Ph.D. DISSERTATION

Parallelism Management for Co-Located Parallel Applications

동시에 실행되는 병렬 처리 어플리케이션들을 위한 병렬성 관리

BY

Younghyun Cho

August 2020

DEPARTMENT OF ELECTRICAL ENGINEERING &
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Ph.D. DISSERTATION

Parallelism Management for Co-Located Parallel Applications

동시에 실행되는 병렬 처리 어플리케이션들을 위한 병렬성 관리

BY

Younghyun Cho

August 2020

DEPARTMENT OF ELECTRICAL ENGINEERING &
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Parallelism Management for Co-Located Parallel Applications

동시에 실행되는 병렬 처리 어플리케이션들을 위한 병렬성 관리

지도교수 Bernhard Egger

이 논문을 공학박사 학위논문으로 제출함

2020 년 4 월

서울대학교 대학원

전기 컴퓨터 공학부

조영현

Younghyun Cho의 공학박사 학위논문을 인준함

2020 년 6 월

위 원 장	이재진
부위원장	Bernhard Egger
위 원	염헌영
위 원	Lawrence Rauchwerger
위 원	David Isaac August

Abstract

Running multiple parallel jobs on the same multicore machine is becoming more important to improve utilization of the given hardware resources. While co-location of parallel jobs is common practice, it still remains a challenge for current parallel runtime systems to efficiently execute multiple parallel applications simultaneously. Conventional parallelization runtimes such as OpenMP generate a fixed number of worker threads, typically as many as there are cores in the system, to utilize all physical core resources. On such runtime systems, applications may not achieve their peak performance when given full use of all physical core resources. Moreover, the OS kernel needs to manage all worker threads generated by all running parallel applications, and it may require huge management costs with an increasing number of co-located applications.

In this thesis, we focus on improving runtime performance for co-located parallel applications. To achieve this goal, the first idea of this work is to ensure spatial scheduling to execute multiple co-located parallel applications simultaneously. Spatial scheduling that provides distinct core resources for applications is considered a promising and scalable approach for executing co-located applications. Despite the growing importance of spatial scheduling, there are still two fundamental research issues with this approach. First, spatial scheduling requires a runtime support for parallel applications to run efficiently in spatial core allocation that can change at runtime. Second, the scheduler needs to assign the proper number of core resources to applications depending on the applications' performance characteristics for better runtime performance.

To this end, in this thesis, we present three novel runtime-level techniques to

efficiently execute co-located parallel applications with spatial scheduling. First, we present a cooperative runtime technique that provides malleable parallel execution for OpenMP parallel applications. The malleable execution means that applications can dynamically adapt their degree of parallelism to the varying core resource availability. It allows parallel applications to run efficiently at changing core resource availability compared to conventional runtime systems that do not adjust the degree of parallelism of the application.

Second, this thesis introduces an analytical performance model that can estimate resource utilization and the performance of parallel programs in dependence of the provided core resources. We observe that the performance of parallel loops is typically limited by memory performance, and employ queueing theory to model the memory performance. The queueing system-based approach allows us to estimate the performance by using closed-form equations and hardware performance counters.

Third, we present a core allocation framework to manage core resources between co-located parallel applications. With analytical modeling, we observe that maximizing both CPU utilization and memory bandwidth usage can generally lead to better performance compared to conventional core allocation policies that maximize only CPU usage. The presented core allocation framework optimizes utilization of multi-dimensional resources of CPU cores and memory bandwidth on multi-socket multicore systems based on the cooperative parallel runtime support and the analytical model.

Keywords: Runtime system, performance modeling, resource management
Student Number: 2013-20887

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Modern shared-memory multiprocessor systems are typically multi-socket multicore systems that consist of dozens of processor cores with an increasing number of CPU sockets and memory nodes to provide sufficient computation power and memory bandwidth. Today's parallel applications, however, are typically able to achieve only a fraction of the peak performance on such complex computer systems [58, 88]. Memory-intensive applications often cannot achieve the best performance when given full use of all available core resources because of the limited memory bandwidth and the non-uniform memory access (NUMA) property of multi-socket systems, while computation-intensive applications may under-utilize the memory system.

In modern data centers and high-performance computing (HPC) systems, running multiple parallel jobs on the same multicore machine is becoming more important to efficiently utilize the given hardware resources [41, 11, 93, 33].

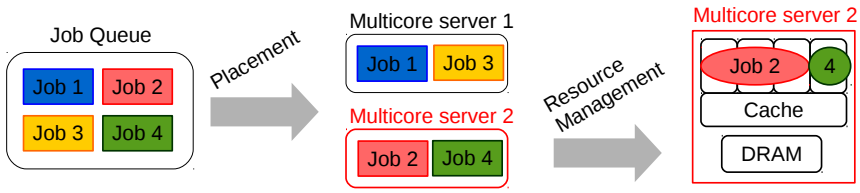


Figure 1.1 Job scheduling in modern data centers.

For example, as shown in Figure 1.1, data centers typically use a cluster-level scheduler that decides which parallel jobs will be co-located on the same multicore node [25, 26, 48, 10, 93]. Co-location of parallel jobs may improve resource utilization if the co-located workloads have different resource requirements, e.g. CPU-intensive and memory-intensive workloads. Then, the runtime system manages multicore resources for the co-located workloads to meet the given optimization goal (e.g. reducing execution time). On the other hand, many parallel workloads are able to run on a varying number of core resources using either compiler/runtime support [53, 71, 40] or Linux's resource isolation API such as `sched_setaffinity` and `cgroup` (as used in Docker [27]). In this context, allocating the proper number of core resources for co-located applications to optimize application performance and/or platform throughput has been an important topic of research in the compiler and runtime community [65, 72, 76, 75, 21, 81, 29, 28, 59, 91, 15]

While co-location of parallel jobs is already common practice, it still remains a challenge for current runtime systems to efficiently execute multiple parallel applications on modern multicore systems. Conventional parallelization runtime systems such as OpenMP [9] and Intel TBB [73] assume that each parallel application can utilize all existing hardware resources and thus generate threads, typically as many as there are cores in the given system, to utilize

all physical core resources. On such conventional runtimes, however, multiple parallel applications do not run efficiently. First, applications may not achieve their peak performance when given full use of all available core resources. Moreover, the execution model is not scalable for an increasingly large number of core resources because the OS kernel needs to manage all (kernel-level) threads generated by all running parallel applications. In this context, the runtime and OS community has pointed out that spatial scheduling is a scalable design for (future) multicore systems [5, 57, 90]. To overcome the scalability issue, the spatial scheduling approach provides distinct core resources to the co-located applications, and then the applications adapt their execution to the allocated core resources.

The spatial scheduling approach has two fundamental research issues. First, applications ideally should have malleability which is the ability to dynamically adjust the degree of parallelism (DoP) to adapt to the changes to the allocated core resources. Without malleability, applications can suffer from a significant slowdown when allocated core resources change due to the thread migration overhead and unbalanced numbers of threads among allocated core resources. Although the parallel computing research community has presented several runtime techniques to achieve malleable parallel execution, these existing techniques require special compiler/runtime [71, 72] that limits their applicability or use a simplistic task/work scheduling which can incur a significant scheduling overhead for a large number of core resources [40]. Second, the spatial scheduling approach needs to decide the proper number (and location) of core resources between co-located applications depending on the given scheduler's policy. Many existing core allocation techniques are predominantly CPU- or speedup-centric, meaning that they assign more core resources to CPU-intensive applications or highly-scalable applications [76, 75, 21, 81, 91], by leveraging runtime heuris-

tics or application speedup information. Although optimizing CPU usage has long been a common strategy for many resource management problems, the approach ignores the fact that multiple resources such as CPU and memory bandwidth are consumed simultaneously. Optimizing only CPU usage can lead to under-utilized memory systems resulting in an inefficient tail execution once all the computation-intensive applications have been finished.

In this thesis, we focus on improving the runtime performance (the execution time) of co-located parallel applications on multi-socket multicore systems thereby reducing the operating cost of HPC and data centers. To achieve this goal, the first idea of this work is to ensure spatial scheduling for executing co-located parallel applications. The first research goal is to provide efficient malleable parallel execution in spatial core allocation for shared-memory parallel applications with minimal changes to the current runtime system. In particular, we focus on providing malleable execution for OpenMP workloads which are widely used in many HPC and data center workloads. Based on the malleable parallel runtime support, the next research goals focus on performance modeling of parallel programs and parallelism management for co-located parallel applications. While performance modeling has long been an important research issue, existing modeling techniques for core resource management rely on additional efforts before applications are executed such as offline training [65] or machine learning [36, 29]. Ideally, the performance modeling and parallelism management should be done online and within a reasonable overhead. Therefore, we focus on an analytical solution that can efficiently estimate the performance of parallel programs and can also be applied to runtime systems while providing useful insights for the resource manager to develop an appropriate scheduling policy. Finally, based on the malleable runtime and the analytical performance model, we aim to provide a parallelism management framework that determines

the core allocation for co-located parallel applications and improve the overall system throughput.

1.2 Background

To better understand the research issues of this thesis, this section provides background information about the OpenMP runtime system. We also provide the information about our target multi-socket multicore systems that are used for evaluation throughout this thesis.

1.2.1 The OpenMP Runtime System

In this thesis, we mainly consider OpenMP parallel applications. OpenMP is the de-facto standard for shared-memory parallel processing in HPC and is also widely used for many data center and big-data workloads.

OpenMP's parallelism is based on the fork-join execution model [9]. Applications consist of multiple parallel fork-join sections that may consist of one or more parallel loops. Parallel loops annotated with the `parallel for` pragma are the basic mechanism to initiate parallelism in OpenMP applications, and different parallel loops exhibit different resource requirements and performance characteristics. To consider applications' changing performance characteristics, in this thesis, we focus on optimizing the execution of co-located parallel applications in the level of parallel loop.

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 itera-

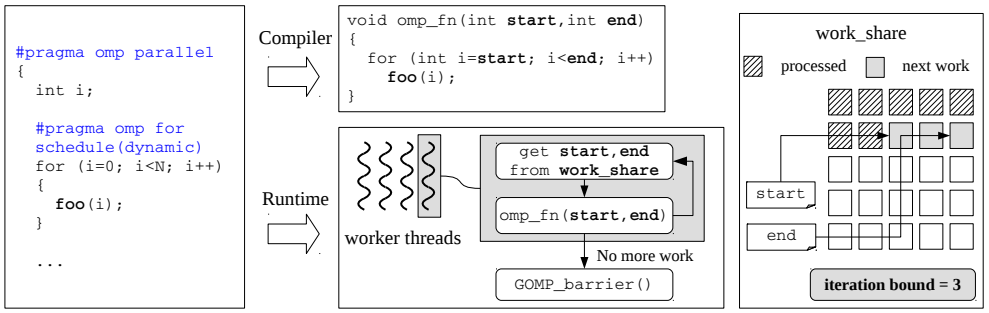
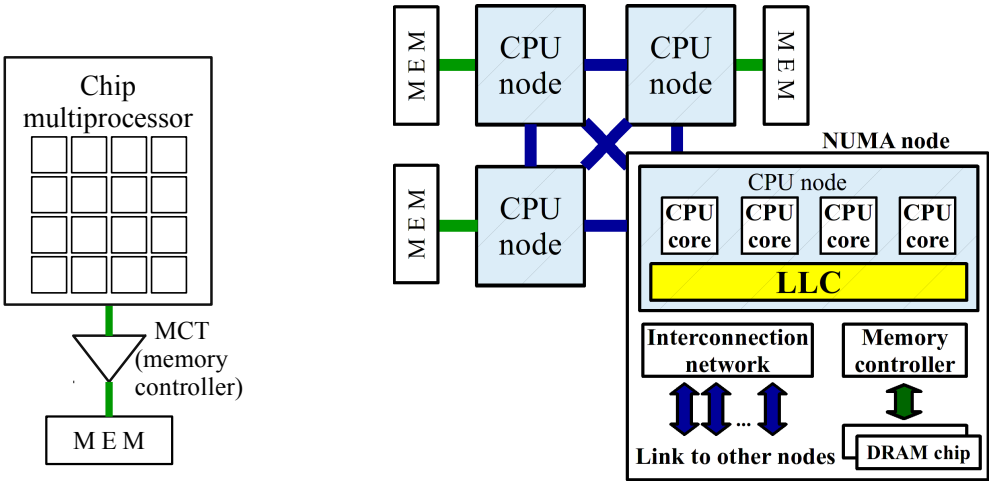


Figure 1.2 Execution model for *parallel for* loop.

tions 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 Figure 1.2 for the GNU OpenMP (GOMP) runtime system [34]. 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, 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 [54], 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.

An important observation of the OpenMP loop scheduling is that, in principle, the work of a parallel loop can be divided into multiple chunks of work which can be scheduled dynamically. While the default OpenMP loop schedulers use a fixed number of worker threads and do not support malleable execution, we exploit the inherent malleability in OpenMP parallel loops and provide malleable execution through a runtime-level support presented in Chapter 2.



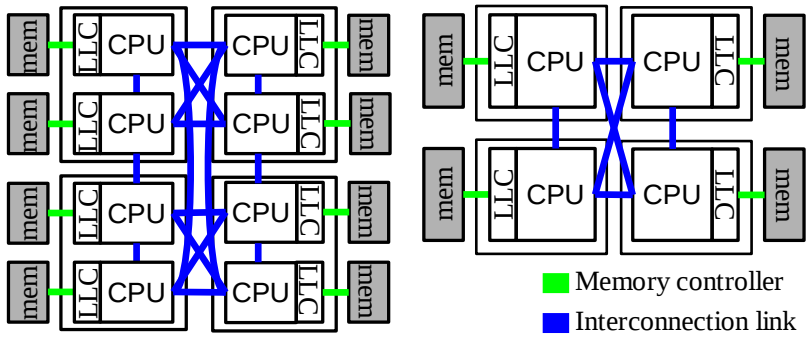
(a) SMP system

(b) Multi-socket multicore system

Figure 1.3 SMP and multi-socket multicore systems.

1.2.2 Target Multi-Socket Multicore Systems

The runtime techniques discussed in this thesis assume multi-socket multicore systems that contain multiple CPU sockets and memory nodes. Such multi-socket multicore systems are becoming more common in HPC and even data centers to provide more computational power and memory bandwidth. Figure 1.3 provides a simplified view of symmetric multiprocessing (SMP) and multi-socket multicore systems. Unlike an SMP system that comprises multiple cores and one memory, multi-socket systems contain a number of memory controllers to increase the memory bandwidth in the presence of a large number of cores. In such systems, one node consists of a CPU node, itself composed of a group of CPU cores, and its attached memory node. The individual nodes are connected by an interconnection network such as AMD's HyperTransport [74] or Intel's QPI (Quick Path Interconnect) [68]. These architectures exhibit Non-Uniform Memory Accesses (NUMA) characteristics because of the varying ac-



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

Figure 1.4 Block diagram of the target multi-socket multicore platforms.

cess latencies of the cores to the different memory controllers.

Our work has been evaluated on two commodity multi-socket platforms, a 64-core AMD Opteron and a 72-core Intel Xeon system. The AMD platform, shown in Figure 1.4 (a), comprises a total of eight CPU nodes in four physical processor packages (AMD Opteron 6380 [1]) and 128 GB of memory. The AMD Opteron processors run at 2.5GHz. 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 [74] with a maximum hop distance of two. The Intel system, shown in Figure 1.4 (b), has four Intel Xeon E7-8870 v3 [42] processors (2.1GHz) each consisting of 18 cores sharing a 45 MB LLC. The system is equipped with 512 GB of memory (some experiments used 756 GB of memory (e.g. Section 4.4) before we changed the DRAM chips). Each processor represents a CPU node, the four nodes are connected with Intel's QPI [68].

1.3 Contributions

In this thesis, we propose novel runtime-level techniques for the aforementioned goals for executing co-located parallel applications.

1.3.1 Cooperative Runtime Systems

The first research contribution of this thesis is a runtime-level technique that provides malleable parallel execution for OpenMP applications under dynamic spatial scheduling. We call a runtime system using this technique a *cooperative runtime system*. The cooperative runtime techniuqe has been implemented into the GNU OpenMP runtime [34] and allows OpenMP programs to dynamically adjust the DoP with a low overhead. OpenMP applications typically have multiple parallel code sections (e.g. *parallel for* loops) that may exhibit different performance characteristics. Therefore, applications should be able to adjust the DoP for each parallel loop to fully exploit the benefits of spatial core allocation. The cooperative runtime system provides this ability without any modifications to the application code or the compiler. To dynamically change the DoP of a parallel loop, i.e. number of active threads, the technique splits the workload into multiple chunks of work and dynamically schedules the chunks on the provided core resources. In Chapter 2, we show the benefits of malleable execution using the cooperative runtime system when executing OpenMP applications in dynamic spatial core allocation compared to the traditional OpenMP runtime that does not adjust the DoP at runtime. Some parts of this runtime technique have been discussed in our previous paper at The International Conference on Parallel Architectures and Compilation Techniques (PACT) 2018 [15] and also presented at The Workshop of Programmability and Architectures for Heterogeneous Multicores (Multiprog) 2017 [18].

1.3.2 Performance Modeling

The second contribution of this thesis is an analytical performance model for estimating resource utilization and the performance of parallel loops on shared-

memory multi-socket multi-core systems in dependence of the provided core resources. The approach employs queueing theory to model memory accesses in multicore systems; the queueing model allows us to compute useful performance information such as memory response time and bandwidth utilization by using closed-form equations. Based on the key insight that scalability of OpenMP parallel loops is typically 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. In Chapter 3, evaluated with 24 OpenMP parallel loops, we validate the accuracy of the presented queueing system by comparing the measured and modeled speedup curves. This work has led to several publications in parallel computing venues such as PACT 2016 [17] and TPDS 2020 [19]. In PACT 2016 [17], we presented an earlier version of the analytical model. A more sophisticated version has been presented in TPDS 2020 [19].

1.3.3 Parallelism Management

Employing the cooperative runtime system and the analytical model, we finally present a parallelism management and core allocation framework called NuPoCo (**NU**MA **p**erformance **o**ptimizations for **co**-located parallel applications). From the cooperative runtime support, for an OpenMP application, the framework keeps track of changing parallel loops and manages parallelism for each of the co-located parallel loops.

NuPoCo maximizes the overall system utilization by considering the uti-

lization of both CPU cores and memory bandwidth to determine the proper number of cores for each of the co-located parallel loops. NuPoCo leverages the analytical model to compute CPU and memory bandwidth utilization and then determines the degree of parallelism based on a greedy allocation algorithm to maximize the sum of CPU utilization and memory bandwidth utilization. Our evaluation shows that NuPoCo improves the average system throughput (i.e. reduction of execution time) on commodity AMD and Intel multi-socket systems in the order of 10 to 20% over conventional execution models using standard Linux time-sharing. This parallelism management framework has been presented in PACT18 [15].

1.4 Related Work

1.4.1 Cooperative Runtime Systems

To enable spatial scheduling on a Linux-based system, the most simple approach is using system calls such as `sched_setaffinity` or Linux's `cgroup` for CPU affinity masking. For example, in SBMP [76] and C3PO [75], an application's worker threads are pinned to the assigned cores without changing the degree of parallelism of the application. Hence, applications still use a fixed number of kernel-level threads and the approach cannot avoid thread interference on the same cores and suffers from a load imbalance due to thread migration and unbalanced numbers of threads among cores.

To dynamically adjust the DoP (i.e. the number of kernel-level threads) of parallel programs, runtime systems often use supports from the application runtime. The OpenMP runtime system [34] already provide a similar feature. For example, OpenMP's *OMP_parallel* code regions can run with any number of threads (if not defined by the application programmer) that can be selected

when entering the parallel region. Several runtime systems [21, 36, 28] assign a varying number of threads for such parallel regions. Once created, however, the number of worker threads within a parallel section remains constant and the benefit is limited compared to fully malleable execution.

To provide malleability, several compilers [53, 71, 72] generate flexible code. The basic idea is to divide the total work into composable chunks of work and schedule them onto provided core resources. Varuna's [81] virtual tasks (VTask) decouple software from hardware threads and require no compiler support. The VTask technique splits the parallel work by intercepting creation of Pthreads and manages them using a work pool. However, these approaches require special compiler and runtime system which limits the applicability and makes it difficult to exploit advanced runtime optimizations provided by the existing runtime system such as OpenMP.

Callisto [40] uses a scheduler activation technique in the OpenMP runtime to provide malleable execution for OpenMP parallel programs. Callisto is similar to our work as our approach also achieves workload malleability by leveraging dynamic loop scheduling logic in the OpenMP runtime system. However, our approach has merits that we perform dynamic granularity control and can preserve data locality optimizations on NUMA multi-core systems through hierarchical scheduling. Chapter 2 discusses the merit of our work scheduling.

1.4.2 Performance Modeling

Several performance modeling techniques have been presented for multi-socket systems. Pandia [35] 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 [88] is an analytical model to predict the optimal core allocation for multi-threaded applica-

tions. 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 [89], 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. On the other hand, the presented method in this thesis requires a small number of input parameters that can be obtained from hardware performance counters at runtime. 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 [47, 46] 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. [85, 84] 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 preliminary work [17], we have presented a speedup prediction model using *M/M/1/N/N* queueing systems. In current model, we extended the previous work in a number of ways. First, the model now 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. The presented technique in this thesis provides more accurate prediction results than the maximum bound of accuracy when using fixed memory service rates, as shown by our evaluation in Chapter 3.

1.4.3 Parallelism Management

To determine the proper thread or core count between co-located parallel applications, SBMP [76], SCAF [21], and Varuna [81] execute a parallel program in several configurations at runtime and perform a regression analysis to estimate the performance scalability. Then, they determine the thread count according to a policy. For example, SCAF [21] selects the thread count that maximizes the speedup of all running applications. Parcae [72], C3PO [75], and Aurora [60] perform hill-climbing to reach an optimal thread count. For example, Parcae [72] initially reserves an equal number of cores to all running parallel applications. ACTOR [22] adjusts thread count for power and performance optimizations based on a prediction model that requires hardware performance counters. Emani et al. [29, 28] and ADAPT [52] apply machine learning models to compute the number of threads assigned to applications. These approaches, however, do not provide information about the memory performance on modern multi-socket systems. We focus on maximizing overall system utilization of all CPU cores and memory bandwidth with an analytical solution.

Thread placement is known to strongly affect performance on multi-socket systems [92, 24]. While previous parallelism managers do not consider thread placement or uses simplistic linear partitioning [76, 75], we consider the architecture's NUMA properties to determine a good placement of threads to cores. A number of thread and data placement techniques have been presented for multi-threaded applications on multi-socket systems [92, 62, 24, 82]. LIRA [20] performs heuristics to determine thread placement of co-located applications

on NUMA CPU nodes. Threads are placed in order to minimize resource contention while preserving an efficient data placement. Unlike these approaches, we focus on assigning the proper number of threads between co-located parallel applications at runtime.

Recently, the co-scheduling approach, i.e. co-locating multiple parallel jobs on the same multicore node, is gaining importance in the HPC domain. Breitbart *et al.* [12, 10] perform co-scheduling of multiple applications on multi-core nodes. To maximize system throughput their scheduler detects an application's main memory bandwidth utilization at runtime and co-locate applications that may benefit from co-scheduling on the same nodes using migrations. However, they do not adjust the number of threads for parallel applications; parallelism management has a large potential to improve single node performance as shown by our previous researches [15].

In data centers, workloads usually use a workload abstraction such as Hadoop and Spark and are managed by several tools such as virtualization and Linux's control groups. Similar to HPC schedulers, data center schedulers also require applications' resource requirements and user runtime estimates. Mesos [41] is a data center operating system that considers workload's varying resource requirements. There are several scheduling techniques to consider heterogeneous clusters [86] and to reduce the burden of user runtime estimates [70].

1.5 Organization of this Thesis

The remainder of this thesis is organized as follows. Chapter 2 introduces the cooperative runtime system that achieves efficient dynamic spatial scheduling for simultaneously running parallel programs. Chapter 3 presents the analytical performance model for estimating resource utilization and the scalability of

parallel programs. Then, our parallelism and core management techniques that leverages the runtime support from Chapter 2 and Chapter 3 are presented. Chapter 4 presents NuPoCo, the parallelism management framework to manage parallelism of parallel loops for co-located parallel applications. We finally conclude this thesis in Chapter 5.

Chapter 2

Dynamic Spatial Scheduling with Cooperative Runtime Systems

2.1 Overview

As more and more core resources are integrated into a single shared-memory multicore system, spatial scheduling that provides distinct core resources to running parallel applications (i.e. space-sharing) is regarded as a promising approach for efficiently executing multiple co-located parallel applications [87, 76, 29, 21, 28]. A key challenge of spatial scheduling is providing an efficient runtime environment for parallel applications to execute efficiently in spatial core allocation that can dynamically vary at runtime (i.e. dynamic spatial scheduling). For example, spatial schedulers may assign a new core allocation once a new parallel job begins or ends. Moreover, applications have multiple phases (e.g. different parallel loops) that may have different performance characteristics. Spatial schedulers thus need to determine a new allocation to consider applications' different phases. In this context, parallel programs ideally should

have malleability meaning that programs need to be able to dynamically adjust the degree of parallelism (DoP) in accordance with the provided core resources.

Conventional parallelization runtimes such as OpenMP [9], Intel TBB [73], Cilk [8], and OpenCL [49], however, do not support such a malleable execution. These runtime systems generate a fixed number of worker threads using kernel-level threading such as Pthreads, typically as many cores as there are in the given multicore system, to utilize all physical core resources. Consequently, parallel applications may suffer from a significant slowdown when allocated core resources dynamically change due to the thread migration overhead and unbalanced numbers of worker threads among allocated core resources. In this context, the parallel computing research community has presented several runtime techniques to achieve malleable parallel execution. The basic idea is to divide the total work into a certain number of chunks of work and dynamically schedule them onto the available core resources at runtime. Existing runtime techniques, however, require their special compiler/runtime framework [71, 72] that limits their applicability or remain rooms for performance improvement in terms of their simplistic task/work scheduling [40] which can suffer from a scheduling overhead for a large number of cores.

In this chapter, we present a cooperative runtime technique to provide malleable parallel execution for OpenMP applications. We implemented this runtime support in the GNU OpenMP runtime system (provided by gcc-9) [34]. With the cooperative OpenMP runtime system, OpenMP parallel programs can dynamically adjust the DoP, i.e., the number of active worker threads, for varying core allocation. The cooperative OpenMP runtime system consists of two key runtime techniques called *cooperative user-level tasking* and *cooperative dynamic loop scheduling* that provide malleability at the level of parallel loop. To be able to dynamically adjust the DoP of a parallel loop, the coopera-

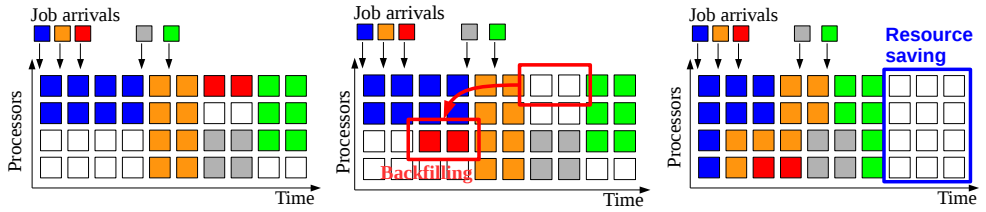
tive runtime system splits the parallel loop's iteration space into multiple tasks (with user-level tasks) and dynamically schedule them onto the available core resources. To reduce the overhead from dynamic scheduling, the work scheduler uses hierarchical scheduling where the tasks are partitioned into multiple regions and scheduled by distributed schedulers. We have evaluated the cooperative runtime system for eight NPB applications [4] on a 64-core AMD Opteron and a 72-core Intel Xeon system. The evaluation shows that the cooperative OpenMP runtime system can execute OpenMP applications more efficiently in spatial core allocation, with 20–30% shorter execution time on average, compared to the conventional spatial scheduling that does not adjust the DoP of the application.

As a summary, the following research contributions are discussed in this chapter.

- A runtime-level support, called a cooperative runtime system, to enable malleable parallel execution for OpenMP applications without any modifications to the application code or the compiler.
- An efficient user-level tasking and dynamic scheduling technique to achieve both malleability and high performance for OpenMP parallel loops.
- An evaluation of the cooperative runtime system that shows the benefits of malleable parallel execution in spatial scheduling.

2.2 Malleable Workloads

In this section, we show the benefit of achieving malleability in spatial scheduling. Conventional spatial schedulers in the HPC domain have assumed rigid jobs that run only with a specific number of processors and inform the scheduler that how much time the job will run (called user runtime estimates) [32].



(a) FCFS (b) FCFS with backfilling (c) Malleable jobs

Figure 2.1 Spatial scheduling depending on the job flexibility.

However, such an approach suffers from resource fragmentation. For example, in First-Come First-Served (FCFS) scheduling shown in Figure 2.1 (a), processors that cannot meet the requirements of the next job need to remain idle until additional processors have become available. Backfilling is a technique that allows jobs to execute earlier as long as the jobs do not delay the start of other jobs, as shown in Figure 2.1 (b). Backfilling schedulers can reduce resource fragmentation of FCFS scheduling and have been adopted in real HPC centers in the past two decades [56, 66]. In practice, however, backfilling scheduling still suffers from resource fragmentation depending on the job resource requirements. In addition, user runtime estimates are typically much longer than the actual job runtime and result in resource underutilization. In this context, scheduling malleable jobs is gaining importance [40, 6] to achieve maximal performance. Malleable jobs are able to execute with any number of threads and can change during the job runtime. Thanks to the flexibility, as shown in Figure 2.1 (c), malleable jobs can theoretically provide higher resource utilization and shorter job response times [6].

As we discussed in Section 1.2.1, the default OpenMP runtime system does not support malleable execution because the runtime system generates a fixed number of worker threads. To execute such parallel applications in spatial core allocation that changes dynamically, on Linux-based systems, the most simple

approach [76, 75, 91] is using system calls such as `sched_setaffinity` or Linux's `cgroup` to allow the application's threads to execute on the provided core resources. On such runtime systems, applications can suffer from a significant slowdown when allocated core resources change due to the thread migration overhead and unbalanced numbers of threads among allocated core resources. To execute parallel applications efficiently in spatial scheduling, therefore, applications ideally should have malleability to dynamically adjust the DoP to adapt to the changes to the allocated core resources. Such malleable execution then allows only one active thread to execute on each core resource.

2.3 Cooperative OpenMP Runtime System

As discussed in Section 1.2.1, an OpenMP application consist of one or more parallel each exhibiting different performance characteristics. The presented cooperative runtime technique provides the ability for each parallel loop to adjust the DoP at runtime. The technique exploits inherent malleability in the OpenMP programming model, and allows for a malleable execution without requiring any modifications to the application code.

The cooperative OpenMP runtime system enables malleable parallel execution for OpenMP parallel loops based on two key scheduling techniques called cooperative user-level tasking (COOP-ULT) and cooperative dynamic loop scheduling (COOP-DYN). In OpenMP, parallel loops annotated with the `parallel for` pragma are the basic mechanism to initiate parallelism in OpenMP applications. For an OpenMP parallel loop, the outermost loop iterations represent the smallest parallel unit of work which are scheduled by the OpenMP runtime system. COOP-ULT and COOP-DYN are orthogonal techniques which provide malleability for different types of OpenMP parallel loops. While COOP-

ULT enables malleable execution for OpenMP parallel regions (`OMP_parallel`) that may incorporate multiple parallel loops with static scheduling, COOP-DYN enables malleable execution for parallel loops annotated with dynamic loop scheduling (e.g. `schedule(dynamic)`). These two techniques consider common parallel programming patterns in OpenMP and can provide malleability for existing OpenMP applications in well-known benchmarks such as NPB3.4 [4], Parsec [7], and Rodinia [14]. We present the details of these techniques in the following sections.

