Best practices for HPM-assisted performance engineering on modern multicore processors

Jan Treibig, Georg Hager, and Gerhard Wellein

Erlangen Regional Computing Center (RRZE)
Friedrich-Alexander-Universität Erlangen-Nürnberg
Martensstr. 1, 91058 Erlangen, Germany
{jan.treibig,georg.hager,gerhard.wellein}@rrze.fau.de

Abstract. Many tools and libraries employ hardware performance monitoring (HPM) on modern processors, and using this data for performance assessment and as a starting point for code optimizations is very popular. However, such data is only useful if it is interpreted with care, and if the right metrics are chosen for the right purpose. We demonstrate the sensible use of hardware performance counters in the context of a structured performance engineering approach for applications in computational science. Typical performance patterns and their respective metric signatures are defined, and some of them are illustrated using case studies. Although these generic concepts do not depend on specific tools or environments, we restrict ourselves to modern x86-based multicore processors and use the likwid-perfctr tool under the Linux OS.

1 Introduction and related work

Hardware performance monitoring (HPM) is regarded as a state of the art advanced tool to guide code optimizations. While there are countless publications about HPM-based optimization efforts, a structured method for using hardware events is often missing. even from the same vendor. One exception is the use of cache miss events, which are very popular since memory access is regarded to be a major bottleneck on modern architectures. In fact, miss events are often seen as the most useful metrics in HPM. Many optimization efforts solely focus on minimizing cache miss ratios [1]. Another popular application of HPM is automatic performance tuning via a runtime approach [2,3]. Recent work attempts to apply statistical methods such as regression analysis to achieve automatic application characterization based on HPM [4,5].

This paper will present a more holistic view on how to use HPM in a sensible way. It suggests HPM as one aspect embedded in a structured performance engineering approach (see Fig. 1). The central idea is the iterative development of a diagnostic performance model enforcing a better understanding of the code properties and the hardware capabilities, leading to a deeper understanding how a code interacts with a given architecture. HPM is one important source of information to improve this understanding. We want to stress that HPM is in many cases only meaningful if related to other information like, e.g., microbenchmark results or static code analysis. In the following we will concentrate on what role HPM can play in order to identify performance properties and problems, and to implement a structured optimization effort.

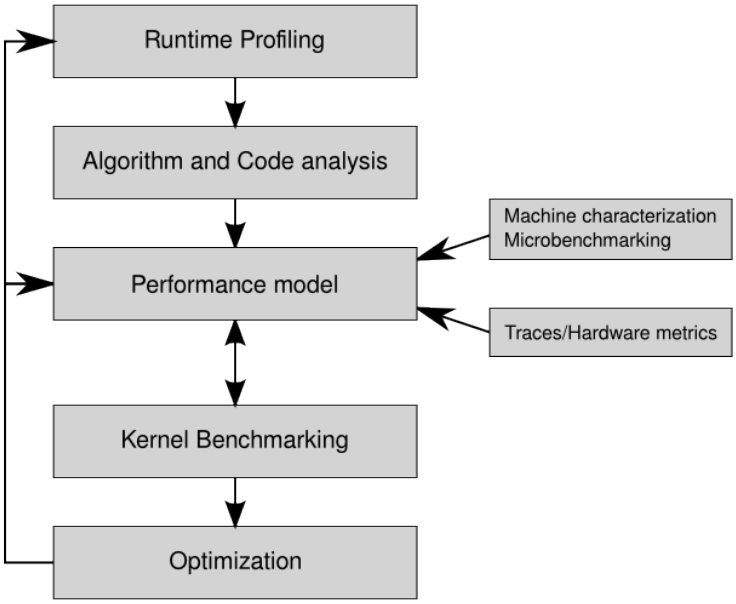


Fig. 1: Structured performance engineering process. The HPM results come in via traces and hardware metrics, but machine parameters and microbenchmarking are of equal importance. A performance model for each core loop is constructed from this data and successively refined and adapted during the benchmarking and optimization process.

While all these ideas are not new, we believe that the emphasis on "patterns" can help tackle problems that are not so easily recognized using automatic tools that are too focused on HPM. We concentrate here on the typical patterns and their identification. The structured performance engineering approach will be published in detail elsewhere.

