Data Transfer Requirement Analysis with Bandwidth Curves

Josef Weidendorfer

Technische Universität München
Munich, Germany
weidendo@in.tum.de

Abstract. The amount of data transfer required by (parallel) applications can be of significant importance for performance due to bandwidth limitations in real systems. For intra-node, data transfer refers to memory accesses, and usually, cache miss counts are used for performance analysis. However, they cannot explicitly show utilization of links and buses in the memory hierarchy. In this paper, we propose an explicit visualization of the bandwidth requirements of applications within memory hierarchies. This is based on a machine model without bandwidth limits. We show its usefulness within the context of a simple 2D stencil iterative solver, with and without cache optimization applied.

1 Introduction

A lot of applications are bound not by compute capability, but by limited bandwidth available in a system. It is predicted that for future architectures, the energy required to transfer a byte will go down much slower due to technology advances than the energy required for a calculation [2]. Thus, bandwidth requirements will play an even more important role for the performance achievable by an application. As the distance of a data transfer is significant for cost regarding both energy and time, the exploitation of a memory hierarchy using caches is essential for good performance. Further, with multi-core architectures, contention on shared links can be a major obstacle for performance. For applications with low computational intensity, cache optimization may not be important because it makes the sequential code run faster, but much more so because it reduces contention and allows scaling with the number of used cores.

Detection of whether a given bandwidth restriction is actually the bottleneck is astonishingly tricky. The reason is that achievable bandwidth depends on whether prefetchers are active, whether the accesses are sequential, strided or random. Slowdown due to memory accesses may appear on the connection to the memory module, between chips, or within the cache hierarchy on-chip. Using hardware performance counters to understand these issues is difficult. Miss counts often are almost useless, and prefetcher behavior and bus utilization can not easily be attributed to source.

In this paper, we propose an analysis method for the bandwidth requirements of applications which decouples bandwidth analysis from other effects.

This is done using a machine model whose performance is purely bound by only one resource limit, such as peak instruction throughput or peak computational performance. The model assumes that there are no stall cycles due to data dependencies or other latency, and that any peaks in bandwidth requirements are smoothed out by corresponding hardware or compiler techniques such as instruction reordering or prefetching for best utilization. This way, the analysis is able to show steady bandwidth requirements of hot code paths. As visualization, we propose *bandwidth curves*, which show how much of the runtime a given bandwidth is exceeded by application demands. They make it is easy to see the slowdown induced by a bandwidth limitation, and provide an upper bound for the performance achievable. For our proposed analysis method, we developed a corresponding tool. It uses runtime instrumentation (via Intel's Pin tool [7]) to work with compiled binaries. Execution is simulated within the assumed model, and bandwidth curves are generated. To show the usefulness of our approach, we present a case study in which cache optimization is applied to a memory bound iterative solver. The positive effect of the optimization is easily visible in the curves.

2 Related Work

Most performance analysis tools today use hardware performance counters to show behavior regarding the memory system [1,6]. Typically, hit and miss counts are provided, as well as utilization of the memory bus (percentage of cycles the bus is busy). On newer processors, it is possible to get a statistical distribution of memory access latency (e.g. the data load latency analysis on Intel processors). However, contention detection can be difficult by using such measurements.

Cache optimization tries to increase the locality of memory accesses, as caches help most if memory accesses are "local", either in a spatial or temporal way. Spatial locality can be interpreted as how many bytes of a cache line are used before being evicted [10]. For temporal locality, the stack reuse distance metric is valuable [4]. It shows how large a fully associative cache needs to be to cover a given percentage of memory accesses. A lot of research exists to calculate this metric in a reasonable fast way [3,5,8], and extend it to multi-core [9]. However, stack reuse does not provide any information about bandwidth requirements. The roof-line model [11] is a 2D visualization showing the inherent performance limits of a machine for single kernels, taking the computational intensity into account. While this is very intuitive, it cannot provide contention analysis at various points of the memory hierarchy, and does not apply to complex applications. To the best of the author's knowledge, there are no tools concentrating on bandwidth requirement visualizations similar to the one proposed in this paper.

