

Modeling memory bandwidth patterns on NUMA machines with performance counters

Daniel Goodman

Oracle Labs

daniel.goodman@oracle.com

Roni Haecki*

Oracle Labs

Tim Harris†

Oracle Labs

ABSTRACT

Computers used for data analytics are often NUMA systems with multiple sockets per machine, multiple cores per socket, and multiple thread contexts per core. To get the peak performance out of these machines requires the correct number of threads to be placed in the correct positions on the machine. One particularly interesting element of the placement of memory and threads is the way it effects the movement of data around the machine, and the increased latency this can introduce to reads and writes. In this paper we describe work on modeling the bandwidth requirements of an application on a NUMA compute node based on the placement of threads. The model is parameterized by sampling performance counters during 2 application runs with carefully chosen thread placements. Evaluating the model with thousands of measurements shows a median difference from predictions of 2.34% of the bandwidth. The results of this modeling can be used in a number of ways varying from: Performance debugging during development where the programmer can be alerted to potentially problematic memory access patterns; To systems such as Pandia which take an application and predict the performance and system load of a proposed thread count and placement; To libraries of data structures such as Parallel Collections and Smart Arrays that can abstract from the user memory placement and thread placement issues when parallelizing code.

1 INTRODUCTION

Modern NUMA computers with multiple sockets per machine, multiple cores per socket, and multiple thread contexts per core are often used for data analytics type workloads. To get the peak performance out of these machines requires the correct number of threads to be placed in the correct positions on the machine. One particularly interesting element of the placement of memory and threads is the way it effects the movement of data around the machine. This effects the latency of reads and writes both through the inherent different latencies of different connections, but also through the different bandwidths of the connections and the effect

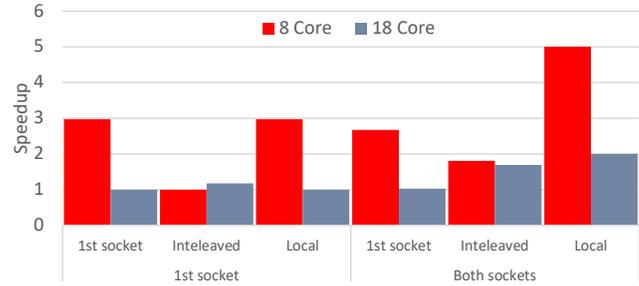


Figure 1: Graph showing the performance of a memory intensive application on different dual socket Intel Xeon machines with different thread and memory placements. Speedup is relative to the slowest configuration for each machine. Results are labeled by the memory placement, 1st socket, interleaved, or local, and then by the thread placement, 1 socket, or both sockets.

if these become saturated. This paper describes work on modeling the bandwidth requirements of an application on a NUMA compute node based on the placement of threads. The model is parameterized by sampling performance counters during 2 application runs with carefully chosen thread placements. Previous work using performance counters has tended to focus on the IPC and LLC miss counters, but as the number of performance counters available on modern processors increases [26] the opportunity to construct more intricate models also increases. Evaluating the model over many thousands of measurements has shown a median difference between the predicted and the measured bandwidth of 2.34% of the bandwidth, with larger errors primarily restricted to applications that transfer little data. The results of the modeling can be used in a number of ways varying from: Performance debugging during development where the programmer can be alerted to potentially problematic memory access patterns; To systems such as Pandia [10] which take an application and predict the performance and system load of a proposed thread count and placement; To libraries of data structures such as Parallel Collections [21] and Smart

*Work carried out while at Oracle Labs

†Work carried out while at Oracle Labs.

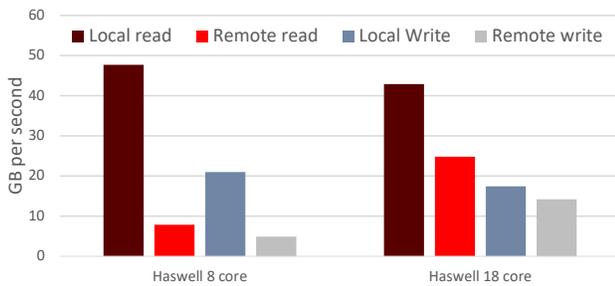


Figure 2: Graph of the different memory bandwidths available on the test systems.

Arrays [22] that can abstract from the user memory placement and thread placement issues when parallelizing code.

As motivation for this work consider the graph in Figure 1. This shows the performance of different thread and memory placements on 2 different 2 socket Intel Haswell machines. Each run is normalized relative to the slowest run on a given machine. The same memory intensive synthetic benchmark is used for all runs and was run with enough threads to place a single thread on each core of a single socket, 8 and 18 threads respectively. When running with 2 sockets these threads are split evenly with each thread still placed in its own core. The memory placements are: 1) all memory on the first socket; 2) Memory interleaved between sockets at the granularity of a page giving 50% remote accesses; 3) All of a threads memory located locally to the thread giving 0% remote accesses. From this we can see that when using a single socket for the 18 core system there is little difference between accessing data remotely and accessing it locally with the CPU acting as the limiting factor, while for the 8 core system there is a 3x slowdown relative to the quickest placement. We can also observe that when running an application with shared memory, the fastest placement for the 18 core machine is to spread the threads and the data evenly across the machine interleaving the memory to spread the bandwidth load evenly. For the 8 core machine peak performance is achieved by keeping all the data and threads on a single socket avoiding remote communication. It is clear from this that the 18 core machine is far more forgiving of thread and memory placement, however as shown in Figure 2 the 8 core machine has a higher bandwidth to the local memory, and is a substantially cheaper machine with a suggested retail price per CPU of \$667 vs \$4115. This means that if the placement of memory and threads can be correctly organized there is the potential to save both time and money on memory limited applications. Figure 2 also demonstrates the dramatic difference in bandwidth for 2 apparently similar machines.

Having demonstrated the effects that different memory placements can have on a system we will now consider how this model could be used in the use cases mentioned earlier.

Performance prediction Systems that seek to model the performance of a workload in a given configuration need to be able to predict the resource demands. Currently in systems such as Pandia they either use a static placement for all applications, or measure the distribution of bandwidth during one of the measurement runs and assume that it will stay the same throughout all further thread placements. The use of this model would allow for more detailed bandwidth requirements to be added so allowing for a more accurate prediction of the performance with each possible thread count and placement.

Libraries of data structures. When libraries such as Smart Arrays abstract the memory placement from the user, the library needs to make decisions about how best to layout the memory. Currently they make the assumption that when the collection requires bandwidth for data processing the application will not be using bandwidth for anything else. However the assumption that all the data used by the application at the time will be in collections belonging to the library is a restrictive one. The use of a bandwidth model would provide the opportunity to model the bandwidth requirements and make allowances for this at run time when placing the data stored by the library into memory.

Debugging and development. Applications will typically run in many environments and performance critical applications have to be tested in each environment that they wish to run in. By modeling the bandwidth requirements of the application with different thread placements and against different hardware descriptions, it would be possible to flag up potential problems to the programmer before the application reaches the testing stage, so allowing an earlier fix.

Our contributions are the bandwidth model, the techniques for parameterizing it to fit real programs, and the evaluation of the model.

The rest of this paper is structured as follows: We will first introduce an outline of the system that we are modeling; Then we will look at the way we model the bandwidth; Having introduced the model we will look at how it can be used to predict the bandwidth requirements of a thread placement; Before we consider how we measure the values for the model for a given application; We then look at the stability and accuracy of this modeling across different hardware with a range of synthetic and real benchmarks; Before concluding with thoughts on future work and looking at related work.