1.1 Hardware performance metrics

Hardware performance counters are available in every modern microprocessor design. They allow the measurement of many (sometimes hundreds) of metrics that are related to the way code is executed on the hardware. Although many of those metrics are unimportant for the developer writing numerical simulation code, some of them can be very useful in assessing resource utilization and general performance properties. A large variety of tools exist, from basic to advanced, that allow easy access to HPM data, and some of them even give optimization advice derived from the measurements.

Fortunately, although there is considerable variation in the kinds of hardware events that are available on different processors (even from the same manufacturer), a rather small subset of them is sufficient to identify the most prevalent performance problems in serial and parallel code. These are available on all modern processor designs. We call a specific combination of hardware event counts and possible other sources of infor-

mation a "signature." Together with information about runtime performance behavior and code properties, signatures indicate the presence of so-called "performance patterns," which help to assess the quality of code and, most importantly, identify relevant bottlenecks to enable a structured approach to performance optimizations.

This paper is organized as follows. In Sect. 2 we introduce a (non-exhaustive) list of performance patterns and the typical metric signatures that go with them. Sect. 3 then presents two case studies from different sectors of computational science, and Sect. 4 gives a summary and outlook to future work.

1.2 likwid-perfctr

Given sufficient experience, simple and lightweight tools are often adequate to accomplish the goals described above. Hence, we restrict ourselves to x86 architectures under the Linux OS and employ the likwid-perfctr tool from the LIKWID toolsuite [6,7]. LIKWID[1] is a collection of command line programs that facilitate performance-oriented program development and production in x86 multicore environments under Linux. The concept of event sets with connected derived metrics, which is implemented in likwid-perfctr by means of performance groups, fits well to the signature approach presented in this paper. We will not go into details on how to employ likwid-perfctr, since other tools and frameworks can do similar things.

2 Performance patterns and event signatures

The following performance patterns have been found to be most useful when analyzing scientific application codes on multicore-based nodes. Other application domains may have different issues, but the basic principle could still be applied. The categorization is to some extent arbitrary, and some patterns are frequently found together.

- **Load imbalance**
  Load balancing issues are an impediment for parallel scalability, and hence performance, and they should be resolved first.
- **Bandwidth saturation**
  Whenever the bandwidth of a shared data path is exhausted, there is a natural limit to scalability. Most frequently this happens on the main memory interface or the (usually shared) outer-level cache (OLC).
- **Strided or erratic data access**
  Cache-based architectures require contiguous data accesses to make efficient use of bandwidth due to the cache line concept. Strided access (often caused by inappropriate data structures or badly ordered loop nests) is one of the most frequent causes for low data transfer efficiency (between cache levels and to/from memory).
- **Bad instruction mix**
  Inefficient code execution due to an instruction mix that is inadequate to solve the problem can be a complex issue. It encompasses diverse effects such as general purpose instruction overhead created by inefficient compiler code (often occurs

[1] "Like I Knew What I'm Doing"

with C++), but also the degree of vectorization or the use of expensive operations like divide and square root.

- **Limited instruction throughput**
  There is always a limit for the number of instructions that can be executed per cycle (e.g., 4 or 6), independent of their types. Even if a code does not hit this limit, it could still suffer from a bottleneck in a specific execution port (such as load or multiply). Finally, dependencies could cause pipeline bubbles, which further diminish the throughput. This pattern is closely related to the bad instruction mix pattern, but tends to require different code optimization strategies.

- **Microarchitectural anomalies**
  This is a very architecture-specific pattern which may have different manifestations depending on the type of CPU. Typical examples are false store forward aliasing, unaligned data accesses or instruction code, and shortage of load/store buffers.

- **Synchronization overhead**
  Barriers at the end of parallel loops or locks protecting shared resources may have a large performance impact if the workload between such synchronization points is too small. This pattern may also incur secondary effects like load imbalance or bad instruction mix (see above).

- **False cache line sharing**
  Different threads accessing a cache line (and at least one of them modifying it) lead to frequent evictions and reloads, impacting performance a lot.

- **Bad page placement on ccNUMA**
  All modern multi-socket servers are of ccNUMA type. Memory-bound codes must implement proper page placement in order to profit from the bandwidth advantages that ccNUMA provides. The two main problems with bad page placement are nonlocal data access and bandwidth contention, with load imbalance as a possible secondary effect.