3 Performance and Machine Model

A real machine has a lot of different resources, such as core pipeline with computational units, the memory system with cache hierarchies, buses, and I/O.

Each of them has its own capacity limits, and for different phases of an application, different resource limitations may be relevant for the observed performance. However, it is often difficult to understand whether a given resource really is the dominant bottleneck, as the difference between theoretical peak performance and real exploitable performance is often influenced by micro-architectural effects.

To overcome these obstacles for performance analysis, we propose an abstract machine model which allows us to focus on application requirements regarding bandwidth demands. We assume that the performance of the application is fully determined just by the restrictions of one resource. For numerical applications, a good choice for the latter is the achievable computational capacity of a machine in GFlops per second. This way, the bandwidth requirements of the application can be expressed in the amount of bytes moved per floating point operation executed, which is known in literature as *computational intensity* of a given program phase. Another choice to determine the base performance of the application may be the assumption that the machine can execute a given number of instructions per clock cycle, and is therefore restricted by the throughput of the instruction decoder. Thus, we assume that the application always runs at some constant peak performance, and we observe the resulting bandwidth requirements. A comparison with real bandwidth limitations allows us to check whether this limitation is a real bottleneck.

Our tool simulates the execution of a binary on a machine model. It observes how much data is moved over specific connections in the system (e.g. from core to L1) in a given time unit, which is determined by the execution of one instruction or one floating point operation. Thus, if T is the number of time units for the complete program, for $t \in \{1, \ldots, T\}$, r_t may be the number of bytes read at time t by a core from its L1 cache. Then $R = \sum_{t=1}^{T} r_t$ is the total amount of bytes read at this point in the system. If we strictly use this model, we will observe a number of bytes moved in some cycles, and nothing in others. However, real systems will smooth out such peak requirements by starting transfers early, or reordering instructions while waiting for data to arrive. Thus, a real system will exploit the available bandwidth capacities, and not stop using the bandwidth only because the application does not execute a move instruction in a given clock cycle. It is important to model this behavior. To this end, we average the bandwidth requirement using a sliding window of a given time length W. The average number of bytes moved at time t is

$$\bar{r}_t = \frac{1}{W} \sum_{w=0}^{W-1} r_{t-w}$$

(with $r_t = 0$ for $t < 0$). A good value for W may be in the order of the latency of a memory access (a few hundred cycles), as such latency is covered by hardware prefetchers. Also with average bandwidths, we easily can calculate the total amount of data transferred at a given point (with $r_t = 0$ for $t > T$):

$$R = \sum_{t=1}^{T+W} \bar{r}_t = \sum_{t=1}^{T} r_t \ .$$

Thus, if we draw $\bar{r}_t$, the area under the curve is the amount of data transferred.

To understand bandwidth requirements of multi-threaded code, it is useful to assume that every thread is bound to run on a dedicated core, and there is no over-subscription. This will show a worst-case scenario regarding contention, which should help the analysis to find bottlenecks. In addition, it makes our model independent from scheduling and migration decisions in the operating system, and thus makes the analysis more general. The model should approximate multi-core systems. Here, we assume a single memory module shared by all cores. For the shared connection to the memory, we just sum up the bandwidth requirements of all cores, within each time unit of our simulation. This strategy may not match reality, as code executed on different cores on real hardware may be slowed down by various effects which depend on the actually executed code. However, for HPC code, it is quite likely that code with similar behavior is executed (e.g. the same loop body when using OpenMP parallel for).

