Using Bandwidth Throttling to Quantify Application Sensitivity to Heterogeneous Memory

Clément Foyer Brice Goglin
Inria, LaBRI, Univ. Bordeaux
Talence, France
{clement.foyer,brice.goglin}@inria.fr

***Abstract*—In the dawn of the exascale era, the memory management is getting increasingly harder but also of primary importance. The plurality of processing systems along with the emergence of heterogeneous memory systems require more care to be put into data placement. Yet, in order to test models, designs and heuristics for data placement, the programmer has to be able to access these expensive systems, or find a way to emulate them.**

In this paper we propose to use the *Resource Control* features of the Linux kernel and x86 processors to add heterogeneity to a homogeneous memory system in order to evaluate the impact of different bandwidths on application performance. We define a new metric to evaluate the sensibility to bandwidth throttling as a way to investigate the benefits of using high-bandwidth memory (HBM) for any given application, without the need to access a platform offering this kind of memory. We evaluated 6 different well-known benchmarks with different sensitivity to bandwidth on a AMD platform, and validated our results on two Intel platforms with heterogeneous memory, Xeon Phi and Xeon with NVDIMMs. Although representing an idealized version of HBM, our method gives reliable insight of potential gains when using HBM.

Finally, we envision a design based on *Resource Control* using both bandwidth restriction and cache partitioning to simulate a more complex heterogeneous environment that allows for hand-picked data placement on emulated heterogeneous memory. We believe our approach can help develop new tools to test reliably new algorithms that improve data placement for heterogeneous memory systems.

***Index Terms*—heterogeneous memory, bandwidth throttling, data placement, application characterization**

I. Introduction

The memory wall [1] is a well known problem that arises once again with the emergence of exascale supercomputers. With more data to be processed by Big Data applications, and microarchitectures that rely on a constant flow of data to be used at full capacity, the need for efficient memory systems is growing stronger. A perfect memory system would provide a low latency, a high bandwidth, and would retain data in case of power outage, all while being cost-effective. However, there is no such thing as a perfect solution, and computer architects are composing with multiple different technologies, each providing specific characteristics (e.g. low latency, high bandwidth, non volatility, etc.). The heterogeneity of the memory systems adds an extra layer of complexity to the already known issues related to non-uniform memory accesses (NUMA), as memory targets may expose intrinsically different performance and traits.

Therefore, application characterization (i.e. bandwidth-bound or latency-bound for example), data selection and careful placement is essential for achieving optimal performance. For example, a bandwidth-bound application may benefit from a high-bandwidth memory (HBM) that could sustain the input flow required to use large vector registers. In such a case, the application developer needs to be able to determine whether poor memory access is indeed degrading the performance, and which data are critical to relocate into HBM.

Moreover, because of the relatively high-cost of the different memory systems (comparatively to the DRAM), not all facilities can provide high-end systems with different kinds of heterogeneous memories. It is then important to be able to evaluate in advance the relevance of each memory type, in order to use the resources sparingly. The solution we are presenting in this paper offers a new route for evaluating application's sensitivity to bandwidth. We use the *Resource Control* feature of x86 processors allowing a fine grain control over bandwidth. This bandwidth throttling first allows to simulate platforms with different memory characteristics which can then be used to evaluate the sensitivity of any given application. We speculate that the benefit of using bandwidth optimized memory applications can be inferred from the sensitivity to bandwidth throttling. To evaluate the sensitivity of an application, we define a metric based on how the figure of merits (FoM) of the given application varies with throttling. We tested our metric on a range of well known benchmarks and proxy-applications: a pointer-chasing application, XSBench [2], NASA Parallel Benchmark (NPB) BT [3], Lulesh [4], miniFE [5] and STREAM [6].

The remainder of this article is organized as follows. In Section II we present the pre-existing work on bandwidth affecting techniques and the Linux kernel module `resctrl` on which we based our work. Section III presents the preliminaries tests we did to be able to evaluate the impact of our solution on applications while Section IV introduces and evaluates our approach. Finally, conclusions and future work are presented in Section V.

II. Background

A. Allocating Data Buffers in Heterogeneous Memory

The advent of the Intel Xeon Phi processors several years ago raised the issue of deciding whether a data buffer should

be allocated in high-bandwidth low-capacity (16 GB of MCDRAM) or in the normal memory (hundreds of GBs). Indeed, such heterogeneous memory systems expose different access performance depending on where the target buffer is allocated. New programming interfaces were proposed to explicitly allocate buffers in different kinds of memory [7], or to expose memory characteristics to runtimes and applications through explicit performance metrics [8].

However these allocations first require to identify which data buffers are latency- or bandwidth-sensitive. Post-mortem analysis, for instance using hardware counters, is a way to detect which buffer accesses slowed down the application and may provide hints for better allocation in the next runs [9], [10]. Another approach consists in observing the application performance degradation when the memory performance changes. This has been proposed by having a *Bandwidth Bandit* process [11] saturate the memory subsystem while an application is running [12]. However this strategy does not allow a fine tuning or understanding of the contention caused by the pirate and its impact on bandwidth and latency for different application access pattern(s). In this work, we rather propose to use a hardware-based approach that precisely throttles the actual available memory bandwidth, without restricting neither to parts of the resources (cores reserved for the bandit's threads or processes) nor requiring any code modifications.

B. Bandwidth Throttling with Linux Resource Control

The *Resource Control* subsystem (`resctrl` [13]) was added in Linux kernel 4.10 to expose *Resource Director Technology* features of Intel processors. Since then, it has been ported to the similar *Platform Quality of Service* features of AMD processors. Provided its support is enabled by the operating system, it allows — among other things — partitioning the cache between applications, or throttling the available memory bandwidth.