Each of those patterns can be mapped to one or more "signatures," which consist of a combination of performance behavior (scalability, sensitivity to problem size, etc.) and a particular pattern in raw or derived hardware metrics. While the former is often independent of the underlying architecture, the latter is very hardware-specific. Ideally a given tool should provide these event sets and derived metrics in a similar way on all supported processor architectures. likwid-perfctr tries to support this by "performance groups." In Table 1 we give a correspondence of each performance pattern with its signatures in the performance behavior and to the relevant anomalies in hardware metrics (together with the likwid-perfctr performance group, if available). In some cases the signature also involves information from other sources such as microbenchmarks or static code analysis, since some HPM signatures may be easily misinterpreted. We deliberately do not give any general optimization hints, since optimization is only possible trough a thorough code review together with a suitable performance model.

	Signature	
Pattern	**Performance behavior**	**HPM (and likwid-perfctr group(s))**
Load imbalance	Saturating speedup	Different count of instructions retired[2] or floating point operations among cores (`FLOPS_DP`, `FLOPS_SP`)
OLC bandwidth saturation	Saturating speedup across cores in OLC group	OLC bandwidth comparable to peak bandwidth of a suitable microbenchmark (`L3`)
Memory bandwidth saturation	Saturating speedup across cores sharing a memory interface	Memory bandwidth comparable to peak bandwidth of a suitable microbenchmark (`MEM`)
Strided or erratic data access	Large discrepancy between simple bandwidth-based model and actual performance	Low bandwidth utilization despite LD/ST domination / Low cache hit ratios, frequent evicts/replacements (`CACHE`, `DATA`, `MEM`)
Bad instruction mix	Performance insensitive to problem size fitting into different cache levels	Large ratio of instructions retired to FP instructions if the useful work is FP / Many cycles per instruction (CPI) if the problem is large-latency arithmetic / Scalar instructions dominating in data-parallel loops (`FLOPS_DP`, `FLOPS_SP`, `CPI` is always measured)
Limited instruction throughput	Large discrepancy between actual performance and simple predictions based on max Flop/s or LD/ST throughput	Low CPI near theoretical limit if instruction throughput is the problem / Static code analysis predicting large pressure on single execution port / High CPI due to bad pipelining (`FLOPS_DP`, `FLOPS_SP`, `DATA`, `CPI` is always reported)
Microarchitectural anomalies	Large discrepancy between actual performance and performance model	Relevant events are very hardware-specific, e.g., stalls due to 4k memory aliasing, conflict misses, unaligned vs. aligned LD/ST, requeue events. Code review required, with architectural features in mind.
Synchronization overhead	Speedup going down as more cores are added / No speedup with small problem sizes / Cores busy but low FP performance	Large non-FP instruction count[2] (growing with number of cores used) / Low CPI (`FLOPS_DP`, `FLOPS_DP`, `CPI` always measured)
False cache line sharing	Very low speedup or slowdown even with small core counts	Frequent (remote) evicts (`CACHE`)
Bad ccNUMA page placement	Bad/no scaling across locality domains	Unbalanced bandwidth on memory interfaces / High remote traffic (`MEM`)

Table 1: Performance patterns and corresponding signatures for parallel code on multicore systems

3 Case studies

3.1 Abstraction penalties in C++ code

The basis for this case study is a recent analysis of Expression Template (ET) frameworks for basic linear algebra operations [8,9]. While classic ETs show good performance for simple BLAS1-type (vector-vector) loop kernels, they have severe problems with BLAS2- and BLAS3-type operations and sparse arithmetic, since they are based on accesses to individual elements of data structures and have no notion of standard optimizations for nested loops. "Smart Expression Templates" (SETs) ameliorate this problem since they provide a high-level approach to complex loop nests and can substitute the whole operations by calls to optimized libraries (such as Intel MKL) or well-written plain C or compiler intrinsics code. (S)ET approaches must also be compared to standard coding techniques like operator overloading (which is plagued by the generation of temporary objects) and classic C loop nests.

