# Chapter 3
# likwid-bench: An Extensible Microbenchmarking Platform for x86 Multicore Compute Nodes

**Jan Treibig, Georg Hager, and Gerhard Wellein**

**Abstract** Microbenchmarking is an essential tool for characterizing modern compute nodes. Apart from determining raw performance capabilities microbenchmarking can be used to aquire input parameters for performance models or mimic the behavior of more complex applications. Many existing microbenchmarks are not extensible and implemented in C or Fortran. One problem with microbenchmarks in a high level language is that many performance issues are only apparent on the instruction level. The code quality of the compiler is an additional source of variation. likwid-bench is a framework enabling rapid prototyping of loop-based, threaded assembly kernels. It eases the process of implementing assembly kernels by providing a portable assembly language independent from any concrete assembler program. likwid-bench already includes many standard microbenchmarking testcases and can be used out of the box as a microbenchmarking tool.

## 3.1 Introduction

Microbenchmarking is an essential tool to investigate the interaction of software with the hardware. It can be used to determine upper limits of performance characteristics on compute nodes as well as pin down anomalies in the microarchitecture of a processor. Another important application is to acquire input parameters of performance models. A prominent example for a microbenchmark is the STREAM benchmark proposed by McCalpin [6], which is the standard for sustained main memory bandwidth and is often used as baseline for the balance metric approach [10]. The theoretical performance numbers published by the hardware

J. Treibig (✉) · G. Hager · G. Wellein
Erlangen Regional Computing Center (RRZE), Friedrich-Alexander Universität Erlangen-Nürnberg, Martensstr. 1, D-91058, Erlangen, Germany
e-mail: jan.treibig@rrze.uni-erlangen.de; georg.hager@rrze.uni-erlangen.de; gerhard.wellein@rrze.uni-erlangen.de

H. Brunst et al. (eds.), *Tools for High Performance Computing 2011*, DOI 10.1007/978-3-642-31476-6_3, © Springer-Verlag Berlin Heidelberg 2012

vendors do often not reflect the performance achievable by real applications. Since instruction code has a large impact on performance, a high level language such as C is not suited to determine these numbers because it introduces another layer of abstraction depending on compiler quality. This is even more important in the x86 world were new instruction set extensions are released with high frequency. In [8] the Open Source LIKWID tool suite [2] was introduced as a set of lightweight performance-related command line tools for the Linux operating system. In [9] new functionality and application scenarios were described, including the benchmarking application likwid-bench. In this paper we want to cover likwid-bench in more detail and elaborate on implementation questions. likwid-bench is a microbenchmarking framework allowing rapid prototyping of stream based, threaded assembly kernels. Because it comes with a set of common microbenchmarking kernels it can be used out of the box as a benchmarking application. It is also a valuable tool for measuring instruction code performance issues for code generators or compilers.

Implementing an assembly kernel with consistent time measurement is non-trivial. On multicore chips threading is required to determine issues like resource contention or bottlenecks. Complex node topologies require strict thread core affinity to get reliable performance characteristics. Finally data placement issues due to ccNUMA effects have to be taken into account.

likwid-bench provides a framework which allows the implementer to concentrate on the instruction code of the benchmarking kernel. The high level assembly language provided eases this process as far as possible without hiding any important aspect. The runtime cares for threading and affinity, for data allocation and placement, for accurate time measurement, and for result presentation. All necessary configuration for benchmark execution is passed on the command line.

This paper is organized as follows. In Sect. 3.3 the software architecture of likwid-bench is described. The special benchmark text file format used to create new benchmarks is introduced in Sect. 3.4. Section 3.6 shows application examples, and Sect. 3.7 presents conclusion and an outlook to future work.

## 3.2 Related Work

About a decade ago existed benchmarking tools for low level measurement of latency and bandwidth of the memory hierarchy on modern cache based processors especially in the UNIX environment. Two prominent examples are LLCBench [4] and lmbench [5]. In [1] the evaluation of hardware performance counter data acquired with PAPI using microbenchmarking is described. In [11] a tool is described to automatically determine hardware parameters later used for automatic performance optimization. That paper puts much emphasis on the use of C as a portable "assembly language." The IPACS project [3] aimed at providing a unified framework for various benchmarking backends and result presentation.

likwid-bench ultimately tries to provide the user with enough insight into the microarchitecture to understand the interaction of application code with the hardware on the lowest sensible level. It does not use C for benchmarking kernels