The administrator may define policies in the `sysfs resctrl` virtual filesystem. Applications are affected either because they were specified by their process identifier, or because they run on specific cores. Cache partitioning then consists in allowing or disallowing slices of a given cache through a bit field where each bit represents a slice of the cache. Memory bandwidth throttling is also defined per cache: a policy defines which memory bandwidth is available to tasks when their local L3 cache accesses the main memory. The available bandwidth is specified either in percentage or in $\mathrm{MB/s}$ (depending on the underlying CPU driver).

This feature provides us with an easy way to observe how applications' performance varies when the available memory bandwidth changes without switching between hardware platforms or having a bandwidth pirate possibly occupy some cores, impact the latency, overheat the CPU, etc.

III. Bandwidth throttling on x86 architectures

In order to evaluate our solution, we first tested two platforms on which we would use our approach. The first objective was to measure the impact bandwidth throttling has on latency, if any, and verify that no unexpected NUMA effect would appear depending on data location. The two platforms used for applying our metric were x86 architectures, one Intel based (described in Section III-A), and one AMD based (described in Section III-B). The baseline for the bandwidth was set with the STREAM benchmark. The baseline for the latency was measured with a simple, home brewed pointer-chasing application, where each entry were separated by a $128\,\mathrm{bit}$ wide stride, and the destinations were randomly chosen by an initial shuffling of the pointers destinations. In both cases, the application was bound to the package cores and its memory was interleaved between the requested NUMA nodes to avoid contention (using `hwloc-bind`).

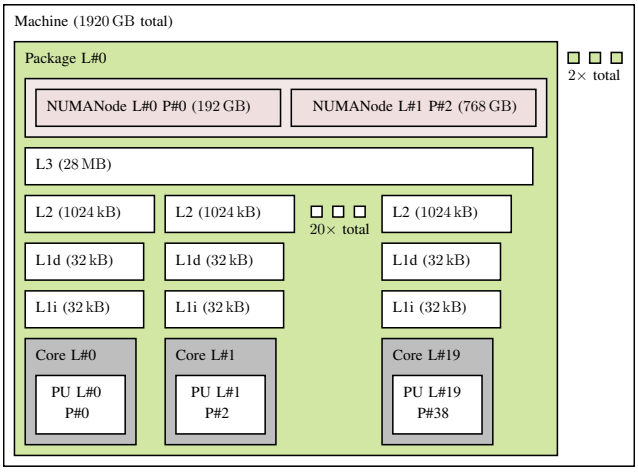


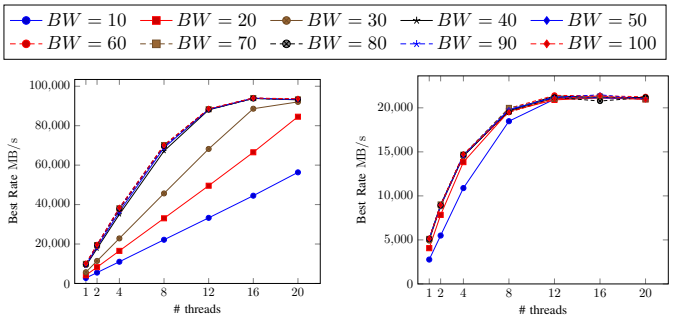
Fig. 1: Topology of the dual Intel *Cascade Lake* Xeon Gold 6230 platform as reported by `lstopo` (factorized version).

A. Intel

The first platform used dual-processor 20-core Intel *Cascade Lake* Xeon Gold 6230 ($2.1\,\mathrm{GHz}$), providing up-to 20 threads per package (hyperthreading disabled). This platform (Fig. 1) had $192\,\mathrm{GB}$ of DRAM per package, along with $768\,\mathrm{GB}$ of NVRAM (Intel Optane NVDIMM in *AppDirect system-ram* mode, exposed as additional NUMA nodes). On this processor, `resctrl` defines the throttling in percentage of the total bandwidth, with a step of $10\,\%$.

The results shown in Fig. 2 only display for the *Triad* operation from the STREAM benchmark, as the behaviour was similar with the other operations. We only report the results of a single application of the STREAM benchmark as the reproducibility and stability of the results was verified. The NUMA numbers that are indicated are the logical index as returned by hwloc.

The figures displayed are characteristic for each of the NUMA targets. We observe a gradual increase of maximum bandwidth achieved as we add threads, for local DRAM, cross-NUMA DRAM and local NVDIMM. The maximum bandwidth ($\approx 94000\,\mathrm{MB/s}$) is usually obtained for 16 to 20 threads. Bandwidth throttling seems to have very little effect for any value higher than 30. Although characteristic, we



(a) Bandwidth for *Triad* operation to cores in package 0 from DRAM (NUMA#0)

(b) Bandwidth for *Triad* operation to cores in package 0 from NVDIMM (NUMA#1)

(c) Bandwidth for *Triad* operation to cores in package 0 from cross-NUMA DRAM (NUMA#2)

(d) Bandwidth for *Triad* operation to cores in package 0 from cross-NUMA NVDIMM (NUMA#3)

Fig. 2: Bandwidth as reported by the STREAM benchmark, depending on the number of threads and `resctrl` setting, on Intel Xeon.

