parallelism. Modern processors have a pipelined floating-point (FP) unit. To exploit the parallel capability of modern processors programmers must determine which instructions could be executed in parallel. For instance, even if two code statements are independent, their assembly instructions can be scheduled to execute in parallel, improving the execution speed.

Floating-Point Stalls. Many of the FP instructions have latency greater than one cycle [4]; therefore, on modern processors, the next FP instruction cannot access the result until the first operation has finished execution. To hide this latency, instructions should be inserted between the pair that causes the pipe stall. These instructions can be integer instructions or FP instructions that will not cause a new stall themselves.

The number of instructions that should be inserted depends on the latency length. Because of the out-of-order nature of modern processors, stalls will not necessarily occur on an instruction or micro-operation basis [4]. However, if an instruction has a very long latency, then scheduling can improve the throughput of the overall application.

When a FP instruction depends on the result of the immediately preceding instruction, and it is also a FP instruction, it is advantageous to move integer instructions between the two FP instructions, even if the integer instructions perform loop control [4].

3.2 Improving Branch Predictability

Branch optimisations are possibly the most important optimisations on modern processors. Understanding the branches flow and improving the branches predictability can increase application performance.

During the process of instruction prefetch, the instruction address of a conditional instruction is checked with the entries in the *Block History Buffer* (BHB) [2]. When the address is not in the BHB, execution is predicted to fall through to the next instruction. This suggests that branches should be followed by code that will be executed. The code following the branch will be fetched and, the fetched instructions will be speculatively executed. Therefore, never follow a branch instruction with data.

Additionally, when an instruction address for a branch instruction is in the BHB and it is predicted to be taken, it suffers a one-clock delay [4]. To avoid the delay of one clock for taken branches, programmers must simply insert additional work between branches that are expected to be taken. However, programmers must take care that some processors (like

Pentium Pro, Pentium II and Pentium processors with MMX technology) have a branch predictor that correctly predicts regular patterns of branches (up to a length of four) [4].

Software developers must, when possible, try to eliminate and reduce the number of branches. Eliminating branches improves performance by removing the possibility of mispredictions and reducing the number of BHB entries required.

3.3 Data Arrangement for Improved Cache Performance

Has we have seen, cache behaviour can dramatically affect an application performance. By having a good understanding of how the cache works, programmers can structure their code and data to take advantage of cache capabilities.

Declaration of Data to Improve Cache Performance. Compilers generally control allocation of variables, and the developer cannot control how variables are arranged in memory after optimisation. Specifically, compilers allocate structured and array values in memory in the order the values are declared as required by language standards. However, when in-line assembly is inserted in a function, many compilers turn off optimisation, and the way programmers declare data in that function becomes important. Additionally, order of data declaration is important when declaring data at the assembly level.

Sometimes an unaligned data can be avoided by changing the data layout. Developers must rearrange the data so the larger elements are declared first, thereby avoiding misalignment. Let us see one example:

```
Short a[15];       /* 2 bytes data*/
int b[15], c[15]:  /* 4 bytes data*/
for ( i=0; i<15; i++)
{ a[i] = b[i]+c[i] }
```

Rearranging the data could optimise such code:

```
int b[15], c[15]:  /* 4 bytes data*/
Short a[15];       /* 2 bytes data*/
for ( i=0; i<15; i++)
{ a[i] = b[i]+c[i] }
```

Data Structure Declaration. Data structure declaration can be very important to speed up data accessing in structures. Data structure must use as little space as possible. That could be accomplished by always having some precautions when declaring arrays and structures.

The way arrays are declared within a structure is dependent on how they are accessed in the code. The array could be declared as a structure of two separate arrays, or as a compound array of structures, as shown in the following code segments:

Separate Array:

```
struct{int a[500];
       int b[500];}s;
```

Compound Array:

```
struct {int a;
        int b;} s[500];
```

Using separate arrays, the elements of array `a` are located sequentially in memory followed by elements of array `b`. In the compound array, the elements of each array are alternated so that for every iteration, `b[i]` is located after `a[i]`.