The model works fine with simulating arbitrary caches in the data paths. Caches exploit locality properties of memory access patterns, and thus work as a kind of filter for bandwidth requirement analysis. They potentially reduce the requirements. But caches also play another role in multi-core chips: they allow for fast communication among cores by keeping the data to be communicated on-chip. By simulating a cache coherence protocol (MSI), we capture bandwidth requirements between cores. Fig. 1 shows the configuration of a typical multi-core chip with a separate cache for each core, and a shared cache for all cores. Bandwidth figures are useful for each of the data paths denoted by arrows, as each of them could become a bottleneck. The arrows between the private caches denote communication among cores where the data needs to be fetched from the other core. For this, available bandwidth typically is lower than coming from the shared cache, as coherence actions are involved. Our model captures bandwidth requirements of data paths in one multi-core chip. However, it easily can be extended to match multi-socket systems with a single memory module. Extending the model further for systems with multiple memory modules is straight-forward, but needs the simulation of allocation strategies for memory pages. This is future work. Similarly, extending the analysis to systems with different memory name spaces (such as clusters) is possible.

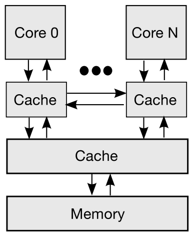

Fig. 1. A multi-core system with connections interesting for bandwidth analysis

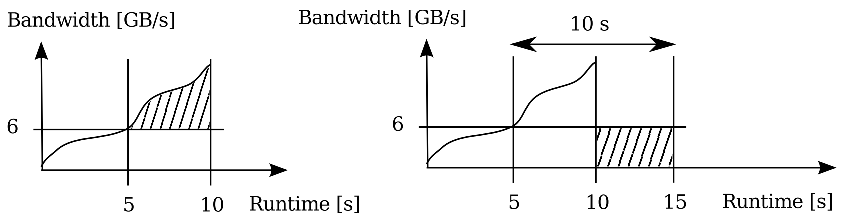

Fig. 2. Slowdown effect of limited bandwidth shown by a Bandwidth Curve

4 Bandwidth Curves

To understand whether a given bandwidth requirement actually constitutes a bottleneck on an approximated real system, a comparison with bandwidth restrictions of the system must be done. Up to now, we assume the bandwidth requirement analysis to be a trace over the program run for every interesting point in some data path, showing the (smoothed) bandwidth requirement for each point in time of the program execution. As the collected amount of data easily gets prohibitive, we need to aggregate data online. Thus, instead of drawing a figure for $\bar{r}_t$ for $t \in \{1, \ldots, T\}$, we reorder the time steps by permutation $o : \{1, \ldots, T\} \to \{1, \ldots, T\}$ such that $\bar{r}_{o(t)} \leq \bar{r}_{o(t+1)}$. The resulting monotonic increasing curve will show how long/often the bandwidth requirement (in bytes per time unit) exceeds at a given limitation.

In the example of Fig. 2 (left), the curve exceeds 6 GB/s during half the runtime of the program corresponding to our machine model. On a real system, the connection cannot transfer at higher bandwidth. The hatched area will result in an observed slowdown. As shown on the right, by filling a rectangle of the same area, we can calculate the slowdown exactly in absolute time. Even more, as half the runtime within our machine model is also limited by the bandwidth, these 5 seconds add up to the absolute time estimation due to the limitation (this is in contrast to the first part, which is only limited by the single resource restriction in the machine model). Thus, in the example, at least 10 seconds are definitely needed due to the bandwidth limitation of 6 GB/s on the real system.

5 Prototype Implementation

Our prototype tool uses Pin [7] on Linux. In contrast to Valgrind (an open-source framework very similar to Pin), it can run multiple guest threads (the program to be observed) in parallel on a multi-core host system[1]. We model a 2-level inclusive cache hierarchy as shown in Fig. 1. Replacement policy is LRU with write-back for L1 and L2. A MSI cache coherence protocol between private L1 caches is implemented, with invalidations done implicitly by incrementing an epoch counter on write for each line in L2. Gnuplot-compatible files for the bandwidth

[1] There is work going on to allow for this feature in future Valgrind versions.

