Performance Modeling of Parallel Loops on Multi-Socket Platforms using Queueing Systems

Younghyun Cho, Surim Oh, Bernhard Egger

Abstract—Predicting the performance of parallel loops on modern shared-memory multi-socket multi-core systems in dependence of the allocated resources is an important means to achieve better system utilization. Previous prediction techniques are tied to specific architectures and do not allow for purely online performance predictions without requiring an offline analysis of the parallel program. This paper presents a practical approach based on queueing theory to model the performance of parallel programs in dependence of the number of allocated core resources. Based on the key insight that scalability of scientific parallel loops is limited by memory performance, a hierarchically constructed *M/M/1/N/N* queue system is used to analytically compute the response time at the different congestion points in the memory system of modern NUMA architectures. After automatically tuning the model to a specific architecture by executing a number of micro-benchmarks, the required parameter values are obtained at runtime from hardware performance counters present in modern commodity AMD and Intel processors. Evaluated with 24 OpenMP parallel loops on a 64-core AMD and a 72-core Intel multi-socket platform, the presented queueing system is able to accurately predict the speedup of parallel loops with a mean absolute percentage error of 8.3% on the AMD system and 6.7% on the Intel platform.

Index Terms—Performance modeling, parallel loop, queueing system, multi-socket system, OpenMP, NUMA.

1 Introduction

Parallel loops such as OpenMP's *parallel for* [1] are the basic parallel programming construct on shared-memory platforms. Understanding the performance of parallel loops is important for optimizations as these loops dominate the execution time of many scientific applications. Such parallel loops are usually executed with a configurable number of worker threads managed by a runtime system [2], [3], [4] to maximize the performance or to meet a certain performance goal. Modeling performance of parallel loops in dependence of the number of threads therefore has been an important research issue.

One approach to modeling performance of parallel programs employs queueing models. Based on stochastic processes, queueing models are analytical tools to evaluate the performance of queueing systems such as the mean waiting time, the queue length, and the server utilization [5]. Previous work [6], [7], [8] predicts the performance scalability of parallel programs by computing the mean response time of memory requests for a varying number of threads. These approaches regard the threads of parallel programs as queueing customers accessing memory system resources, and the memory system as the queueing server. Queueing models are not only computationally efficient thanks to their closed-form expressions, but also allow predicting the speedup of parallel programs and provide insights into the response time and utilization of the memory system.

- *Younghyun Cho and Bernhard Egger✉ are with the School of Computer Science and Engineering, Seoul National University, Seoul, Korea. E-mail: younghyun, bernhard@csap.snu.ac.kr.*
- *Surim Oh was with the School of Computer Science and Engineering, Seoul National University. She is now with SAP Labs Korea, Seoul, Korea. She contributed this paper when she was in Seoul National University. E-mail: surim.oh@sap.com*

Applying queueing models to modern multi-core systems in practice, however, remains a challenge. Large shared memory systems, called multi-socket multi-core systems, comprise multiple processor sockets and memory controllers connected by an interconnection network. Memory operations from cores thus contend for both the memory controllers and the interconnection links. Such architectures require a proper queueing network to model the different contention points. Moreover, memory systems act differently on read and write memory operations and perform hardware-level optimizations such as data sharing and prefetching. The effectiveness of such optimizations depends on the parallel program and the number of worker threads. Consequently, memory systems provide different service rates that depend on the workload. Existing techniques [6], [7], [8] employ simple queueing systems that do not properly consider the different contention points in the memory system and ignore the effects of hardware optimizations. These simplifications render existing techniques ineffective on modern hardware architectures.

This paper presents a practical approach to model performance of parallel for loops without loop-carried dependencies on multi-socket multi-core systems using queueing systems. First, runs of OpenMP parallel loops on real systems confirm that the *M/M/1/N/N* queueing model [5] is adequate to model parallel loops on multi-cores systems. The architecture of multi-socket systems is reflected by a hierarchically constructed *M/M/1/N/N* queueing system that is able to compute the mean response time of memory requests at each memory controller and each interconnection link. To deal with the varying memory system performance in the presence of hardware optimizations, the service rates of memory controllers and interconnection links are computed based on the ratio between memory read, write, and prefetch operations of a given workload. The presented ap-

proach can be easily applied to different platforms because all information required to compute the parameter values of the queueing systems is obtained from existing hardware performance counters on AMD and Intel systems.

The queueing system is used to construct a speedup model that is able to predict the performance scalability of parallel loops on multi-socket systems. An evaluation with 24 OpenMP parallel loops shows that, on average, the model achieves a mean absolute percentage error of 8.3% on a 64-core AMD and 6.7% on a 72-core Intel platform. The results demonstrate that the presented queueing system is able to provide accurate information about the performance of memory controllers and interconnection links in multi-socket multi-core systems.

To summarize, this paper makes the following contributions.

- A summary of the key assumptions to apply queueing systems to model parallel loops on multi-socket systems, and an experimental study that shows how the targeted parallel loops can be modeled using *M/M/1/N/N* queueing systems (Section 3).
- A methodology to model memory system performance on multi-socket multi-core platforms using a hierarchical queueing system (Section 4).
- A speedup model that is able to predict the speedup of OpenMP parallel loops based on the queueing system (Section 5).
- An evaluation of the presented speedup model for 24 OpenMP parallel loops on an AMD and an Intel multi-socket multi-core platform (Section 6).

In addition, Section 2 introduces the necessary background on OpenMP parallel loops and queueing systems. Section 7 discusses the limitations of the presented approach, and Section 8 compares the presented approach with related work. Section 9, finally, concludes this paper.

2 Background

2.1 Parallel Loops in OpenMP

OpenMP [1] is the most widely used parallel programming model for shared-memory systems. Parallel loops annotated with the `parallel for` pragma are the basic mechanism to initiate parallelism in OpenMP applications. In an OpenMP parallel loop, the outermost loop iterations represent the smallest parallel unit of work.

OpenMP supports three loop scheduling methods: static, dynamic, and guided. Programmers can select a scheduling discipline by annotating the specific method to the `parallel for` pragma. Static scheduling, selected by `schedule(static)`, divides and assigns the loop iterations of a parallel loop equally to the worker threads. This policy benefits from a small dispatch overhead but may suffer from load imbalance. With dynamic scheduling, selected by `schedule(dynamic)`, loop iterations are assigned to the worker threads at runtime; this process is illustrated by Fig. 1 for the GNU OpenMP (GOMP) runtime system [9]. Each thread repeatedly fetches and executes a fixed number of loop iterations from the global shared `work_share` data structure until there is no more work. Guided scheduling,

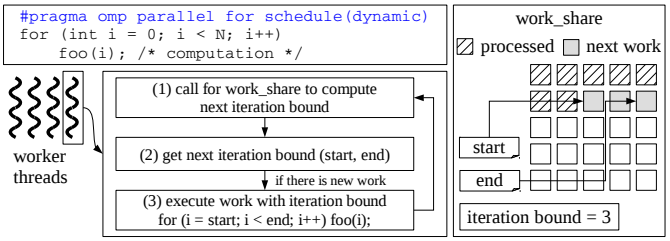


Fig. 1. Dynamic loop scheduling in the GOMP runtime system.

annotated by `schedule(guided)`, operates similar to dynamic scheduling but dynamically adjusts the number of loop iterations assigned to a thread. Li's guided scheduling [10], for example, assigns $\lceil items/2N \rceil$ loop iterations where $items$ represents the number of remaining loop iterations and N stands for the number of worker threads.

2.2 Queueing Models

Queueing models that compute the waiting time of queueing systems using stochastic processes have often been used for operations research in computer science such as designing system architectures or developing scheduling policies [11]. They are also well-suited to analytically model the performance of shared resources such as memory controllers [6], [8], [12] and network switches [13]. The focus of this work is on modeling the performance of the shared memory system. In the following, we briefly discuss two well-known queueing models, the *M/M/1* and the *M/M/1/N/N* model. For details about queueing models the interested reader is referred to [5], [14].

2.2.1 The M/M/1 Model

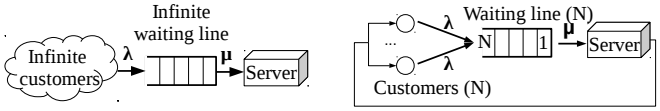
The *M/M/1* model is the simplest and most popular queueing model. An *M/M/1* queueing system, illustrated in Fig. 2 (a), considers requests from an infinite number of customers and one single server. The arrivals of the requests follow a Poisson distribution, and the server has an exponential service time. The requests are served in First-In-First-Out (FIFO) order. For an arrival rate λ and a service rate μ, $\mu > \lambda$, the mean waiting time r is given by Little's Law.

$$r = \frac{1}{\mu - \lambda} \tag{1}$$

Previous research [6], [7] often employed the *M/M/1* queueing model to model memory performance on multi-cores where cores are considered to be queueing customers. In multi-core systems with a finite number of cores, however, the presence of more or fewer cores can have a strong effect on the distribution of memory requests which calls for a queueing model for a finite number of customers.

