Modeling Memory System Performance of NUMA Multicore-Multiprocessors

Zoltán Majó

DISS. ETH No. 22006

Modeling Memory System Performance of NUMA Multicore-Multiprocessors

A thesis submitted to attain the degree of
DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by
ZOLTÁN MAJÓ
Ing. dipl., Technical University of Cluj-Napoca
born on June 18, 1983
citizen of Hungary and Romania

accepted on the recommendation of
Prof. Dr. Thomas R. Gross, examiner
Prof. Dr. Frank Müller, co-examiner
Prof. Dr. Michael Stumm, co-examiner

2014

Hatházi Anna-Máriának.

Abstract

The performance of many applications depends closely on the way they interact with the computer's memory system: Many applications obtain good performance only if they utilize the memory system efficiently.

Unfortunately, obtaining good memory system performance is often difficult, as developing memory system-aware (system) software requires a thorough and detailed understanding of both the characteristics of the memory system and of the interaction of applications with the memory system. Moreover, the design of memory systems evolves as newer processor generations appear on the market, thus the problem of software–hardware interaction must be revisited to understand the interaction of (already existing) software with newer memory system designs as well.

This thesis investigates the memory system performance of a recent class of machines, multicore-multiprocessors with a non-uniform memory architecture (NUMA). A NUMA multicore-multiprocessor system consists of several processors where each processor integrates multiple cores. Typically, cores of a multicore processor share resources (e.g., last-level caches) and contention for these shared resources can result in significant performance degradations.

NUMA multicore-multiprocessors are shared-memory computers, but the memory space of a NUMA multicore-multiprocessor system is partitioned between processors. Accessing the memory of a local processor takes less time than accessing the memory of other (remote) processors, therefore data locality (a low number of remote memory accesses) is critical for good performance on NUMA machines.

This thesis presents a performance-oriented model for NUMA multicore-multiprocessors. The model considers two application classes, multiprogrammed workloads (workloads that consist of multiple, independent processes) and multithreaded programs (programs that consist of a number of threads that operate in a shared address space). The thesis presents an experimental analysis of memory system bottlenecks experienced by each application class. Moreover, the thesis presents techniques to reduce the performance-degrading effects of these bottlenecks.

We determine (based on experimental analysis) that the performance of multiprogrammed workloads depends on both multicore-specific and NUMA-specific aspects of a NUMA multicore-multiprocessor's memory system. Therefore, a process scheduler must find a balance between reducing cache contention and improving data locality; the N-MASS scheduler presented by the thesis attempts to strike a balance between these, sometimes contradicting, goals. N-MASS improves performance up to 32% over the default setup in current Linux implementations on a recent 2-processor 8-core machine.

Based also on experimental analysis we find that data locality is of prime importance for the performance of multithreaded programs. The thesis presents extensions to two popular parallel programming frameworks, OpenMP and Intel's Threading Building Blocks. The extensions allow a programmer to express affinity of data and computations, which, if done appropriately, helps to improve data locality and thus performance on NUMA multicore-multiprocessors (by up to 220% on a recent 4-processor 32-core machine). The thesis also shows that adding NUMA support not only to the programmer interface, but also to the underlying runtime system, allows programs to be portable across different architectures as well as to be composable with other programs (that use the same runtime system).

Zusammenfassung

Die Rechenleistung vieler Software-Applikationen hängt von der Interaktion der Applikation mit dem Speichersystem des Computers ab: Viele Applikationen erreichen eine gute Rechenleistung nur wenn sie das Speichersystem des Computers effizient nutzen.

Es ist aber leider oft schwierig Software zu entwickeln, die das Speichersystem effizient benützt, da Programmierer sowohl die Merkmale des Speichersystems wie auch das Zusammenspiel der Software mit dem Speichersystem verstehen müssen, um effiziente Software entwickeln zu können. Darüber hinaus muss häufig das Zusammenspiel (schon existierender) Software erneut analysiert and verstanden werden, wenn neue Computerarchitekturen (eventuell mit einem neuen Typ von Speichersystem) auf den Markt gebracht werden.

Diese Dissertation analysiert das Speichersystem einer neuen Klasse von Rechnern, Multikern-Multiprozessoren mit einer nicht-uniformen Speicherarchitektur (engl.: non-uniform memory architecture (NUMA)). Ein NUMA Multikern-Multiprozessor besteht aus mehreren Prozessoren; jeder Prozessor des Systems besteht aus mehreren Kernen. Die Kerne eines Multikernprozessors teilen in der Regel Ressourcen (z.B. den Cachespeicher des Prozessors) und der gleichzeitige Gebrauch von geteilten Ressourcen kann zu einer Erhöhung der Laufzeit von Applikationen führen (im Vergleich mit dem Fall wenn keine Ressourcen geteilt sind).

Jeder Prozessor eines NUMA Multikern-Multiprozessors hat Zugriff auf alle Speicherstellen des Systems, der Adressraum des Systems ist aber zwischen den Prozessoren partizioniert. Zugriffe auf Speicherstellen des lokalen Prozessors dauern weniger lang als Zugriffe auf Speicherstellen eines entfernten Prozessors, daher ist Datenlokalität (eine niedrige Zahl von Zugriffen auf die Speicherstellen eines entfernten Prozessors) entscheidend für die Rechenleistung vieler Applikationen.

Diese Dissertation beschreibt ein leistungsorientiertes Modell für NUMA Multikern-Multiprozessoren. Das Modell betrachtet zwei Klassen von Software-Applikationen, multiprogrammierte Applikationen, welche aus mehreren unabhängigen Prozessen bestehen, und multithreaded Applikationen, welche aus mehreren Threads bestehen, die miteinander Daten teilen. Die Dissertation identifiziert Engpässe des Speichersystems, welche die Laufzeit von Applikationen beider Klassen negativ beinflussen. Die Dissertation beschreibt auch Methoden um die negativen Auswirkungen der Engpässe zu reduzieren.

Wir stellen fest (mittels experimenteller Analyse), dass sowohl Datenlokalität als auch die gleichzeitige Benutzung geteilter Ressourcen für die Laufzeit multiprogrammierter Applikationen entscheidend ist, und dass der Scheduler des Betriebsystem eine Balance zwischen den beiden, oft miteinander im Konflikt stehenden Faktoren, finden muss. Die Dissertation beschreibt einen neuen Scheduler-Algorithmus, N-MASS. Die Verwendung von N-MASS ergibt eine Verbesserung der Laufzeit multiprogrammierter Applikationen von bis zu 32% (verglichen mit einer Standard Linux-Implementation auf einem modernen NUMA Multikern-Multiprozessor mit zwei Prozessoren und 8 Kernen).

Ausserdem stellen wir fest (auch mittels experimenteller Analyse), dass Datenlokalität für eine effiziente Ausführung von multithreaded Applikationen unerlässlich ist. Die Dissertation präsentiert Erweiterungen für zwei bekannte Frameworks für parallelle Progammierung, OpenMP und Intel Threading Building Blocks. Mit diesen Erweiterungen können Programmierer die Affinität von Daten und Berechnungen ausdrücken, was, wenn dies in geeigneter Weise getan wird, dazu führt, dass Datenlokalität und somit auch Rechenleistung sich deutlich (bis zu 220% auf einem Rechner mit vier Prozessoren mit 32 Kernen) verbessert. Die Dissertation zeigt auch, dass, wenn nicht nur die Programmierschnittstelle sondern auch das Laufzeitsystem für NUMA Multikern-Multiprozessoren angepasst wird, optimierte Programme portabel sind und ihre Datenlokalität auch dann bewahren, wenn sie mit anderen Programmen zusammengesetzt werden.

Acknowledgments

Many people helped me complete my doctoral studies.

I am grateful to my advisor, Thomas R. Gross, for giving me the opportunity to study at ETH. I thank Thomas for the advice and guidance he provided me during the years of my doctoral studies and also for the excellent working environment that he has maintained in the research group.

I thank my co-examiners, Frank Müller and Michael Stumm, for being on the examination committee and for their valuable feedback on the dissertation.

Many thanks to my colleagues in the Laboratory of Software Technology for the friendly and collaborative environment. My thanks go to current colleagues, Luca Della Toffola, Faheem Ullah, Martin Bättig, Animesh Trivedi, Antonio Barresi, Remi Meier, Michael Fäs, Aristeidis Mastoras, and Ivana Unković, as well as former members of the research group, Florian Schneider, Yang Su, Mihai Cuibus, Susanne Cech, Oliver Trachsel, Nicholas Matsakis, Stephanie Balzer, Christoph Angerer, Mathias Payer, Stefan Freudenberger, Michael Pradel, and Albert Noll.

I was fortunate enough to have many of my friends by my side during the PhD years. The list is, fortunately, so long that it would, unfortunately, not fit it onto one (and possibly not even onto several) page(s). But I thank you all.

My biggest thanks go to my wife (Ágota), to my parents (Julianna and Zoltán), to my sister (Zsuzsa), and to all my family for their unconditional love and support.

Contents

1

Introduction

The performance of a large number of applications depends critically on the memory system; that is, numerous applications achieve good performance (low end-to-end execution time) only if they efficiently utilize the memory system of the machine.

Obtaining good memory system performance poses challenges for both hardware and software engineers. On the hardware side, it is challenging to design a memory system architecture that provides high memory bandwidth and, at the same time, low memory access latency for all processor(s)/core(s) of a system. The problem of providing adequate memory access is further exacerbated by the recent trend of core counts increasing with every processor generation.

Producing (system) software that uses the capabilities of the memory system efficiently is difficult as well. To achieve good performance, programmers must understand the interaction between hardware and software which requires detailed understanding of the properties of both the memory system and the applications using it. More specifically, given a memory system architecture, programmers need a *performance-oriented model* that comprehensively describes the memory system performance of that architecture. Ideally, such a model

- identifies and describes the *performance bottlenecks* of the memory system architecture;
- defines and describes a set of *application classes*, where applications of the same class interact with the memory system architecture in a well-defined and well-understood way (that is different from the way applications in other classes interact with the same memory system architecture);
- identifies and describes *performance optimization techniques*, that is, ways to improve the memory system performance of a given application class.

Although a processor manufacturer's manuals contain plentiful information about the properties of a given processor [45], they usually specify only a few aspects related to the performance of the processor. Thus, manuals are usually augmented either by the processor manufacturer itself (e.g., in the form of optimization guides [44, 57]) or by the research community (usually in the form of experimental studies [4, 18, 22, 39, 73, 108]).

Despite the efforts, however, there are still numerous open questions about memory system performance. Moreover, as processor memory system design evolves, existing performance models must be revisited and updated; moreover, if needed, new models must be developed.

For example, recent shared-memory multiprocessor systems (systems built with multiple processors operating within the same physical address space) are based on multicore processors

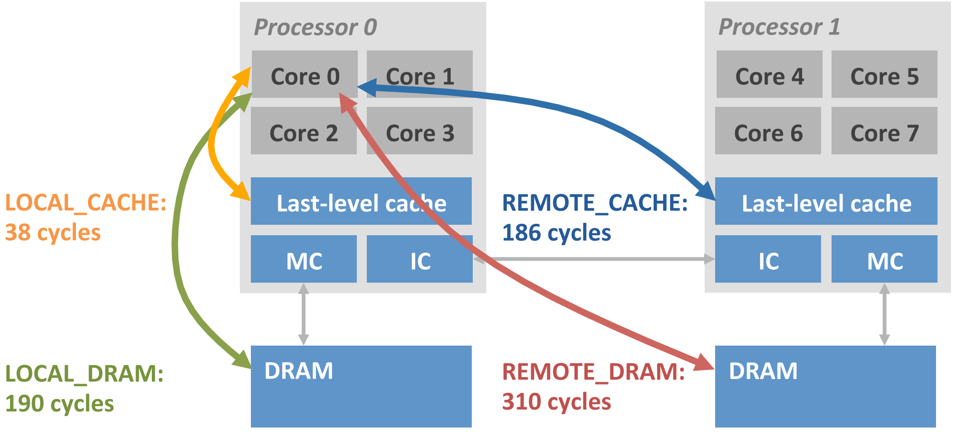


Figure 1.1: 2-processor 8-core NUMA multicore-multiprocessor.

(processors with multiple cores per processor). Moreover, these systems are typically equipped with a non-uniform memory architecture (NUMA). These *NUMA multicore-multiprocessors* are an attractive and popular platform that are used as both stand-alone systems (e.g., server computing) or as building blocks in supercomputers, thus it is important to understand the memory system performance of this class of machines.

The memory system of a typical NUMA multicore-multiprocessor is complex as it has characteristics specific to *both* its building blocks, that is, it has characteristics specific to both multicore and NUMA systems. Although both types of memory system architectures are reasonably well understood *in isolation*, the *combination* of the two memory system architectures requires further investigation.

To better illustrate the features of the memory system of a NUMA multicore-multiprocessor, consider the example 2-processor 8-core system shown in Figure 1.1. In the example system each processor accesses a part of the main memory that is connected to the processor through an on-chip memory controller (MC). Moreover, to allow each processor to access the memory of the other processor as well, processors are connected to each other with a cross-chip interconnect (each processor has an additional memory interface (IC) that allows the processor to connect to the cross-chip interconnect).

Partitioning the memory space between processors has the benefit that the memory bandwidth available to applications scales with the number of processors. Nevertheless, a partitioned space can have disadvantages as well, as applications can experience large performance penalties due to *remote memory accesses*. Remote memory accesses are transferred on the cross-chip interconnect, thus they encounter an overhead relative to *local memory accesses* (accesses handled by the on-chip memory controller). Figure 1.1 shows the disparity between local- and remote memory access latencies, as measured on a NUMA multicore-multiprocessor that is composed of processors based on the recent Intel Nehalem microarchitecture [73] [1]. NUMA multicore-multiprocessor systems based on other microarchitectures, for example those developed by AMD, show similar disparities between memory access latencies [39].

The effect of the remote memory access penalty on application performance is well-studied

---

[1] The latencies indicated in the Figure 1.1 are measured for read transfers (i.e., read accesses last-level caches and to DRAM). For cache accesses we report latencies to cache lines in the E cache coherency state. See [73] for the complete information, e.g., latencies for accessing cache lines in other cache coherency states as well.

and well-understood and there exists a large body of work on data locality optimizations (optimizations that attempt to reduce the number of remote memory accesses) [56, 60, 67, 68, 72, 79]. However, the memory system of current multicore-based NUMA systems has aspects specific to multicore processors as well. In a multicore system, processor cores share (some) memory system resources; shared resources must be managed properly to obtain good memory system performance. For example, recent multicore processors typically have a last-level cache that is shared between a (subset) of the cores on a processor (e.g., in the example system in Figure 1.1 four cores share a last-level cache). As shown recently, cache sharing can both improve [112] and deteriorate [33, 52] application performance in multicore systems.

In summary, due to the bivalence of NUMA multicore-multiprocessors several questions arise:

- Do NUMA multicore-multiprocessors systems exhibit multicore-specific or NUMA-specific performance bottlenecks, or both?
- What kind of performance optimization techniques are worthwhile to be pursued to address these bottlenecks?
- Which are the application classes that benefit from these optimization techniques?

This thesis answers these questions by providing a *performance-oriented model of application memory system performance on NUMA multicore-multiprocessor systems*. The contributions of this thesis are threefold. First, using experimental analysis, the thesis identifies and describes a set of performance bottlenecks specific to NUMA multicore-multiprocessor memory systems. Second, the thesis investigates the implications of the previously identified bottlenecks for two different application classes. Third, the thesis describes performance optimization techniques applicable for each application class and presents a prototype implementation (and an experimental evaluation) of each technique.

1.1 Thesis statement

The model presented by the thesis considers two application classes: (1) multiprogrammed workloads, that is, workloads that consists of multiple, independent processes, and (2) multithreaded programs, that is, programs that consist of a set of threads that operate on (possibly) shared data.