2.3.1 Cooperative User-Level Tasking

To understand the COOP-ULT technique, we first discuss the type of OpenMP parallel code that can be executed under the COOP-ULT technique. Figure 2.2 shows an example OpenMP fork-join parallel code that can be executed with COOP-ULT and illustrates how the default (GNU) OpenMP runtime system executes the code. In the figure, the loops annotated with "# pragma omp for" are executed in parallel, and there is an implicit barrier between the two parallel loops executing functions `A` and `B`. Then, the compiler generates code `fn` on the right side. The code contains two functions `omp_fn1` and `omp_fn2` that execute *parallel for* loops for A and B. The code also contains a barrier function, shown by `GOMP_Barrier`, between these two functions `omp_fn1` and `omp_fn2`. For a barrier, all worker threads need to stop until all other threads reach the barrier point.

The OpenMP runtime system then executes the code `fn` using multiple threads. For example, in the figure, four threads are executed in parallel. The amount of work (i.e. the number of loop iterations) between the worker threads is equally partitioned given the static partitioning method.

The OpenMP programming model provides several ways to select the num-

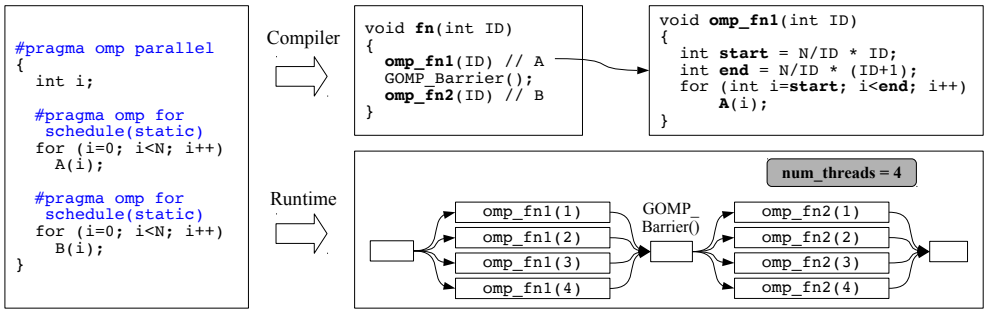


Figure 2.2 Execution model for OpenMP parallel sections.

ber of worker threads. First, the programmer can annotate the number of worker threads using the `num_threads` clause. If not defined by the `num_threads` clause, thread count can basically be any number. The OpenMP runtime [34], by default, generates threads as many as system's core resources to utilize all physical cores. Users can also use environment variable `OMP_NUM_THREADS` to use a specific number of worker threads.

Achieving Malleability

An important observation of the OpenMP runtime model is that the thread count can be any number if not specified by the programmer. Therefore, if we create more worker threads, each thread will execute a smaller amount of work. We use this property to achieve malleable execution.

Figure 2.3 illustrates the basic idea of the cooperative runtime technique. To be able to change the number of threads dynamically, the key idea of cooperative user-level tasking is to (1) generate many number (larger than the number of physical cores) of threads (2) then dynamically execute them on the allocated core resources. Thanks to this dynamic scheduling, cooperative user-level tasking can allow only one worker thread to execute on a active core

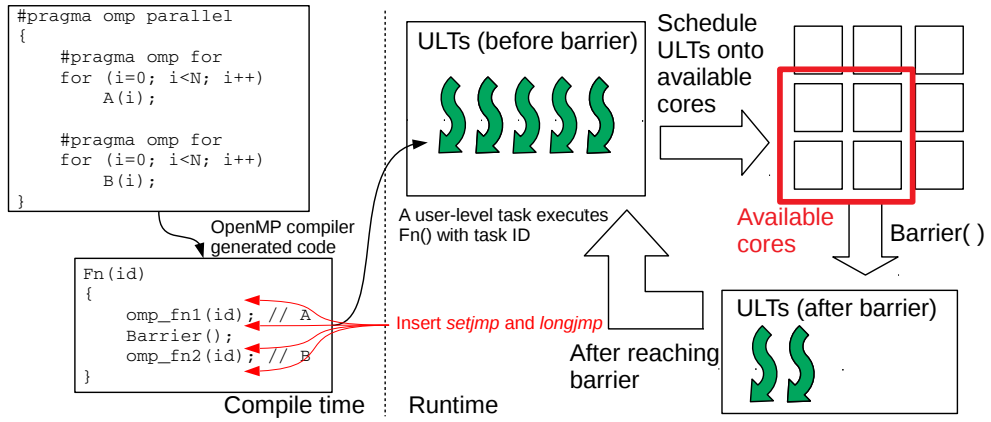


Figure 2.3 Execution model for cooperative user-level tasking.

resource and thus can reduce interference caused by unbalanced numbers of threads on allocated cores. However, by default, the OpenMP runtime system uses kernel-level threads (Pthreads) for each thread object. Managing a large number of kernel-level threads requires a huge management cost and incurs a high runtime overhead due to frequent sleep and wakeup operations. Also, the default OpenMP runtime system prevents from creating a large number of kernel-level threads (e.g. more than a thousand) due to its high management cost.

To reduce this runtime overhead, we use user-level tasks (ULTs) instead of kernel-level threading. Cooperative user-level tasking first creates a many number of user-level tasks that execute the parallel code. (We currently create $\#cores \times 20$ ULTs; we will discuss why in the following section.) Each physical core executes only one kernel-level thread that dynamically fetches ULTs at runtime. Since switching ULTs does not require an access to the Linux kernel, this approach can minimize the threading overhead while achieving malleability. There are a number of OpenMP runtime systems such as Callisto [40] and

Bolt/Argobots [79, 45] that exploit user-level threading. Compared to other user-level theading techniques used in these previous works that typically use a timer interrupt to get into the scheduler routine, we use simpler ULT scheduling (called zero-interrupt ULT scheduling) as we do not need to switch tasks before completing a task or reaching a barrier. We implemented this by inserting *setjmp* and *longjmp* API that saves and restores the minimal execution status of the executing code. Our technique requires small modifications to the OpenMP runtime system. In addition, in contrast Callisto and Bolt/Argobots, our ULT scheduling considers the NUMA property in multi-socket systems and also aims to reduce the dynamic scheduling overhead. In the below, we present a NUMA-aware and efficient ULT scheduling method called hierarchical scheduling.

Hierarchical Scheduling

Furthermore, to preserve data locality optimizations of applications in NUMA systems and to minimize ULT scheduling overhead, we apply hierarchical scheduling (HS) where we partition the ULTs into multiple regions and performs distributed scheduling. The dynamic scheduling overhead becomes more and more critical with an increasing number of core resources. To reduce the overhead from scheduling contention, the hierarchical scheduling partitions the ULTs into multiple regions based on the CPU nodes. Each CPU node has its ULT region and schedules ULTs from the local region. Load balancing is also achieved through work stealing from the local queues of other CPU nodes. Then, a local work queue distributes ULTs to the threads in that node. The regions are equally partitioned according to OpenMP's *static* scheduling policy. Such a partitioning can be effective when neighboring work items exhibit high locality and preserve manual optimizations for static scheduling with a technique such as [63].

Balancing Malleability and Performance

The COOP-ULT technique executes parallel loops with a given number of ULTs. If we create more ULTs, the workload has more flexibility to adapt to the changing core allocation (because each ULT executes a smaller amount of work) but may suffer from a runtime overhead to schedule a large number of tasks. It is therefore important to create a proper number of ULTs to achieve both malleability and high performance.

Figure 2.4 shows the runtime performance (execution time) of NPB3.4 OpenMP Fortran applications [4] that are executed under the COOP-ULT technique with different numbers of ULTs. For the experiments, we used two multi-socket multicore platforms, the 64-core AMD (Opteron 6380 [1]) and the 72-core Intel (Xeon E7 8870 v3 [42]) system described in Section 1.2.2. In the figure, the first option creates as many ULTs as there are physical cores in the system; this leads to no malleability, but no scheduling overhead. The other options show the performance when increasing the number of ULTs. The results show that several applications such as `CG` and `MG` in AMD and `LU` in Intel suffer from a low performance when we create many ULTs due to the scheduling overhead. The overhead from scheduling ULTs becomes more prominent if the execution time of a ULT is small compared to the scheduling overhead. For example, in the case of `LU` in Intel, parallel loops execute significantly faster in Intel compared to our AMD platform, and the scheduling overhead became prominent. On the other hand, applications typically have significantly less performance degradation if we have a larger problem size (class D) because the scheduling overhead is less prominent compared to the execution time of each ULT.

Based on this experiment, to balance the malleability and performnace, the COOP-ULT technique currently creates $\#cores \times 20$ ULTs. Although it

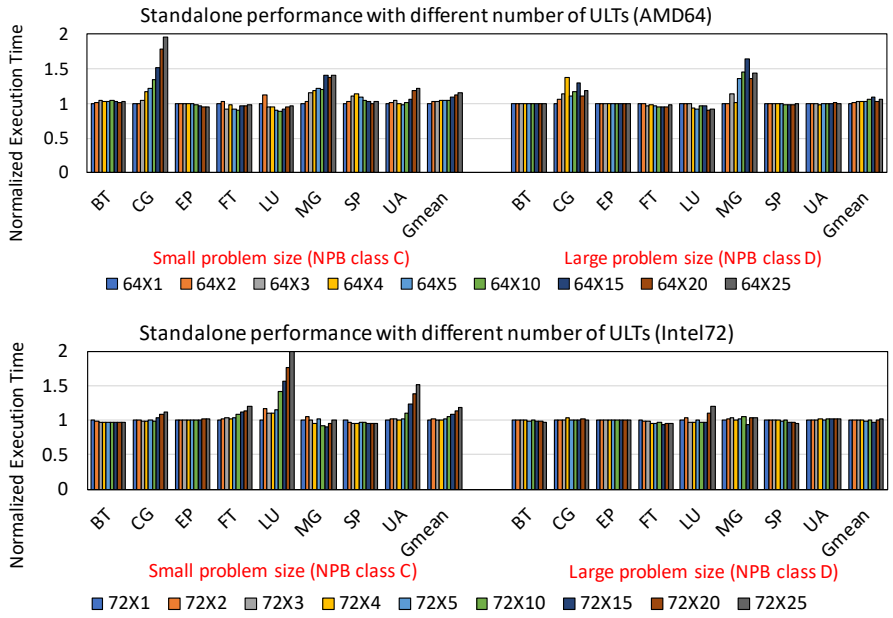


Figure 2.4 Application performance with different number of user-level tasks.

may still suffer from scheduling overhead for applications with a small problem size, applications are able to change their DoP with enough flexibility, and the scheduling overhead is generally negligible for applications with a sufficient problem size. Selecting the optimal number of ULTs (depending on the application) can improve the performance and flexibility, and is left as future work.

2.3.2 Cooperative Dynamic Loop Scheduling

As discussed in Section 1.2.1, OpenMP supports three loop scheduling methods: static, dynamic, and guided. These scheduling policies, by default, are not able to change the degree of parallelism at runtime. Static and guided scheduling are not malleable at all because they assign a comparatively large amount of work in the first assignment. Parallel loops with dynamic scheduling, on the other hand, may be adequate to achieve malleable execution by modifying the

Algorithm 1 Cooperative worker threads

```
1: while there is more work do
2:     if own core is not available then
3:         go to sleep
4:     for each thread ∈ worker_threads do
5:         if thread's core is available then
6:             wake up thread
7:     work_chunk ← get_work_chunk(chunk_size)
8:     if work_chunk received then
9:         work_chunk → execute( )
10:        if id == 0 and elapsed_time < epoch then
11:            chunk_size ← (chunk_size × 2)
```

loop scheduler. For example, the loop scheduler can dynamically assign the chunk of work onto the provided core resources. However, the dynamic scheduler can suffer from a significant dispatch overhead depending on the allocation granularity. In this section, we present the COOP-DYN technique that achieves malleable execution for OpenMP parallel loops with dynamic scheduling while optimizing the scheduling performance.

Achieving Malleability

We have implemented dynamic spatial scheduling into the GNU OpenMP runtime [34] by modifying the dynamic loop scheduling method in the OpenMP loop scheduler. To provide malleability while minimizing overhead and maximizing load balance, we use an adaptive dynamic scheduling technique as explained in Algorithm 1. A parallelism manager can keep track of the execution of OpenMP applications by intercepting calls that initiate or terminate par-

allel loops. The results of the core allocation are then communicated to the OpenMP parallel runtimes through shared memory. The runtimes dynamically change the DoP of cooperative parallel loops by adjusting the number of worker threads and pinning them to the assigned cores.

In the cooperative loop scheduling, before requesting new work, each thread checks the availability of its core. If the core is no longer available, the thread goes to sleep (lines 2–3). Active worker threads review the current core allocation and wake up threads whose core has become available (lines 5–6). Each thread acquires a chunk of work by calling the *get_work_chunk* function on line 7. To decrease the dispatch overhead, the master thread (id 0) dynamically adjusts the work chunk size based on the elapsed execution time of a work chunk (lines 12–13). The details of the work chunk size control are explained in the following section "Balancing Malleability and Performance".

Hierarchical Scheduling

Similar to the cooperative user-level tasking, here cooperative loop scheduling also support hierarchical scheduling for NUMA systems. The idea is basically the same with the cooperative user-level tasking. Instead of scheduling ULTs, here we partition the loop items into multiple regions for each CPU node and a local scheduler schedules items for cores in the same CPU node. Load balancing is also achieved through work stealing among CPU nodes.

Balancing Malleability and Performance

To provide malleability while minimizing overhead and maximizing load balance, we use an adaptive dynamic scheduler as illustrated in Algorithm 1. If the processing time of a work chunk is smaller than the global scheduling period of the space-sharing scheduler, we increase the chunk size. To provide sufficient op-

portunities for load balancing, the maximum chunk size is set to $\lceil W/2N \rceil$ where W represents the remaining iterations, N the number of available cores in the system. This is similar to the guided loop scheduling algorithm for multi-core systems [54].

The OpenMP runtime implements a work sharing approach in which each worker thread shares the data structure containing information about the processed and still unprocessed loop iterations. We designate one worker thread as the delegate thread that is allowed to change the work chunk size.

2.4 Experimental Results

In this section, we evaluate the cooperative runtime system on the 64-core AMD Opteron and the 72-core Intel Xeon system (Section 1.2.2).

2.4.1 Standalone Application Performance

The cooperative runtime system provides malleability by leveraging dynamic loop scheduling and user-level tasking. A concern is whether, despite its flexibility, the performance of the cooperative runtime system is on par with the existing schedulers. Here, we show that the cooperative runtime system can provide comparable performance to the default execution mode for standalone application execution while providing the ability to dynamically change the number of threads.

Figure 2.5 shows the turnaround times of standalone applications in NPB3.4 [4] OpenMP fortran applications executed with `COOP-ULT` compared to the default mode (`SMP Scheduling` and `Affinity Binding`). All these applications in this scenario use an input data size of class D [4]. In `SMP Scheduling` and `Affinity Binding`, applications generate worker threads with the number of system cores.

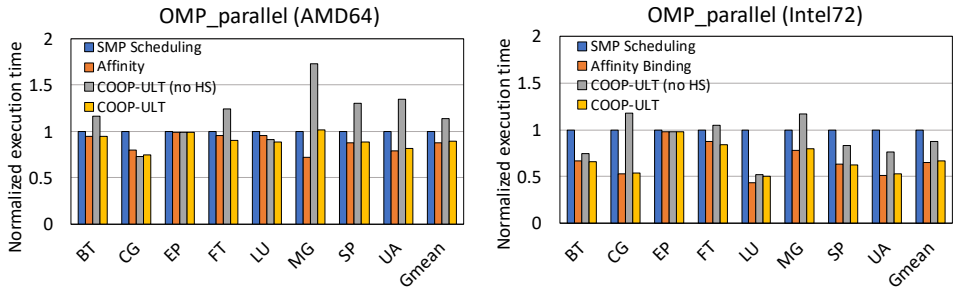


Figure 2.5 Standalone applications performance (OMP_parallel).

In `SMP Scheduling`, the generated threads are managed by Linux's default SMP scheduling (threads can execute on any core resource). In the `Affinity Binding` mode, on the other hand, each thread is bound to each core by managing their CPU affinity (only one thread can execute on each core). For these applications, cooperative user-level tasking performance is comparable to the default mode. Comparing `COOP-ULT` and `COOP-ULT (no HS)`, we observe that single global ULT scheduling without hierarchical scheduling (`COOP-ULT (no HS)`) incurs high scheduling overhead. The presented cooperative user-level tasking combines the best of both worlds by respecting data locality, yet being able to react to workload imbalance while also supporting malleable parallelism.

In Figure 2.6, we modify the NPB applications to use dynamic loop scheduling (by annotating "schedule(dynamic)" pragma) for all parallel loops in the application. The cooperative dynamic loop scheduling can also provide malleable execution for dynamic loops. In this scenario, we also observe that cooperative loop scheduling can even improve the single application performance a lot. For example, `COOP-DYN (no HS)` improves performance by managing the work chunk size dynamically. Since the original dynamic loop scheduler assigins

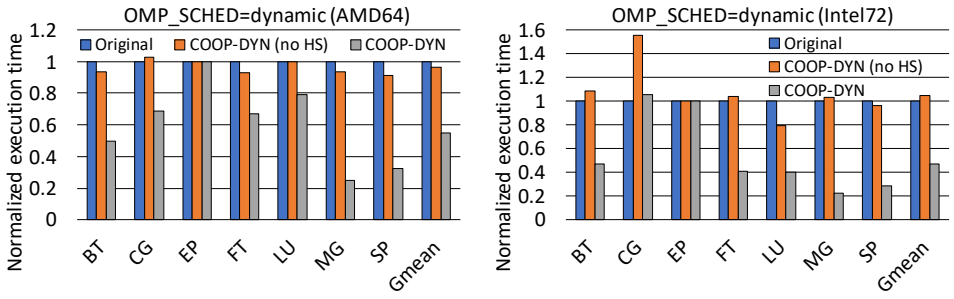


Figure 2.6 Standalone application performance (*parallel for*).

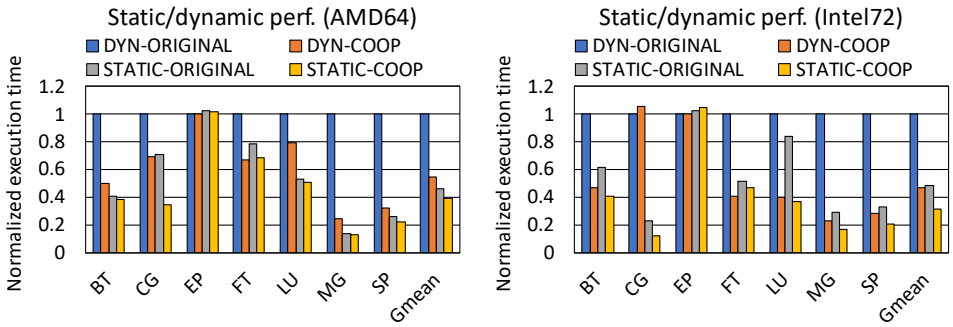


Figure 2.7 Performance under COOP-ULT and COOP-DYN.

works with a basic granularity of 1, it often incurs a high scheduling overhead for a large number of cores. In addition, the hierarchical scheduling and work stealing in `COOP-DYN` can further improve the performance.

Here, we compare the overall performance of static (cooperative user-level tasking) and dynamic versions (cooperative loop scheduling). Cooperative user-level tasking is beneficial than cooperative dynamic loop scheduling. The performance under loop scheduling generally depends on the application. While in our experiments we observe that static implementation provides overall higher

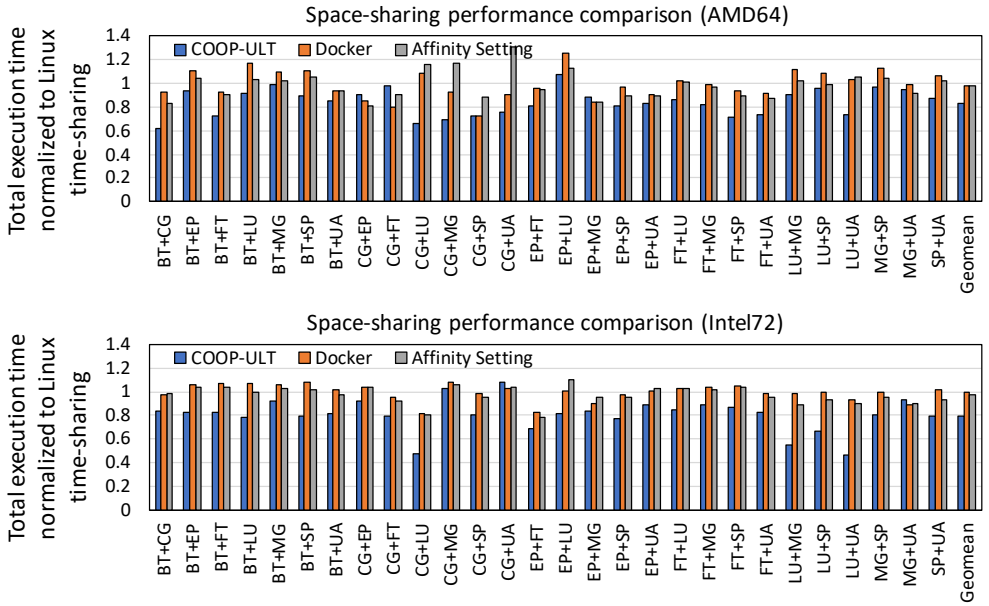


Figure 2.8 Space-sharing performance comparison with different tools.

performance compared to when using dynamic loop scheduling. Some applications such as EP and FT has benefit of dynamic loop scheduling. Because we mainly consider high-performance kernels and the workloads exhibit rather regular patterns. However, with cooperative runtimes we can also improve the performance of dynamic loops while providing flexibly to change the number of threads.

2.4.2 Performance in Spatial Core Allocation

We claim that, in spatial core allocation, `COOP-ULT` provides better performance than existing performance isolation tools that are not able to manage the number of threads dynamically. In Figure 2.8, we evaluate the performance (the total execution time) of two co-located parallel applications with spatial core

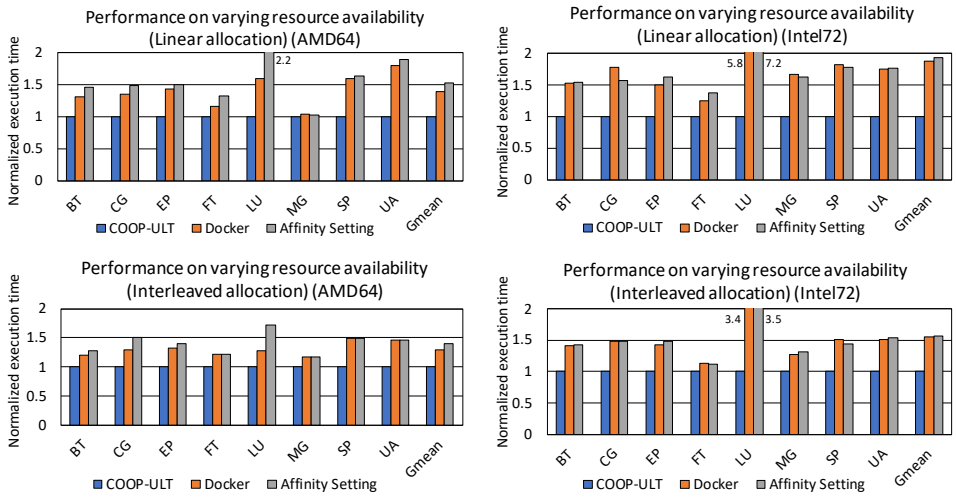


Figure 2.9 Performance under varying resource availability.

allocation using different resource isolation techniques. We evaluate the runtime performance under different resource isolation options, `COOP-ULT`, `Docker` and `Affinity Setting`. `COOP-ULT` represents our cooperative user-level tasking technique, and `Docker` uses its resource isolation based on Linux cgroup. To minimize other performance effect from using specific resource manager `Docker`, we have also implemented our specific tool based on a kernel module to manage CPU affinity for application's threads (`Affinity Setting`). For an application, the kernel module assigns CPU affinity for the application's spawned threads. All the worker threads of the application can be assigned to any core resource among the allocated core resources. The Linux SMP scheduler manages the threads on the given allocated core resources. We first evaluate the two application mixes performance with spatial core allocation (each application uses the same number of core resources; cores are allocated in an interleaved way).

To show the potential benefit of malleable execution, we now consider extreme scenarios where core allocation changes frequently. Figure 2.9 shows the

performance (the total execution time) of applications when the core allocation frequently changes (switch full cores and half cores for every second using linear/interleaved allocation). The results show that MOCA's cooperative scheduling can improve performance (about 30–50% on average) compared to the Docker resource isolation tool because the resource isolation tool using Linux's `cgroup` is not able to change the number of threads. Moreover, without cooperative scheduling, resource isolation requires runtime overhead for thread migration.

2.5 Discussion

2.5.1 Contributions

The presented cooperative runtime system enables malleable parallel execution for OpenMP applications with high performance through efficient user-level tasking and dynamic scheduling. Compared to other runtime techniques to provide malleability [53, 71, 72, 81, 40], our runtime system makes several contributions to the parallel computing community.

First, our runtime technique is highly applicable; our technique enables malleable execution for modern OpenMP applications with a small patch to the OpenMP runtime system. A number of previous auto-tuning/auto-parallelization researches such as DoPE [71] and Parcae [72] have shown how the compiler/runtime can achieve malleable execution. However, ther works are based on their specific compiler and runtime framework, which limits their applicability for generic parallel applications. Our research provides a novel solution to achieve malleable execution in the OpenMP runtime through user-level tasking and dynamic scheduling.

Second, we present an efficient task/loop scheduling technique that achieves

both malleability and high performance. Varuna [81] and Callisto [40] are conceptually similar to our cooperative runtime system and provide malleable execution for generic applications such as TBB and OpenMP. Varuna splits the parallel work by intercepting creation of Pthreads and manages/schedules them using a work pool. Similar to our work, Callisto uses user-level tasking to abstract the OpenMP workloads and achieves malleable execution through dynamic scheduling. However, Varuna and Callisto use a simplistic task scheduling method; their scheduler allocates a task to an idle core resource using a single global work pool. As we have seen in our evaluation (Section 2.4.1), this global scheduling approach suffers from a significant scheduling overhead on a large number of cores. Our hierarchical scheduling and the granularity control policy are key to provide higher performance compared to the simplistic scheduling.

2.5.2 Limitations and Future Work

The current cooperative runtime system implementation has several limitations, and we left these as future work.

First, as we discussed in Section 2.3.1, the COOP-ULT technique has a performance issue depending on the number of ULTs. Our current simple policy achieves an acceptable performance for OpenMP applications while providing malleability. One future work is selecting the optimal number of ULTs (depending on the application) can improve the performance and flexibility.

Second, the cooperative runtime system provides malleable execution at the level of parallel loop annotated with static/dynamic loop scheduling. This can support most of existing OpenMP benchmark applications, however, recent OpenMP runtimes provide task parallelism and nested parallelism. Our runtime technique does not support such a new parallelism type. Another interesting future work is to extend the idea of cooperative runtime and enable malleable

execution for these parallelism.

2.5.3 Summary

In this section, we have shown how the OpenMP runtime can achieve efficient dynamic spatial scheduling for simultaneously running multiple parallel applications. The proposed techniques require small modifications to the existing OpenMP parallel runtime system.

Based on the cooperative runtime system, researchers can design and implement their resource allocation techniques for single and co-located parallel applications. This technique provides the basic execution environment for our extensive study in the following chapters.

Chapter 3

Performance Modeling of Parallel Loops using Queueing Systems

3.1 Overview

In this chapter, we introduce an analytical performance model for parallel loops. Parallel loops such as OpenMP's *parallel for* [23] are the basic parallel programming construct on shared-memory platforms and an important target for optimizations because these parallel loops dominate the execution time of many scientific applications. Moreover, as discussed in Chapter 2, such parallel loops are now able to run with a configurable number of worker threads based on malleable parallel runtime systems such as COOP-ULT and COOP-DYN presented in Chapter 2. Modeling performance of parallel loops in dependence of the number of allocated cores/threads therefore has been an important research issue to maximize the performance or to meet a certain optimization goal.

Our approach to modeling performance of parallel programs employs queueing models. Queueing models are powerful analytical tools based on stochastic

processes to evaluate the performance of queueing systems such as the mean waiting time, the queue length, and the server utilization [83]. Several existing performance models [85, 84, 17] predict the performance scalability of parallel programs by computing the mean response time of memory requests for a varying number of threads using a queueing system. 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.

Applying queueing models to modern multicore systems in practice, however, remains a challenge. Large shared memory systems, called multi-socket multicore 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. Previous techniques [85, 84, 17] based on simple queueing systems 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.

In this chapter, we present a practical approach to model performance of parallel for loops on multi-socket multicore systems using queueing sys-

tems. First, runs of OpenMP parallel loops on real systems confirm that the *M/M/1/N/N* queueing model [83] 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 approach 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 multicore systems.

To summarize, we make 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.

- A methodology to model memory system performance on multi-socket multicore platforms using a hierarchical queueing system.

- A speedup model that is able to predict the speedup of OpenMP parallel loops based on the queueing system.
- An evaluation of the presented speedup model for 24 OpenMP parallel loops on an AMD and an Intel multi-socket multicore platform.

3.2 Background

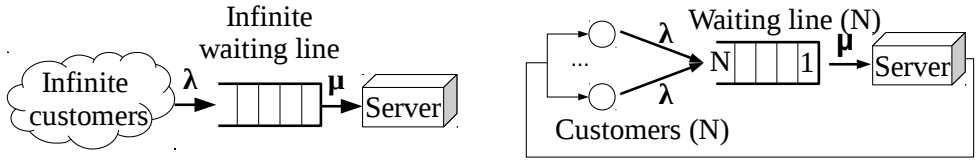
3.2.1 Queueing Models

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 [6]. They are also well-suited to analytically model the performance of shared resources such as memory controllers [47, 85, 17] 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 [83, 39].

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 Figure 3.1 (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.

$$r = \frac{1}{\mu - \lambda} \tag{3.1}$$



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

Figure 3.1 Illustration of the *M/M/1* and *M/M/1/N/N* 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.

Previous research [85, 84] often employed the *M/M/1* queueing model to model memory performance on multicores where cores are considered to be queueing customers. In multicore 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.

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 Figure 3.1 (b). The requests of the customers follow a Poisson distribution, and the server has an exponential service time. 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 3.2.

$$r = \frac{1}{\mu}(\frac{N}{U_s} - \frac{\mu}{\lambda}) \tag{3.2}$$

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

$$U_s = 1 - \left(\sum_{k=0}^{N} \frac{N!}{(N-k)!} (\frac{\lambda}{\mu})^k \right)^{-1} \tag{3.3}$$

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

3.2.2 Insights on Performance Modeling of Parallel Loops

This section discusses our key insights under which *M/M/1/N/N* queueing models can be applied to performance modeling of multi-socket architectures.

Queueing Models and Multi-Socket Systems

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 multicore systems (Section 1.2.2), 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 3.2.3.
- Processor cores execute instructions out-of-order and can issue several memory requests. In addition, requests can be reordered by caches and

memory controllers [51, 67]. 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 3.3 details this approach.

Parallel Loops

As discussed in Chapter 1, we mainly target OpenMP parallel loops. Table 3.1 shows the parallel loops used for performance modeling and evaluation throughout this chapter. The loops were obtained from the NAS parallel benchmark suite (NPB) [4, 80] containing HPC workloads and two OpenMP applications from the Parsec benchmark suite [7], 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. All benchmarks in this chapter are run with the *interleaved* NUMA memory allocation policy. In total, 24 different parallel loops are selected from the seven parallel applications.

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

Table 3.1 Selected parallel loops.

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 3.2.3 and 3.2.3.
- 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 3.1) do not have dependencies. Experiments in Section 3.2.3 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.