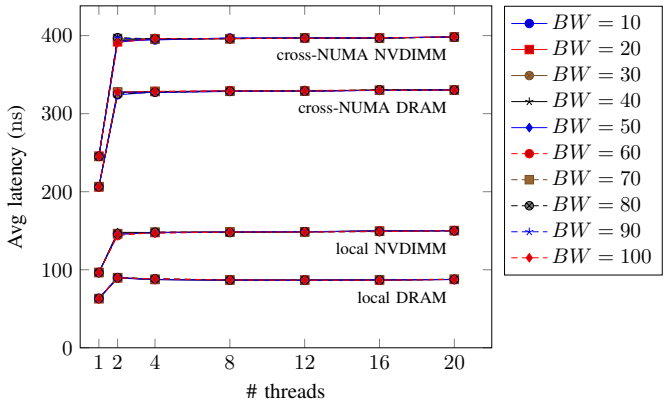


Fig. 3: Latency depending on the number of threads and `resctrl` setting on Intel Xeon.

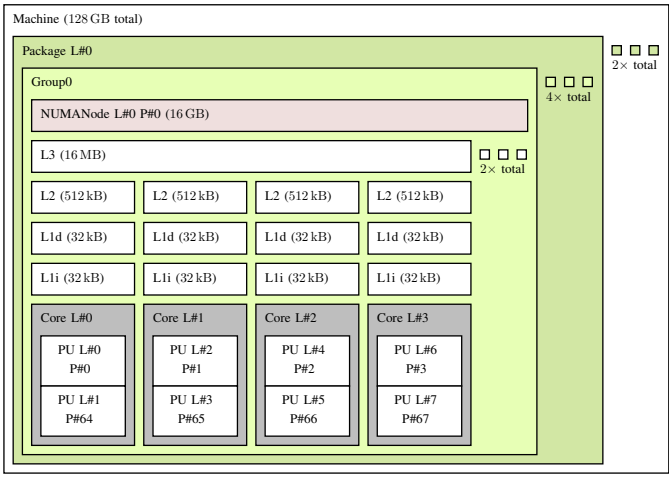


Fig. 4: Topology of the dual AMD *Zen2 Rome* EPYC 7502 platform, as reported by `lstopo` (factorized version).

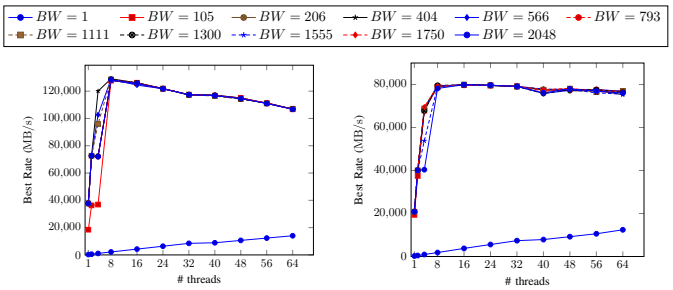
observe some differences when the data is located in the cross-NUMA NVDIMM. The measures show a drop in performance when more than 8 threads are used. *Copy* and *Scale* are showing a very steep performance loss when using 12 threads ($-70\,\%$) and losing $20\,\%$ of the remaining when using 16 threads. Comparatively, *Add* and *Triad* performance are more gradually decreasing between 8 and 16 threads, as shown in Fig. 2d.

As for the latency in the Intel architecture, we observe a strong NUMA effect, associated with a strong influence of the number of concurrent threads. The bandwidth throttling however does not seem to influence the latency otherwise (Fig. 3). This difference with AMD (see Section III-B) might come from the step of the bandwidth throttling setting that does not allow to reduce it enough to see any effect on latency.

The simple memory structure on the Intel platform eases the memory binding and makes it a prime platform for testing heterogeneous memory systems. However, the underwhelming effect of bandwidth throttling and the small range of settings make it less suitable for evaluating the impact of progressive bandwidth throttling on applications' performance.

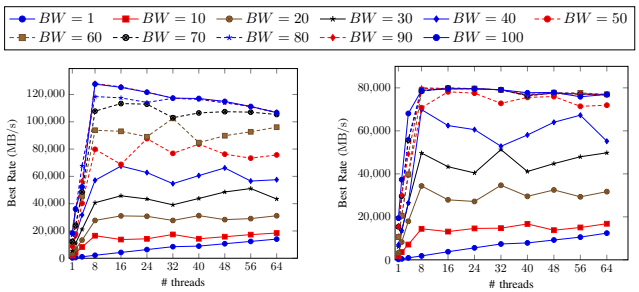
B. AMD

The second platform is a dual-processor AMD 32-core *Zen2 Rome* EPYC 7502 (2.5 GHz), providing up-to 64 threads per package, two threads per core, sharing L1 and L2 caches, eight threads sharing L3 caches, and sixteen threads sharing the NUMA domain of 16 GB of DRAM (Fig. 4). On this processor, `resctrl` defines the throttling in the range 1 to 2048, with a step of 1. It is unclear whether the value sets the absolute value in MB/s or whether it corresponds to an arbitrary scale. As our measurements presented in Fig. 5 show, modifying the allocated bandwidth has little to no effect on the measured bandwidth in the range 100 to 2048. It hence suggests the use of an arbitrary scale. We therefore mainly focused our experiments in the 1–100 range.