Thesis statements:

1. *The memory system performance of multiprogrammed workloads on NUMA multicore-multiprocessors strongly depends on two factors: contention for shared resources and interconnect overhead. Information about a workload's memory behavior enables process scheduling to improve the performance of multiprogrammed workloads by accounting for both factors.*
2. *The memory system performance of multithreaded programs critically depends on the locality of DRAM/cache accesses. If provided with task/data affinity information, a NUMA-aware runtime system can achieve good data locality without compromising load balance, furthermore, a NUMA-aware runtime system enables portable and composable data locality optimizations and thus helps amortize the optimization effort.*

1.2 Organization of this dissertation

This thesis is structured as follows.

The first part of the thesis (Chapters 2 and 3) focuses on multiprogrammed workloads. Chapter 2 presents an experimental analysis of the memory system performance of multiprogrammed workloads on NUMA multicore-multiprocessors. The analysis shows that a typical NUMA multicore-multiprocessor memory system has several bottlenecks; good performance can be achieved only if all these bottlenecks are taken into consideration. Then, in Chapter 3 we describe the N-MASS process scheduling algorithm that attempts to find a balance between two memory system bottlenecks, cache contention and interconnect overhead.

The second part of the thesis (Chapters 4, 5, and 6) focuses on multithreaded programs. Chapter 4 analyzes the performance impact of different schemes for mapping data and computations. Then, Chapters 5 and 6 present language-level extensions to two parallel programming frameworks, OpenMP and TBB, to support locality-aware programming.

Chapter 7 presents related work and Chapter 8 concludes the thesis.

2

Resource sharing and interconnect overhead

This chapter presents an experimental analysis of the memory system performance of NUMA multicore-multiprocessors. We focus on a single application class, multiprogrammed workloads (workloads that consist of a set of independent processes that do not share data).

The main question addressed by this chapter is: How much does resource sharing impact application performance and what is (comparatively) the impact of interconnect overhead? To provide an answer, this chapter considers two aspects of NUMA-multicore memory system performance. First, in Section 2.1 we look into the relative cost of interconnect overhead and memory controller contention by experimentally analyzing the sharing of local/remote memory bandwidth in a recent NUMA multicore-multiprocessor system. Then, in Section 2.2 we extend the scope of the analysis and consider contention for a different shared resource, the last-level caches of a system, as well.

2.1 Sharing of local/remote memory bandwidth

In NUMA multicore-multiprocessors each processor accesses part of the physical memory directly (via an on-chip memory controller) and has access to the other parts of the physical memory via the memory controller of other processors. Other processors are reached via the cross-processor interconnect, and major processor manufacturers have developed their proprietary cross-chip interconnect technology that connects the processors (e.g., AMD's HyperTransport [1] or Intel's QuickPath Interconnect (QPI) [64]).

Remote memory accesses (accesses via the interconnect) are subject to various overheads. The bandwidth provided by the cross-chip interconnect is lower than the bandwidth provided by the local (on-chip) memory controller. Moreover, the latency of remote memory accesses is higher than the latency of local (on-chip) memory accesses: Remote memory accesses are first sent to the interconnect (arbitration may be needed if multiple cores access remote memory at the same time), then a request is transmitted to another processor, and finally additional steps may be needed on this remote processor before the memory access can be done.

As a result, the performance penalty of remote memory accesses is significant (we call this penalty the *NUMA penalty*): In current systems the NUMA penalty can be as high as 2 (equivalent to a 2X slowdown) for some applications. The NUMA penalty in recent systems is somewhat lower than in earlier implementations of NUMA (e.g., 3 to 5 in the Stanford FLASH [107]), but a slowdown of 2X is still high, thus avoiding the NUMA penalty must be considered in recent systems as well.

Traditionally, performance optimizations for NUMA systems aim for increasing *data locality* (i.e., reducing or even eliminating remote memory accesses in the system) by changing the allocation of memory and/or the mapping of computations in the system [13, 17, 27, 55, 58, 62, 67, 68, 72, 77, 79, 102–104, 107, 109]. As the performance of many applications is ultimately limited by the performance of the memory system, it is important to understand the memory system of NUMA multicore-multiprocessors as simple and realistic models are crucial to find mappings (of data and computations) that result in good performance on these systems.

Previous research has focused on evaluating the bandwidth and latency of the on-chip memory controller and of the cross-chip interconnect of modern NUMA machines *in separation* (i.e., when there are either local or remote memory accesses in the system, but not both of them at the same time) [39, 44, 66, 73]. However, it rarely happens in real systems that a computation's memory traffic exclusively flows through either the local memory interface or the cross-chip interconnect that connects to the memory controller of a remote processor. So it is also important to understand how these two types of memory accesses (local and remote) interact.

A recent study [6] evaluates the problems and opportunities posed by having multiple types of memory controllers in a system. The authors show that a page placement algorithm that accounts not only for data locality, but also for contention for memory controllers and for cross-chip interconnects can obtain good application performance in a system with multiple memory controllers. However, the study is more concerned about future architectures and less with existing ones. In the following, we analyze the bandwidth sharing properties of a commercial microprocessor and discuss the implications of these properties for optimizing programs in multicore systems. We show that in some cases–when the machine is highly loaded–the cross-chip interconnect outperforms the on-chip memory controller. Mapping computations so that all memory traffic flows through the local memory interface is bound to be suboptimal in many situations due to resource contention.

2.1.1 Experimental setup

In the following, we describe the memory system architecture of the evaluated system, the benchmark programs, and the experimental methodology we use.

Hardware

We investigate the memory system of a multicore-multiprocessor machine based on the Intel Nehalem microarchitecture. The machine is equipped with two Intel Xeon E5520 quad-core CPUs running at 2.26 GHz and a total of 12 GB RAM. The memory system architecture of the Nehalem-based system is sketched in Chapter 1 (see Figure 1.1). In this section we describe the memory system of the machine in more detail.

Figure 2.1 shows the Nehalem-based system we investigate. In this system each processor has a direct connection to half of the memory space via a three-channel integrated memory controller. The on-chip integrated memory controller (IMC) provides a maximum theoretical throughput of 25.6 GB/s. Additionally, each processor has two QuickPath Interconnect (QPI) interfaces [64], one connecting to the remote processor and one to the I/O hub. The interconnect has a maximum theoretical throughput of 11.72 GB/s in one direction and 23.44 GB/s in both

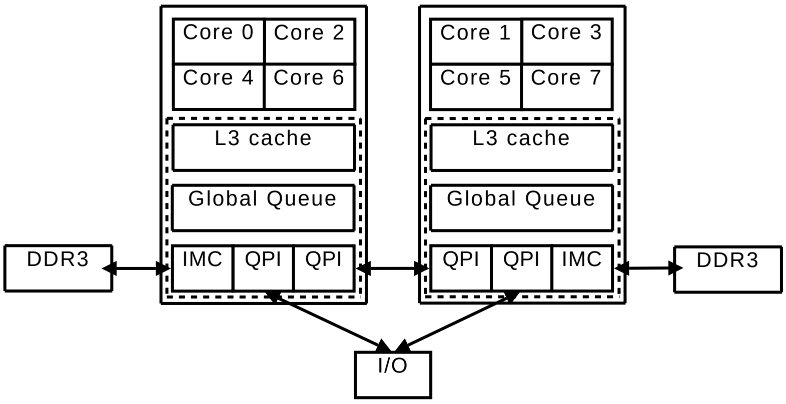


Figure 2.1: Intel Nehalem in a 2-processor configuration.

directions.

Although the throughput of the QPI is almost as high as the throughput of the IMC, there are two IMCs in the system, while there is only one QPI link connecting the two processors. Thus, if an application has good data locality (it predominantly accesses local memory), the application can exploit the throughput of the two IMCs (2X25.6 GB/s). Otherwise, the application's performance can be limited by the throughput of the single QPI cross-chip interconnect in the system (23.44 GB/s). (In addition to bandwidth-related limitations, the application's performance is negatively influenced by the increased latency of remote memory accesses relative to local accesses [73].)

Furthermore, each core of a Nehalem processor has its own level 1 and level 2 exclusive cache, but the per-processor inclusive 8 MB last-level cache (LLC) is shared between all cores of the same processor. We refer to the subsystem incorporating the LLC, the arbitration mechanisms, and the memory controllers as the *uncore* (the uncore is marked with dotted lines on Figure 2.1).

When a processor accesses a memory location, there are many different locations that can hold the data (e.g., local or remote caches, local or remote RAM). Similarly, there can be several memory requests outstanding, from multiple cores and processors, in flight at any point of time, so a routing and arbitration mechanism for these requests is necessary. On the Nehalem, a part of the uncore called the Global Queue (GQ) arbitrates these requests [44]. The GQ controls and buffers data requests coming from different subsystems of the processor. For each subsystem (processor cores, L3 cache, IMC, and QPI) there is a separate port at the GQ, as shown in Figure 2.2. Requests to local and remote memory are tracked separately. As many different types of accesses go through the GQ, the fairness of the GQ is crucial to assure that each subsystem experiences the same service quality in terms of the share of the total system bandwidth.

Intel Nehalem processors feature a dynamic overclocking mechanism called Turbo Boost that allows raising the clock rate of processor cores over their nominal rate if the per-processor thermal and power limits still remain within the processor's design specifications [22]. Turbo Boost results in a performance improvement of up to 6.6% in both single- and multithreaded configurations of the benchmarks we use, but we disable it (together with dynamic frequency scaling) to improve the stability of our measurements and to allow a focus on the memory system interface. The hardware and adjacent cache line prefetchers are enabled for all experiments.

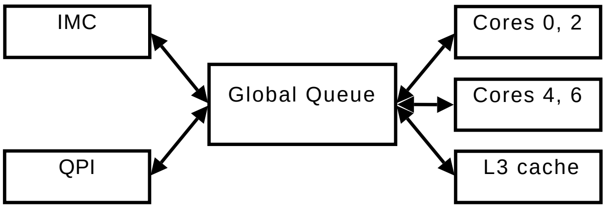


Figure 2.2: Global Queue.

Although our 8-core 2-processor evaluation system is small, it nevertheless allows interesting experiments, as it already offers the opportunity to study the interaction between local and remote memory accesses. It is possible to build larger systems (up to eight sockets) based on the Intel Nehalem microarchitecture. These systems use a processor with a larger number of QPIs to allow point-to-point connections between all processors. We used such a system with 4 processors and 32 cores, but the uncore of these systems is more complicated [42]. Thus, a presentation of the possible interactions between the uncore components would make the experimental analysis much more complicated without giving significantly more (or different) insight into the problem under investigation (i.e., the sharing of bandwidth between local/remote memory accesses). Nevertheless, to assess the performance implications of having more than four cores per processor, in Section 2.1.7 we briefly evaluate the memory system performance of the 6-core die shrink of the Nehalem, the Westmere.

Benchmarks

We use the `triad` workload of the STREAM benchmark suite [71] to evaluate the sustainable memory bandwidth of individual cores, processors, and the complete system. The `triad` workload is a program with a single execution phase with high memory demands. Figure 2.3 shows the main loop of `triad`: `triad` operates on three arrays of double-precision floating-point numbers (`a[]`, `b[]`, and `c[]`, as shown in the figure); the arrays must be sized so that they are larger than the last-level cache to cause memory controller traffic.

```
for (i = 0; i < ARRAY_SIZE; i++)
{
    a[i] = b[i] + SCALAR * c[i];
}
```

Figure 2.3: `triad` main loop.

A single instance of the `triad` workload is not capable of saturating any of the memory interfaces of our evaluation machine, thus it does not allow us to explore the limits of the machine's main memory subsystem. Besides, a single `triad` instance does not allow for evaluating the interaction between the different types of memory controllers, because we need at least one `triad` instance for each type of memory controller to have two types of memory accesses in the system at the same time. Hence, we construct multiprogrammed workloads that consist of co-executing instances of `triad` (also referred to as `triad` clones). We refer to these workloads as x`P`, where x is the number of `triad` clones the workload is composed of (e.g., `3P` represents a workload composed of three `triad` clones).

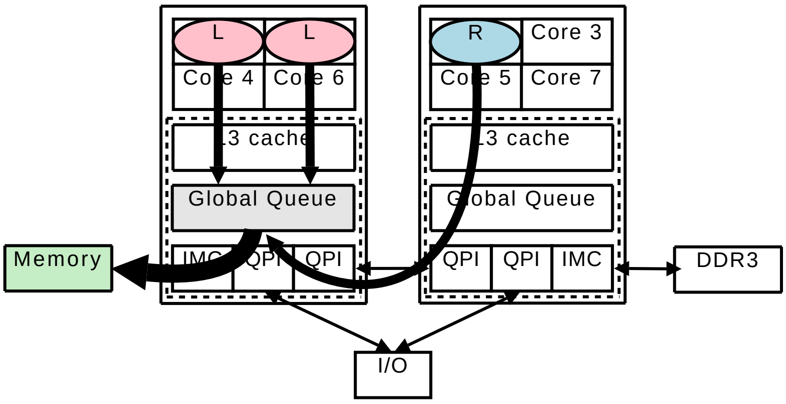


Figure 2.4: `3P` workload in `(2L, 1R)` configuration.

To match the application class we study in this chapter (multiprogrammed workloads), the version of the `triad` benchmark we use is multiprogrammed and uses processes. (The original implementation of the STREAM benchmark suite is implemented with OpenMP and uses threads.) As a result of using processes, there is no data shared between co-executing `triad` clones. The Nehalem microarchitecture implements the MESIF cache coherency protocol, and accesses to cache lines in each different state (i.e., **M**odified, **E**xclusive, **S**hared, **I**nvalid, or **F**orwarding) involve different access latencies [73]. By using a multiprogrammed benchmark, we restrict the types of cache lines accessed to M, E, and I. Therefore, we do not need to account for the different latencies of accesses to cache lines in all possible states. As a result, our measurement data are easier to interpret and to understand. Nevertheless, our analysis can be easily extended to evaluate the bandwidth sharing properties of accesses to cache lines in other coherency states as well, following the methodology described in [73].

There are two useful properties of `triad` that make it well suited for the main memory system evaluation. First, `triad`'s cache miss rate per instruction executed is the same (around 53 misses per thousand instructions) for all configurations of the workload (for configurations when the workload is composed of several `triad` clones as well as the configuration when the workload consists of a single `triad` instance); that is, co-executing `triad` clones that share a LLC do not cause additional inter-core misses [93] to each other. Therefore, `triad` is a *cache gobbler* type of program (according to the classification proposed by Sandberg et al. in [90]). The second useful property of `triad` is that 94–99% of its read memory accesses that request data from the LLC miss the LLC and are therefore served by main memory (for details about write accesses see Section 2.1.2).

As a result of these properties, `triad` slows down only because of bandwidth saturation, increased memory access latencies, and contention on the memory controllers of the system, but not due to an increase of its LLC miss rate. Hence, `triad` is well suited for evaluating the throughput of the memory interfaces, and its performance is not influenced by caching effects (e.g., cache contention) at all (see Section 2.2 for an analysis that considers cache contention as well).

We use standard Linux system calls [56] to control on which processor the memory is allocated and where the operating system processes executing the triad clones are scheduled. We use the terms `triad` process and `triad` clone interchangeably, as there is a one-to-one mapping between a clone (an instance of the `triad` program) and the process executing it. To

Bit type	Bit name
Request	`DMND_DATA_READ`
	`DMND_DATA_RFO`
	`DMND_DATA_IFETCH`
	`PF_DATA_READ`
	`PF_DATA_RFO`
	`PF_DATA_IFETCH`
Response	`LOCAL_DRAM`
	`REMOTE_DRAM`

Table 2.1: Configuration of `OFFCORE_RESPONSE_0`.

evaluate the interaction between the IMC and the QPI, the memory used by `triad` processes is always allocated on a single processor, Processor 0, and we change only the process-to-core mapping in our experiments. The workloads can execute in multiple configurations, depending on the number of `triad` processes mapped onto the same processor. The terms *local* and *remote* are always relative to the processor that holds the data in memory. We denote with *x*L (*y*R) the number *x* (*y*) of local (remote) processes. For example, a three-process `3P` workload executing in the `(2L, 1R)` configuration means that two cores access memory locally and one core accesses memory remotely, as shown in Figure 2.4. We refer to the instances of the workload executing locally as *L processes* and we call instances executing remotely *R processes*. Memory accesses of L processes must pass just through the Global Queue and the IMC, while R processes have the additional overhead of passing through the processor cross-chip interconnect (QPI) and the GQ of the remote processor. The datapaths used by the `(2L, 1R)` workload are also illustrated in Figure 2.4.