2 TARGET HARDWARE

This work is aimed at multi-socket NUMA machines with performance counters to record their activity. It was carried out on Intel Xeon machines. An idealized example of a 2 socket machine can be seen in Figure 3 with a memory bank attached to each socket via memory channels, and the sockets connected via an interconnect. Information about the performance of applications is captured via performance counters the processors provide through tools such as Performance Counter Monitor (PCM). While some variation appears between manufacturers, the counters offered by manufacturers are sufficiently similar that we believe that the techniques described here can be applied to other hardware with minimal modifications.

2.1 Performance counters

The counters of interest in this work cover the time elapsed, the number of instructions executed, and the volume of data read or written to each memory bank separated out into data to and from the local socket and data to and from remote sockets. While the performance counters recording the volume of data to and from the memory banks are almost certainly located on the memory controller contained within the CPU, in Figure 3 we have drawn them on the memory banks to emphasize that on the Intel processors used for this work the counters report from the perspective of the memory bank, not the processor regarding local and remote accesses. So for example if we have 2 threads on CPU1 and 1 thread on CPU 2 all running at the same speed, and all sending $\frac{1}{2}$ of their accesses to each of the memory banks, then from the point of view of the CPUs $\frac{1}{2}$ of memory accesses are to remote locations, and $\frac{1}{2}$ are to local locations. However from the point of view of memory banks 1 and 2, $\frac{2}{3}$ and $\frac{1}{3}$ of the accesses are local respectively with the remainder being remote accesses, and it is this view that is reported by the program counters.

2.1.1 lessons learned. When selecting the performance counters we noted two sets of counters that we initially thought would be useful but it turned out were insufficient for our use case:

Quick Path Interconnect (QPI). The first of these was the QPI counters. We had hoped that these would allow us to observe directly how much of our data was moving along which interconnects. However, there is a substantial amount of additional traffic that makes use of the QPI, much of which appears to die away when the QPI is required for moving application data. So this traffic is not a limiting factor on the performance of an application, but does make for a very noisy signal when trying to model the bandwidth. For this

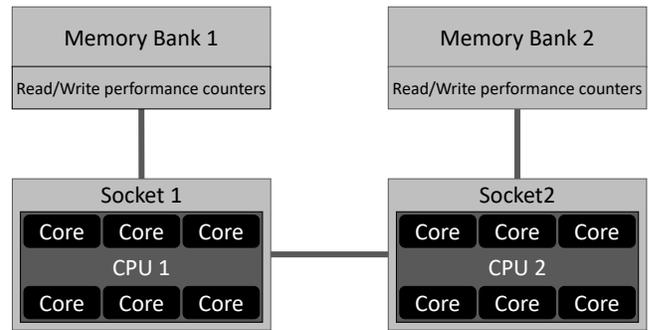


Figure 3: Example of a 2 socket NUMA machine with banks of memory attached to each socket, and each socket containing a CPU with 6 cores. We will use the same machine in all our examples, but for visual clarity we will dispense with drawing the cores. Each memory bank is connected to its respective socket by a memory channel, and an interconnect joins the sockets. While the performance counters recording the data transfer are almost certainly located on the memory controller contained within the CPU, we have drawn them on the memory banks here to emphasize that on the Intel processors used for this work the counters report from the perspective of the memory bank, not the processor regarding local and remote accesses.

reason we instead chose to look at the traffic to the memory banks which is considerably less noisy as a metric.

Instructions Per Cycle (IPC). Many performance counter libraries still include IPC as one of the performance counters that users can request, and while the value returned is the IPC, as the frequency of the chips can change at a very fine granularity this information is extremely misleading without a corresponding record of the chip frequency. To overcome this we instead opt to record the number of instructions executed and the time in which they were executed.

3 BANDWIDTH MODEL

Our model of the bandwidth utilization across a machine is constructed by splitting the usage into 4 classes of memory access pattern that can be combined to describe the memory access pattern of the application threads. The fraction of each threads memory accesses that can be attributed to each of these patterns is measured through 2 profiling runs and we will call this a bandwidth signature. The resulting signature can then be used to apply bandwidth requirements to any thread placement. Separate signatures are constructed for reads and writes, but the measurements required for these two signatures are taken during a single set of runs. The

access types are described here on a system with s sockets and executing a workload using n threads:

Static Access to memory that is allocated on the RAM attached to a single socket but used by all threads. For example if the master thread loads the input data or the output array is allocated on a single socket.

Local Access to memory that is only accessed by the threads on the same socket as the memory. For example a replicated data structures, or thread local data.

Interleaved Access to memory that is allocated evenly across the used sockets such that each socket has $\frac{1}{s}$ of the data. For example if the interleave flag has been used in numactl.

Per-thread Access to memory where each thread allocates $\frac{1}{n}$ of the memory locally, but the memory is used by all threads. For example if each thread loads $\frac{1}{n}$ of the data, or the threads are constructing and using a shared data structure such as a tree.

Examples of these patterns can be seen in Figure 4.

We view that the read bandwidth and the write bandwidth requirements for each workloads is made up of a mix of these 4 classes of access pattern. To encode this for both reads and writes we use 3 properties in the range $[0 \dots 1]$ describing the fraction of the accesses that is Per thread, Local, and Static. We call these the *Per thread fraction*, *Local fraction*, and *Static fraction*. Any remaining bandwidth is deemed to be Interleaved. The sum of the three fractions must be ≤ 1 . This gives 6 properties in total, 3 for reads and 3 for writes. These 6 are augmented by a property for read bandwidth and a property for write bandwidth that records which socket the static bandwidth is associated with. We call this the *Static socket*. Together these properties make up the bandwidth signature of the application.

Before considering how to calculate the 4 properties that make up the model for reads and the 4 properties for writes we now consider how to apply them to construct the bandwidth requirements for different thread placements.

4 APPLYING BANDWIDTH SIGNATURE TO A THREAD PLACEMENT

As discussed the model describes how the bandwidth from a thread on a given socket should be distributed across the system, the total volume of data for each thread will need to be calculated independently. For example in Pandia this is achieved by taking the bandwidth requirement of a single thread, applying this to every thread and then scaling the bandwidths on a thread by thread basis to allow for changes in thread performance due to issues such as resource saturation.