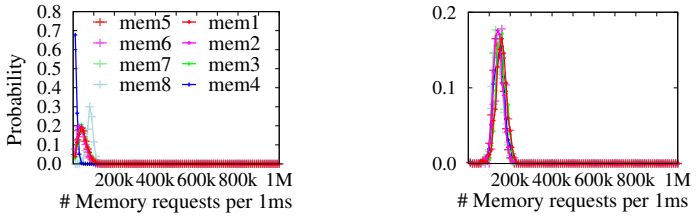
- Parallel loops are dynamically scheduled (refer to Chapter 2) because this policy allows runtime systems to dynamically adjust the number of threads.

3.2.3 Performance Analysis

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 our target multi-socket systems: the 64-core AMD system and the 72-core Intel system (Section 1.2.2).

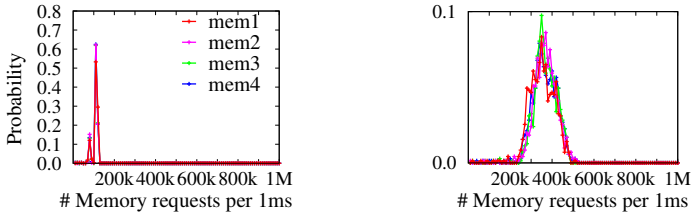
Memory Access Distribution

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 Figure 3.2 and Figure 3.3 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 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*. Appendix A contains the PMF of all targeted parallel loops on both architectures as well as the results of the two-sample Kolmogorov-Smirnov (KS) test [77] confirming that the majority of the loops follows a Poisson distribution.



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

Figure 3.2 PMF of the number of memory requests per time for each memory node on the 64-core AMD platform.



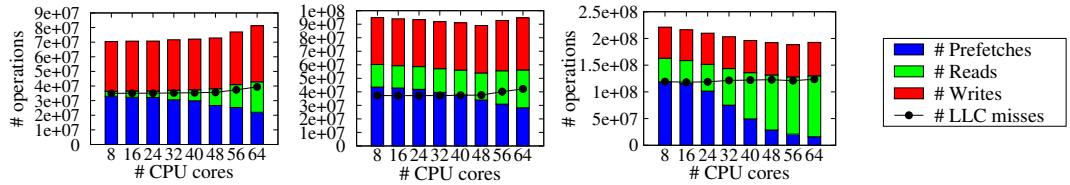
(a) x_solve (BT) (b) x_solve (SP)

Figure 3.3 PMF of the number of memory requests per time for each memory node on the 72-core Intel platform.

Memory Access Pattern

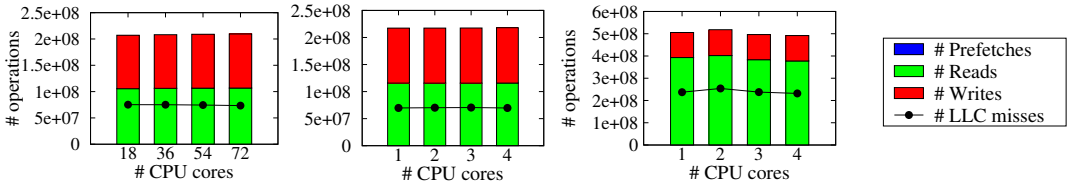
Modern memory systems perform optimizations such as memory prefetch operations that can cause a variation in the memory access pattern. Figure 3.4 and Figure 3.5 show the number of memory operations collected from hardware performance counters for the three parallel loops *cffts1–3* of *FT* with a varying number of worker threads on the AMD and the Intel system.

The memory access pattern varies for different workloads and the number of worker threads. For example, in Figure 3.4 (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 Figure 3.4 (a), on the other hand, the number of memory operations increases for a larger number of threads.



(a) cffts1 (FT) (b) cffts2 (FT) (c) cffts3 (FT)

Figure 3.4 Number of memory operations of parallel loops for a varying number of worker threads on the AMD platform.



(a) cffts1 (FT) (b) cffts2 (FT) (c) cffts3 (FT)

Figure 3.5 Number of memory operations of parallel loops for a varying number of worker threads on the Intel platform. Our Intel platform does not support measuring the number of prefetches (# Reads includes # Prefetches).

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.

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 syn-

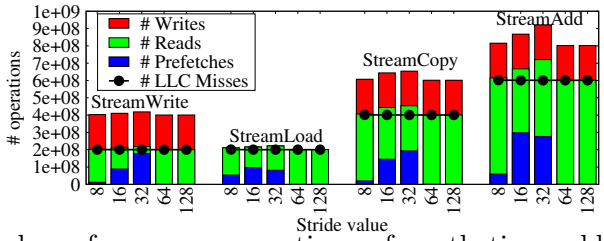


Figure 3.6 Number of memory operations of synthetic workloads (using one CPU thread) with different stride values on the AMD platform.

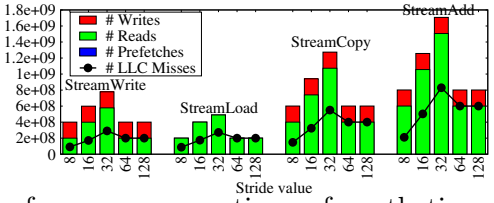


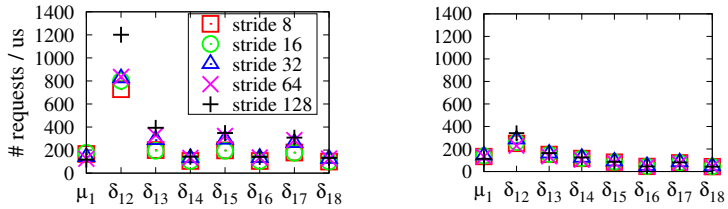
Figure 3.7 Number of memory operations of synthetic workloads (using one CPU thread) with different stride values on the Intel platform. In our Intel platform, # Reads includes # Prefetches.

thetic workloads *StreamWrite*, *StreamLoad*, *StreamCopy*, and *StreamAdd* from the Stream benchmark suite [64]. The following code shows the *StreamWrite* workload that writes a scalar value to the elements of an array.

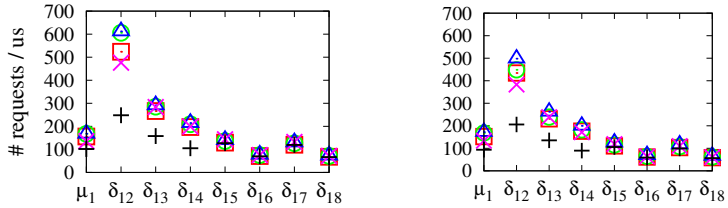
```
1:  for (int i=0; i<stride; i++)
2:    for (int j=i; j<arr_size; j+=stride)
3:      A[j] = scalar;
```

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.

Figures 3.6 and 3.7 show the number of memory operations of the synthetic workloads for varying `stride` values, on the AMD and the Intel system, re-

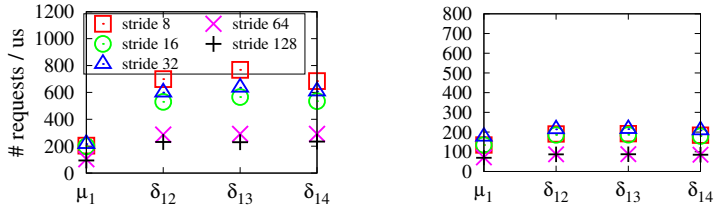


(a) StreamWrite (b) StreamLoad

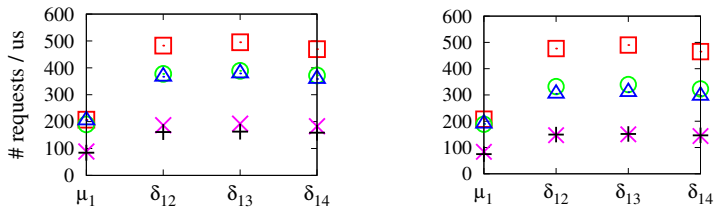


(c) StreamCopy (d) StreamAdd

Figure 3.8 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.



(a) StreamWrite (b) StreamLoad



(c) StreamCopy (d) StreamAdd

Figure 3.9 Measured service rates for the synthetic workloads on the Intel platform for varying stride values.

spectively. The values are set such that the workload is completely memory bound. Similar to the observation from Figure 3.4 and Figure 3.5, different synthetic workloads in Figure 3.6 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

$$\mu_j = \frac{\#\ total\ memory\ operations}{total\ execution\ time_{jj}} \tag{3.4}$$

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.

$$\delta_{ij} = \frac{\#\ total\ memory\ operations}{(total\ execution\ time_{ij} - total\ execution\ time_{jj})} \tag{3.5}$$

Figure 3.8 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 Figure 3.6. Figure 3.9 shows the measured service rate and the interconnection links between CPU node 1 and four memory nodes on the Intel system. We observe that the memory service rate depends on the workload. For example, comparing *StreamWrite* (Figure 3.8 (a)) and *StreamLoad* (Figure 3.8 (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,

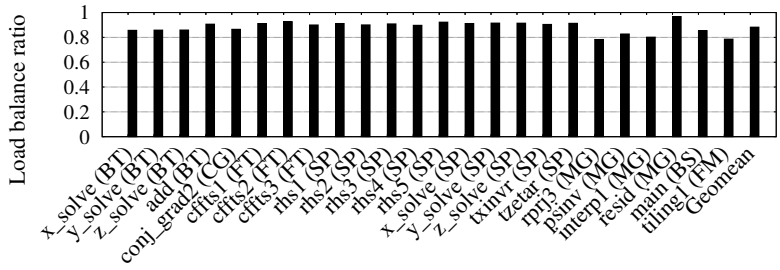
in *StreamCopy* (Figure 3.8 (c)) and *StreamAdd* (Figure 3.8 (d)), the service rates tend to be higher with a stride value of 32 because there are a larger number of memory prefetch operations as visible in Figure 3.6. The experiments demonstrate that it is necessary to consider the memory access pattern 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 3.2.3, this assumption is justified using the KS test for the synthetic workloads. The details are provided in Appendix A.1.2.

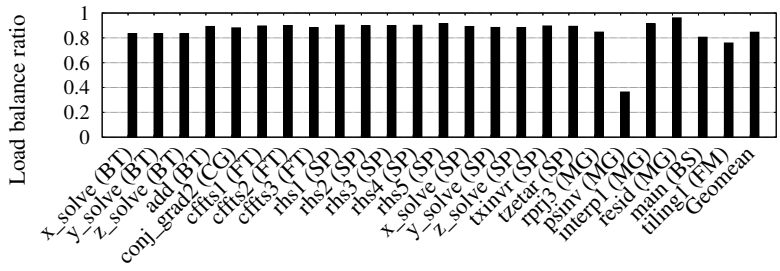
Synchronization Overhead

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. 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 ($\textit{Load balance ratio} = t_{shortest}/t_{longest}$).

Figure 3.10 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.



(a) 64-core AMD platform.



(b) 72-core Intel platform.

Figure 3.10 Measured load balancing ratio of the parallel loops.

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.

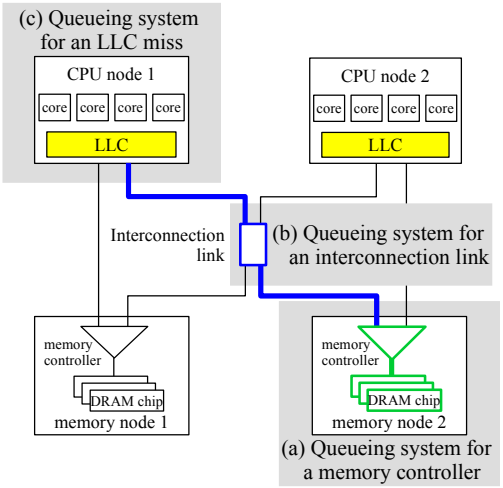


Figure 3.11 A two-socket multicore system and the data path for an LLC miss of CPU node 1 to be served by memory node 2.

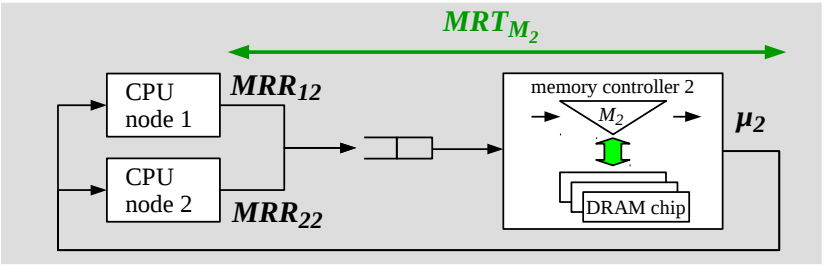
3.3 Queueing Systems for Multi-Socket Multicores

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

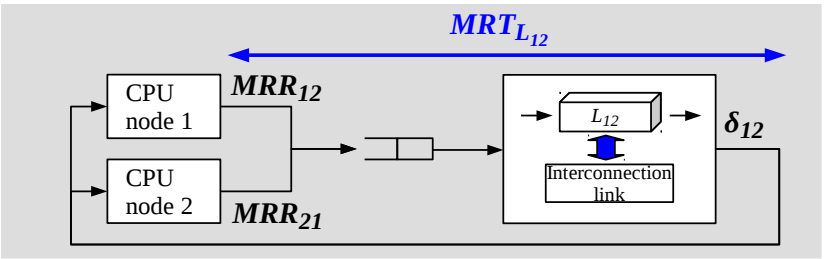
3.3.1 Hierarchical Queueing Systems

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.

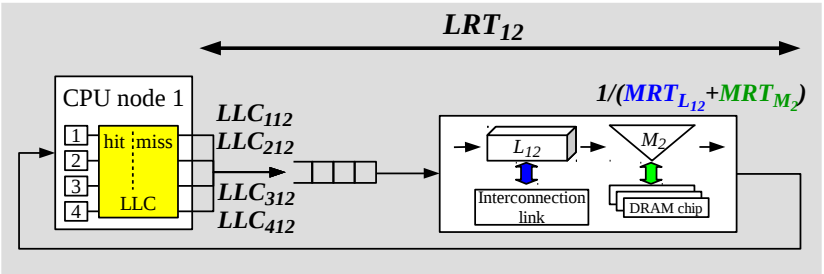
A multi-socket system with two CPU nodes and two memory nodes as shown in Figurere 3.11 is used for the explanations. Each CPU node has four cores and an LLC. The shaded boxes in the figure depict the contention points ob-



(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

Figure 3.12 The hierarchical queueing systems for the data path.

served 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 Figure 3.11 (c). The contention at the memory controller and the interconnection network is considered by the queueing systems shown in Figure 3.11 (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.

Figure 3.12 depicts these queueing systems. The input parameters of the queueing systems and the modeled performance are described in Table 3.2 and Table 3.3, respectively. Details of each model are presented in the following sections.

Queueing System for Memory Controllers

Figure 3.12 (a) shows the queueing system for memory controller 2 (M_2) of the two-node system from Figure 3.11. 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 3.2). λ_{M_2} in Equation 3.2 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 3.2 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

$$\lambda_{M_j} = \frac{\sum_{i=1}^{N} MRR_{ij}}{N} \tag{3.6}$$

and the service rate μ_j, Equation 3.2 computes the mean response time of memory controller j, MRT_{M_j}.

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 Figure 3.11, serves requests from CPU node 1 to memory node 2 and from CPU node 2 to memory node 1. The queueing system, shown in Figure 3.12 (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 3.2 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

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 3.2 Input parameters of 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

Table 3.3 Modeled performance information from the queueing systems.

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

$$\lambda_{L_{ij}} = \frac{\sum_{l=1}^{N} \sum_{k \in LSet_{ij}^{l}} MRR_{lk}}{N} \tag{3.7}$$

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 3.2, we can compute the mean response time of the interconnection link $MRT_{L_{ij}}$.

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

$$TRT_{ij} = MRT_{M_j} + MRT_{L_{ij}} \tag{3.8}$$

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. Figure 3.12 (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 3.3). 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 3.4.

3.3.2 Computing the Parameter Values

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 [2] and Intel's uncore events [44] 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 3.2, 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.

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.

$$MRR_{ij} = \frac{\#\ Requests_{ij}}{CPU\ Time} \tag{3.9}$$

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.

$$LLC_{kij} = \frac{\#\ LLC\ Miss_{kij}}{CPU\ Time} \tag{3.10}$$

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.

$$Total\ Time = CPU\ Time + \sum_{j=1}^{N}(\sum_{k=1}^{C} \#\ LLC\ Miss_{kij} \cdot LRT_{ij}) \tag{3.11}$$

where C represents the number of cores in a CPU node, and LRT_{ij} is computed from the queueing system given in Section 3.3.1. Solving Equation 3.11 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 3.12.

$$CPU\ Time^{k+1} = Total\ Time - \sum_{j=1}^{N}(\sum_{k=1}^{C} \#\ LLC\ Miss_{kij} \cdot LRT_{ij}^{k}) \tag{3.12}$$

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 3.2.3, 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 3.4.2.

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 3.2.2, 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 3.13 computes the mean service time for memory controller j.

$$MST_{\mu_j} = \alpha_{\mu_j} \cdot \frac{\#\ Prefetches}{\#\ Requests} + \beta_{\mu_j} \cdot \frac{\#\ Reads}{\#\ Requests} + \gamma_{\mu_j} \cdot \frac{\#\ Writes}{\#\ Requests} \quad (3.13)$$

To compute the coefficient values of α_{μ_j}, β_{μ_j}, and γ_{μ_j}, the four synthetic workloads from Section 3.2.3 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 work-

loads can increase the accuracy of the model but is left for future work.

3.4 The Speedup Prediction Model

This section presents the speedup prediction model using the presented queueing systems in Section 3.3 for parallel loops.

3.4.1 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. $CPU\ Time_S$ 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

$$S(M) = \frac{CPU\ Time_S/C + Stall\ Time(1)}{CPU\ Time_S/(M \cdot C) + Stall\ Time(M)} \tag{3.14}$$

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

$$Stall\ Time(M) = \sum_{j=1}^{N} \Big(\frac{CPU\ Time_S}{M \cdot C} \cdot LLC_j \cdot \frac{\sum_{i=1}^{M} LRT_{ij}}{M} \Big) \tag{3.15}$$

where the value of LRT_{ij} is computed by applying the $M/M/1/N/N$ queueing systems from Section 3.3. 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.

3.4.2 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*.

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.

LoopPerf-T (Twice)

As discussed in Section 3.2.3, 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.

Best-F (Best Fixed parameter values)

Best-F from our previous work [17] employs simpler *M/M/1/N/N* queueing systems to model the speedup of parallel workloads and does consider 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 3.1. *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.

3.5 Evaluation

This section evaluates the presented speedup prediction model with the 24 parallel loops from Table 3.1 on two NUMA architectures, an AMD 64-core and and Intel 72-core platform. Details of the platforms are given in Section 1.2.2.

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

$$MAPE = \frac{100\%}{n} \sum_{k=1}^{n} \left| \frac{a_k - p_k}{a_k} \right| \tag{3.16}$$

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 Figure 3.14 (AMD) and Figure 3.15 (Intel).

3.5.1 64-core AMD Opteron Platform

The results in Figure 3.14 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 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 Figure 3.14 *(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 Figure 3.2 (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 com-

pared to *LoopPerf-S* and *Best-F*. For example, looking at *(5) conj_grad2*, *(8) cfffts3*, *(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 3.2.2, we have shown experimentally that the memory accesses of these workloads do not follow a Poisson distribution (Figure 3.2 (c)) and that these workloads also suffer from a load imbalance in Figure 3.10 (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 multicore platforms. Despite the higher error rates for pathological curves from irregular workloads, the high accuracy of the

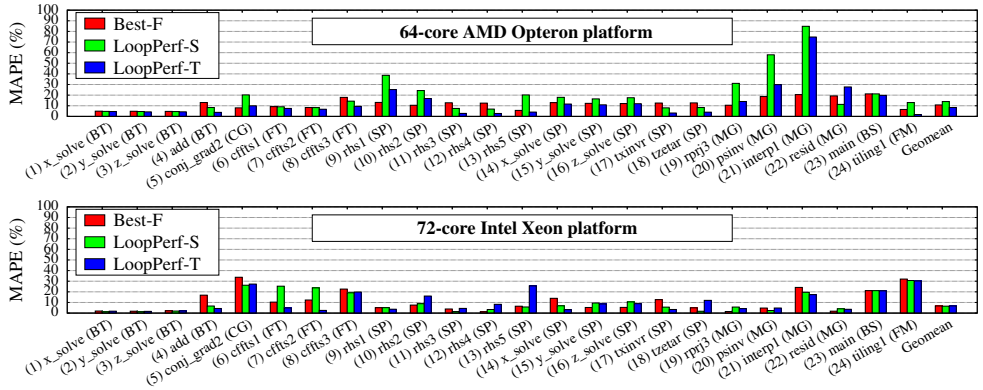


Figure 3.13 MAPE of the predicted speedup compared to the measured speedup of the parallel loops on the 64-core AMD and the 72-core Intel platform.

prediction technique for regular workloads makes the presented queueing system a good candidate for performance modeling and optimization in multi-socket multicore systems.

3.5.2 72-core Intel Xeon Platform

Figure 3.15 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 3.2.2.

In Figure 3.15, 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 Figure 3.14. *LoopPerf-T* provides good predictions for memory-

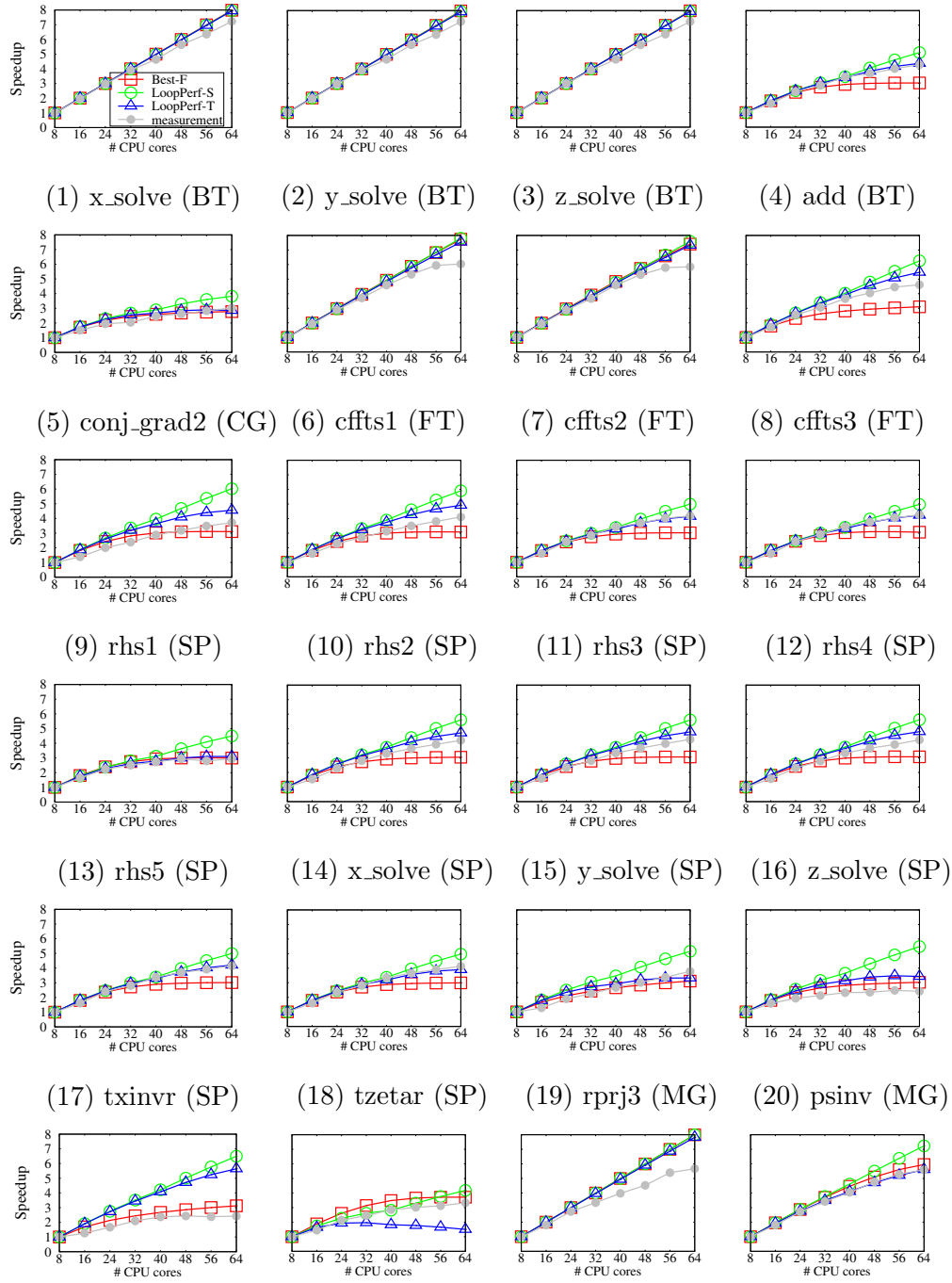


Figure 3.14 The predicted speedup and the measured speedup of the parallel loops on the 64-core AMD platform.

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.

3.6 Discussion

3.6.1 Applicability of the Model

The evaluation of the presented performance model in this chapter mostly assumed (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 [72, 15]. We argue that the model is also applicable for other types of loop scheduling and data distribution policies, and we provide some additional experimental results in Appendix A.

First, the model can support other types of loop scheduling. The dynamic loop scheduling scheme considered in this chapter is mostly close to the execution under COOP-DYN (that uses also dynamic loop scheduling) in Chapter 2. The COOP-ULT technique presented in Chapter 2 also enables malleable execution for parallel loops with the static loop scheduling mode by dynamically scheduling the work abstracted with user-level tasks. In principle, since the execution under both COOP-DYN and COOP-ULT is also based on dynamic scheduling, our model can work for both COOP-DYN and COOP-ULT. Appendix A.2.2 validate this claim by providing some additional experiments that

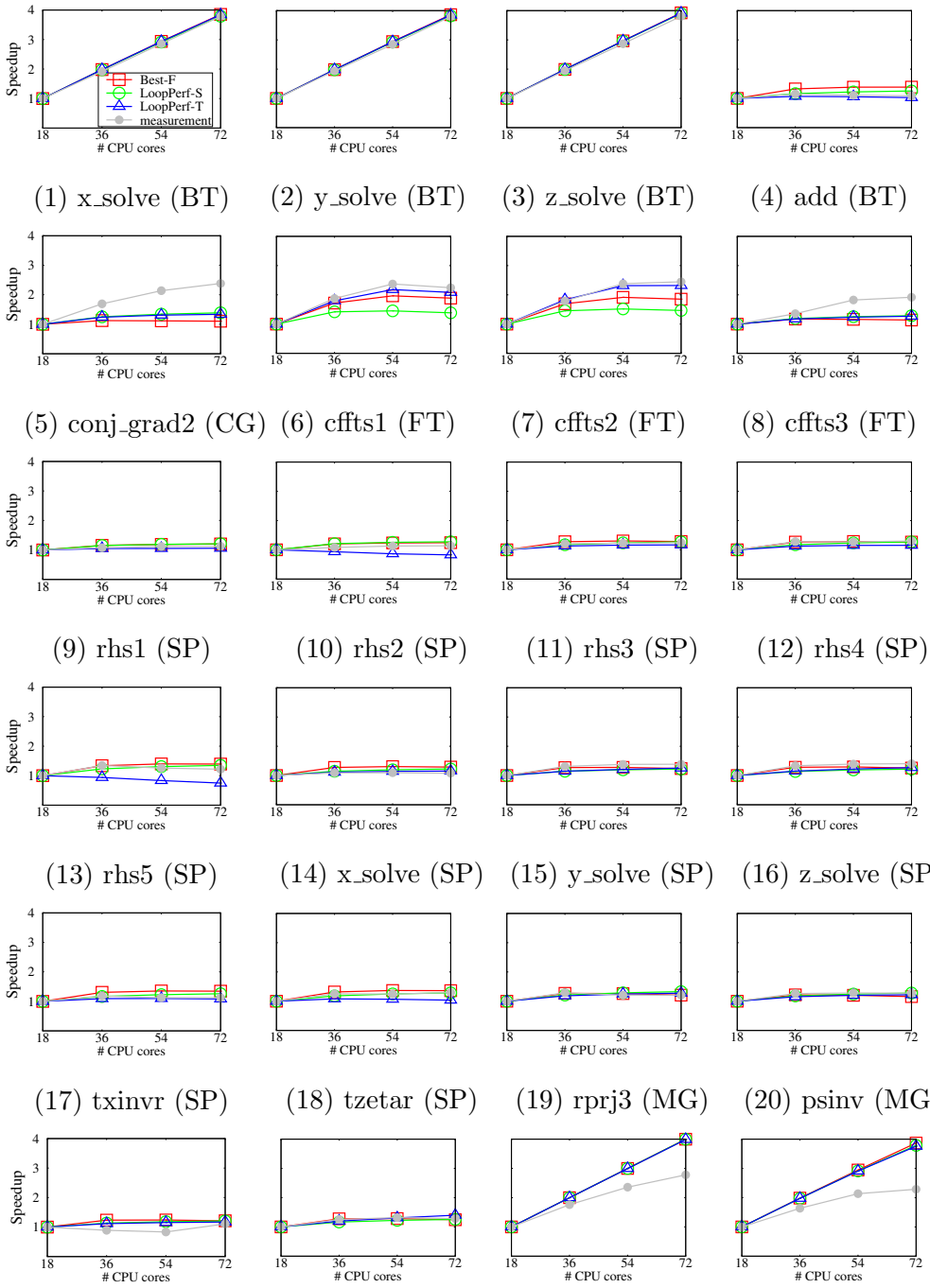


Figure 3.15 The predicted speedup and the measured speedup of the parallel loops on the 72-core Intel platform.

evaluate the accuracy of the performance model for static parallel loops executed with COOP-ULT. The results show that our model can also predict the performance of parallel loops with COOP-ULT.

In addition, in our previous PACT 2016 paper [17], the model based on the *Best-F* method has also been evaluated for other scheduling methods including static, guided and dynamic scheduling as well as for different memory allocation schemes. The results have shown 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. For these results, please see Appendix A.2.3 and A.2.4.

3.6.2 Limitations of the Model

The presented analytical approach makes several assumptions. 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 [69] 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 3.2.2, 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 [78] 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.

3.6.3 Summary

In this chapter, we presented a methodology to model the memory system performance of multi-socket multicore 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. The following chapter introduces a runtime-level parallelism management technique for co-located parallel applications by leveraging the analytical model presented in this chapter.

Chapter 4

Maximizing System Utilization via Parallelism Management

4.1 Overview

In this chapter, we focus on managing parallelism of co-located parallel workloads by leveraging the cooperative parallel runtime support (Chapter 2) and the analytical model for performance estimation (Chapter 3). The parallelism management aims to fully utilize system resources and therefore to achieve an increased co-location performance (i.e. reduction of the total execution time) for shared-memory multiprocessor systems consisting of multiple CPU sockets and memory controllers with NUMA latencies.

Existing work typically assigns more worker threads to computation-intensive applications [65, 76, 75, 21]. This can lead to under-utilized memory systems resulting in inefficient tail execution once the computation-intensive applications have finished. In contrast, the method proposed in this chapter aims at maximizing the overall utilization of both the CPU cores and the memory system.

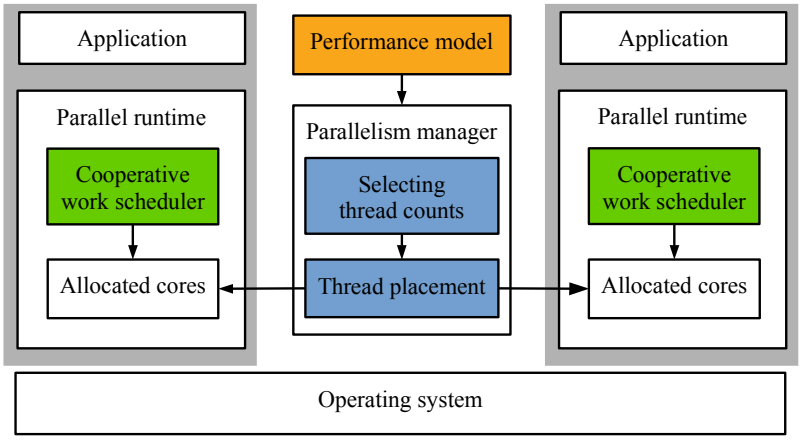


Figure 4.1 The NuPoCo framework.