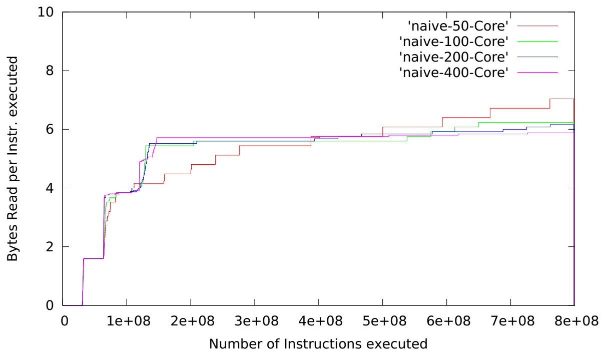

Fig. 3. Influence of smoothing parameter W (50, 100, 200, 400), naive version

curves of each interesting data path are generated at program termination. The current version is not optimized and has a slowdown factor of around 100. We expect that we can improve on that significantly by doing the sliding window smoothing in batches. We will make the tool publicly available as open source.

6 Case Study

For the following examples of the bandwidth curves, we analyzed an iterative Jacobi solver over a quadratic domain $N \times N$ with fixed boundary condition, using the following update rule:

$$v^{t+1}_{x,y} := \frac{1}{4}(v^t_{x-1,y} + v^t_{x,y-1} + v^t_{x+1,y} + v^t_{x,y+1})$$

With $N = 2500$, the alternately used two matrices need 100 MB of memory with every value stored in double precision. The naive version, which sweeps over the matrices for one domain update, needs to load 16 bytes and write 8 bytes per update from memory (2 rows fit into on-chip cache, half of read bytes due to read-for-ownership before write). After some initialization, we run 24 iterations. Intel compiler 13.0 is used with OpenMP enabled.

6.1 Influence of Smoothing Window Length

Fig. 3 shows the effects of changing the parameter W for smoothing the bandwidth requirements. The 24 iterations need around 800 million instructions, and the average number of bytes read is around 6 bytes per instruction (the actual number depends on the code produced by the compiler) most of the time. The lower numbers (around 150 million instr.) come from start-up and initialization. As the solver code in the naive versions consists of a tight inner loop with equal memory access requirements, we would expect just a simple line over the program run. However, for $W = 50$, this is not true. Checking the assembler code,

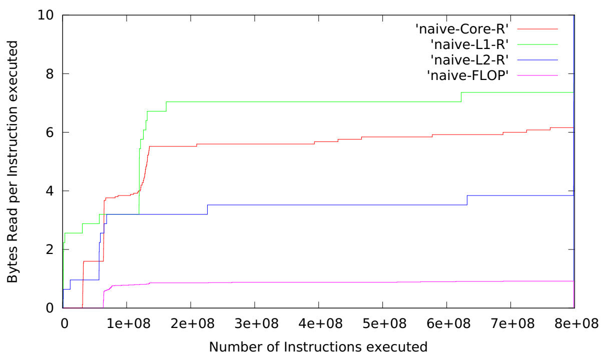

Fig. 4. Bytes read into Core, L1, shared L2, and FLOPs, naive version

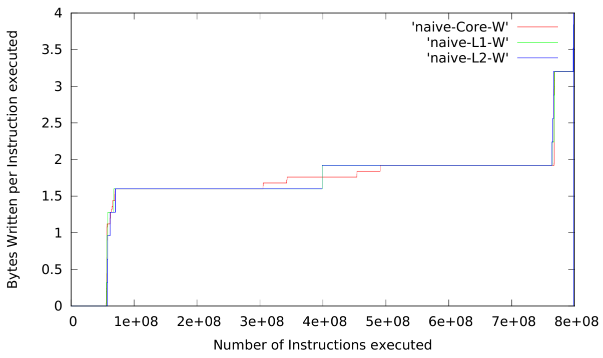

Fig. 5. Bytes written from Core, L1, shared L2, naive version

the compiler did some unrolling, and 50 instructions are not enough to get the steady behavior for the inner loop. However, the curves for $W = 100$, 200 and 400 are almost the same.