(a) Bandwidth for *Triad* operation to cores in package 0 from DRAM (package local)

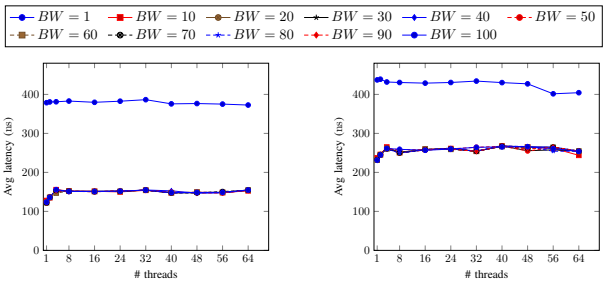
(b) Bandwidth for *Triad* operation to cores in package 0 from DRAM (cross package)



(c) Bandwidth for *Triad* operation to cores in package 0 from DRAM (package local)

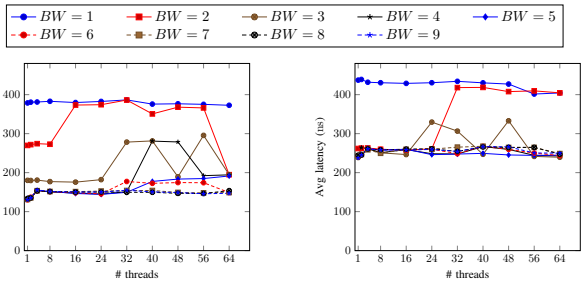
(d) Bandwidth for *Triad* operation to cores in package 0 from DRAM (cross package)

Fig. 5: Bandwidth as reported by the STREAM benchmark, depending on the number of threads and `resctrl` setting on AMD EPYC.



(a) Cores in package 0 from DRAM (package local)

(b) Cores in package 0 from DRAM (cross package)



(c) Cores in package 0 from DRAM (package local)

(d) Cores in package 0 from DRAM (cross package)

Fig. 6: Latency depending on the number of threads and `resctrl` setting on AMD EPYC.

The measures were done by taking the median value of 10 executions of the STREAM benchmark and a single execution of the pointer-chasing benchmark as a baseline for future experiments. The STREAM FoM being based on the best bandwidth achieved, the NUMA groups within the package are ignored, considering at least one of the threads within one of the groups would be accessing data locally to its NUMA node, hence achieving the highest possible bandwidth. Our measures (Fig. 5) show the maximum bandwidth seems to be achieved for 8 threads with one thread per L3 cache, at $128\,000\,\mathrm{MB/s}$, and after that the peak performance is decreasing regularly while threads are being added on new cores. Accessing data outside one's package creates a penalty of $33\,\%$ (Fig. 5b), but the increase in the number of threads does not affect the peak performance as much. Overall, the bandwidth throttling seems to have very little impact over the bandwidth for any value higher than 105 when using more than 8 threads, or for values higher than 206 otherwise. On the other hand, between 1 and 80, each step has a significant impact on performance (Fig. 5c), while the range 80–100 only impacts the performance when less than 32 threads are used. Finally, although the $33\,\%$ penalty seems conserved for accesses outside the package (Fig. 5d), a bandwidth setting of 50 at most seems to be required to observe any effect.

In terms of latency (Fig. 6), our measures are showing no impact of the `resctrl` for any value above $\approx$ 6–7) for package local accesses. Also, the hyperthreading does not show any effect on latency.

The AMD processors provides a large range of settings when it comes to bandwidth throttling, with fine grain tuning, and a visible effect even for small restrictions, contrary to the Intel platform. Therefore, despite the more complex memory structure, we decided to prefer this platform for our metric evaluation, as it would provide a wider spectrum of settings on which to compare our applications. Finally, as we planned on using values ≥ 10 for bandwidth allocation, we can be sure not to affect the latency during our experiments.

IV. Experiments and Results

The experiments followed a two step process. First, we ran the selected applications on the AMD platform with a progressive bandwidth throttling, as explained in Section IV-B. It allowed us to evaluated the possible benefits of running those applications on a platform with HBM. Then, we validated our results on a KNL platform and on Xeon processors using DRAM and NVDIMMs (details in Section IV-C).

A. Application test set

We chose a set of applications among the most frequent found in the literature, keeping a variety of programming languages. Excepted when specified otherwise, the applications were using the OpenMP multi-threading model [14], and the optimization flags were `-O3 -march=knl` on KNL, and `-O3 -march=core-avx2` otherwise. The applications were compiled with `icc` for C applications, `icpc` for C++

applications, and `ifort` for FORTRAN applications, all from the Intel oneAPI toolkit version 2021.2.0.

a) Pointer-chasing: This application was developed internally, and is a simple multithreaded pointer chasing application, which jumps from one pointer to the next at each step. The application is available online[1], and can be parametrized in different ways. For our tests we did 100 time measurements of $1\,000\,000$ jumps, through $8\,\mathrm{GB}$ of data, where every two pointers were at least $128\,\mathrm{B}$ apart, to ensure two pointers would not be found in the same cache line. The pointers were shuffled in order to create a cycle of size $64\,\mathrm{M\,elements}$. The multi-threading support was done using POSIX pthread. The figure of merit considered was the number of pointer resolutions per second.

We expect to see this application as the less sensitive to bandwidth throttling as the range chosen for the bandwidth restriction did not show any effect on latency. Moreover, each pointer reference only requires a single cache line to be read from memory, therefore it should require much less bandwidth than the maximum allowed from the resource control.

b) XSBench: This application is a mini-app representing a key computational kernel of the Monte Carlo neutron transport algorithm [2]. We used a custom version[2] which diverges marginally from the original version to allow for batch execution, support for improved error management and support of user provided parameters. The simulation model used was *history* and the problem size chosen was *large* (11 303 grid points per nuclide). For other parameters, the default values were used. We considered the number of lookups per second as the figure of merits for this application.

c) NASA Parallel Benchmark BT: The block Tri-Diagonal solver is a pseudo-application specified in NPB 1 that mimics the computation and data movement in CFD (Computational Fluid Dynamics) applications [3]. This is part of a set of programs and kernels that were designed to help evaluate the performance of parallel supercomputers. We used the version 3.4.2[3], and the C size class. The figure of merit of this application was the number of million operations per second.