2.2.2 The M/M/1/N/N Model

For a finite number of customers, the *M/M/1/N/N* model, also known as the "machine repair problem", can be applied. It consists of N customers, a waiting line having N entries with FIFO discipline, and one server, as shown in Fig. 2 (b). The requests of the customers follow a Poisson distribution, and the server has an exponential service time.



(a) The *M/M/1* queue (b) The *M/M/1/N/N* queue

Fig. 2. The queueing systems. In the *M/M/1* system (a), λ represents the server request rate from infinite queueing customers, and in the *M/M/1/N/N* system, λ represents the mean server request rate per customer. In both systems, μ represents the server's mean service rate.

In the *M/M/1/N/N* model, once a request has been issued from a customer, the customer does not send a new request until the previous request has been served. Given an arrival rate λ per customer and a service rate μ, the mean waiting time r is given by Equation 2.

$$r = \frac{1}{\mu}(\frac{N}{U_s} - \frac{\mu}{\lambda}) \tag{2}$$

U_s, representing the server's utilization, is computed by

$$U_s = 1 - \left(\sum_{k=0}^{N} \frac{N!}{(N-k)!} (\frac{\lambda}{\mu})^k \right)^{-1} \tag{3}$$

In this work, the *M/M/1/N/N* model is applied to model the mean memory response time on multi-socket systems.

3 Insights of Parallel Loop Performance

This section justifies the application of the *M/M/1/N/N* queueing model to predict the performance of scientific parallel loops on modern out-of-order NUMA systems through experiments on actual architectures.

3.1 Queueing Models and Multi-Socket Systems

Multi-socket systems contain multiple processor sockets and memory controllers to provide the necessary memory bandwidth to a large number of cores. A *node* consists of a CPU node that itself is composed of a group of CPU cores and an attached memory node. The individual nodes are connected by an interconnection network such as AMD's Hyper Transport (HT) [15] or Intel's Quick Path Interconnect (QPI) [16]. Memory requests of a CPU node are sent to a buffer and served by the interconnection network that enables each CPU node to access all memory nodes. Such architectures typically exhibit Non-Uniform Memory Access (NUMA) characteristics because of the varying access latencies from the cores to the different memory controllers.

Queueing models require an even, Poisson-distributed request distribution from all customers. In addition, customers wait for their requests to complete before issuing a new request. Even though these requirements are not satisfied in general by multi-socket multi-core systems, the following key observations allow us to apply *M/M/1/N/N* queueing systems to such architectures.

- The presented approach models performance of scientific parallel loops where memory wait time is the major limiting factor of scalability. The memory access pattern of common workloads satisfies the requirement of even and Poisson-distributed request distributions as demonstrated in Section 3.3.2.
- Processor cores execute instructions out-of-order and can issue several memory requests. In addition, requests can be reordered by caches and memory controllers [17], [18]. These properties do not satisfy the requirements of the queueing models, however, the presented approach models the average mean memory request time for a large number of requests in the steady state. In this case, reordering or parallel individual requests do not invalidate the model.
- Each memory operation is served by an interconnection link and a memory controller. A memory controller can receives requests from all CPU nodes; the number of inputs of an interconnection link depends on the architecture.
- Multiple queueing systems are used to model the performance on the multiple contention points. First, we use separate queueing systems to model memory response time at each interconnection link and each memory controller. Additionally, another queueing system is used to model the thread stall time on each CPU core. Section 4 details this approach.

TABLE 1
Selected parallel loops.

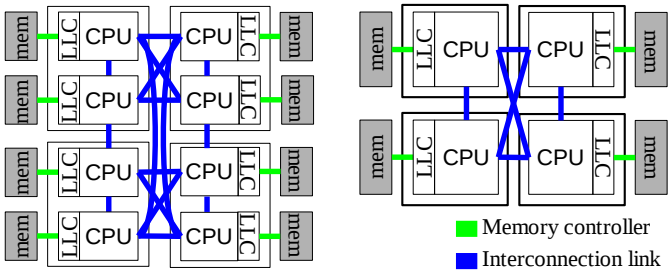
Loop	App	Input size	Loop	App	Input size
x_solve	BT	class D	rhs5	SP	class D
y_solve	BT	class D	x_solve	SP	class D
z_solve	BT	class D	y_solve	SP	class D
add	BT	class D	z_solve	SP	class D
conj_grad2	CG	class D	txinvr	SP	class D
cffts1	FT	class C	tzetar	SP	class D
cffts2	FT	class C	rprj3	MG	class D
cffts3	FT	class C	psinv	MG	class D
rhs1	SP	class D	interp1	MG	class D
rhs2	SP	class D	resid	MG	class D
rhs3	SP	class D	main	BS	native
rhs4	SP	class D	tiling1	FM	native

3.2 Parallel Loops

We mainly target scientific OpenMP parallel loops that are used in HPC and big data computing. Table 1 shows the parallel loops used for performance modeling and evaluation throughout this paper. The loops were obtained from the NAS parallel benchmark suite (NPB) [19], [20] containing HPC workloads and two OpenMP applications from the Parsec benchmark suite [21], Blackscholes (*BS*) and Freqmine (*FM*). We did not evaluate loops that perform data initialization because such loops are usually executed only once to activate the placement of the data under a given NUMA allocation policy. In total, 24 different parallel loops are selected from the seven parallel applications.

The assumptions of the presented model and the justifications for the selected parallel loops are as follows:

- Memory requests of parallel loops follow a Poisson distribution, and memory service times are exponential. These assumptions are a requirement of the *M/M/1/N/N* model and verified based on experiments in Sections 3.3.2 and 3.3.4.



(a) 64-core AMD platform (b) 72-core Intel platform

Fig. 3. Block diagram of the target multi-socket multi-core platforms.

- Synchronization overhead is not considered. In other words, loops have no loop carried dependencies and do not suffer from load imbalance. Most loops of NPB applications (Table 1) do not have dependencies. Experiments in Section 3.3.5 show that most of the targeted loops exhibit a good load balance.
- Similarly, atomic operations or critical sections are not considered. Modeling the performance of atomic operations and critical sections is difficult in practice. For example, the number of issued atomic operations to obtain a lock is not deterministic. Moreover, such operations are rarely used in data intensive loops.
- Parallel loops are dynamically scheduled (Section 2.1) because this policy allows runtime systems to dynamically adjust the number of threads [2], [3]. Section 7 discusses the effects of other scheduling schemes on performance modeling.

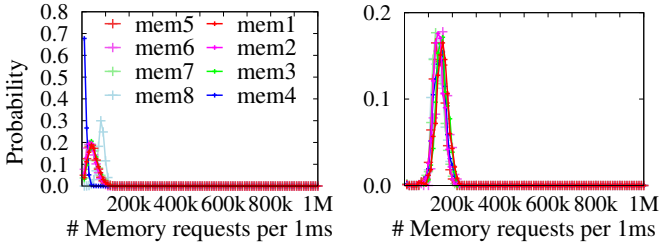
3.3 Performance Analysis

3.3.1 Evaluation Environment

This work is verified and evaluated on two multi-socket platforms, a 64-core AMD Opteron and a 72-core Intel Xeon system. The AMD platform, shown in Fig. 3 (a), comprises a total of eight CPU nodes in four physical processor packages (AMD Opteron 6380 [22]) and 128 GB of memory. Each CPU node contains eight computing cores that share a last-level cache (LLC) of 12 MB. The processor nodes are connected by AMD's Hyper Transport [15] with a maximum hop distance of two. The Intel system, shown in Fig. 3 (b), has four Intel Xeon E7-8870 v3 processors [23] each consisting of 18 cores sharing a 45 MB LLC. The system is equipped with 512 GB of memory. Each processor represents a CPU node, the four nodes are connected with Intel's QPI [16]. All benchmarks are run with the *interleaved* NUMA memory allocation policy.

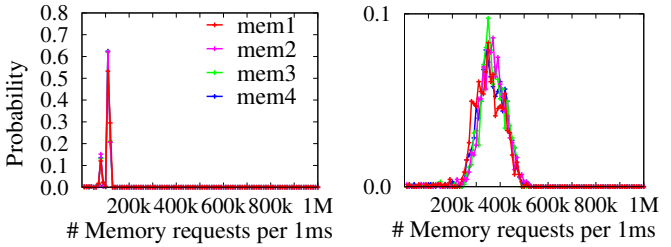
3.3.2 Memory Access Distribution

The assumption of the *M/M/1/N/N* model that memory accesses from worker threads follow a Poisson distribution is verified by measuring the number of memory requests over a fixed interval on the AMD and the Intel platform. The collected numbers of memory requests at each memory node for the entire run of the parallel loop are plotted in Fig. 4 and Fig. 5 using a probability mass function (PMF). The figures show that the vast majority of memory requests per time is distributed around the expected value, and



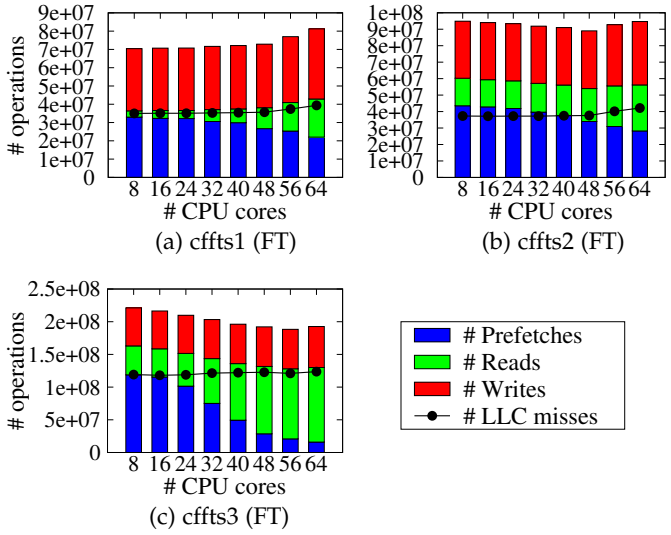
(a) x_solve (BT) (b) x_solve (SP)

Fig. 4. PMF of the number of memory requests per time for each memory node on the 64-core AMD platform.