Measurements and methodology

The Nehalem machine runs Linux 2.6.30 patched with `perfmon2` [30]. We use the processor's performance monitoring unit (PMU) to obtain information about the elapsed CPU cycles and the amount of last-level cache (LLC) misses a program generates. For this purpose we use the performance monitoring events `UNHALTED_CORE_CYCLES` and `OFFCORE_RESPONSE_0` [45], respectively. These performance monitoring events allow the measured quantities to be attributed to individual cores of a processor, thus they are referred to as *per-core events*. The `OFFCORE_RESPONSE_0` event is configured with a bitmask, each bit of the bitmask corresponds to a type of request/response processed by the processor's uncore. Table 2.1 shows the bits in the bitmask we use for the measurements.

We calculate the generated memory bandwidth using Equation 2.1. The cache line size of the LLC of the Intel Nehalem is 64 bytes. The processors in our system are clocked at 2.26 GHz. We report only the read bandwidth generated by cores because with the per-core PMU it is possible to measure only the read bandwidth, but not the write bandwidth generated by cores [44].

$$bandwidth = \frac{64 \cdot LLC\ misses \cdot 2.26 \cdot 10^9}{CPU\ cycles \cdot 10^6} MB/s \tag{2.1}$$

In addition to per-core performance monitoring events, Nehalem-based systems support *uncore events*. Accordingly, the processor's uncore has a PMU as well, in addition to the PMUs of the individual processor cores. Uncore events can be used to measure quantities for the complete system, but, unlike per-core events, they do no allow quantities to be attributed to individual cores. We use the uncore PMU to monitor the state of the GQ and to cross-check the bandwidth readings obtained with the per-core performance counters (in which case we compare cumulative per-core event counts with the corresponding uncore counts).

We compile the `triad` workload with the `gcc` compiler version 4.3.3, optimization level `O2`. To reduce the variation of the results, OS address space layout randomization is disabled and we also reduce the number of services running concurrently with our benchmarks as much as possible. As the `triad` workload is bound on memory bandwidth and has a single program phase, its bandwidth and performance readings are very stable and do not depend on the factors reported by Mytkowicz et al. [75] (e.g., the size of the UNIX environment and link-order). All measurement data are the average of three measurement runs; the variation of the measurements is negligible.

2.1.2 Memory system performance

To measure the bandwidth sharing properties of the Nehalem microarchitecture, we measure the bandwidth achieved by each instance of the `triad` benchmark. We configure the benchmark with a number of processes ranging from one to eight (the number of cores on our machine), and then measure all possible local-remote mapping configurations for any given number of `triad` processes. Recall that only per-process read bandwidth can be reported due to limitations of the processor's PMU. However, the total amount of read and write bandwidth on the interfaces of the system can be measured using uncore performance counters (using the `UNC_QHL_NORMAL_READS` and `UNC_QHL_WRITES` events). These measurements show that the `triad` benchmark is read-intensive, as in any configuration around 75% of the total main memory bandwidth is caused by reads, and around 25% is due to writes. Therefore, the measurements of the read bandwidth of the system are representative for the behavior of the memory interfaces of the Nehalem-based system.

Figure 2.5 shows results for the scenario where four local processes share the IMC bandwidth with different numbers of remote processes. If there are no remote processes, the complete bandwidth is allocated to local requests `(4L, 0R)`. As a single remote process is added (resulting in configuration `(4L, 1R)`), the total bandwidth increases. Then, as the number of remote processes further increases, the total bandwidth is reduced slightly (configurations `(4L, 2R)` to `(4L, 4R)`). Two remote processes consume the maximal bandwidth that can be obtained through remote accesses; the share of the remote processes does not grow as we increase the number of remote processes. As a consequence, each remote process realizes a smaller and smaller absolute memory bandwidth.

The data in Figure 2.5 might convince a system developer to favor mapping a process onto the processor that holds the data locally. However, the situation is more complex: Figure 2.5 shows the total bandwidth achieved by all processes. Figure 2.6 contrasts Figure 2.5 by showing the performance of individual R and L processes. If there is a single L and a single R process, the L process captures almost 50% more of the memory bandwidth (L: 6776 MB/s, R: 4412 MB/s). As the number of L processes increases, these L processes compete for local access, and although the R process's declines as well (to 3472 MB/s), the bandwidth obtained by each L process declines a lot more (to 2622 MB/s).

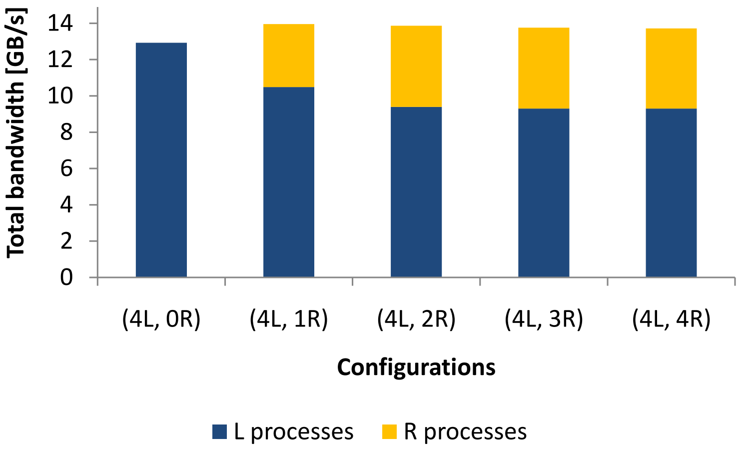


Figure 2.5: Bandwidth sharing: 4 L processes with variable number of R processes (Nehalem).

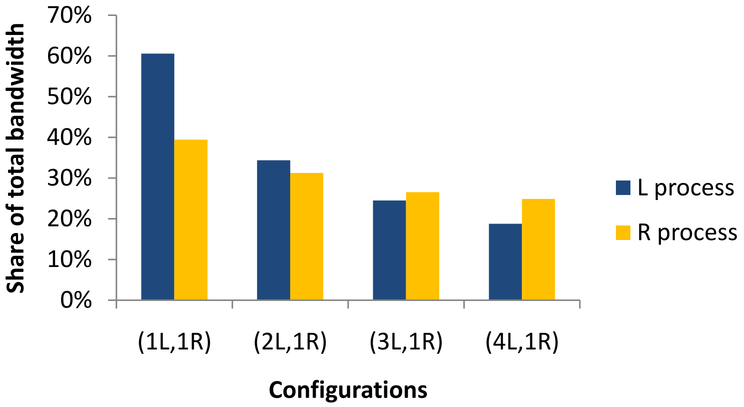


Figure 2.6: Percentage of total memory bandwidth obtained by an L and an R process (Nehalem).

Table 2.2 shows the complete measurement data. Each row reports the bandwidth obtained by an L instance in the presence of varying number of R processes. Table 2.3 shows the bandwidth obtained by an R process in the same set of configurations as in Table 2.2. Figure 2.6 contrasts column 2 of Tables 2.2 and 2.3. Tables 2.4 and 2.5 report the cumulative data (for all the L respectively R processes in an experiment). Figure 2.5 is based on the last row of Tables 2.4 and 2.5; the sum of these rows yields the total shown in the figure.

2.1.3 A simple model

Our experiments show that if there are only local processes running on the system, the total bandwidth obtained by these processes can be described as:

$$bw_{L_{total}} = min(active\_cores * bw_L, bw_{L_{max}}) \tag{2.2}$$

In Equation 2.2, bw_L is the bandwidth of a single, locally executing `triad` clone (L process) (see column “0 R” of Table 2.2). If the sum of the bandwidth of the individual cores $bw_{L_{total}}$ is greater than the threshold $bw_{L_{max}}$ (see column “0 R” of Table 2.4 for the exact values), each core obtains an equal share of the threshold value (12925 MB/s).

	0 R	1 R	2 R	3 R	4 R
0 L	0	0	0	0	0
1 L	7656	6776	6325	6185	6210
2 L	5512	4460	4189	4142	4121
3 L	4202	3389	3078	3047	3048
4 L	3231	2622	2348	2325	2326

Table 2.2: Per-core L bandwidth [MB/s].

	0 R	1 R	2 R	3 R	4 R
0 L	0	4844	3499	2313	1702
1 L	0	4412	3010	2007	1486
2 L	0	4056	2664	1778	1319
3 L	0	3675	2383	1581	1175
4 L	0	3472	2236	1486	1104

Table 2.3: Per-core R bandwidth [MB/s].

	0 R	1 R	2 R	3 R	4 R
0 L	0	0	0	0	0
1 L	7656	6776	6325	6185	6210
2 L	11024	8921	8379	8283	8242
3 L	12607	10167	9235	9141	9145
4 L	12925	10487	9393	9299	9302

Table 2.4: Total L bandwidth [MB/s].

	0 R	1 R	2 R	3 R	4 R
0 L	0	4844	6998	6938	6807
1 L	0	4412	6020	6020	5945
2 L	0	4056	5329	5335	5275
3 L	0	3675	4765	4742	4699
4 L	0	3472	4472	4459	4416

Table 2.5: Total R bandwidth [MB/s].

	0 R	1 R	2 R	3 R	4 R
0 L	0	4844	6998	6938	6807
1 L	7656	11188	12345	12205	12155
2 L	11024	12977	13708	13618	13517
3 L	12607	13842	14001	13882	13844
4 L	12925	13959	13865	13758	13719

Table 2.6: Total cumulative bandwidth [MB/s].

Similarly, the total bandwidth obtained by remote processes can be characterized as:

$$bw_{R_{total}} = min(active\_cores * bw_R, bw_{R_{max}}) \tag{2.3}$$

In Equation 2.3, bw_R is the bandwidth achieved by a single `triad` instance executing remotely (R process) (see column “0 L” of Table 2.3). The maximum throughput of the R processes ($bw_{R_{max}}$) is limited by the QPI interface and is 6998 MB/s (experimentally determined). The QPI is also fair in the sense that if the threshold is to be exceeded, each R processes obtains an equal share of the total bandwidth.

The total bandwidth obtained by the system is composed of the bandwidth achieved by L and R processes and is shown in Table 2.6 for all configurations of the `triad` benchmark. The limit $bw_{L_{max}}$ of L processes can be observed in row “4 L” column “0 R” of Table 2.6. Similarly, the limit $bw_{R_{max}}$ of R processes can be observed in row “0 L” and column “2 R” of Table 2.6. Remote processes hit their limit $bw_{R_{max}}$ with two active cores, while four local processes are needed to hit the limit $bw_{L_{max}}$. This is because the QPI is already saturated by two `triad` clones, however all four cores need to be active to saturate the IMC. Next generations of the Nehalem have a larger number of cores connected to the same local memory controller, therefore not all cores of a processor are required to achieve the saturation limit of the IMC. In Section 2.1.7 we briefly look at such a machine.

Formally the total bandwidth in the system can be expressed as:

$$bw_{total} = (1 - \beta) * bw_{L_{total}} + \beta * bw_{R_{total}} \tag{2.4}$$

We call the variable β the *sharing factor*. The sharing factor determines the share of the total bandwidth received by local and remote `triad` clones. β is a real value between 0 and 1. If β is 1, all bandwidth is obtained by R processes. Similarly, if β is 0, all bandwidth is obtained by L processes. The value of β is not constant: As the Global Queue (GQ) arbitrates between local and remote memory accesses, the GQ determines the value of β based on the arrival rate of requests at its ports.

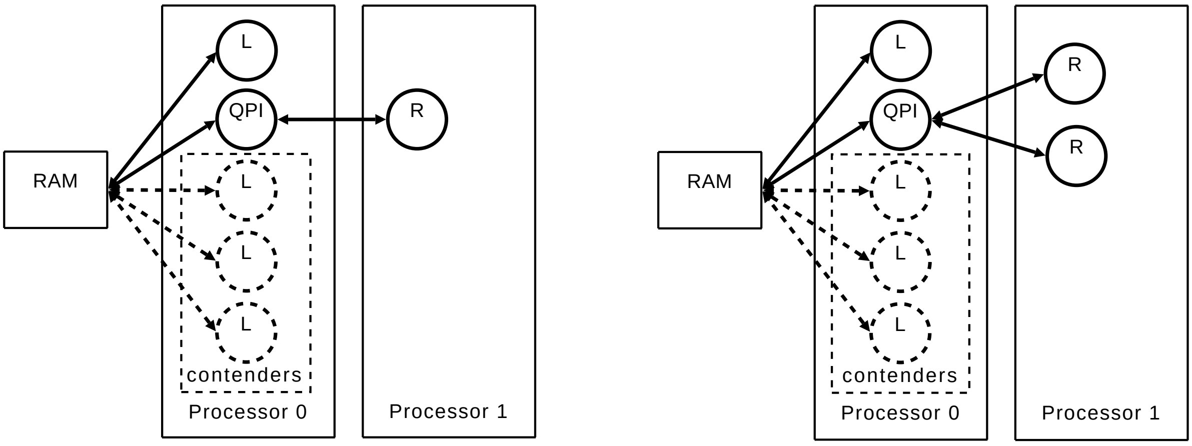
If the system must handle memory requests coming from a small number of cores, the bandwidth (and thus the performance) of local processes is much better than the bandwidth of remote ones. As the load on the system increases and there are more local processes, the bandwidth obtained by individual local processes (bw_L) becomes comparable to the cumulative bandwidth of the QPI ($bw_{R_{total}}$). Situations when the bandwidth of the QPI is better than the bandwidth of individual local processes are also possible (e.g., configuration `(4L, 1R)` and `(3L, 1R)`). Overloading the QPI with a large number of remotely executing memory-bound processes should be avoided, as the lower throughput of the QPI interface is divided between R processes, resulting in low performance of R processes, if their number is too large. In conclusion, if the memory system has a low utilization, local execution is preferred. Nevertheless, as the load on the memory system increases, remote execution becomes more favorable, but care needs to be taken not to overload the cross-chip interconnect.

To fully understand the system, the dependence of the sharing factor β of the GQ on the load coming from the local cores and remote memory interfaces needs to be characterized. However, as most implementation details of the Nehalem queuing system are not disclosed, moreover, the performance monitoring subsystem of our Nehalem-based processor does not allow for measuring queue status directly, such a model is difficult to construct. Instead, we describe two empirically observed properties of the GQ that help understanding the bandwidth sharing properties of our evaluation system: queuing fairness (Section 2.1.4) and aggregate throughput (Section 2.1.5).

2.1.4 Queuing fairness

Table 2.5 shows that for any number of local processes there is a significant difference between the throughput of the non-saturated QPI executing a single R process (the “1 R” column), and the throughput of the QPI transferring the data for two R processes (the column labeled “2 R”). Adding more R processes (columns “3 R” and “4 R”) does not modify the overall bandwidth allocation of the system, as the throughput limit of the QPI has already been reached, and the QPI is saturated. However, a large difference in the total bandwidth obtained by the L and R processes is observed by varying the number of L processes (rows “1 L” to “4 L”).

In the following, we consider the QPI as a fifth *agent* connected to the GQ (in addition to the four local cores), executing either the 1R workload, or a workload equivalent to the memory intensity generated by the 2R workload. We take as baseline the performance of two cases. In the first case, the GQ is serving 1R from the QPI and 1L from the local cores, as depicted by Figure 2.7(a). In the second case, the GQ is serving the 2R in combination with 1L, as depicted by Figure 2.7(b). Using the previously defined notation, these workloads can



(a) Baseline workload: `(1L, 1R)`. (b) Baseline workload: `(1L, 2R)`.

Figure 2.7: Setup to evaluate GQ fairness.

be denoted with `(1L, 1R)` and `(1L, 2R)`. To increase the contention on the GQ, one, two, or three additional L process(es) are executed on the system. These L processes (the base L process plus the additional L processes) contend with the QPI for IMC bandwidth.