d) Lulesh: LULESH[4] is a highly simplified application, developed at Lawrence Livermore National Laboratories that represents the numerical algorithms, data motion, and programming style typical in scientific C or C++ based applications [4]. We used the version 2.0 over 50 iterations on cube meshes of size 200. The figure of merit we considered for this application was the number of elements solved per micro-second.

e) miniFE: Finite element mini-application[5] is a proxy application for unstructured implicit finite element codes, similar to HPCCG and pHPCCG but providing a much more complete vertical covering of the steps in this class of applications [5]. We used the optimized OpenMP version on AMD and Intel Xeon, and the KNL OpenMP optimized version on KNL. We used a $256 \times 256 \times 256$ problem size for all the experiments. The figure of merit of this application was the total MFLOPS of the Conjugate Gradient.

f) STREAM: The last application we considered was the STREAM benchmark [6][6]. This application is a well known benchmark that measures sustainable memory bandwidth (in $\mathrm{MB/s}$) and the corresponding computation rate for simple vector kernels. The compilation options had 20 iterations to be run for arrays containing $80\,000\,000$ elements. We used the maximum achieved bandwidth (in $\mathrm{MB/s}$) as the figure of merit. This benchmark is expected to be the most sensitive as it aims at measuring the maximum achievable bandwidth by design.

B. Metric

For this study, we defined a new metric to evaluate the sensitivity to bandwidth for an application. We speculate that it could be used as a hint to identify what performance bottlenecks are to be expected for an unknown application, and what potential benefits can be expected from using a different system. Moreover, one advantage of the method designed is to not require *in-situ* testing as the measures are taken with homogeneous hardware memory environment. The only requirement for our approach to be applied, is to rely on an increasing figure of merit, where higher is better, and using processor (and kernels) supporting the `resctrl` feature.

We define our metric as a percentage of the maximum figure of merit (FoM) achieved while the bandwidth varies. Then, we compute the $90\,\%$, $75\,\%$ and $50\,\%$ of that maximum. We make the assumption that bandwidth-sensitive applications will show a decrease in performance when increasing the throttling using `resctrl`. Therefore, the 90/75/50 thresholds for bandwidth-sensitive applications will be passed with lesser bandwidth restriction than for bandwidth-insensitive applications. On the other hand, latency sensitive applications would show little to no influence of the throttling, under the circumstance of bandwidth throttling not affecting the latency, as described in Section III. We chose to keep the three values ($90\,\%$, $75\,\%$ and $50\,\%$) as it provides hints on how steep the decrease in performance is.

Following the metric definition, we computed for each application the average FoM achieved over 10 runs and took the maximum of all averages as the baseline. The chosen FoM for each application corresponds to the numerical quantity reported to evaluate the efficiency of one given execution (e.g. maximum achieved bandwidth for STREAM, MFLOPS for BT, etc.). Using this value, we evaluated the corresponding 90/75/50 thresholds (Fig. 7) and recorded each time the throttling value where the median falls below each specified threshold (see values reported in Table I).

The metric was measured twice. First, with 8 threads (1 thread per L3) in order to evaluate the impact of bandwidth

[1]Available at https://gitlab.inria.fr/cfoyer/diana

[2]Commit `f799af9` of https://github.com/clementFoyer/XSBench

[3]Available at https://www.nas.nasa.gov/assets/npb/NPB3.4.2.tar.gz

[4]Available at https://github.com/LLNL/LULESH/tree/46c2a1d6db

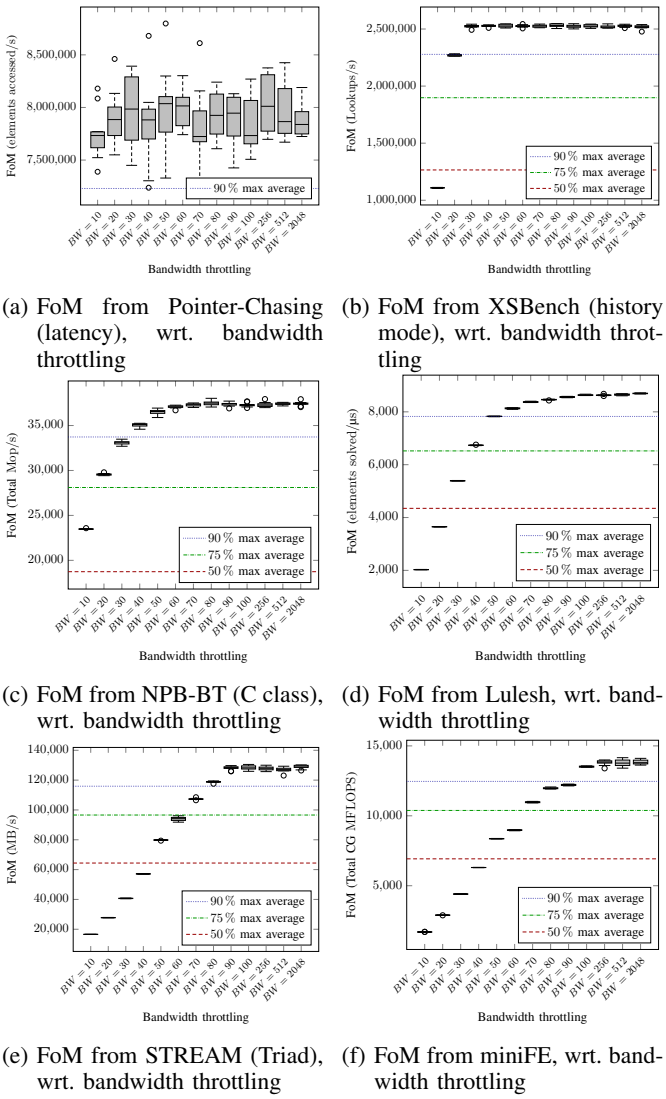
[5]Available at https://github.com/Mantevo/miniFE