(a) x_solve (BT) (b) x_solve (SP)

Fig. 5. PMF of the number of memory requests per time for each memory node on the 72-core Intel platform.



(a) cffts1 (FT) (b) cffts2 (FT)

(c) cffts3 (FT)

Fig. 6. Number of memory operations of parallel loops for a varying number of worker threads on the AMD platform.

the variance increases with a higher expected value. In addition, all memory nodes exhibit the Poisson property. For the sake of simplicity, the figures present the results for only the *x_solve* loops of *BT* and *SP*. The supplementary materials contain the PMF of all targeted parallel loops on both architectures as well as the results of the two-sample Kolmogorov-Smirnov (KS) test [24] confirming that the majority of the loops follows a Poisson distribution.

3.3.3 Memory Access Pattern

Modern memory systems perform optimizations such as memory prefetch operations that can cause a variation in the memory access pattern. Fig. 6 shows the number of memory operations collected from hardware performance

counters for the three parallel loops *cffts1–3* of *FT* with a varying number of worker threads.

The memory access pattern varies for different workloads and the number of worker threads. For example, in Fig. 6 (c), the total number of memory requests in *cffts3* decreases with an increasing number of threads because the loop can benefit from data sharing. For *cffts1* in Fig. 6 (a), on the other hand, the number of memory operations increases for a larger number of threads. Therefore, an *M/M/1/N/N* queueing system needs to use a changing memory request rate when modeling the memory response time for a varying number of threads. In addition, different programs have different ratios between the read, write, and prefetch operations. The following section analyzes this effect on the service rate of the memory system.

3.3.4 Memory Service Rate

On multi-socket systems, the requested data is transmitted through an interconnection link and a memory controller. We measure the service rate μ_j of an arbitrary memory controller j and the data transfer rate δ_{ij} of the interconnection link that connects CPU node i with memory controller j for the four synthetic workloads *StreamWrite*, *StreamLoad*, *StreamCopy*, and *StreamAdd* from the Stream benchmark suite [25]. The following code shows the *StreamWrite* workload that writes a scalar value to the elements of an array.

```
1:  for (int i=0; i<stride; i++)
2:    for (int j=i; j<arr_size; j+=stride)
3:      A[j] = scalar;
```

The other workloads execute different types of operations in line 3. *StreamLoad* executes `sum+=A[j]` and thus generates only memory read operations. *StreamCopy* executes `A[j]=B[j]`, generating one memory write for `A[j]` and two memory reads for `A[j]` and `B[j]`. Last, *StreamAdd*'s code `A[j]=B[j]+C[j]` consists of three memory read and one memory write operations.

Fig. 7 shows the number of memory operations of the synthetic workloads for varying `stride` values. The values are set such that the workload is completely memory bound. Similar to the observation from Fig. 6, different synthetic workloads in Fig. 7 have different ratios between memory read, write, and prefetch operations. To compute μ_j, the data of the arrays `A`, `B`, and `C` is allocated to memory node j. The workload is executed on one core in CPU node j using one thread, and the total runtime of the workload, denoted *total execution time*$_{jj}$, is measured. Since all memory accesses are served by the local memory node without passing through other interconnection links, the mean service rate of memory controller j can be computed by

$$\mu_j = \frac{\#\ \textit{total memory operations}}{\textit{total execution time}_{jj}} \tag{4}$$

To compute the service rate of an interconnection link δ_{ij}, the *total execution time*$_{ij}$ is measured by executing the workload on a core in CPU node i and the data located in memory node j. The execution time of such an allocation includes the data transfer time through the interconnection link and the memory controller. The data transfer rate of the interconnection link is computed as follows.

$$\delta_{ij} = \frac{\#\ \textit{total memory operations}}{(\textit{total execution time}_{ij} - \textit{total execution time}_{jj})} \tag{5}$$

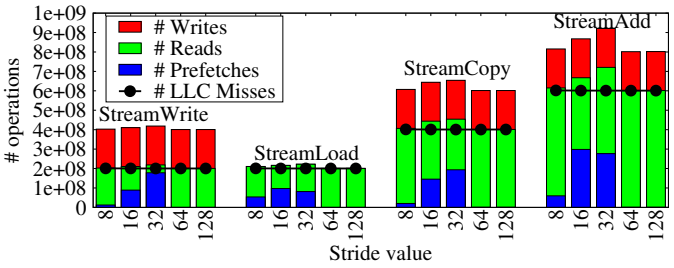


Fig. 7. Number of memory operations of synthetic workloads (using one CPU thread) with different stride values on the AMD platform.

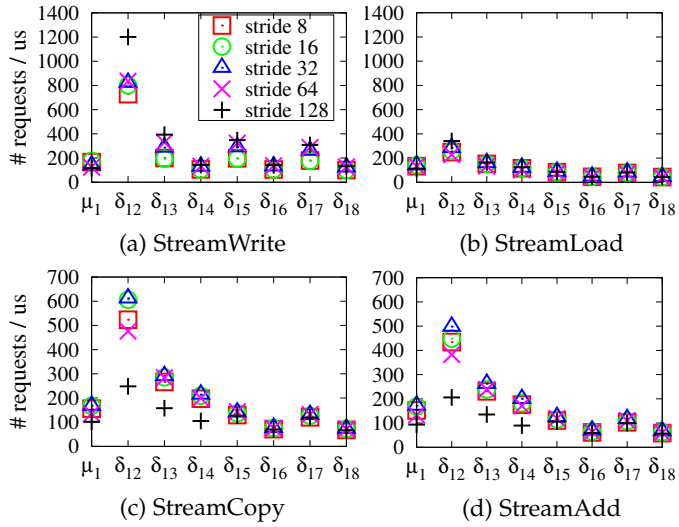


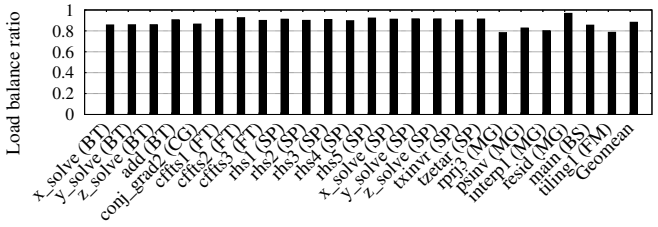
Fig. 8. Measured service rates for the synthetic workloads on the AMD platform for varying stride values. μ_1 represents the service rate of memory controller 1, and $\delta_{1\ 2-8}$ represents the service rate of the interconnection links connecting CPU node 1 and memory controllers 2–8.

Fig. 8 shows the measured service rates of memory controller 1 and the interconnection links between CPU node 1 and memory nodes 2–8 on the AMD system for the four synthetic workloads from Fig. 7. We observe that the memory service rate depends on the workload. For example, comparing *StreamWrite* (Fig. 8 (a)) and *StreamLoad* (Fig. 8 (b)) reveals that *StreamWrite* tends to have higher memory service rates than *StreamLoad*, suggesting that a higher ratio of memory write operations causes a higher service rate. In addition, in *StreamCopy* (Fig. 8 (c)) and *StreamAdd* (Fig. 8 (d)), the service rates tend to be higher with a stride value of 32 because there are a larger number of memory prefetch operations as visible in Fig. 7. The experiments demonstrate that it is necessary to consider the memory access pattern of the given workload to compute the memory service rate.

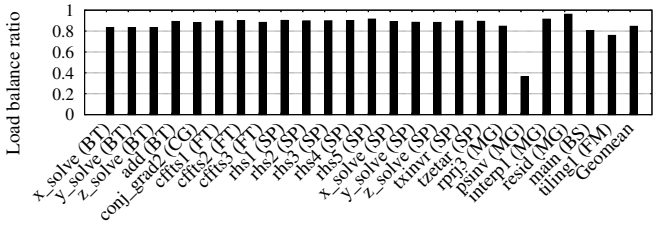
The *M/M/1/N/N* model assumes that the server exhibits exponential service times. Similar to the analysis of memory accesses in Section 3.3.2, this assumption is justified using the KS test for the synthetic workloads. The details are provided in the supplementary materials.

3.3.5 Synchronization Overhead

Parallel loops have implicit barriers at the end of the loops that can affect the performance of the parallel loops if the load is unbalanced. Here, we investigate the effect of this implicit barrier by measuring the load balance ratio.