Convoluted code like the one generated by strongly abstract C++ source when dealing with matrix-type operands typically shows the "**bad instruction mix**" pattern, since a lot of instructions are generated that are not actually needed to solve the problem. In the Expression Template example this shows most prominently in the number of retired instructions. Table 2 shows events and derived metrics for a 5000×5000 matrix-matrix multiplication using four different code versions: The "Classic" code uses traditional overloading of `operator*()` so that an expression like `C=A*B`, with `A`, `B`, and `C` being objects of some matrix class, results in a call to a function that implements a naive version of the matrix multiply and returns the result as a temporary copy. "Boost *uBLAS*" supports matrix operations directly with a slightly different syntax, and avoids the temporary (which is the main reason for using ETs in the first place). *Eigen3* is an SET framework that is able to recognize arithmetic expressions involving complex data types and employs an optimized version of the operation. However, it still relies heavily on the inlining capabilities of the compiler. "MKL dgemm" denotes a direct call to the vendor-optimized BLAS library for Intel processors.

The results show a striking agreement in the number of generated instructions between the Classic and *uBLAS* versions, although the performance of the Classic code is a factor of eight higher. In both cases the compiler has generated bloated, scalar machine code; the reason is that the access to individual matrix elements is strongly abstracted. The "Classic" code, uses an overloaded `operator(int,int)` for accessing the matrix elements in the loop nest, and the *uBLAS* relies on a similar mechanism. Both strategies impede the compiler's view on what the actual operation is and limit its optimization capabilities. The result is a factor of five to six in retired instructions compared to *Eigen3* and MKL, of which a factor of two can be attributed to scalar (as opposed to SIMD-parallel) instruction code.

The fact that *uBLAS* is so much slower than the Classic version results from a very unfortunate loop ordering, leading to stride-5000 accesses to one of the matrices in the product (for details see [8]). As a consequence, the code becomes latency-dominated and makes inefficient use of the memory bandwidth (second column in Table 2). This

[2] Load imbalance and frequent synchronization often go together, leading to large non-FP instruction counts that are caused by spin-waiting loops.

is the "**strided data access**" pattern at work. The Classic version, despite its inefficient machine code, at least implements a loop nest that has stride-one accesses to all relevant data structures. This is also reflected in the CPI metric (fourth column), which indicates massive pipeline bubbles due to long-latency loads. The Classic version can still not exhaust the available memory bandwidth for a single thread (see "STREAM"), although the kernel should be bandwidth-bound. This is again a consequence of the code spending too much time with in-core execution.

The *Eigen3* version, with the help of optimized kernels and massive inlining, achieves 76% of the MKL performance, which is impressive for compiler-generated code. The memory bandwidth of the MKL code is not so different from the *uBLAS* version, but this is purely coincidental: Memory access is not a bottleneck for the highly optimized dgemm implementation.

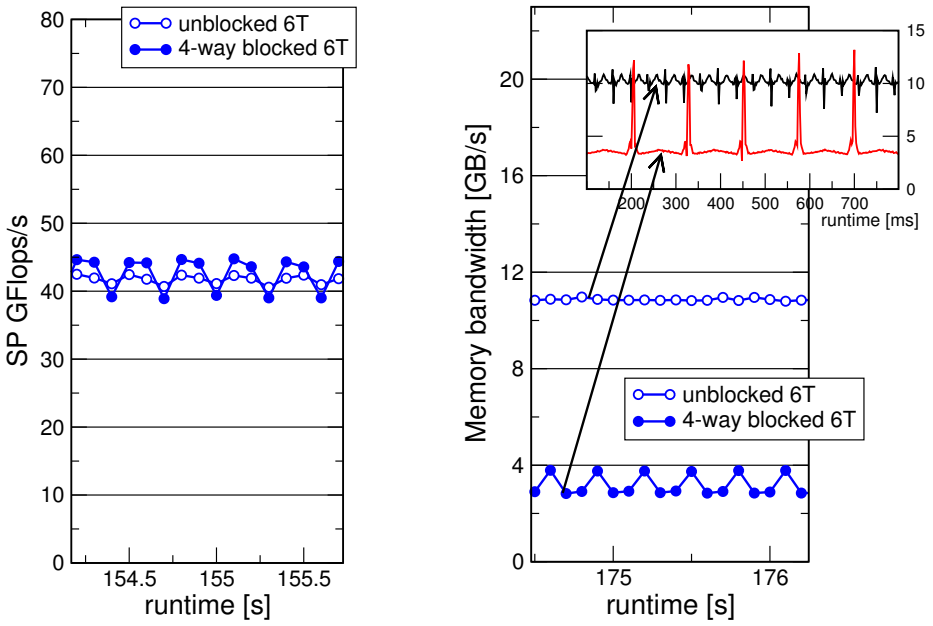
	Memory Bandwidth [MByte/s]	Total Retired Instructions [10^{11}]	Cycles Per Instruction (CPI)	Performance [MFlop/s]
STREAM	11814	—	—	—
Classic	5314	12.5420	0.440861	1249
Boost *uBLAS*	630	10.1207	4.61834	156
Eigen3	371	2.1014	0.41168	8555
MKL dgemm	531	2.03448	0.321115	11261