[6]Available at https://github.com/jeffhammond/STREAM

TABLE I: Bandwidth sensitivity metric results

Application	Max Avg FoM	Bandwidth sensitivity 50 %	75 %	90 %	Fig.
Pointer-chasing	8 033 367	0	0	0	7a
XSBench	2 530 884	10	10	20	7b
XSBench.16[1]	4 742 125	10	10	20	
NPB-BT	37 475	0	10	30	7c
Lulesh	8695	20	30	40	7d
XSBench.24[1]	6 523 765	20	30	40	
NPB-BT.16[1]	67 731	10	20	50	
NPB-BT.24[1]	87 574	10	30	50	
NPB-BT.32[1]	103 534	20	30	50	8b
STREAM (Copy)	120 319	20	40	50	
STREAM (Scale)	117 375	20	40	50	
XSBench.32[1]	7 225 668	30	40	60	8a
STREAM (Triad)	128 791	40	60	70	7e
STREAM (Add)	131 993	40	60	80	
miniFE	13 845	40	60	90	7f

[1] *bench-name.N* corresponds to the benchmark run with N threads, as shown in Fig. 7. Other benchmarks have been run with 8 threads.



(a) FoM from Pointer-Chasing (latency), wrt. bandwidth throttling

(b) FoM from XSBench (history mode), wrt. bandwidth throttling

(c) FoM from NPB-BT (C class), wrt. bandwidth throttling

(d) FoM from Lulesh, wrt. bandwidth throttling

(e) FoM from STREAM (Triad), wrt. bandwidth throttling

(f) FoM from miniFE, wrt. bandwidth throttling

Fig. 7: Bandwidth sensitivity metric on AMD EPYC, 8 threads, 4 NUMA nodes, interleaved.

throttling without the risk of congestion on either the main memory or the L3 caches, especially with high values in `resctrl`. Moreover, from Fig. 5c, it seems that an optimal number of threads to expose the bandwidth sensitivity would be 8 threads, as it reached the highest results with the best discrimination between the different level of bandwidth throttling. The results are being shown in Fig. 7. In order to reflect closer to real conditions, we measured a second time with 32 threads (all cores used, no hyperthreading, as in Fig. 8) in order to compare with the 8-threads case and check whether the relative sensitivity are consistent. In the second case, only the important results are being showed in the figure. From the metric results, we can sort the different applications by their respective bandwidth sensitivity. The order is shown in Table I.

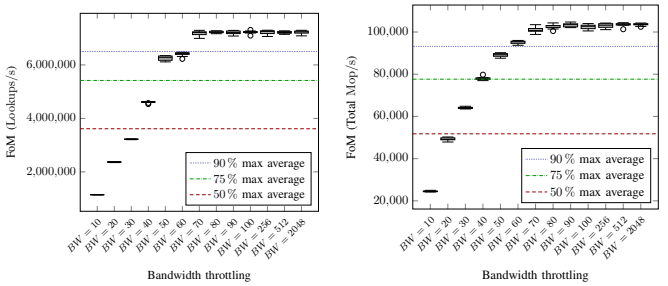
One interesting point to be noted is that results from our pointer-chasing application confirm that our approach has no effect on latency which was one of the objective. One unexpected result is however coming from the STREAM benchmark as it seems to be less sensitive to bandwidth throttling than miniFE while we expected STREAM to be the most sensitive one.

The additional testing was then done to evaluate our metric on 32 threads, one thread per core (no hyperthreading). The results were similar to the version with 8 threads, in terms of relative performance and ordering, excepted for XSBench which behaves more closely to the BT benchmark. Testing with different number of threads showed that the overall application sensitivity may depend on that number. We identified two possible reasons that would explain this behaviour. First, fewer threads usually means more cache space is available for each thread, therefore they rely less on the performance between the last level cache and the main memory. Second, the more threads are being used, the lesser impact `resctrl` has on the overall performance as the bandwidth throttling parameters are applied independently to each thread and not globally to the whole application. It appeared that the overall results are stable when using 32 threads, with the exception of XSBench which becomes increasingly sensitive to bandwidth when increasing the number of concurrent threads. Additional testing showed that adding threads for XSBench diminishes the number of L3 accesses while maintaining the cache miss rate (less than 2 points variation between the two cases at $\approx 76\,\%$). In the mean time, BT's cache miss rate increases greatly when adding threads, from $66\,\%$ to $80\,\%$. We attribute this increase in miss rate to the concurrency between the threads that leads to more cache evictions.

In order to validate the results obtained with out metric, we ran some test using a system offering heterogeneous memory, as presented in the next section.