To this end, this chapter presents NuPoCo, a framework for **NU**MA multi-core **P**erformance **O**ptimization of **CO**-located parallel applications. NuPoCo maximizes the overall system utilization by considering the utilization of multi-dimensional resources such as CPU cores and memory bandwidth on memory controllers to determine the proper number of threads for each co-located parallel loops. Figure 4.1 depicts the structure of the NuPoCo framework. The three core components are (1) a performance model, (2) a parallelism manager, and (3) cooperative loop schedulers of parallel runtime systems. The performance model is based on the model presented in Chapter 3 and predicts the utilization of CPU cores and memory controllers for co-located parallel applications. Predicting utilization is based on a queueing system that models memory accesses on multi-socket multicore systems. The parallelism manager periodically performs core allocation (i.e., deciding on the number of threads per application and their location) by leveraging the performance model and monitoring hardware performance counters. The cooperative loop schedulers (Chapter 2), finally, dynamically adapt their execution to the core allocation dictated by the

parallelism manager.

We evaluate NuPoCo on two multi-socket multicore platforms, the 64-core AMD Opteron and the 72-core Intel Xeon platform (Section 1.2.2). Experimental results for various workload mixes obtained from NPB [4], Parsec [7], and Rodinia [14] show that NuPoCo is able to execute multiple OpenMP applications in significantly less total execution time compared to the default Linux scheduler and a parallelism management scheme maximizing CPU utilization. The NuPoCo framework and the results are presented in PACT 2018 [15].

NuPoCo currently uses a simple greedy algorithm for parallelism management. In other words, for each core resource (or an allocation unit) we assign the core resource to an application that maximizes the summation of CPU utilization and memory bandwidth. In Section 4.5, we also present MOCA (**M**ulti-**O**bjective **C**ore **A**llocation), an ongoing research on the core allocation problem. MOCA employs evolutionary meta-heuristics inspired by genetic algorithms (GAs) [50, 31] for core allocation to find a better allocation than the greedy algorithm. We provide some experimental results of MOCA in Section 4.6.

4.2 Background

This section provides background information about the performance property of parallel loops. Then, we compare the performance of core allocation policies through queueing theory.

4.2.1 Modeling Performance Metrics

Queueing models are able to compute important performance-related metrics such as the CPU utilization or the memory controller utilization. Let us first

consider a simple SMP system with one memory controller (MCT) and 16 cores (Figure 1.3 (a) in Section 1.2.2 shows the architecture diagram). Such a system can be modeled using an *M/M/1/N/N* queueing system [83] with a finite number of N customers and 1 server. We presented the details about the background information of the *M/M/1/N/N* queueing model in Section 3.2.1. Please refer to Section 3.2.1 for the details. Employing the queueing model, the CPU cores are regarded as the queueing customers, and the memory system is considered the queueing server. The N queueing customers (the cores) each generate requests with a *mean arrival rate* λ following a Poisson distribution that are served by one queueing server (the memory controller) with a *mean service rate* μ with exponential service times.

Based on this queueing model, the speedup, the per-core utilization, and the MCT utilization in dependence of the number of allocated cores can be derived as follows. The *Speedup* of a program is defined by dividing the execution time on one core, *Total Time(1)*, by the execution time on N cores, *Total Time(N)*

$$Speedup\,(N) = Total\ Time\,(1)/\,Total\ Time\,(N) \tag{4.1}$$

Under the assumption that cores block on outstanding memory requests, *Total Time(N)* is composed of the execution time on N cores, *CPU Time(N)*, and the total memory response time, *MCT Time(N)*

$$Total\ Time\,(N) = CPU\ Time\,(N) + MCT\ Time\,(N)$$

For data-parallel workloads where the total amount of work is constant and balanced, the execution time on N cores is given by

$$CPU\ Time\,(N) = CPU\ Time\,(1)/N$$

The estimated number of generated memory requests for N cores is the product of the CPU time and the per-core memory request rate, *MRR*. *MCT Time(N)*

is obtained by multiplying *MRR* with the mean memory response time for N cores, *MRT(N)*.

$$MCT\ Time\,(N) = CPU\ Time\,(N) \times MRR \times MRT\,(N) \tag{4.2}$$

Each application has its own *MRR* value, and the *MRT* for a varying number of cores is regarded as the scaling factor of the parallel application.

For a memory controller with a service rate μ, the mean memory response time is given by (refer to the closed-form equation of the *M/M/1/N/N* model (Equation 3.2 in Section 3.2.1)).

$$MRT\,(N) = \frac{1}{\mu}\left(\frac{N}{MCT\ Util\,(N)} - \frac{\mu}{MRR}\right) \tag{4.3}$$

where *MCT Util(N)* denotes the memory controller utilization corresponding to the server utilization U_s from the closed-form equation (Equation 3.3 in Section 3.2.1)

$$MCT\ Util(N) = 1 - \left(\sum_{k=0}^{N} \frac{N!}{(N-k)!}(\frac{MRR}{\mu})^k\right)^{-1} \tag{4.4}$$

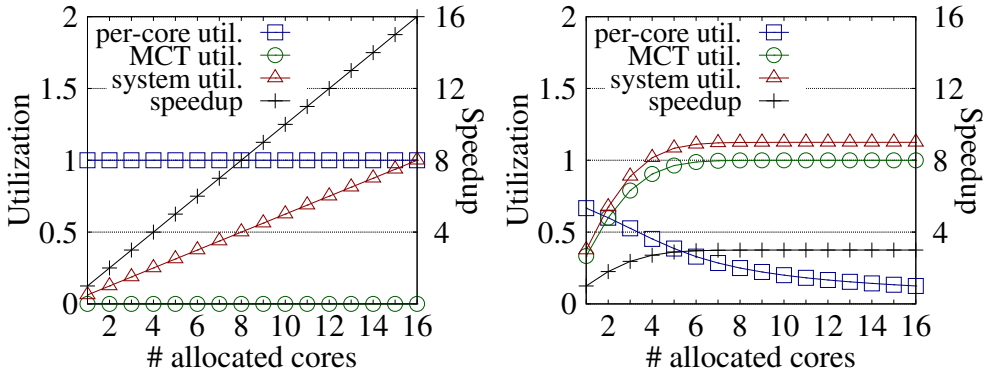
Finally, the per-core utilization, *CPU Util(N)*, is defined by the ratio of CPU time over the total time

$$CPU\ Util\,(N) = \frac{CPU\ Time\,(N)}{Total\ Time\,(N)} \tag{4.5}$$

To consider overall utilization of both the CPU and the memory controller, we suggest a new metric, the *system utilization*, defined as the sum of CPU and MCT utilization. For an application using N of the total M system cores, the system utilization, *System Util(N)*, is defined as

$$System\ Util\,(N) = CPU\ Util\,(N) \times \frac{N}{M} + MCT\ Util\,(N) \tag{4.6}$$

To solve this model for co-located applications, the weighted average of the workloads' *MRR* of all assigned cores is used to compute the mean memory



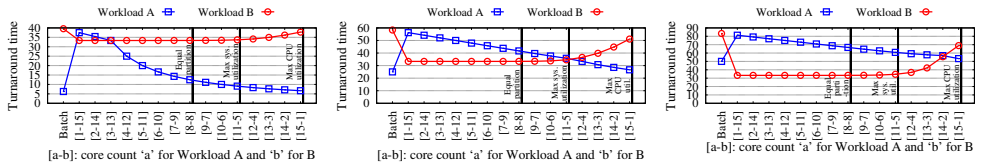
(a) Workload A ($MRR = 0.0$) (b) Workload B ($MRR = 0.01$)

Figure 4.2 Performance metrics for two workloads with different *MRRs* at a mean service rate μ of 50.

response time and the memory controller utilization (Equations 4.3 and 4.4). Based on the computed *MRT* value, we compute the per-application *Speedup* and *CPU Util* using the application-specific *MRR* value.

4.2.2 Our Resource Management Policy

Figure 4.2 plots the analytical results of the presented model for the four metrics *Speedup*, *MCT Util*, *CPU Util*, and *System Util* for two workloads and a varying number of cores. The results show that the completely CPU-bound workload A is able to fully utilize the given CPU resources, but its *MCT Util* is 0. For workload B with a memory request rate $MRR = 0.01$, *CPU Util* decreases with an increasing number of cores while *MCT Util* increases. Looking at *System Util*, the system utilization of workload B is always higher than that of workload A. However, *System Util* of workload B is saturated at a relatively small number of cores while the *System Util* of workload A increases linearly. The insight of this analytical result is that co-locating workload A with workload B has the potential to achieve a higher system utilization.



(a) A = 100, B = 100. (b) A = 400, B = 100. (c) A = 800, B = 100.

Figure 4.3 Turnaround times of co-located workloads A and B. Both workloads are started at the same time and executed with the core allocation given in the X-axis. The vertical bars indicate the core distribution yielding the best performance for the *equal partitioning*, *max system utilization*, and *max CPU utilization* policies, respectively. The line points on the Y-axis indicate the turnaround time of each workload. Subfigures (a)-(c) differ in the amount of work per workload (metric: turnaround time when executed in isolation on a single core).

Using the queueing model, we can simulate co-location performance based on the speedup value of each workload. Figure 4.3 shows the computed total turnaround time of the co-located workloads for different core allocations and a varying amount of work on a 16-core SMP system with one memory controller. Figure 4.4 visualizes the core allocation over time for three common and the presented allocation policies using the workload distribution from Figure 4.3 (b).

The first policy, *Batch*, executes the workloads sequentially. *Equal partitioning* executes the two workloads in parallel, assigning the same number of cores to both. The policy *Max CPU utilization* finds the core allocation that maximizes the total CPU utilization. We observe that *Max CPU utilization* allocates 15 cores to the perfectly scalable workload A and only the minimum of one core to workload B. The proposed *Max system utilization* policy, finally, maximizes the *System Util* as defined by Equation 4.6. *Max system utilization* achieves the shortest total turnaround time of the four policies, demonstrating

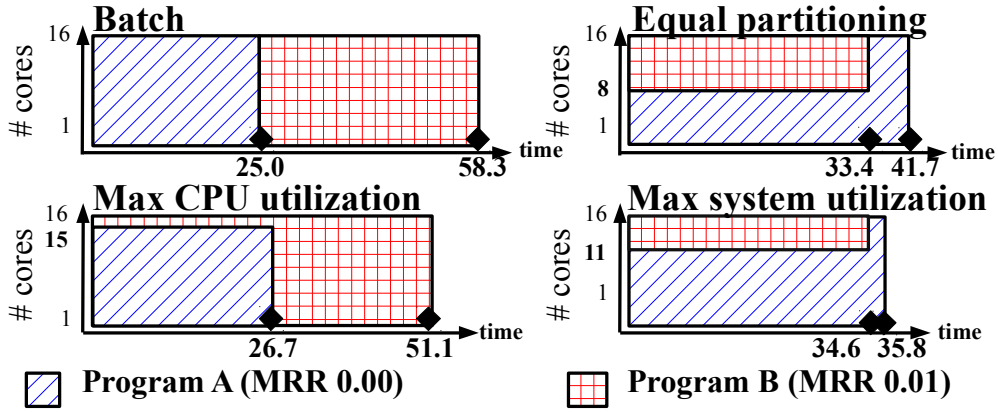


Figure 4.4 Illustration of the performance for the core allocation policies in Figure 4.3 (b).

that focusing only on CPU utilization may not lead to optimal results.

In Figure 4.4, the *Max system utilization* policy yields the best turnaround time among all possible core allocations with a 40% of reduction compared to the *Batch* configuration. For *Max CPU utilization*, after workload A has ended, the execution of workload B policy experiences an inefficient tail execution caused by congestion in the memory system. It is also important to note that the optimal partitioning minimizing the total turnaround time depends on the amount of work of the co-located applications. While both workloads end around the same time with *Max system utilization* in Figure 4.3 (b), yielding the best possible turnaround with a core allocation of 11:5 cores assigned to workload A and B, respectively, this is not the case for Figures 4.3 (a) and (c). For (a), the best distribution is 6:10 cores, and for (c) it is 14:2. The total turnaround time of the *Max system utilization* policy, however, achieves comparable performance to the best distribution and in all situations performs better than *Max CPU utilization*.

The analysis in this section suggests that for co-located parallel applications, maximizing the transient overall system utilization is beneficial if the workloads' size is unknown. Without special provisions, the total execution time of a parallel section is typically not known in advance.

In the NuPoCo framework, we aim to maximize the overall system utilization $NuUtil$ of a multi-socket system. Such a NUMA system is a group of SMP systems, as shown in Figure 1.3 (b). $NuUtil$ is therefore defined as the sum of all individual nodes' system utilization:

$$NuUtil = \sum_{i=0}^{num\_nodes} System\ Util_i \tag{4.7}$$

4.3 NuPoCo: Parallelism Management for Co-Located Parallel Loops

4.3.1 Online Performance Model

Multi-socket systems comprise multiple CPU nodes and memory controllers (Figure 1.3 (b)). Based on a queueing system network for multi-socket multicore systems in Chapter 3, NuPoCo considers the memory controllers and the interconnection links as separate queueing servers and predicts the mean memory response time.

Memory Controller Utilization

To predict the utilization of individual memory controllers, we model each controller with a queueing system as shown in Figure 4.5. A memory controller serves the memory requests issued by the last-level caches (LLC) of the individual CPU nodes. For the queueing system of memory controller m, let N_{cpu_node} be the number of CPU nodes and $MRR_{i,m}^{cpu\_node}$ represent the memory request

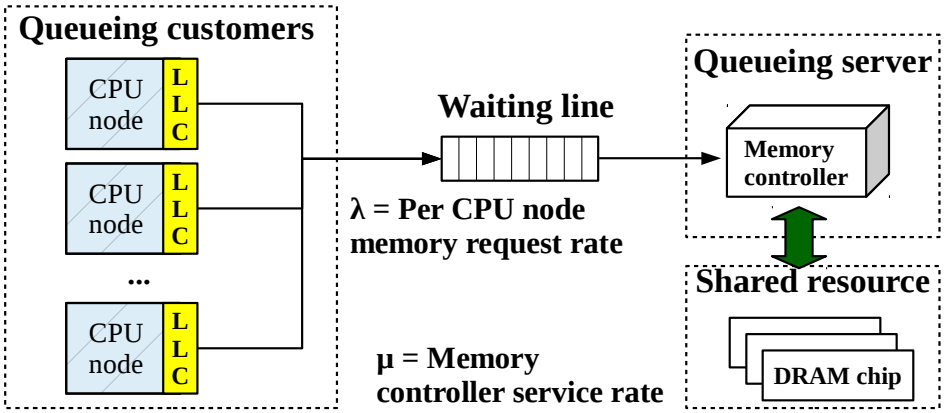


Figure 4.5 Queueing system for an individual memory controller.

rate from CPU node i to memory node m. The mean request arrival rate at memory controller m, $MRR_{avg,m}^{cpu\_node}$, is the average of the individual CPU nodes' request rates

$$MRR_{avg,m}^{cpu\_node} = \frac{\sum_{i=0}^{N_{cpu\_node}} MRR_{i,m}^{cpu\_node}}{N_{cpu\_node}}$$

With the average memory request rate $MRR_{avg,m}^{cpu\_node}$ and the memory service rate μ_m for a memory controller m, we can compute the memory controller utilization $MCT\ Util_m$ and the mean response time MRT_m using Equations 4.4 and 4.3 from Section 4.2.1, respectively. The value of MRT_m is used to compute the CPU core utilization in the section below.

CPU Core Utilization

To compute the CPU core utilization of a CPU node, we first need to calculate the memory request time to each memory controller. To do so, the queueing system depicted in Figure 4.6 is employed. This queue models the serialization of outgoing memory requests from the node's LLC to one memory controller.

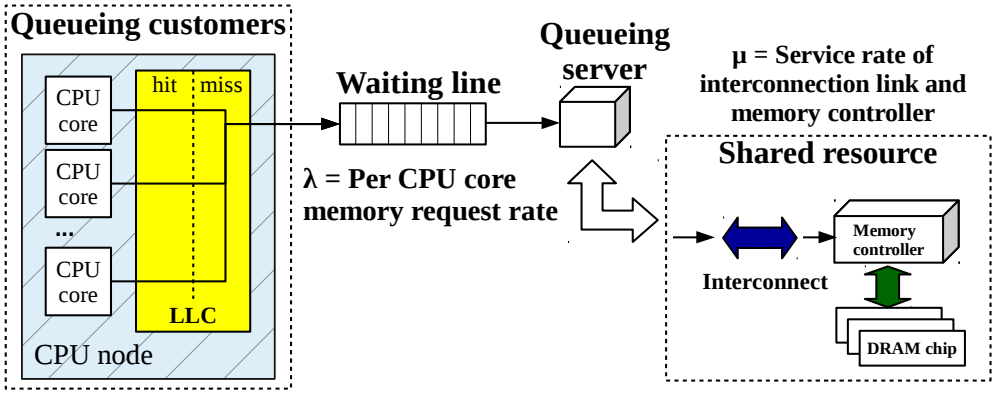


Figure 4.6 Queueing system for CPU core utilization prediction.

Outgoing memory requests include missed read and write operations and hardware prefetch requests.

With $N_{cores\_in\_node}$ representing the number of cores in CPU node i accessing memory node m, the average memory request rate $MRR_{avg,m}^{cpu\_core}$ is estimated as follows

$$MRR_{avg,m}^{cpu\_core} = \frac{\sum_{j=0}^{N_{cores\_in\_node}} MRR_{j,m}^{cpu\_core}}{N_{cores\_in\_node}}$$

The service rate includes the service rate of the interconnection link, $link_{i,m}$, from CPU node i to memory node m, and the mean response time of the memory node (MRT_m) as obtained in Section 4.3.1. The service rate $\mu_{i,m}$ is given by

$$\mu_{i,m} = \frac{1}{1/link_{i,m} + MRT_m}$$

With $MRR_{avg,m}^{cpu\_core}$ and $\mu_{i,m}$, the total mean memory response time from CPU node i to memory node m, $MRT_{i,m}$, is computed from Equation 4.3.

The total memory response time for a core is obtained according to Equation 4.2. While all outgoing memory requests of an LLC affect the mean memory response time, only requests caused by read misses stall a core and thus affect the CPU core utilization. We estimate the rate of outgoing read requests from

every core to each memory node using the per-core number of LLC read requests per time[1]. For the cores in CPU node i with an LLC read request rate to memory node m, $LLC_{i,m}^{read}$, the total memory response time is computed by

$$MCT\ Time = \sum_{m \in M} CPU\ Time \times LLC_{i,m}^{read} \times MRT_{i,m}$$

Based on the ratio between *CPU Time* and *MCT Time*, we can compute the CPU utilization using Equation 4.5.

Implementation and Validation

The required inputs for the performance model are obtained from the hardware performance monitoring unit. We measure LLC accesses, LLC misses, all memory requests that affect memory utilization (read, write, prefetch) to all memory controllers, and the total number of CPU cycles. AMD [2, 3] and Intel [43, 44] systems support all required counters.

The application-specific parameters $MRR_{i,m}^{cpu\_core}$, $MRR_{i,m}^{cpu\_node}$, and $LLC_{i,m}^{read}$ are computed at runtime without depending on offline information. A direct measurement of the the per-core LLC accesses and misses is not supported by the hardware. NuPoCo gets around this limitation by initially allocating only threads of one application to the cores in one CPU node, then divide the node's LLC accesses and misses by the number of cores. This happens once for each parallel section during a one-time brief online profiling phase (see Section 4.3.2). The machine-dependent parameters μ_{mct} and $link_{i,m}$ are determined by executing a synthetic workload from the Stream benchmark [64] that generates memory accesses from one core to specific memory nodes and measures the mean memory service time. This process

[1]To compute the LLC read request rate per core, we initially allocate only threads of the same application to the cores in a node, then divide the node's LLC read request rate by the number of allocated cores, see Section 4.3.2.

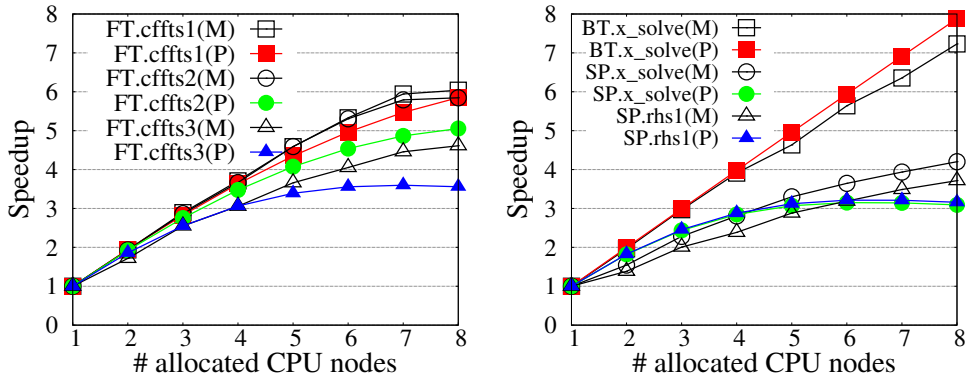


Figure 4.7 Speedup predictions (P) and measurements (M) of several parallel loops from FT and SP (NPB) [4] on a 64-core AMD Opteron system.

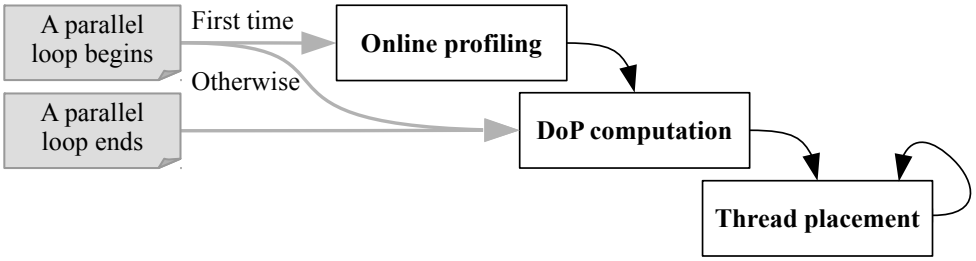
is required only once for a given hardware platform.

Figure 4.7 compares the predicted with the actual speedup for several parallel loops from an NPB implementation [80]. The results show that the performance model can capture the trend of the speedup. Since the speedup is computed from the predicted CPU core utilization (Equation 4.1), this result confirms that predictions of resource utilization are also possible with the presented model. An extensive analysis for other (co-located) NPB parallel loops on a 64-core AMD and a 72-core Intel system shows that the performance model predicts the speedup with moderate absolute percentage errors of 10-15%, similar to the results in the prior work [17].

4.3.2 Managing Parallelism

The degree of parallelism and the core assignment of co-located applications is managed at runtime by NuPoCo.

NuPoCo's parallelism manager is activated whenever a parallel loop begins or ends. It performs the following three steps: *online profiling*, *DoP computation*, and *thread placement*. When a parallel loop is executed for the first time, the



online profiling phase is initiated that profiles the new parallel loop for a short period of time; profiling is skipped for the second and later invocations of the same loop. The *DoP computation* step uses the queueing systems presented in Section 4.3.1 to compute a thread allocation that maximizes the overall system utilization. Once the thread count for each co-located application has been determined, the *thread placement* phase begins during which individual threads of an application are relocated if opportunities exist to improve performance.

Online Profiling

During online profiling, all cores of the system are assigned to the new parallel section for a short period of time. This serves two purposes. First, it ensures that the data of an application is distributed in a similar manner as in a standalone execution under a NUMA first-touch allocation policy. Second, it allows NuPoCo to infer the LLC miss rate per core by measuring the node's LLC rate and divide it by the number of cores in the node. This initial profiling period is set to 150ms; long enough to ignore cache warming effects and sufficently short not to affect other running applications much.

DoP Computation

The goal of this step is to maximize system utilization by allocating the proper thread counts for running parallel applications. Algorithm 2 shows how the

Algorithm 2 DoP computation

```
1: for each cpu_node ∈ system do
2:     util_list = [ ]
3:     for each wl ∈ running workloads do
4:         cpu_node.allocate(wl)
5:         NuUtil ← performanceModeling()
6:         util_list.append(NuUtil)
7:         cpu_node.deallocate(wl)
8:     best_wl ← bestExpectedNuUtil(util_list)
9:     for each cpu_core ∈ cpu_node do
10:        cpu_core.allocate(best_wl)
11: Communicate core allocation to parallel runtimes
```

parallelism manager determines the degree of parallelism for each application. The number of cores per workload is determined in a greedy manner. The basic allocation unit in this stage is a CPU node. Starting with an empty allocation, each CPU node in the system (line 1) is assigned in turn to the application (lines 9–11) that is expected to yield the best overall system utilization *NuUtil* (Section 4.2.2) (lines 5–8). The prediction of the system utilization *NuUtil* (line 6) is based on the performance prediction model from Section 4.3.1.

The number of CPU nodes is assumed to be larger than the number of co-located applications. At least one CPU node is allocated to each application executing a parallel section. Applications in serial sections are assigned a single core.

Thread Placement

The DoP computation assigns all core resources of a CPU node, i.e., cores sharing the same LLC (Section 1.2.2), to one application. This leaves room for additional performance improvements. Individual threads of memory-intensive applications may require substantial LLC resources. If allocated to the same CPU node, thus sharing the same LLC, this may lead to contention or, even

Algorithm 3 Thread placement

```
1: Initialize cpu_node_list in descending order of LLC accesses since the last
   invocation
2: repeat
3:     busy_nd ← cpu_node_list.pop_front()
4:     idle_nd ← cpu_node_list.pop_back()
5:     if busy_nd.llc_accesses() / idle_nd.llc_accesses() > threshold then
6:         busy_wl ← busy_nd.max_llc_miss_rate()
7:         idle_wl ← idle_nd.min_llc_accesses()
8:         SwapCores(busy_wl, idle_wl)
9: until cpu_node_list is empty
10: Communicate core allocation to parallel runtimes
```

worse, thrashing in the LLC. CPU-bound applications, on the other hand, typically contend less for LLC resources. Co-locating memory-intensive with CPU-bound workloads in the same CPU node thus has the potential to yield an improved overall system utilization.

Algorithm 3 outlines the implementation of this idea. The algorithm is invoked periodically every 50ms after Algorithm 2 has ended. It repeatedly retrieves the CPU nodes that exhibit the highest (*busy_nd*) and lowest (*idle_nd*) number of LLC accesses since the last iteration (lines 3–4). If the ratio of LLC accesses exceeds a given threshold (currently set to 2; line 5), we select the workload that observed the highest LLC miss rate from *busy_nd* (line 6) and the one with the lowest number of LLC accesses from *idle_node* (line 7), based on the information inferred during online profiling (Section 4.3.2). The algorithm then swaps the location of a number of cores (NuPoCo exchanges two cores by default) of the two applications (line 8). This process is repeated until the list is empty (line 9). Although this thread placement technique is a hill-climbing method, it quickly reaches a steady state as later demonstrated in Section 4.4 and Figure 4.10.

4.4 Evaluation of NuPoCo

4.4.1 Evaluation Scenario 1

Here, we provide an evaluation of NuPoCo based on the results presented in our PACT 2018 paper [15]. The experimental results are obtained from NuPoCo for OpenMP C-version workloads which are executed under COOP-DYN (Section 2.3.2). We provide some additional experimental results based on COOP-ULT (Section 2.3.1) in Section 4.4.2.

We evaluated NuPoCo on the 64-core (8-node) AMD Opteron platform and the 72-core (4-node) Intel Xeon platform described in Section 1.2.2. The Intel platforms are equipped 756GB of DRAM memory in this scenario. The Linux kernel versions are 4.4.35 for AMD and 4.4.0 for the Intel platform. We use a modified version of OpenMP v5.4.0 [34] with the COOP-DYN loop scheduler (Chapter 2).

Target Applications

For the co-location scenarios, we utilize target applications from NPB [4], Parsec [7], and Rodinia [14] (Table 4.1). NPB applications represent HPC workloads that require large amounts of memory and/or lots of computational resources. We selected *BT*, *FT*, *SP*, and *EP* from an OpenMP NPB implementation [80]. *BT*, *FT*, and *SP* are both CPU- and memory-intensive workloads. The data set of *FT* and *SP* is very large. We categorize these three applications as `Type-A` to represent applications that require a significant amount of system resources. On the other hand, *EP* is an almost perfectly scalable kernel that rarely accesses memory. We classify *EP* as `Type-B`, a class that extremely under-utilizes the memory system. The four NPB applications use input class D with a large problem size. The number of iteration steps is adjusted to obtain standalone turnaround times that are similar to those of the other applications.

App	Resource requirement			
	CPU	**Memory**	**Data size**	**Type**
BT	High	Medium	Medium	A
FT	High	High	Huge	A
SP	High	High	Huge	A
EP	High	Almost none	Almost none	B
KM	Low	Medium	Small	C
BS	Low	Low	Small	C
SC	Low	Medium	Small	C

Table 4.1 Target applications.

Parsec's *blackscholes (BS)* consists of long serial sections and one parallel loop that does not require a lot of system resources compared to `Type-A` applications. *BS* is executed with the `native` input data set. *kmeans (KM)* from the Rodinia benchmark is executed with 3,000,000 objects and represents a non-scalable application with frequent synchronization between cores and a long serial section at the beginning of its execution. *KM* under-utilizes CPU resources. In addition, we also evaluate *KM* (K-means clustering) and *SC* (StreamCluster) from the Rodinia OpenMP implementations. These applications are classified as `Type-C`, representing applications that under-utilize CPU resources.

Co-location Scenarios

The presented approach is compared with the following execution modes:

- **Batch.** Applications are executed serially. The number of threads is equal to the number of system cores, each thread is pinned to a core [2].

- **Native.** Applications generate as many threads as there are cores in the

[2]On our platforms, thread binding performs better in standalone execution, but worse in co-located executions.

system and are co-located by the Linux scheduler. Thread binding is disabled to allow the Linux scheduler to perform thread and data placement.

- **Equal.** This policy assigns the same number of cores to all running parallel sections and a single core to a serial process. The cores are allocated linearly.

- **Scalability.** This core allocator is based on a hill-climbing approach. We have implemented a hill-climbing algorithm inspired by the algorithm proposed in C3PO [75]. The implemented algorithm in this configuration observes CPU utilization and increases core count of CPU intensive jobs. The algorithm changes the number of assigned cores to the applications based on the measured CPU utilization.

- **NuPoCo Greedy.** To demonstrate the effect of the thread placement technique (Section 4.3.2), this policy performs only DoP computation (Section 4.3.2).

- **NuPoCo** Our proposal.

With **Batch** and **Native**, loops are scheduled statically as this yields the best performance among all available OpenMP loop schedulers on our platforms. For **Equal**, **Scalability**, **NuPoCo Greedy**, and **NuPoCo**, we use the cooperative work scheduler presented in Chapter 2 to provide dynamic spatial scheduling. NuPoCo is executed with the following parameters: the initial profiling phase is 150ms (Section 4.3.2). The thread placement algorithm (Algorithm 3) is invoked every 50ms and uses a threshold value of 2 for core swapping.

To measure the co-location performance, we consider the total execution time from start to finish of all co-located applications. Each scheduler is evaluated using the normalized total turnaround time (NTT) with regards to **Na-**

Co-location type	Co-located workloads
Mix of Type-A	(1) BT, FT (2) BT, SP (3) FT, SP (4) BT, FT, SP
Mix of Type-A and B	(5) BT, FT, EP (6) BT, SP, EP (6) BT, SP, EP (7) FT, SP, EP
Mix of Type-A and C	(8) BT, KM (9) FT, KM (10) SP, KM

Table 4.2 Co-location scenarios.

tive. We also report the speedup relative to the harmonic mean (Hmean) which is known as a speedup metric that also considers the fairness of co-located jobs [61]. All results are obtained by executing each scenario three times and taking the average.