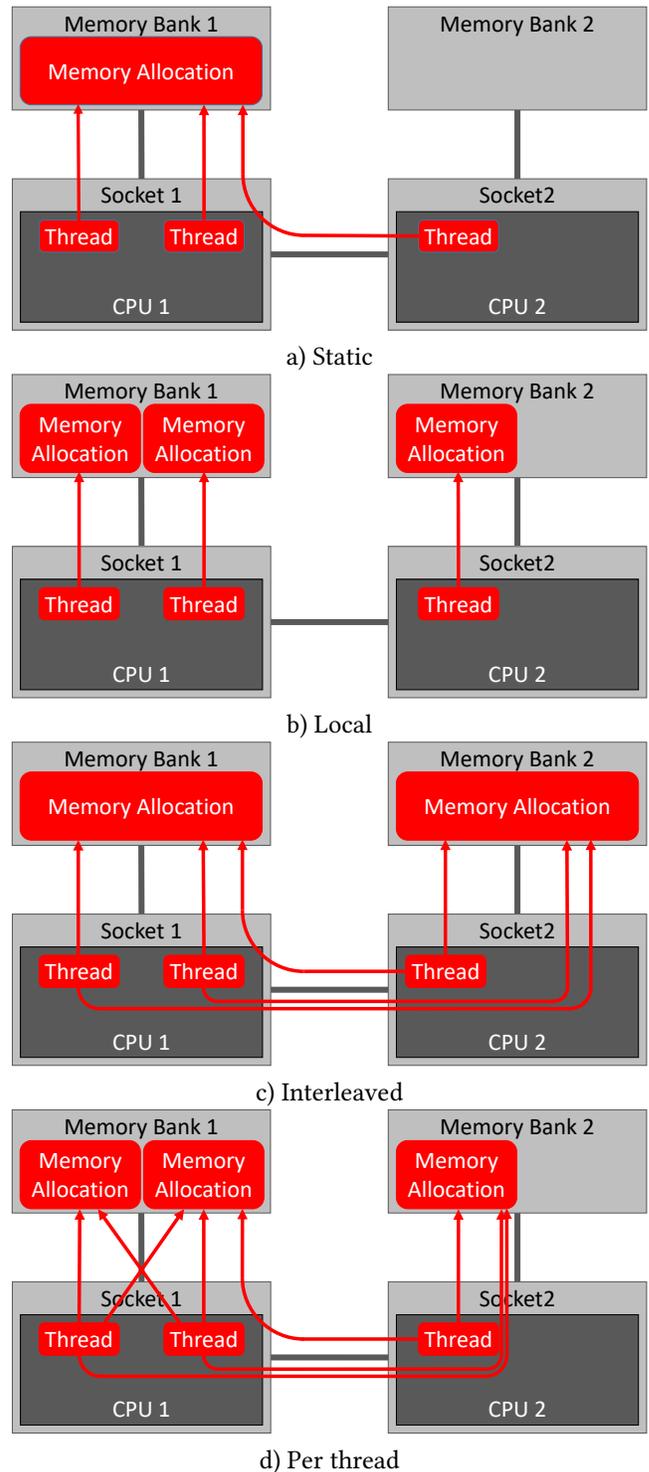


Figure 4: Examples of the 4 different types of memory placement types that we model. In this case they are an application with 3 threads placed on a machine with 2 sockets.

To demonstrate the applying the signature we will use an example which has the properties Static socket = 2, Static fraction = 0.2, Local fraction = 0.35, and Per thread fraction = 0.3 with $1 - (0.2 + 0.35 + 0.3)$ giving 0.15 as the value of the fraction of the bandwidth that is due to interleaved placements. We will consider a placement of 4 threads on a two socket machine with 3 threads placed on the first socket and 1 thread on the second socket.

One way to think about this is as a matrix computation where we have a matrix for each type of memory traffic. Within these matrices each row represents data traveling to or from a CPU, and each column represents data traveling to or from a memory bank. Each cell represent the fraction of the data traveling to or from a given CPU and a given memory bank. As a result the sum of each row will be 1. The four matrices that are then constructed as follows:

$$\begin{array}{|c|c|} \hline \text{Static} \\ \hline \text{BANK} \\ \hline \text{CPU} \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 0 & 1 \\ \hline \end{array} \\ \hline \end{array} \cdot 0.2 + \begin{array}{|c|c|} \hline \text{Local} \\ \hline \text{BANK} \\ \hline \text{CPU} \begin{array}{|c|c|} \hline 1 & 0 \\ \hline 0 & 1 \\ \hline \end{array} \\ \hline \end{array} \cdot 0.35 + \begin{array}{|c|c|c|} \hline \text{Per Thread} \\ \hline \text{BANK} \\ \hline \text{CPU} \begin{array}{|c|c|} \hline \frac{3}{4} & \frac{1}{4} \\ \hline \frac{3}{4} & \frac{1}{4} \\ \hline \end{array} \\ \hline \end{array} \cdot 0.3 + \begin{array}{|c|c|} \hline \text{Interleaved} \\ \hline \text{BANK} \\ \hline \text{CPU} \begin{array}{|c|c|} \hline \frac{1}{2} & \frac{1}{2} \\ \hline \frac{1}{2} & \frac{1}{2} \\ \hline \end{array} \\ \hline \end{array} \cdot 0.15 = \begin{array}{|c|c|} \hline \text{Combined} \\ \hline \text{BANK} \\ \hline \text{CPU} \begin{array}{|c|c|} \hline \frac{13}{20} & \frac{7}{20} \\ \hline \frac{6}{20} & \frac{14}{20} \\ \hline \end{array} \\ \hline \end{array}$$

Figure 5: Diagram demonstrating for our example how calculation of the fractions of bandwidth to different memory banks can be treated as a matrix computation. This example is for a 2 socket machine with 3 threads on the first socket and 1 thread on the second. Note that every row sums to 1, but not every column.

Static The matrix for static memory accesses represents all traffic going to a single memory bank and so consists of the column identified by the static socket property containing 1's and all other columns containing zeros.

Local The matrix for local memory accesses models all data accesses from a socket going to their corresponding memory bank. This is represented by an identity matrix.

Per Thread The matrix for per thread data consists of a series of columns weighted by the fraction of the threads that are on each socket, so the weights for column i can be calculated by $\frac{n_i}{\sum_{j=1}^s n_j}$ where n_i is the number of threads on socket i .

Interleaved The matrix for interleaved data models all accesses being spread evenly across the system. There for cells where both the memory bank and the CPU are from used sockets contain $\frac{1}{s}$ and the other cells contain 0 where s is the number of sockets in use in the placement.

These matrices can the be scaled by their respective fractions, and summed. This results in a single matrix mapping a threads socket to the fraction of its bandwidth it is predicted place on each link to the memory banks. Figure 5 shows this process for the worked example.

5 MEASURING AN APPLICATIONS BANDWIDTH SIGNATURE

Having looked at how the model can be used we will now consider how to measure the parameters required for the model. As our aim is to model the bandwidth utilization of the various elements of the machine it is not sufficient to just observe where the memory is allocated as a relatively small piece of memory may be accessed frequently while a larger piece of memory may be seldom accessed. It is also

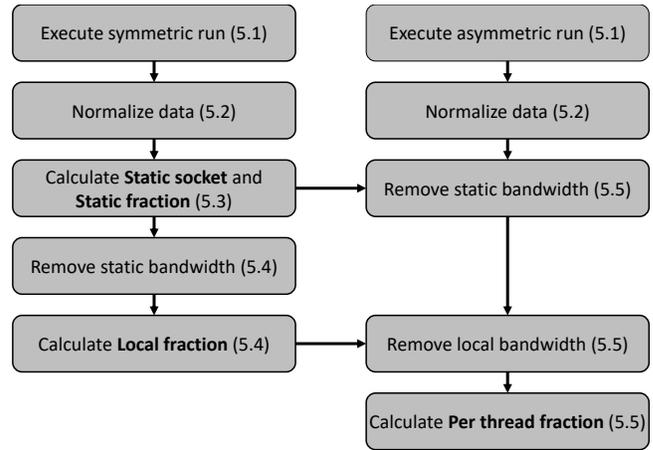


Figure 6: Flow diagram of the steps involved in constructing a bandwidth signature.

not sufficient to just measure the bandwidth requirements of a single thread in a single location and assume that the pattern of accesses it displays will not change as the threads position changes.

The techniques to calculate bandwidth signature can be applied to differing numbers of sockets, but for clarity here we will describe how to calculate the properties using just 2 sockets. A flow diagram of the steps involved in calculating the signature can be seen in Figure 6.

5.1 Profiling runs

To calculate the signature we take advantage of program counters that for the memory attached to each socket report the volume of data sent to and from the local CPU and the volume of data sent to and from the remote CPUs. Using these program counters we collect data from 2 runs of the workload.