C. Applying the metric to the selected applications

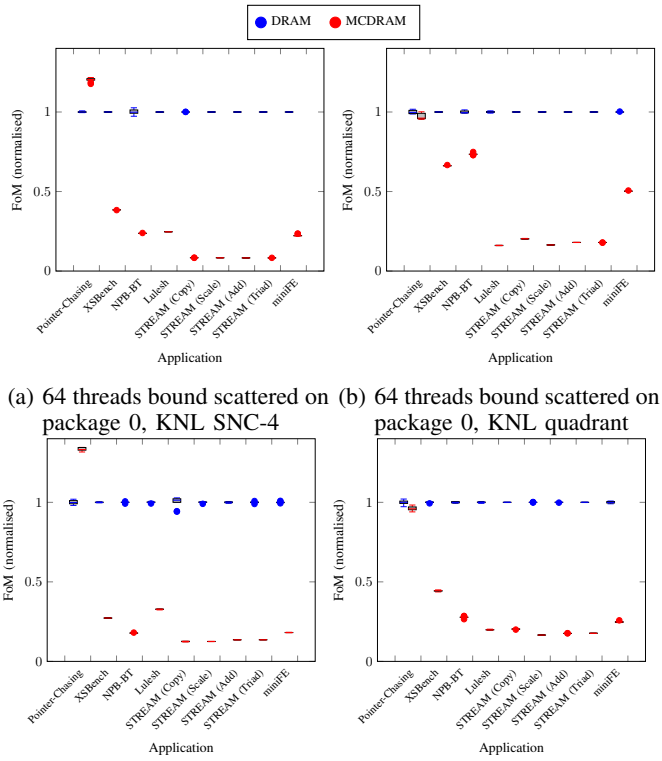
We ran the experiments on a KNL platform made of a 64-core Intel Xeon Phi 7230 processor (1.3 GHz) with



(a) FoM from XSBench, wrt. bandwidth throttling

(b) FoM from NPB-BT (C class), wrt. bandwidth throttling

Fig. 8: Bandwidth sensitivity metric on AMD EPYC, 32 threads, 4 NUMA nodes, interleaved.



(a) 64 threads bound scattered on package 0, KNL SNC-4

(b) 64 threads bound scattered on package 0, KNL quadrant

(c) 256 threads bound to package 0, KNL SNC-4

(d) 256 threads bound to package 0, KNL quadrant

Fig. 9: FoMs on Intel Xeon Phi (KNL) in flat memory configuration, wrt. data placement.

4 threads per core and memory configured in *Flat* mode. We used two clustering configurations for our experiments: SNC-4 (4 GB of MCDRAM per sub-NUMA cluster, Figs. 9a and 9c) and Quadrant (16 GB of MCDRAM, Figs. 9b and 9d). Applications were run in two configurations, once with 64 threads, one thread per core, and once with 256 threads. Each application was run 10 times in each configuration, in order to ensure statistical significance. We evaluated the performance gain using the average figure of merit over the 10 runs with the memory bound to MCDRAM, and normalized it using the average figure of merit over the 10 runs with the memory bound to DRAM.

Comparing applications on the left of Figs. 9a and 9b shows that the SNC-4 mode impacts strongly the latency. The way memory was bound to MCDRAM with interleaving may be responsible for this effect. On the other hand, with the quadrant mode, data can benefit from a better locality, managed by the processor memory unit.

Overall, our preliminary results are respected and the relative gains are superior for the application we deemed more sensitive to bandwidth. Lulesh seems to suffer from the too numerous threads and from the latency penalty in the SNC-4 mode but this effect disappears in quadrant mode, which strengthens our analysis. We observe that contrary to our metric prediction, STREAM benefits the most from the MCDRAM, although miniFE is also close to it when run with 256 threads. We suggest that miniFE application may be on the fence between memory-bound and compute-bound, therefore having a lower bandwidth scalability when changing platform for the KNL. XSBench and BT always showed less gain then Lulesh, STREAM and miniFE, excepted in with 256 threads in SNC-4 mode, where Lulesh suffers the most from the penalty expressed previously Our pointer-chasing application never benefits from the high-bandwidth memory, and even suffer penalty from it in SNC-4, just like the Lulesh application.

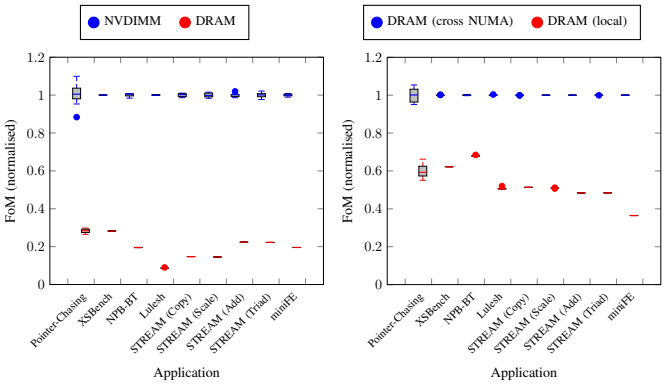
In addition to running the application on KNL, we also ran on the Intel Xeon platform, in order to compare the effect of binding memory on DRAM or on (slower) NVDIMM, and the NUMA effect. The applications were run 10 times with 20 threads and with the memory either bound to their local DRAM, their local NVDIMM (Intel Optane DCPMM) or the cross-package DRAM. The results are shown in Fig. 10a and Fig. 10b, respectively. The results are normalized on the ”slowest” memory in each case, i.e. NVDIMM in the first case, and cross-package DRAM in the second.

Once again, the results are generally conclusive with respect to our metric. However, in the NVDIMM case, we observe a influence of the improved latency that benefits Lulesh notably, participating to its performance gain.

V. Conclusions and Future Work

As heterogeneous memory becomes widespread in HPC platforms, there is a need for tools to simulate various kinds of platforms and various kinds of memories. Determining the sensitivity of applications to memory bandwidth is required to allocate data buffers in the appropriate memory target. In this paper we have presented a metric to evaluate this sensitivity by altering the available memory bandwidth thanks to the *Resource Control* subsystem of the Linux kernel and x86 processors. This metric allows us to estimate whether allocating application buffers in HBM could be beneficial.