We consider various co-location scenarios as follows. First, two `Type-A` applications that require substantial system resources are co-located. To see the performance behavior for different types of applications, the `Type-B` *(EP)* and the `Type-C` *(KM)* application are co-located with several `Type-A` applications. Details of each scenario are given in Table 4.2.

Experimental Results

Figure 4.8 shows the NTT of the six execution modes `Batch`, `Native`, `Equal`, `Scalability`, `NuPoCo Greedy`, and `NuPoCo` on the AMD and the Intel system for ten different scenarios. All co-located applications are started at the same time but finish at different points in time.

The results show that, on average, `NuPoCo` achieves the best system throughput among the six core allocation configurations on both platforms. Under the geometric mean, `NuPoCo` achieves an NTT of 0.91 (9% improvement) on the AMD system and 0.81 (19% improvement) on the Intel platform over the Linux scheduler. The performance improvement with `NuPoCo` is up to 20% on the AMD platform (scenario 4) and 35% on the Intel system (scenario 10). `NuPoCo` also

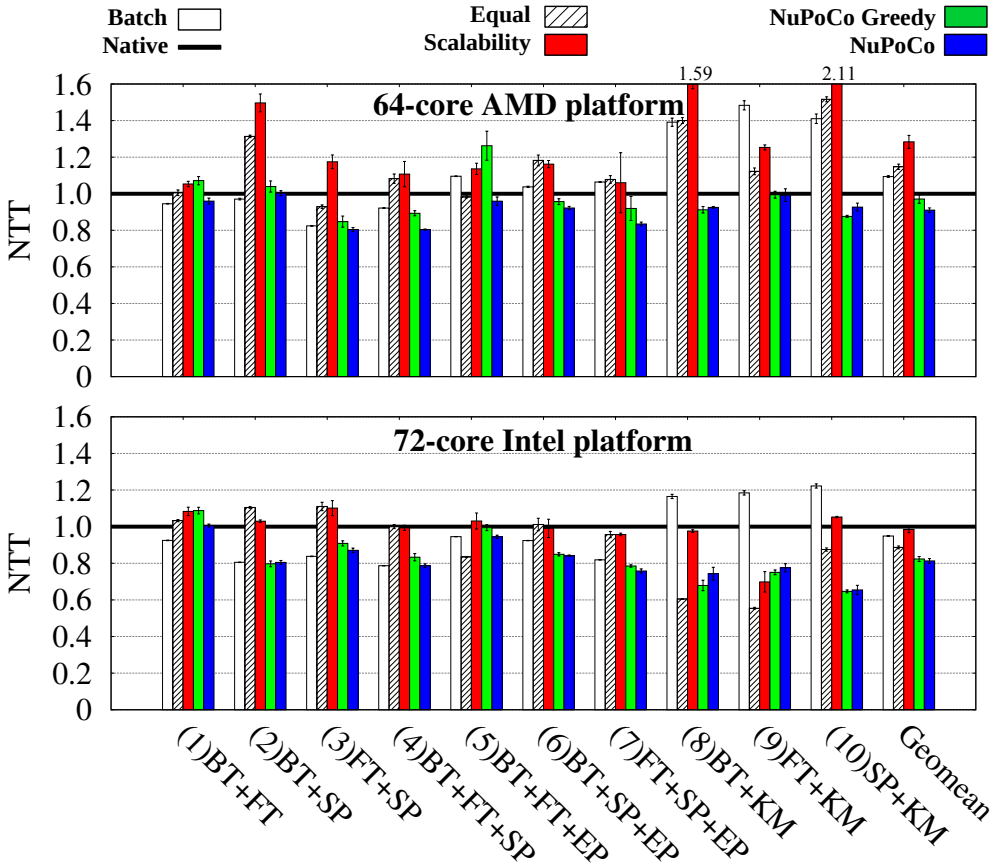


Figure 4.8 Normalized total turnaround time (NTT) to Linux `Native` for the co-location scenarios.

does not report any performance degradation for the ten scenarios. The average job turnaround time of co-located applications with `NuPoCo` is 10.8% and 12.3% shorter than that of `Native` for the AMD and the Intel platform, respectively.

Scenarios 1–4 mix two to three `Type-A` applications causing high competition for platform resources. We observe that the `Batch` configuration is a suitable choice for scenarios 1–4 as they utilize the platform's CPU core and memory systems well and show good scalability. `Native` can not efficiently execute these scenarios (especially scenario 3) compared to `Batch` or `NuPoCo`

because it suffers from a high resource interference as all of the applications have a high degree of resource demands. For scenarios 1–3, `NuPoCo` shows almost the same performance as `Batch` on the AMD/Intel platforms. For scenario 4, `NuPoCo` outperforms `Batch` by 10%. This is because the three `Type-A` applications contain serial sections during which `NuPoCo` is able dynamically assign more cores to parallel sections.

To test the effectiveness of the presented methods when different types of applications are co-located with `Type-A`, we execute *EP*, a (`Type-B`) application with two applications from `Type-A` in scenarios 5–7. We observe that performance of `Batch` decreases compared to scenarios 1–3. Since *EP* puts no pressure on the memory system, the `Batch` configuration suffers from a low utilization when *EP* is executed standalone. On the other hand, `Native` is able to increase resource utilization for co-located *EP* and `Type-A` applications compared to `Batch`. `NuPoCo` achieves better performance than the other schedulers thanks to its online performance prediction model and dynamic thread count adjustment.

For the remaining scenarios 8–10, we co-locate *KM* with *BT*, *FT*, and *SP*. *KM* does not require a lot of CPU resources because of its long serial sections and the synchronizations, hence, allocating only a subset of cores to *KM* is beneficial. As expected, `Batch` experiences a significant performance degradation when executing *KM*. `NuPoCo` performs well for these scenarios, but we observe that on the Intel platform, the `Equal` policy performs best for scenarios 8 and 9. A static core allocation scheme can be beneficial for *KM* and its short parallel section that is executed iteratively.

Overall, we observe that the conventional allocation approach to co-location, `Scalability`, is not beneficial to improve the system's throughput. Although CPU utilization is maximized, co-located applications suffer from memory con-

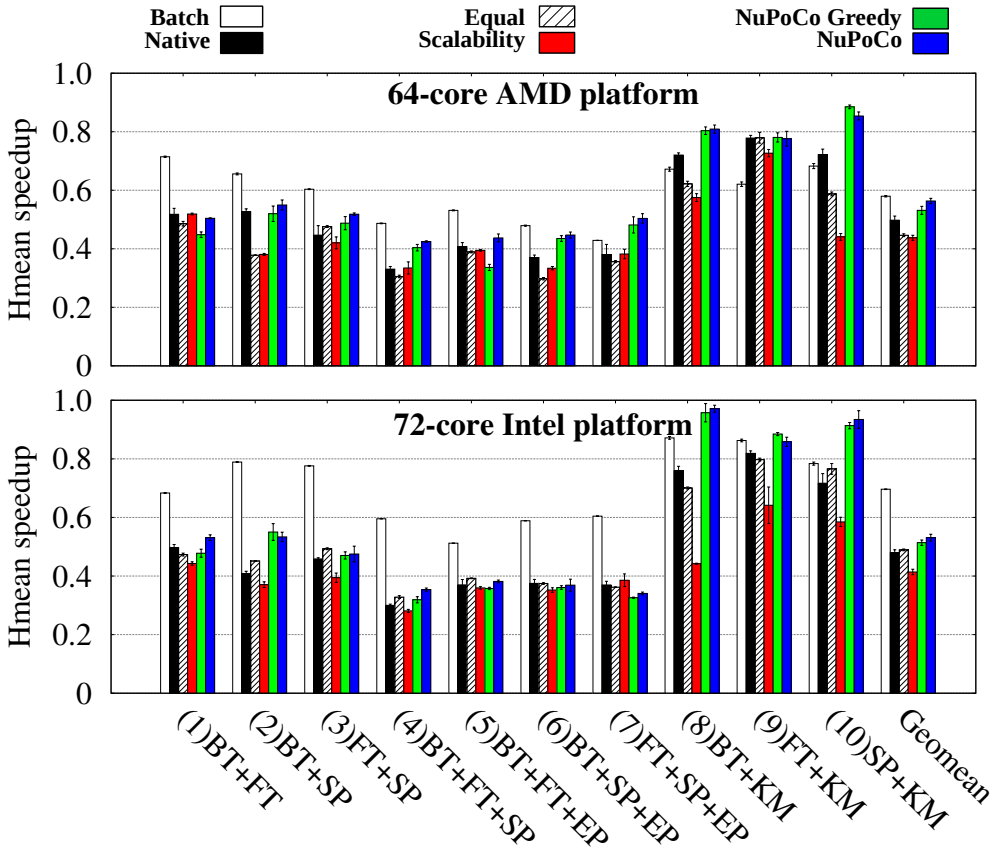


Figure 4.9 Hmean of speedup relative to standalone execution for the co-location scenarios.

tention and a low CPU utilization when scalable applications finish earlier. Additionally, in scenarios 2, 8, and 10, `Scalability` suffers from a severe performance degradation. A closer inspection reveals that the hill-climbing algorithm in some cases is oscillating, thus continuously changing the number of assigned cores. The benefit of the *thread placement* (Section 4.3.2) in NuPoCo is visible in comparison with `NuPoCo Greedy`. Despite the additional runtime overhead, proper thread placement is beneficial in general.

In terms of the Hmean speedup shown in Figure 4.9, `NuPoCo` outperforms

`Native` by 13.2% and 10.8% on the AMD and Intel platforms, respectively. The results show that fairness is not only preserved but improved with `NuPoCo`. The Linux scheduler often favors specific workloads resulting in a slow performance for other applications. This is also visible in Figure 4.10. `Batch` achieves a relatively good Hmean speedup because the first job in `Batch` is always assigned the optimal value.

To summarize, the results show that `NuPoCo` performs well for a diverse mix of applications, especially when the co-located applications exhibit different performance characteristics. If the co-located workloads exhibit similar characteristics, `NuPoCo` consistently provides good performance comparable to the best system configuration (`Batch` or `Native`).

Case-Study and Overhead Analysis

To better understand and demonstrate NuPoCo's operation, three application types, *FT* (`Type-A`), *EP* (`Type-B`), and *BS* (`Type-C`) are co-located. Using the open-source trace visualizer SnuMAP [37], Figure 4.10 visualizes the core allocations over the course of execution on the 64-core AMD platform.

In this scenario, each application starts at a different time. *EP* is started first and monopolizes the core resources. *FT* joins a bit later and starts its first parallel section at `t1` with the clean-profiling phase. NuPoCo then performs the DoP computation followed by the thread placement technique for *EP* and *FT*. We observe that the profiling stage in NuPoCo is almost invisible, and the core allocations quickly converges to the steady state (`t1`–`t2`). The different core allocations indicate that NuPoCo differentiates between multiple parallel sections in *FT*. Once *BS* is started, it uses only one core for its initial long serial section until *BS* reaches the main parallel section. Over the entire execution with a scheduling epoch of 50ms, in total 32 thread count selections and $1,627$

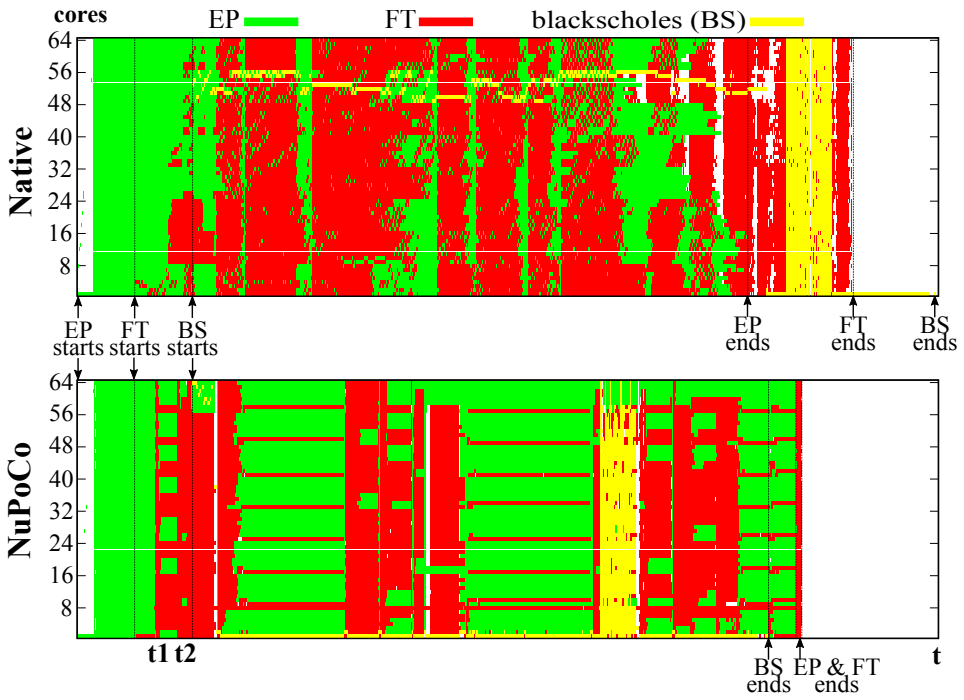


Figure 4.10 Trace visualization for a co-location under Native and NuPoCo on the AMD machine.

thread placements have been executed with an average computational overhead of 1.8ms and 1.5ms. Since NuPoCo runs in parallel to the applications, this overhead is hidden, or rather included in the results. Compared to `Native`, we observe that thread interference of `Native`'s time-sharing model causes severe synchronization delays for parallel sections in *FT* and the serial process of *BS*. For this scenario, NuPoCo reports a 19% shorter total execution time over `Native`.

4.4.2 Evaluation Scenario 2

In addition to the experimental results in Section 4.4.1 which are based on the results presented in the PACT 2018 paper [15], we provide additional experiments for more co-location scenarios where applications are executed under

`COOP-ULT` and OpenMP Fortran applications. In addition, the new experiments compare our approach with other state-of-the-art core allocation techniques by implementing their method.

We used the same AMD and Intel multi-socket systems described in Section 1.2.2. One of the difference between the scenarios in Section 4.4.1 is that, in this experimental setup, we used 512GB memory since we had changed the DRAM chips. For the COOP-ULT technique we used GNU OpenMP runtime provided by gcc-9.

Co-location scenarios

For co-location scenarios, we selected applications that have different resource requirement levels and evaluate the core allocation performance for all possible pairs of the selected workloads. To consider applications with different resource usage, we use five different applications `BT`, `FT`, `SP`, `SC`, and `KM` described in Section 4.6. Then, we evaluate the total turnaround time of the core allocation techniques for all possible combinations (two and three applications pairs; total 18 different co-location scenarios) except few cases where the total memory consumption is more than 128GB or if the scenario eventually has a tail serial execution (only a single thread executes during a significant tail execution time).

Comparisons

The presented approach is compared with the following execution modes:

- **Timesharing (Baseline):** applications execute in the default mode (i.e. they create as many threads as there are cores in the system) and are co-located by the Linux scheduler. Thread binding is disabled to allow the Linux SMP scheduler to perform thread and data placement.

- **Simple Policy:** resource managers may use a simple decision making process to select the proper option to execute multiple applications. For example, a resource manager can execute applications with the default time-sharing mode or execute them in series with a batch-style scheduling. In addition, threads can be bound to specific cores or can be scheduled using Linux SMP scheduling. This simple policy represents the best option among these simple options (i.e. timesharing and batch with or without thread affinity setting).

- **C3PO (CPU-centric):** We implemented C3PO's hill-climbing algorithm as presented at PACT'13 [75]. At runtime, the algorithm increases or decreases the number of assigned cores to the applications based on the measured CPU utilization.

- **AB (speedup-centric):** this policy provides core resources for the co-located applications in proportion to their Amdahl speedup curve (i.e. high speedup applications use more cores). To obtain the speedup curve for each application, the AB method requires at least two profiling runs of an application with different core allocations. We implemented the Amdahl Bidding algorithm as presented at HPCA'18 [91].

- **EQP (naive):** this policy assigns the same number of cores to all running parallel sections and a single core to a serial process. The cores are allocated linearly. This policy has been used in Callisto (EuroSys'14 [40]).

- **NuPoCo** Our proposal.

With **Batch** and **Native**, loops are scheduled statically as this yields the best performance among all available OpenMP loop schedulers on our platforms. Since the original source code of **C3PO** and **AB** is not available, our

implementation may have some differences in some detailed configurations (e.g. scheduling period). For **C3PO** and **AB**, applications use the given allocated core resources using thread affinity setting using a kernel module as implemented in their frameworks. In other words, in **C3PO** and **AB**, for an application, the kernel module assigns CPU affinity for the application's spawned threads. All the worker threads of the application can be assigned to any core resource among the allocated core resources. For **Equal** and **NuPoCo**, we use the `COOP-ULT` technique presented in Chapter 2 to execute applications in spatial core allocation.

Experimental Results

Figure 4.11 shows the normalized total turnaround time for the all two and three applications pairs on the AMD and the Intel system. The results show that, on average, `NuPoCo` achieves the best system throughput among the six core allocation configurations on both platforms. Under the geometric mean, `NuPoCo` achieves an NTT of 0.78 (22% improvement) on the AMD system and 0.80 (20% improvement) on the Intel platform over the Linux scheduler. `NuPoCo` also does not report significant performance degradation over all the scenarios showing the benefit of space-shared core allocation for co-located parallel applications.

We observe that the `Simple policy` configuration often improves performance over `time-sharing` for specific scenarios such as `FT+SP`; these workloads require high resource usage compared to other scenarios and `Simple policy` can smartly chose not to co-locate these applications and execute them in the batch mode. `Native` can not efficiently execute these scenarios (especially scenario 3) compared to `Batch` or `NuPoCo` because it suffers from a high resource interference as all of the applications have a high degree of resource demands.

Overall, we observe that the state-of-the-art core allocation approaches such

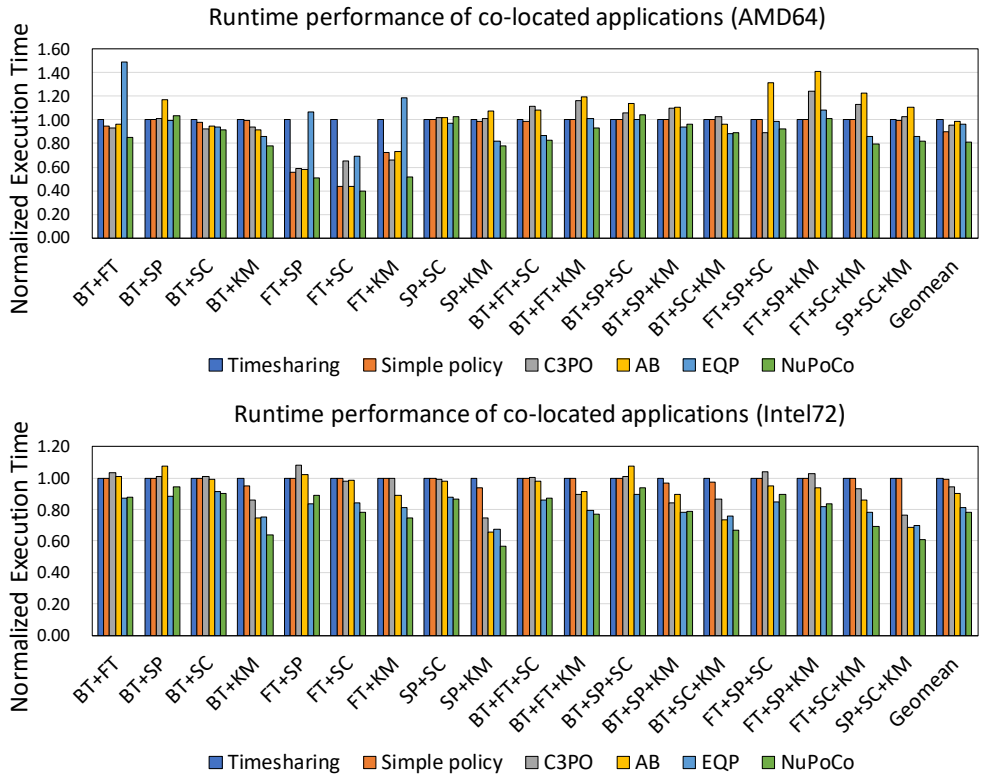


Figure 4.11 Normalized total turnaround time (NTT) to Linux `Native` for the co-location scenarios.

as `C3PO` and `AB` are not beneficial to improve the system's throughput compared to `NuPoCo`. Although CPU utilization is maximized, co-located applications suffer from memory contention and a low CPU utilization when scalable applications finish earlier. Additionally, in specific scenarios such as `FT+SP+SC` in AMD, `C3PO` suffers from a severe performance degradation. A closer inspection reveals that the hill-climbing algorithm in some cases is oscillating, thus continuously changing the number of assigned cores.

To summarize, similar to the evaluation in Section 4.4.1, the results show that `NuPoCo` performs well for a diverse mix of applications for these new experiments, especially when the co-located applications exhibit different perfor-

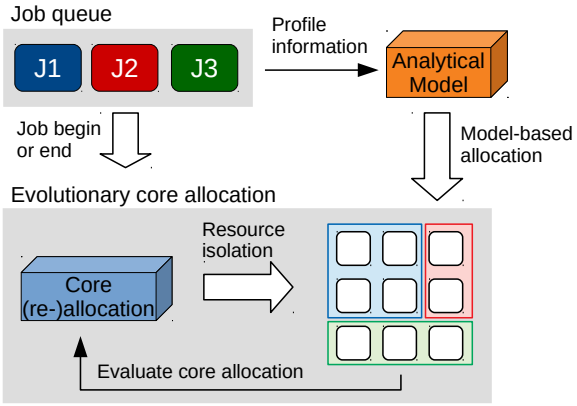


Figure 4.12 Overview of MOCA.

mance characteristics. If the co-located workloads exhibit similar characteristics (e.g. `FT+SP` in AMD) compared to other scenarios, `NuPoCo` consistently provides good performance comparable to the best system configuration selected by the `Simple policy`.

4.5 MOCA: An Evolutionary Approach to Core Allocation

In this section, we present MOCA (**M**ulti-**O**bjective **C**ore **A**llocation), an ongoing research on the core allocation problem. While MOCA aims to find a better allocation over NuPoCo, selecting the best core allocation among all possible allocations is still not realistic because there are too many possible allocations. Instead of finding the best allocation, therefore, MOCA focuses on finding an appropriate allocation that achieves a desired utilization goal within a reasonable amount of time.

Figure 4.12 illustrates an overview of the MOCA's core allocation. MOCA employs evolutionary meta-heuristics inspired by genetic algorithms (GAs) [50, 31] for multi-objective optimization problems (MOOPs). Starting with ran-

domized allocations, MOCA's evolutionary method changes the core allocation using the concepts of crossover and mutation in GAs to improve a core allocation. In Section 4.5.1, we present the details of the evolutionary approach. In the evolutionary core allocation, MOCA uses model-based allocation that leverages the analytical model in Chapter 3 in the evolutionary process to estimate the resource utilization for a given allocation without executing on the given hardware platform. Section 4.5.2 presents the details of the model-based allocation.

MOCA (re)starts its core allocation if there is any update in the job queue (e.g. a new application joins to the job queue or an application has been completed). Note that, unlike NuPoCo, MOCA currently allocates core resources per application (not at the level of parallel loop). While NuPoCo determines thread count at CPU-node granularity (8 nodes à 8 cores on AMD, 4 nodes à 18 cores on the Intel system), MOCA allocates individual cores based on the evolutionary approach which is more complex than the greedy algorithm. Per-parallel loop management often incurs a high overhead in the evolutionary allocation approach depending on the loop's problem size and the platform's performance. Managing the resource management granularity (e.g. per parallel loop or per application management) is a future work of MOCA.

MOCA provides spatial scheduling for a core (re-)allocation. MOCA uses the cooperative runtime system (COOP-ULT) to exploit malleable execution of OpenMP applications (Chapter 2). For other types of applications such as Spark, MOCA uses Docker [27] for spatial core allocation.

4.5.1 Evolutionary Core Allocation

Figure 4.13 illustrates the MOCA's evolutionary method. We first explain how to encode a core allocation to the concept of chromosome used in GA ap-

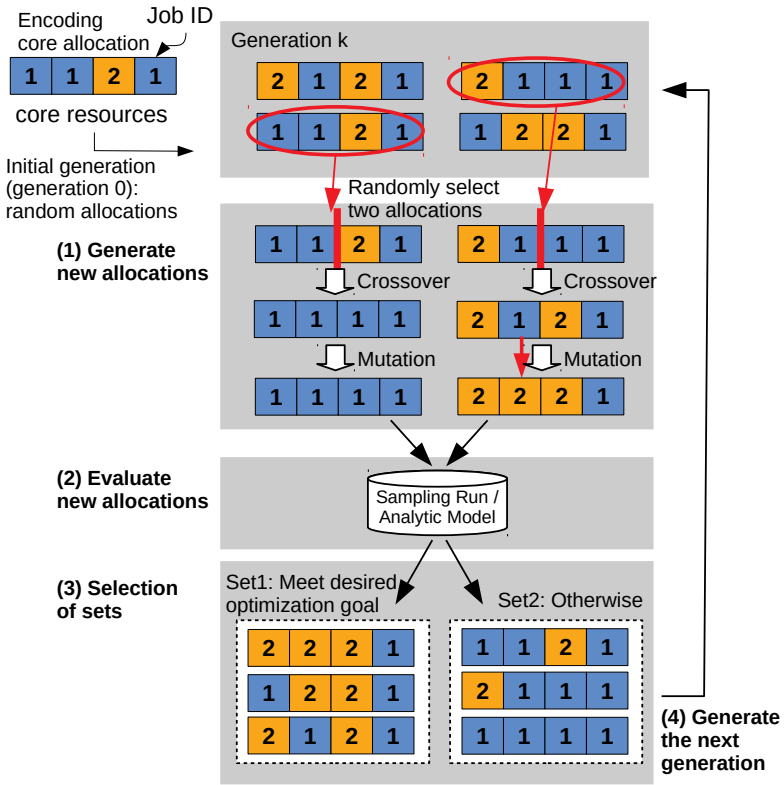


Figure 4.13 Evolutionary core allocation.

proaches. The approach considers each core resource to be a gene of a chromosome. Each core can execute any of the co-located applications. Hence, each gene in the encoded core allocation contains the job ID.

The evolution starts with randomized allocations and then follows the general process used in typical GA approaches. For generation k, we keep four (the typical number used in many GA approaches) different allocations along with their utilization information. ① **New Allocations**: in the first step, we randomly pick two allocations in generation k and perform *crossover* among the selected two allocations. We also allow *mutation*; cores can randomly change the assigned job ID with a given probability. This is necessary to overcome the

local-optimum problem. ② **Evaluate**: for these two new allocations, we need to evaluate their resource utilization. MOCA's *model-based allocation* exploits an analytical model to estimate resource utilization without executing the allocation on the given system. If not using the analytical model, MOCA can use a sampling run for each of these allocation to measure their CPU utilization and memory bandwidth. ④ **Sets**: MOCA maintains two separated sets that store core allocations depending on their resource utilization. `Set1` stores core allocations where both the CPU utilization and memory bandwidth usage exceeds the given utlization goal. `Set2` stores other core allocations. ⑤ **Next Generation**: we finally select four core allocations from `Set1` and `Set2` to generate next generation $k + 1$. We first select allocations from `Set1` if there are any allocations in `Set1`. If the number of allocations in `Set1` exceeds the population (4), we select allocations that have higher CPU utilization. This means that, in the MOCA's policy, once a desired goal of CPU (*cpu_threshold*) and memory bandwidth (*bw_threshold*) usage is achieved, MOCA focuses on maximizing CPU utilization with an assumption that once the memory system is saturated enough, improving CPU throughput is more beneficial for system performance. Otherwise, to reach the desired resource utilization goal, MOCA's evolutionary allocation selects next core allocation generations based on the summation of CPU and memory bandwidth utilization as the objective function. If the number of allocations in `Set1` is less than 4, we then select allocations that have higher values of the summation of CPU utilization and memory bandwidth usage from `Set2`.

4.5.2 Model-Based Allocation

The evolution process in Section 4.5.1 requires frequent changes to the core allocation to find better allocations. To remove or reduce runtime core (re-

)allocations, the model-based allocation uses an analytical model.

Evaluation of Core Allocation

Here, we describe how we define and measure CPU utilization and memory bandwidth from available hardware performance counters.

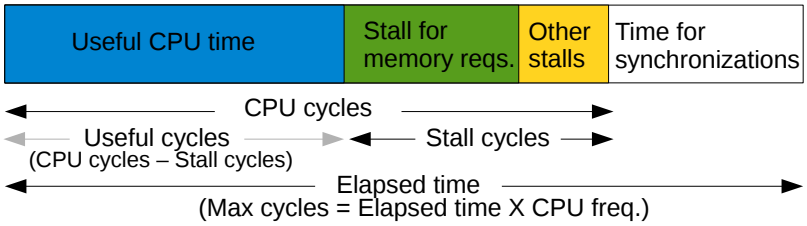


Figure 4.14 CPU time breakdown and available measures.

cpu_util: Figure 4.14 shows a CPU time breakdown and some information that we can measure from hardware performance counters. Many monitoring tools such as `htop` use cpu cycles when computing per-core utilization. However, this overestimates the usefulness of the core resource because cpu cycles include stalls in shared resources. Therefore, for each core resource c, we measure per-core utilization using cpu cycless and stall cycles, and compute the average of per-core utilization as the system's overall cpu utilization cpu_util.

$$cpu\_util = \frac{1}{|C|} \cdot \sum_{c \in C} \left(\frac{useful\ cycles(c)}{max\ cycles(c)} \right) \tag{4.8}$$

bw_util: To estimate memory bandwidth usage, MOCA uses the number of memory requests per time. AMD's *Memory Controller Requests (NBPMCx1F0)* [2, 3] and Intel's *UNC_H_IMC_WRITES/READS* [43, 44] performance counters provide the necessary information. For each memory node m in the set of all memory nodes M, the utilization of memory bandwidth is defined as the measured number of memory requests divided by the maximal number of memory requests that the memory node can serve, as follows.

$$bw\_util = \frac{1}{|M|} \cdot \sum_{m \in M} \left(\frac{\#\ measured\ requests\ per\ time(m)}{\#\ maximum\ requests\ per\ time(m)} \right) \tag{4.9}$$

To measure the maximum number of requests per time for each memory node, we use synthetic workloads `StreamTriad` from the Stream benchmark suite [64]. For each measurement, we allocate all data to the specific memory node and use all available cores to generate as many memory requests as possible during the given time.

Profiling

The required information for the analytical model is collected using hardware performance counters. We measure per-core LLC miss rate (llc) and per-core memory requests (mrr) that affect memory bandwidth utilization of individual memory controllers, and the total number of CPU cycles and stall cycles to take into account an application's synchronization overhead ($sync$). AMD [2, 3] and Intel [43, 44] systems support all required counters. The list of per-application profile is given in Table 4.3.

To obtain this information, one profiling run is needed to execute the application in isolation using all available hardware cores. The profiling cost is smaller than other analytical core allocation techniques based on scalability information (i.e. at least two profiling runs are required to estimate the speedup curve). Application profiling can also theoretically be done at runtime by (temporarily) allocating all cores to the application.

