Journal of Physics: Conference Series **331** (2011) 012003 doi:10.1088/1742-6596/331/1/012003

How to harness the performance potential of current multi-core processors

Sverre Jarp, Alfio Lazzaro, Julien Leduc, Andrzej Nowak

CERN openlab, Geneva, Switzerland

<Sverre.Jarp@cern.ch>

Abstract. Leakage currents have put a stop to the semiconductor industry's ability to increase processor frequency in order to enhance the performance of new microprocessors. Instead, we observe a slew of changes inside the micro-architecture with an aim of enhancing the performance. Several of these changes, however, do not translate into automatic speed improvements for the software. This paper discusses the increased complexity of modern microprocessors by separating out into dimensions each feature that impacts performance and mentions briefly ways of improving software, in particular that of the High Energy Physics community, to take full advantage.

1. Introduction

The purpose of this paper is to highlight the increasing complexity of modern microprocessors and the corresponding difficulty of programming for peak performance. We focus on the x86-64 architecture, but complexity is equally present in processors based, for instance, on the SPARC architecture from Oracle, the Power architecture from IBM, or the Itanium architecture also from Intel. The AMD x86-64 variant is slightly different from the Intel one, mainly due to the current lack of support for hardware multithreading. For this reason, we concentrate on the Intel flavour of the architecture and describe what we call the seven dimensions of performance that are available in the modern designs.

We underline the fact that the dimensions are multiplicative, but that a lot of existing software was not designed to take advantage of all of them, especially the ones that naturally relate to data-level parallelism.

In Section 2 we describe each hardware dimension in detail, stressing the fact that it is going to become more and more vital to program correctly for this level of complexity.

In Section 3, we describe the typical issues with the mainstream software in use today in the High Energy Physics (HEP) community and make a set of proposals for making improvements, either incrementally or by more substantial rewrites.

One thing that we must explain right away is our "high throughput" orientation. Whether we deal with physics simulation, event reconstruction, or physics data analysis, HEP computing profits enormously from the fact that events are independent of each other. This leads to the concept of "trivial parallelism" that has always helped simplify the computing model needed for a given experiment. If NxM events need to be processed, N and M can be chosen freely. If we need to simulate 1'000'000 events, for instance, we can split the work in 10'000 jobs processing 100 events each, or 100 jobs processing 10'000 events each (or any other combination for that matter) and simply join the output files at the end.

Finally, we conclude by pointing out that throughput can be disturbingly low when the performance dimensions are not properly filled.

2. The increasing complexity of processor hardware

In this section we review the historic evolution of the x86 microprocessor from the days of the Pentium, the period when HEP started realising that x86 was a credible platform for cost-effective computing [1], to the most recent offerings in the market today. The dimensions we will be describing are illustrated in Figure 1.

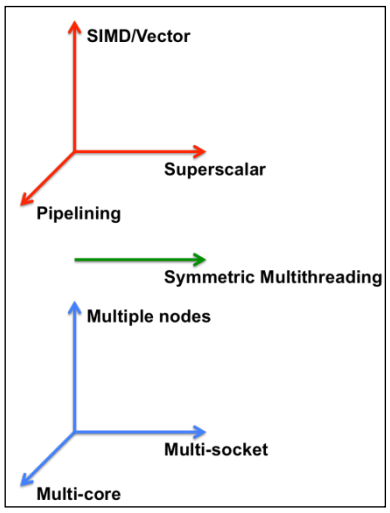

Figure 1. The seven dimensions of performance in a modern microprocessor

2.1. The evolution of the performance dimensions in x86 processors

In the Pentium days, from an execution point of view, there were basically only two major performance dimensions. Although the processor itself was superscalar with two semi-independent ports for scheduling instruction execution (the so-called “U and V pipes”), its performance was largely decided by the frequency of the execution units. PCs with more than one processor on the motherboard were rare, so, when people needed more performance than what was available in a single system, they would split the load across multiple systems. As we explain towards the end of this section this was extremely easy given the paradigm of trivial parallelism that is present in our workloads.

In general this meant that when more performance was needed, the solution typically did not come from the programming community, but from one of two external sources. Either the semiconductor manufacturers increased the speed of the CPU – the Pentium processor, for instance, went from 50 to 200 MHz – or one would acquire more boxes to cope with the total load.

