Vol.2017-HPC-160 No.13
2017/7/26

A Memory Performance Analysis Tool Based on Linux Perf

Christian Helm[1,a)] Kenjiro Taura[1,b)]

Abstract: In current high performance computing applications many performance bottlenecks are caused by the memory system. Such performance bugs are hard to identify precisely. Thus analysis tools play an important role in performance optimization. Especially hardware assisted instruction sampling has gained attention for performance analysis. We present a concept for a tool implementation which relies on Linux perf as backend to record instruction sampling data. Our implementation provides support for all recent processor architectures. It combines the perf data with other sources and makes the combined data easily accessible through a database for future experiments and tool development. It is greatly simplifying the access to instruction sampling data and overcomes perf's major weaknesses. Using this database we plan to implement new visualizations and analysis methods.

1. Introduction

Finding the cause of performance problems in modern HPC applications can be a difficult task. Memory accesses are a major contributor to bad performance and unsatisfactory parallel scaling. Throughout this paper the term memory refers to the memory system from the L1 caches to DRAM. Memory is not referring to permanent storage like HDDs.

CPU speed has been increasing much faster than memory speed. The gap between memory performance and processor performance has widened over the years. Because of this every modern processor employs many techniques to provide faster access to data. For example cache memory. While caches increase the average performance cache misses can occur and limit the performance gain. The cache capacity is limited and it might be too small to contain all required data. On multicore systems there are usually shared caches on the lower levels of the hierarchy. Different cores compete for the cache capacity on those shared caches evicting each others data. Conflict misses occur due to limited associativity of caches. Shared data which is used and modified by multiple cores causes invalidation of cache lines and increased communication. All those effects increase the memory access latency. The DRAM is connected to the processor with a limited bandwidth. If an application requires more bandwidth than there is available the applications performance will be bound by the memory performance. Today multi socket systems are common. In those systems every processor has its own DRAM. All DRAMs belong to the same shared address space. But an access to a remote DRAM will result in higher latency than an access to a local DRAM. Bandwidth to remote DRAMs is limited because of the used communication protocols between the processors (e.g. Intel QPI). Thus wrong allocation of data can have negative implications on performance. Figure 1 shows a typical hardware architecture. Overall the observed latency depends on where the requested data is actually stored. For example in an Intel Nehalem processor a local cache hit in L1 cache has a latency of 4 cycles and an access to a remote DRAM has a latency of 310 cycles. Respectively the read bandwidth is limited to 9.1 GB/s on a remote DRAM compared to 45.6 GB/s on a local L1 cache [1].

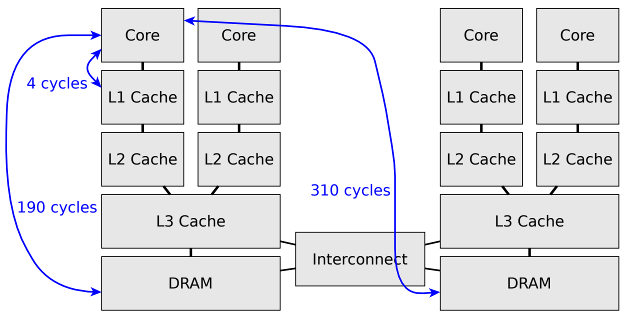


Fig. 1: A typical multi processor system with multiple cache levels and NUMA memory arrangement. Blue lines and numbers show the access latency from core to memory.

Looking at the software side, nowadays many applications are bound by memory performance rather than CPU performance and a lot of effort is spent optimizing those programs. An example is mentioned in [10] where an application is run on a NUMA system and performs mainly matrix multiplications. 63% of its memory accesses are on remote memory.

1 The University of Tokyo
a) christian@eidos.ic.i.u-tokyo.ac.jp
b) tau@eidos.ic.i.u-tokyo.ac.jp

A first optimization lead to a performance improvement of up to 15%. The matrices where allocated interleaved on all nodes. This even increases the amount of remote access but it decreases contention on one node which leads to the higher performance. A second optimization leads to a performance improvement of up to 41%. It was found that one matrix that causes the most remote accesses is never modified. Thus this matrix can be duplicated to all nodes and remote accesses and contention are decreased.