There are also several machine-dependent parameters. In multi-socket systems, one node consists of a CPU node, itself composed of a group of CPU cores, and its attached memory node, as shown in Figure 4.15. The individual nodes are connected by an interconnection network such as AMD's HyperTransport [74] or Intel's QPI (Quick Path Interconnect) [68]. To take the

Name	Description (for computation, refer to Figure 4.14)
$sync$	synchronization overhead (e.g. scheduling overhead, thread synchronization). $sync = \frac{1}{\|C\|} \cdot \sum_{c \in C} \frac{\text{time for sync.(c)}}{\text{elapsed time(c)}}$
$mrr(m)$	per-core memory request (from application cores to memory node m) rate; the number of requests per useful compute cycle. $mrr(m) = \frac{1}{\|C\|} \cdot \frac{\text{\# measured requests(c)}}{\sum_{c \in C} \text{useful cycles(c)}}$
$llc(m)$	per-core LLC miss rate (from application cores to memory node m); the number of LLC misses per useful compute cycle. $llc(m) = \frac{1}{\|C\|} \cdot \sum_{c \in C} \frac{\text{\# llc misses}(c \rightarrow m)}{\text{useful cycles(c)}}$

Table 4.3 Profiled application information.

Name	Description
$\mu(m)$	memory service rate of memory node m
$link(c \rightarrow m)$	delay of interconnection link that connects between core c to memory node m

Table 4.4 Hardware-dependent parameters.

machine's available memory bandwidth into account, we measure the machine's memory service rate $\mu(m)$ for each memory node m and interconnection delay $link(c \rightarrow m)$ that connects between CPU core c and memory node m. These parameters are determined by executing synthetic workloads from the Stream benchmark [64] that can generate memory accesses from specific cores to specific memory nodes. The process takes only a few minutes and is required only once for a given hardware platform.

Analytical Model

As shown in Figure 4.15, the analytical model estimates utilization of CPU cores and memory node bandwidth for a given core allocation X. In Chapter 3, we have shown that, typically memory intensive parallel workloads exhibit almost a Poisson memory access distribution and the memory accesses can be

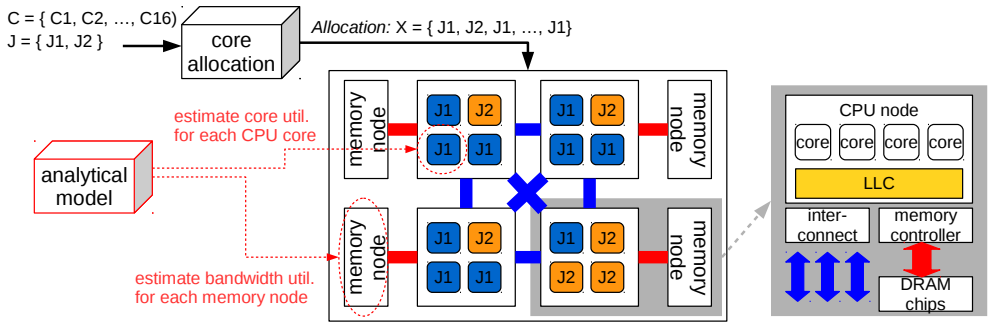
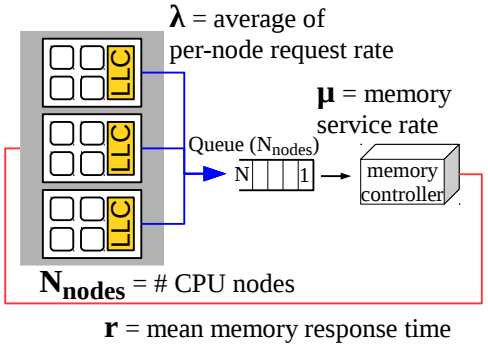


Figure 4.15 Analytical model for multi-socket multicore system.

modeled with the analytical queueing model. We employ the idea of the presented analytical model in Chapter 3 to estimate memory bandwidth usage and CPU utilization in MOCA.

bw_util: To estimate *bw_util*, i.e. the average of utilization of individual memory nodes, we first model each memory node (bandwidth) using an *M/M/1/N/N* queueing system given in Figure 4.16. The queueing system models the mean memory response time and utilization of an individual memory node. For each memory node, a memory controller serves the memory requests issued by the last-level caches (LLC) of all CPU nodes in the system. For the queueing system of memory node m, therefore, we regard each CPU node as a queueing customer and the memory bandwidth is considered to be the queueing server, as shown in Figure 4.16. The arrival rate (i.e. memory request rate) from each CPU node to the memory node m is defined as the summation of per-core memory request from all cores in the CPU node. The service rate of each memory node is computed offline as discused in Section 4.5.2. With the average memory request rate of CPU nodes and the memory service rate, we can compute the memory controller utilization $U(m)$ and the mean response time $r(m)$ using closed-form expression. Then, *bw_util* is computed as the average of utilization on individual memory nodes, i.e. $bw\_util = \frac{1}{|M|} \cdot \sum_{m \in M}(U_{mem}(m))$.



Parameters
$\lambda = \frac{\sum_{c \in C}(X[c].mrr(m))}{N_{nodes}}$
$\mu(m)$ = obtained offline (refer to Table 4.4)
Computed information
$r(m) = \frac{1}{\mu}(\frac{N}{U_{mem}(m)} - \frac{\mu}{\lambda})$
$U_{mem}(m) = 1 - (\sum_{k=0}^{N} \frac{N!}{(N-k)!}(\frac{\lambda}{\mu})^k)^{-1}$

Figure 4.16 Queueing system for an individual memory node m.

cpu_util: As explained in Figure 4.14 (Equation 4.8), core utilization is defined as the rate of useful CPU time over the total time. To model per-core utilization ($U_{core}(c)$), we estimate the overhead of synchronizations and memory responses for a given total time 1, as follows.

$$U_{core}(c) = \frac{1 - sync\ time(c) - memory\ time(c)}{1} \tag{4.10}$$

where *sync time(c)* and *memory time(c)* represents the time taken by synchronization and memory responses, respectively.

The synchronization time *sync time(c)* is computed using a linear equation, assuming that an application's synchronization overhead increases in dependence of the number of allocated cores for the application, as follows.

$$sync\ time(c) = (X[c].sync) \cdot \frac{(X[c].\#cores)}{|C|} \tag{4.11}$$

where $X[c]$ indicates the application assigned to core c, and $X[c].sync$ represents the application's synchronization overhead (the rate of synchronization over total time) when using all available cores in C which is obtained by application profiling. $X[c].\#cores$ represents the total number of cores assigned for application $X[c]$.

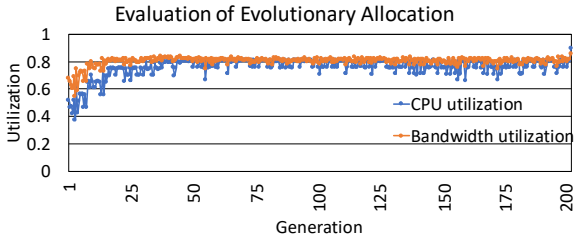


Figure 4.17 An evolutionary process with the analytical model. This scenario considers 16 core resources and four memory nodes each with a service rate of $\mu(m) = 150$ and remote link delay $link = 1/150$. There are two co-located applications each with a memory request rate $mrr(m)$ randomly assigned between 0 and 40. To test 200 generations, the process takes 17.3ms using an Intel Xeon E7-8870 processor core [42].

The time for memory responses is computed based on the number of read misses in the LLC during time 1 and their expected response times, as given by the following equation.

$$memory\ time(c) = \sum_{m \in M} (X[c].llc_m) \cdot (r(m) + link(c \rightarrow m)) \tag{4.12}$$

where $X[c].llc_m$ represents the LLC read miss rate of the core c, and $r(m)$ represents the mean response time at memory node m and is computed by the queueing model in Figure 4.16.

For remote memory accesses from core c to memory m, we add the interconnection delay $link(c \rightarrow m)$. Hence, a high number of remote accesses (llc_m) will decrease utilization of the core. Since our evolutionary method tries to improve both CPU utilization and memory bandwidth, the allocation after the evolution is expected to reduce the number of remote memory accesses in NUMA systems.

Applying to MOCA

The aim of this analytical model is to quickly estimate the resource usage for the evolutionary method. Figure 4.17 shows how the evolution reaches a desired resource utilization goal with the analytical model. The evaluation shows that the analytical model tasks only a few tens of miliseconds to complete testing 200 generations, and we observe that using only 25–50 generations is sufficient to reach a good allocation. Hence, to expedite the evolutionary process, MOCA's model-based allocation tests 50 generations using this analytical model to determine the core allocation.

4.6 Evaluation of MOCA

We evaluate MOCA on a 64-core (8-node) AMD Opteron platform and a 72-core (4-node) Intel Xeon platform. The AMD system has 128 GB, the Intel platform contains 512GB of DRAM memory. The Linux kernel versions are 4.4.35 for AMD and 4.4.0 for the Intel platform.

Target Applications

For co-location, we use applications from NPB [4], Spark bench [55] (Table 4.5). Similar to our previous experiments, we selected *BT*, *FT*, *SP*, and *EP* from NPB3.4 (Fortran). The four NPB applications use input class D with the large problem size. The number of iteration steps is adjusted to obtain similar standalone turnaround times for all applications. We also evaluate other types of workloads from the Spark bench. From Spark bench, we selected *LR* (Linear Regression), *PR* (Page rank), and *SVM* (Support Vector Machine). For these workloads, we use Docker images for resource isolation.

Table 4.5 Target applications and their performance characteristics.

Name	**Application**	*cpu_util*	*bw_util*	*sync.*	**Data size**
BT	BT	High	Medium	Low	Medium
FT	FT	High	High	Low	Huge
SP	SP	High	High	Low	Huge
EP	EP	High	Low	Low	Small
LR	linear regression	High	Medium	Medium	Medium
PR	page rank	Low	Medium	Medium	Medium
SVM	support vector machine	Low	Medium	Medium	Huge

Evaluated Core Allocation Policies

Similar to NuPoCo's experiments, MOCA is evaluated against the four other policies **Baseline**, **EQP**, **C3PO**, and **AB** which are described in Section 4.4.2. Unlike the evaluation in Section 4.4.2, in this experimental setup, we also apply COOP-ULT technique for **C3PO** and **AB** for executing applications in spatial core allocation. Note that, since directly comparing with other existing frameworks is difficult due to the lack of availability, we implemented the existing policies in our core allocation framework.

Experimental Results

We first show that optimizing multi-dimensional resources improves system throughput compared to CPU and speedup-centric approaches and can reduce the total execution time. We evaluate total turnaround time for multiprogrammed scenarios under different core allocation policies. In each co-location scenario, all co-located applications are started at the same time (but finish at different points in time). The first co-location scenario co-locates 2 and 3 workloads from the six applications `BT`, `FT`, `SP`, `LR`, `PR`, `SVM` obtained from NPB and Spark. From these six applications, therefore, $\binom{6}{2}$ plus $\binom{6}{3}$ distinct co-location

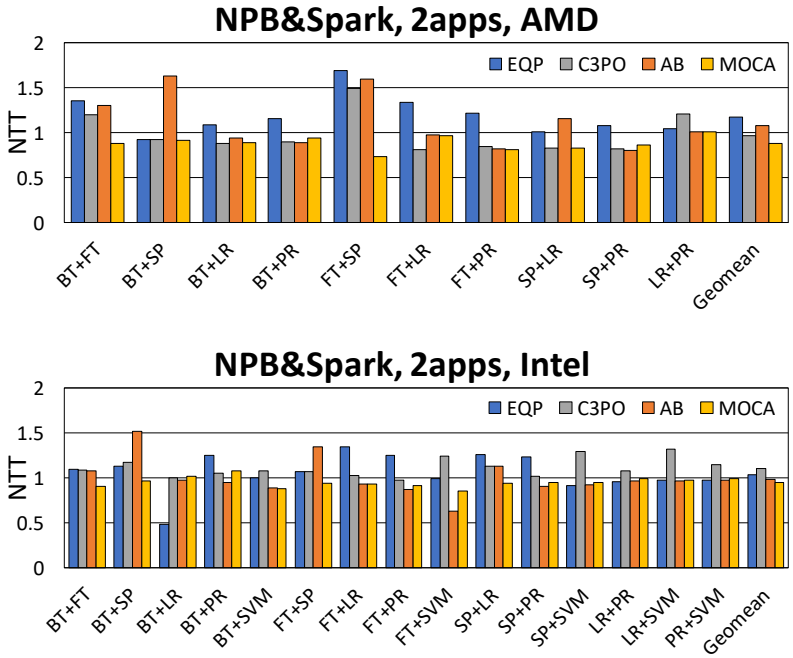


Figure 4.18 Mixes of two applications from NPB and Spark.

scenarios are evaluated. Note that the `SVM` application was excluded on the AMD system because this workload exhibits large memory requirements that cause memory oversubscription when co-located with other workloads on the AMD system. Figure 4.18 and Figure 4.19 show the NTT of the four execution modes `EQP`, `C3PO`, `AB`, and `MOCA` on the AMD and the Intel system for all possible combinations.

The results show that, in general, `MOCA` achieves a better system throughput (shorter execution time) than the default Linux's time-shared execution model. In Figure 4.18 (co-location of two application from NPB and Spark), under the geometric mean, `MOCA` achieves an NTT of 0.88 (12% improvement) on the AMD system and 0.95 (5% improvement) on the Intel platform over the Linux scheduler. In Figure 4.19 (three-application mixes of NPB and Spark),

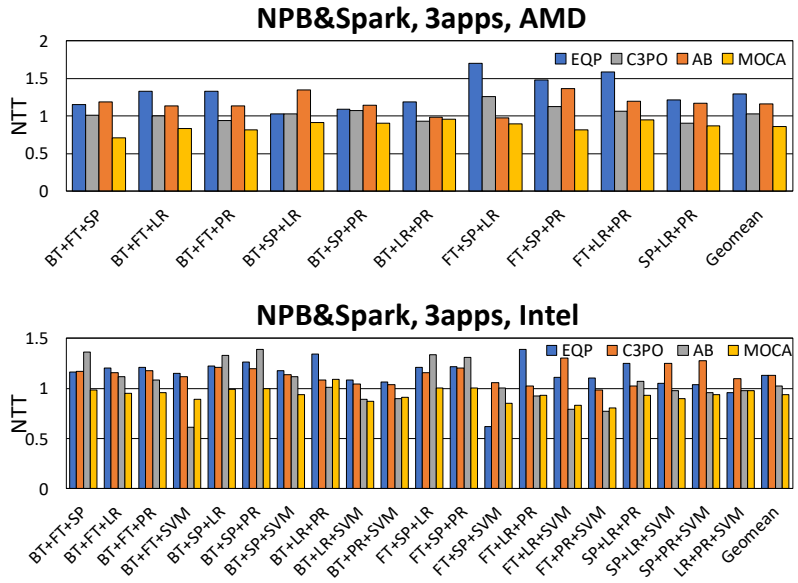


Figure 4.19 Mixes of three applications from NPB and Spark bench.

`MOCA` achieves an NTT of 0.865 (13.5% improvement) on AMD and 0.935 (6.5% improvement) on Intel.

For these experiments, MOCA achieves also significant performance improvements compared to other CPU-centric (`C3PO`) and speedup-centric (`AB`) core allocation policies. For NPB and Spark application mixes (Figures 4.18 and 4.19), `C3PO` and `AB` achieve only marginal improvements or even suffer a performance loss compared to the baseline. These allocations often incur a significantly higher turnaround time compared to the baseline or MOCA. For example, looking at `BT+SP` in Figure 4.18, `AB` exhibits a significantly larger total turnaround time on both systems. This is because `AB` prioritize the highly-scalable application `BT`. The scalable application `BT` finishes quickly, however, this leads to an inefficient execution of `SP` which does not scale well.

In addition to the makespan (total turnaround time) analysis, we also show that MOCA's core allocation improves the overall resource usage compared

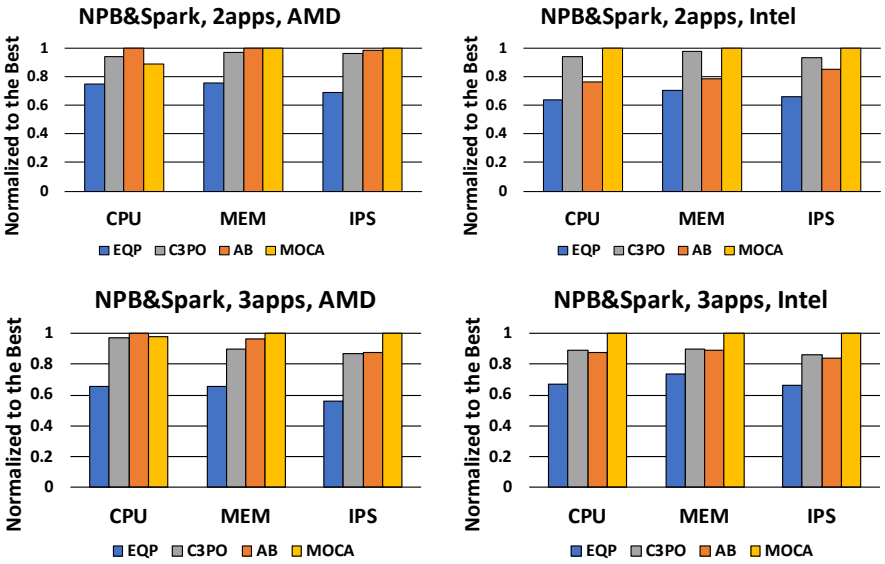


Figure 4.20 Resource usage and IPS comparison for the co-location scenarios. `CPU` refers the number of useful CPU cycles per time (for details, refer to Figure 4.14). `MEM` refers the number of memory requests per time.

to other core allocation policies. To evaluate performance of core allocation policies when all co-located workloads are executing, we compare resource usage of core allocation during 10 seconds on the Intel and 20 seconds on the AMD (considering the different platform throughput). Figure 4.20 shows the resource usage in terms of the number of useful CPU cycles, the number of memory requests and instructions per second for different allocations for the co-location scenarios in Figures 4.18, 4.19. The results show that, for most scenarios, `MOCA` achieves high resource usage. MOCA often achieves low CPU utilization (e.g. NPB&Spark on AMD) compared to other CPU-centric approaches, however, its overall utilization and IPS are higher compared to other policies, as we aim to maximize overall utilization beyond CPUs. In many cases in Figure 4.20, MOCA achieves about 20% improvement in terms of IPS compared to `EQP`,

C3PO, and AB, on both the AMD and the Intel system.

4.7 Discussion

Co-location of multiple parallel jobs on the same multicore machine is increasingly important. In this chapter, we presented parallelism management techniques that leverages the cooperative runtime support (Chapter 2) and the analytical model (Chapte 3) for co-located parallel applications.

4.7.1 Contributions and Limitations

One of our key contribution is that we have shown how we can leverage an analytical model to estimate utilization of multiple resources. As we have seen in Section 4.2.2, if the wall times of the parallel jobs are not given, it is important to maximize overall system utilization to efficiently utilize the given hardware resources and improve the runtime performance. This study provides a useful experience to bridge the gap between analytical (and theoretical) modeling and practical resource management.

One limitation of the study is that, our core allocation policy does not always provide the optimal result depending on the types of parallel jobs and their execution time. For example, if there are a CPU-intensive jobs and a memory-intensive job and if the CPU intensive job has much longer execution time, maximizing CPU utilization may lead to better result in terms of the total turnaround time. To address this issue, parallelism managers need to consider how the parallel job scheduler will co-locates parallel jobs and understand the performance characteristics of the given jobs (based on some information provided by the user or job profiling). The parallelism manager can then intelligently apply the appropriate parallelism management policy or provide a feedback to the (cluster-level) job scheduler to decide a better job co-location

that may benefit from the given parallelism management policy.

4.7.2 Summary

In this chapter, we have presented NuPoCo, a parallelism management framework for co-located parallel workloads on NUMA multi-socket multicore systems. At-runtime performance prediction of CPU and memory controller utilization is used to determine the degree of parallelism for all running workloads with the goal of maximizing system resource utilization. The evaluations show that the NuPoCo framework executes multiple OpenMP parallel applications with a significantly shorter total turnaround time than the Linux time-sharing model and a existing parallelism management policy maximizing the CPU utilization. Then, we presented MOCA, an elastic core allocation approach that leverages the idea of evolution in genetic algorithms to optimize the utilization of the multi-dimensional CPU and memory resources on NUMA multi-socket multicore systems. To provide an efficient parallel execution, MOCA uses two runtime techniques, model-based allocation and cooperative user-level tasking, to allow parallel programs to dynamically change the parallelism with low overhead and to evaluate core allocations without executing on real-hardware. Evaluated with various scenarios of co-located OpenMP applications on a 64-core AMD and a 72-core Intel machine, our core allocation achieves a reduction of the total turnaround time by 5-30% compared to the default Linux scheduler and state-of-the-art existing parallelism management policies focusing only on CPU utilization.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Shared-memory multiprocessor platforms are becoming more complex with an increasing number of CPU sockets and memory controllers. A single application is typically not able to achieve the peak system performance on such complex hardware platforms. It is therefore gaining importance to execute multiple parallel applications simultaneously on the same multicore machine for an increased resource utilization. Malleable parallel execution and spatial core allocation are key to execute multiple co-located parallel applications efficiently on multicore servers that contain an increasingly large number of core resources. However, it still remains a challenge for existing runtime techniques to execute co-located parallel applications efficiently under the spatial scheduling approach.

In this thesis, we presented novel runtime and resource management techniques for co-located parallel applications. First, we presented cooperative runtime system that provides malleable parallel execution under dynamic spatial

core allocation. The runtime technique provides an ability for OpenMP parallel programs to dynamically adapt their degree of parallelism to the varying core resource availability through efficient user-level tasking and dynamic scheduling. The experimental results in Chapter 2 have shown that, with changing core resource availability the cooperative runtime system can execute parallel applications more efficiently (i.e. 20–30% shorter execution time on average) compared to the traditional execution model that does not adjust the degree of parallelism of the application. Another contribution of this thesis is an analytical performance model that can estimate resource utilization and the performance of parallel programs in dependence of the allocated core resources. The analytical model is based on the *M/M/1/N/N* queueing model to analytically compute memory bandwidth and CPU utilization using closed-form expressions. Evaluations in Chapter 3 have shown that our model can predict the speedup of parallel programs with a high accuracy (less than 10% of percentage errors on average) for various OpenMP parallel loops. Thanks to the simplicity of the model, the model can also be applied to resource management problems. Based on the cooperative parallel runtime support and the analytical model, we presented core allocation techniques to optimize system utilization by managing core resources between co-located parallel applications. The core allocation technique particularly focuses on optimizing utilization of multi-dimensional resources of CPU cores and memory bandwidth on multi-socket systems. Evaluated with various scenarios of co-located parallel applications, our core allocation achieves a reduction of the total turnaround time by 5-30% compared to the traditional execution under Linux's time-shared scheduler.

In conclusion, in this thesis, we have shown how we can achieve efficient malleable parallel execution through a runtime-level support and how we can analytically model the performance of parallel programs using queueing sys-

tems. Then, we have shown that, with an appropriate core allocation we can improve resource utilization and runtime performance for co-located parallel applications. I hope that this research can provide valuable experience for future data centers and HPC systems to provide an efficient runtime environment.

5.2 Future work

There are a number of future research directions. The first research direction is improving the core allocation policy and algorithm for better runtime performance and resource utilization. The other interesting direction is applying our runtime techniques to HPC systems that employ an idea of parallel job co-scheduling.

5.2.1 Improving Multi-Objective Core Allocation

The core allocation techniques presented in this thesis aim to optimize utilization of multi-dimensional resources of CPU cores and memory bandwidth by using a greedy algorithm or an evolutionary approach. While the core allocation approach is more sophisticated than other previous runtime core management policies that focus only on CPU utilization or application speedup. However, there are still rooms for improvements in the multi-resource scheduling.

First, we can improve the core allocation research by considering utilization of other shared resources (e.g. shared caches) beyond the CPU and memory bandwidth. Specially, we expect the evolutionary method based on genetic algorithms can also be used for optimizing multi-dimensional resources. Moreover, the system may have scheduling constraints. For example, applications may have different service level agreements or different entitles. The core allocation algorithms used in this thesis focused only on maximizing system utilization without considering the different entitlements of parallel jobs. It is an impor-

tant research direction to consider different entitlements of different jobs and their fairness.

5.2.2 Co-Scheduling of Parallel Jobs for HPC Systems

The HPC community is heading towards the exascale era where power and energy efficiency is of utmost importance. Conventional HPC schedulers, however, will not be able to achieve maximal system efficiency because they provide distinct multicore nodes between parallel jobs, which results in under-utilization of multicore nodes. Executing multiple parallel jobs on the same machine is becoming more important to achieve maximal energy efficiency for exascale computing. However, the co-location approach will raise a number of research challenges because the approach requires changes to conventional FCFS (and backfilling) scheduling.

First, to apply job co-location in the node-level, schedulers need to predict performance of parallel jobs if they are executed on the same multicore node. The analytical performance modeling for OpenMP parallel loops will be useful to develop the necessary performance profiling and prediction tools. Moreover, to be able to dynamically change the number of threads (i.e. the number of allocated resources) for parallel jobs, we can leverage our cooperative runtime system presented in Chapter 2. The co-scheduling approach is not fit for conventional FCFS and backfilling schedulers because these schedulers assume a fixed amount of resources and job runtime which are informed by users. To apply a co-scheduling scheme, we need a different job description model and different pricing model, which is also an interesting topic of future research.

Appendix A

Additional Experiments for the Performance Model

This appendix contains additional experimental results of the performance model presented in Chapter 3. In addition, we provide a goodness of fit test to justify the model's assumptions that memory requests follow a Poisson distribution and that memory controllers exhibit an exponential service time.

A.1 Memory Access Distribution and Poisson Distribution

A.1.1 Memory Access Distribution

Results in Section 3.2.3 plot the distribution of the number of memory accesses per unit time for a number of parallel loops. The full results of all 24 target parallel loops in Chapter 3 are given by the following Figure A.1 (PMF of memory request rates (AMD)) and Figure A.2 (PMF of memory requests rate (Intel)).

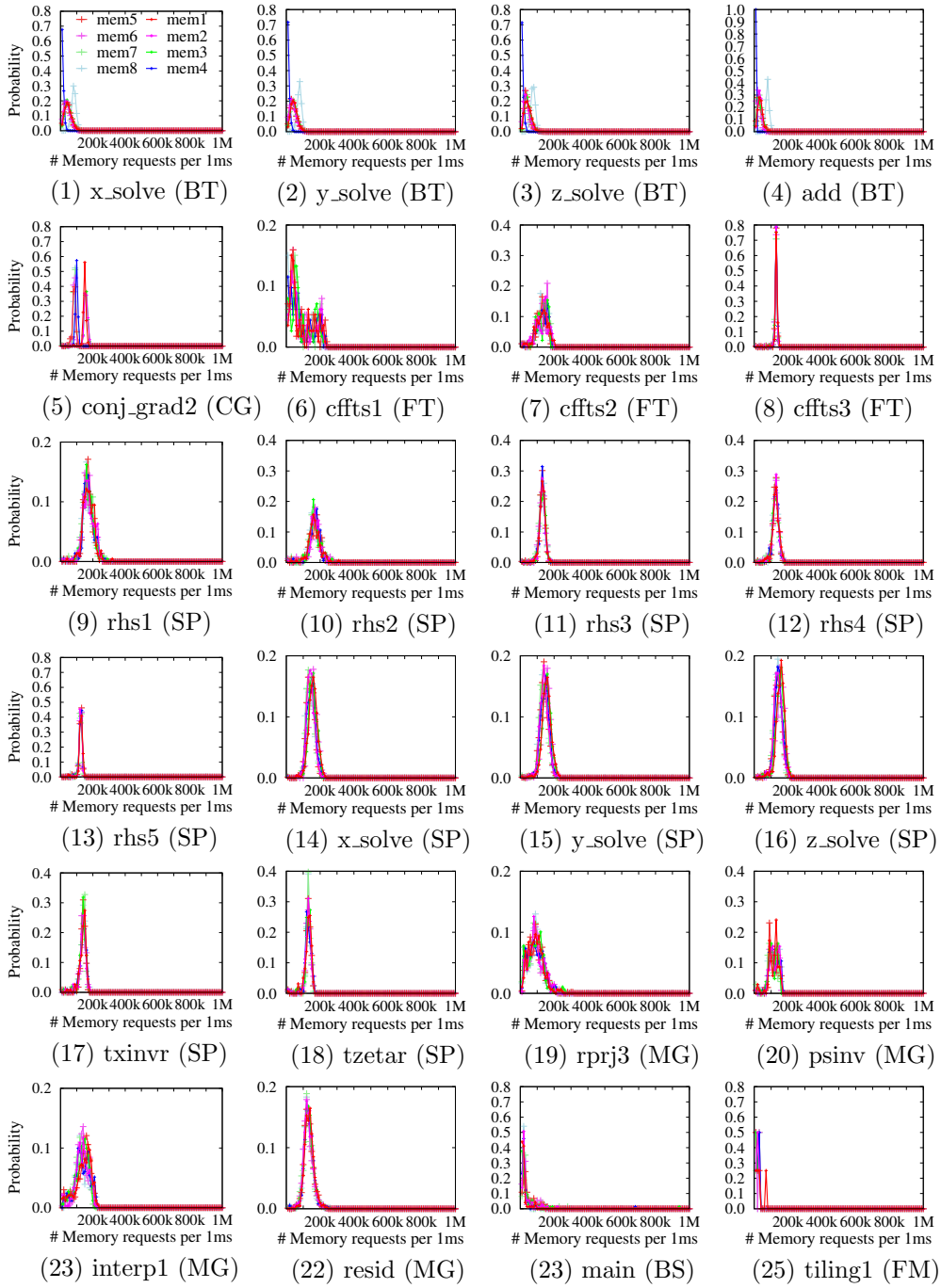


Figure A.1 Probability mass function (PMF) of the number of memory requests per time of the parallel loops in the main paper at each memory node on the 64-core AMD platform.

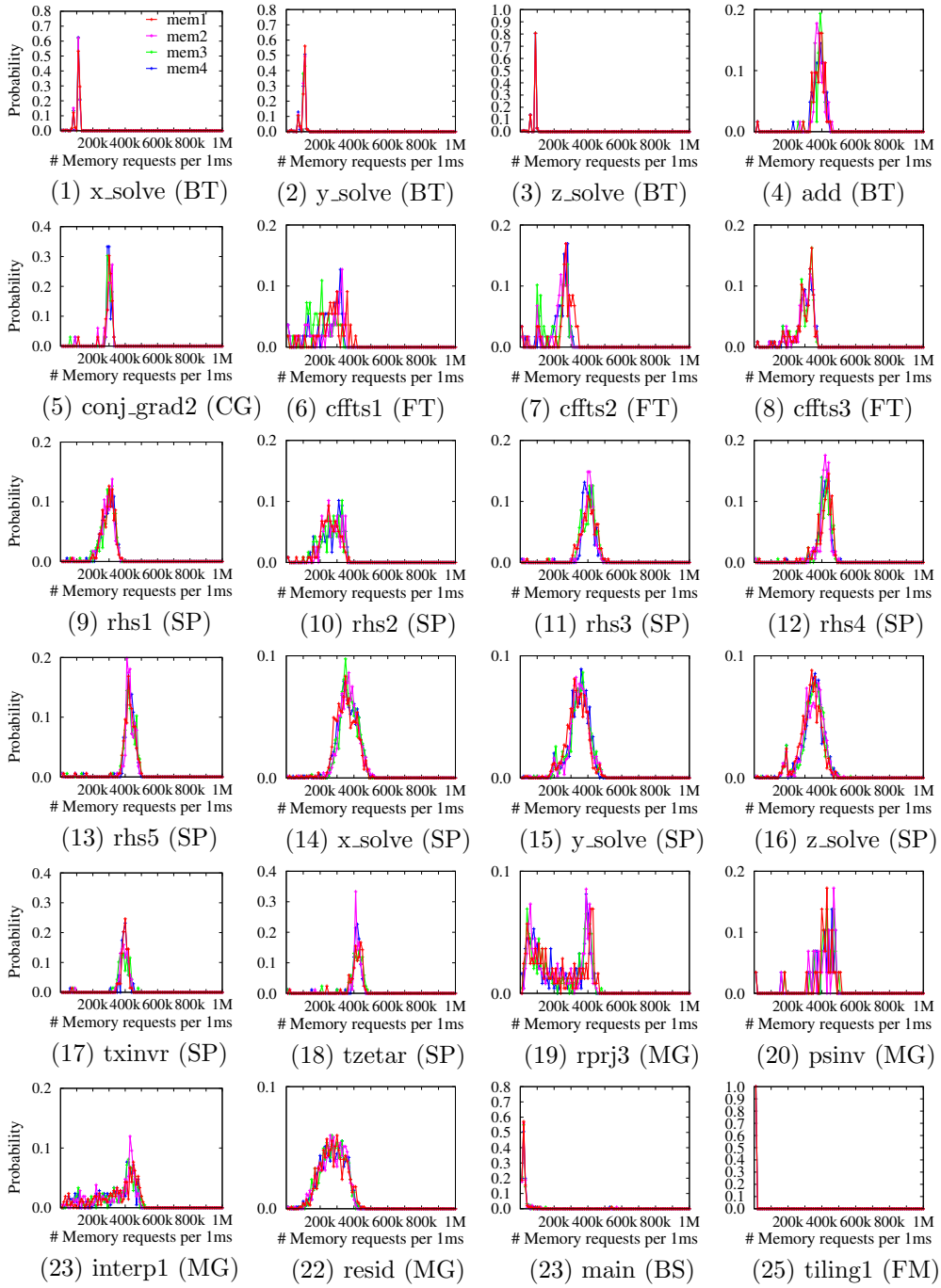


Figure A.2 Probability mass function (PMF) of the number of memory requests per time of the parallel loops in the main paper at each memory node on the 72-core Intel platform.