Although another performance dimension became more commonplace with the Pentium Pro, namely the multiple-socket dimension, the global view on performance did not change rapidly or fundamentally. One initial reason was maybe the fact that Linux did not really support multiprocessing until version 2.1. As a matter of fact, when CERN acquired dual-socket Pentium II PCs for the NA48 experiment in 1998, the second processor was switched off in production mode.

So, as already stated, in this period when increased throughput performance was needed, either new systems would be acquired with higher frequency or one would install more boxes in the computing centres.

Since then, we have seen several added performance dimensions. In 2001 we saw the introduction of SIMD vectors in the second wave of Streaming SMD Extensions (SSE2). In a 128-bit register, one could now perform two 64-bit (double-precision) operations or four 32-bit (single-precision) operations in parallel. Thus another performance dimension was born.

In 2005 we were blessed with multi-core processors – our fifth dimension (which we discuss before the fourth one). At the time of the Pentium 4, the semi-conductor industry had come to realise that continued increase in frequency was impossible because of leakage currents in the underlying transistors. The solution, in order to deliver more performance, given Moore's law, was to propose two or more cores on a die. These cores were complete processing units with an entire set of execution logic, their own instruction and data caches, and so on. For every workload that exhibited parallelism this dimension was, in most cases, immediately useful. For single-threaded software, it was, however, of little or no use.

The fourth dimension, hardware multithreading, is just a "pseudo-dimension" that was first introduced on the x86-64 architecture one year prior to multi-core with the availability of the Intel Pentium D. "Symmetric Multi-Threading ", or SMT differs from multi-core since it does not provide more execution logic on the processor die; it simply allows two (or more) hardware threads to compete for the available logic and caches. This can be interesting in a multithreaded (or multi-processing) software environment where threads can suffer long stalls. In its simplest incarnation, hardware multithreading allows another ready-to-execute thread (or process) to take over execution when the first one suffers a long stall, such as last-level cache miss. In SMT, the hardware designers have gone one step further and allow instructions from two or more threads to be scheduled in parallel. For instance, the hardware may execute a load from thread 1 and a store from thread 2 in the same cycle.

The sixth dimension was already mentioned. Multi-socket servers have been commonplace since the days of the Pentium II. In the HEP community dual-socket servers have made up the sweet spot, since they were more cost-effective than single socket servers. Quad-socket servers, on the other hand, were always considered too costly, since they were over-equipped with expensive reliability features. This may be about to change since CERN recently purchased a batch of quad-socket AMD Magny-Cours servers for use as batch nodes.

The seventh (and final) performance dimension is simply the number of nodes in use, either in a cluster or in a grid. This dimension has always been present in our data centres, and as mentioned above, it has been the principle method for obtaining more throughput performance, when a single server was not enough.

Dimension	Hardware Implementation
1	Pipelined execution units
2	Superscalar execution
3	Single Instruction Multiple Data (SIMD/Vector)
4	Symmetric Multithreading (SMT)
5	Multi-core
6	Multi-socket
7	Multiple nodes (cluster/grid)

Table 1: Performance dimensions in a modern microprocessor

After this long enumeration of hardware performance dimensions, a few comments are needed.

- The first three dimensions in Table 1 belong to what we call the "inside-the-core" category and are best exploited by data parallel constructs inside a software program. Vectors without data dependency between the elements are great for this kind of parallelism, but they are not often present in the intensive parts of our programs. The next three dimensions are referred to as "across cores" and would best be exploited by task or process parallelism. Operating systems detect multiple cores, SMT and multiple sockets as a multiplication of available

CPUs. So, a quad-socket server with 8-core chips that are two-way SMT-enabled, will display 4*8*2 (64) CPUs in response to the Linux "cat /proc/cpuinfo" command. All of these CPUs are available for scheduling, but the corresponding throughput will vary depending on the actual execution resources that a CPU represents and on their topology within the system.

- The second comment is that the HEP community has, thanks to the paradigm of trivial parallelism, always been able to run multiple jobs across CPUs. Rather than having one single job process one million events, a physicist can freely split such a job in multiple parallel runs as already discussed. One of issues when doing this with the LHC software frameworks is the fact that they need large amounts of memory, typically between one and three gigabytes per process. Although modern servers can be equipped with large amounts of memory this demand by the software frameworks is becoming an impediment to the full use of "across cores" parallelism.
- A final comment is the fact that all dimensions in Table 1 are multiplicative, i.e. when they are exploited by a given software program, the final performance gain is the multiplication of the individual gains obtained in each dimension. This is well illustrated by the "Trackfitter" benchmark (see Section 3.1.