There are many general profiling tools and they are frequently used when optimizing applications. They can help programmers to identify the functions where the most execution time is spent. Those sections are called hot spots and are typically the point where the programmer begins to look for optimization opportunities. An example is shown in Figure 2. It can be seen that 25% of the execution time is spent in a memcpy function linked through libc. Another 25% is spent in the function called worker. 21% are spent in the function called make_array. But only about 2% are spent in the function itself. Its children account for the majority of the execution time. The output has been truncated because it reports many more functions.

```
# Samples: 28K of event 'cycles:pp'
# Event count (approx.): 21724696824
#
# Children      Self   Shared Object     Symbol
# ........  ........  ................  ...............
#
    25.67%    25.60%   libc-2.23.so      __memcpy_sse2
           25.67%
                __memcpy_sse2

    25.66%    25.60%   mbw               worker
           25.66%
                worker

    21.71%     1.94%   mbw               make_array
           21.71%
                make_array
...
```

Fig. 2: Output of *perf report* in a typical profile.

But those profilers can not answer what is going on inside those functions. Modern processors have many possible bottlenecks that can cause performance problems if the software is not optimal. It is difficult to figure out what is going on inside the memory hierarchy without any tool support. Thus specialized tools are necessary which can provide more answers than simply finding hotspots. Programmers who want to achieve higher performance have a desire to figure out quickly which performance problems occur and what is the cause for those problems. It is very useful if a tool can understand and report the optimization potential of fixing a reported performance problem. Also code locations and data responsible for the performance problem should be identified by a tool.

To get metrics about memory performance detailed information from the hardware is needed. Modern processors offer two different ways to get performance data. First there are performance counters. Performance counters are registers which can be configured to count certain events. For example cache misses or branch mispredictions. There is a fixed number of registers available. For example 3 fixed and 4 programmable on Intel Haswell processors. There are hundreds of events available which can be selected for counting. The counter values are read at certain intervals and saved in memory or on disk. At this time the current instruction pointer or call stack can be recorded. The overhead mainly depends on how often and how many counters are read. It is difficult to attribute a certain counter value to a specific piece of code because of order processing, speculative execution and reading counters more often increases the overhead. The second method is instruction sampling. Instruction sampling works by marking an instruction and observing its execution as it goes through the pipeline of the processor. For load instructions detailed information can be obtained. For example the load latency, the actual place where the data was found (L1, L2, L3, remote or local DRAM) and the coherency protocol state at the time of access. AMD calls this method Instruction Based Sampling (IBS). Intel calls it Precise Event Based Sampling (PEBS). The overhead of the sampling method is low since there is dedicated hardware for observing the instructions. Data gathered with instruction sampling can be attributed to precise code locations.

2. Related Work

Numerous memory performance analysis tools have been developed over the years. This paper is limited to profiling tools for Linux since most HPC applications are run on Linux systems. This paper also only covers profiling tools that use metrics from real hardware. Tools that are completely based on simulation are not considered in this study because of the high overhead they usually have.

2.1 Linux Perf

Perf, which is included in the Linux kernel is a general purpose profiler not limited to memory performance. Perf is well maintained and offers the best support for different and new hardware architectures out of all considered tools. Perf supports the usage of performance counters and instruction sampling. The user is responsible to select which events to count. The user has to know which events are meaningful and there are hundreds of events available to choose. It requires knowledge about the hardware and its potential bottlenecks to configure the right events. It is a powerful tool but it also requires lots of experience to use efficiently.

Perf is a command line tool and there is no possibility for visualizing metrics using Perf. In addition to the text output there is an interactive text interface (TUI). Accessing the source code from this interface is supported and there are basic filtering functions to restrict the printed data. The whole perf data must be re-read every time a new type of evaluation is requested. This can easily take minutes for realistic workloads. The evaluation for memory metrics is limited compared to the use of the profiler. Dynamically allocated data can not be resolved.