A.1.2 Kolmogorov Smirnov Test

A goodness of fit test was conducted in order to justify the assumptions of Poisson-distributed memory requests and exponential service times (i.e., that the number of completed memory operations per time follows a Poisson distribution). The test was conducted for all target parallel workloads and both target platforms (the 64-core AMD and 72-core Intel system).

We used the two-sample Kolmogorov-Smirnov (KS) test to verify the Poisson distribution mathematically. From two different datasets, the KS test computes the statistical value (we used `scipy.stats.ks_2samp` [77]), where a lower value means that the two datasets are similar. For each parallel loop, a histogram with ten categories for a number of data samples containing the number of memory requests per unit time is used for comparison with a Poisson distribution. Overall, the majority of parallel loops exhibit a Poisson distribution. Detailed results are listed in the following tables.

Analyzing Poisson memory requests

- Figure A.3: Histogram of memory requests (AMD).
- Figure A.4: Histogram of memory requests (Intel).
- Table A.1: KS test results (AMD).
- Table A.2: KS test results (Intel).

Analyzing exponential memory service times

- Figure A.6: Histogram of memory services (AMD).
- Figure A.8: Histogram of memory services (Intel).
- Table A.3: KS test results (AMD and Intel).

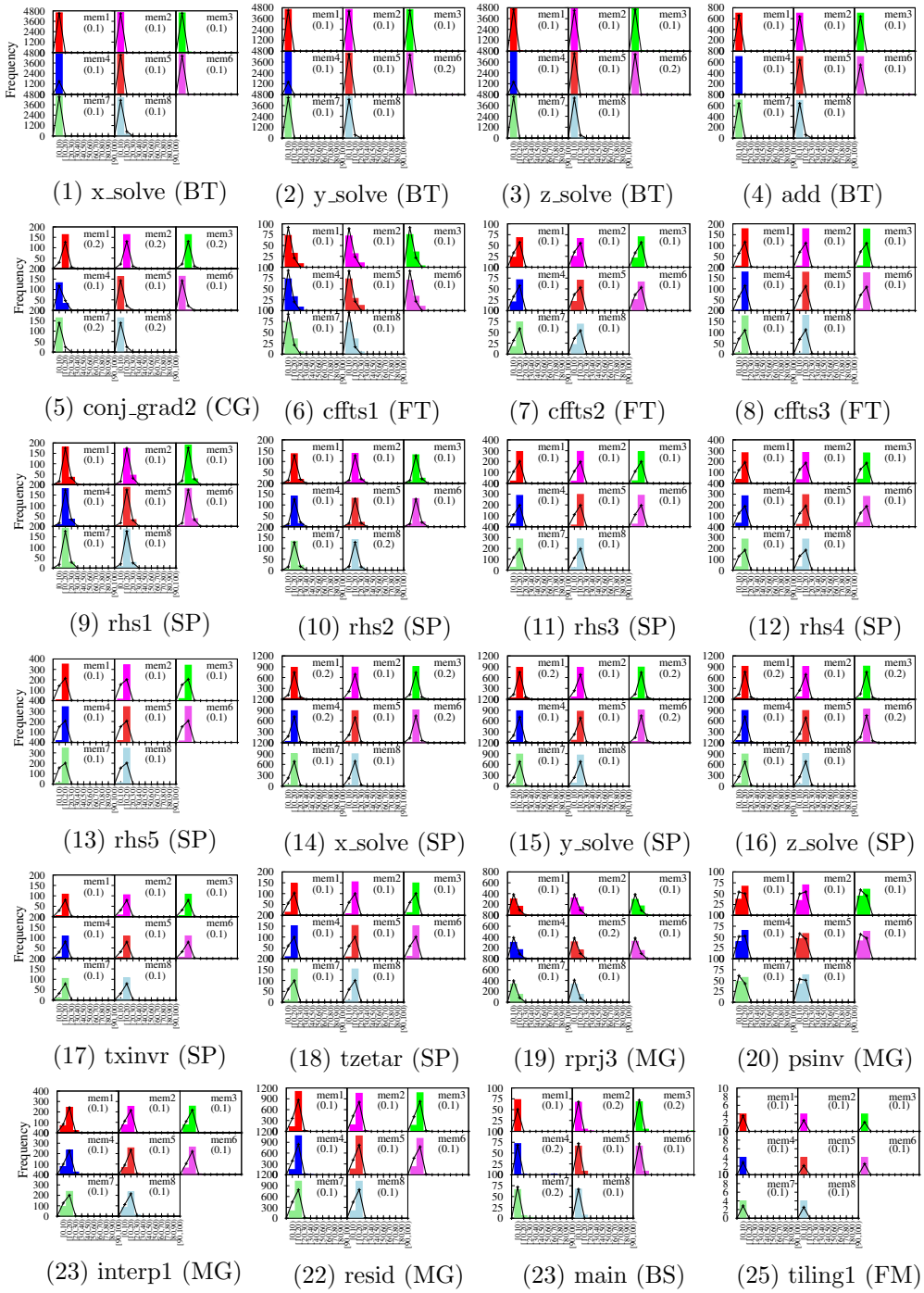


Figure A.3 Histogram of the measured number of memory requests per time (1us) of the parallel loops in the main paper at each memory node on the 64-core AMD platform, and comparison with a Poisson distribution.

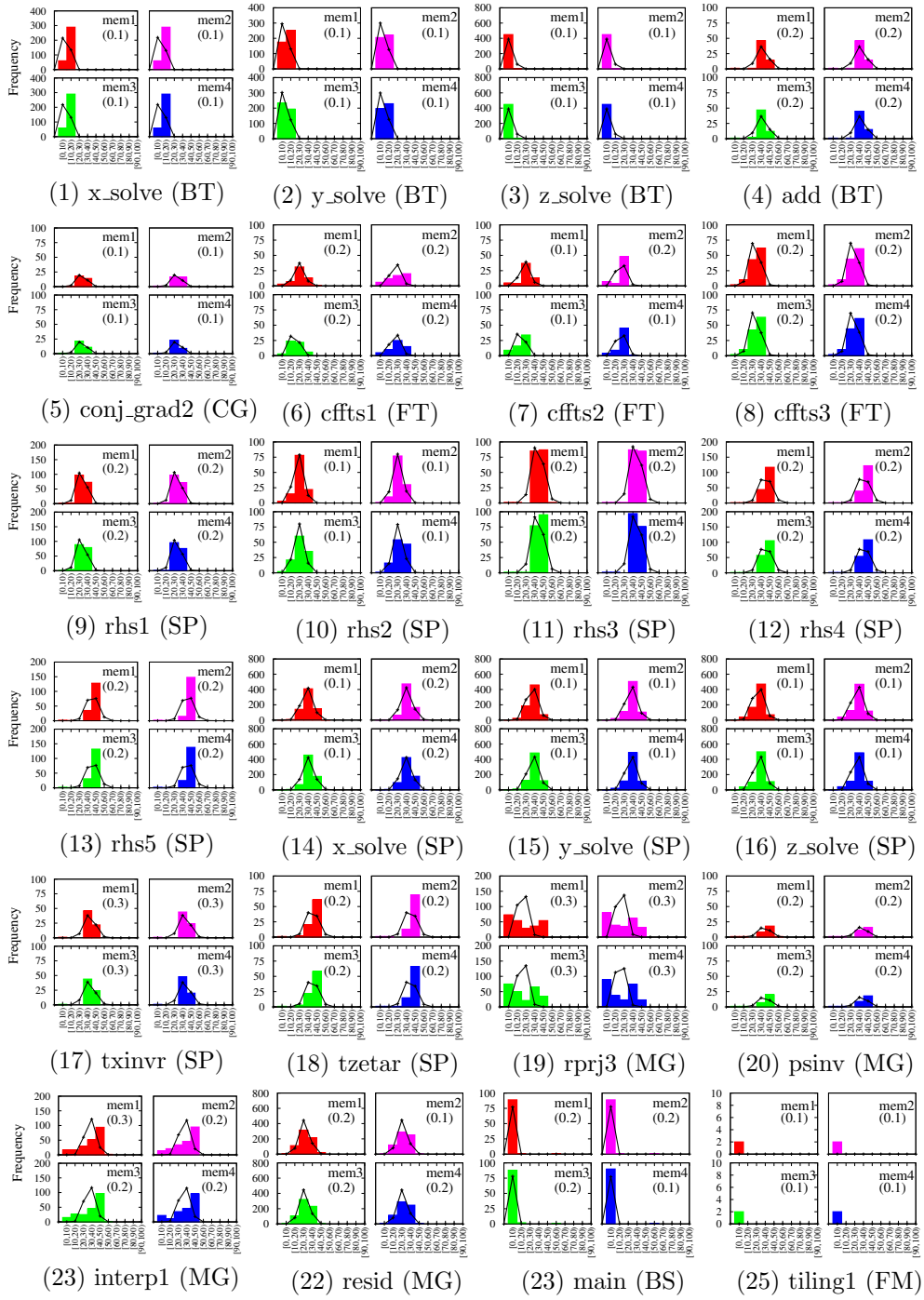


Figure A.4 Histogram of the measured number of memory requests per time (1us) of the parallel loops in the main paper at each memory node on the 72-core Intel platform, and comparison with a Poisson distribution.

Loop	N1	N2	N3	N4	N5	N6	N7	N8	Avg.	Geo.	Crit.	Passed
x_solve (BT)	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.19	✓
y_solve (BT)	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.2	✓
z_solve (BT)	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.2	✓
add (BT)	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.19	✓
conj_grad2 (CG)	0.3	0.3	0.3	0.1	0.1	0.1	0.2	0.2	0.2	0.18	0.19	✓
cffts1 (FT)	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.19	✓
cffts2 (FT)	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.2	✓
cffts3 (FT)	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.19	
rhs1 (SP)	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.19	
rhs2 (SP)	0.2	0.2	0.2	0.1	0.2	0.2	0.2	0.2	0.19	0.18	0.19	✓
rhs3 (SP)	0.1	0.1	0.1	0.2	0.1	0.1	0.1	0.1	0.11	0.11	0.19	✓
rhs4 (SP)	0.2	0.2	0.1	0.1	0.1	0.2	0.2	0.2	0.16	0.15	0.19	✓
rhs5 (SP)	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.19	✓
x_solve (SP)	0.1	0.2	0.2	0.2	0.2	0.1	0.2	0.2	0.17	0.17	0.19	✓
y_solve (SP)	0.2	0.1	0.2	0.2	0.1	0.2	0.2	0.1	0.16	0.15	0.19	✓
z_solve (SP)	0.2	0.1	0.1	0.2	0.1	0.2	0.2	0.1	0.15	0.14	0.19	✓
txinvr (SP)	0.2	0.2	0.2	0.2	0.2	0.2	0.1	0.1	0.18	0.17	0.19	✓
tzetar (SP)	0.2	0.2	0.2	0.2	0.2	0.2	0.1	0.2	0.19	0.18	0.19	✓
rprj3 (MG)	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.19	✓
psinv (MG)	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.2	✓
interp1 (MG)	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.19	✓
resid (MG)	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.19	✓
main (BS)	0.1	0.2	0.1	0.3	0.3	0.2	0.2	0.2	0.2	0.19	0.23	✓
tiling1 (FM)	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	2.0	✓
average	0.14	0.14	0.14	0.14	0.13	0.14	0.14	0.13				
geomean	0.13	0.13	0.13	0.13	0.12	0.13	0.13	0.13				

Table A.1 Results of the two-sample Kolmogorov-Smirnov test for the targeted parallel loops on the 64-core AMD platform. The test does not reject the hypothesis that the datasets follow a Poisson distribution if the statistical value is less than the critical value with significance level of $\alpha = 0.05$.

Loop	Node1	Node2	Node3	Node4	Avg.	Geo.	Crit.	Passed
x_solve (BT)	0.2	0.2	0.2	0.1	0.18	0.17	0.19	✓
y_solve (BT)	0.1	0.1	0.1	0.1	0.1	0.1	0.19	✓
z_solve (BT)	0.1	0.1	0.1	0.1	0.1	0.1	0.19	✓
add (BT)	0.3	0.3	0.3	0.3	0.3	0.3	0.24	
conj_grad2 (CG)	0.2	0.1	0.1	0.1	0.13	0.12	0.33	✓
cffts1 (FT)	0.2	0.2	0.2	0.2	0.2	0.2	0.25	✓
cffts2 (FT)	0.2	0.2	0.2	0.2	0.2	0.2	0.23	✓
cffts3 (FT)	0.2	0.2	0.2	0.2	0.2	0.2	0.19	
rhs1 (SP)	0.4	0.4	0.3	0.3	0.35	0.35	0.19	
rhs2 (SP)	0.2	0.2	0.2	0.2	0.2	0.2	0.19	✓
rhs3 (SP)	0.4	0.4	0.4	0.4	0.4	0.4	0.19	
rhs4 (SP)	0.3	0.4	0.3	0.4	0.35	0.35	0.19	
rhs5 (SP)	0.5	0.4	0.5	0.5	0.47	0.47	0.19	
x_solve (SP)	0.1	0.1	0.2	0.1	0.13	0.12	0.19	✓
y_solve (SP)	0.3	0.2	0.2	0.2	0.22	0.22	0.19	
z_solve (SP)	0.2	0.3	0.3	0.2	0.25	0.24	0.19	
txinvr (SP)	0.2	0.3	0.2	0.1	0.2	0.19	0.23	✓
tzetar (SP)	0.2	0.2	0.2	0.2	0.2	0.2	0.21	✓
rprj3 (MG)	0.3	0.3	0.3	0.3	0.3	0.3	0.19	
psinv (MG)	0.2	0.1	0.1	0.1	0.13	0.12	0.34	✓
interp1 (MG)	0.2	0.2	0.1	0.2	0.17	0.17	0.19	✓
resid (MG)	0.2	0.2	0.1	0.1	0.15	0.14	0.19	✓
main (BS)	0.2	0.1	0.1	0.1	0.13	0.12	0.21	✓
tiling1 (FM)	0.1	0.1	0.1	0.1	0.1	0.1	-	
average	0.23	0.22	0.21	0.2				
geomean	0.21	0.2	0.18	0.17				

Table A.2 Results of the two-sample Kolmogorov-Smirnov test for the targeted parallel loops on the 72-core Intel platform. The test does not reject the hypothesis that the datasets follow a Poisson distribution if the statistical value is less than the critical value with significance level of $\alpha = 0.05$. The runtime of *tiling1* is too short to provide a meaningful number of samples and has thus been omitted.

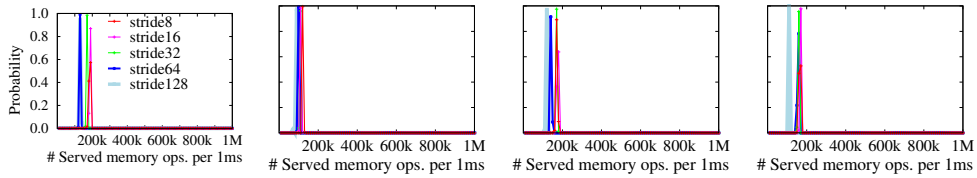


Figure A.5 PDF of the number of served memory operations for the synthetic workloads from the memory system on the AMD platform.

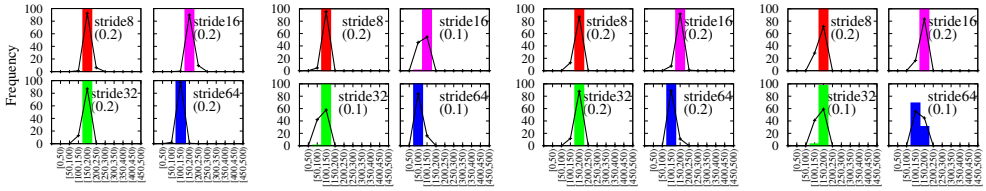


Figure A.6 Histogram of the number of served memory operations for the synthetic workloads from the memory system on the AMD platform, and comparison with a Poisson distribution.

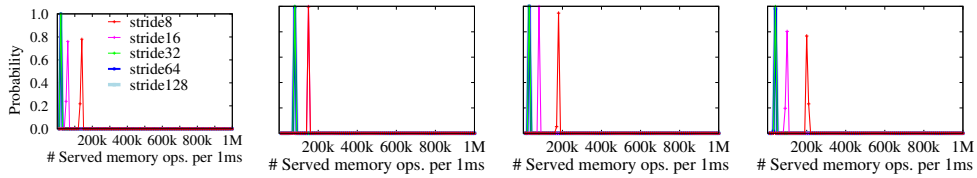


Figure A.7 PDF of the number of served memory operations for the synthetic workloads from the memory system on the Intel platform.

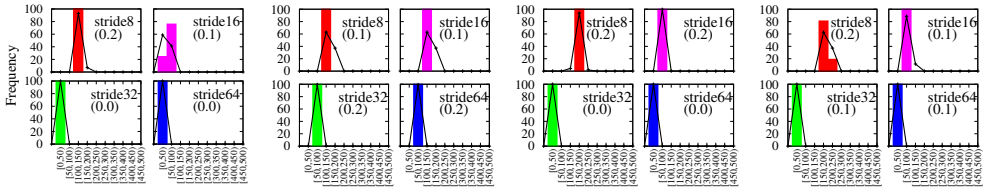


Figure A.8 Histogram of the number of served memory operations for the synthetic workloads from the memory system on the Intel platform, and comparison with a Poisson distribution.

Arch.	Workload	Stride				
		8	**16**	**32**	**64**	**128**
AMD	StreamWrite	0.20	0.20	0.20	0.20	0.20
	StreamLoad	0.20	0.10 ✓	0.10 ✓	0.10 ✓	0.10 ✓
	StreamCopy	0.20	0.20	0.20	0.20	0.20
	StreamAdd	0.20	0.20	0.10 ✓	0.10 ✓	0.10 ✓
Intel	StreamWrite	0.20	0.10 ✓	0.00 ✓	0.00 ✓	0.00 ✓
	StreamLoad	0.10 ✓	0.10 ✓	0.20	0.20	0.20
	StreamCopy	0.20	0.20	0.00 ✓	0.00 ✓	0.00 ✓
	StreamAdd	0.20	0.10 ✓	0.10 ✓	0.10 ✓	0.10 ✓

Table A.3 Results of the two-sample Kolmogorov-Smirnov test to check whether the memory services times of the Stream microbenchmarks follow a Poisson distribution. The critical value is 0.19 with $\alpha = 0.05$. A checkmark (D) indicates that the test has been passed. 50% of the benchmarks pass the test, and the other 50% fail the test only by a minimal margin, i.e., are very close to a Poisson distribution.

A.2 Additional Performance Modeling Results

A.2.1 Results with Intel Hyperthreading

In Chapter 3, Intel's Hyperthreading was not enabled to minimize the intra-node resource interference. With Hyperthreading enabled, the presented model loses some accuracy (especially *LoopPerf-T*) but is still able to capture the trend of a loop's speedup as shown in Figure A.9.

A.2.2 Results with Cooperative User-Level Tasking

The evaluation of the presented analytical model in Chapter 3 assumed workloads with dynamic loop scheduling. In other words, the execution scenario is almost the same with the execution under `COOP-DYN` discussed in Chapter 2. In principle, the model can be applied to model workloads with COOP-ULT since the COOP-ULT technique is also based on dynamic scheduling. Here, to show that our model can support COOP-ULT, we additionally provide experiments to evaluate the accuracy of the performance model for static parallel loops executed with `COOP-ULT`. The MAPE values and the speedup curves are given in Figure A.10 and Figure A.11. The results show that, the proposed techniques *LoopPerf-S* and *LoopPerf-T* achieve less than 10% of MAPE values. The speedup curves are given in Figure A.10 and Figure A.11. We observe that *LoopPerf* also predict the speedup of the parallel loops without showing a significant difference compared to the results in Figure 3.14 and Figure 3.15.

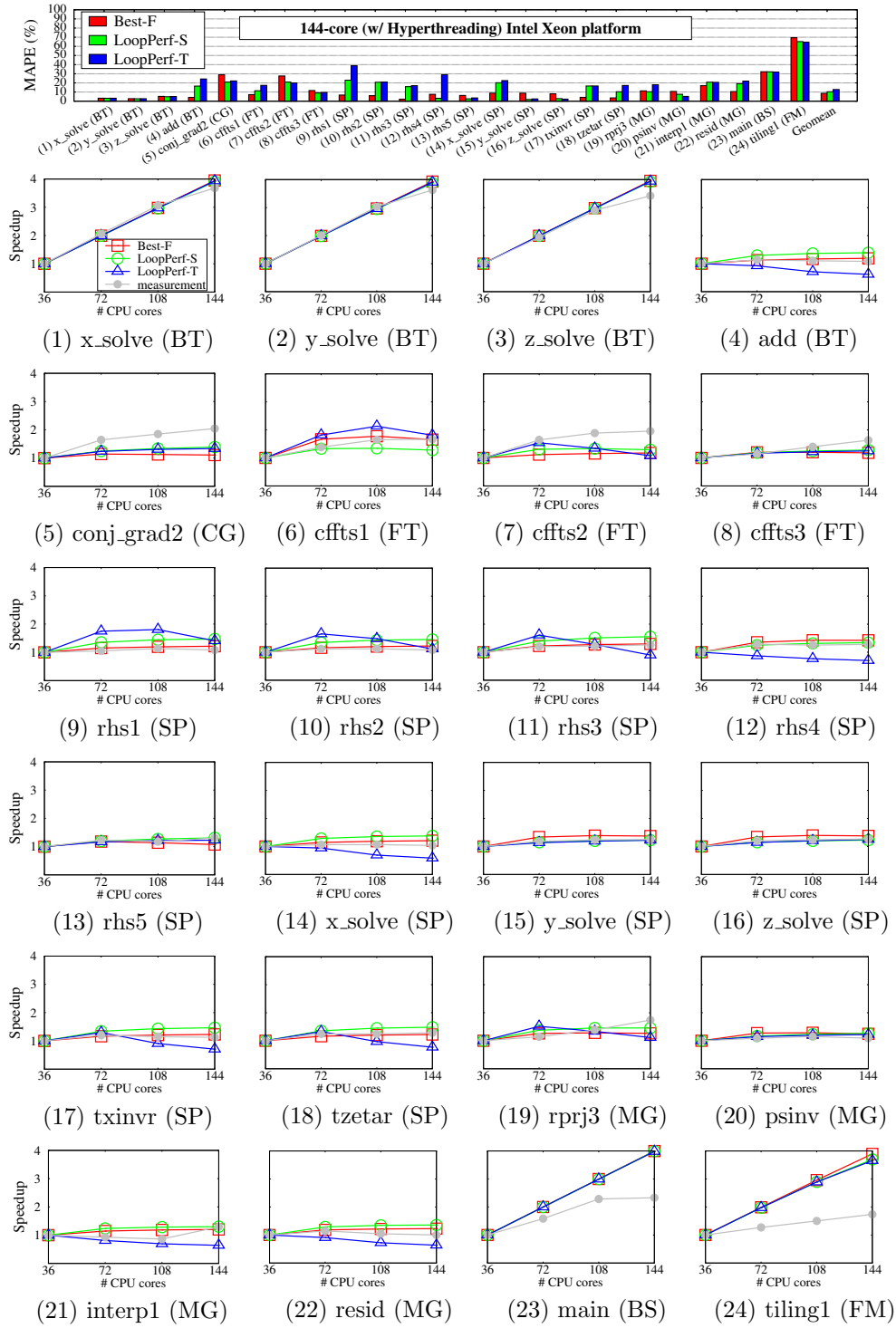


Figure A.9 Predicted versus measured speedup of the parallel loops on the Intel platform with Hyperthreading enabled (144 cores).

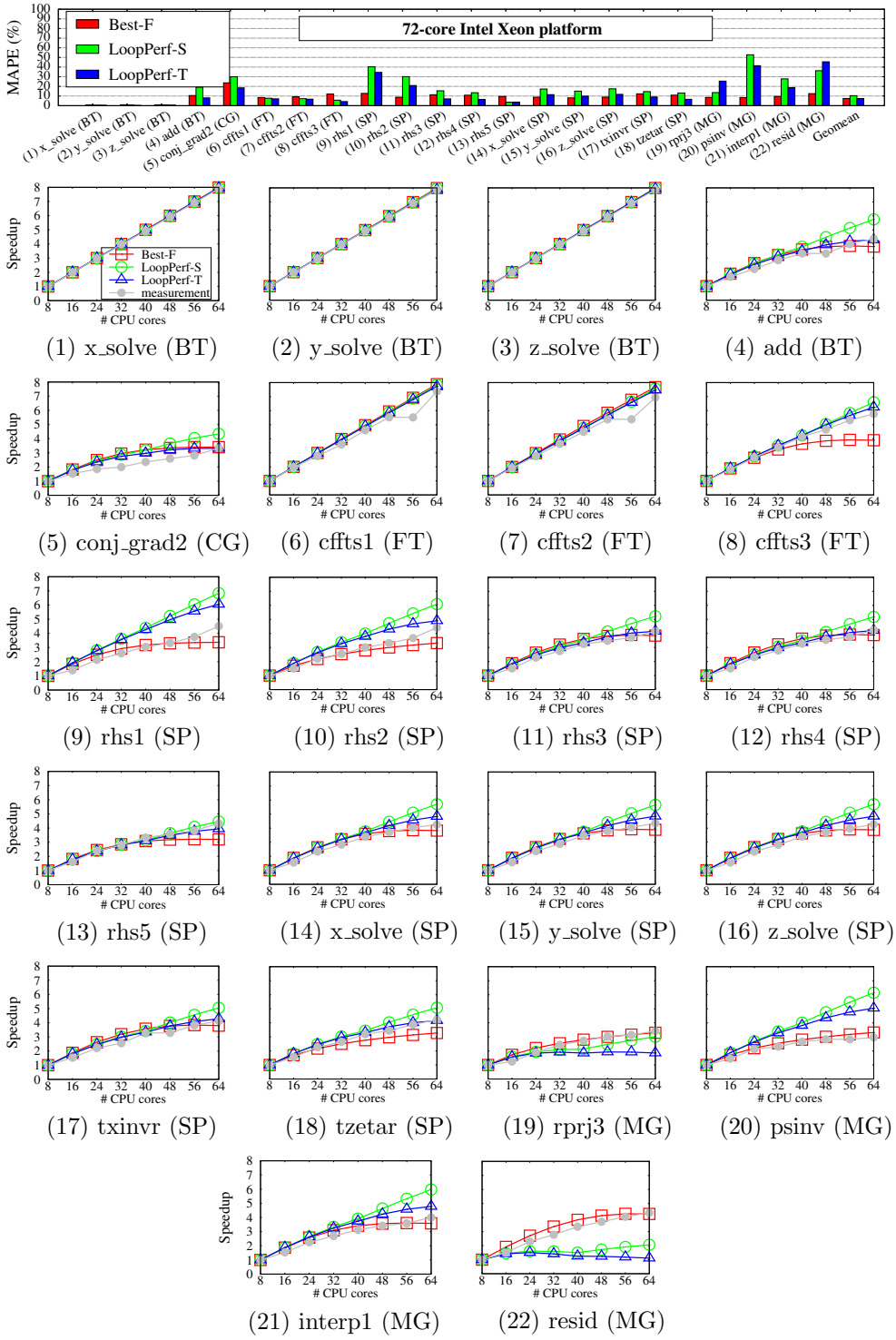


Figure A.10 The predicted speedup and the measured speedup of the parallel loops executed under COOP-ULT on the 64-core AMD platform.

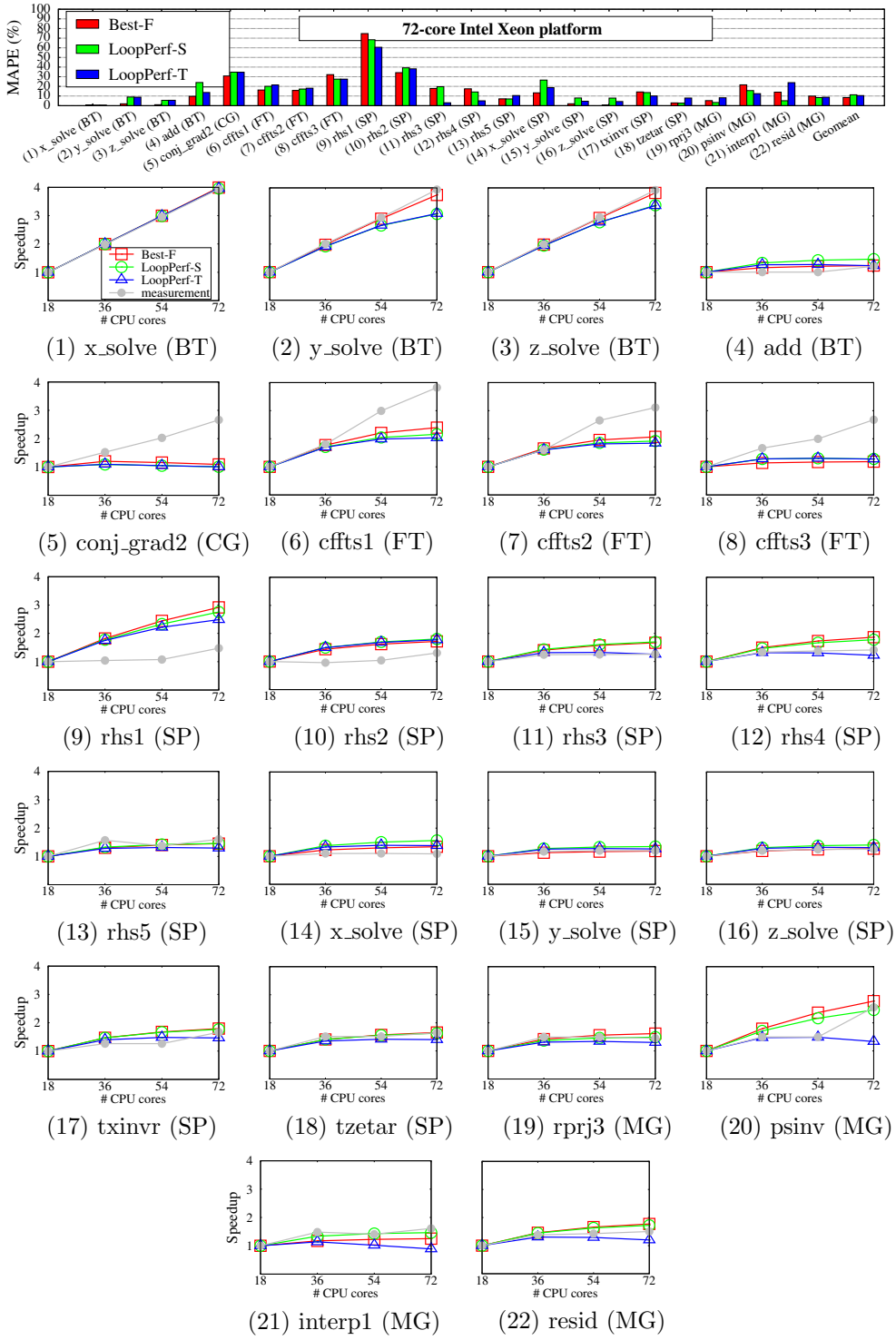


Figure A.11 The predicted speedup and the measured speedup of the parallel loops executed under COOP-ULT on the 72-core Intel platform.

