2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)

Modeling Cache and Application Performance on Modern Shared Memory Multiprocessors

Junsung Yook and Bernhard Egger
Seoul National University
Seoul, Republic of Korea
{junsung,bernhard}@csap.snu.ac.kr

***Abstract*—Modern processors include a cache to reduce the access latency to off-chip memory. In shared memory multiprocessors, the same data can be stored in multiple processor-local caches. These private copies reduce contention on the memory system, however, incur a replication overhead. Multiple copies consume valuable cache resources and thus increase the likelihood for capacity misses. Maintaining cache coherence is another difficulty caused by multiple copies. In particular, to set a cache line's status to exclusive in one cache requires invalidating all other shared copies, which can significantly stress the processor interconnect. Furthermore, loading data from a remote cache incurs a large overhead. In the absence of source code or data layout modifications, a rearrangement of a parallel application's threads can often reduce cache line replication significantly. By mapping threads that frequently access the same cache lines to the same processor node, redundant duplicates and excessive invalidation can be minimized. In this paper, we devise a closed queuing network model to compare the performance of different thread arrangements onto the nodes of a multiprocessor system in order to predict the expected optimal arrangement. The inputs to the model are obtained through a single profiling run. The outputs of the queuing network are performance indices such as throughput, utilization, and latency for the different components of the memory system. Based on these metrics, we compute the memory stall time of individual cores and predict application runtime. Evaluated on a 72-core 4-node Intel Xeon architecture, the presented model is able to identify the best thread arrangement from a set of six configurations for 20 out of 21 parallel applications from various benchmark suites.**

***Index Terms*—Memory system model, closed queueing networks, cache line profile, thread placement**

I. Introduction

Processor caches are one of the most successful ideas to bridge the gap between the fast processor cores and the comparatively slow off-chip memory. By keeping data nearby, the cores can avoid long stall times caused by long-latency accesses to off-chip memory. In the context of multiprocessors, each processor node has its own private cache, and the same physical data can be present in multiple caches. This has the advantage of reducing contention on the off-chip memory at the expense of occupying more resources in the individual caches by the same data. Additionally, private copies require a coherence protocol that invalidates other copies in remote caches upon writes to a cache line.

The design of modern memory subsystems has followed the Non-Uniform Memory Access (NUMA) architectures with private caches to exploit data locality. Non-Uniform Cache Access (NUCA) architectures, on the other hand, employ a shared last level cache (LLC) that is statically organized into private partitions. The access latency to cached data varies in dependence of the location of the accessing core and the physical location where the data is stored [1], [2]; hence, models estimating the memory stall time need to take the distance to the accessed data into account [3].

Multiprocessor architectures with NUMA properties typically interconnect processor sockets point-to-point. Such a design scales to the typical number of processor sockets and also renders the last level cache private to the processor. A disadvantage of a dedicated last-level cache is that duplicated data residing in different last-level caches can cause a lower capacity utilization. Another problem is that a processor node needs to exclusively own a cache line before modifying it. To grant exclusive ownership, all copies of that cache line residing in other nodes need to be invalidated. If several threads residing on different processor nodes frequently write to a shared cache line, the resulting cache coherence messages can lead to a so-called invalidation storm.

In NUMA architectures composed of chip multiprocessors, multiple threads can be mapped to the same node. On that node, threads run in parallel on multiple cores. Since the last-level cache is shared by all cores inside a node, the arrangement of threads to nodes has a significant impact on performance. To optimally map the threads of a parallel application to cores, the affinity in terms of accessed shared data needs to be considered.

The latency of a memory access is composed of the aggregate service times of the different components in the memory system and the time spent waiting due to contention. To increase the efficiency of the memory subsystem, our goal is to minimize the reload overhead. This can be achieved by two means. First, we can reduce the reload frequency caused by capacity misses. Placing threads that operate on the same data onto the same processor node lowers the amount of distinct cache lines in the LLC and, as a consequence, the cache experiences fewer capacity misses. The second mean is to reduce the reload penalty. When filling a cache line, the accessed data may be located in local memory, in remote memory, or in a remote cache. The location of the data with respect to the accessing thread has a significant impact on the memory system's service time. Placing threads close to the accessed data not only reduces off-chip memory access time but also the overall traffic in the memory subsystem and, thus, contention.

978-0-7381-2646-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00158

These two techniques, however, are not well suited for comparison and do not allow for a direct estimation of performance for a given thread allocation because many other factors affect application performance. For example, if a thread is migrated to another node to improve data sharing between threads, this may cause additional off-chip access latency to now remote memory. The complex hardware and interference by the operating system make it extremely challenging to accurately predict the execution time of parallel applications with respect to a given thread assignment.