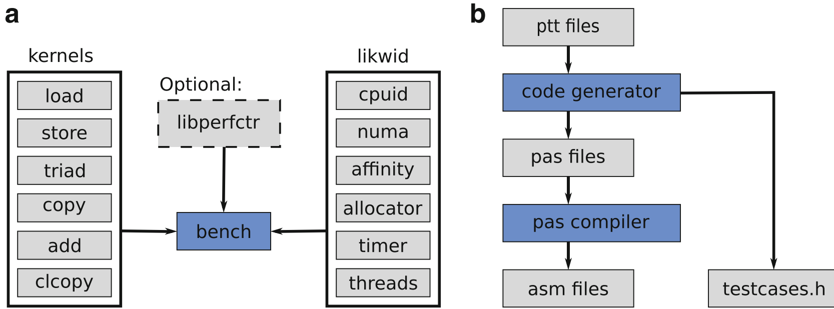


**Fig. 3.1** Software architecture of likwid-bench. (**a**) Architecture. (**b**) Benchmark generation

because the exact instruction code has a crucial influence on performance metrics. We do not know of any project providing a user configurable runtime environment for threaded assembly kernels with the same flexibility in thread and data placement.

## 3.3 Architecture

likwid-bench is builds on the components of the LIKWID toolsuite. It uses the cpuid module to extract topology information and the affinity module to pin threads. The allocator module manages aligned data allocation and placement using the ccNUMA information provided by the numa module. For the threaded execution the threads module provides a flexible interface with thread groups and fast synchronization using the `pthreads` API. Optionally likwid-bench can be configured to use the libperfctr API instrumentation calls providing very accurate performance counter measurements. The overall architecture of likwid-bench is illustrated in Fig. 3.1.

The benchmarking kernels are text files in the .ptt file format, describing the loop body instruction code accompanied by meta information necessary for the execution and result presentation. Benchmarks are named according to the ptt file names. During the build process a perl script reads in all .ptt files in a specific directory and converts them to a high level assembly file format (.pas) and C header with prototype declarations of all testcases. For every testcase an assembly function is generated which is later called from the benchmarking core application.The .pas files are converted to assembler files (currently for the gas assembler) and finally assembled to an object file. The intermediate .pas file provides an abstraction from the assembler format and would allow, e.g., to port likwid-bench to Windows using the Microsoft assembler. All intermediate build products are available allowing to closely review the generated assembly code. To add a new benchmark the user must add a new .ptt file in the bench directory and recompile.

## 3.4 Benchmark .ptt File Format

likwid-bench is focused on stream based loop kernels. Still it can be also used for instruction throughput limited kernels. The following listing shows the file format on the example of the copy kernel:

```
STREAMS 2
TYPE DOUBLE
FLOPS 0
BYTES 16
LOOP 8
movaps    FPR1, [STR0 + GPR1 * 8]
movaps    FPR2, [STR0 + GPR1 * 8 + 16]
movaps    FPR3, [STR0 + GPR1 * 8 + 32]
movaps    FPR4, [STR0 + GPR1 * 8 + 48]
movaps    [STR1 + GPR1 * 8]      , FPR1
movaps    [STR1 + GPR1 * 8 + 16], FPR2
movaps    [STR1 + GPR1 * 8 + 32], FPR3
movaps    [STR1 + GPR1 * 8 + 48], FPR4
```

Data is provided by means of vector streams (STR0, STR1). The necessary number of streams used by the benchmark must be configured accompanied by the data type (at the moment single and double precision floats and integers are supported). For result output it must be further specified how many flops are executed and how many bytes are transfered in a single scalar update operation. The LOOP statement marks the beginning of the loop body. The loop control code is automatically generated, GPR1 being the default loop counter. The number following the LOOP statement configures how many scalar updates are performed in one loop body iteration. The loop counter is incremented by this number after the execution of the loop body. In above example the loop body is four-way unrolled. Since the code performs two updates per instruction due to SIMD vectorization, one loop body iteration performs eight updates. The assembly code must use Intel assembler syntax. For convenience a number of placeholders are provided: Before the LOOP statement setup instruction code can be placed, e.g. to initialize registers.

| | |
|---|---|
| FPR[0-15] | SIMD vector registers |
| GPR[0-15] | General purpose registers |
| STR[0-10] | Registers with stream addresses |
| SCALAR | Vector double precision constant |
| SSCALAR | Vector single precision constant |
| ISCALAR | Vector integer constant |