The first of benchmarking runs, is a job with an even number of threads where every thread has its own core, and both sockets have the same thread count. In this placement some cores are left unused to leave space to allow the asymmetric

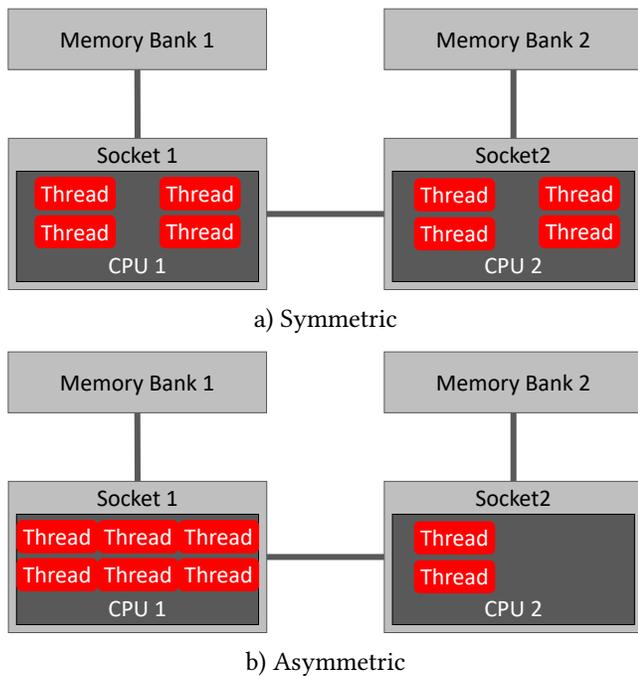


Figure 7: Examples of the two placements we use to determine the types of memory access of an application. These are on a 2 socket machine with 6 cores per socket.

placement to use the same number of threads while maintaining the 1 thread per core policy. The second run uses the same thread count, but has a different number of threads on each socket. An example of these placements can be seen in Figure 7. With our current choice of performance counters, 2 runs is the minimum number from which sufficient data points can be measured to calculate all 8 properties of the signature. While the values for the profiling could be calculated with many different placements the choice to use a symmetric placement for the first run greatly simplifies the process. For example the interleaved and per thread access patterns become identical when the number of threads on each socket is equal, and the thread local accesses place an equal load on each memory bank. Symmetric runs where the symmetry in the loads is not present can be used to detect applications that don't fit the model well. This is discussed in more depth in the evaluation.

To keep profiling times to a minimum it is interesting to note that it is only necessary to execute the workload until a stable state is achieved and the program counters can be read, not until completion. In addition if adding this work into existing performance prediction tools such as Pandia the symmetric run already appears in the existing runs used by the tool, so the asymmetric run is the only additional run.

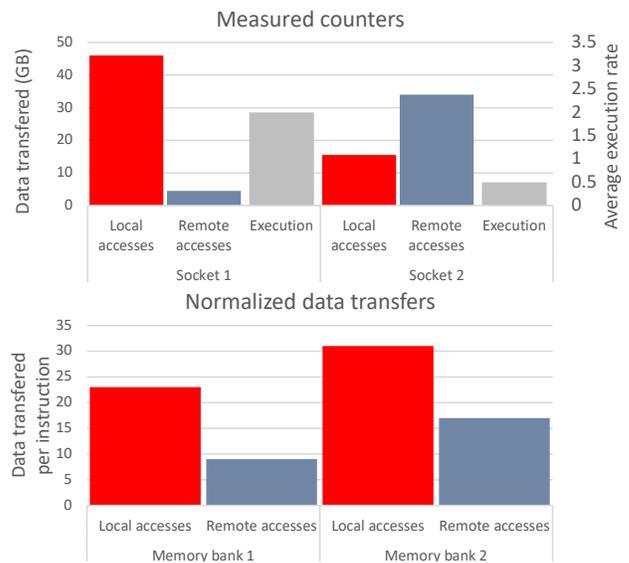


Figure 8: Graphs showing an example of the data recorded by the performance counters and the change to this data after normalization. Execution is measured from the perspective of the CPU while memory accesses are from the perspective of the memory bank.

5.2 Data normalization

The first step is to normalize the data rate per socket relative to the rate of instructions being executed on the socket. This is required because even on relatively simple workloads there can be a significant variation in execution rate of threads on different sockets. This can be caused by a number of issues including, different latency times on memory accesses, and different available bandwidths to memory banks. By way of example on some lower spec processors the QPI interlink between sockets can be saturated by a single thread, yet to run both the symmetric and asymmetric placements using the same thread count we need at least 2 threads per socket for the symmetric placement.

These differing rates of execution can make the raw counter output unrepresentative of the per thread memory access patterns. For example if we are running the symmetric placement and the threads are performing $\frac{3}{4}$ their accesses locally and $\frac{1}{4}$ their accesses remotely. If all the threads are running at the same speed then both memory banks will report that $\frac{3}{4}$ their accesses are local and $\frac{1}{4}$ are remote, and they each have returned the same amount of data. However, if the threads on the second socket are running at half the speed of the threads on the first socket then the ratios change so that $\frac{6}{7}$ of the accesses to bank 1 and $\frac{6}{10}$ of the accesses to bank 2 are local. Whats more bank 1 only gets $\frac{7}{8}$ and bank 2 gets $\frac{5}{8}$ of the traffic they would receive if the threads ran at full speed.

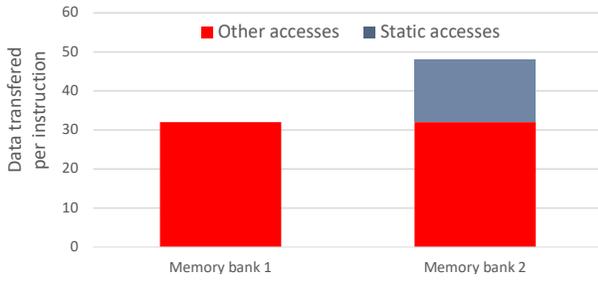


Figure 9: Total bandwidth per socket, with the data associated with accesses to static allocations highlighted.

To overcome this we first normalize the data transfer counters to the rate that the threads are executing at a per socket granularity. The resulting output is the data sent or received per average instruction execution rate per memory bank from each socket. An example can be seen in Figure 8. To do this for each memory bank we record the remote reads, the remote writes, the local reads, and the local writes. Each of these is then divided by the average rate instructions were executed by the threads on the socket that that traffic was to or from.

5.3 Static fraction

After normalizing the data transfers the first properties calculated for the job description are the static socket, and the static fraction. To calculate these we use the normalized results of the symmetric profiling run. For each memory bank we sum the local and remote reads (l_reads and r_reads) or writes to generate the total normalized reads or writes to the memory bank. For example:

$$reads_{bank\ 1} = l_reads_{bank\ 1} + r_reads_{bank\ 1}$$

From these we first calculate the socket that the static fraction is associated with by observing which memory bank transferred the largest volume of data, in the case of the example in Figure 8 this is socket 2 as shown in Figure 9.

Having determined the static socket we next calculate the static fraction. To do this we calculate the additional data transfer on the static socket relative to the other sockets and divide this additional transfer by the total transfer used by the workload. This gives us the fraction of the transfer that was for data statically allocated to a single socket, and there in the fraction of the bandwidth that will be to the static socket for each thread.

$$static\ fraction = \frac{reads_{bank\ 2} - reads_{bank\ 1}}{reads_{bank\ 1} + reads_{bank\ 2}}$$

In this example this works out as 0.2 or 20%.

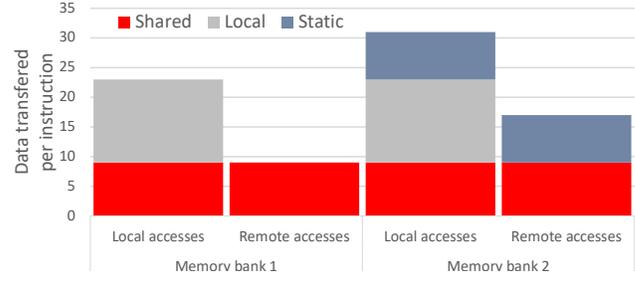


Figure 10: Remote and local accesses to memory banks on each socket. The accesses already classed as static and removed from our calculations are highlighted along with the accesses that will be classes as thread local.

5.4 Local fraction

To calculate the fraction of data transfers due to accesses to thread local data we first remove the data transfer due to the static fraction. In the working example this is just a case of deducting half the bandwidth associated with the static fraction from bank 2's remote accesses and half from its local accesses.