Table 2: Hardware counter performance analysis of the single-threaded multiplication of two large dense matrices ($N = 5000$). The given STREAM bandwidth is the practical limit for one thread on the used processor. (Adapted from [8])

3.2 Medical image reconstruction by backprojection

This case study was part of a work aiming at optimized volume reconstruction on multicore processors [10]. The optimization target is the open benchmark RabbitCt, which implements volume reconstruction by backprojection, which is the computational bottleneck in many medical imaging applications. From a simple roofline model analysis [11] the algorithm was identified to be bandwidth-limited on the platforms investigated. However, it turned out that a complex combination of patterns is involved here: **memory bandwidth saturation**, **limited instruction throughput**, and **load imbalance**.

In a first attempt it was checked if the memory bandwidth saturation pattern applies using the `MEM` performance group of likwid-perfctr. To get a meaningful bandwidth baseline a benchmark was constructed that mimics the basic data access pattern, which in this case is an array update kernel (`A(:)=s*A(:)`). The Intel Westmere processor used in the analysis can sustain 20.3G̃B/s using all cores of one socket for this operation. Bandwidth measurements with likwid-perfctr revealed that the application showed a much lower bandwidth of roughly 10G̃B/s. Hence, memory bandwidth saturation is



(a) FP performance

(b) Memory bandwidth

Fig. 2: Performance counter timeline monitoring of floating-point performance (a) and memory bandwidth (b), comparing blocked/nonblocked variants of the best implementation on one full Westmere socket (six cores) at 100 ms resolution. The inset in (b) shows a zoomed-in view with 2 ms resolution. (Adapted from [10])

not a limiting bottleneck for this implementation on Intel Westmere. A static code analysis (using the Intel IACA tool [12]) showed that the scattered load of necessary pixel data into SIMD registers requires a large number of instructions on the instruction code level. This makes the code limited by instruction throughput, which is not evident from the high-level language implementation. The measured performance was near the prediction of this static loop body runtime analysis, which was based on the instruction throughput capabilities of the architecture. A final confirmation that the code is indeed limited by instruction throughput was achieved by comparing with measured CPI values, which were in accordance with the static L1 cache prediction.

For a more severely bandwidth-starved architecture (Intel Harpertown) a cache-blocked version of RabbitCt was implemented, showing a significant performance improvement. This code version was also run on the Westmere platform for comparison. The result of a likwid-perfctr timeline measurement of the floating-point performance and main memory bandwidth is shown in Fig. 2: The blocking effectively lowers the bandwidth demands, but there is no impact on the overall performance.

One of the other applied optimizations was a work reduction strategy; after cutting over 30% off the total work the runtime benefit was still negligible. To check a possible

Core Id	0	1	2	3	4	5
FP_COMP_OPS_EXE_SSE_FP_PACKED [10^{10}]	2.74	9.39	9.23	9.30	9.29	3.07

Table 3: Instruction count per core for packed SSE arithmetic floating point instructions of the RabbitCt benchmark without load balancing