2.2 ScaAnalyzer and HPCToolkit-NUMA

The authors have published two papers [2], [3] related to this tool which is implemented as an extension of HPC-Toolkit. As the names suggest ScaAnalyzer focuses on parallel scalability problems and HPCToolkit-NUMA is specialized for detecting NUMA problems. Unfortunately the tools have not been made available by the authors. Still the papers introduce valuable concepts.

The memory architecture is separated into layers to simplify analysis. Private layer (L1 and L2), Shared Layer (Shared L3 and DRAM) and NUMA Layer (remote socket DRAM). Performance problems are attributed to one of the layers.

To identify scalability bottlenecks differential analysis is applied. Basically an application is run twice increasing the number of cores used. ScaAnalyzer quantifies the scaling loss in the memory by comparing both runs. It uses the latency information and the scalability information to judge the optimization potential. High latency and high scalability losses indicate high benefit from optimization. High latency and low scalability losses indicate memory bottlenecks that are not related to scalability.

ScaAnalyzer and HPCToolkit-NUMA are GUI tools that allow to browse the source code which is augmented with metrics.

As a low level interface for controlling hardware instruction sampling it uses the perfmon2 library. Identification of the first touch of a page is implemented using a custom SIGSEGV handler. First, new pages are created protected. Because of this the SIGSEGV handler is called upon the first access to every page. Inside the handler the call stack can be recorded. Afterwards the original permissions of the page are restored and the original access can be executed.

2.3 Intel VTune Aplifier XE

VTune Aplifier XE by Intel is a general purpose profiling tool but it also has some specialized memory performance features. Many of them were added in the considered 2016 Linux version [4]. The tool is available as binary and is only one out of the evaluated tools that is not free of charge.

All features are accessible through a single GUI application. The source code can be viewed inside of the application. DRAM and QPI bandwidth can be visualized using a histogram. Based on this histogram the code locations of high bandwidth usage can then be selected from a table. It offers more visualizations like a time resolved display of the used bandwidth and tables support many different memory related performance metrics. It is also possible to do a data centric analysis to show the objects responsible for high bandwidth utilization.

Data is obtained though a custom driver which only supports Intel processors. Performance counters and instruction sampling are used. Dynamic memory allocations and stack frames are tracked to resolve variables.

2.4 Memprof

Memprof [5] is a profiler for NUMA multicore systems published in 2012. Along with the paper the source code was published [6].

Data is obtained using three different sources. First, the object life cycle tracking which records allocations and disallocations of dynamic memory. Second, the life cycle of threads is tracked through kernel hooks for creation and destruction of threads. Third, memory access instructions are sampled to track memory accesses. They use a custom kernel module to control the hardware and record the data. It is limited to AMD processors.

Using this data the thread event flow and object event flow is built. The thread event flow lists the memory accesses performed by each thread. The object event flow shows which threads access an object. For each of those access entires the latency, access type (read/write) and call chain is saved. All event flows are chronologically sorted. Using these event flows indicators for performance problems can be found. For example objects that are accessed from multiple threads running on different nodes.

The evaluation features that come with the tool are limited to text based output. The percentage of remote DRAM accesses can be displayed. A memory profile can be created where the objects, functions and object accesses which cause the highest delays can be listed.

2.5 Aftermath

Aftermath [7] is graphical tool for performance analysis of fine-grained task-parallel applications. It is not limited to memory performance but also has support for other hardware metrics like branch mispredictions. It has been developed to be used together with the OpenStream [8]. It reads a trace generated by OpenStream. Hardware metrics are recorded using PAPI [9] which relies on performance counters.