This leaves us with the accesses to data shared between sockets which is made up of per thread data and interleaved data, and the accesses to data that is only used by threads on a given socket, local data. For each socket we then calculate the fraction of accesses that are remote.

$$r_i = \frac{remote\ accesses_{bank_i}}{remote\ accesses_{bank_i} + local\ accesses_{bank_i}}$$

For the symmetric pattern Interleaved and Per thread accesses are indistinguishable meaning we can deduce that if there were no accesses to local memory allocations, for s sockets we would expect the fraction of remote accesses to be $r = \frac{s-1}{s}$. Adding back in the possibility of a non zero local fraction including scaling to allow for the static fraction that we have already removed from the bandwidths gives:

$$r = \frac{s-1}{s} \left(1 - \frac{local\ fraction}{(1 - static\ fraction)} \right)$$

This can be rearranged to get the local fraction. In our example the measured value of r is 0.28125, with no bandwidth categorized as local memory accesses we would expect $r = 0.5$ in this example as $s = 2$. From this we calculate that the local fraction is 0.35. This split along with the static accesses can be seen in Figure 10.

5.5 Per thread fraction

As discussed on a symmetric placement per thread and interleaved accesses are indistinguishable. To overcome this we use a run with an asymmetric placement to calculate this

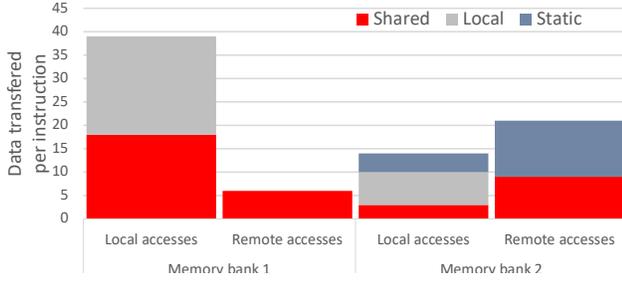


Figure 11: The normalized results for the asymmetric case with the static and local components calculated and removed from the computation of the per thread fraction.

value. For our worked example we will use the placement in Figure 7. Taking the results for this new placement we sum the bandwidth for each CPU, for example for reads:

$$\text{reads}_{\text{CPU } 1} = \text{l_reads}_{\text{bank } 1} + \text{r_reads}_{\text{bank } 2}$$

Next we remove from the static socket the static fraction of the bandwidth. In our example this is done for reads as follows:

$$\text{r_reads}'_{\text{bank } 2} = \text{r_reads}_{\text{bank } 2} - \text{static fraction} \times \text{reads}_{\text{CPU } 1}$$

$$\text{l_reads}'_{\text{bank } 2} = \text{l_reads}_{\text{bank } 2} - \text{static fraction} \times \text{reads}_{\text{CPU } 2}$$

We then remove the fraction of the local bandwidth associated with each memory bank. This is done as follows:

$$\text{l_reads}'_{\text{bank } 1} = \text{l_reads}_{\text{bank } 1} - \text{local fraction} \times \text{reads}_{\text{CPU } 1}$$

$$\text{l_reads}''_{\text{bank } 2} = \text{l_reads}'_{\text{bank } 2} - \text{local fraction} \times \text{reads}_{\text{CPU } 2}$$

The result of this for our worked example can be seen in Figure 11.

Having removed the already accounted for elements of the bandwidth we calculate for each CPU the fraction of its bandwidth that is used for transfers to its local memory bank (l).

$$l = \frac{\text{local accesses}_{\text{bank } i}}{\text{local accesses}_{\text{bank } i} + \sum_{j=1}^{s, j \neq i} \text{remote accesses}_{\text{bank } j}}$$

In our worked example these are $\frac{2}{3}$ and $\frac{1}{3}$ for sockets 1 and 2 respectively.

We also calculated the expected value of l if all the data is accessed on a per thread basis. This value is given by:

$$\text{Per thread data}_i = \frac{n_i}{\sum_{j=1}^s n_j}$$

where n_i is the number of threads on socket i and s is the number of sockets. Next we calculate the expected local fraction if all the data is interleaved:

$$\text{Interleaved}_i = \frac{1}{s}$$

In our example this provides fractions of $\frac{3}{4}$ and $\frac{1}{4}$ if all bandwidth is to Per thread data and $\frac{1}{2}$ and $\frac{1}{2}$ if all bandwidth is to Interleaved data. As the combination for a given application will be somewhere between these two points we can interpolate between them:

$$l_i = \frac{\text{Per thread data}_i}{\text{data}_i} \times p + \frac{\text{Interleaved data}_i}{\text{data}_i} \times (1 - p)$$

This equation can then be rearranged to get the value of p , in our example $\frac{2}{3}$. p can then be scaled to get the Per thread fraction as follows:

$$\text{per thread fraction} = p \times (1 - \text{local fraction} - \text{static fraction})$$

In the worked example this is 0.3. The Per thread fraction is bounded between $[0 \dots 1]$ to ensure that unusual data patterns cannot cause unexpected effects. We will discuss this further in Sections 6 and 7.

6 EVALUATION

We describe the evaluation in 3 sections. First we examine synthetic applications where we have control over the memory placement and compare the measured placement with the known values to confirm the model is successfully identifying the access patterns. Second we look at the signatures generated for a set of benchmarks designed to mimic real world applications. We compare these on the different hardware to examine the stability of the model with more complex workloads. Finally we compare the prediction of the model using these signatures with measured results to gauge the accuracy of the model.

Two experimental systems were used to evaluate this work. These systems are dual socket machines populated with Xeon E5-2630 v3, and Xeon E5-2699 v3 processors which are 8 core, and 18 core Haswell processors respectively. While the processor architectures are similar, the communications profile of the system as a whole is very different as shown in the Figure 2. From this we can see that both systems have similar read and write bandwidths to local memory, but drastically different performance when accessing remote memory where the 8 core processors only have 0.16 of the bandwidth for remote reads and 0.23 of the bandwidth for remote writes relative to local reads and writes. On the 18 core processors the bandwidths are much more comparable to local bandwidths with 0.59 of the bandwidth for remote reads and 0.83 of the bandwidth for remote writes.

As Linux may try to adjust the placement of memory during the execution of the benchmark we disable autonuma

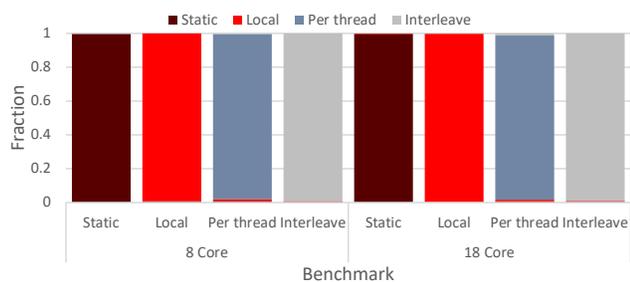


Figure 12: Graphs of the memory signature measured for each of the synthetic benchmarks on 8 and 18 core Haswell processors.

in all our tests. This ensures we measure the benchmarks in a stable state.

6.1 Synthetic benchmarks

The synthetic benchmarks perform index chasing through an array. Arrays of integers are constructed such that each element in the array is an index to the next element that should be read i.e. at each step $i = A[i]$. For these tests the indexes access elements sequentially with a stride size of a cache line and the final element indexing the first. The arrays are typically in the order of gigabytes in size. This means they fit in memory, but not in the cache. This results in a regular access pattern that hardware prefetchers can use to improve performance, minimal use of the cache so we have the strongest signal to noise ratio possible, and code that cannot be analyzed and optimized away at compile time as the content of the array is unknown to the compiler. For Static, Local, and Interleaved each thread constructs its own loop in an array and iterates round it. In the cases of Static and Interleaved we used `numactl` to enforce the placement of the memory, while local relies on a first touch memory policy to ensure the required pages are placed local to the requesting thread. For Per thread each thread constructs an array, and each thread will then iterate through each array in turn.