## 3.5 Command Line Syntax

The command line syntax provides a simple and intuitive but yet powerful interface for thread and data placement. likwid-bench uses the notion of *thread groups*. A thread group is a subset of threads operating on the same data set. For every thread group a workgroup command line option must be present describing the thread and optionally data placement. For thread and data placement the notion of so called *thread domains* introduced is used. A thread domain is an entity on the node shared by a thread group. The top level thread domain is the node, followed by ccNUMA locality domains, sockets, and shared caches. If there are multiple domains of the same type they are numbered consecutively starting with zero. Thread domains are specified using single characters. This allows a simple interface accounting for all current and also future topology developments. At the moment the following thread domains are supported:

| | |
|---|---|
| `N` | Node |
| `M[0-N]` | ccNUMA locality domain |
| `S[0-N]` | Sockets |
| `C[0-N]` | Last level shared cache |

Required information for defining a work group is the thread domain to be used and the size of the working set. This size is the total size used for all streams in a benchmark. The necessary vector lengths for each stream is calculated by the environment (alignment and padding issues are handled automatically). The simplest invocation for likwid-bench using the copy example is:

```
$ likwid-bench -t copy -i 1000 -g 1 -w S0:2MB
```

The `-t` parameter specifies the benchmark type and `-g` allows to set the number of thread groups. At the moment the number of iterations must be given on the command line with the `-i` option. It is planned to make this an optional information with the default ensuring a runtime sufficient to get exact results. The `-w` option configures one thread group. Reasonable defaults are used for all omitted information. In the example above the default is to use all threads available in the thread domain, including logical cores. The following invocation will use a limited number of four threads:

```
$ likwid-bench -t copy -i 1000 -g 1 -w S0:2MB:4
```

Every successive thread is pinned to distinct cores. This means that for, e.g., an Intel Quad core Nehalem processor four threads will be pinned to all physical cores. If six threads are specified the physical cores will be used first and the remaining two threads will be pinned to the first two logical cores. This simple approach of filling up the thread domains has one disadvantage: It does not support more sophisticated thread placement policies. This is apparent on the upcoming AMD “Interlagos”

processor: Two cores share one floating point unit. With the current approach it is not possible, e.g., to skip every other core to only use one core per floating point unit. We plan to extend the current syntax to allow strides without complicating the basic approach.

For data placement the default is that all data is placed in the same thread domain the threads are running in. Optionally the exact placement for every stream used can be configured. The copy benchmark has two streams; this value can be queried from likwid-bench with the command likwid-bench -l <testcase>. The following command:

```
$ likwid-bench -t copy -i 1000 -g 1 \
  -w S0:2MB:4-0:S1,1:S1
```

will place both streams (0 and 1) in the threading domain socket 1 (S1). Each stream is page aligned per default. Optionally an offset may be specified separately for each stream.

## 3.6 Examples

In the following we present possible applications of likwid-bench. The examples demonstrate critical issues of modern multicore compute nodes: Memory and last level cache bandwidth saturation, and ccNUMA characteristics.

### *3.6.1 Identifying Bandwidth Bottlenecks*

Determining bandwidth bottlenecks on a compute node is critical for judging application performance. If it is known what bottlenecks are apparent on a node, performance limitations can be better understood and techniques can be applied to work around those bottlenecks. This applies to main memory but also to shared last level cache bandwidths.

likwid-bench provides special cacheline wise variants of basic data operations such as load, store, and copy. Because in our case the raw capabilities of the memory subsystem are needed the runtime spent in executing instructions must be kept at a minimum [7]. This is ensured by only touching one data item per cacheline, which forces each cacheline into the L1 cache but minimizes the time spent on instruction execution. These benchmark variants show the peak cache bandwidths achievable by software. Figure 3.2a shows last level cache bandwidth scalability when increasing the number of threads. Clearly the Intel Westmere processor shows a bandwidth bottleneck starting at three threads. It can also be seen that while for the load operation the processor is able to saturate its peak cache bandwidth, contention occurs for the copy operation leading to roughly half of the possible peak bandwidth. The successor SandyBridge solves both issues, since it incorporates a cache design which uses multiple segments connected by a segmented ring bus. This makes the