The main visualization is a gantt chart timeline view of all cores in the system. It can be overlayed with hardware metrics thus providing time resolved analysis and attribution to tasks and cores. For example the percentage of remote DRAM accesses can be shown. Using execution time of tasks on certain cores can help identify NUMA problems. A connection to the source code is possible on the granularity of tasks but not on the level of individual instructions. With filtering only certain tasks with an execution time in a certain range or tasks that write to a specified NUMA node can be selected.

2.6 DProf

This paper [10] and associated thesis [11] describe a tool to locate cache performance bottlenecks. The source code of this tool is available online [12]. A modified Linux kernel is required to run this analysis tool.

DProf has four different views. First, the data profile. It is a list of datatypes sorted by the total number of cache misses. Second, the miss classification view. It shows which type of misses (capacity, associativity conflict, true sharing or false sharing) are the most common for each data type. Third, the working set view. It shows which data types are the most active and how much are active at a given time. It can also be shown which associativity sets are used which helps to find if certain data types are aligned with each other and cause conflict misses. Last, the data flow view. It shows which functions access a given data type. It can also indicate when an object is accessed from multiple cores.

To get information about memory accesses AMD IBS and specially configured debug registers and interrupts are used. One object is tracked at a time on all CPUs and the tracked object is changed during the profile run to provide coverage of many but not all objects. Dynamically allocated objects can only be resolved for kernel code.

An important internal data structure is the path trace. It stores the life cycle of a data object from allocate to free including all accesses to the object. Cache hit probabilities and latencies are recorded in each path trace. Some of the detailed analysis features like categorizing the type of cache miss require a simulation.

2.7 NUMA Access Visualizer

This tool's [13] key concept is a visualization that is based on the physical hardware. It is an enhanced version of previous work [14]. The previous version relied on custom code for programming the PMU instead of using a library and it had less sophisticated visualization. The tools itself has not been released by the authors.

Data is gathered using the likwid library [15]. This library uses performance counters to get the used bandwidth. Both for QPI links and main memory. It is open source and supports all recent processors from AMD and Intel.

For each socket there is a visualization similar to a table. The cell entries show the QPI link utilization from one socket to another. The graph in each cell is a time resolved display of the bandwidth. The diagonal from top left to bottom right where source and destination socket are the same shows the memory bandwidth utilization of that socket.

The maximum possible bandwidth is obtained by running a micro benchmark before the actual analysis. This maximum value is then used to color the cells. Red cells indicate a bandwidth saturation problem. Attribution to hardware sockets is possible but not to code or objects.

2.8 Summary

Key findings of the survey are that there exist reliable and wide spread tools that manage to give an overview and statistics about the whole program execution. Those are useful to find what kind of performance problem is limiting an applications performance. But those tools can not help to pinpoint the location in the code where the problem is coming from. These more specialized tools that provide more details often suffer from low potability to across CPU architectures, lacking maintenance and add obstacles for practical use such as requiring custom kernel modules.

3. Implementation Concept

Our concept uses instruction sampling because it can more precisely identify the concerned data and code regions. The central component is Linux perf which is the tool that controls and executes the instruction sampling. We add a few other software components around perf to make its use easier and tailored to the analysis of memory accesses. Figure 3 shows the software components in the proposed implementation. In the beginning a script is executed which sets the

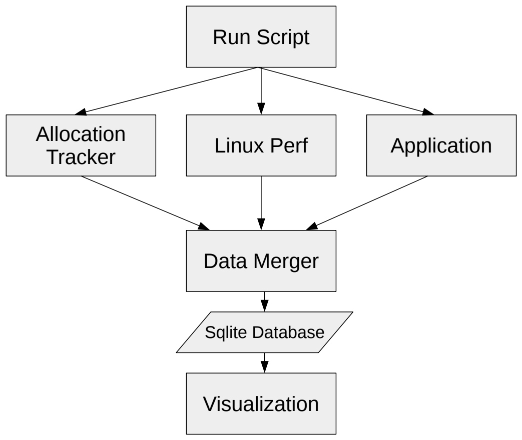


Fig. 3: Components of the proposed tool implementation.

right options for perf to set up the instruction sampling. The script also sets up the allocation tracker and starts the application under test. While the application is running the allocation tracker and perf are recording the relevant info about the application. Once the application has finished execution a data merger is started. This data merger exports the data that was recorded by perf and puts it into a sqlite database. It also adds the allocation trackers data to the database which allows to resolve the accessed data addresses to variable names.