The results from profiling these benchmarks can be seen in Figure 12. For all the benchmarks the results are as expected with the largest volume of miscategorized bandwidth measuring less than 0.9% which is spread evenly across the remaining 3 categories and can easily be accounted for by background noise.

6.2 Real benchmarks

The workloads for these tests are drawn from a range of sources: The NAS parallel benchmark suite (NPB) [1]; The SPEC OpenMP workloads (OMP) [20]; In-memory graph

Benchmark	Description
Applu	Parabolic / Elliptic PDE solver (OMP)
Apsi	Meteorology pollutant distribution (OMP)
Art	Neural network simulation (OMP)
BT	Block tri-diagonal solver (NPB)
Bwaves	Blast wave simulation (OMP)
CG	Conjugate gradient (NPB)
EP	Embarrassingly parallel (NPB)
Equake	Earthquake simulation (OMP)
FMA-3D	Finite-element crash simulation (OMP)
FT	Discrete 3D fast Fourier transform (NPB)
IS	Integer sort (NPB)
LU	Lower-upper Gauss-Seidel solver (NPB)
MD	Molecular dynamics simulation (NPB)
MG	Multi-grid on a sequence of meshes (NPB)
NPO	No partitioning, optimized hash join (DBJ)
PRHO	Parallel radix histogram optimized hash join (DBJ)
PRH	Parallel radix histogram hash join (DBJ)
PRO	Parallel radix optimized hash join (DBJ)
Page rank	In-memory parallel Page rank (GA)
Sort join	In-memory sort-join (DBJ)
SP	Scalar Penta-diagonal solver (NPB)
Swim	Shallow water modeling (OMP)
Wupwise	Wuppertal Wilson fermion solver (OMP)

Table 1: Description of benchmarks.

analytics workloads (GA) from Harris *et al* [11]; And database join operators (DBJ) from Balkesen *et al* [2]. They are designed to mimic real world applications and we describe them in Table 1.

6.2.1 Model stability. Calculating the bandwidth description on both machines for each benchmark for both reads and writes we get the signatures in Figure 13. Figure 14 compares the percentage of the bandwidth that is reallocated between the two different signatures. At first glance some of these results appear to be very bad with a change in excess of 80% for equake writes. This is due to this benchmark performing almost exclusively reads with the very small number of writes resulting in a very low signal to noise ratio. So while the signature does change for these examples the bandwidth associated with this is negligible. To illustrate this we also plot the difference in signature value if instead of constructing a separate signature for reads and writes, we combine the bandwidths and construct the signatures using these combined bandwidth figures. The combined figures for equake change by 5.4%. The mean change for all the benchmarks 6.8% and the median is 4.2%.

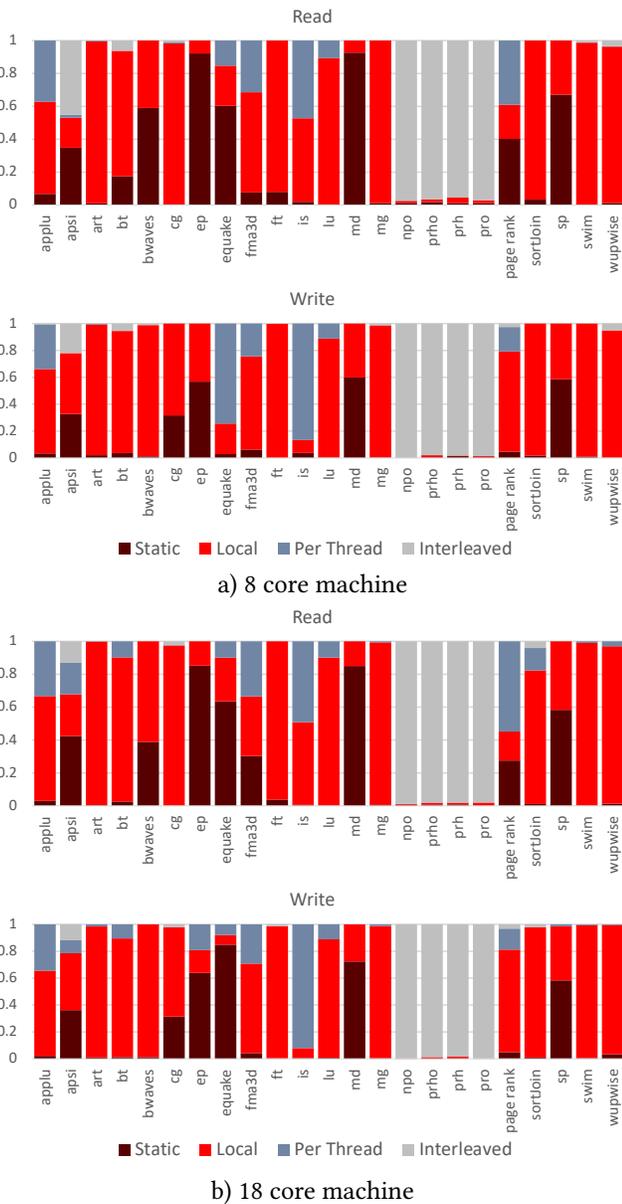


Figure 13: Graph of the bandwidth signatures generated by the benchmarks for reading and writing on the different systems

The difference between the mean and median highlights that while for most of the benchmarks the signature is stable and usable for a small number there are significant errors. This can be seen in Figure 15 where we show a cumulative frequency graph of the benchmarks and the change between signatures. From this we can see that over 50% of applications have a change of less than 5% and over 75% of applications have a change of less than 10%. However, there

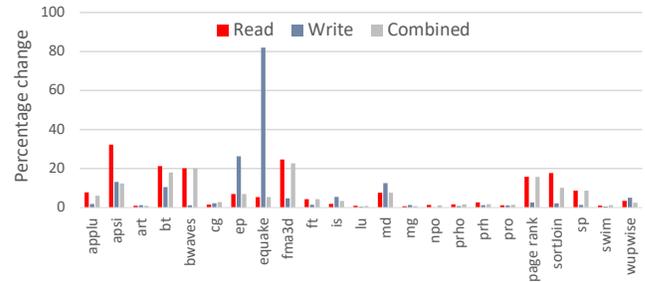


Figure 14: Graph of the percentage change in the bandwidth placement between systems.

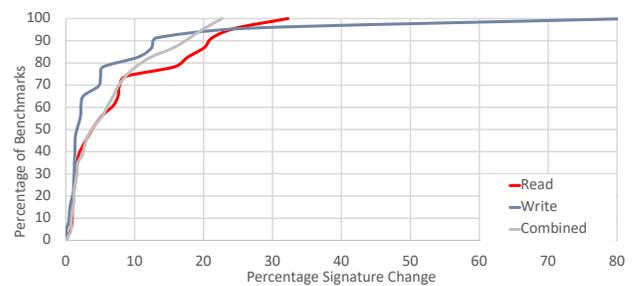


Figure 15: Cumulative frequency graph of the change in the benchmarks signatures relative to the percentage of the benchmarks.

is a set of benchmarks for which the model does not provide a very good fit to the access patterns. These occur when the bandwidth requirements varies between threads and is especially problematic when it changes with the number and position of the threads. For example in the Page rank benchmark the nodes in the graphs are listed in the order they were visited when the dataset was collected. As a result of being collect through walking the graph well connected nodes are more likely to be met first and so appearing earlier in the dataset. This happens because after a short period of exploring the graph the walker may reach a well connected segment. From here it will probably mostly reach other well connected nodes. The later nodes to be reached will more likely be weakly connected hence taking longer to be discovered by the walker. As a result the part of the graph that appears earlier in the dataset is better connected on average than the rest of the graph. This results in higher local bandwidth requirements on the first socket which will erroneously be marked as static bandwidth. This then confuses the calculation of both local and per thread fractions. When the threads are moved around, the bandwidth requirements fail to change in the way described. An example of this can be seen in Figure 16.