(a) 64-core AMD platform.



(b) 72-core Intel platform.

Fig. 9. Measured load balancing ratio for the parallel loops.

The load balance ratio is computed by comparing the turnaround time of the two worker threads that take the longest ($t_{longest}$) and the shortest ($t_{shortest}$) to complete their execution (*Load balance ratio* $= t_{shortest}/t_{longest}$).

Fig. 9 shows the measured load balance ratio for the 24 parallel loops on the AMD and the Intel platform. As shown in the figure, many loops have a high load balance ratio (larger than 0.9). This implies that, for many parallel loops the overhead from load imbalance is limited to only a fraction of the overall performance. Based on this observation, such overhead is not modeled in this work. Several loops (*rprj3*, *psinv*, and *interp1*) of the *MG* application, however, exhibit a low load balance ratio. The *MG* application is based on an unstructured grid where the inner loops have different loop iteration bounds. The *tiling1* loop contains an inner loop with varying iteration counts and also conditional branches that cause this load imbalance.

3.3.6 Summary

The performance analysis shows that the *M/M/1/N/N* queueing model is adequate to model memory requests of parallel loops. For the majority of loops, the distribution of the memory accesses exhibits a Poisson distribution, and the limited amount of synchronizations during the execution of parallel work units allows us to focus on memory system performance as the limiting factor of program scalability. The analysis, however, also shows that there are challenges to use a queueing model when computing the mean memory request rate and the memory service rate for a varying number of worker threads. These parameter values need to be carefully computed for accurate performance modeling.

4 Queueing Systems for Multi-Socket Architectures

This section shows how to employ the *M/M/1/N/N* queueing model to model memory performance on multi-socket multi-core architectures.

4.1 Hierarchical Queueing Systems

The presented approach employs different queueing systems to model the memory response time of a NUMA multi-socket system. The response time of a memory read request observed by an individual CPU core is composed of the service time of the LLC, the interconnection link, and the memory controller. The architectural contention points are modeled by individual queueing models for each memory controller, each interconnection link, and each last-level cache. A multi-socket system with two CPU nodes and two memory nodes as shown in Figure 10 is used for the explanations. Each CPU node has four cores and an LLC. The shaded boxes in the figure depict the contention points observed by a core in CPU node 1 issuing a memory read request to memory node 2. Each contention point is modeled by an *M/M/1/N/N* queueing system. Memory wait time manifests in the form of stalled threads waiting for LLC read misses to complete. This stall time is modeled by the queueing system illustrated in Fig. 10 (c). The contention at the memory controller and the interconnection network is considered by the queueing systems shown in Fig. 10 (a) and (b). In these queueing systems, memory requests are served for each CPU node via an interconnection network, and the queueing systems model the response time for a varying number of CPU nodes. Unlike the model for the LLC wait time that only considers memory read operations, the contention models at the interconnection links and the memory controllers also consider the effect of memory write and prefetch operations. Fig. 11 depicts these queueing systems. The input parameters of the queueing systems and the modeled performance are described in Table 2 and Table 3, respectively. Details of each model are presented in the following sections.

4.1.1 Queueing System for Memory Controllers

Fig. 11 (a) shows the queueing system for memory controller 2 (M_2) of the two-node system from Fig. 10. There are two queueing customers, CPU node 1 and 2 with a memory request rate (*MRR*) to memory node 2 of MRR_{12} and MRR_{22}, respectively (refer to Table 2). λ_{M_2} in Equation 2 is given by

$$\lambda_{M_2} = (MRR_{12} + MRR_{22})/2$$

where MRR_{12} and MRR_{22} represent the memory request rate from CPU node 1 and 2, respectively. With the request rate λ_{M_2} and the memory service rate μ_2 of memory controller 2, Equation 2 yields MRT_{M_2}, the mean response time of memory controller 2.

This approach can be generalized for an arbitrary multi-socket system. For a system with N nodes, a memory controller j is considered a queueing server that serves the resources of DRAM chips with a mean service rate of μ_j, and CPU node i is considered a queueing customer that accesses the server with a mean request rate of MRR_{ij}. Using the average mean request rates of all CPU nodes to memory node j

$$\lambda_{M_j} = \frac{\sum_{i=1}^{N} MRR_{ij}}{N} \quad (6)$$

and the service rate μ_j, Equation 2 computes the mean response time of memory controller j, MRT_{M_j}.

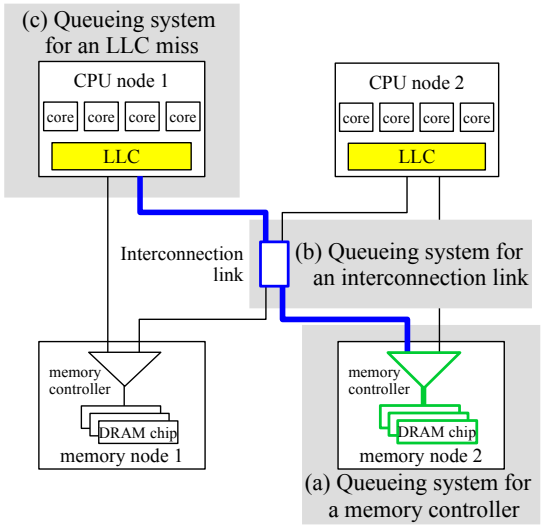


Fig. 10. A two-socket multi-core system and the data path for an LLC miss of CPU node 1 to be served by memory node 2.

4.1.2 Queueing System for Interconnection Links

Contention at interconnection links is modeled by a separate queueing system. In a fully-connected network such as Intel's QPI, contention cannot occur at the interconnection links and no modeling is required. Architectures such as AMD's HT share interconnection links whose response time can be modeled as follows. The interconnection link, as shown in Fig. 10, serves requests from CPU node 1 to memory node 2 and from CPU node 2 to memory node 1. The queueing system, shown in Fig. 11 (b), treats CPU nodes 1 and 2 as customers to obtain the link's request rate $\lambda_{L_{12}} = (MRR_{12} + MRR_{21})/2$. Equation 2 is applied to compute the the mean response time $MRT_{L_{12}}$ of interconnection link L_{12} with the mean transfer rate δ_{12}.

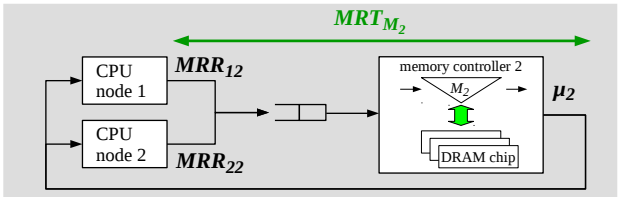
In general, for an interconnection link L_{ij} connecting CPU node i with memory node j at a service rate δ_{ij}, all memory request rates from all CPU nodes that are served by interconnection link L_{ij} need to be considered. The average

TABLE 2
Input parameters of the queueing systems.

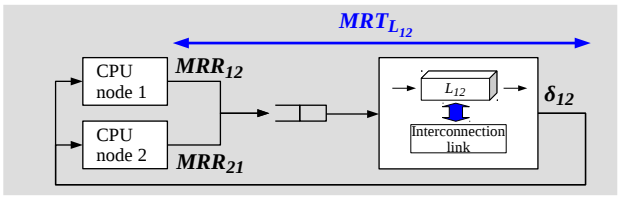
MRR_{ij}	mean memory request rate from CPU node i to memory node j; it considers all read, write, and prefetch memory operations
LLC_{kij}	mean LLC miss rate from CPU core k in CPU node i to memory node j; it considers only read LLC misses
μ_j	mean service rate of memory controller j
δ_{ij}	mean data transfer rate of an interconnection link connecting CPU node i with memory controller j

TABLE 3
Modeled performance information from the queueing systems.

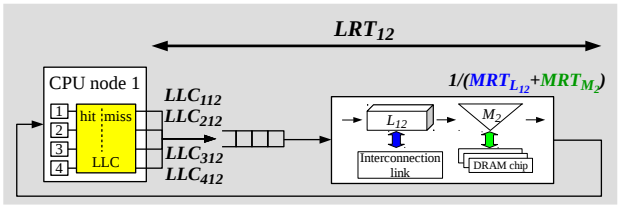
MRT_{M_j}	mean response time of memory requests at memory controller j
$MRT_{L_{ij}}$	mean response time of memory requests at interconnection link connecting CPU node i with memory node j
TRT_{ij}	total mean response time for memory requests from CPU node i to be served by memory node j
LRT_{ij}	mean response time for LLC misses from CPU node i to be served by memory node j



(a) Modeling MRT_{M_2} the mean response time of memory controller 2



(b) Modeling $MRT_{L_{12}}$ the mean response time of the interconnection link connecting CPU node 1 and memory node 2 (L_{12})



(c) Modeling LRT_{12} the mean response time of LLC misses from CPU node 1 that are served by memory node 2

Fig. 11. The hierarchical queueing systems for the data path of Fig. 10.

of the memory request rates $\lambda_{L_{ij}}$ is computed as follows.

$$\lambda_{L_{ij}} = \frac{\sum_{l=1}^{N} \sum_{k \in LSet_{ij}^{l}} MRR_{lk}}{N} \quad (7)$$

where $LSet_{ij}^{l}$ is the set of memory controllers accessed from CPU node l passing through link L_{ij}. These sets are constructed according to the interconnection topology of the target architecture. Using Equation 2, we can compute the mean response time of the interconnection link $MRT_{L_{ij}}$.