3.1 Perf Settings

A certain set of command line options is used to configure perf for the purpose of analyzing memory accesses. The perf command line that is used to enable instruction sampling for analyzing memory accesses is *perf record -d -W -g -e cpu/mem-loads/pp -k CLOCK_MONOTONIC* The command line options have the following purpose:

- -d activates collection of the accessed data address.
- -W enables reporting the latency of sampled instructions.
- -g activates capturing callstacks for every sample taken. This is an optional setting. It can be disabled if callstacks are not required.
- -e cpu/mem-loads/pp sets the observed event to memory loads. Thus only memory load instructions will be considered. Store instructions could be added by using the additional event cpu/mem-stores/pp. The *pp* refers

to precise events. Which means the event can be associated with the executed code. In this case this is using instruction sampling internally.
- -k CLOCK_MONOTONIC sets the clock source which is required to correlate perf timestamps with timestamps captured by other tools

Because this tool is using instruction sampling and the techniques to gather information from the hardware are very similar to the ones discussed in [3] and [6] we expect a similar overhead which is between 2.5% and 20%. The sampling frequency can be adjusted. The parameters *-c* or *-F* can be used to set the period between samples or the frequency respectively. By adjusting this parameter the overhead and accuracy can be balanced.

3.2 Allocation Tracker

The allocation tracker is based on the one used in the Memprof [5] [6] tool. The memory allocation functions like malloc and free are replaced with the ones defined in the allocation tracker. Those new functions record a callstack, allocated address range and a timestamp before calling the original memory allocation functions. The allocation tracker is using the LD_PRELOAD feature which is available on Linux systems to replace the existing allocation functions. The records are stored in text files. One file for each thread. The original allocation tracker from the Memprof project did not keep track of memory free instructions and did not support freeing memory in the application. This makes the collection and subsequent analysis much easier because address ranges can not be reused. But it limits memory consumption of tools and the applications behavior will be less realistic. We have enhanced the allocation tracker in our tool so that also memory frees are supported during execution.

3.3 Perf Export and Data Merger

Perf stores the recorded sampling data in a binary, perf specific format. Perf provides a scripting interface. This interface allows to access the perf data. The scripting interface supports perl and python. We use the python interface and have written a script which exports the data and puts it into a relational sqlite database. Using this database the sampling data is much easier to process compared to the perf binary format which is not well documented. The export is done using the command: *perf script -s <path-to-script> <script-options>*.

Dynamic memory allocations are only valid for a certain time period. Thus to make a lookup from a given accessed data address to an allocation call stack the timestamp of the sample and the interval in which the allocation was valid has to be considered. Two different tools are used to collect data. Perf does the instruction sampling collecting data about the memory access. The allocation tracker records data about the dynamic memory allocations. Both individually record a timestamp for each entry. In order to have comparable timestamps from both sources both must rely on the same clock source. Perf provides a parameter (-k) to specify the clock source. This feature was introduced with the intention to correlate timestamps between perf and userspace tools. The default clock source for perf is one that is only available within the kernel and can not be accessed by other tools. We use the CLOCK_MONOTONIC clock source which returns a timestamp counted in nanoseconds from the startup of the system. The CLOCK_MONOTONIC is internally based on the Time Stamp Counter (TSC) which is present in all modern x86 processors. Another point to look out for is the synchronization of the hardware clock source between multiple cores and sockets of the same system. The TSC is not affected by power management which may change the operation frequency of the processor. It is also synchronized at startup with all cores across all sockets. Some older processors might not have this feature but this can be verified by checking the presence of the constant_tsc and nonstop_tsc flags.