In this paper, we present our approach at tackling this challenge with a technique that predicts the optimal arrangement of threads considering the aformentioned trade-offs. First, we introduce a model that predicts the memory transactions of a parallel workload. This workload model infers the frequency and the types of memory transactions such as the number of reloads caused by cache invalidations or capacity misses. The model is fed with input parameters profiled during a single run of the parallel workload. We query the processor's performance monitoring unit (PMU) to obtain the required parameters such as the total number of memory requests and a sample of the accessed memory addresses.

Next, we devise an analytic queuing network model of an (abstract) memory subsystem to predict application performance by comprehensively analyzing the trade-offs of different thread allocations. A closed queuing network provides performance indexes such as throughput, busy probability and latency for the components of the memory subsystem considering the service time and access frequency of each device. The stall time of the abstracted core represents the actual stall cycle of the physical core caused the access latency to the memory subsystem. By measuring stall cycles, we estimate the number of tokens needed to manage the requests to the components of the memory subsystem. The number of tokens is determined by the degree of ILP (Instruction Level Parallelism) and the ability of the memory subsystem to handle outstanding memory requests. The closed queuing model allows us to predict the runtime of different thread allocations. As a result, the performance of given workload can be evaluated by considering the service time of the components that a given transaction goes through and the delay time due to bottlenecks.

The presented model is evaluated on a 72-core 4-node Xeon E7-8870 v3 system with 512 GiB of DRAM. The model successfully identifies the approximate optimal configuration from a set of total six configurations for 20 of 21 parallel workloads from various benchmark suites.

II. Related Work

While we combine a number of techniques in a novel way, each of the individual techniques have been studied before. The *Thiebaut and Stone model* is based on the notion of a task's footprint in the cache (the group of cache blocks in active use by the task) [4] [5]. In the study, in order to obtain the reload transient when two threads are switched, the size of the set and the allocated block were defined in advance. The identification of cache lines allows the authors to examine the overlap between parallel threads to estimate the number of distinct shared cache lines and thus the occupation of cache blocks. The number of distinct cache lines requested by a node representes the total number of attempts, and the average of the binomial distribution of the last level cache (LLC) hit probability is considered to represent the capacity of the LLC.

Tam used the PMU to sample cache lines. In the study, threads were clustered considering thread-affinity with regards to communication and load balance [6]. In our work, we consider not only communication but also all our principles by inferring transactions and evaluating them comprehensively. In research for memProf, memory objects were considered as the granularity of memory access [7]. Such coarse objects caused inevitable remote accesses due to false sharing. We use the memory page as the granularity of memory access as to treat the distance of access in a more sophisticated way.

Previous studies have used closed queuing networks to explore the design space for an evaluation of the snoop protocol [8], the network topology [8], and the number of outstanding requests of Instruction Level Parallelism (ILP) processors [9]. *Willick* proposed a modeling approach which employs a closed multiclass queuing network model (QNM) to represent a multiprocessor system [8]. In their work, the customers in the QNM represent network requests and replies. Also, the number of customer classes is equal to the number of processors. Each customer class corresponds to the requests generated by one particular processor. The number of customers in each class is equal to the number of outstanding requests a processor can issue before it must block and wait for some replies to return from memory. We also employ a closed multiclass queuing network model for the memory subsystem model. However, our approach differs in terms of the variety of costumers. We distinguish between the load demand caused by the tasks and the prefetch load generated by the hardware prefetcher. Unlike the load demand, the prefetch load does not cause stalls directly. Neverthless, prefetch transactions place an additional burden on the components of the memory subsystem. These side effects, in the end, have a big impact on performance evaluation. In another research on ILP processors, *Sorin* considered data dependencies as another cause of core stall [9]. We also reflect data dependencies in the number of customers.

III. Background

A. Queuing network model

Queuing network modelling is a particular approach to computer system modelling in which the computer system is represented as a network of queues that are evaluated analytically. Analytic evaluation involves using software to solve efficiently a set of equations induced by the network of queues and its parameters. The basic entities in queuing network models are the service centers, which represent system resources, and customers, which represent users, jobs, or transactions. A class of queuing network models (called BCMP models) with different classes of customers has been solved by Baskett et al. [10]. Multiple class models, like single class models, provide estimates for performance measures such as utilization, throughput, and response time [11].