6.2 Effects of Caches

In the following, we use $W = 200$ as smoothing window, and the cache hierarchy of Fig. 1. Each core has a private 32 kB L1 data cache, and there is a shared 3 MB L2 cache. As most HPC codes do fit into L1 instruction cache, we ignore loads into the memory hierarchy due to code fetched by the cores.

As in the previous figure, for Fig. 4 we use the number of instructions executed as time unit. For bytes read, the curves show different data paths: bytes read by Core (as in Fig. 3), bytes read from shared L2 into L1, and bytes read from main memory. This figure shows that the bandwidth requirement by L1 may be higher than by the core. With $N = 2500$, the L1 is not large enough to hold even one line of each matrix, and thus each time data has to be reloaded. This in

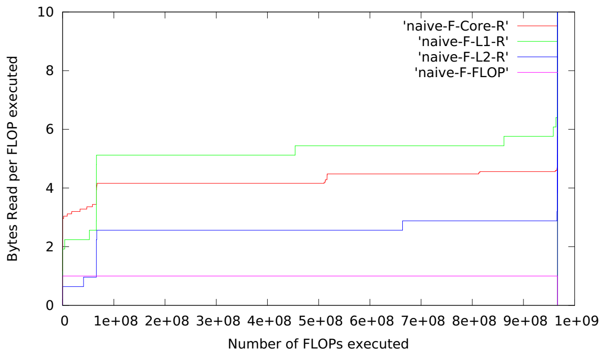

Fig. 6. Bytes read into Core, L1, shared L2, naive version

itself just means the same requirements by the core and L1. However, each write to a newly allocated cache also line triggers a previous read (RFO), explaining the larger figures by L1. We also can measure the number of FLOPs executed per instruction, in the same way as bytes transferred. As can be seen, the code does around 1 FLOP per instruction (this comes from the fact that the compiler mostly produces SSE instructions), and in relation to L2 requirements, there are 4 bytes loaded from memory per FLOP executed. Fig. 5 gives the same for bytes written. For the writes, the caches can not help at all. The average number of bytes written per instruction executed is around 1.8. The peak on the left (around 3.5 bytes) comes from the initialization phase.

6.3 Normalization to FLOPs Executed

Up to now, we used as time unit the instructions executed. Fig. 6 shows the same Fig. 4 (bytes read), but related to FLOPs executed. First, one can see that the FLOP count, normalized to FLOP count, is a straight line at 1, as expected. There are almost one billion FLOPs executed, and there is a strange peak at the very end: as start-up does not do any FLOPs, there is a huge number of bytes read for the first FLOP executed in the program (which is ordered to the left).

6.4 Visualization of Cache Optimization

For cache optimization, the domain is recursively split into blocks fitting into L1. However, within one iteration, there is no reuse of data. Thus, we do multiple iteration updates per block. Because of missing dependencies on block boundaries, the second iteration on a block must not update a boundary of depth 1, the third not a boundary of depth 2 and so on. When two neighboring blocks are done, the missing borders in-between are updated. As block splitting is done recursively by bisection (depending on block dimension either horizontally or vertically), doing the border updates after coming back from the bisection is

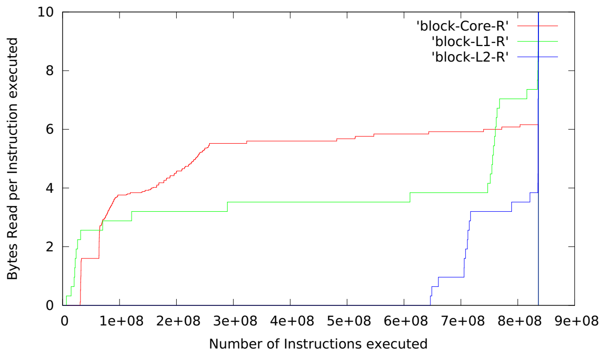

Fig. 7. Bytes read into Core, L1, shared L2, blocking version

straight-forward. The resulting optimization takes the number of iterations to block as variable parameter.