A.2.3 Results with Other Loop Schedulers

In our previous PACT 2016 paper [17], the model (based on the *Best-F* method) has also been evaluated for other scheduling methods including static, guided and dynamic scheduling as well as for different memory allocation schemes. The results have shown 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. For the evaluation, we select the kernels of six applications (BT, CG, EP, FT, MG, SP) from SNU-NPB3.3 OpenCL implementation [80] of the NAS parallel benchmarks on the 64-core AMD system used in this thesis.

Table A.4 shows the results for self scheduling (i.e. dynamic scheduling, but chunk size is fix to 1), guided, and static scheduling. Self- and static scheduling suffer from scalability issues or load imbalances and thus exhibit higher error rates of 10-12%. For example, `FT.init_ui` and `SP.compute_rhs1` display a very high error (427 and 78%) in Table A.4 compared to Li or static scheduling because the central scheduler becomes the bottleneck when delivering work-groups to an increasing number of worker cores. Similarly, `MG.kernel_resid` and `MG.kernel_rprj3` show a high error (66 and 117%) in the static scheduler because of load imbalances caused by the static work distribution. This is an expected limitation of the proposed model as we assume memory contention to be the only limiting factor and do not account for bottlenecks in the scheduler or imbalanced workloads.

A.2.4 Results with Different Number of Memory Nodes

In the next two experiments, the number of available memory nodes are reduced from originally eight to four and two (`Memory-0123` and `Memory-01`, respectively) in order to see how the model performs under increasing memory

Kernels	Self			Guided			Static		
	scal.	err(%)	R^2	scal.	err(%)	R^2	scal.	err(%)	R^2
BT.z_solve	7.19	4.21	0.97	7.21	4.22	0.97	7.15	5.14	0.96
CG.conj_grad_2	3.95	24.36	0.7	4.52	9.85	0.92	3.63	31.51	0.11
CG.conj_grad_6	4.14	22.11	0.64	4.04	28.19	0.55	3.87	19.93	0.72
EP.embar	7.63	1.82	0.99	7.71	1.6	1.0	7.97	1.36	1.0
FT.init_ui	0.92	426.77	-1235.12	3.38	19.3	0.53	2.9	26.99	0.54
FT.compute_indexmap	5.84	4.27	0.88	6.51	2.58	0.95	5.2	9.1	0.76
FT.compute_initial_cond	6.76	4.4	0.94	7.06	3.59	0.97	7.17	3.74	0.97
FT.cffts1	6.6	13.89	0.79	4.09	11.47	0.9	2.77	33.81	0.49
FT.cffts2	5.53	4.85	0.97	5.14	4.34	0.99	2.87	34.96	0.44
FT.cffts3	5.83	3.77	0.98	5.56	3.48	0.99	4.15	16.67	0.8
SP.exact_rhs2	4.23	23.34	0.59	4.11	9.64	0.93	4.3	11.32	0.9
SP.exact_rhs3	5.57	6.39	0.93	5.54	5.19	0.95	5.71	7.53	0.92
SP.exact_rhs4	6.05	5.83	0.95	5.92	3.77	0.98	6.28	5.45	0.94
SP.compute_rhs1	2.59	78.38	-9.67	5.16	7.55	0.94	5.91	3.47	0.98
SP.compute_rhs2	5.1	11.75	0.77	4.02	12.45	0.88	4.33	15.78	0.83
SP.compute_rhs3	4.36	6.02	0.97	4.37	6.2	0.97	4.46	6.25	0.97
SP.compute_rhs4	5.84	3.23	0.99	6.33	3.64	0.98	6.29	2.88	0.99
SP.compute_rhs5	6.56	3.01	0.98	7.42	2.18	0.99	7.52	2.04	0.99
SP.z_solve	3.3	44.15	-1.13	4.52	6.93	0.96	4.77	6.97	0.96
SP.tzetar	4.08	8.77	0.91	4.06	9.52	0.9	4.07	9.07	0.9
MG.kernel_resid	3.97	20.77	0.54	4.13	12.03	0.9	1.07	66.14	0.29
MG.kernel_rprj3	3.72	23.92	0.44	4.08	25.52	0.6	1.27	116.85	0.32
MG.kernel_interp_1	3.77	6.19	0.94	3.54	11.44	0.86	2.91	24.07	-0.42
MG.kernel_psinv	3.67	6.38	0.96	4.22	9.22	0.92	3.62	9.55	0.92
Average	**4.54**	**10.47**	**-51.17**	**4.96**	**6.84**	**0.90**	**4.14**	**10.92**	**0.72**

Table A.4 Scalability prediction accuracy for different work schedulers.

contention. We observe that the error of the estimation increases as the number of memory nodes decreases. `Memory-01` in particular diverges a lot from the predicted value with an average MAPE of 14.7% and an R^2 of -10.3. With only a small number of available memory nodes, memory-intensive kernels often do not scale at all, i.e., the scalability curve is (almost) flat. Even if the model captures the trend of a kernel's scalability well, modest prediction errors can lead to large percentage errors and negative R^2 values. This is, however, rather a limitation of the evaluation metrics than of the model as the visualization of the predicted versus the actual speedup reveals: the prediction of, for example,

Kernels	Memory 0-3			Memory 0-1		
	scal.	err(%)	R^2	scal.	err(%)	R^2
BT.z_solve	6.99	4.43	0.96	3.87	23.88	-0.66
CG.conj_grad_2	2.36	15.96	0.43	1.27	28.77	-16.7
CG.conj_grad_6	2.37	20.0	0.28	1.35	16.97	-3.58
EP.embar	7.73	1.55	1.0	7.68	1.73	0.99
FT.init_ui	3.43	14.44	0.62	0.97	41.84	-113.64
FT.compute_indexmap	6.74	2.71	0.99	5.65	4.77	0.97
FT.compute_initial_cond	6.96	3.04	0.97	6.45	2.71	0.98
FT.cffts1	3.66	9.75	0.86	1.11	21.81	-16.64
FT.cffts2	3.95	9.39	0.88	1.56	26.2	-4.26
FT.cffts3	3.81	9.57	0.85	1.9	14.35	0.09
SP.exact_rhs2	3.8	10.17	0.85	1.12	24.22	-14.54
SP.exact_rhs3	4.22	4.71	0.97	1.64	22.5	-2.29
SP.exact_rhs4	4.53	3.62	0.98	1.76	24.37	-2.41
SP.compute_rhs1	3.99	9.26	0.86	2.01	17.86	-0.54
SP.compute_rhs2	3.3	10.39	0.78	1.18	12.7	-2.49
SP.compute_rhs3	2.84	16.51	0.46	1.42	6.33	0.69
SP.compute_rhs4	3.73	5.97	0.93	1.69	19.31	-1.33
SP.compute_rhs5	6.15	2.55	0.99	2.43	22.68	-0.94
SP.z_solve	3.64	11.1	0.82	1.18	26.17	-8.84
SP.tzetar	2.71	17.01	0.41	1.34	5.92	0.57
MG.kernel_resid	2.63	38.66	-2.33	1.08	22.93	-27.08
MG.kernel_rprj3	2.78	39.54	-1.34	1.22	10.82	-0.83
MG.kernel_interp_1	2.96	24.5	-0.37	1.07	23.22	-27.7
MG.kernel_psinv	2.56	11.93	0.65	1.13	15.67	-7.85
Average	**3.82**	**9.00**	**0.52**	**1.75**	**14.70**	**-10.33**

Table A.5 Prediction accuracy for varying memory configurations.

`FT.init_ui` on two memory nodes has a percentage error of 42% and an R^2 of -114 even though the model catches the scalability trend well. For more details about the experimental scenarios and configurations, please refer to our research paper [17].

Appendix B

Other Research Contributions of the Author

In this thesis, we have discussed three runtime-level parallelism management techniques for co-located parallel applications: cooperative OpenMP runtime systems, an analytical performance model, a core allocation technique. During my Ph.D. years, I have also participated in a number of different research projects and had opportunities to explore other topics of research beyond the work discussed in this thesis. To maintain the unity of this thesis, the details of other research projects are not discussed. Instead, here, we provide some brief information about other research contributions by this author.

B.1 Compiler and Runtime Support for Integrated CPU-GPU Systems

During 2017–2019, I participated in a joint research project between Seoul National University and ETH Zürich, with two PIs, Prof. Bernhard Egger (SNU) and Prof. Thomas R. Gross (ETH Zürich). The main goal of this research

project is to leverage an interaction between compiler and the runtime system for better resource management on heterogeneous systems. Particularly, through this project, I have worked on a number of researches to optimize performance of integrated CPU/GPU systems. Modern mobile/desktop processors often integrate multiple compute devices such as multicore CPUs and GPUs with shared memory.

In one of our research paper at PACT 2018 [16], we presented an online optimization technique for irregular data-parallel workloads that fail to fully exploit the computational power of the GPU on integrated architectures. The idea is to execute work chunks of similar load on the GPU and assign irregular chunks to the CPU on-the-fly. To this end, a source-to-source compiler dynamically creates profiling code that allows the runtime system to collect information about the computational load of the threads immediately before the kernel is launched. Based on this profile information, the workload is reshaped such that all threads with a high computational load above a dynamically determined threshold are executed on the CPU cores while the GPU only executes only threads below that threshold and with a similar computational load.

On integrated architectures, the limited shared memory bandwidth can lead to a reduced performance when both the CPU and the CPU are executing a workload. We have been developing a software-based technique to throttle the number of CPU and GPU threads to improve performance within the limited memory bandwidth without hardware support. We use static analysis to understand the memory access pattern of the kernel code. Based on the features (e.g. the number of consecutive, stride, and random memory access operations) extracted from the kernel code, we apply a machine learning-based model to predict the optimal number of CPU and GPU threads.

B.2 Modeling NUMA Architectures with Stochastic Tool

I have also worked on a joint research with Prof. Reza Entezari-Maleki who conducts research on performance modeling. We apply several analytical modeling techniques to model real hardware architectures. In our paper research paper [30] in which I am co-authored, for example, we employ a stochastic tool called stochastic reward nets (SRNs) to model and evaluate memory performance on NUMA multi-socket systems. I contributed to this work by providing architectural insights and by validating the models on real hardwares.

B.3 Runtime Environment for a Manycore Architecture

In 2014–2016, I participated in a joint research project between five different research groups at SNU (funded by Samsung). In this project, the project teams did research for a full-stack design of a 96-core manycore architecture. The CSAP lab worked on the implementation of the runtime environment for the manycore architecture. In particular, I contributed to develop runtime and resource management techniques to efficiently execute (multiple) OpenCL applications on the manycore chip. Our implementation was built on top of a System-C and Timed QEMU-based manycore simulator [38].

Bibliography

[1] AMD. AMD Opteron 6300 Series Processors 6380 product information. `https://www.amd.com/en/products/cpu/6380`. [online; accessed July 2020].

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

[3] AMD. Revision Guide for AMD Family 15h Models 00h-0Fh Processors, 2014.

[4] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Russell L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[5] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium*

on Operating Systems Principles, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[6] Benjamin Berg, Jan-Pieter Dorsman, and Mor Harchol-Balter. Towards optimality in parallel job scheduling. *Proc. ACM Meas. Anal. Comput. Syst.*, 1(2):40:1–40:30, December 2017.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[8] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[9] OpenMP Architecture Review Board. OpenMP. `http://openmp.org`, 2018. [online; accessed July 2020].

[10] Jens Breitbart, Simon Pickartz, Stefan Lankes, Josef Weidendorfer, and Antonello Monti. Dynamic co-scheduling driven by main memory bandwidth utilization. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 400–409, Sept 2017.

[11] Jens Breitbart, Josef Weidendorfer, and Carsten Trinitis. Case study on co-scheduling for hpc applications. In *2015 44th International Conference on Parallel Processing Workshops*, pages 277–285, Sept 2015.

[12] Jens Breitbart, Josef Weidendorfer, and Carsten Trinitis. Automatic co-scheduling based on main memory bandwidth usage. In *Proceedings of the 20th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, JSSPP '16, May 2016.

[13] Marc Casas and Greg Bronevetsky. Active measurement of the impact of network switch utilization on application performance. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 165–174, May 2014.

[14] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Oct 2009.

[15] Younghyun Cho, Camilo A. Celis Guzman, and Bernhard 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*, PACT '18, pages 14:1–14:14, New York, NY, USA, 2018. ACM.

[16] Younghyun Cho, Florian Negele, Seohong Park, Bernhard Egger, and Thomas R. Gross. On-the-fly workload partitioning for integrated cpu/gpu architectures. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, pages 21:1–21:13, New York, NY, USA, 2018. ACM.

[17] Younghyun Cho, Surim Oh, and Bernhard Egger. Online scalability characterization of data-parallel programs on many cores. In *2016 International*

Conference on Parallel Architecture and Compilation Techniques (PACT), pages 191–205, Sept 2016.

[18] Younghyun Cho, Surim Oh, and Bernhard Egger. Cooperative parallel runtimes for multicores. In *10th workshop on Programmability and Architectures for Heterogeneous Multicores*, MULTIPROG'17, January 2017.

[19] Younghyun Cho, Surim Oh, and Bernhard Egger. Performance modeling of parallel loops on multi-socket platforms using queueing systems. *IEEE Transactions on Parallel and Distributed Systems*, 31(2):318–331, Feb 2020.

[20] Alexander Collins, Tim Harris, Murray Cole, and Christian Fensch. Lira: Adaptive contention-aware thread placement for parallel runtime systems. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, page 2. ACM, 2015.

[21] Timothy Creech, Aparna Kotha, and Rajeev Barua. Efficient multiprogramming for multicores with scaf. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 334–345, Dec 2013.

[22] Matthew Curtis-Maury, Filip Blagojevic, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Prediction-based power-performance adaptation of multithreaded scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 19(10):1396–1410, Oct 2008.

[23] Leonardo Dagum and Rameshm Enon. OpenMP: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1), 1998.

[24] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: A holistic approach to memory placement on numa systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 381–394, New York, NY, USA, 2013. ACM.

[25] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 77–88, New York, NY, USA, 2013. Association for Computing Machinery.

[26] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 127–144, New York, NY, USA, 2014. Association for Computing Machinery.

[27] Docker Documentation. `http://docs.docker.com`. [online; accessed July 2020].

[28] Murali Krishna Emani and Michael O'Boyle. Celebrating diversity: A mixture of experts approach for runtime mapping in dynamic environments. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 499–508, New York, NY, USA, 2015. ACM.

[29] Murali Krishna Emani, Zheng Wang, and Michael F. P. O'Boyle. Smart, adaptive mapping of parallelism in the presence of external workload. In

Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pages 1–10, Feb 2013.

[30] Reza Entezari-Maleki, Younghyun Cho, and Bernhard Egger. Evaluation of memory performance in numa architectures using stochastic reward nets. *Journal of Parallel and Distributed Computing*, 144:172 – 188, 2020.

[31] Yuping Fan, Zhiling Lan, Paul Rich, William E. Allcock, Michael E. Papka, Brian Austin, and David Paul. Scheduling beyond cpus for hpc. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '19, page 97–108, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Dror G Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling—a status report. In *Job Scheduling Strategies for Parallel Processing*, pages 1–16. Springer, 2005.

[33] Alvaro Frank, Tim Süss, and André Brinkmann. Effects and benefits of node sharing strategies in hpc batch systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 43–53, May 2019.

[34] GNU libgomp documentation. `http://gcc.gnu.org/onlinedocs/libgomp`. [online; accessed July 2020].

[35] Daniel Goodman, Georgios Varisteas, and Tim Harris. Pandia: Comprehensive contention-sensitive thread placement. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 254–269, New York, NY, USA, 2017. ACM.

[36] Dominik Grewe, Zheng Wang, and Michael F. P. O'Boyle. A workload-aware mapping approach for data-parallel programs. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 117–126, New York, NY, USA, 2011. ACM.

[37] Camilo A. Celis Guzman, Younghyun Cho, and Bernhard Egger. SnuMAP: an Open-source Trace Profiler for Manycore Systems. `https://csap.snu.ac.kr/software/snumap/`, 2017. [online; accessed July 2020].

[38] Shin haeng Kang, Donghoon Yoo, and Soonhoi Ha. Tqsim: A fast cycle-approximate processor simulator based on qemu. *Journal of Systems Architecture*, 66-67:33 – 47, 2016.

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

[40] Tim Harris, Martin Maas, and Virendra J. Marathe. Callisto: Co-scheduling parallel runtime systems. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. ACM.

[41] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, Berkeley, CA, USA, 2011. USENIX Association.

[42] Intel. Intel Xeon Processor E7-8870 v3. `https://ark.intel.com/content/www/us/en/ark/products/84682/`

intel-xeon-processor-e7-8870-v3-45m-cache-2-10-ghz.html. [online; accessed July 2020].

[43] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, 2015.

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

[45] Shintaro Iwasaki, Abdelhalim Amer, Kenjiro Taura, Sangmin Seo, and Pavan Balaji. Bolt: Optimizing openmp parallel regions with user-level threads. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 29–42, Sep. 2019.

[46] Henk Jonkers. Queueing models of parallel applications: the glamis methodology. In *Computer Performance Evaluation Modelling Techniques and Tools*, pages 123–138. Springer, 1994.

[47] Henk Jonkers. Queueing models of shared-memory parallel applications. In *Computer and Telecommunication Systems Performance Engineering*. Pentech Press Ltd, 1994.

[48] Karthik Kambatla, Vamsee Yarlagadda, Íñigo Goiri, and Ananth Grama. Ubis: Utilization-aware cluster scheduling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 358–367, May 2018.

[49] Khronos Group. The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/. [online; accessed August 2020].

[50] Abdullah Konak, David W. Coit, and Alice E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering*

& System Safety, 91(9):992 – 1007, 2006. Special Issue - Genetic Algorithms and Reliability.

[51] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 81–87, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[52] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. Adapt: A framework for coscheduling multithreaded programs. *ACM Trans. Archit. Code Optim.*, 9(4):45:1–45:24, January 2013.

[53] Janghaeng Lee, Haicheng Wu, Madhumitha Ravichandran, and Nathan Clark. Thread tailor: Dynamically weaving threads together for efficient, adaptive parallel applications. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 270–279, New York, NY, USA, 2010. ACM.

[54] Hui Li, Sudarsan Tandri, Michael Stumm, Sevcik, and Kenneth C. Locality and loop scheduling on numa multiprocessors. In *1993 International Conference on Parallel Processing - ICPP'93*, volume 2, pages 140–147, Aug 1993.

[55] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, New York, NY, USA, 2015. Association for Computing Machinery.

[56] David A Lifka. The anl/ibm sp scheduling system. In *Job Scheduling Strategies for Parallel Processing*, pages 295–303. Springer, 1995.

[57] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz. Tessellation: Space-time partitioning in a manycore client os. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 10–10, Berkeley, CA, USA, 2009. USENIX Association.

[58] Xu Liu and John Mellor-Crummey. A tool to analyze the performance of multithreaded programs on numa architectures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 259–272, New York, NY, USA, 2014. ACM.

[59] Qiuyun Llull, Songchun Fan, Seyed Majid Zahedi, and Benjamin C. Lee. Cooper: Task colocation with cooperative games. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 421–432, Feb 2017.

[60] Arthur Francisco Lorenzon, Charles Cardoso de Oliveira, Jeckson Dellagostin Souza, and Antonio Carlos Schneider Beck. Aurora: Seamless optimization of openmp applications. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2018.

[61] Kun Luo, Jayanth Gummaraju, and Manoj Franklin. Balancing thoughput and fairness in smt processors. In *IsPASS*, volume 1, pages 164–171, 2001.

[62] Zoltan Majo and Thomas R. Gross. Memory management in numa multicore systems: Trapped between cache contention and interconnect overhead. *SIGPLAN Not.*, 46(11):11–20, June 2011.

[63] Zoltan Majo and Thomas R Gross. Matching memory access patterns and data placement for NUMA systems. In *Proceedings of the Tenth Interna-*

tional Symposium on Code Generation and Optimization, pages 230–241. ACM, 2012.

[64] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[65] Ryan W. Moore and Bruce R. Childers. Using utility prediction models to dynamically choose program thread counts. In *2012 IEEE International Symposium on Performance Analysis of Systems Software*, pages 135–144, April 2012.

[66] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *Parallel and Distributed Systems, IEEE Transactions on*, 12(6):529–543, 2001.

[67] Onur Mutlu and Thomas 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*, ISCA '08, pages 63–74, Washington, DC, USA, 2008. IEEE Computer Society.

[68] Bhyrav Mutnury, Frank Paglia, James Mobley, Girish K. Singh, and Ron Bellomio. Quickpath interconnect (qpi) design and analysis in high speed servers. In *19th Topical Meeting on Electrical Performance of Electronic Packaging and Systems*, pages 265–268, Oct 2010.

[69] Angeles Navarro, Rafael Asenjo, Siham Tabik, and Calin Cascaval. Analytical modeling of pipeline parallelism. In *2009 18th International Conference*

on Parallel Architectures and Compilation Techniques, pages 281–290, Sep. 2009.

[70] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 2:1–2:17, New York, NY, USA, 2018. ACM.

[71] Arun Raman, Hanjun Kim, Taewook Oh, Jae W. Lee, and David I. August. Parallelism orchestration using dope: The degree of parallelism executive. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 26–37, New York, NY, USA, 2011. Association for Computing Machinery.

[72] Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. Parcae: A system for flexible parallel execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 133–144, New York, NY, USA, 2012. ACM.

[73] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[74] Gabriele Sartori. Hypertransport Technology. Platform Conference, 2001.

[75] Hiroshi Sasaki, Satoshi Imamura, and Koji Inoue. Coordinated power-performance optimization in manycores. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 51–61, Sept 2013.

[76] Hiroshi Sasaki, Teruo Tanimoto, Koji Inoue, and Hiroshi Nakamura. Scalability-based manycore partitioning. In *Proceedings of the 21st Inter-*

national Conference on Parallel Architectures and Compilation Techniques, PACT '12, pages 107–116, New York, NY, USA, 2012. ACM.

[77] Scipy stats.ks_2samp document. `https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.stats.ks_2samp.html`. [online; accessed July 2020].

[78] Vicent Selfa, Julio Sahuquillo, Salvador Petit, and María E. Gómez. A hardware approach to fairly balance the inter-thread interference in shared caches. *IEEE Transactions on Parallel and Distributed Systems*, 28(11):3021–3032, 2017.

[79] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, Shintaro Iwasaki, Prateek Jindal, Laxmikant V. Kalé, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, Kenjiro Taura, and Pete Beckman. Argobots: A lightweight low-level threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):512–526, March 2018.

[80] Sangmin Seo, Gangwon Jo, and Jaejin Lee. Performance characterization of the nas parallel benchmarks in opencl. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 137–148, Nov 2011.

[81] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. Adaptive, efficient, parallel execution of parallel programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 169–180, New York, NY, USA, 2014. ACM.

[82] Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen. Data sharing or resource contention: Toward performance transparency on multicore systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 529–540, Santa Clara, CA, 2015. USENIX Association.

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

[84] Bogdan Marius Tudor and Yong Meng Teo. A practical approach for performance analysis of shared-memory programs. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 652–663. IEEE, 2011.

[85] Bogdan Marius Tudor, Yong Meng Teo, and Simon See. Understanding off-chip memory contention of parallel programs in multicore systems. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 602–611. IEEE, 2011.

[86] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 35:1–35:16, New York, NY, USA, 2016. ACM.

[87] András Vajda. *Programming many-core chips*. Springer Science & Business Media, 2011.

[88] Wei Wang, Jack W. Davidson, and Mary Lou 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), pages 419–431, March 2016.

[89] Wei Wang, Tanima Dey, Jack W. Davidson, and Mary Lou 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)*, pages 380–391, Feb 2014.

[90] David Wentzlaff, Charles Gruenwald III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An operating system for multicore and clouds: mechanisms and implementation. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 3–14. ACM, 2010.

[91] Seyed Majid Zahedi, Qiuyun Llull, and Benjamin C. Lee. Amdahl's law in the datacenter era: A market for fair processor allocation. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–14, Feb 2018.

[92] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 129–142, New York, NY, USA, 2010. ACM.

[93] Pengfei Zou, Xizhou Feng, and Rong Ge. Contention aware workload and resource co-scheduling on power-bounded systems. In *2019 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–8, Aug 2019.

초록

멀티코어 시스템에서 여러 개의 병렬 처리 어플리케이션들을 함께 실행시키는 것은 주어진 하드웨어 자원을 효율적으로 사용하기 위해서 점점 더 중요해지고 있다. 하지만, 현재 런타임 시스템에서 여러 개의 병렬 처리 어플리케이션들을 동시에 효율적으로 실행시키는 것은 여전히 어려운 문제이다. OpenMP와 같이 통상 사용되는 병렬화 런타임 시스템들은 모든 하드웨어 코어 자원을 사용하기 위해서 일반적으로 코어 개수 만큼 스레드를 생성하여 어플리케이션을 실행시킨다. 이 때, 어플리케이션은 모든 코어 자원을 활용할 때 오히려 최적의 성능을 얻지 못할 수도 있으며, 운영체제 커널의 부하는 실행되는 어플리케이션의 개수가 늘어날수록 관리해야 하는 스레드의 개수가 늘어나기 때문에 계속해서 커지게 된다.

본 학위 논문에서, 우리는 함께 실행되는 병렬 처리 어플리케이션들의 런타임 성능을 높이는 것에 집중한다. 이를 위해, 본 연구의 핵심 목표는 함께 실행되는 어플리케이션들에게 공간 분할식 스케줄링 방법을 적용하는 것이다. 각 어플리케이션에게 독립적인 코어 자원을 할당해주는 공간 분할식 스케줄링은 점점 더 늘어나는 코어 자원의 개수를 효율적으로 관리하기 위한 방법으로 많은 관심을 받고 있다. 하지만, 공간 분할 스케줄링 방법을 통해 어플리케이션을 실행시키는 것은 두 가지 연구 과제를 가지고 있다. 먼저, 각 어플리케이션은 가변적인 코어 자원 상에서 효율적으로 실행되기 위한 런타임 기술을 필요로 하고, 스케줄러는 어플리케이션들의 성능 특성을 고려해서 런타임 성능을 높일 수 있도록 적당한 수의 코어 자원을 제공해야한다.

이 학위 논문에서, 우리는 함께 실행되는 병렬 처리 어플리케이션들을 공간 분할 스케줄링을 통해서 효율적으로 실행시키기 위한 세가지 런타임 시스템 기술을 소개한다. 먼저 우리는 협동적인 런타임 시스템이라는 기술을 소개하는데, 이는 OpenMP 병렬 처리 어플리케이션들에게 유연하고 효율적인 실행 환경을 제공한

다. 이 기술은 공유 메모리 병렬 실행에 내재되어 있는 특성을 활용하여 병렬처리 프로그램들이 변화하는 코어 자원에 맞추어 병렬성의 정도를 동적으로 조절할 수 있도록 해준다. 이러한 유연한 실행 모델은 병렬 어플리케이션들이 사용 가능한 코어 자원이 동적으로 변화하는 환경에서 어플리케이션의 스레드 수준 병렬성을 다루지 못하는 기존 런타임 시스템들에 비해서 더 효율적으로 실행될 수 있도록 해준다.

두번째로, 본 논문은 사용되는 코어 자원에 따른 병렬처리 프로그램의 성능 및 자원 활용도를 예측할 수 있도록 해주는 분석적 성능 모델을 소개한다. 병렬 처리 코드의 성능 확장성이 일반적으로 메모리 성능에 좌우된다는 관찰에 기초하여, 제안된 해석 모델은 큐잉 이론을 활용하여 메모리 시스템의 성능 정보들을 계산한다. 이 큐잉 시스템에 기반한 방법은 유용한 성능 정보들을 수식을 통해 효율적으로 계산할 수 있도록 하며 상용 시스템에서 제공하는 하드웨어 성능 카운터만을 요구하기 때문에 활용 가능성 또한 높다.

마지막으로, 본 논문은 동시에 실행되는 병렬 처리 어플리케이션들 사이에서 코어 자원을 할당해주는 프레임워크를 소개한다. 제안된 프레임워크는 동시에 동작하는 병렬 처리 어플리케이션의 병렬성 및 코어 자원을 관리하여 멀티 소켓 멀티코어 시스템에서 CPU 자원 및 메모리 대역폭 자원 활용도를 동시에 최적화한다. 해석적인 모델링과 제안된 코어 할당 프레임워크의 성능 평가를 통해서, 우리가 제안하는 정책이 일반적인 경우에 CPU 자원의 활용도만을 최적화하는 방법에 비해서 함께 동작하는 어플리케이션들의 실행시간을 감소시킬 수 있음을 보여준다.

주요어: 런타임 시스템, 성능 모델링, 자원 관리
학번: 2013-20887

Acknowledgements

I consider myself fortunate that I was able to study and do research in an excellent environment at Seoul National University (SNU). The Ph.D. study, however, was sometimes very difficult and overwhelming with many challenges and a lot of worries about research. It would not have been possible to complete my Ph.D. study without the help and support of the many people around me. I would like to extend my sincere thanks to all of them.

First and foremost, I would like to express my deep appreciation to my advisor Professor Bernhard Egger who gave me the opportunity to study at SNU and provided an excellent working environment in the Computer Systems and Platforms (CSAP) lab. I am sincerely grateful for his continuous support, advice, patience, and encouragement during the years of my Ph.D. study. I also thank him for providing an international and collaborative working environment through various research projects.

I also thank the members of my Ph.D. committee, Professor Jaejin Lee, Professor Heon-Young Yeom, Professor Lawrence Rauchwerger, and Professor David August for their kind agreement to examine this thesis and for their valuable feedback and suggestions on this work.

I would also like to thank my past research collaborators. I am grateful to

Professor Thomas Gross for his guidance on our joint research project and providing a comfortable working environment during my stay at ETH Zürich. I also thank Dr. Florian Negele for working closely with me on this SNU-ETH project. Thanks to him, I have lots of good memories of Zürich and Liechtenstein.

Special thanks go to my good friend and research collaborator Professor Reza Entezari-Maleki. Through our joint work, I was able to learn lots of theoretical background of analytical modeling from him.

I was fortunate to have many good fellow students in the CSAP lab. I met more than 30 people including master and doctoral students, visiting students, and research assistants. I thank all of them for spending time with me. My special thanks go to Camilo Celis Guzman for working hard with me on our research projects and for being a good friend to me. Also, I thank Changyeon Jo who has spent the longest time with me in the lab.

I am grateful for the immeasurable support I have received from my parents, parents-in-law, and all the family members. Their support helped me consistently focus on the research. My deepest thanks go to my parents, Jinhong Cho and Inhee Hwang, for their unconditional love and their support in every of my decision in my life. Also, I thank my sister Kyungseon Cho for her encouragement during my Ph.D. study.

My biggest thanks go to my wife Surim Oh. She has been both great colleague and companion, and the motivation for me to be a better person. I thank her for working with me while she was in the CSAP lab. More importantly, thanks to her constant support and encouragement, I was able to overcome the difficult steps in the last Ph.D. years. I am looking forward to starting our next journey very much.

Younghyun Cho, Seoul, July 2020