Figure 2.8 shows the variation of the sharing factor (parameter β of Equation 2.4) when contention on the local port of the GQ increases. The sharing factor depends on the load on the GQ: the more traffic L processes generate, the larger a share of the bandwidth they obtain, and the more the share of the R processes (given by β) decreases. Nonetheless, if we consider the performance degradation of the two baseline workloads `(1L, 1R)` and `(1L, 2R)` (shown in Figure 2.9 and in Figure 2.10, respectively), the performance of individual L process in each of the two workloads degrades more than the performance of the QPI does. Therefore, the more load there is on the GQ, the more attractive it is to execute some processes remotely.

In conclusion, if the GQ is contended, the Nehalem microarchitecture is unfair towards local cores (vs. the QPI), as local cores experience a performance degradation that is larger than the performance degradation of the QPI. Still, this behavior is reasonable as the GQ does not allow remote cores to starve, and thus it avoids further aggravating the penalty of remote memory accesses. Nevertheless, this property of the Nehalem is undocumented and can be discovered only with experimental evaluation.

2.1.5 Aggregate throughput

To further motivate the benefit of having a good proportion of local and remote memory accesses, we show in Figure 2.11 the total system throughput for the 4P workload in different mapping configurations (ranging from the configuration when all processes execute locally to the configuration with all processes executing remotely). In the configurations with some remote memory accesses the throughput of the memory system can be better (at a peak of 13842 MB/s) relative to the configuration when all memory accesses are local (12925 MB/s).

To take a closer look at the total system throughput, we examine two cases. First, we map the processes of the `triad` workload onto local cores. This way, all memory operations use the local ports of the Global Queue. Then, we move one process to the remote processor, thus the QPI port of the GQ is also used to actively handle memory requests. For both cases, we compute

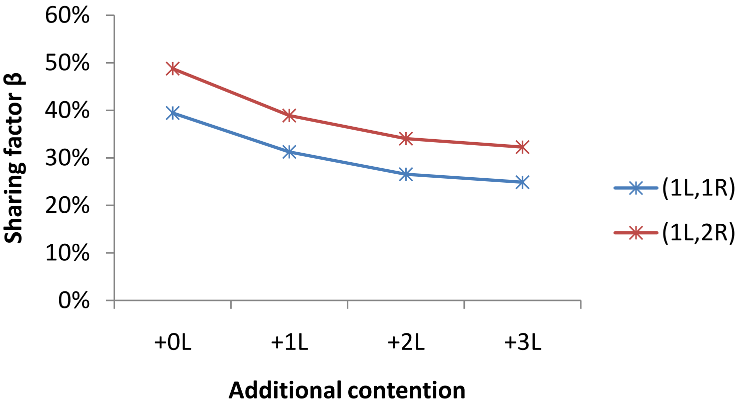


Figure 2.8: Dependence of β on aggregate load.

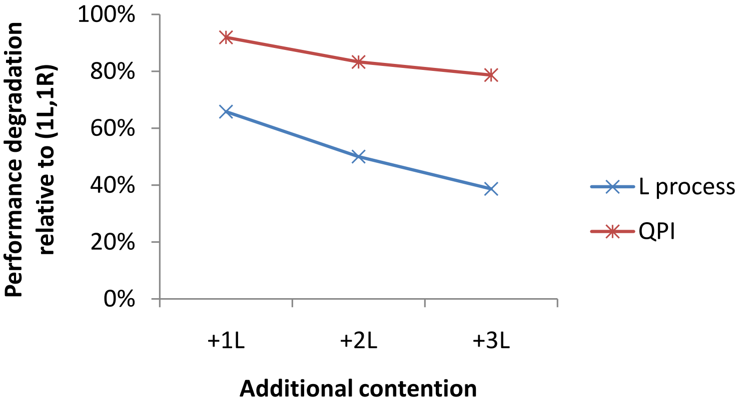


Figure 2.9: Performance degradation of `(1L, 1R)`.

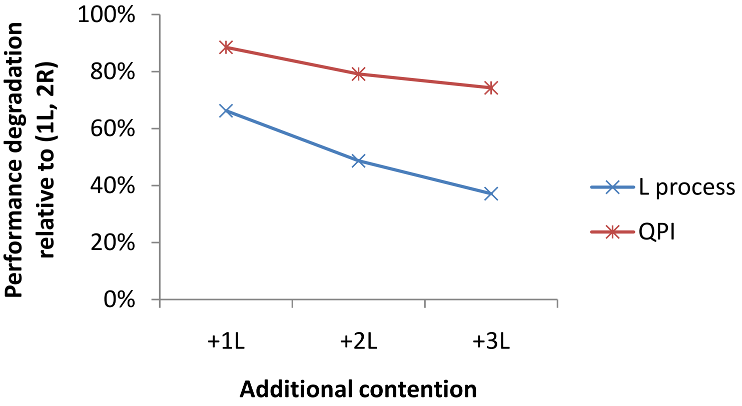


Figure 2.10: Performance degradation of `(1L, 2R)`.

overall system throughput as the sum of the instructions per cycle (IPC) values obtained by the processes:

$$IPC_{total} = \sum_{p \in Processes} IPC_p \tag{2.5}$$

The number of instructions executed by the `triad` workload is measured using the performance monitoring event `INSTRUCTIONS_EXECUTED`.

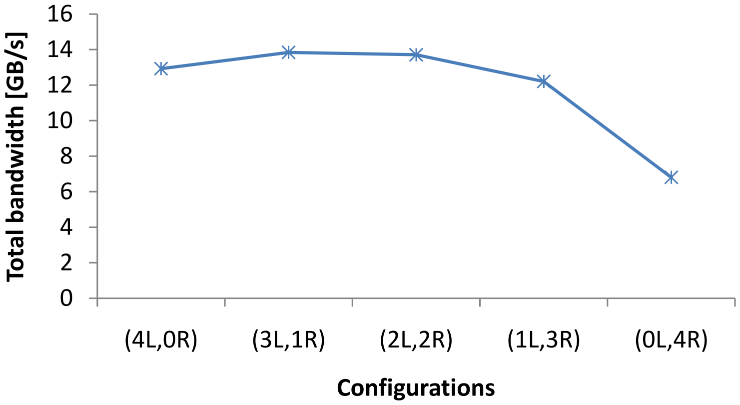


Figure 2.11: Total bandwidth of the `4P` workload in different configurations.

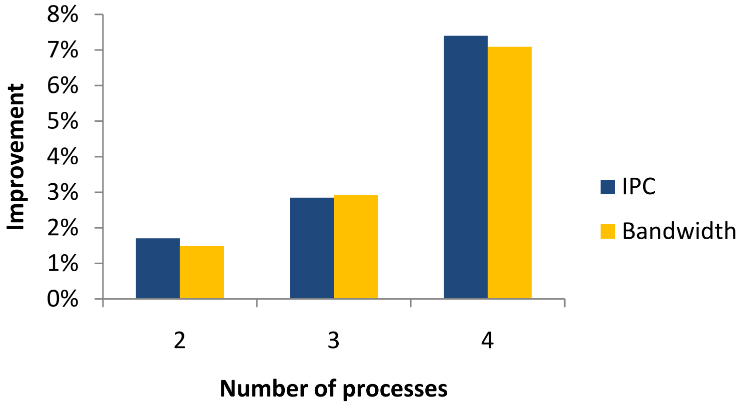


Figure 2.12: Improvement of aggregate IPC and total memory bandwidth when an IMC and a QPI are used (relative to the case when a single IMC is used).

We are aware that in case of heterogeneous workloads (workloads that execute different instruction streams) using the metric defined by Equation 2.5 may not be appropriate, as pointed out by Eyerman [31]. However, in our case all processes execute the same tight memory-intensive loop (shown in Figure 2.3) that processes identically sized data, therefore the instructions executed by each workload are the same. The clock rate of all processor cores is also the same, so the ratio of instructions executed and cycles consumed is a precise measure for system throughput. As most memory accesses of `triad` miss the last-level cache (and are thus served by main memory), the aggregate memory bandwidth achieved on the system is also directly proportional to the system throughput. This metric does not characterize the fairness of the system, but it accurately reflects the throughput of the main memory system.

Figure 2.12 shows the benefit of mapping one process remotely over the all-local case (where all memory requests come from local cores). The benefit is minor (1.7%) if there are just two processes running on the system, but it gets significant (7.4 %) if there are four processes. This increase of performance in the four-process case can be explained by the distribution of contention on the GQ. When the GQ handles four locally executing `triad` clones, its local port is saturated (it is full 10% of the time). (GQ saturation is measured with the `UNC_GQ_CYCLES_FULL` event [45]). Moving one process to the remote processor transfers some of load from the local port of the GQ to its remote port. In this new configuration neither the local nor the remote port of the GQ is saturated, therefore system throughput increases.

However, if all processes execute remotely, the remote port of the GQ gets saturated (it is full 31% of the time).

In conclusion, in a single-threaded context the bandwidth and latency of the on-chip memory interface greatly outperform the same parameters of the QPI. However, in the case when multiple cores are competing, this advantage diminishes as contention on the queuing system increases. Distributing computations such that there are both local and remote accesses in the system helps to improve aggregate throughput.

2.1.6 Limitations

In our analysis we did not account for the overhead of the cache coherency protocol. On every cache miss, there is a snoop request towards the cache of the adjacent processor (as measured on the read-, write-, and peer–probe-tracker of each processor's uncore). Snoop requests are transferred on the cross-chip interconnect of the system. However, while normal reads usually request data of the size equal to a cache line, we do not know the amount of data transferred with a snoop request. Therefore, we cannot calculate the amount of traffic generated by snoop requests, and so we cannot calculate the bandwidth overhead of the cache coherency protocol.

We use a single, homogeneous workload (composed of multiple `triad` clones) to evaluate the memory system performance of a NUMA-multicore machine. Because the `triad` benchmark does not benefit from caching (at least not at the level of the LLC), `triad`'s performance describes the performance of the memory interfaces of our evaluation system well. In case of heterogeneous workloads (workloads composed of multiple programs with different memory intensity that have possibly more cache locality than `triad`), however, caching effects also come into play in addition to memory controller throughput. We discuss this scenario in Section 2.2.

2.1.7 The next generation

In 2010 Intel released a die shrink of the Nehalem codenamed Westmere. To see if the previously described principles apply also to systems based on the Westmere microarchitecture, we perform all experiments with a Westmere-based machine as well. The Westmere-based system we evaluate has also two processors, however it shows some differences to the Nehalem-based machine we have looked at previously. The most important difference is that the processors in our Westmere-based system contain six cores per processor (two cores more than the processors of the Nehalem-based system). A detailed comparison of the two systems is shown in Table 2.7.

We conduct the same set of experiments with the Westmere as with the Nehalem, but we do not present the complete data. Instead, we present two projections of the Westmere data, similar to the ones presented for the Nehalem in Figure 2.5 and Figure 2.6, respectively. In Figure 2.13 four local processes share the bandwidth of the IMC of Processor 0. As one R process is added, the total achieved bandwidth increases. Adding more R processes increases the share of R processes until the saturation limit of the QPI is achieved (in the case of Westmere four R processes are required to saturate the QPI versus two R processes in the case of the Nehalem).

Figure 2.13 shows a breakdown of the total bandwidth of the two types of processes, L and R. Figure 2.14 shows the memory bandwidth of a single L resp. R process with increasing number of L processes. On the Westmere a single R processes is able to achieve more bandwidth

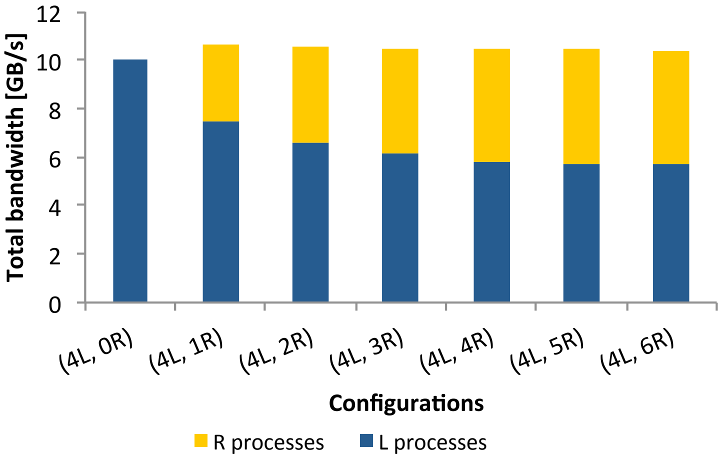


Figure 2.13: Bandwidth sharing: 4 L processes with variable number of R processes (Westmere).

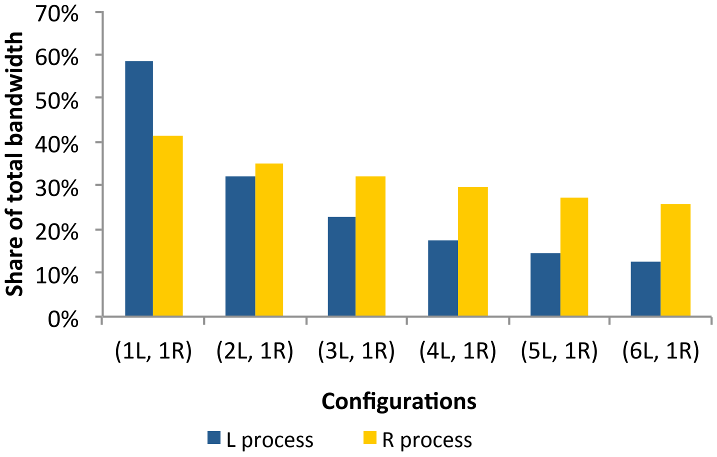


Figure 2.14: Percentage of total memory bandwidth obtained by an L and an R process (Westmere).

	Nehalem	Westmere
Model number	Intel Xeon E5520	Intel Xeon X5680
Number of processors	2	2
Cores per processor	4	6
Clock frequency	2.26 GHz	3.33 GHz
L3 cache size	2x8 MB	2x12 MB
IMC bandwidth	2x25.6 GB/s	2x19.2 GB/s
QPI bandwidth	2x11.72 GB/s (2x5.86 GTransfers/s)	2x12.8 GB/s (2x6.4 GTransfers/s)
Main memory	2x6 GB DDR3	2x72 GB DDR3

Table 2.7: Parameters of the evaluation machines.

than an L process already in the configuration with two locally executing processes.

To compare the Nehalem to the Westmere, Figures 2.15 and 2.16 show the total read bandwidth measured on the Nehalem and the Westmere, respectively. As the Westmere includes two

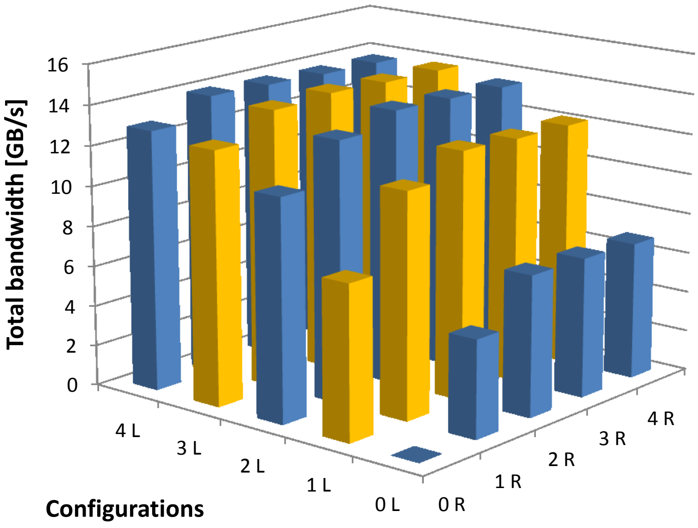


Figure 2.15: Total read bandwidth (Nehalem).