4.1.3 Queueing System for LLC Misses

The queueing systems from the preceding two sections compute the mean response time of each memory controller (MRT_{M_j}) and each interconnection link ($MRT_{L_{ij}}$). For a memory request from CPU node i to be served by memory node j, the total mean response time TRT_{ij} is given by

$$TRT_{ij} = MRT_{M_j} + MRT_{L_{ij}} \quad (8)$$

This response time, however, is not sufficient to model the performance of parallel threads. The insight is that cores (i.e., threads) are stalled only for memory read requests occurring from LLC misses. In other words, the threads keep executing while memory write operations or prefetch operations are being served. It is therefore necessary to compute the response time of LLC misses that stall a thread's execution. Fig. 11 (c) shows the queueing system to model the response time for an LLC miss from CPU node 1 handled by memory node 2. All cores within the same CPU node constitute the queueing customers. Assuming a crossbar switch, a CPU node's LLC misses that access the same memory node are served in FIFO order while accesses to

different memory nodes can be processed simultaneously. For the input request rate, the LLC miss rate per core is considered, where LLC_{k12} represents the LLC miss rate for memory node 2 from core k in CPU node 1. The service rate of this queueing system is computed as $1/TRT_{12}$, that means an LLC miss requires services from both the memory controller and the interconnection link. Then, the queueing model computes the mean response time LRT_{12} (Table 3). The mean value of the response times obtained from this queueing system represents the mean thread stall time for LLC misses.

The LLC miss response time LRT_{ij} can be computed for an arbitrary CPU and memory node i and j by replacing 1 and 2 with i and j, respectively. This LLC miss response time is used to compute the performance scalability of parallel loops in Section 5.

4.2 Computing the Parameter Values

4.2.1 Performance Counters

To compute the parameter values of the queueing systems, the number of memory operations at each memory controller and the number of LLC misses at each CPU node are collected. AMD's NorthBridge [26] and Intel's uncore events [27] provide the necessary performance counters. Linux's *perf* interface is used to query the performance counters. The `Memory Controller Requests (NBPMCx1F0)` counter measures the number of memory operations at each memory controller, and `L3 Cache Misses (NBPMCx4E1)` counts the number of LLC misses. Similarly, on the Intel platform, we use `UNC_H_IMC_WRITES/READS` to measure the number of memory operations and `OFFCORE_RESPONSE:L3_MISS` to count the number of LLC misses for each node.

As outlined in Table 2, the presented queueing systems require the parameters MRR_{ij}, LLC_{kij}, μ_j, and δ_{ij}. The following section discusses the computation of the parameter values from the performance counters obtained from a profiling run for a given number of worker threads.

4.2.2 Memory Request Rate and LLC Miss Rate

The value of MRR_{ij}, referring to the number of memory requests per time in the steady state, is computed as follows.

$$MRR_{ij} = \frac{\#\ Requests_{ij}}{CPU\ Time} \tag{9}$$

where $\#\ Requests_{ij}$ is the number of memory requests issued from CPU node i to memory node j. Since $\#\ Requests_{ij}$ is collected in the steady state of a workload, it already includes the effects of different cache write miss policies. *CPU Time* denotes the execution time of threads excluding the stall times caused by the LLC misses. Threads are assumed to have the same execution time with perfect load balance.

Similarly, the LLC miss rate is computed as the number of LLC misses per time as follows.

$$LLC_{kij} = \frac{\#\ LLC\ Miss_{kij}}{CPU\ Time} \tag{10}$$

where $\#\ LLC\ Miss_{kij}$ is the number of LLC misses issued from core k in CPU node i and served by memory node j.

Measuring *CPU Time* is not trivial because existing processors can measure only the total runtime, *Total Time*, that includes the memory response times. *Total Time* is defined as *CPU Time* plus the response times for LLC misses as follows.

$$Total\ Time = CPU\ Time + \sum_{j=1}^{N}\left(\sum_{k=1}^{C} \#\ LLC\ Miss_{kij} \cdot LRT_{ij}\right) \tag{11}$$

where C represents the number of cores in a CPU node, and LRT_{ij} is computed from the queueing system given in Section 4.1.3. Solving Equation 11 for *CPU Time* is not trivial because the queueing system for LRT_{ij} requires *CPU Time* to compute the input parameters of MRR_{ij} and LLC_{kij}. To compute *CPU Time* with a reasonable overhead, we use an iterative method using Equation 12.

$$CPU\ Time^{k+1} = Total\ Time - \sum_{j=1}^{N}\left(\sum_{k=1}^{C} \#\ LLC\ Miss_{kij} \cdot LRT_{ij}^{k}\right) \tag{12}$$

LRT_{ij}^{k} and $CPU\ Time^{k+1}$ are iteratively computed based on $CPU\ Time^{k}$. Since $Total\ Time \geq CPU\ Time$, the initial input of $CPU\ Time^{0}$ is set to *Total Time*. Five iterations were empirically determined to be sufficient on both architectures.

The method presented in this section computes the parameter values from the measured performance counter values. However, as explained in Section 3.3.3, the memory request rate changes for a varying number of threads. A practical profiling method that considers varying memory request rates in dependence of thread counts is discussed in Section 5.2.2.

4.2.3 Memory Service Rate

The mean service rate, *MSR*, μ_j for memory controller j and δ_{ij} for interconnection link L_{ij}, is computed from the mean service time *MST*, $MSR = 1/MST$. As discussed in Section 3, the service rate of the memory resources varies depending on the ratio between memory operations.

A linear equation is used to compute the mean service time for each memory controller and interconnection link. For example, Equation 13 computes the mean service time for memory controller j.

$$MST_{\mu_j} = \alpha_{\mu_j} \cdot \frac{\#\ Prefetches}{\#\ Requests} + \beta_{\mu_j} \cdot \frac{\#\ Reads}{\#\ Requests} + \gamma_{\mu_j} \cdot \frac{\#\ Writes}{\#\ Requests} \tag{13}$$

To compute the coefficient values of α_{μ_j}, β_{μ_j}, and γ_{μ_j}, the four synthetic workloads from Section 3.3.4 are executed with varying stride values (8, 16, 32, 64, 128) and the MST_{μ_j} is measured for each configuration. The coefficient values are obtained by applying linear regression to the measured MST_{μ_j} values. This procedure is performed for each interconnection link L_{ij} to calculate $MST_{L_{ij}}$. Some architecture may not support collecting the number of prefetches of the L3 caches and the counts for read operations include prefetches; this is the case for our Intel platform. Once the coefficient values are obtained from the synthetic workloads, the memory service time for varying parallel programs is computed by using the collected number of memory read, write, and prefetch operations during the profiling. Computing individual coefficients for each workloads can increase the accuracy of the model but is left for future work.

5 The Speedup Prediction Model

This section presents the speedup prediction model for parallel loops. The model computes the thread stall times for LLC misses from Section 4 for a varying number of threads.

5.1 The Speedup Model

For an N-node system, the speedup of parallel loops for M number of CPU nodes each consisting of C cores is computed as follows. Let *CPU Time*$_S$ denote the CPU time required to complete the workload when using a single thread. Note that, *CPU Time*$_S$ does not include thread stall times. If there is no contention in the memory system and assuming perfect load balancing, we can expect a linear speedup and thus divide *CPU Time*$_S$ by $M \cdot C$. Let *Stall Time*(M) be the total stall times of a thread for all LLC misses from the thread when there are $M \cdot C$ threads. The speedup for M CPU nodes, $S(M)$, is given by

$$S(M) = \frac{\textit{CPU Time}_S/C + \textit{Stall Time}(1)}{\textit{CPU Time}_S/(M \cdot C) + \textit{Stall Time}(M)} \quad (14)$$

To compute *Stall Time*(M), the number of LLC misses for each memory node is computed by multiplying the CPU time per thread for M nodes (*CPU Time*$_S/(M \cdot C)$) by the LLC miss rate to each memory node j, LLC_j (note that $\forall_k$ and $\forall_i$ $LLC_j = LLC_{kij}$, because all threads have the same memory access ratio to each memory node). Then, for each memory node, the number of LLC misses to memory node j is multiplied by the average of the mean response times from M CPU nodes to memory node j, $\sum_{i=1}^{M} LRT_{ij}/M$. Hence, the total stall time of a thread is computed by

$$\textit{Stall Time}(M) = \sum_{j=1}^{N} \Big(\frac{\textit{CPU Time}_S}{M \cdot C} \cdot LLC_j \cdot \frac{\sum_{i=1}^{M} LRT_{ij}}{M} \Big) \quad (15)$$

where the value of LRT_{ij} is computed by applying the *M/M/1/N/N* queueing systems from Section 4. The product in the parentheses computes the total stall time of LLC misses served by memory node j. The stall time is the sum over all N memory nodes.