2.2. Performance dimensions in current Intel processors (Nehalem and Sandy Bridge micro-architectures)

As we have seen, there are six real performance dimensions plus symmetric multithreading in a modern Intel microprocessor architecture. This is also the case in the current Nehalem design [1]. Execution units are pipelined, allowing new instructions to be initiated every cycle. Processors are superscalar, allowing up to six operations to be spread over six ports in a burst. Since these processors can only decode and retire four operations in parallel, four is the more realistic number for this superscalar parallelism. The third dimension is also unchanged on the Nehalem architecture; the SIMD vector width is as before 128-bit and allows two double-precision numbers to be processed in parallel. This doubles, however, with the arrival of the Sandy Bridge architecture which introduces vector registers with 256 bits in 2011[1]. With such a size it is going to be somewhat embarrassing if programs continue to execute in scalar, rather than packed mode. In the former, only 64 bits are used (in double precision mode) inside the registers and execution units and the remaining 192 bits are left unused. In the latter, the compiler or the programmer will pack four double-precision (or eight single-precision) numbers and execute everything in parallel. Although one rarely obtains the theoretical peak inside a dimension, it is going to be difficult to explain why a potential speed-up factor of 4 (or 8) should simply be left unused.

2.3 Performance dimensions in current and future AMD processors

As already mentioned in the introduction the AMD processor architecture features the same dimensions as the Intel one, with the exception of the pseudo-dimension, defined by SMT. In its next architectural update, "Bulldozer", AMD is going to reduce one dimension by having two cores share the floating-point (FP) pipelines [3]. This implies that a program running alone will always have access to all of the FP hardware, but if another program is running on the adjacent core, the FP pipeline may be busy when needed. Depending on the FP contents in the two programs there may be contention or not. HEP programs typically consist of 15-20% FP instructions, so it will be interesting to measure if there is a difference in throughput in the two cases.

3. Implications on the software

3.1. HEP software and the seven performance dimensions

[1] AMD will do the same in their "Bulldozer" architecture (see Section 2.3).

In this section we look at some of the issues with HEP software frameworks and the given performance dimensions.

Our C++ software frameworks do not exploit the first two performance dimensions very well. The dimensions of superscalar and pipelined execution are jointly referred to as Instruction-Level Parallelism (ILP), and it is commonly assumed that it is the job of the compiler to maximise it. Unfortunately, compilers are limited in what they can generate as object code once the source code has been fixed by the programmers, especially if heavy use of dynamic C++ mechanisms is made (which is our case). In our frameworks, we measure on average a minimum value of about 1.0 for the CPI (cycles per instruction), whereas it could be closer to 0.25 which is the theoretical minimum. A related concern is that fact that about half of the instructions are load/store instructions (and over 10% are branches). Although some of these "house-keeping" instructions are surely always needed, others might be considered simply as overhead to cope with constructs such as the object hierarchy, virtual function calls, and so on.

As already stated, the third dimension is almost never exploited in our community, so the vector capability of modern microprocessors is largely left unused. In the late 80s a lot of effort was invested in trying to vectorise Geant3 [4]. The effort failed, probably for multiple reasons. Vector computing is best done when vectors are relatively static (in order to avoid unnecessary scatter/gather operations for reshaping them), and also when the same algorithm is applied to each element (avoiding loss of efficiency when elements have to be masked off). In particle simulation, however, one of the complications is definitely the fact that some particles may annihilate and others may be created, so a vector of particles might never stay the same for any length of time. Even when it stays the same, each particle may be present in materials and geometries that require entirely unique algorithms to be deployed. Both arguments make it challenging to achieve an efficient use of vectors in simulation.

The situation is different in track reconstruction and also in data analysis. In online track reconstruction vectors have been used successfully in the "Trackfitter" benchmark created by the CBM/ALICE High Level Trigger (HLT) programmers [5]. In data analysis identical algorithms are often applied to data elements gathered from a large set of events.

3.2. Recommendations for future software

In the last part of this paper we propose some ways of trying to improve the situation. These recommendations are based on performance studies carried out regularly in the CERN openlab.

3.2.1. Broad programming talent

Given the complexity of modern hardware combined with the complexity of modern C++ programs, it is practically impossible for a single person to know how best to create a clean design, develop the code so that it remains maintainable and readable, and still get close to peak performance from the hardware at optimal cost. One approach is therefore to ensure that experts from multiple fields work together to create complex toolkits and frameworks.