The data from the allocation tracker is initially stored in text files. To provide easy access to this data it is also imported to the sqlite database. First the data is imported into a table which contains the allocations. The data in the original files is sorted by time. Thus the deallocation entry comes much later in the data than the corresponding allocation entry. When encountering an allocation entry in the text file an entry in the database is created. When encountering a disallocation entry the previously added allocation entry is updated and the correct disallocation timestamp is set. When reading a disallocation entry the corresponding allocation entry can be found in the database because the both refer to the same address range. After all records are imported a reverse lookup is done to update the sample table. Queries to the database usually use the sample table as main input. The sample table contains a foreign key that shows for each sample to which allocation the accessed data belongs. But in the initial import this key is not set because the text file does not contain this information. For each allocation entry in the database all samples which are in the address interval and time interval of that allocation are selected from the sample table. For all of those sample the foreign key is updated to the currently selected allocation entry. By doing this reverse lookup once during the import the following queries to the sample table can be done more efficiently.

3.4 Database Layout

The central table in a database is the samples table. This table holds all the recorded samples. Each sample contains the following information:
- The type of recorded event.
- Process and thread.
- Application and function identifier.
- Instruction pointer.
- Timestamp.
- CPU on which the sample was collected.
- Accessed data address.

- Latency of the instruction.
- Memory Opcode.
- Cache hit or miss.
- Memory hierarchy level where data was found.
- Memory snoop and coherency protocol status.
- Locked memory transaction information.
- DTLB hit or miss.
- DTLB hierarchy level where lookup was found.
- Callstack.
- Allocated memory including address range, time interval and callstack of the allocation.

To keep a relational structure some of the mentioned information is stored in separated tables and linked with foreign keys.

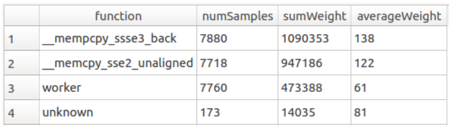
4. Examples of Collected Data

In this section we show some examples of the collected data to provide a better understanding of the data content. The data was recorded by analyzing a memory bandwidth benchmark [16]. This application allocates two large arrays of integers, fills them with content and copies the data from one array to the other. It is using three different methods. First, the memcpy function applied to the whole array. Second, the memcpy function applied to a smaller block size. Third, a loop in which elements are copied one by one. It causes a lot of memory accesses and is a good example to generate many memory access samples. This section is not intended as a case study to evaluate the usefulness of the data and analysis methods. The results can be viewed with a database browser. The following examples all show evaluations of the data that standard perf does not support.

The sql statement in Listing 1 and the table in Figure 4 show the total instruction latency and number of captured memory read samples aggregated by functions.

```
select (select name from symbols where id =
    symbol_id) as function, count(*) as
   numSamples, sum(weight) as "sumWeight",
   sum(weight)/count(*) as averageWeight
from samples group by symbol_id order by
   sumWeight desc
```

Listing 1: Sql query for getting the total latency of fuctions.



	function	numSamples	sumWeight	averageWeight
1	__mempcpy_ssse3_back	7880	1090353	138
2	__memcpy_sse2_unaligned	7718	947186	122
3	worker	7760	473388	61
4	unknown	173	14035	81

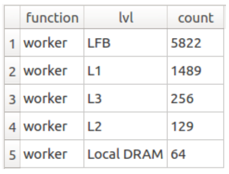
Fig. 4: Latency of read memory accesses of functions.

The unknown function is one that can not be resolved because it is a kernel function for which no debugging information is available. The functions with a high latency are a potential target to check for optimizations.