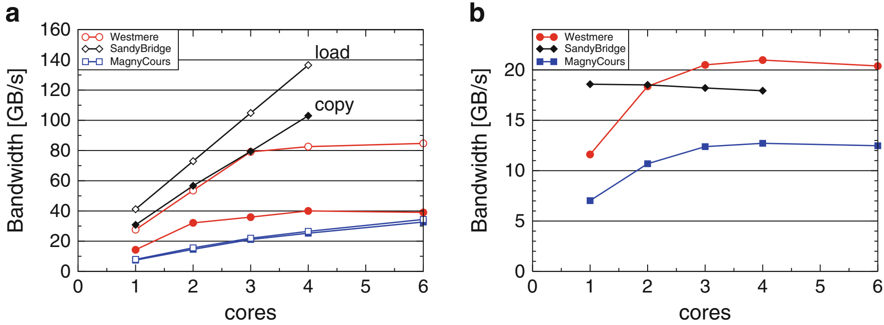


**Fig. 3.2** Bandwidth saturation of shared last level caches (**a**) and main memory (**b**). In both cases special cacheline wise versions of load and copy (**a**) or copy (**b**) are used (Intel Westmere Xeon X5670, Intel SandyBridge Core i7-2600 and a AMD MagnyCours Opteron 6176SE)

L3 cache a parallel device and provides scaling L3 bandwidth. Also the difference in performance between load and copy is much smaller.

For AMD MagnyCours a six core element as the largest entity with a shared cache is used for comparison. The L3 cache on MagnyCours is not well suited for bandwidth-demanding operations. It scales well up to six cores, but on a low absolute performance level. Up to three cores the bandwidth is not significantly larger than main memory bandwidth. Still on the full socket it can draw level with Intel Westmere. There is no difference between load and copy, which can be attributed to the write allocate exclusive cache design, triggering the same number of cacheline transfers for load and copy.

Figure 3.2b shows memory bandwidth scaling inside one ccNUMA locality domain. Both Intel Westmere and AMD MagnyCours show a bandwidth saturation starting with three threads. The desktop SandyBridge processor is able to saturate its bandwidth with a single thread.

### *3.6.2 Characterizing ccNUMA Properties*

Cache coherent non-uniform memory architectures are used today to provide scalable memory bandwidth performance on the node level. This comes at the price of an increased complexity for the programmer with regard to performance. While on current Intel machines the ccNUMA locality domain is equivalent to the socket, on recent AMD processors one physical socket is made up of two dies eachg with its own locality domains. This means that on a four socket AMD MagnyCours system eight locality domains exist connected by the Hypertransport bus. To obey ccNUMA locality is therefore crucial to get good performance on such nodes. With likwid-bench it is easy to mimic application characteristics with regard to ccNUMA

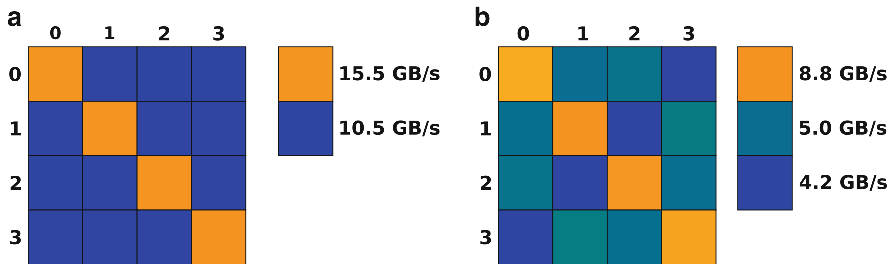


**Fig. 3.3** NUMA bandwidth map of Intel Nehalem EX four socket node (**a**) and AMD Magny Cours two socket node with four ccNUMA locality domains. The Testcase is a standard memory copy operation with point to point transfer between two locality domains. All cores of a locality domain are used. Bencwidth result whoen are effective bandwidth seen by the application, i.e., not counting write allocate transfers. (**a**) Nehalem EX 4S. (**b**) Magny Cours 2S