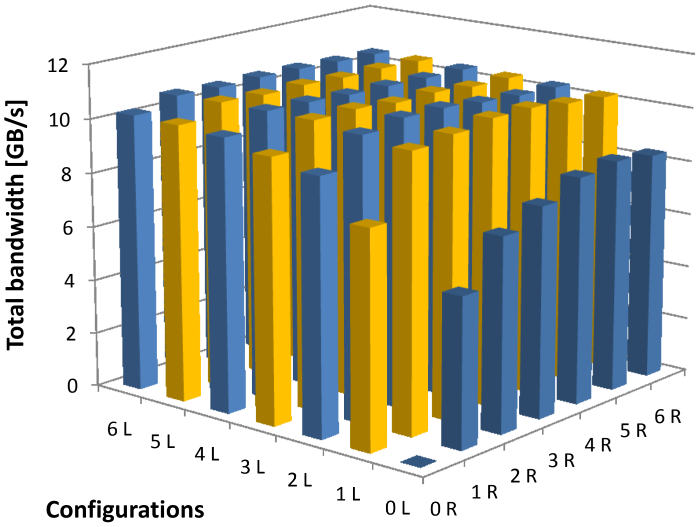


Figure 2.16: Total read bandwidth (Westmere).

cores more than the Nehalem, the thresholds $bw_{L_{max}}$ and $bw_{R_{max}}$ defined in Equations 2.2 and 2.3 are more prominent than on the Nehalem: on the Westmere four `triad` clones are required to saturate the QPI (vs. two on the Nehalem), while the IMC saturates with four `triad` clones, just as on the Nehalem.

In conclusion, the principles we describe for the Nehalem also apply for the Westmere microarchitecture. Because the Westmere has more cores, a different LLC size, and memory interfaces with slightly different throughput as the Nehalem, the bandwidth sharing properties of this machine are quantitatively, but not qualitatively, different.

2.1.8 Summary

Today's multicore processors integrate a memory controller with the cores and caches on a single chip. Such a design leads to the new generation of NUMA multicore-multiprocessors that present software developers with a new set of challenges and create a different class of performance optimization problems. The cores put pressure on the memory controller to service the local memory access requests while, at the same time, the memory controller must deal with requests by other processors as well. So it is important that the software finds a balance between local and remote memory accesses if overall performance is to be optimized.

In this section (Section 2.1) we presented an experimental analysis of the bandwidth sharing properties of two commercially available multicore systems, the Intel Nehalem and its die-shrink, the Intel Westmere. The evaluation shows that if a large part or all of the cores of a processor are active and thus contention on memory controllers is high, favoring data locality may not lead to optimal performance. In addition to data locality, the bandwidth limits of the memory controllers and the fairness of the arbitration between local and remote accesses are important. Moreover, the overhead of arbitration and queuing is likely to become more important in larger systems as the complexity of this mechanism increases with a growing number of processors in the system. Therefore, it is important that software developers understand the memory system to be able to balance the memory system demands on such a system so that the best tradeoff between local and remote accesses can be found.

2.2 Cache contention and interconnect overhead

In recent multicore architectures (including the Nehalem- and Westmere-based systems analyzed in Section 2.1) there are usually multiple cores connected to a single last-level cache (LLC). Contention for shared LLCs (short: cache contention) appears when multiple, independent programs (e.g., programs of a multiprogrammed workload) are mapped onto a multicore machine so that several programs use the same LLC simultaneously. Programs using the same LLC at the same time can evict each other's cache lines. Such activities result in less cache capacity available to each program (relative to the case when a program uses the cache alone). Therefore, for memory intensive programs cache contention can result in severe performance degradation, as shown by [8, 14, 20, 34, 40, 47, 52, 69, 70, 84, 97, 113].

Section 2.1 analyzes the relative cost of memory controller contention and interconnect overhead by looking at the bandwidth sharing properties of Intel Nehalem- and Westmere-based machines. In this this section we extend the experimental analysis to take into account the relative cost of cache contention as well. Section 2.2.1 describes the experimental setup we use, in Section 2.2.2 we discuss the effects of cache contention on application performance.

2.2.1 Experimental setup

The analysis in Section 2.1 uses the `triad` workload for experiments. The `triad` workload has a cache miss rate of almost 100% and thus its performance critically depends only on the available memory bandwidth, but not at all on the amount of cache capacity available to it. As a result, `triad` is well suited to analyze the relative cost of memory controller contention and interconnect overhead, but it cannot be used to compare the cost of cache contention and the cost of interconnect overhead.

In the current setup we consider programs of the SPEC CPU2006 benchmark suite, instead of `triad`. Programs in the SPEC CPU2006 suite have various memory intensities, Sandberg et al. [90] provides a classification of programs based on their memory behavior. Some programs are not memory intensive at all, that is, their performance does not depend on the memory system at all (i.e., the "don't care" category in the classification provided by the paper). Some programs are similar to `triad`, that is, they are memory intensive but their performance does not dependent on the available LLC capacity (i.e., the "cache gobbler" category in the classification). Finally, numerous programs in the suite are memory intensive and their performance closely depends on the amount of LLC capacity available (e.g., "victim" and "gobblers and victims" category in the classification in [90]).

We use programs from the "gobblers and victims" category to evaluate the relative cost of cache contention and interconnect overhead. The programs are executed with the reference input size (the largest input size available in the suite). The hardware and software environment used to run experiments is the same as the environment described in Section 2.1 (i.e., use the same Nehalem- and Westmere-based machines), but we perform a different set of the experiments. First, we analyze the performance impact of cache contention and data locality using a simple example: mapping two programs onto the 2-processor Nehalem-based system. Then, we extend the scope of the experiment and analyze the impact of cache contention and data locality by investigating the mapping of a larger number of programs (on both the Nehalem- and Westmere-based system).

2.2.2 Memory system performance

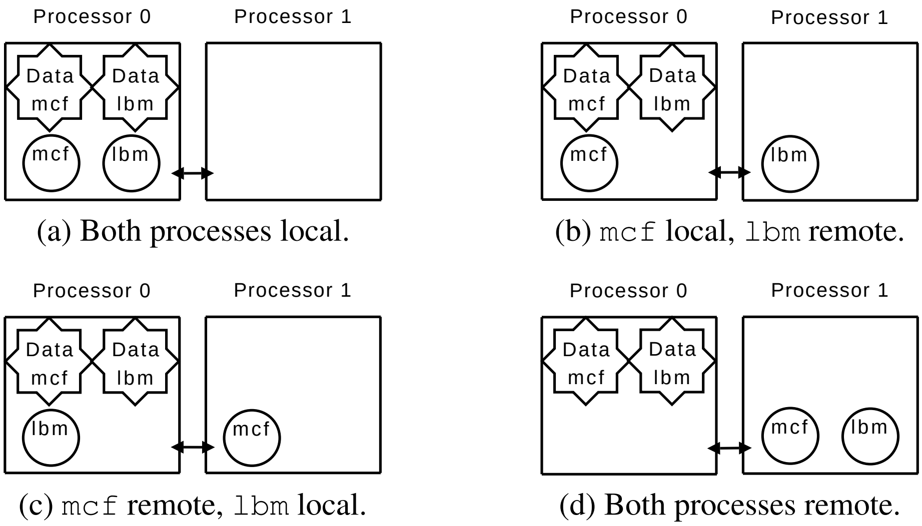
Simple example: Mapping a 2-program workload

In this section we consider a simple example: mapping two programs, `mcf` and `lbm`, onto the 2-processor Nehalem-based system. Both programs belong to the "cache gobbler and victim" category [90]. Moreover, both programs are single-threaded, so there is a one-to-one mapping between a program and the operating system process executing it. In the discussion that follows we use the term program and process interchangeably.

In a NUMA system a process's data can be allocated in the memory of any processor in the system. We say that a process p is *homed* at Processor i of the system if the process's data was allocated only at Processor i. If a process runs on its home processor, it is executed *locally*. Similarly, if a process runs on a processor different from its home processor, it is executed *remotely*. For our experiments we assume that both processes (executing `mcf` resp. `lbm`) are homed on Processor 0 of the machine. Because both programs are single-threaded, there are four ways the processes executing the two programs can be mapped onto the system given this memory allocation setup. Figure 2.17 shows all possible mappings:

(a) **Both processes executed locally.** As both processes execute on their respective home node (Processor 0), they both have fast access to main memory. As Processor 0 has only one LLC, the processes contend for the LLC capacity of Processor 0.

(b) **`mcf` executed locally, `lbm` executed remotely.** As `lbm` is executed remotely (on Processor 1), it accesses main memory through the cross-chip interconnect, therefore it experiences lower throughput and increased latency of memory accesses relative to local execution. Additionally, as the two processes execute on two different processors, they do not share an LLC, therefore there is no cache contention in the system.



(a) Both processes local.

(b) `mcf` local, `lbm` remote.

(c) `mcf` remote, `lbm` local.

(d) Both processes remote.

Figure 2.17: Possible mappings of a 2-process workload (`mcf` and `lbm`).

(c) **`mcf` executed remotely, `lbm` executed locally.** This case is similar to case (b), but in this case `mcf` uses the cross-chip interconnect to access main memory instead of `lbm`.

(d) **Both processes executed remotely.** Both processes share the LLC, and both processes execute remotely. This setup is clearly the worst possible scenario for performance, therefore we exclude this case from further investigation.

To measure how each mapping effects memory system performance, we execute the two programs simultaneously. More specifically, we start both programs at the same time. Then, if a program finishes execution earlier than the other program does, it is restarted (multiple times, if needed) and is executed until the other program has completed execution as well. We report performance as the wall clock running time of the first execution of each program. The performance degradation of a program is calculated as the percent slowdown in wall clock running time relative to the running time in *single-process mode* (i.e., when the program is executed alone and locally on the system, also referred to as *solo mode*). We use the cache miss rate per thousand instructions executed (MPKI) to characterize a program's usage of the shared LLC. If a program contends with other programs for LLC capacity, the program's MPKI increases (relative to the MPKI measured on the program's solo mode execution). A program's MPKI can be measured using the hardware performance counter events listed in Section 2.1.1. (Due to the limitations mentioned in Section 2.1.1, we can measure only read cache misses.)

Figure 2.18 shows the increase of the MPKI of `mcf` and `lbm` relative to each program's execution in single-process mode. In case (a) (both processes locally executed), the MPKI increases by 47% resp. 62% due to cache contention. In cases (b) and (c) (when the processes are mapped onto different processors, therefore different LLCs), the MPKI increases by at most 4% relative to solo mode. It is unclear what the reason for this increase is, but as the amount of increase is small, we did not consider investigating this issue further.

Good data locality is crucial for obtaining good performance in NUMA systems. Figure 2.19 shows the distribution of bandwidth over the interfaces of the system. In case (a), when both processes are executed locally, the system has good data locality: 100% of the memory bandwidth in the system is provided by the local memory interface of Processor 0. In

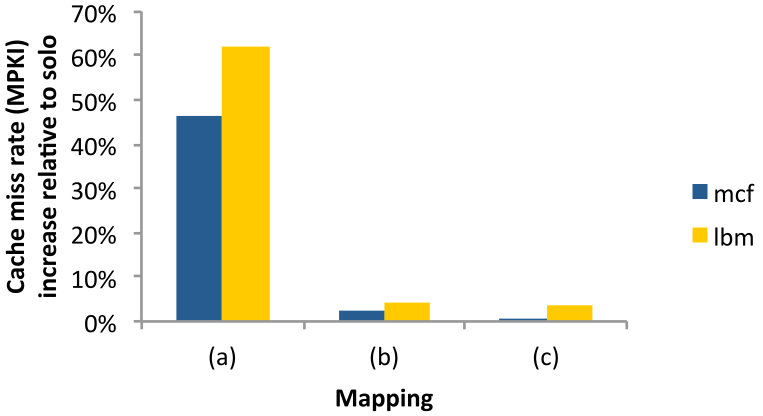


Figure 2.18: Increase of cache miss rate in different mapping scenarios.

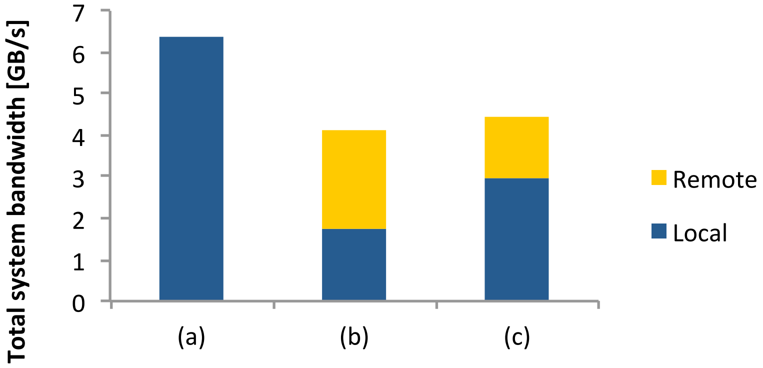


Figure 2.19: Bandwidth distribution in different mapping scenarios.

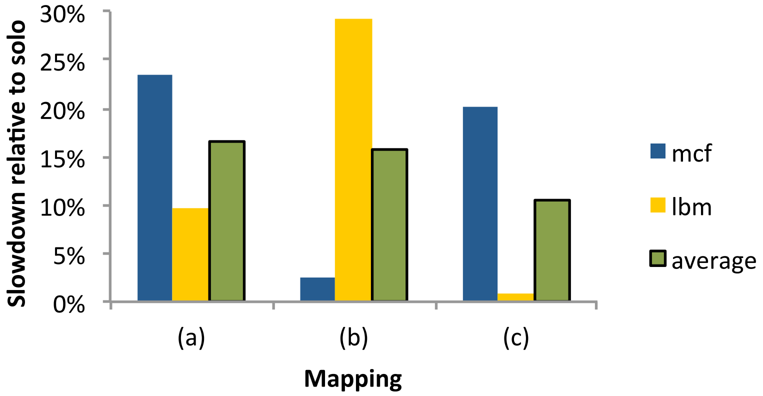


Figure 2.20: Performance degradation in different mapping scenarios.

cases (b) and (c), when one of the two processes executes on Processor 1, data is transferred also on the cross-chip interconnect of the system: 56% (resp. 33%) of the generated bandwidth is due to one of the two processes executing remotely. Figure 2.19 also shows the total bandwidth measured on the interfaces in the system. If the processes execute locally and thus share the cache (case (a)), the total bandwidth is approximately 50% higher than in cases (b) and (c) (when caches are not shared).

We investigate now: Which mapping leads to best performance: when cache contention

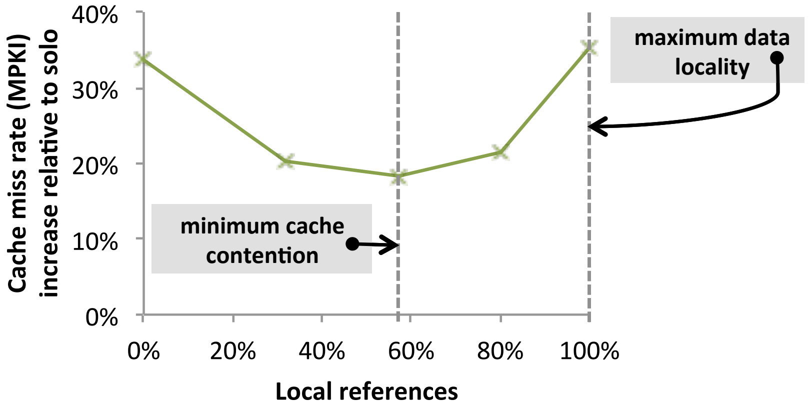


Figure 2.21: Increase of cache miss rate vs. data locality of `soplex` (Nehalem).