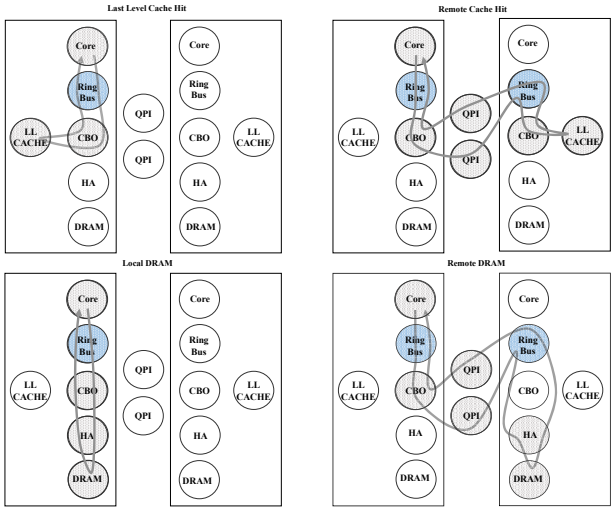

Fig. 1. **Transaction Types.** We distinguish four transactions types that each exercise different parts of the memory system: *last level cache hit, remote cache hit (modified line, forward line), local DRAM, and remote RAM access*.

B. The performance monitoring unit

Modern processors include performance monitoring units (PMUs) that collect usage information of the many hardware components in the processor. Every core has its own PMU that counts core events such as the number of executed instructions, the number of L1 cache accesses, etc. Multi-core processors contain additional PMUs to monitor the shared resources. A PMU is first programmed by writing the desired event ID and other settings to the control register, then, the event count can be read from the performance monitoring counter (PMC) register [12]. PMUs were primarily intended for verification purposes [13]. These days, however, PMUs are an important source of information to detect performance issues [14] and to optimize execution of workloads on a given system. Eranian [15], for example, show that PMUs can be used to detect performance problems, although their study does not include a systematic analysis of the available events.

IV. The Workload Model

For our trace-based workload model, samples of addresses to cache lines classified by response type are needed. This enables us to analyze the current pattern of transactions and to predict changes in dependence of the thread allocation. The closed multiclass queuing network is then used to model the performance parameters of the memory system and thereby predict overall application performance.

A. Types of transactions from memory requests

As Fig. 1 illustrates, we distinguish between four transaction types that each exercise different parts of the memory system. The four transactions types are *last level cache hit, remote cache hit (modified line, forward line), local DRAM, and remote RAM access*. Each transaction type goes through a different path in the memory system and thus determines which resources are involved and where the target data resides. To identify trace, target nodes should be identified.

TABLE I
Table for estimated capacity.

Access Weight Level	Estimated Capacity
1	Sum of hit rate for cache lines belonging to Lv1
...	...
16	Sum of hit rate for cache lines belonging to Lv16

TABLE II
Table for estimated hit rate.

Access Weight Level	Estimated Hit Rate
1	Estimated Capacity on Lv1 / Number of distinct lines on Lv1
...	...
16	Estimated Capacity on Lv16 / Number of distinct lines on Lv16

B. Samples of demand load for cache line

We first sample the demand for load and store operations from all cores. The samples contain the addresses of the virtual memory space and the physical addresses that allow us to identify the remote node to which the requested page belongs. These addresses aligned with the cache line size provide evidence to identify the remote target node. The samples are classified by the issuing thread and into the following load response types that are *hit in cache, hit and forward from other core, hit on modified line, forward from remote node, modified line at remote node*.

C. Obtaining the base parameters

- **Table for occupied cache blocks on each access weight.** Using the number of distinct cache lines issued by a node, we can estimate the capacity of the last level cache. The number of distinct cache lines represents the overlapped footprint by intrinsic demand of threads on the same node. We use a binomial distribution as a spatial way. In a sequence of N (number of distinct cache line) independent experiments of the probability of success P (last level cache hit rate), the expected value (NP) of the binomial distribution represents the capacity of the last level cache. The hit rate to a last level cache for a cache line reflects the probability that the cache line is occupying a cache block in the last level cache. Fig. I demonstrates table for estimated capacity.
  To obtain this parameter, we selected the node in which demand loads occur most frequently. We divide the distinct cache lines into 16 levels by access weight uniformly. As Fig. II illustrates, the partitioning reflects not only the weight of the cache line but also the least recently used (LRU) replacement policy. In the presented model, the sum of hit rates of distinct cache lines issued by a given set of threads in a node represents the capacity of the last level cache. That is, the sum of hit rates for