If programmers code accesses arrays `a` and `b` sequentially, arrays must be declared sequentially. That way, a cache line fill that brings in element `a[i]` into the cache, also brings in the adjacent elements of the array into the cache. If programmers code accesses arrays `a` and `b` in parallel, developers should use compound array declaration. Then a line fill that brings in element `a[i]` into the cache, also brings in element `b[i]`.

3.3.3 Loop Transformations for Memory Performance

In addition to the way data is structured in memory**,** developers can also improve cache performance by improving the way the code accesses the data.

Loop fusion is a transformation that combines two loops that access the same data so that more work can be completed on the data while it is in the cache. It splits a loop into two loops, so that the data brought into the cache is not flushed before the work is completed.

Loop interchanging changes the way the data is accessed. Some compilers store matrices in memory, in row order, while others store them in columns order. By accessing the data as it is stored in memory, programmers can avoid many cache misses and improve performance.

Developers must note that some of the showed transformations may not be valid for some programs [4].

3.3.4 Use of Static Variables and Functions

When variables are allocated on the stack they may not be aligned. Programmers must try to use static variables when it is possible, because compilers do not allocate static variables on the stack, but on the memory, allowing the variable alignment if necessary, and also because in most cases when the compiler allocates static variables, they are aligned.

Static keywords can also be used, with performance improvement, on functions. The advantage of using static keywords for functions is that compiler is free to move code between functions. Declaring a function as static means that that function will be created just once, and that way, the compiler can directly replace the function call by its code.

4. Conclusion

During this communication, the goal was to explain and give some tips on how performance could be achieved.

But, despite of everything that has been said until now, the main question - “what optimisations should be left to the compiler, and what are the ones that should be made by

the programmers?" - is still without an answer. Such answer is not easy, and depends on a variety of factors. There are optimisation techniques that without a doubt should be performed by the compiler, or even by hardware, like loop unrolling. There are others that, regardless the fact that they are already performed by the compiler, the programmers can make things easier to the compiler, like for instance, rearranging the data. For example, the techniques presented in sections 3.1 and 3.2 should often be left to compilers, because they are much low level (assembly level) programming, and the developer when trying to increase performance can actually decrease it, if he/she does not have the necessary skills and knowledge.

In the next lines some suggestions are given, that programmers should always execute:

- Eliminate excessive function calls. Move computations out of loops when possible.
- Try to write cache-friendly (good locality) code. When possible use local variables rather than global variables.
- Branch trees can be a problem, so they must avoid them when possible. Do not use a bevy of "if" statements when a "switch" statement can do the job.
- Whenever possible, use the *static* keyword for anything that doesn't have to be seen outside of the source in the file.

Another question that could be asked is "Ok, the programmer can make such optimisations, but that would not be compilers or even hardware job?". This tricky question leaves us to the following conclusion: "yes, of course compilers should solve it, but if programmers can simplify their job why not help?". That help means less compiler work and of course best performance.

The knowledge that we have to take from this communication is that compilers should and actually do most of the optimisation needed, but if programmers effectively know how to help them, they could use some of the techniques explained, always keeping in mind that low level optimisations must be left to compilers, because they probably will do a better job on it.

References

[1] Bryant, Randal E., O'Hallaron, David R.: Computer Systems a Programmer's Perspective, Beta Draft (2001); also available from http://www.cs.cmu.edu/~ics

[2] Huang, Jian., Lilja, David J: Extending Value Reuse to Basic Blocks with Compiler Suport. IEEE Transactions on Computers, Vol. 49, Nº 4, (April 2000)

[3] Intel Technology Essentials Platform Integration – Course #IN102

[4] Intel,: Intel Architecture Optimization Manual, available in http://www.intel.com/design/PentiumII/manuals/24281603.pdf

[5] Panda, Preeti R., Dutt, Nikil D.: Augmenting Loop Tiling with Data Alignment for Improved Cache Performance. IEEE Transactions On Computers, Vol. 48,Nº 2, (February 1999)