Core Id	0	1	2	3	4	5
FP_COMP_OPS_EXE_SSE_FP_PACKED [10^{10}]	7.16	7.17	7.16	7.17	7.17	7.17

Table 4: Instruction count per core for packed SSE arithmetic floating point instructions with round robin scheduling

load imbalance the instruction count on the cores was measured using likwid-perfctr with the `FLOPS_SP` group. As the innermost loop body is fully vectorized the number of packed (vectorized) arithmetic instructions is a good indicator for a potential load imbalance. Table 3 shows the results of a likwid-perfctr measurement of the packed SSE arithmetic floating point instructions. Evidently the outer threads have only one third of the workload of the others in terms of packed FP instructions. The runtime for this case is 61.72 s. A simple way to improve load balancing in in OpenMP was to change the loop scheduling to a round robin distribution, using `static,1`. Results are shown in Table 4: The load imbalance was removed and the runtime was reduced to 43.9 s.

4 Summary and outlook

This paper presented an initial proposal of a structured usage of HPM embedded in an overall software engineering process. We classify relevant performance patterns and formulate signatures which indicate that a certain pattern applies. The signatures are based on HPM data alone or combined with other sources of information such as microbenchmarking data and static code and algorithmic analysis. We are aware that this approach nevertheless requires an intimate knowledge of the algorithm, the code and the hardware. Still we believe that there is no alternative to a performance engineering process build on knowledge of the programmer himself. The application of our approach was illustrated on the example of two case studies. Future work involves further settlement of the performance patterns and corresponding signatures. This will be achieved by applying the approach to various practical examples.

References

1. F. Günther, M. Mehl, M. Pögl and C. Zenger. *A cache-aware algorithm for pdes on hierarchical data structures based on space-filling curves*. SIAM Journal on Scientific Computing **28(5)**, (2006) 1634–1650. `http://www5.in.tum.de/pub/int/guenther_siam06.pdf`

2. T. Klug, M. Ott, J. Weidendorfer and C. Trinitis. *autopin - automated optimization of thread-to-core pinning on multicore systems*. T. HiPEAC **3**, (2011) 219–235.
3. H. Chen, W. chung Hsu, J. Lu and P. chung Yew. *Dynamic trace selection using performance monitoring hardware sampling*. In: *in Proceedings of the 1st International Symposium on Code Generation and Optimization*. 79–90, (2003).
4. R. de la Cruz and M. Araya-Polo. *Towards a multi-level cache performance model for 3d stencil computation*. Procedia Computer Science **4(0)**, (2011) 2146 – 2155. ¡ce:title¿Proceedings of the International Conference on Computational Science, ICCS 2011¡/ce:title¿. `http://www.sciencedirect.com/science/article/pii/S1877050911002936`
5. W. Pfeiffer and N. Wright. *Modeling and predicting application performance on parallel computers using hpc challenge benchmarks*. In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. ISSN 1530-2075, 1 –12, (2008).
6. J. Treibig, G. Hager and G. Wellein. *LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments*. In: *PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures* (IEEE Computer Society, Los Alamitos, CA, USA), 207–216, (2010). `http://dx.doi.org/10.1109/ICPPW.2010.38`
7. *LIKWID performance tools*. `http://code.google.com/p/likwid`
8. K. Iglberger, G. Hager, J. Treibig and U. Rüde. *Expression templates revisited: A performance analysis of current ET methodologies*. SIAM Journal on Scientific Computing **34(2)**, (2012) C42–C69. `http://dx.doi.org/10.1137/110830125`
9. K. Iglberger, G. Hager, J. Treibig and U. Rüde. *High performance smart expression template math libraries*. In: *Proceedings of APMM 2012, the 2nd International Workshop on New Algorithms and Programming Models for the Manycore Era at HPCS 2012, July 2-6, 2012, Madrid, Spain*. (Accepted).
10. J. Treibig, G. Hager, H. G. Hofmann, J. Hornegger and G. Wellein. *Pushing the limits for medical image reconstruction on recent standard multicore processors*. International Journal of High Performance Computing Applications (Accepted). `http://arxiv.org/abs/1104.5243`
11. S. W. Williams, A. Waterman and D. A. Patterson. *Roofline: An insightful visual performance model for floating-point programs and multicore architectures*. Tech. Rep. UCB/EECS-2008-134, EECS Department, University of California, Berkeley, Oct 2008. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-134.html`
12. *Intel architecture code analyzer*. `http://software.intel.com/en-us/articles/intel-architecture-code-analyzer/`