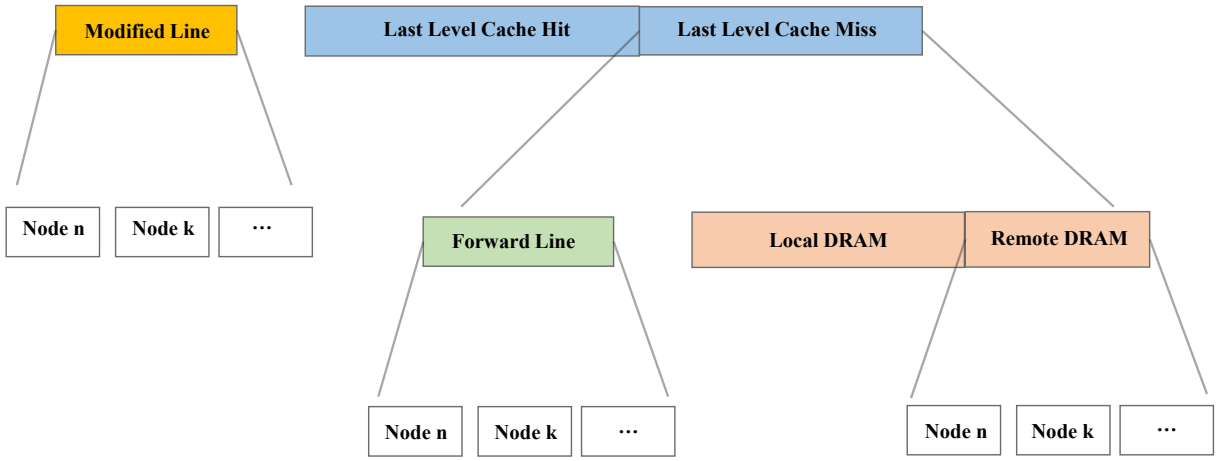

Fig. 2. **Inference for fractions of transactions** First, the fraction of the transaction to the remote hit for modified lines are separated. With the exception of the fraction for the modified line, the rest is divided by the estimated cache hit rate. In the miss fraction, forward were was separated first.

the distinct cache lines belonging to a level denotes the portion of the capacity of the last level cache occupied by the cache lines belonging to this level.

- **Maps for inter-thread communications.** To identify the remote node to which a modified cache line belongs, a mapping between threads communicating with each other is obtained by sampling cache line addresses. Each profiled thread is associated with the threads that store to the same cache line where the events regarding *modified line* occur by the profiled thread. Assuming each thread is dealing with the same amount of data, the number of invalidations of a cache line due to communication is roughly the same for each run. Thus, the number of invalidations due to a relative thread that performs a store to that line can be estimated by the number of *hit on modified line* events of the profiled thread. The invalidation rate over all demand requests for communication between subjected and relative threads are recorded.
- **Maps with forwarding threads.** To identify the remote node to which a forwarded cache line belongs, a mapping of the forwarding threads is performed using the cache line address samples. Each subjected thread is mapped with the threads that forward to the same cache line where the events regarding hit *forward* occur by the subjected thread.

  The two events A (the requested cache line have been loaded first from a thread) and B (the requested cache line misses on the last level cache) are independent. As a result, the conditional probability of the *forward* event to each target thread for the last-level cache miss event is recorded for each of the relative threads that perform a forward operation.

D. Inference for fractions of transactions

The requests of each thread are divided into fractions of transactions issued by the threads. Every thread has a ratio of fractions divided by types of transactions and target nodes. Fractions of transactions are shown in Fig. 2

First, the fraction of transactions to the remote hit for exclusive line is separated, because this type of transaction is independent from the other transactions. Regardless of the hit ratio in the last level cache, invalidations due to inter-threads communication occur consistently. To estimate the fraction of hits in remote modified lines, the maps for inter-threads communications are used. The modified line is divided first by using the invalidation rate caused by communication between nodes.

The sum of the hit rate for each cache line in a level represents the capacity of the last level cache for that level. The new hit rates for the given set of threads on the node can be estimated by referring to the table of the estimated capacity for each access weight level. The capacity of the last level cache on the level divided by footprints represented by the number of distinct cache lines on the level represents the new hits rate.

The predicted last level cache hit fraction determines the fraction of the last level cache hit related transactions. The last level cache miss fraction is split into forward line operations and DRAM accesses. The forward line hit fraction of each target node is inferred by the joint probability of the last level cache miss event and the forward line event. This is because the forward line represents the situation where the cache line was first requested by another thread before the thread tries to access the cache line. To find the fraction of the transactions and the target node, the maps with forwarding threads are used.

The remainder of the last level cache miss fraction subtracted by forward line hits fraction is classified as DRAM

accesses. In the case of accesses to remote DRAM, the physical address of the cache lines is used to identify the remote node to which the requested page belongs.