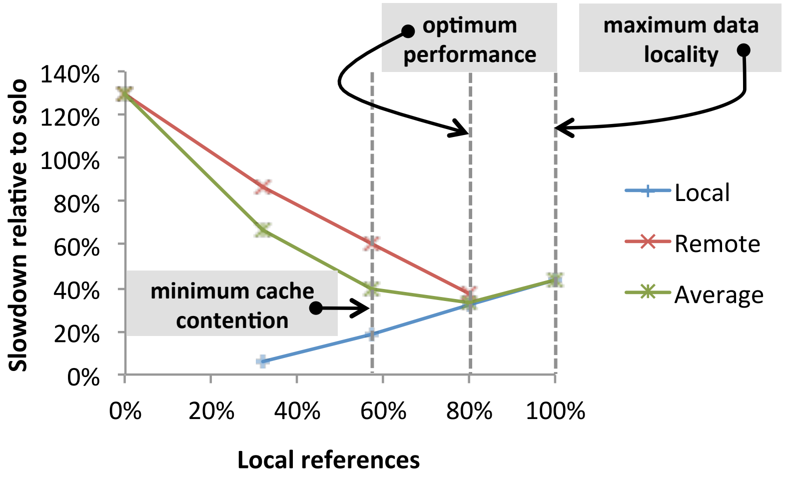


Figure 2.22: Performance vs. data locality of `soplex` (Nehalem).

is minimized (cases (b) and (c)), or when data locality is maximized (case (a)). Figure 2.20 shows the individual and average performance degradation of `mcf` and `lbm` in all three mapping scenarios. The average performance of the workload consisting of `mcf` and `lbm` is better in cases (b) and (c) than in case (a). Case (b) shows only a minor improvement over case (a) because remote execution slows down `lbm` by almost 30%. However, in case (c) the degradation is reduced relative to case (a); `mcf` sees a small improvement, `lbm`'s slowdown is reduced from 11% to 1%, so the average degradation is reduced from 17% to 11%.

Extended experiments

In general, the tradeoff between local cache contention and interconnect overhead can be observed with memory-bound programs. We focus in the presentation on the `soplex` benchmark. This program keeps large amounts of data in the caches, and its performance is hurt if the available cache capacity is reduced because other memory-bound programs use the same cache at the same time. There are several other memory-bound programs in the SPEC suite that show this behavior (see [90] for details), and the principles we discuss here are valid for these programs as well. We construct a multiprogrammed workload that consists of four identical copies (clones) of `soplex`. We allocate the memory of all clones on Processor 0 of the 2-processor

8-core Nehalem-based NUMA-multicore system. We execute the multiprogrammed workload in various mapping configurations with a different number of clones executed locally respectively remotely. The mapping configurations range from all four clones executed locally (on Processor 0) to the configuration where all four clones execute remotely (on Processor 1). If a clone finishes earlier than the other clones in the workload, we restart it. We run the experiment until all clones execute at least once. We report data about the first execution of each clone.

Figure 2.21 shows the average of the number of misses per thousand instructions (MPKI) of all `soplex` clones relative to the solo mode MPKI of `soplex` (when the clone is executed alone in the system). Remember that the MPKI of a program increases if in its execution the program contends for LLC capacity with other programs using the same LLC. When the data locality is maximal in the system (100% of the references are local because all clones execute locally on Processor 0), the average increase of MPKI peaks at 35% because all clones execute on the same processor and thus use the same LLC. When the data locality in the system is 57% (two clones execute on Processor 0 and two clones execute on Processor 1), cache contention is minimal as the MPKI increase is also minimal (19%).

In the mapping configuration with 57% data locality two locally executing clones obtain a higher fraction of the total memory bandwidth (57%) than two remotely executing clones (43%). Memory bandwidth is not equally partitioned between local and remote clones because remote clones encounter a higher memory access latency than local clones. As a result remote clones can issue memory request at a lower rate than local clones.

Figure 2.22 shows the slowdown of the locally respectively remotely executing `soplex` clones. The slowdown is calculated relative to the solo mode execution of `soplex`. We also plot the average degradation of the clones. Clearly, neither the mapping with minimum cache contention, nor the mapping with maximum data locality performs best. The average slowdown (and also the individual slowdown) of the clones is smallest if there is 80% data locality in the system (i.e., in the mapping configuration with three locally and one remotely executing clone).

We perform a similar experiment on a 2-processor 12-core machine (the Westmere-based machine described in Section 2.1.7). In this experiment we use a workload composed of 6 `soplex` clones. Similar to the Nehalem-based experiments, we vary the number of clones executed locally and remotely. The memory of all clones is allocated at Processor 0. As there are 6 cores on each processor, the mapping configurations range from all 6 clones executed locally to all clones executed remotely.

Figure 2.23 shows the average MPKI of all clones relative to the solo mode MPKI of `soplex`. Similar to the Nehalem case, cache contention is the lowest when there is 56% data locality in the system, moreover, when data locality is best (100% local accesses), cache contention is highest.

Figure 2.24 shows the average slowdown of clones, as well as the individual slowdown of local and remote clones. The average slowdown is similar in the configuration with 56% data locality (and minimum cache contention) and in the configuration with 69% data locality. However, the slowdown of local and remote clones is around the same in the 69% case, while in the 56% case remote clones experience more slowdown than local clones. Thus, the optimal mapping (with 69% data locality) is between the mapping with the lowest cache contention and the mapping with the best data locality.

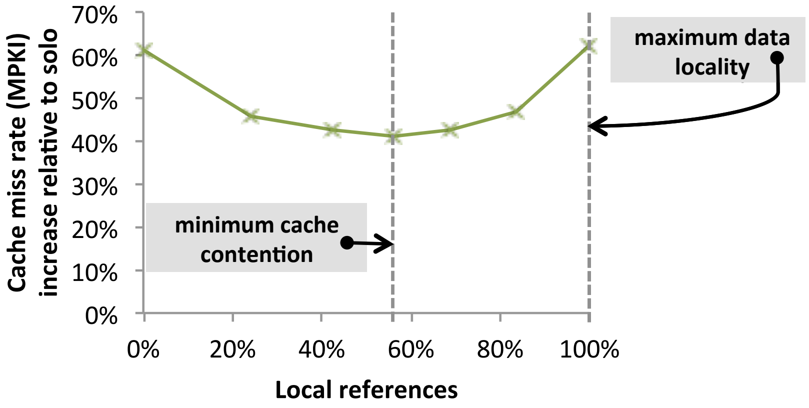


Figure 2.23: Increase of cache miss rate vs. data locality of `soplex` (Westmere).

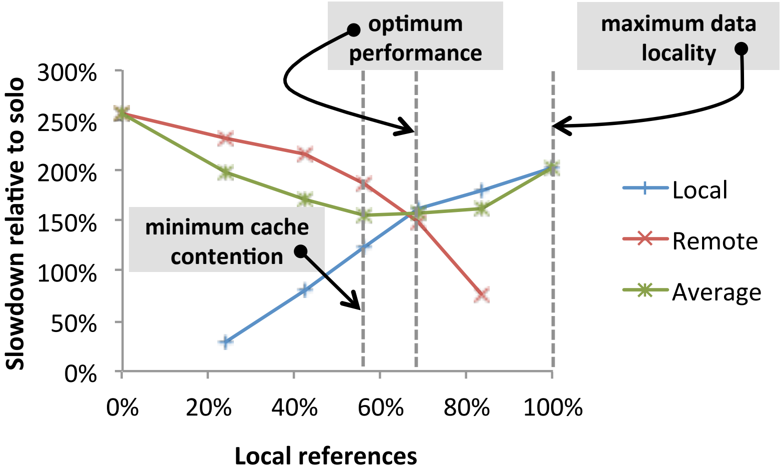


Figure 2.24: Performance vs. data locality of `soplex` (Westmere).

2.2.3 Summary

In NUMA multicore-multiprocessor systems, good data locality is often beneficial for performance because local memory accesses have better memory access latency and bandwidth than remote memory accesses [44, 73]. The experimental analysis in this section (Section 2.2) has shown, however, that in scenarios when the load on LLCs is high, avoiding cache contention can improve performance, even at the cost of compromising data locality. In other words, trading local memory accesses for a combination of cache- and remote memory accesses can improve performance; software must be aware of this tradeoff to be able to efficiently use the memory system.

The results presented in this section (Section 2.2) strongly depend on memory system parameters (e.g., LLC size and memory access latencies) that are specific to our Nehalem- and Westmere-based machines. On other machines (with possibly different parameters) the results may differ. For example, on a system with larger LLCs, cache contention might not cause significant performance degradations; moreover, on systems with more processors and more elaborate interconnects the penalty of remote memory accesses might be prohibitively high to allow favoring remote execution over local execution. Nevertheless, the analysis presented in this section provides insight on (and also compares the cost of) two performance-degrading factors that effect programs on current NUMA multicore-multiprocessors, cache contention and interconnect overhead. Moreover, the experiments can be easily repeated and be used to characterize any NUMA multicore-multiprocessor, including newer generations of NUMA-multicore systems.

2.3 Conclusions

A typical NUMA multicore-multiprocessor memory system has several bottlenecks. This chapter analyzes two types of bottlenecks: (1) the sharing of memory system resources (e.g., memory controllers and caches), and (2) interconnect overhead. Both types of bottlenecks have significant effect on application performance.

The performance of multiprogrammed workloads ultimately depends on the mapping of the workload onto the hardware (i.e., the distribution of the workload's data and the schedule of the workload's computations). It depends on the mapping which bottlenecks of the memory system influence the performance of an application, if at all. More importantly, however, in some cases bottlenecks can impose limitations *at the same time*. Thus, a mapping must consider *all* bottlenecks of a memory system to obtain good performance.

3

Cache-conscious scheduling with data locality constraints

The memory system performance of a multiprogrammed workload critically depends on the way the workload is mapped onto the hardware. On NUMA multicore-multiprocessors a mapping has two components: (1) the *distribution of data* (i.e., the way the application's data is distributed across the processors of the system), and (2) the *schedule of computations* (i.e., the way the computations of an application are scheduled onto the cores/processors of the system).

In this chapter we focus on the second component of a mapping, that is, on scheduling computations to cores/processors. (We assume that the distribution of data in the system is given, that is, data cannot be migrated.) As we consider multiprogrammed workloads, determining the schedule of computations is equivalent to deciding on how a set of independent operating system processes are mapped onto the cores/processors of a system.

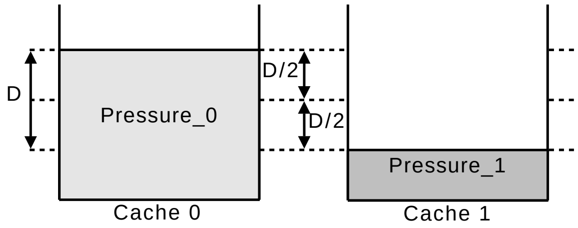
This chapter presents the design (Section 3.1), implementation (Section 3.2), and evaluation (Section 3.3) of N-MASS, a process scheduler tailored for NUMA-multicore systems. Ideally, a process scheduler balances between *all* bottlenecks of a NUMA-multicore memory system. N-MASS takes two bottlenecks into account, contention for shared last-level caches (LLCs) and interconnect overhead, and targets finding a balance these two bottlenecks.

To find a mapping that uses the memory system appropriately, a process scheduler requires information about the interaction between the hardware and the scheduled processes (e.g., information about a process's utilization of shared caches, about the performance penalty the process experiences due to remote memory accesses, as well as about the locality of the process's main memory accesses). As some of these information is not yet available in current systems, we simulate their availability by using static traces of program memory behavior. In Section 3.4 we elaborate on possibilities for future extension that could make our approach more practical.

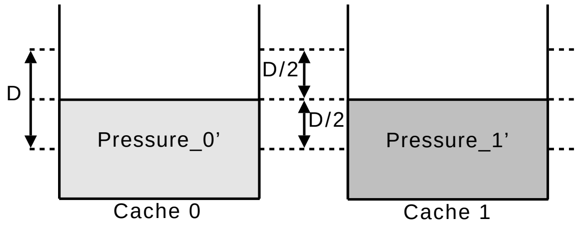
3.1 Design

3.1.1 Modeling memory system behavior

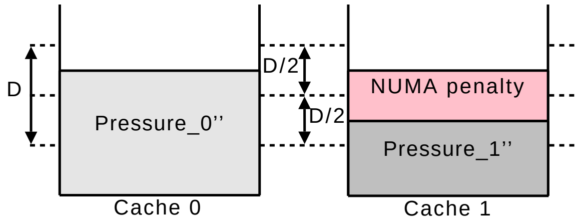
As shown in Chapter 2, process scheduling on NUMA-multicores must target a tradeoff between improving data locality and reducing cache contention (the optimum performance point on Figures 2.22 and 2.24). To simplify the discussion, we focus on a setup with two processors. We also assume that all the cores of a processor share an LLC. There are systems that do not support this assumption (see multi-socket implementations like the AMD Magny-Cours). In



(a) Initial configuration.



(b) SMP.



(c) NUMA.

Figure 3.1: Cache balancing in SMP and NUMA context.

this case you should consider all the cores that share a cache to form a "processor".

To reduce cache contention in a system, a process scheduler must characterize the way processes utilize the shared caches of the system; that is, a process scheduler must characterize both the *contentiousness* and *sensitivity* of each scheduled process [99, 113]. A process's contentiousness is a measure of how much performance degradation the process causes to other processes with which it shares a cache. A process's sensitivity is a measure of the performance degradation a process suffers due to sharing the cache with other processes.

There are several approaches that enable a process scheduler to determine (or to estimate) the contentiousness and the sensitivity of a process [113]. In practice, however, many contention-aware process scheduler implementations use a single measure, the *cache pressure*, to estimate both the contentiousness and sensitivity of a process [13, 83, 113]. A process's cache pressure is, by definition, the ratio of the number of cache misses encountered by the process and the number of instructions executed by the process. In current systems with a shared LLC a process's cache pressure is often defined as the number of LLC misses encountered by the process divided by the number of instructions executed by the process. The pressure on a shared cache is, by definition, as the sum of the cache pressure of each process using that cache.

In recent systems, both a process's LLC misses and the number of instructions executed by a process can often be measured using the processor's performance monitoring unit (PMU) [52]. The low cost of hardware performance counter measurements allow a process scheduler to characterize a process's shared cache utilization at runtime with low overhead. Other synthetic metrics like stack-distance profiles [20, 90] or miss-rate curves [97] can estimate shared cache utilization with better precision, but generating these metrics might result in higher runtime overhead than PMU-based measurements. Therefore, we use the cache pressure metric in our work to characterize the way processes utilize shared caches.

Our approach builds upon the idea of scheduling algorithms for non-NUMA multicore-multiprocessors (i.e., symmetric multiprocessors (SMPs) built with multicore chips) with shared caches [52, 113]. The basic principle of these algorithms is illustrated in Figure 3.1 (for a system with two LLCs). If the difference D between the pressure on the two caches of the system is large (Figure 3.1(a)), some processes (preferably processes with a cumulative cache pressure of $D/2$) are scheduled onto the cache with the smaller pressure. As a result, the difference between the pressure on the two caches is reduced (Figure 3.1(b)).

Our approach is similar to cache-balancing algorithms in SMPs and relies on two principles. First, we also distribute pressure across caches, however not evenly as in an SMP system: The amount of cache pressure transferred to Cache 1 is less than $D/2$ (half of the difference) – as illustrated in Figure 3.1(c). If mapping processes onto a different LLC results in the remote execution of the re-mapped process, then we account also for the performance penalty of remote execution (i.e., the NUMA penalty previously defined in Section 2.1). The performance degradation experienced by processes is equal if this penalty is also considered. The second principle of our approach states that overloading the cross-chip interconnect with too many remotely executing processes must be avoided. Therefore, if the pressure on the remote cache is above a threshold, we do not re-map processes for remote execution.

Section 3.1.2 discusses a practical way to characterize a program's memory behavior (i.e., estimating the cache pressure and the NUMA penalty). Then, Section 3.1.3 presents the N-MASS algorithm that implements the previously discussed two principles.

3.1.2 Characterizing the memory behavior of processes

A scheduling algorithm targeting memory system optimizations must be able to quickly estimate the memory behavior of scheduled processes. Three parameters are necessary to characterize the memory behavior of a process on a NUMA multicore-multiprocessor: (1) the cache pressure of the process, (2) the remote execution penalty the process experiences (i.e., the NUMA penalty), and (3) the distribution of the process's data.