Problem
Design, Algorithms, Data structures
Source program
Compilers; Libraries
System architecture
Instruction set architecture
Microarchitecture
Circuits
Electrons

Table 2: The layers of expertise when asking electrons to solve a computing problem

The layers shown above are adapted from a presentation by Prof. Y.Patt [6]. Fortunately little knowledge is mandatory of the bottom two layers. We would claim that it is difficult to find experts who cover more than 3 or 4 layers and impossible to find somebody who can cover 7 layers with a consistently high level of expertise. The solution is therefore to actively combine talents as already mentioned. One example would be to unite a "solution specialist" handling the first four layers and a "technology specialist" handling the layers from the micro-architecture to the compilers, but other combinations of skills could be equally valid.

As always, increasing the span of knowledge by training and workshops should also be encouraged.

3.2.2. Holistic view

A second recommendation is to take a holistic view of, at least, the performance sensitive part of the program. In this paper we can only cover a few aspects since it is broad enough to be the topic of an entire book. We firmly believe that a clear three-phase split between preparation, computation and clean-up is the right approach.

It is important that the preparatory phase does not just initialize constructs and data areas in a hap-hazard fashion but concentrates on the preparation of streamlined layouts for optimized throughput in the execution phase. In our opinion this is key to allowing compilers to perform a good job of optimization. Modern compilers can achieve good results in terms of optimizing for ILP and vectors (our first three dimensions), but only when they are given a playing-field with clear rules. Confusion concerning pointer aliasing or data alignments, multiple layers of indirection, lack of data independence in loop constructs or general memory abuse can destroy the talents of any compiler.

It should be obvious that performing a large number of new/delete operations in the kernel routines of a framework also will degrade the performance of the code significantly.

For task/thread parallelism to be implemented optimally, the software designers must indicate in an unequivocal way to the compiler which data is shared and which data must be allocated as thread private.

3.2.3. C++ programming with performance in mind

As already mentioned compilers should be given clear, unambiguous guidelines in order to be able to produce optimal code. One of the laudable aims of C++ is to allow programmers to produce code that remains readable and maintainable for humans. Nobody will argue against such a goal, but in this paper we claim that this should not be done at the cost of comprehension as far as the compiler is concerned. We mention here a few areas where care must be taken, at least in the compute-intensive part of the program, but for a more in-depth understanding of the issues, we refer to existing literature [7].

One of the obvious examples is the avoidance of virtual function calls since these force compilers to generate dynamic branching instructions and hinders inlining, which in turn reduces the possibility of creating rich code sections.

Inlining typically yields improved speed when done in compute intensive parts. There may be rare cases where it leads to code bloat, but, as a rule, the source code should be structured such that the compiler can itself select when to do inline. Far too frequently in performance profile studies, do we observe tiny, non-inlined functions, such as getter methods with almost no ILP, simply because the compiler was not given visibility of the functions at compile time.

Another area of concern is data structures. Programmers should, in general, prefer Structures of Arrays (SoA), rather than Arrays of Structures (AoS), as depicted in Figure 2. SoA is usually a good enabler of compiler optimisation when correctly programmed via loop constructs because it can help the code generator exploit all three dimensions inside the core; generation of vectors, pipelines filled with instructions from different loop iterations, as well as independent operations identified for superscalar execution.