V. Memory Subsystem Model

The memory subsystem model consists of cores and uncore sub-system components. The service of the thread being loaded represents the execution of the context of a program until a memory transaction occurs. This means that the idle time of the core can be incurred by dependency depending on the latency caused by the memory subsystem. Uncore sub-system components provide the necessary services to process memory transactions occurring in the core. The transactions cause a load on all visited components. In our model, the performance indexes of the system are acquired by loading the transactions into a closed queuing network model. The cores and uncore sub-system components refer to the devices that make up the model. A thread has a unique number of credits required to handle transactions. The number of all credits indicates the number of jobs in the closed queuing network model.

A. Considerations

We assume that the cost of forwarding data from another core and the cost of examining the last level cache are almost same. This is because the sliced last level cache is almost of the same size as the L2 cache. Also, we do not consider the private caches of a core. Moreover, writebacks caused by translations to share the state of dirty pages are not taken into account since data can be delivered to the requesting core before writeback. For the interconnections, we only account for data flit but not the control flit. The control flit is negligible because it is about one eighth the size of the data flit and it is some sort of a maze of regulations. Based on the uncore subsystem of Intel Xeon E7 v3, the ring bus has four devices, the caching agent (CBO) has 18 devices, and IMC has a maximum of eight devices. These components are treated as a device consisting of multiple servers in the queuing network model.

B. Evaluating the service time of components

In order to simulate the performance with a closed queuing network, the service time of the device is required. The utilization rate for devices is calculated as *NOT EMPTY CYCLES / CYCLES* for devices which can be profiled by the PMU. The service time of devices is the same as the *THROUGHPUT / UTILIZATION RATE*.

C. Evaluating the core service time

A core device represents the intrinsic memory demand of a thread. The service of a core device reflects the execution of the thread context. The period with which a load operation occurs represents the service time of the core device. In other words, the utilization rate of the core device is the same as *(CYCLES-STALL CYCLE) / CYCLES*. The service time of the core also includes the time for instruction level parallelism (ILP). Depending on the degree of ILP, the latency

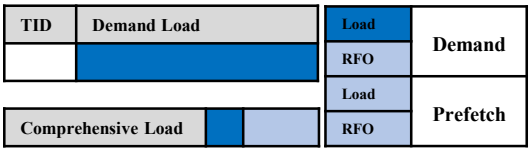

Fig. 3. **Scaling factors for demand load.** The factor that causes stall cycles in the core due to the uncore sub-system is considered only the demand load. Transactions related to prefetch load and Read for Ownership (RFO) affects latency by putting a burden on the devices belonging to the trace.

of outstanding demand loads is covered. Covered areas are compensated for by adjusting the Number of Credits. Prefetch operations do not cause stalls in most cases. For example, the Streamer Prefetcher on Intel architectures can simultaneously maintain up to 32 streams of data accesses and run up to 20 lines ahead of the load request.

D. Scaling factors for comprehensive load operations

In our model, the factor that causes stall cycles in the core due to the uncore sub-system is considered only the Demand Load. Transactions related to prefetch load from the prefetcher and Read for Ownership (RFO) affect the latency by putting a burden on the devices belonging to the trace, as it is considered as side effect of congestion. Our main concern are demand load transactions which determine the stall cycles of the core. We track the demand load transactions to get the busy probability of core device. Therefore, the burden imposed by all load transactions is normalized based on the demand load. To this end, the visits to the devices are scaled based on the demand load transaction that leads all other load transaction. As shown in Fig. 3, Comprehensive Load is composed of Demand and Prefetch requests.

A scaling factor is maintained for each device visited by the transaction of a thread. The bus and interconnection devices have different bandwidths from 64B cache line size that can be transferred at a time. Also, the collected number of CAS READ Event of Integrated Memory Controller (IMC) does not correspond to the number of demand loads because it also includes read and write events of the cache coherence protocol. Thus, we scale the number of load to devices, corresponding to the actual visit count. The IMC translates the write command for the store instruction to the DRAM when the requested load is not in the pending queue. Therefore, we ignore store transaction visits for the IMC components. Also, it only considers data transmit in the interconnection.