A second example is displaying the data source of memory reads for a specific function. In this case the function called worker is being selected and the number of references to each memory level for this function is displayed as shown in Figure 5. The required sql statement is shown in Listing 2.

```
select (select name from symbols where id =
    symbol_id) as function, (select name
   from memory_levels where id =
   memory_level) as lvl, count (*) as "
   count"
from samples where function = "worker"
   group by lvl order by count desc
```

Listing 2: Sql query for obtaining the memory hierarchy level of memory references for a specified function.



	function	lvl	count
1	worker	LFB	5822
2	worker	L1	1489
3	worker	L3	256
4	worker	L2	129
5	worker	Local DRAM	64

Fig. 5: Accessed memory hierarchy for read memory references of the function worker. LFB stands for line feed buffer.

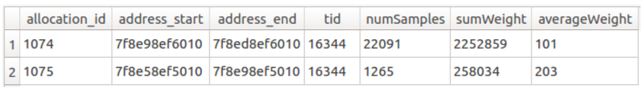
The original perf can only do this kind of memory level analysis for the whole program not for individual functions. In complex programs the whole program analysis often fails to point the programmer to the right locations in the code. Using our implementation it is possible to get this data on for every function with a simple sql statement.

The example in Listing 3 shows how to get the latency information of individual memory allocations. Each allocation corresponds to one object in the code.

```
select allocation_id, printf('%x',
    address_start) as address_start, printf
    ('%x',address_end) as address_end, (
    select tid from threads where id =
    allocations.thread_id) as tid, count(*)
     as numSamples, sum(weight) as
    sumWeight, sum(weight)/count(*) as
    averageWeight
from samples inner join allocations on
    samples.allocation_id = allocations.id
group by allocation_id having allocation_id
     is not null
```

Listing 3: Sql query for getting memory access information for allocated objects.

The clause *having allocation_id is not null* excludes samples for which the accessed data can not be resolved. Figure 6 shows the result of the query. The start address and end address of the allocated memory region and the thread which allocated it are shown. Along with the number of samples, total latency and average latency of accessing these objects.



	allocation_id	address_start	address_end	tid	numSamples	sumWeight	averageWeight
1	1074	7f8e98ef6010	7f8ed8ef6010	16344	22091	2252859	101
2	1075	7f8e58ef5010	7f8e98ef5010	16344	1265	258034	203

Fig. 6: Latency information of objects.

Using this allocation information the call stack of the allocation can be printed. It requires multiple sql queries to print the hierarchical call stack. The output for one of the allocations is shown in Figure 7. Each line shows one stack level. Depending on which information can be resolved due to the availability of debugging symbols the content in the lines can differ. In the first two lines all information can be resolved. There is the binary name, address, function and location in the source file. In third line the location in the source file can not be resolved thus there is only the function name and offset. In the last line no information can be resolved thus there is only the address printed. The file name and line in the source code has been resolved with the GNU binutils tool addr2line.

```
./mbw [0x400ece] make_array at mbw.c:84
 ./mbw [0x400c2d] main at mbw.c:277
  libc.so [0x7f8edca02830] __libc_start_main+0xf0
   [0xafb84220541]
```

Fig. 7: Call stack of the allocation with id 1074.

5. Conclusion and Future Work

We present a tool implementation which is using Linux perf and thus inherits all its good features like extensive and updated hardware support. Our implementation enhances perf to overcome its major weaknesses. We select the right events out of hundreds available ones for the use case of memory performance analysis. The burden to access the data and to implement new analysis tools is reduced by providing the sqlite export capability for perf. In previous approaches the perf source code has been modified to implement new analysis features. Using the prepared data visualizations can be easily created by querying the database and displaying the data. New metrics based on the existing data can be explored. Also our tool adds support to resolve dynamically allocated objects. New analysis of the data can be done efficiently by querying the database instead of having to read the whole perf data file every time a new type of analysis is requested. The unmodified perf that comes with the Linux kernel can be used and it can be installed by using packages available in many Linux distributions. This makes our implementation easy to install and use.