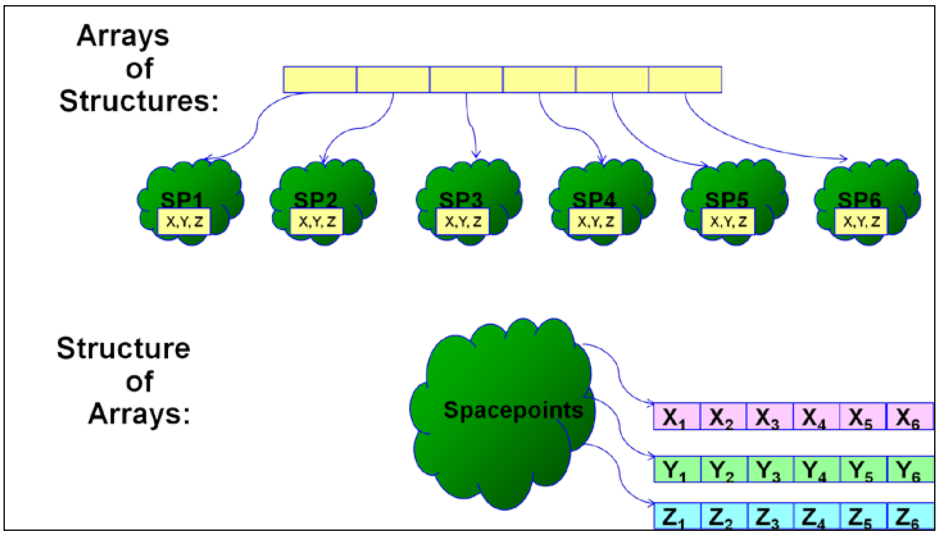

Figure 2. The difference between Structures or Arrays and Arrays of Structures

The last thing we comment on here is the use and the corresponding cost of mathematical functions. HEP programs are heavy consumers of several categories. They obviously use add and multiply instructions, but these have relatively low execution cost (A three or five cycle latency, respectively; fully pipelined native instructions). Things get worse with divisions and square root (typically 20 – 30 cycles, native instructions BUT no pipelining). The cost really becomes high when the algorithms are based on functions, such as exp, log and pow, sin, cos, tan, and atan2. All of these may turn up in HEP profiling studies. Since the cost is easily one hundred cycles or more, it is obvious that programmers must exercise care when the code is written.

3.2.4. Controlled memory usage

We have already discussed the fact that, in the HEP community, we can easily fill a multi-CPU server with multiple independent processes. The problem with this approach is that the memory demand increases linearly per process. Fortunately, several solutions exist.

The easiest one, especially for the current software frameworks, is to fork multiple processes from a mother process[2] and exploit the "copy-on-write" feature of Linux. When the fork command is issued, the daughter processes are simply given read-only access to all the pages in the page tables of the mother. During execution of the daughters, only pages that are changed will be copied from the mother and attached as separate pages to the relevant daughter process. Tests by ATLAS [8] and CMS [9] have shown that 30-40% of the memory needs of their reconstruction frameworks can be saved in this way.

A more elaborate way is to create frameworks that are properly multithreaded. A multithreaded version of Geant4, developed at the North-eastern University [10], is a good example. This prototype, based on the "FullCMS" example in the Geant4 software repository, has been developed with optimal memory sharing in mind. Each thread (and therefore each core) is used to simulate an entire event with a maximum of data shared between threads. The shared data encompasses the physics tables, the magnetic field-map as well as other data that is considered to be global. The code itself has been made re-entrant so that a unique copy is used by all the threads. Such a memory layout was already proposed at CHEP07 [11]. Analysing the memory requirements of the "FullCMS" multithreaded prototype we observe that each additional thread only requires about 25MB per additional thread/core, instead of the

[2] This works best if the mother process completes the treatment of the first event, so that all the initialization work has been taken care of.

current requirement of one to three Gigabytes. Smaller memory demands are a must if we want to take advantage of accelerators, such as Intel's forthcoming MIC (Many Integrated Cores) architecture.

3.2.5. Multithreading C++ code

Unfortunately there is no native support of multithreading in C++. It is hoped that such support might appear with the forthcoming standard [12]. In any case, there are multiple choices available today for those who want to use threading. Here we mention some of the most common ones:

- POSIX threads ("pthreads"): This is the native threading mechanism in Linux (and the basis for nearly all the other higher-level threading implementations). Although it requires programming at a relatively low level, it is not beyond the scope of good programmers to implement, for instance, a master-slave combination for event-level processing using pthreads.
- OpenMP [13]: This is a standard that has been around since the late 80s. The support is built into all modern compilers, whether it supports C, C++, or FORTRAN. For a long time its capabilities were centred on loop-level parallelism, but the most recent version (3.0) also supports task-level parallelism.
- Intel's Threading Building Blocks (TBB): This is an effort to compensate for the lack of threading support in standard C++. TBB, a C++ template library, exists both as an open-source (free) and as a commercial offering. The current version is 3.0 which comes with a large set of features: concurrent data containers, locks, task scheduler, a scalable memory allocator and select constructs from the forthcoming standard.
- Intel's Array Building Blocks (ArBB): This product is currently in beta test. It takes its inspiration from NESL [Blelloch G., A Nested Data Parallel Language, CMU Technical Report, 1993] and targets primarily data-level parallelism. It will, however, also spawn threads as required. Given its ability to cover both data and task parallelism, Intel has already successfully implemented a demo version of the Trackfitter benchmark using ArBB.
- Boost threads: Boost is a set of C++ libraries and it includes facilities for threading. Basic thread management and synchronization primitives are provided, creating an interface similar to that of pthreads. Boost is valued, amongst other things, for its closeness to the C++ philosophy.