E. Transaction load

In order to calculate the visit ratio for devices, a load is applied to the devices affected by the transaction. By investigating the system, we can obtain all devices that the trace of transaction visits. These devices refer to the scaling factors table to obtain the corresponding scaling factors for the transaction of the thread. Load is added to devices belonging to the trace according to the fraction of the transactions. The

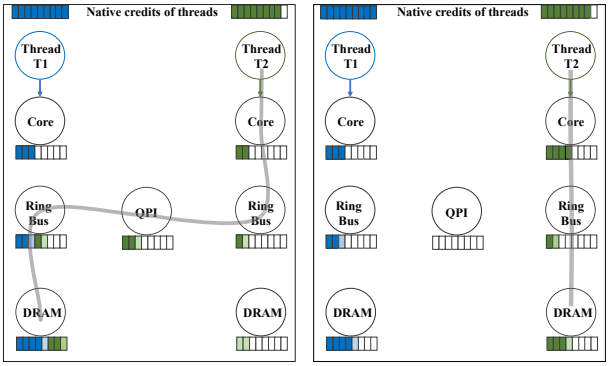

Fig. 4. **The credits of threads.** The blue cells represent the private credits of the T1 thread. Light colored cells represent credits that load the device, excluding demand loads, such as prefetch. The private credits of each thread correspond to a class of customers in the queuing network.

device manages the comprehensive load field and demand load field for each thread. The fraction of the transaction is scaled by the comprehensive load factor and added to the comprehensive load field at the device. On the other hand, the demand load field added up into the demand load field for each thread. First, initial profiled transactions are loaded to obtain some model parameters such as visit ratio and the number of credits.

$$Lc \mathrel{+}= fc * V * F \tag{1}$$

$$Ld \mathrel{+}= fd * V * F \tag{2}$$

Lc = Comprehensive Load at the Device
fc = Comprehensive load factor
Ld = Demand Load at the device of the thread
fd = Demand load factor
V = Visit Frequency of the thread
F = Fraction of the Transaction of thread

F. Total number of credits

The total number of credits refers to all the tokens needed to perform transactions from the load execution unit of cores and the prefetcher. In the closed queuing network model, the visit ratio for devices is proportional to the throughput of the device. We reflected the measured throughput of cores in the visit ratio. That is, if the measured throughput of a core is equal to the evaluated throughput of the core device from the model, the measured throughput and evaluated throughput of others are also equal. To get the appropriate total number of credits, the throughput as model output is tested with a variable number of credits as model input with binary search. The tests are performed iteratively up to the maximum number until the deviation between the measured throughput and the evaluated throughput is the smallest or less than epsilon. Through this binary search style iterated test, the total number of credits is inferred.

$$Deviation = \sum_{i=1}^{\#of Threads} (measured[i] - evaluated[i])^2$$

G. Number of private credits

The number of classes of the credits is equal to the number of cores plus a prefetcher. That is each core has private credits which represent outstanding requests and data dependency result in core stalls. The credits of threads are demonstrated in Fig. 4. In our closed multiclass queuing network model for presented memory subsystem model, a thread represents a class and the number of private credits represents the number of costomers in the class. Our main concern is the demand load transaction issued by the core device. In order to obtain the performance index corresponding to the demand load transaction issued from the a core device, the transactions are tracked. The contribution of the demand load from the mean number of credits belonging to the devices is considered as the number of private credits.

$$Cp = M * Ld / Lc \tag{3}$$

Cp = Number of Private credits
M = Mean number of credits of the device
Ld = Demand Load at the device of the thread
Lc = Comprehensive Load at the Device

VI. Finding the Optimal Thread Arrangement

A. Memory page migration to an appropriate node

To reduce the penalty from remote accesses, pages can be migrated to the most referenced node. References to the page can be identified by the sampled physical page address.

B. Optimal arrangement

In order to avoid the problem in a system with a huge number of processors, threads are clustered as elements of combinations so that the number of combinations is manageable. For threads clustering, we used the cache line sharing-aware method *Tam* as proposed by [6] .

To find the optimal arrangement, all arrangements of the cluster are evaluated by the presented model. Each node constitutes a combination of the cluster. After removing symmetric clusters configurations, we create the next combination for the remaining clusters. Among the results of all arrangements evaluated by the presented model, the arrangement with the smallest maximum stall time among the cores is selected as the optimal arrangement. This is because the maximum stall time among the cores is regarded as the critical path of the entire executions of program.

VII. Evaluation

In this section we compare the performance evaluation between measurements obtained on our Intel machine and the predictions of our model.