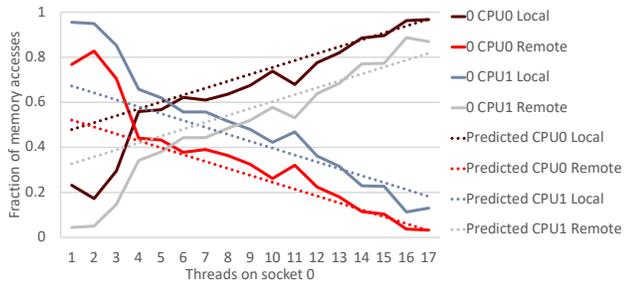


Figure 16: Graph of the measured and predicted results for the combination of reads and writes with Page rank.

Fortunately it is possible to detect when situations like this occur as there is redundant information in the program counters that highlights the inconsistency. For example once we remove the static fraction with the symmetric placement we expect the placement to be symmetric. If when we examine the local remote ration for each socket we find that it is not symmetric this is a sign that the application does not fit the model. The bigger the difference the worse the fit.

6.2.2 Model accuracy. To test the accuracy of the model with the different benchmarks we executed each benchmark with the largest thread count that it could support on a single socket with at most one thread per core. For each benchmark configuration we then varied the distribution of the threads between the two sockets maintaining a single thread per core. Measuring the local and remote reads and writes for each socket and comparing against the read, write, and combined model predictions gives a large number of comparison points with the 18 core machine providing 2322 data points to compare the predicted results against. The results of this for the Page rank benchmark when comparing the combined reads and writes bandwidth prediction can be seen in Figure 16. This shows the discussed poor fitting of the model to threads working on the first section of the graph, and the more effective modeling of the bandwidth requirements for processing the rest of the graph.

For each data point we record the difference between the measured value and the predicted value. Looking at these across all our experiments we can generate the cumulative frequency graph in Figure 17. This shows that for over 50% of the measurements the difference is less than 2.5% of the total bandwidth, and for 75% of measurements the difference is less than 10% of the total bandwidth. Looking at where these differences occur in Figure 18 we plot the average difference for each benchmark relative to the average bandwidth used by the benchmark. This shows that the substantial errors only occur in the benchmarks with low bandwidth requirements. Such benchmarks both have a low signal to noise

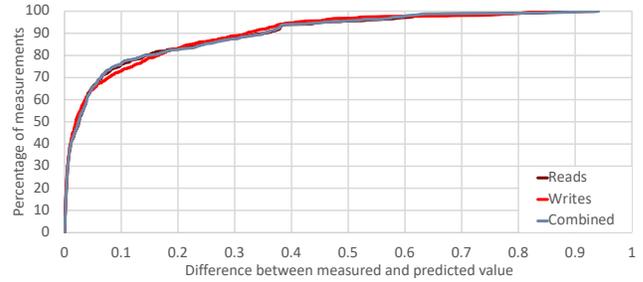


Figure 17: Cumulative frequency graph of the percentage of the benchmark measurements vs the size of the error.

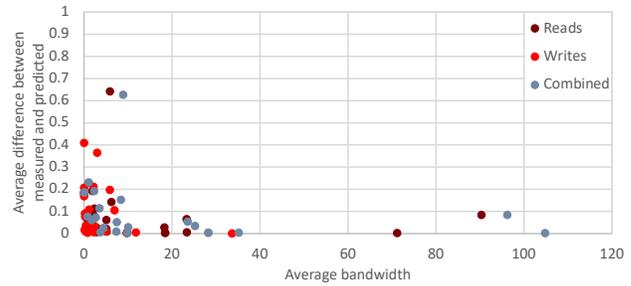


Figure 18: Graph of the average size of the error vs the average amount of bandwidth required by the benchmark.

ratio, but also do not require high accuracy as by definition they are not moving large quantities of data so will be less affected, and less likely to interfere with the data transfers of other workloads.

7 FUTURE WORK

As mentioned in the evaluation the current model has one key limitation. It assumes that each thread accesses data with the same frequency relative to its rate of execution. However as described there is information that allows for these situations to be detected. It would be preferable to instead of just detecting these situations to model them as well, extending the applicability of this model. This is a focus for future work.

8 RELATED WORK

Asymmetry and variance of bandwidth performance on NUMA nodes leading to contention on resources has long been identified as a problem [3, 19, 27]. This has resulted in the development of many techniques to improve the placement of

threads and memory. These pieces of work measure the performance of the NUMA machine or the loads being executed through a range of techniques including:

- Monitoring performance counters while executing either known benchmarks, or arbitrary workloads [3, 16, 19]. The performance counters monitored are typically LLC misses and instructions executed, but can be far more extensive [10, 22], and extend to applying ML techniques to every counter on the machine [26].
- Sampling instructions as they are executed [6] to identify memory loads and stores along with information about where they are located on the system. This can provide very fine grained information about locations of memory accesses.
- Monitoring page faults to identify suboptimal page placements and configurations [13].
- Running benchmarks with known loads and timing their execution to determine latency and or bandwidth properties [14, 23].

These techniques are applied to many different locations including, improvements to the OS task scheduler, improvements to the OS memory management, and tools and libraries to assist the programmer both by providing information about the expected behavior of the system, and to automate memory placement.

Approaches to scheduling and memory management generally fall into two categories, real time approaches that detect when something is suboptimal, and ahead of time approaches that measure the applications and or system beforehand to provide guidance to the OS.

Real time approaches such as DINO [3] migrate threads and their associated memory pages based on the LLC miss rate, while Merkel *et al.* [19] leave memory where it is allocated and just move threads around. Carrefour [6] addresses congestion on memory controllers by sampling the executed instructions observe the loads and stores of different threads. From this it determines if the load on the memory controllers is too high and follows a decision tree to determine if pages should be migrated, interleaved, or replicated to reduce load. Lankes *et al.* [13] produced suggestions for extending the OS memory management with a page table per NUMA node, allowing pages to be replicated on demand if the load on the interconnect is too high.

Ahead of time tool such as Pandia [10] and libraries like Smart Arrays [22] perform sample runs and provide suggested thread or memory placements, these lack detailed bandwidth models so cannot apply these runs to different thread counts and placements as effectively.

Many of the approaches require detailed specifications of the system. This is typically generated ahead of the time

through the use of benchmarks and either performance counters or wall clock time. Mc Cormick *et al.* [23] measuring the NUMA properties based on 2 benchmarks, measuring latency, and bandwidth to help the scheduler calculate costs and change tasks node/core based on the location of the application memory. Majo *et al.* [16] took a similar approach using a single memory intensive benchmark. Pandia uses an extensive set of synthetic benchmarks to measure the memory architecture, as well as the processors compute performance. Li *et al.* [14] take a similar extensive approach considering data transfer to disks, networks and GPU's. To calculate the data transfer rates they copy large amounts of data with *memcpy* and time the transfer. From this they build an NxN matrix. This is expensive to maintain so they categorize the machine into groups and provide timings for each group.

So Many Performance Events, So Little Time [26] does not model the memory traffic directly, but uses a set of benchmarks which cause specific problems and ML approaches which look at all performance counters. From these they construct a model with a small number of features that can be driven from the counters available to a single run. The intention of this work is to act as a debugging tool for programmers, with the memory bandwidth and latency issues being detected. As with all ML approaches it requires a sizable training set, and exploring all of the performance counters takes time.