5.2 Implementation

The speedup prediction model has been implemented as a library called *LoopPerf*. The GOMP runtime system (version 5.4) has been modified to allow control the number of worker threads of a parallel loop. *LoopPerf* creates as many worker threads as there are cores in the system. Each thread is pinned to an individual core, parallelism is controlled by putting threads on non-allocated cores to sleep. The dynamic loop scheduler in our GOMP runtime system determines the amount of work to assign to a core based on the execution time of previous work. When fetching new work, the GOMP runtime system increases the amount of work assigned until it reaches an execution time of 30ms. This threshold has empirically been found to yield good results, but can be tuned for different architectures. *LoopPerf* provides three different versions of performance prediction, *LoopPerf-S*, *LoopPerf-T*, and *Best-F*. The following sections discuss the implementation details.

5.2.1 LoopPerf-S (Single)

LoopPerf-S predicts the performance of a parallel loop based on a single profiling run using one CPU node. For a parallel loop, it collects the memory request rates and LLC miss rates accessing to individual memory nodes. This assumes that the memory request rates and the LLC miss rates are constant for a varying number of threads.

5.2.2 LoopPerf-T (Twice)

As discussed in Section 3.3.3, the memory request rate can vary depending on the number of threads. *LoopPerf-T* considers such variations by allowing two profiling runs. The first profiling uses one CPU node and collects the mean memory request rate and LLC miss rate for each memory node. The second profiling uses all CPU nodes and applies linear regression to compute the parameter values for a varying number of CPU nodes. To benefit from the speedup information given by this option, therefore, a parallel loop needs to be executed more than two times. This is not a big concern because numerical applications usually execute the same parallel loops dozens or hundreds times.

5.2.3 Best-F (Best Fixed parameter values)

Best-F from prior work [8] employs simpler *M/M/1/N/N* queueing systems to model the speedup of parallel workloads and also does not take into account variations in the workload's memory service rate. Instead a fixed memory service rate is used for all benchmarks. The service rate of *Best-F* is found using an exhaustive search of the service rates of memory controllers and interconnection links and chooses the values that yield the minimum prediction errors for the 24 parallel loops of Table 1. *LoopPerf-S* and *LoopPerf-T* are compared to *Best-F* to show the benefits of the more accurate queueing models and the variable memory service rates. Section 8 discusses the differences and contributions of this work over *Best-F* in more detail.

6 Experimental Results

This section evaluates the presented speedup prediction model with the 24 parallel loops from Table 1 on two NUMA architectures, an AMD 64-core and and Intel 72-core platform. Details of the platforms are given in Section 3.3.1.

The accuracy of the prediction model is validated using the mean absolute percentage error (MAPE). MAPE is computed by taking the arithmetic mean of the percentage errors based on the difference between the measured and the predicted value. It is given by

$$MAPE = \frac{100\%}{n} \sum_{k=1}^{n} \left| \frac{a_k - p_k}{a_k} \right| \quad (16)$$

where a_k represents the actual and p_k the predicted value. In addition to MAPE, the speedup prediction curves for both platforms are presented in Fig. 12 (AMD) and Fig. 13 (Intel).

6.1 64-core AMD Opteron Platform

The results in Fig. 12 show that, in general, the presented speedup model, *LoopPerf-T* accurately predicts the speedup of the parallel loops with a geometric mean error of 8.3% confirming that the speedup model can be practically used

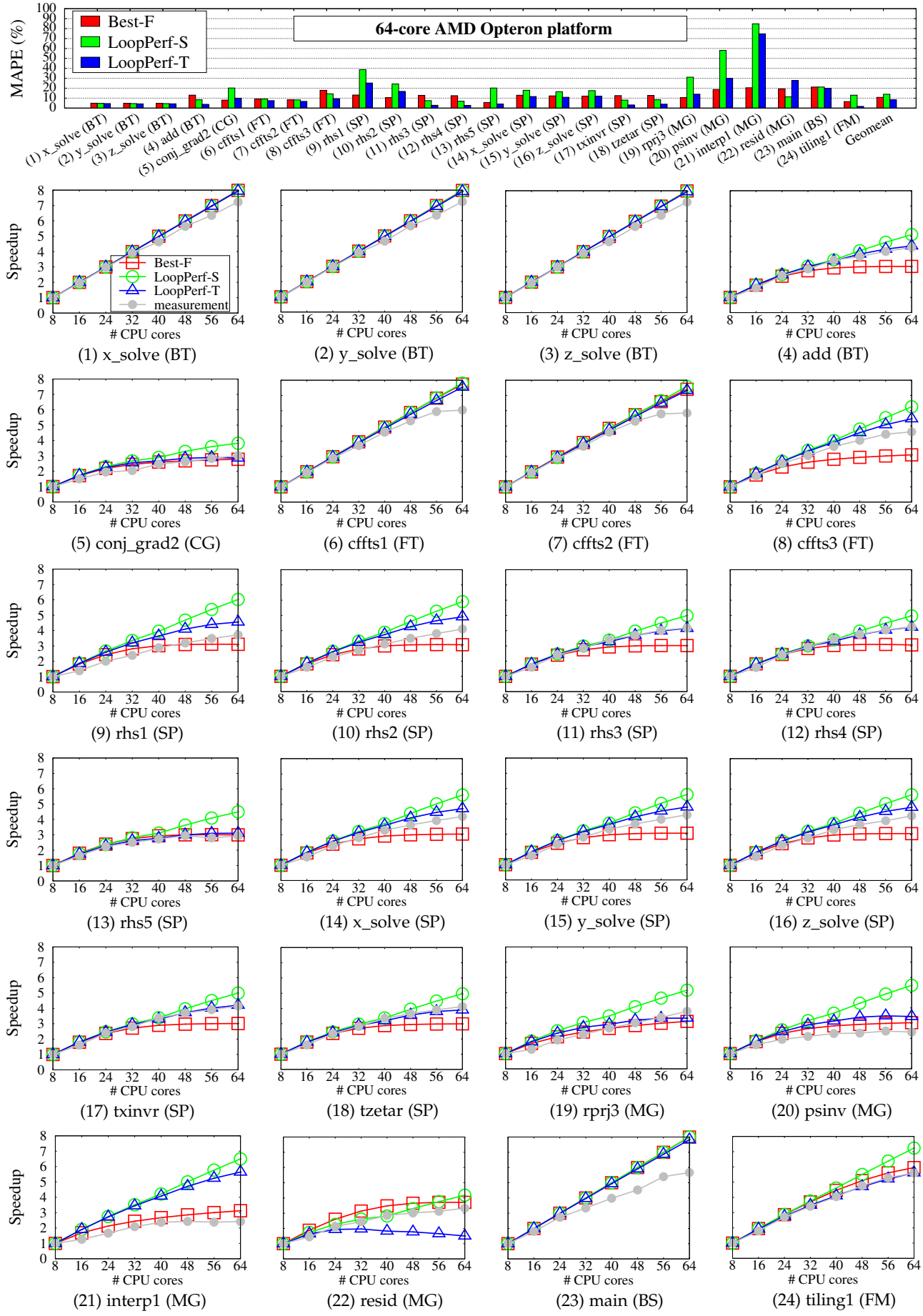


Fig. 12. The predicted speedup and the measured speedup of the parallel loops on the 64-core AMD platform.