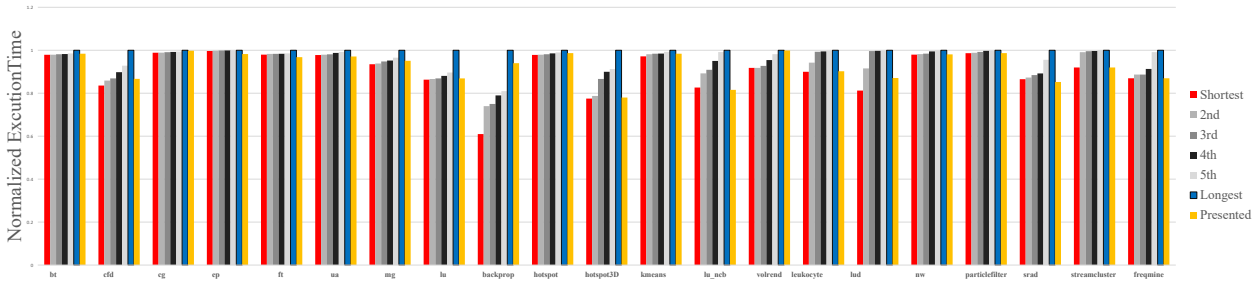

Fig. 5. **Comparison of performance between arrangements.** Each bar represents normalized execution times of the set of six configurations from a set of 256 configurations selected by regular patterns for placements. Yellow bar indicates the normalized execution times of the optimal arrangement identified by our models. The model is able to identify the best thread arrangement from a set of 256 configurations for 20 out of 21 parallel applications from various benchmark suites

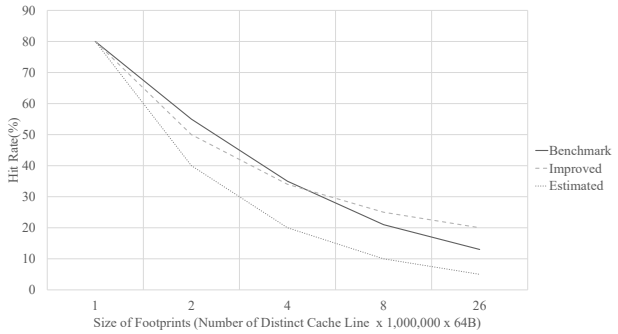

Fig. 6. **Comparison of last level cache hit rate.** The prediction of the improved model partitioned into 16 steps to reflect the LRU policy was more accurate than the default estimated prediction model

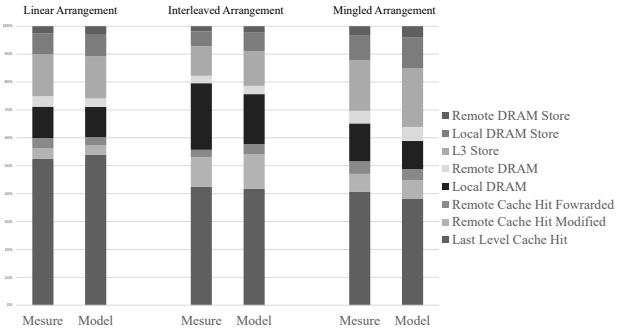

Fig. 7. **Comparison of fractions of transaction types.** The *Remote Cache Hit Modified* fraction regarding invalidation demonstrated the most accurate predictions.

A. Experimental platform

We evaluate our implementation of the closed queueing model using several benchmark suites *NPB, PARSEC, RODINIA*. We run microbenchmarks to measure the accuracy of inference of transactions which is the base of the estimation. A 72-core (4-node) Intel Xeon E7-8870 v3 processor with 512 GiB DRAM is used for the evaluation.

B. Comparison of last level cache hit rate

In the prediction model of the last level cache hit rate using the statistics of the distinct cache line usage of threads, the prediction of the improved model is partitioned into 16 steps to reflect the LRU policy was more accurate than the default estimated prediction model. In particular, Fig. 6 shows that under the improved model that considers the access weights of the cache lines, the LRU policies resulted in higher hits in cache lines with high access weight.

Our analysis illustrates the benefits of the improved model that make it more accurate in estimating the total capacity of the last-level cache by predicting the size of the occupied last-level cache row size. These accurate predictions of the total capacity of the last-level cache allows to accurately estimate the probabilities of the occurrences of transactions for a given thread arrangement onto the nodes. The actual importance of the correct estimation of the probability of last level cache hit transaction is that the last level cache plays a large role in the memory subsystem. Typically, the last level cache hit rate reaches 80%.

C. Comparison of fractions of transactions

We divided the transactions into last level cache hit, remote cache hit(modified line, forwarded line), local DRAM, and remote RAM access. Based on the sampled addresses providing hints to examine the footprints derived from a node for a given arrangement, we can infer the fractions of transactions. Except for the last level cache hits where most requests are processed, we focus on invalidation due to the communication overhead that increases linearly with the number of scalable nodes. This is because invalidation for exclusive ownership of the cash line will result in a penalty for a cache miss, regardless of the capacity of the last level cache. As shown in Fig. 7, the *Remote Cache Hit Modified* fraction regarding invalidation yields the most accurate prediction.