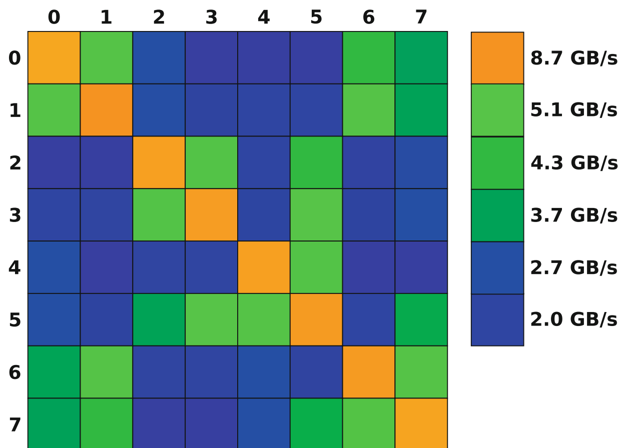


**Fig. 3.4** NUMA bandwidth map of four socket AMD MagnyCours system using a standard memory copy operation

performance. In recent releases of the likwid tool suite a small perl script is included which automatically creates a graphical ccNUMA bandwidth map of a complete node based on a memory copy operation.

Figure 3.3a shows the NUMA bandwidth map for an Intel Nehalem EX four socket node. Surprisingly it has only two performance domains: Local with 15.5 GB/s and remote with 10.5 GB/s bandwidth. On the two socket MagnyCours node with four locality domains three performance domains can be distinguished. Still the difference between the remote domains is negligible. The drop from local to remote is larger on this AMD system than on the Intel Nehalem EX. Figure 3.4 shows the bandwidth map of a four socket AMD MagnyCours system with eight locality domains. This system shows a strong NUMA characteristic with a bandwidth drop for remote accesses to only 25 % of the local bandwidth.There is

a large variation between remote accesses of over a factor of 2. The results cannot be fully explained by the Hypertransport topology on the node alone. It is suspected that routing issues influence the results.

## 3.7 Conclusion and Outlook

likwid-bench is a flexible microbenchmarking platform for streaming loop kernels. It offers great flexibility with regard to thread and data placement and allows to measure many performance-relevant aspects of modern compute nodes. The inherent limitation to a certain family of architectures can also be seen as an advantage because it allows full control over the executed instruction code. We have shown the application of likwid-bench on several examples.

One drawback of the current implementation is that it is restricted to simple stream based data access. To mimic the behavior of more complex algorithms like, e.g., stencil codes multi-dimensional data layouts have to be supported. Also the current syntax specifying thread placement is not flexible enough as it does not allow free placement but is based on filling up thread domains.

One aspect of likwid-bench is to form a community platform for exchanging optimal implementations of low level kernels on different platforms. This point will be in the focus for future activities. There are plans to provide domain specific kernel packages which can be installed and reviewed to get information how to implement algorithmic primitives.

**Acknowledgements** We are indebted to Intel Germany for providing test systems and early access hardware for benchmarking. This work was supported by the Competence Network for Scientific and Technical High Performance Computing in Bavaria (KONWIHR) under the project "OMI4papps."

## References

1. Araiza, R., Pham, T., Aguilera, M.G.: Towards a cross-platform microbenchmark suite for evaluating hardware performance counter data. In: Richard Tapia Celebration of Diversity in Computing Conference, Albuquerque. New Mexico (2005)
2. Homepage of LIKWID tool suite. http://code.google.com/p/likwid/
3. Kredel, H., Merz, M.: The design of the IPACS distributed software architecture. In: 2nd Workshop on Distributed Objects Research, Experiences Applications (DOREA 2), Las Vegas (2004)
4. LLCbench – Low Level Architectural Characterization Benchmark Suite *Homepage*. http://icl.cs.utk.edu/projects/llcbench/
5. LMbench – Tools for Performance Analysis *Homepage*. http://lmbench.sourceforge.net
6. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. In: IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter. New York (1995)

7. Treibig, J., Hager, G.: Introducing a performance model for bandwidth-limited loop kernels. In: Proceedings of the Workshop Memory issues on Multi- and Manycore Platforms at PPAM 2009, Wroclaw (2009)
8. Treibig, J., Hager, G., Wellein, G.: LIKWID: a lightweight performance-oriented tool suite for x86 multicore environments. In: Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego (2010)
9. Treibig, J., Hager, G., Wellein, G.: LIKWID performance tools. In: Bischof, C., et al. (eds.) Competence in High Performance Computing 2010, pp. 165–175. Springer, ISBN 978-3-642-24025-6 (2012)
10. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**, 65–76 (2009). New York
11. Yotov, K., Pingali, K., Stodghill, P.: X-ray: a tool for automatic measurement of hardware parameters. In: Second International Conference on the Quantitative Evaluation of Systems, Torino. IEEE Computer Society, Los Alamitos (2005)