Cache pressure On many modern processors the cache pressure of process can be measured using the processor's PMU. In the Intel Nehalem-based system that we use, however, it is possible to measure only the LLC read misses of a process (see Section 2.1.1 for more details). Therefore, we estimate a program's cache pressure by the read misses per thousand instructions executed by the process (MPKI).

NUMA penalty As defined in Section 2.1, the NUMA penalty quantifies the slowdown of a remote execution of a process relative to the process's local execution. Let CPI_{local} denote the CPI (cycles per instruction) of a process executing locally, and let CPI_{remote} denote the CPI of the same process executing remotely. Using this notation the NUMA penalty is:

$$\text{NUMA penalty} = CPI_{remote}/CPI_{local} \tag{3.1}$$

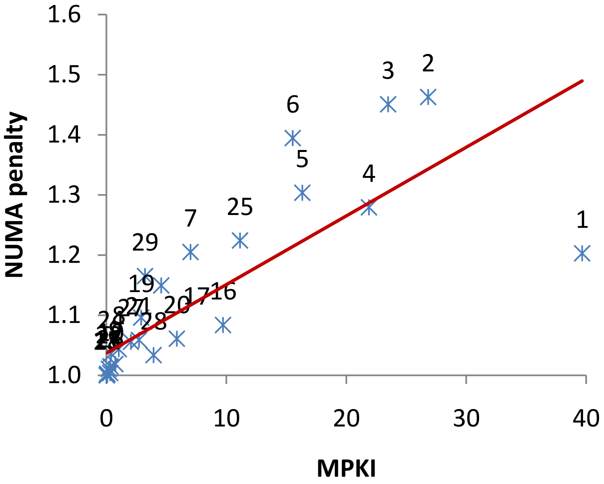


Figure 3.2: NUMA penalty vs. MPKI.

The NUMA penalty is a lower-is-better metric and its minimum value is 1 (if a process does not slow down in its remote execution). For example, if a process has a NUMA penalty equal to 1.3, the process slows down 30% on remote execution (i.e., the process's remote execution takes 30% more time than its local execution). We measure the NUMA penalty of a process by executing the process twice, once locally and once remotely. During the measurements all cores are inactive, except the core that executes the process. Figure 3.2 plots the NUMA penalty of all programs of the SPEC CPU2006 benchmark suite against their MPKI (we use the SPEC programs for evaluating the N-MASS scheduler). All programs in the SPEC CPU2006 suite are sequential, therefore each program maps to a single OS process.

Table 3.1 shows the complete data. The numbers on Figure 3.2 correspond to the numbers in this table. For comparison, the table lists the MPKI and NUMA penalty of the sequential version of STREAM `triad` as well.

Figure 3.2 also plots a linear model fitted onto the data. (The `triad` benchmark is not shown in the figure, nor is it included in the linear model.) Although the two parameters are positively correlated (the NUMA penalty increases with the MPKI), the coefficient of determination (R^2) is relatively low, 0.64. However, as today's hardware does not provide information on the NUMA penalty experienced by a process, this regression model can be a simple (and fast) way to estimate a process's NUMA penalty based on its MPKI.

Data distribution To simplify the discussion, we assume that all the data of a process is allocated to one processor. This assumption includes scenarios when co-executing processes have their memory allocated on specific, possibly different, processors; that is, we assume only that a single process's memory is not scattered around in the system. We also assume that the home processor of a process (i.e, the target of the most of the process's memory accesses) cannot be changed. That is, we assume a process's data is not migrated, neither by the process scheduler, nor by the underlying OS. In case a process's data is scattered around in the system, the home processor of the process must be determined at runtime. In most current systems this information is not easy to obtain, we discuss this limitation in Section 3.5.

#	Program	MPKI	NUMA penalty	Type (C: compute-bound, M: memory-bound)
1	mcf	39.67	1.20	M
2	milc	26.82	1.46	M
3	libquantum	23.49	1.45	M
4	soplex	21.89	1.28	M
5	lbm	16.34	1.30	M
6	omnetpp	15.54	1.39	M
7	gcc	7.01	1.21	M
8	sphinx	1.04	1.04	C
9	gobmk	0.74	1.02	C
10	perlbench	0.32	1.01	C
11	namd	0.05	1.00	C
12	h264	0.03	1.00	C
13	gamess	0.00	1.00	C
14	povray	0.00	1.00	C
15	bzip2	0.07	1.00	C
16	bwaves	9.72	1.08	M
17	zeusmp	4.57	1.15	C
18	gromacs	0.07	1.00	C
19	cactusADM	2.90	1.10	C
20	leslie3d	5.87	1.06	C
21	dealII	2.72	1.06	C
22	calculix	0.21	1.01	C
23	hmmer	0.03	1.00	C
24	sjeng	0.38	1.04	C
25	gemsFDTD	11.14	1.22	M
26	tonto	0.32	1.00	C
27	astar	2.05	1.06	C
28	wrf	3.93	1.03	C
29	xalanbmk	3.22	1.17	C
30	STREAM triad	53.48	1.58	M

Table 3.1: MPKI and NUMA penalty of SPEC benchmarks and STREAM `triad`.

3.1.3 The N-MASS algorithm

The **N**UMA–**M**ulticore-**A**ware **S**cheduling **S**cheme (N-MASS) implements the two principles of cache-aware scheduling in NUMA systems described in Section 3.1.1. Algorithm 1 presents an outline of N-MASS. The algorithm is described for a 2-processor NUMA system, but it can be extended to handle a higher number of processors as well. The algorithm is invoked after a scheduler epoch has elapsed, and it calculates the mapping $M_{final} : \text{Processes} \mapsto \text{Cores}$ of processes onto cores. The number of scheduled processes n equals at most the number of cores in the system (this limitation is discussed later in this section). The algorithm uses the following performance data about each scheduled process i: the process's *cache pressure* ($mpki_i$) and

Algorithm 1 N-MASS: maps n processes onto a 2-processor NUMA-multicore system.

Input: List of processes P_0 and P_1 homed on Processor 0 respectively Processor 1.

Output: A mapping M_{final} of processes to processor cores.

```
1: // Step 1: Sort list of processes by NUMA penalty
2: $P_{sorted_0} \leftarrow sort\_descending\_by\_np(P_0)$
3: $P_{sorted_1} \leftarrow sort\_descending\_by\_np(P_1)$
4: // Step 2: Calculate maximum-local mapping
5: $M_{maxlocal} \leftarrow map\_maxlocal(P_{sorted_0}, P_{sorted_1})$
6: // Step 3: Refine maximum-local mapping
7: $M_{final} \leftarrow refine\_mapping(M_{maxlocal})$
```

Algorithm 2 map_maxlocal: maps n processes onto a 2-processor system NUMA system; processes are mapped onto the processor that holds their data in descending order of their NUMA penalty.

Input: List of processes P_{sorted_0} and P_{sorted_1} homed on Processor 0 respectively Processor 1. The lists are sorted in descending order of the processes' NUMA penalty (np).

Output: A mapping $M_{maxlocal}$ of processes to processor cores.

```
$M_{maxlocal} \leftarrow \emptyset$
$p_0 \leftarrow pop\_front(P_{sorted_0}); p_1 \leftarrow pop\_front(P_{sorted_1})$
while $p_0 \neq NULL$ or $p_1 \neq NULL$ do
    if $p_1 = NULL$ or $np_{p_0} > np_{p_1}$ then
        $core \leftarrow get\_next\_available\_core$(Processor 0)
        $push\_back(M_{maxlocal}, (p_0, core))$
        $p_0 \leftarrow pop\_front(P_{sorted_0})$
    else if $p_0 = NULL$ or $np_{p_0} \leq np_{p_1}$ then
        $core \leftarrow get\_next\_available\_core$(Processor 1)
        $push\_back(M_{maxlocal}, (p_1, core))$
        $p_1 \leftarrow pop\_front(P_{sorted_1})$
    end if
end while
```

an estimate of the process's *NUMA penalty* (np_i). The N-MASS algorithm has three steps. First, for each processor, it sorts the list of processes homed on the processor in descending order of the processes' NUMA penalty (lines 2-3). Second, it maps the processes onto the system using the *maximum-local* policy (line 5). If the pressure on the memory system of the two-processor system is unbalanced, then, in the third step, the algorithm refines the mapping decision produced by the maximum-local mapping (line 7). In the following paragraphs we describe Step 2 and Step 3 of N-MASS.

Step 2: Maximum-local mapping The maximum-local scheme (described in detail by Algorithm 2) improves data locality in the system by mapping processes onto their home nodes in descending order of their NUMA penalty. The algorithm has as its input two lists of processes, P_0 and P_1. The processes in list P_0 (P_1) are homed on Processor 0 (Processor 1). The lists are sorted in descending order of the NUMA penalty of the processes they contain. The algorithm merges the two lists. During the merge, the algorithm determines which core each process is mapped onto. The algorithm guarantees that processes with a high NUMA penalty are mapped

Algorithm 3 refine_mapping: refines the maximum-local mapping of n processes to reduce cache contention.

```
Input: Maximum-local mapping of processes $M_{maxlocal}$. For each process $i$ the NUMA penalty
    respectively the MPKI of the last scheduler epoch is available in $np_i$ respectively $mpki_i$.
Output: A mapping $M_{final}$ of processes to processor cores.
$M_0 = \{(p, core) \in M_{maxlocal} \mid core \in \text{Processor } 0\}$
$M_1 = \{(p, core) \in M_{maxlocal} \mid core \in \text{Processor } 1\}$
$pressure_0 = \sum\{mpki_p \mid (p, core) \in M_0\}$
$pressure_1 = \sum\{mpki_p \mid (p, core) \in M_1\}$
repeat
    $\Delta \leftarrow |pressure_1 - pressure_0|$
    $(p_0, core_0) \leftarrow back(M_0); (p_1, core_1) \leftarrow back(M_1)$
    $\Delta_{\text{MOVE}_{0\to1}} \leftarrow mpki_{p_0} \cdot np_{p_0}$
    $\Delta_{\text{MOVE}_{1\to0}} \leftarrow mpki_{p_1} \cdot np_{p_1}$
    if $\Delta_{\text{MOVE}_{0\to1}} < \Delta_{\text{MOVE}_{1\to0}}$ then
        $pressure_0 \leftarrow pressure_0 - mpki_{p_0}$
        $pressure_1 \leftarrow pressure_1 + mpki_{p_0} \cdot np_{p_0}$
        $core \leftarrow get\_next\_available\_core(\text{Processor } 1)$
        // Could be on Processor 0 if $\nexists$ free core on Processor 1
        if $core \notin \text{Processor } 0$
        and $pressure_1 < \text{THRESHOLD}$ then
            $pop\_back(M_0, (p_0, core_0))$
            $push\_front(M_1, (p_0, core))$
            $decision \leftarrow \text{MOVE}_{0\to1}$
        else
            $decision \leftarrow \text{CURRENT}$
        end if
    end if
    if $\Delta_{\text{MOVE}_{0\to1}} \geq \Delta_{\text{MOVE}_{1\to0}}$
    or $decision = \text{CURRENT}$ then
        // Similar to the $\text{MOVE}_{0\to1}$ case
    end if
until $decision \neq \text{CURRENT}$
$M_{final} \leftarrow M_0 \cup M_1$
```

onto a core of their home node with higher priority than processes with a lower NUMA penalty that are homed on the same processor. The lists P_0 and P_1, and the mapping $M_{maxlocal}$ of processes, are double-ended queues. The function $pop\_front(l)$ removes the element from the front of the list l; the function $push\_back(l, e)$ inserts element e at the back of the list l. The function $get\_next\_available\_core(p)$ returns the next free core, preferably from processor p. If there are no free cores on processor p, the function returns a free core from a different processor.

Step 3: Cache-aware refinement If the maximum-local mapping results in increased contention on the caches of the system, Step 3 of the N-MASS algorithm refines the mapping produced by the maximum-local scheme in Step 2. This step implements the two principles of scheduling in NUMA-multicores previously discussed in Section 3.1.1, and is described in detail in Algorithm 3. First, the algorithm accounts for the performance penalty of remote execution by multiplying the MPKI of remotely mapped processes with their respective NUMA penalty (lines 8, 9, 12). Second, the algorithm avoids overloading the cross-chip interconnect by moving processes only if the pressure on the remote cache is less than a predefined threshold (line 14). We discuss in Section 3.2 how the threshold is determined. By construction (line 6 and 10 of Algorithm 2) $M_{maxlocal}$ contains pairs $(process, core)$ ordered in descending order of the processes' NUMA penalty. The function $back(l)$ returns the last element of list l without removing it from the list; $push\_front(l, e)$ inserts element e to the front of list l.

Limitations The N-MASS algorithm requires the number of processes n to be at most the total number of cores on the system, therefore it can only decide on the spatial multiplexing of processes (and not on their temporal multiplexing). Nevertheless, if the OS scheduler decides on the temporal multiplexing (the set of processes that will be executed in the next scheduler epoch), N-MASS can refine this mapping so that the memory allocation setup in the system is accounted for, and the memory system is efficiently used.

3.2 Implementation

To demonstrate that N-MASS is capable of finding the tradeoff between reducing cache contention and increasing data locality, we implemented a prototype version of N-MASS. N-MASS takes a set of programs and then executes these programs. The current implementation of N-MASS supports only sequential programs. As all programs are sequential, each program maps to a single OS process. N-MASS periodically gathers information about the memory behavior of each process and then, based on the processes' memory behavior, it decides on the way processes are mapped onto the hardware.

As mentioned in Section 3.1.2, a process's memory behavior is estimated based on three parameters: the process's MPKI, its NUMA penalty, and its home node. We simulate the availability of "perfect" information about a program's memory behavior: We obtain the MPKI and NUMA penalty of a process (more precisely: the MPKI and NUMA penalty of the program executed by the process) from a trace generated on a separate profiling run (with the program run in solo mode). We use programs of the SPEC CPU2006 benchmark suite; a program trace contain samples of the program's memory behavior. Each sample contains information about the MPKI and NUMA penalty of 2.26 million of instructions of the program's lifetime. In Section 3.4 we also evaluate N-MASS with an on-line measurement of the MPKI and and on-line estimation of the NUMA penalty based on the MPKI. We assume, furthermore, that a process's data is allocated at a single processor (each process's home node is given as parameter to N-MASS that takes care of data placement).

N-MASS is designed to adapt to program phase changes. Each sample read by N-MASS includes the MPKI of a process. The scheduler is invoked if a process's MPKI changes by more than 20% relative to the previous scheduler epoch. The costs of re-schedules (moving a process to a different core) can be high. We select an epoch length of 1 second. This epoch length almost completely eliminates the costs of re-schedules, while the scheduler is still able to quickly react to program phase changes. We select 60 MPKI for the threshold used by the refinement step of the N-MASS algorithm (line 14 of Algorithm 3). We base our selection on the experimental analysis of the Intel Nehalem memory system (Chapter 2). `triad` is the most memory-bound program we have encountered (it has the highest MPKI). The MPKI of a `triad` instance is around 53 (see Table 3.1). We want to allow the remote execution of a number of processes (processes that have a cumulative cache pressure that is approximately the same as that of a single `triad` instance), but we want to disallow the remote execution of processes with a cumulative cache pressure larger than 60 MPKI in order not to saturate the QPI interconnect of our Nehalem-based system (as shown in Chapter 2, two `triad` instances saturate the cross-chip interconnect).

The performance measurements of N-MASS include the overhead of performance monitoring, as we sample the number of instructions executed by each process to keep track of the process's execution in the trace file. We also record the number of elapsed processor cycles to measure performance. N-MASS uses the perfmon2 [30] performance monitoring library to gather hardware performance counter information.

3.3 Evaluation

3.3.1 Experimental setup

We use the 2-processor 8-core system based on the Intel Nehalem microarchitecture for evaluation. Detailed information about the machine's hardware- and software configuration can be found in Section 2.1.1.