D. Comparison of performance between arrangements

As shown in Fig. 5, our workload model and the abstracted closed queuing model for the memory subsystem show that we can identify the best thread allocation for almost all benchmarks. For the set of the six evaluated configurations,

except for the backprop benchmarks, the prediction of the optimal arrangement by the model and the measurement are identical and achieve the shortest execution time. For the backprop benchmark, there is a significant change in the proportion of prefetch load that affects the number of demand load and leads to a misprediction.

VIII. Conclusion

In this paper, we have evaluated the comprehensive performance of NUMA systems with the goal of finding the optimal thread arrangement of a parallel application by examining the innate demand of the individual threads to the memory subsystem.

We showed that a single profile run with a given thread assignment is sufficient to extract the necessary parameters that allow us to model memory and application performance for differing thread allocations. A memory subsystem model is then used to model the traffic and gives insight into the impact of the trade-off between the central and distributed use of resources.

Our workload model is designed to infer transactions and scales well. The closed queuing network for the memory subsystem allows us to model the performance index of a queuing network consisting of a large number of devices in a short time. On our 72-core 4-node Intel machine, the performance evaluation model requires at most a few milliseconds to make a prediction.

Our analysis illustrates that the predicted performance with respect to a given arrangement of nodes and threads is close to the actual, measured performance. In our evaluation, we show that the model successfully identifies the best thread allocation from a set of six given allocation strategies for 20 out of 21 parallel workloads from different parallel benchmark suites.

Acknowledgments

We thank the anonymous reviewers for their comments and suggestions. This work was supported, in part, by the National Research Foundation of Korea through grants 0536-20210093 and 21A20151113068 (BK21 Plus for Pioneers in Innovative Computing - Dept. of Computer Science and Engineering, SNU), and by the Neural Processing Research Center of the Samsung Advanced Institute of Technology. ICT at Seoul National University provided research facilities for this study.

References

[1] H. Dybdahl and P. Stenstrom, "An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 2–12.

[2] D. P. JL Hennessy, "Computer architecture: A quantitative approach," vol. 5, 2011.

[3] R. E. Matick, T. J. Heller, and M. Ignatowski, "Analytical analysis of finite cache penalty and cycles per instruction of a multiprocessor memory hierarchy using miss rates and queuing theory," *IBM Journal of Research and Development*, vol. 45, no. 6, pp. 819–842, 2001.

[4] D. Thiebaut and H. S. Stone, "Footprints in the cache," *ACM Trans. Comput. Syst.*, vol. 5, no. 4, p. 305–329, Oct. 1987. [Online]. Available: https://doi.org/10.1145/29868.32979

[5] M. Squillante and E. Lazowska, "Using processor-cache affinity information in shared-memory multiprocessor scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, pp. 131–143, 1993.

[6] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: Sharing-aware scheduling on smp-cmp-smt multiprocessors," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 47–58. [Online]. Available: https://doi.org/10.1145/1272996.1273004

[7] R. Lachaize, B. Lepers, and V. Quema, "Memprof: A memory profiler for NUMA multicore systems," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, Jun. 2012, pp. 53–64. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/lachaize

[8] D. L. Willick and D. L. Eager, "An analytic model of multistage interconnection networks," in *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 192–202. [Online]. Available: https://doi.org/10.1145/98457.98758

[9] D. Sorin, V. Pai, S. Adve, M. Vemon, and D. Wood, "Analytic evaluation of shared-memory systems with ilp processors," in *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, 1998, pp. 380–391.

[10] e. a. Baskett, F., "Open, closed and mixed networks of queues with different classes of customers," *JACM, Journal of the ACM*, vol. 22, no. 2, 1975.

[11] K. C. S. E. D. Lazowska. J. Zahorjan, G. S. Graham, "Quantitative system performanceh," 1984.

[12] AMD, "Bios and kernel developer's guide (bkdg) for amd family 15h models 00h-0fh processors," 2013.

[13] W. Yoo, K. Larson, L. Baugh, S. Kim, and R. H. Campbell, "Adp: Automated diagnosis of performance pathologies using hardware events," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 283–294. [Online]. Available: https://doi.org/10.1145/2254756.2254791

[14] Intel, "Intel64 and ia-32 architectures optimization reference manual," 2016.

[15] S. Eranian, "What can performance counters do for memory subsystem analysis?" in *Proceedings of the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness: Held in Conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, ser. MSPC '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 26–30. [Online]. Available: https://doi.org/10.1145/1353522.1353531