Our metric relies on a progressive throttling of the bandwidth allocated to a thread and its impact on the overall performance. We evaluated our metric on a sample of well-known benchmarks from the industry and validated our results



(a) DRAM and NVDIMM, 20 threads bound to package 0, Intel Xeon

(b) DRAM NUMA, 20 threads bound to package 0, Intel Xeon

Fig. 10: FoMs on Intel Xeon wrt. data placement.

on two platform providing heterogeneous memory. Although our metric requires attention in its analysis due to culprits affecting the output, its use only relies on already existing systems which is an economical way of gaining insight on an unknown application, for example.

Our work currently only looks at an entire application being placed in a single memory tier. The objective was to generate more heterogeneity on our memory accesses in order to simulate a system with HBM and DRAM, for example. Determining the sensitivity of individual buffers requires to throttle accesses to these buffers while keeping other accesses unthrottled. However, the throttling being at the L3 cache level, we could not easily apply it based on the memory target, resulting in a simulated environment where either all or no data would be located on HBM. We experimented using dynamic thread binding relying on `resctrl` bit masks to apply or not the throttling to the application. Although it provides a dynamic way to control the bandwidth allocated to one process during execution, it does not solve the ”all or nothing” issue.

One proposed solution could be to also rely on the L3 cache splitting feature from `resctrl` and on the manual control of when and what data are being loaded to the cache. That way, it would be possible to manually preload some data in L3 cache without suffering from the throttling, then to apply the throttling before accessing the rest of the data without the risk of eviction for the preloaded data. This procedure could provide a way to simulate, for example, a two tiered memory system with different bandwidths, but the protocol aforementioned is purely theoretical and has not been further studied yet.

Applying this proposed protocol could be a way to emulate heterogeneous memory systems with buffer selection in order to provide a framework for data placement heuristics testing and validation.

We are also looking at supporting additional platforms. ARM support for `resctrl` is under preparation for the Linux kernel and this will certainly be an interesting target because several ARM platforms with heterogeneous memory are expected in the next years.[7]

Acknowledgments

This work was supported in part by the French National Research Agency (ANR) in the frame of the ANR-DFG H2M project (ANR-20-CE92-0022-01). Some experiments presented in this paper were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LaBRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d’Aquitaine (see https://www.plafrim.fr/).

References

[1] W. A. Wulf and S. A. McKee, “Hitting the memory wall: Implications of the obvious,” *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995. [Online]. Available: https://doi.org/10.1145/216585.216588

[2] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, “XSBench — the development and verification of a performance abstraction for Monte Carlo reactor analysis,” in *PHYSOR 2014 — The Role of Reactor Physics toward a Sustainable Future*, Kyoto, 2014. [Online]. Available: https://www.mcs.anl.gov/papers/P5064-0114.pdf

[3] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon, “NAS parallel benchmark results,” NASA, Tech. Rep., 1992.

[4] I. Karlin, “Lulesh programming model and performance ports overview,” Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2012.

[5] P. S. Crozier, H. K. Thornquist, R. W. Numrich, A. B. Williams, H. C. Edwards, E. R. Keiter, M. Rajan, J. M. Willenbring, D. W. Doerfler, and M. A. Heroux, “Improving performance via mini-applications.” Sandia National Laboratories (SNL), Albuquerque, NM, and Livermore, CA (United States), Tech. Rep., Sep. 2009. [Online]. Available: https://www.osti.gov/biblio/993908

[6] J. D. McCalpin, “STREAM: Sustainable memory bandwidth in high performance computers,” University of Virginia, Charlottesville, Virginia, Tech. Rep., 1991–2007, a continually updated technical report. http://www.cs.virginia.edu/stream/. [Online]. Available: http://www.cs.virginia.edu/stream/

[7] C. Cantalupo, V. Venkatesan, J. R. Hammond, and S. Hammond, “User Extensible Heap Manager for Heterogeneous Memory Platforms and Mixed Memory Policies,” 2015. [Online]. Available: http://memkind.github.io/memkind/memkind_arch_20150318.pdf

[8] E. A. León, B. Goglin, and A. R. Proano, “M&MMs: Navigating Complex Memory Spaces with hwloc,” in *The Fifth International Symposium on Memory Systems Proceedings (MEMSYS19)*. Washington, DC: ACM, Oct. 2019, pp. 149–155. [Online]. Available: http://hal.inria.fr/hal-02266285

[9] H. Servat, A. Pena, G. Llort, E. Mercadal, H.-C. Hoppe, and J. Labarta, “Automating the Application Data Placement in Hybrid Memory Systems,” in *Proceedings of the IEEE International Conference on Cluster Computing*, Hawaii, USA, Sep. 2017.

[10] A. Narayan, T. Zhang, S. Aga, S. Narayanasamy, and A. K. Coskun, “MOCA: Memory Object Classification and Allocation in Heterogeneous Memory Systems,” in *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium*. Vancouver, BC, Canada: IEEE, May 2018.

[11] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten, “Bandwidth bandit: Quantitative characterization of memory contention,” in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013, pp. 1–10.

[12] H. Brunie, J. Jaeger, P. Carribault, and D. Barthou, “Profile-guided scope-based data allocation method,” in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS ’18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 169—182. [Online]. Available: https://doi.org/10.1145/3240302.3240313

[13] T. kernel development community, “Linux kernel.” [Online]. Available: https://www.kernel.org/doc/html/latest/index.html

[14] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[7] ETRI’s K-AB21 processor and SiPearl’s Rhea exascale processor have already been announced with both HBM and DDR5.