3.2.6. Excellent tools

Given the complexity of the hardware and software, excellent support tools are paramount.

- Compilers: The GNU compilers play a predominant role in our community, but there are also good Linux compilers from other sources. Intel continues to invest heavily in their own compilers (which are always GNU compatible). We should also mention the Portland Group compilers. Finally, there is the open64 compiler suite, which was originally based on the SGI MIPS compiler.
- Debuggers and correctness tools: The obvious choice is *gdb*, but there are several other choices. Intel provides *idb* and has recently released the *Inspector XE 2011* for checking both memory leaks and thread checking. The Rogue Wave provides *TotalView* with similar capabilities. *Valgrind*, which is an open source tool, can also be used for detecting memory management bugs and threading errors.
- Libraries: This is a very broad category and it is beyond the scope of this paper to list individual libraries. What is relevant, however, is to mention that libraries do not always come with a performance guarantee and it is up to the developers to comprehend the performance implications in each case.
- Performance Analysers: *Valgrind* may also be used for profiling but it may be slow for use with large frameworks because of the instrumentation overhead. Intel offers the recently released *Amplifier XE 2011* product and Rogue Wave offers *SlowSpotter* and *ThreadSpotter*.

4. Conclusions

In this paper we have highlighted the fact that modern microprocessors have an increasing complexity which, in turn, augments the challenge of programming for peak performance. Six dimensions, and even seven with hardware multithreading, need to be taken into account. In the HEP programming community the first three ("inside-the-core") are the most difficult ones to get to grips with, basically because it is commonly believed that the compiler will do it all. The truth is, however, that for programs with a throughput of one instruction per cycle (when the maximum is four) issuing scalar instruction rather than packed vector instructions (which again is 1 out of four on Sandy Bridge/Bulldozer), one can argue that the programs are extracting 1/16 (or about 6%) of peak performance from the hardware. This is obviously quite a disturbing fact.

Concerning the dimensions "across cores", the situation is a bit more encouraging, fortunately. First of all, there is the trivial parallelism which is prevalent in our computing models. Additionally, efforts to use forking and, hopefully, full-fledged multithreading will alleviate the pressure on memory. It is vital to exploit the cores close to 100% to avoid that the total efficiency across all dimensions shrinks to just a few percent.

Whenever software is given a major overhaul, efficient use of the performance dimensions must be one of the prime targets.

Finally, it should be mentioned, that the "holy grail" is to create programs that not only achieve close to peak performance on today's hardware, but are able to absorb increased capabilities in all dimensions, whether it is longer vectors, more cores or additional hardware threads, without a drop in relative efficiency.

References

[1] Jarp et al, PC as Physics Computer for LHC, CHEP 1995
[2] Thomadakis M 2010 The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms, *Texas A&M University Report 2011*
[3] Butler M, "Bulldozer", a new approach to multithreaded compute performance, *Hotchips 22, August 2010*
[4] Dekeyser J 1987 Vectorization of the GEANT3 geometrical routines for a cyber 205 *Nuclear Instruments and Methods in Physics Research Section A, Volume 264, Issue 2-3, p.* ***291-296****.*
[5] Gorbunov S, Kebschull U, Kisel I, Lindenstruth V and Muller W 2007 Fast SIMDized Kalman filter based track fit *Comp. Phys. Comm. 178*
[6] Patt Y. 2009 Multicore/Manycore: What Should We Demand from the Hardware? *International Supercomputing Conference 2009*
[7] Meyer S., Effective C++/More Effective C++, Addison-Wesley
[8] Binet S, Calafiura P, Jackson K, Legett C, Levinthal D, Tatarkhanov M and Yao Y 2010 Parallelizing Atlas Reconstruction and Simulation: Issues and Optimization Solutions for Scaling on Multi- and Many-CPU Platforms *CHEP 2010*
[9] Jones C, Elmer P, Sexton-Kennedy E, Green C and Baldocci A 2010 Multi-core Aware Applications in CMS *CHEP 2010*
[10] Apostolakis J, Cooperman G and Dong X 2010 Multithreaded Geant4: Semi-Automatic Transformation into Scalable Thread-Parallel Software *Europar 2010*
[11] Jarp S 2007 How good is the match between LHC software and current/future processors? *CHEP 2007*
[12] ISO/IEC 2010 Working Draft, Standard for Programming Language C++
[13] OpenMP Architecture Review Board 2008 OpenMP Application Program Interface, version 3.0