We show the results corresponding to the naive version (Fig. 4) in Fig. 7. We use an iteration blocking of 12. With blocking, the number of instruction executed is increased to around 840 million. The curve showing reads from the core did not change much though, in contrast to the figures for bytes read by L1 and by L2. L1 bandwidth requirements is much reduced. Typically for blocking, only for a small part of the time there is the original requirement. Similar for L2, however, most of the time, there is nothing read from memory at all. This shows the effectiveness of the optimization.

6.5 Multi-threaded Code

The OpenMP version of the 2D iterative solver splits the matrices into slices of equal size. As each thread executes the same code, the curves are the same for the privately used data paths, and are scaled by the number of threads for the shared connection to memory.

Table 1 shows user runtime numbers (threads summed up) for the example code measured on a quad-core Ivy Bridge (2.7 GHz), and the time predicted by our model to be spent at least due to a 15 GB/s read limit from memory. As expected, in contrast to the blocked version, the memory limitation plays an important role for the naive version, and becomes more significant in relation to measured runtime for a larger number of threads.

Table 1. Runtimes of iterative solver and estimations due to memory limitation

[s]	runtimes				due to limitation			
threads	1	2	3	4	1	2	3	4
naive	0.23	0.38	0.57	0.76	0.13	0.28	0.43	0.59
blocked	0.13	0.16	0.20	0.26	0.02	0.04	0.06	0.08

7 Conclusion

In this paper, we propose a way to analyze the data transfer requirements of applications independent from other effects. For an example code, the resulting visualizations are quite insightful. The assumed peak resource restriction in the machine model may be too optimistic for good estimations in the absence of bandwidth bottlenecks. But even in this case, our method at least assures that memory bandwidth is not an issue.

An analysis tool must provide a relation to source lines/data structures. We plan to use a sampling approach. At random points in time, we store instruction pointer and data structures referenced for all accesses in the smoothing window. Thus, each sample gives us a combination of accesses which make up a given bandwidth. Further, bandwidth requirement analysis can be combined with spatial locality analysis to show how much of the transferred data is actually used by cores. The combination will tell whether data layout changes are useful to achieve better bandwidth behavior.

Acknowledgment. Thanks to C. Trinitis for fruitful discussions and proofreading, and to the anonymous reviewers whose critics resulted in a much better paper.

References

1. Intel VTune Amplifier, `http://software.intel.com/en-us/intel-vtune-amplifier-xe`
2. International Technology Roadmap for Seminconductors (2012), `http://www.itrs.net/Links/2012ITRS/2012Chapters/2012Overview.pdf`
3. Almási, G., Caşcaval, C., Padua, D.A.: Calculating stack distances efficiently. In: Proceedings of the 2002 Workshop on Memory System Performance, MSP 2002, pp. 37–43. ACM, New York (2002)
4. Bennett, B.T., Kruskal, V.J.: LRU stack processing. IBM Journal of Research and Development 19, 353–357 (1975)
5. Berg, E., Hagersten, E.: Statcache: A probabilistic approach to efficient and accurate data locality analysis. In: Proc. of the Int. Symposium on Performance Analysis of Systems and Software (2004)
6. de Melo, A.C.: Performance Counters on Linux. Presentation at the Linux Plumbers Conference (September 2009)
7. Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proc. of PLDI 2005. ACM, New York (2005)
8. Niu, Q., Dinan, J., Lu, Q., Sadayappan, P.: PARDA: A fast parallel reuse distance analysis algorithm. In: Int. Parallel and Distributed Processing Symposium (2012)
9. Schuff, D.L., Kulkarni, M., Pai, V.S.: Accelerating multicore reuse distance analysis with sampling and parallelization. In: Proc. of the 19th Int. Conf. on Parallel Architectures and Compilation Techniques. ACM, New York (2010)
10. Weidendorfer, J., Trinitis, C.: Collecting and exploiting cache-reuse metrics. In: Sunderam, V.S., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2005. LNCS, vol. 3515, pp. 191–198. Springer, Heidelberg (2005)
11. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Comm. ACM 52(4) (April 2009)