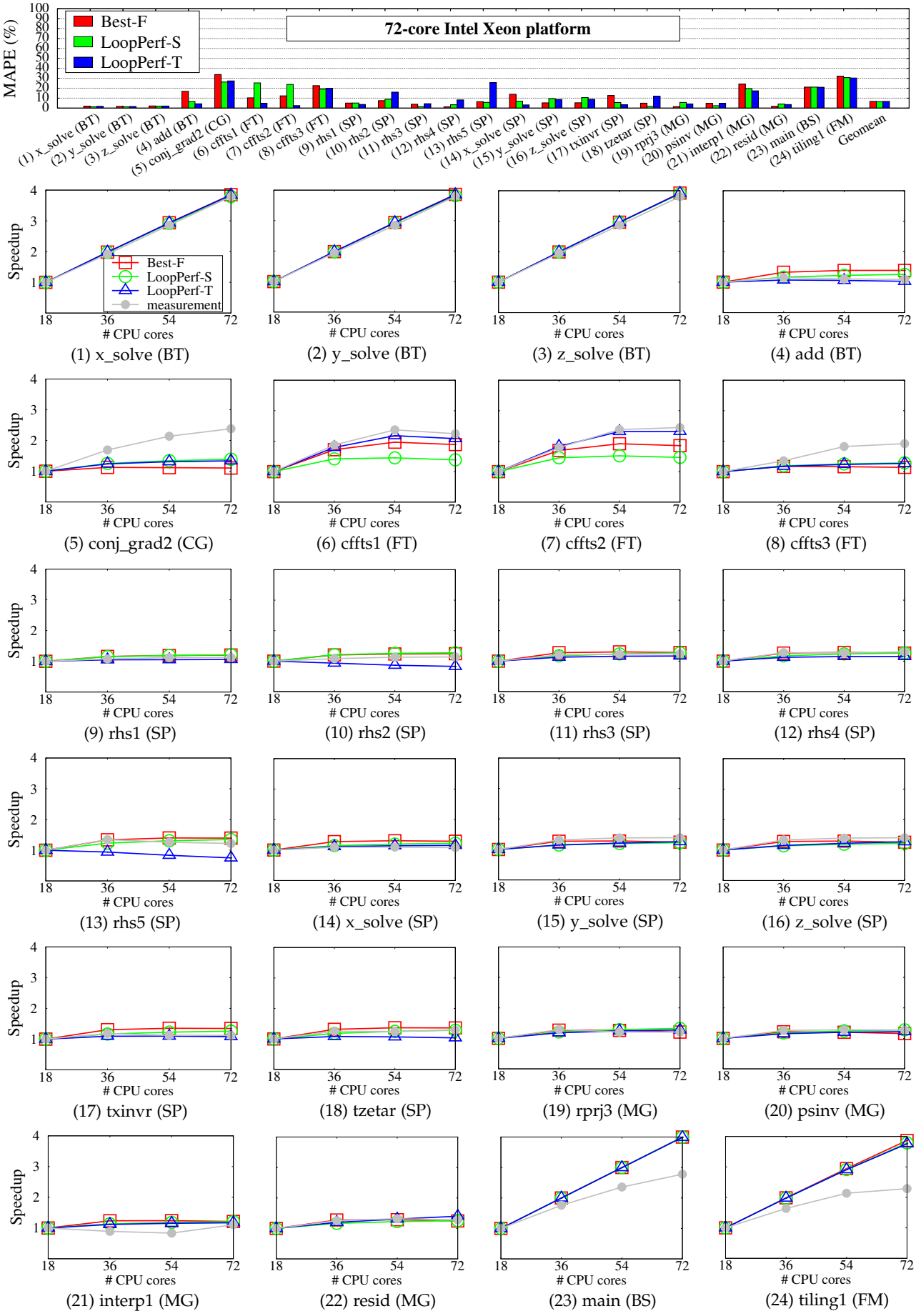


Fig. 13. The predicted speedup and the measured speedup of the parallel loops on the 72-core Intel platform.

for OpenMP applications. *LoopPerf-S* has a higher error with a geometric mean of 13.9%, and the *Best-F* configuration also shows a higher geometric mean error of 10.8%.

For Fig. 12 *(1)–(3) x/y/z_solve (BT)*, *(6)–(7) cffts1-2*, and the *(23) main* loop, *LoopPerf-S*, *LoopPerf-T*, and *Best-F* predict almost a linear speedup. These workloads have low memory access rates as shown in Fig. 4 (a), (c), and (d), and the speedup models consider these workloads to be CPU-intensive. However, the predictions have small errors on a large number of threads because of the loop scheduling overhead. Comparing *LoopPerf-T* with *LoopPerf-S* and *Best-F*, the advantages of *LoopPerf-T* become apparent for memory-intensive loops such as *(4) add*, *(5) conj_grad2*, *(8) cffts3*, *(9)–(13) rhs1-5*, *(14)–(16) x/y/z_solve (SP)*, *(17) txinvr*, and *(18) tzetar*. The results imply that *LoopPerf-T* can successfully compute the parameter values of the queueing systems compared to *LoopPerf-S* and *Best-F*. For example, looking at *(5) conj_grad2*, *(8) cffts3*, *(9)–(13) rhs1-5*, *LoopPerf-S* failed to accurately predict the speedup for loops with a larger number of CPU nodes. *LoopPerf-S* often over-estimates the speedup compared to the measurements. Workloads tend to have a higher ratio of memory prefetch operations (a higher memory service rate) on a small number of CPU nodes. *LoopPerf-S*, however, computes the memory service rates based only on a single profiling using one CPU node. Therefore, `LoopPerf-S` may over-estimate the memory service rates for a larger number of CPU nodes and yield a lower memory response time than the actual one. For speedup curves such as *(9)–(13) rhs1-5* and *(14)–(16) x/y/z_solve (SP)*, on the other hand, *Best-F* often over-estimates the speedup for a small number of CPU nodes and under-estimates for a larger number of CPU nodes. The trend shows that using one constant mean memory service rate does not capture the variance of the memory service rate well. *LoopPerf-T* provides good prediction accuracy and similar speedup curves with measurements for most parallel loops.

Analytical modeling through queueing models admittedly has limitations for irregular workloads regarding their memory access distribution and load imbalance. The *LoopPerf-T* technique does not accurately predict the speedup of irregular loops such as *(21) interp1* and *(22) resid* of the *MG* application. In Section 3, we have shown experimentally that the memory accesses of these workloads do not follow a Poisson distribution (Fig. 4 (c)) and that these workloads also suffer from a load imbalance in Fig. 9 (a).

Overall, the results show that *LoopPerf-T* is able to accurately predict the performance scalability of regular parallel loops. The experiments validate that the presented methodology can practically model memory performance of parallel loops in modern multi-socket multi-core platforms. Despite the higher error rates for pathological curves from irregular workloads, the high accuracy of the prediction technique for regular workloads makes the presented queueing system a good candidate for performance modeling and optimization in multi-socket multi-core systems.

6.2 72-core Intel Xeon Platform

Fig. 13 presents the speedup prediction results for the 72-core Intel Xeon platform. *LoopPerf-T* accurately predicts the speedup with a geometric mean error of 6.7%. *LoopPerf-S* and *Best-F* also achieve good accuracy with an error of 6.5% and 6.7%, respectively. The difference among the three methods is not prominent because the speedup is predicted for only four different allocations (1-4 CPU nodes). Note that hyperthreading has been disabled to not incur interference in a physical core in accordance with the simplifications stated in Section 3.1.

In Fig. 13, the speedup curves of CPU-intensive loops such as *(1)–(3) x/y/z_solve (BT)* and *(6)–(7) cffts1-2*, show a similar pattern to the AMD platform from Fig. 12. *LoopPerf-T* provides good predictions for memory-intensive parallel loops. For irregular loops such as *(21) interp1* and *(22) resid* of the *MG* application, *LoopPerf-T* provides better prediction results compared to the AMD platform. The effects of these irregular loops were smaller in the Intel system compared to the AMD system. All the three speedup models, however, do not predict the speedup of *(5) conj_grad2* and *(24) tiling1* well. In these cases, the performance is limited by other factors such as loop scheduling and cache coherence overhead between multiple sockets rather than the memory system.

7 Discussion

The analytical approach presented in this paper makes several assumptions as discussed in Section 3. Although the assumptions are justified for parallel loops of scientific applications in NPB, it remains a challenge to apply the presented approach to other types of parallel loops that do not satisfy these assumptions. Here, we briefly discuss potential solutions to address these challenges.

A first important assumption is that the loops have no loop-carried dependencies. In the presence of loop-carried dependencies (e.g., pipelined parallelism), the major limiting factors for performance are synchronization and scheduling overhead in addition to the memory performance. The presented performance model is able to offer an insight into the memory performance. To model the synchronization time for N threads *Sync Time (N)*, existing analytical approaches [28] can be employed.

Second, the presented model assumes a Poisson distribution for memory requests and exponential memory service times. For the targeted parallel loops, this assumption is verified in Section 3, however, other loops may exhibit different distributions. In that case, the queueing models need to be solved with discrete event simulation.

Third, contention at intra-node shared resources such as shared caches and floating point units can be modeled in a more sophisticated way. For example, although LLC contention is already implicitly considered in this work because the memory request rates are measured after LLC contention happens, using other intra-node resource interference models [29] can be useful if the performance needs to be estimated on a finer level. Note, however, since the presented model is evaluated using hardware performance counters, the effects of such resource contention are implicitly considered to a certain degree.

Lastly, all experiments in this paper assume (1) dynamic loop scheduling and (2) that the data is spread across all memory nodes as is standard practice for runtime systems that control the parallelism of workloads [2], [30]. In our previous work [8], *Best-F* has been evaluated for other scheduling methods including static, guided and dynamic scheduling as well as for different memory allocation schemes.

The results show that the presented queueing system-based approach works well for the different execution scenarios except for a number of pathological cases where the parallel loops suffer from a large load imbalance with static scheduling.

8 Related Work

Runtime systems often adjust the number of threads of parallel programs for optimizations on multi-cores. ACTOR [31] adjusts thread count for power and performance optimizations based on a prediction model that requires hardware performance counters. Varuna [3] executes a parallel program with a different number of threads at runtime and performs regression to compute the optimal thread count. Parcae [2] and Aurora [4] use hill-climbing approaches to find the optimal thread count. Aurora considers diverse metrics when computing the thread count. These approaches, however, do not provide information about the memory performance on modern multi-socket systems.