We want to develop a tool to provide guidance to users who are unfamiliar with programming memory optimized applications. Our approach is to create a tool that can give detailed hints for programmers where to look in their programs and identify the kind of performance problem thats limiting the performance at certain places. We want to achieve this by evaluating new metrics to judge the severeness and optimization potential of performance problems and by using new visualizations that help users to better understand their performance problems. We will evaluate our tool using benchmarks such as PARSEC [17] and other realistic applications to proof that our tool can give valuable insights for programmers who are aiming to improve the performance of their applications.

References

[1] Hackenberg, D. and Nagel, W. E.: Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems, *IEEE/ACM International Symposium on microarchitecture*, pp. 413–422 (2009).

[2] Liu, X. and Mellor-Crummey, J.: A tool to Analyze the Performance of Multithreaded Programs on NUMA Architectures, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 259–272 (online), DOI: 10.1145/2555243.2555271 (2014).

[3] Liu, X. and Wu, B.: ScaAnalyzer: a tool to identify memory scalability bottlenecks in parallel programs, *International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 47:1—-47:12 (online), DOI: 10.1145/2807591.2807648 (2015).

[4] Oleary, K.: Finding your memory access performance bottlenecks, Intel (online), available from ⟨https://software.intel.com/en-us/articles/finding-your-memory-access-performance-bottlenecks⟩ (accessed 2017-06-16).

[5] Lachaize, R., Lepers, B. and Quéma, V.: MemProf: A Memory Profiler for NUMA Multicore Systems, *Proceedings of the 2012 USENIX Conference Annual Technical Conference*, p. 5 (2012).

[6] Lepers, B.: Memprof Repository, Memprof (online), available from ⟨https://github.com/Memprof⟩ (accessed 2017-06-16).

[7] Drebes, A., Pop, A., Heydemann, K., Cohen, A. and Drachtemam, N.: Aftermath : A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems, *7th workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2014)*, No. 1, pp. 1–13 (2014).

[8] Pop, A. and Cohen, A.: OpenStream: Expressiveness and Data-Flow Compilation of OpenMP Streaming Programs Antoniu, *IACM Transactions on Architecture and Code Optimization* (2013).

[9] Terpstra, D., Jagode, H., You, H. and Dongarra, J.: Collecting Performance Data with PAPI-C, *Tools for High Performance Computing* (2010).

[10] Pesterev, A., Zeldovich, N. and Morris, R. T.: Locating cache performance bottlenecks using data profiling, *EuroSys '10*, p. 335 (2010).

[11] Pesterev, A.: Locating Cache Performance Bottlenecks Using Data Profiling, PhD Thesis, Massachusetts Institute of Technology (2010).

[12] Pesterev, A., Zeldovich, N. and Morris, R. T.: DProf, MIT (online), available from ⟨https://pdos.csail.mit.edu/archive/dprof/⟩ (accessed 2017-06-23).

[13] Weyers, B., Terboven, C., Schmidl, D., Herber, J., Kuhlen, T. W., Müller, M. S. and Hentshel, B.: Visualization of Memory Access Behavior on Hierarchical NUMA Architectures, *Proceedings of VPA 2014: 1st Workshop on Visual Performance Analysis - held in conjunction with SC 2014*, pp. 42–49 (online), DOI: 10.1109/VPA.2014.12 (2015).

[14] Iwainsky, C., Reichstein, T., Dahnken, C., Mey, D. A., Terboven, C., Semin, A. and Bischof, C.: An approach to visualize remote socket traffic on the intel Nehalem-EX, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 6586 LNCS, pp. 523–530 (2011).

[15] Treibig, J., Hager, G. and Wellein, G.: LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments, *Proceedings of the International Conference on Parallel Processing Workshops*, pp. 207–216 (online), DOI: 10.1109/ICPPW.2010.38 (2010).

[16] Horvath, A.: Memory Bandwidth Benchmark, raas (online), available from ⟨https://github.com/raas/mbw⟩ (accessed 2017-06-20).

[17] Bienia, C., Kumar, S., Singh, J. P. and Li, K.: The PARSEC benchmark suite: characterization and architectural implications, *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ACM, pp. 72–81 (2008).