Looking away from the memory interconnects there has been extensive work focusing on the expected behavior of caches and the performance of applications both alone and when sharing a cache. Much of this builds on ideas on measuring the performance of the cache and the applications that are using it, but normally this work constrains itself to just the cache excluding the wider memory system. Predicting based just on cache interference can be done with mathematical models on data gathered from instrumented workloads [4, 17, 24]. McGregor *et al.* [18], Knauerhase *et al.* [12], Fedorova *et al.* [9], Zhuravlev *et al.* [27], Collins *et al.* [5] and Dhiman *et al.* [8] select threads or workloads to collocate based on performance counters giving measurements such as bus transactions per thread, stall cycles per thread, and LLC miss rate per thread. Xie and Loh classify applications [25] based on LLC usage and Lin *et al.* [15] on L2 usage. These all work to a fixed number of threads, ReSense [7] can dynamically controls threads numbers.

9 CONCLUSION

In this paper we have presented and evaluated a model for estimating the bandwidth distribution of analytics applications on NUMA machines. The model is fitted to applications through the use of two instrumented runs with specific thread placements. Testing the results across 1000's of measurements shows a high degree of accuracy for many applications with moderate to high bandwidth requirements, and techniques to detect when the model fails to fit effectively. The ideas presented here have uses ranging from performance debugging tools, to scheduling tools such as Pandia [10], to libraries that can control memory placements [22].

REFERENCES

- [1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks; Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. ACM, 158–165. <https://doi.org/10.1145/125826.125925>
- [2] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. 362–373. <https://doi.org/10.1109/ICDE.2013.6544839>
- [3] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. 2011. A Case for NUMA-aware Contention Management on Multicore Systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'11)*. USENIX Association, Berkeley, CA, USA, 1–1. <http://dl.acm.org/citation.cfm?id=2002181.2002182>
- [4] Dhruva Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. 2005. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*.
- [5] Alexander Collins, Tim Harris, Murray Cole, and Christian Fensch. 2015. LIRA: Adaptive Contention-Aware Thread Placement for Parallel Runtime Systems. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '15)*. ACM, Article 2, 8 pages. <https://doi.org/10.1145/2768405.2768407>
- [6] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 381–394. <https://doi.org/10.1145/2451116.2451157>
- [7] Tanima Dey, Wei Wang, Jack W. Davidson, and Mary Lou Soffa. 2013. ReSense: Mapping Dynamic Workloads of Colocated Multithreaded Applications Using Resource Sensitivity. *ACM Transactions on Architecture and Code Optimization* 10, 4, Article 41 (Dec 2013), 41:1–41:25 pages.
- [8] Gaurav Dhiman, Giacomo Marchetti, and Tajana Rosing. 2009. vGreen: A System for Energy Efficient Computing in Virtualized Environments. In *Proceedings of the 14th International Symposium on Low Power Electronics and Design*. ACM, 243–248.
- [9] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. 2007. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. IEEE, 25–38.
- [10] Daniel Goodman, Georgios Varisteas, and Tim Harris. 2017. Pandia: Comprehensive Contention-sensitive Thread Placement. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, 254–269. <https://doi.org/10.1145/3064176.3064177>
- [11] Tim Harris and Stefan Kaestle. 2015. Callisto-RTS: Fine-Grain Parallel Loops. In *2015 USENIX Annual Technical Conference, USENIX ATC '15*. 45–56. <https://www.usenix.org/conference/atc15/technical-session/presentation/harris>
- [12] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. 2008. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro* 28, 3 (May 2008), 54–66.
- [13] Stefan Lankes, Thomas Bemmerl, Thomas Roehl, and Christian Terboven. 2012. Node-based Memory Management for Scalable NUMA Architectures. In *Proceedings of the 2Nd International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '12)*. ACM, New York, NY, USA, Article 10, 8 pages. <https://doi.org/10.1145/2318916.2318929>
- [14] Tan Li, Yufei Ren, Dantong Yu, Shudong Jin, and Thomas Robertazzi. 2013. Characterization of Input/Output Bandwidth Performance Models in NUMA Architecture for Data Intensive Applications. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing (ICPP '13)*. IEEE Computer Society, Washington, DC, USA, 369–378. <https://doi.org/10.1109/ICPP.2013.46>
- [15] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *14th International Conference on High-Performance Computer Architecture, HPCA-14 '08*. 367–378. <https://doi.org/10.1109/HPCA.2008.4658653>
- [16] Zoltan Majo and Thomas R. Gross. 2011. Memory System Performance in a NUMA Multicore Multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage (SYSTOR '11)*. ACM, New York, NY, USA, Article 12, 10 pages. <https://doi.org/10.1145/1987816.1987832>
- [17] Gabriel Marin and John Mellor-Crummey. 2004. Cross-architecture Performance Predictions for Scientific Applications Using Parameterized Models. *SIGMETRICS Performance Evaluation Review* 32, 1 (June 2004), 2–13. <https://doi.org/10.1145/1012888.1005691>
- [18] Robert L. McGregor, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. 2005. Scheduling Algorithms for Effective Thread Pairing on Hybrid Multiprocessors. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society.
- [19] Andreas Merkel, Jan Stoess, and Frank Bellosa. 2010. Resource-conscious Scheduling for Energy Efficiency on Multicore Processors. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. ACM, New York, NY, USA, 153–166. <https://doi.org/10.1145/1755913.1755930>
- [20] Matthias S. Müller, John Baron, William C. Brantley, Huiyu Feng, Daniel Hackenberg, Robert Henschel, Gabriele Jost, Daniel Molka, Chris Parrott, Joe Robichaux, Pavel Shelepugin, Matthijs van Waveren, Brian Whitney, and Kalyan Kumaran. 2012. *SPEC OMP2012 — An Application Benchmark Suite for Parallel Systems Using OpenMP*. Springer Berlin Heidelberg, Berlin, Heidelberg, 223–236. https://doi.org/10.1007/978-3-642-30961-8_17
- [21] Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. 2011. A Generic Parallel Collection Framework. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II (Euro-Par '11)*. Springer-Verlag, Berlin, Heidelberg, 136–147. <http://>

[//dl.acm.org/citation.cfm?id=2033408.2033425](https://dl.acm.org/citation.cfm?id=2033408.2033425)

- [22] Iraklis Psaroudakis, Stefan Kaestle, Matthias Grimmer, Daniel Goodman, Jean-Pierre Lozi, and Tim Harris. 2018. Analytics with Smart Arrays: Adaptive and Efficient Language-independent Data. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, Article 17, 15 pages. <https://doi.org/10.1145/3190508.3190514>
- [23] Patrick S. McCormick, Ryan Braithwaite, and Wu-chun Feng. 2011. *Empirical Memory-Access Cost Models in Multicore NUMA Architectures*. Technical Report. Virginia Tech Department of Computer Science.
- [24] Richard West, Puneet Zaro, Carl A. Waldspurger, and Xiao Zhang. 2010. Online Cache Modeling for Commodity Multicore Processors. *SIGOPS Operating Systems Review* 44, 4 (Dec. 2010), 19–29. <https://doi.org/10.1145/1899928.1899931>
- [25] Yuejian Xie and Gabriel H. Loh. 2008. Dynamic Classification of Program Memory Behaviors in CMPs. In *Proceedings 2nd Workshop on CMP Memory Systems and Interconnects (CMP-MSI)*.
- [26] Gerd Zellweger, Denny Lin, and Timothy Roscoe. 2016. So Many Performance Events, So Little Time. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '16)*. ACM, New York, NY, USA, Article 14, 9 pages. <https://doi.org/10.1145/2967360.2967375>
- [27] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 129–142. <https://doi.org/10.1145/1736020.1736036>