Several performance modeling techniques have been presented for multi-socket systems. Pandia [32] predicts the performance of parallel applications for different thread counts and placements. The performance prediction is based on six different profiling runs to obtain the performance features. NuCore [33] is an analytical model to predict the optimal core allocation for multi-threaded applications. NuCore finds the core allocation that maximizes the memory bandwidth usage at minimum core count. Integer programming is used to solve the model. A detailed DRAM performance model, DraMon [34], is employed to predict the memory performance in NuCore. DraMon requires a number of parameters that need to be obtained from expert knowledge or architecture data sheets. Our method is orthogonal to these techniques and has unique advantages. The presented method requires a small number of input parameters that can be obtained from hardware performance counters. In addition, the queueing systems analytically compute the performance of each memory controller and interconnection link separately using closed-form expressions; such information can be used for various optimizations.

Applying queueing models to model multiprocessor architectures has been discussed in the literature. Jonkers [12], [35] has presented conceptual queueing models for multiprocessor architectures consisting of multiple memory controllers and an interconnection network. However, these works do not provide an evaluation on real hardware platforms. Tudor et al. [6], [7] applied an *M/M/1* queueing system to evaluate memory contention in an SMP system with Uniform Memory Access (UMA) times. In contrast to our work, they do not apply a queueing system for the interconnection links in NUMA machines; instead, they used regression to evaluate the performance on a different number of CPU nodes. Moreover, the *M/M/1* model assumes an infinite number of queueing customers, however, multiprocessor systems contain a finite number of cores.

In our previous work [8], we have presented a speedup prediction model using *M/M/1/N/N* queueing systems. This paper extends the previous work in a number of ways. First, this paper provides an experimental study showing that parallel loops act like queueing customers. Second, the previous work uses simpler queueing systems assuming a fully-connected interconnection network and does not take into account memory performance variations with hardware optimizations. As reported in Section 6, the presented technique in this paper provides more accurate prediction results than the maximum bound of accuracy when using fixed memory service rates. Last, this paper focuses on OpenMP parallel loops and provides an evaluation for two different platforms while the previous work mostly evaluates OpenCL data-parallel kernels on a single platform.

9 Conclusion

With an increasing number of processor sockets and memory controllers in shared-memory systems, performance modeling of parallel loops poses several challenges that hinder analysts from using existing queueing model-based techniques. In this paper, we presented a methodology to model the memory system performance of multi-socket multi-core systems using queueing systems. For multi-socket systems, we presented hierarchical *M/M/1/N/N* queueing systems that are able to evaluate the performance of each interconnection link and each memory controller. The parameter values are computed in the presence of variations from hardware optimizations while solely relying on hardware performance counters of AMD and Intel processors. Based on the queueing systems, the performance of OpenMP parallel loops is predicted with average percentage errors of 8% for AMD and 7% for Intel multi-socket systems. The information obtained from the model can be used not only for performance modeling of parallel loops but also to improve overall CPU and memory system utilization [30].

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) funded by the Korean government, in part, by grants NRF-2015K1A3A1A14021288, 2016R1A2-B4009193, by the BK21 Plus for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU, grant 21A20151113068), and by the Promising-Pioneering Researcher Program of Seoul National University in 2015. ICT at Seoul National University provided research facilities for this study.

References

[1] L. Dagum and R. Enon, "OpenMP: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, 1998.

[2] A. Raman, A. Zaks, J. W. Lee, and D. I. August, "Parcae: A system for flexible parallel execution," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 133–144. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254082

[3] S. Sridharan, G. Gupta, and G. S. Sohi, "Adaptive, efficient, parallel execution of parallel programs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 169–180. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594292

[4] A. F. Lorenzon, C. C. D. Oliveira, J. D. Souza, and A. C. S. B. Filho, "Aurora: Seamless optimization of openmp applications," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–1, 2018.

[5] J. Sztrik, *Basic queueing theory*. University of Debrecen: Faculty of Informatics, 2011.

[6] B. M. Tudor, Y. M. Teo, and S. See, "Understanding off-chip memory contention of parallel programs in multicore systems," in *Parallel Processing (ICPP), 2011 International Conference on*. IEEE, 2011, pp. 602–611.

[7] B. M. Tudor and Y. M. Teo, "A practical approach for performance analysis of shared-memory programs," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 652–663.

[8] Y. Cho, S. Oh, and B. Egger, "Online scalability characterization of data-parallel programs on many cores," in *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Sept 2016, pp. 191–205.

[9] "GNU libgomp," http://gcc.gnu.org/onlinedocs/libgomp, [online; accessed June 2019].

[10] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik, "Locality and loop scheduling on numa multiprocessors," in *1993 International Conference on Parallel Processing - ICPP'93*, vol. 2, Aug 1993, pp. 140–147.

[11] B. Berg, J.-P. Dorsman, and M. Harchol-Balter, "Towards optimality in parallel job scheduling," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 1, no. 2, pp. 40:1–40:30, Dec. 2017. [Online]. Available: http://doi.acm.org/10.1145/3154499

[12] H. Jonkers, "Queueing models of shared-memory parallel applications," *Computer and Telecommunication Systems Performance Engineering*, 1994.

[13] M. Casas and G. Bronevetsky, "Active measurement of the impact of network switch utilization on application performance," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 165–174.

[14] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

[15] G. Sartori, "Hypertransport Technology," *Platform Conference*, 2001.

[16] B. Mutnury, F. Paglia, J. Mobley, G. K. Singh, and R. Bellomio, "Quickpath interconnect (qpi) design and analysis in high speed servers," in *19th Topical Meeting on Electrical Performance of Electronic Packaging and Systems*, Oct 2010, pp. 265–268.

[17] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, ser. ISCA '81. Los Alamitos, CA, USA: IEEE Computer Society Press, 1981, pp. 81–87. [Online]. Available: http://dl.acm.org/citation.cfm?id=800052.801868

[18] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 63–74. [Online]. Available: https://doi.org/10.1109/ISCA.2008.7

[19] 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 *et al.*, "The NAS parallel benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[20] S. Seo, G. Jo, and J. Lee, "Performance characterization of the nas parallel benchmarks in opencl," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, Nov 2011, pp. 137–148.

[21] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 72–81. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454128

[22] AMD, "AMD Opteron 6300 Series Processors 6380 product information," https://www.amd.com/en/products/6380, [online; accessed June 2019].

[23] Intel, "Intel Xeon Processor E7-8870 v3," http://ark.intel.com/products/84682/Intel-Xeon-Processor-E7-8870-v3-45M-Cache-2_10-GHz, [online; accessed June 2019].

[24] "Scipy stats.ks_2samp document," https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.stats.ks_2samp.html, [online; accessed June 2019].

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

[26] AMD, "BIOS and kernel developer's guide (BKDG) for AMD family 15h models 00h-0fh processors," 2012.

[27] Intel, "Intel Xeon Processor E5 and E7 v3 Family Uncore Performance Monitoring Reference Manual," 2015.

[28] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval, "Analytical modeling of pipeline parallelism," in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2009, pp. 281–290.

[29] V. Selfa, J. Sahuquillo, S. Petit, and M. E. Gmez, "A hardware approach to fairly balance the inter-thread interference in shared caches," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 11, pp. 3021–3032, 2017.

[30] Y. Cho, C. A. C. Guzman, and B. Egger, "Maximizing system utilization via parallelism management for co-located parallel applications," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '18. New York, NY, USA: ACM, 2018, pp. 14:1–14:14. [Online]. Available: http://doi.acm.org/10.1145/3243176.3243199

[31] M. Curtis-Maury, F. Blagojevic, C. D. Antonopoulos, and D. S. Nikolopoulos, "Prediction-based power-performance adaptation of multithreaded scientific codes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 10, pp. 1396–1410, Oct 2008.

[32] D. Goodman, G. Varisteas, and T. Harris, "Pandia: Comprehensive contention-sensitive thread placement," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 254–269. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064177

[33] W. Wang, J. W. Davidson, and M. L. Soffa, "Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale numa machines," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 419–431.

[34] W. Wang, T. Dey, J. W. Davidson, and M. L. Soffa, "Dramon: Predicting memory bandwidth usage of multi-threaded programs with high accuracy and low overhead," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 380–391.

[35] H. Jonkers, "Queueing models of parallel applications: the glamis methodology," in *Computer Performance Evaluation Modelling Techniques and Tools*. Springer, 1994, pp. 123–138.

Younghyun Cho received BS degree in computer science from the University of Seoul in 2013. He is currently working toward the PhD degree at the computer systems and platforms laboratory, Seoul National University. His research interests include runtime systems for manycore systems and performance modeling of parallel applications.

Surim Oh is currently a software developer at SAP Labs Korea, Inc. Surim Oh received B.S. in computer science and engineering from Sogang University in Seoul in 2015 and the M.S. in computer sicence and engineering from Seoul National University in 2017. She served as a research engineer for Hyundai Motor Company for a year. Her research interests include manycore runtime systems, parallel processing, and database management systems.

Bernhard Egger received the diploma in computer science from the Swiss Federal Institute of Technology in Zrich in 2001 and the PhD degree in computer science and engineering from Seoul National University in 2008. After spending three years at SAIT, Samsung Electronics research institute, he rejoined Seoul National University in 2011 as a faculty member where he currently is an associate professor at the Department of Computer Science and Engineering. His research interests include programming language design, compilers, and operating systems for heterogeneous manycore systems. He is a member of the IEEE and ACM. More information can be found at https://csap.snu.ac.kr/.