We are interested in multiprocessor performance, therefore we construct multiprogrammed workloads with the programs of the SPEC CPU2006 benchmark suite. Our evaluation methodology is very similar to the methodology used in [8, 14, 33, 34, 52, 113]. We use a subset of the SPEC CPU2006 benchmark suite (14 programs out of the total 29 in the suite). Our selection includes programs 1–14 in Table 3.1. The MPKI of a program (the second column in Table 3.1) estimates the cache pressure of the program. The selection we use includes both compute-bound programs (programs with an MPKI less than 7.0) and memory-bound programs (programs with an MPKI greater or equal to 7.0). (Compute-bound programs are marked with 'C' in the last column of the table, memory-bound programs are marked with 'M' in the table.) The programs in the subset have a broad range of NUMA penalties (between 1.0 and 1.46).

The subset the SPEC CPU2006 suite we use allows us to construct a number of different multiprogrammed workloads (different in both terms of memory boundedness, i.e., total cache pressure, and in terms of the memory behavior of constituent programs). Therefore, the subset is large enough for a meaningful performance evaluation, yet sufficiently small to allow for a concise presentation of experimental data.

Similarly to [52], we run each multiprogrammed workload exactly one hour. If a program terminates before the other programs in a workload do, we restart the program that terminated early. We use the reference data set and follow the guidelines described in [75] to minimize measurement variance. This setup usually gives us three measurable runs for each workload within the one hour limit. For each run we report the average slowdown of each constituent program relative to its solo mode performance. We also report the average slowdown of the whole workload, as suggested by Eyerman et al. [31].

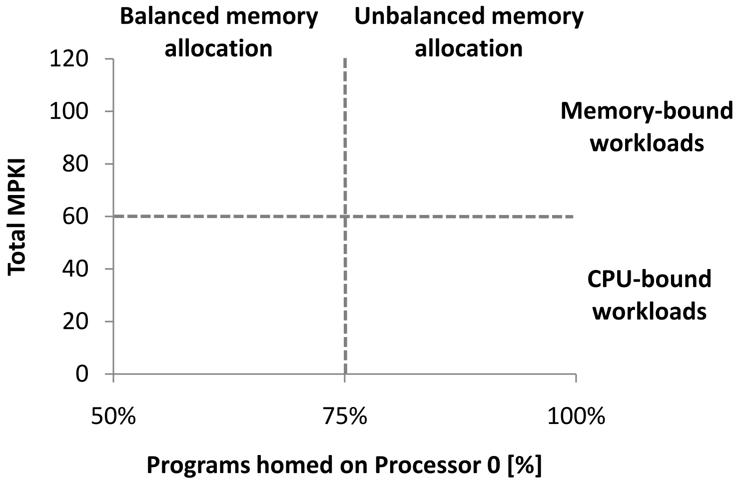


Figure 3.3: Dimensions of the evaluation.

3.3.2 Dimensions of the evaluation

There are two dimensions that must be considered to evaluate the interaction between memory allocation and process scheduling in a NUMA-multicore system. The two dimensions (shown in Figure 3.3) are (1) the *memory boundedness* of workloads (i.e., the total cache pressure of workloads shown on the y-axis) and (2) the *balance of memory allocation* in the system (shown on the x-axis).

Memory boundedness To show that N-MASS can handle workloads with different memory-boundedness, we use 11 different multiprogrammed workloads (WL1 to WL11). This setup corresponds to evaluating N-MASS along the first dimension (the y-axis in Figure 3.3). The workloads are composed of different number of compute-bound (C) respectively memory-bound (M) programs. The memory-boundedness of a workload is characterized by the sum of the MPKIs of its constituent programs (measured in solo mode for each program). The total MPKI of each multiprogrammed workload we use is shown in Figure 3.4. The composition of the multiprogrammed workloads is shown in Table 3.2. The workloads in the set of 4-process workloads (WL1 to WL9) contain one to four memory-bound programs. The 8-process workloads (WL10 and WL11) contain three, respectively four, memory-bound programs. In the case of all 11 workloads we add CPU-bound programs so that at the end there are four (respectively eight) programs in total in each workload.

Balance of memory allocation setup As the performance of process scheduling closely depends on the memory allocation setup in the system, for each workload we consider several ways memory is allocated in the 2-processor evaluation machine. The second dimension of our evaluation (the x-axis in Figure 3.3) is the percentage of the processes of a multiprogrammed workload homed on Processor 0 of the system. (Ideally, we would like to vary the percentage of memory references to local respectively remote memory, but as we can map only complete processes, we vary along this dimension by mapping processes.) The left extreme point of the x-axis represents the configuration with balanced memory allocation (50% of the processes homed on Processor 0). On the other end of the x-axis we find the most unbalanced configura-

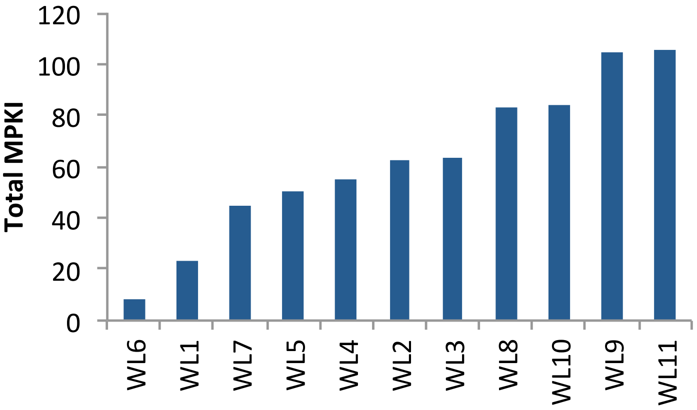


Figure 3.4: Total MPKI of multiprogrammed workloads.

#	Programs				Type
1	soplex	sphinx	gamess	namd	1M, 3C
2	soplex	mcf	gamess	gobmk	2M, 2C
3	mcf	libquantum	povray	gamess	2M, 2C
4	mcf	omnetpp	h264	namd	2M, 2C
5	milc	libquantum	povray	perlbench	2M, 2C
6	sphinx	gcc	namd	gamess	1M, 3C
7	lbm	milc	sphinx	gobmk	2M, 2C
8	lbm	milc	mcf	namd	3M, 1C
9	mcf	milc	soplex	lbm	4M
10	lbm gobmk	milc perlbench	mcf h264	namd povray	3M, 5C
11	mcf gobmk	milc perlbench	soplex namd	lbm povray	4M, 4C

Table 3.2: Multiprogrammed workloads.

tion (100% of the processes homed on Processor 0). Because of the symmetries of the system there is no need to extend the range to the case with 0% of the processes' memory allocated on Processor 0. This corresponds to 100% of the processes homed on Processor 1, which is equivalent to all processes homed on Processor 0.

In Section 3.3.3 we evaluate N-MASS along both dimensions: the evaluation considers workloads with different memory boundedness as well as different memory allocation setups. Then, in Section 3.3.4, we evaluate N-MASS with an unbalanced memory allocation setup.

3.3.3 Influence of data locality and cache contention

The second dimension of our evaluation is defined as the percentage of the processes homed on Processor 0 of the system. This percentage, however, does not specify *which* constituent processes of a multiprogrammed workload are homed on each processor in the system. We define the concept of *allocation maps*. An allocation map is a sequence $M = (m_0, m_1, \ldots, m_n)$, where n is the number of processes in the workload executing on the system, and

$$m_i = \begin{cases} 0, & \text{if the } i^{th} \text{ process is homed on } \mathrm{Processor}\ 0; \\ 1, & \text{if the } i^{th} \text{ process is homed on } \mathrm{Processor}\ 1. \end{cases} \tag{3.2}$$

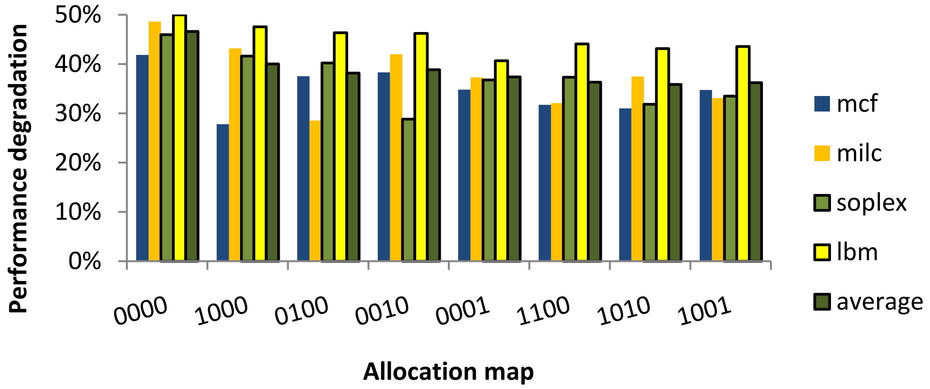
There are $\sum_{i=0}^{4} \binom{4}{i} = 2^4 = 16$ ways to allocate memory for a 4-process workload on the Nehalem system (assuming each program's memory is allocated entirely on one of the two processors of the system). Because of the symmetries of the system, however, the number of combinations is reduced to 8. These allocation maps are shown in Table 3.3. For example, if 50% of the processes are homed on Processor 0, we must consider three different possibilities. If we look at the performance of a mapping algorithm with a multiprogrammed workload that has a composition (M, M, C, C) (the first two processes are memory-bound, the last two compute-bound), then the 50%-allocation maps 1100 and 1010 are different from the point of view of the maximum-local scheduling scheme. Remember that the maximum-local scheme maps processes onto their home nodes if possible. In the case of the 1100 allocation map, maximum-local maps the two memory-bound processes onto the same processor, therefore onto the *same* LLC. This setup results in good data locality but also produces high cache contention. In the case of the 1010 allocation map maximum-local maps the memory-bound processes onto *separate* LLCs. Therefore, the maximum-local policy maximizes data locality and minimizes cache contention in this case.

Processes homed on Processor 0	Allocation maps
50%	1100, 1010, 1001
75%	1000, 0100, 0010, 0001
100%	0000

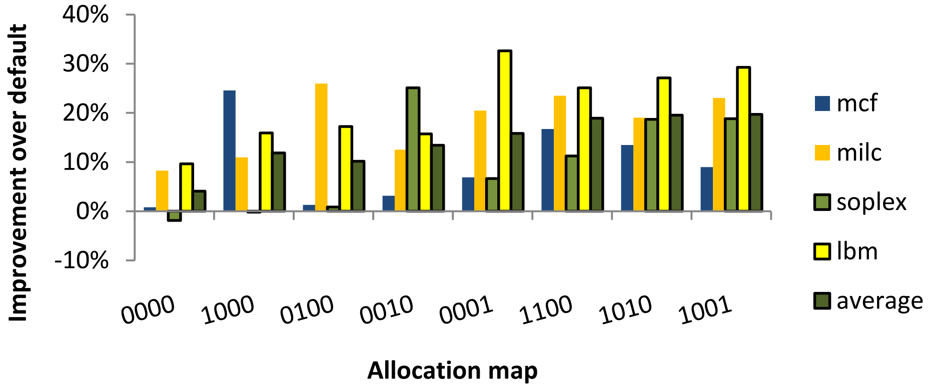
Table 3.3: Allocation maps for 4-process workloads.

For clarity of presentation we use two workloads from opposite ends of the memory-boundedness spectrum to evaluate the performance of N-MASS with different allocation maps: the compute-bound WL1 and the memory-bound WL9. We compare the performance of three mapping schemes: *default*, *maximum-local* and *N-MASS*. If not stated otherwise, in our evaluation *N-MASS* denotes the version of the algorithm that has "perfect" information about the NUMA penalty of the programs from profile-based program traces (in Section 3.4 we also evaluate N-MASS with an on-line measurement of the MPKI and and on-line estimation of the NUMA penalty based on the MPKI). The *maximum-local* policy is similar to N-MASS, except it does not include the cache-aware refinement step of N-MASS (Step 3 of Algorithm 1). We evaluate this scheme to quantify the improvement of the cache-aware refinement step over maximum-local mapping.

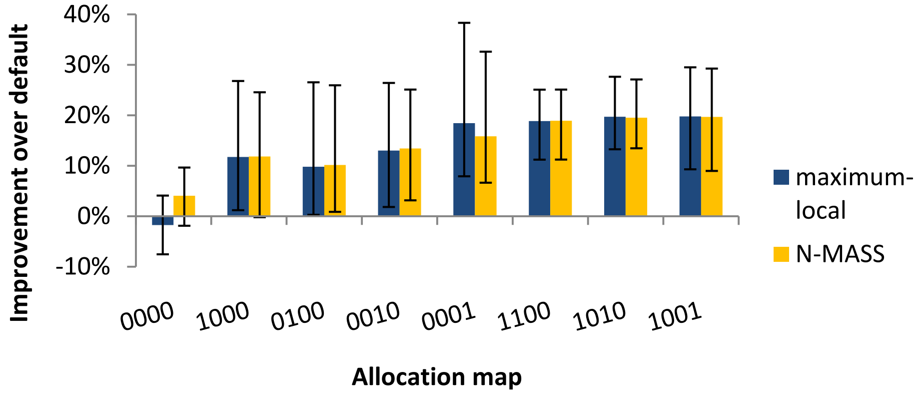
The performance of multiprogrammed workloads varies largely with the default Linux scheduler, and simple factors like the order in which workloads are started influence the performance readings. Because operating system schedulers (including the Linux scheduler) balance only the CPU load and do not account for data locality or cache contention, processes might be mapped so that they use the memory system in the most inefficient way possible. To avoid measurement bias, we account for all schedules that an OS scheduler that balances CPU load



(a) Performance degradation with default scheduling.



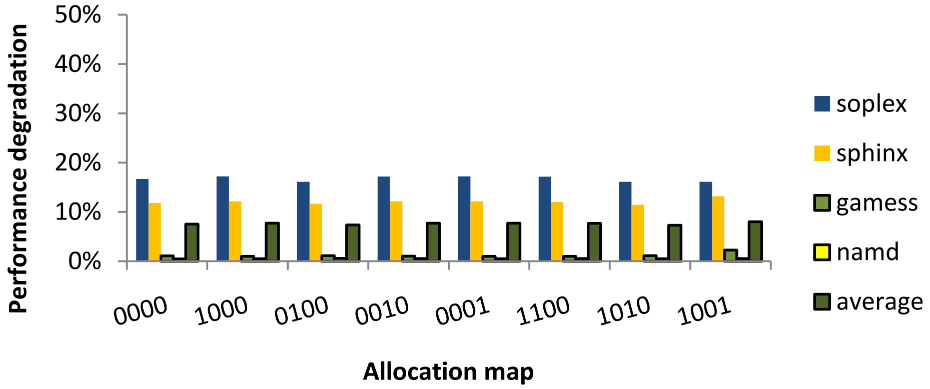
(b) N-MASS improvement over default scheduling.



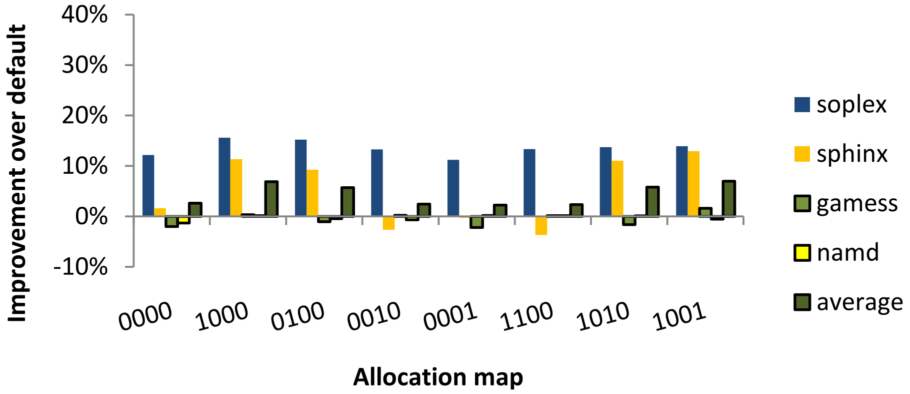
(c) N-MASS compared to *maximum-local*. (Error bars: max/min performance improvement recorded for any constituent program of a workload.)

Figure 3.5: Performance evaluation of the *maximum-local* and N-MASS schemes with WL9.

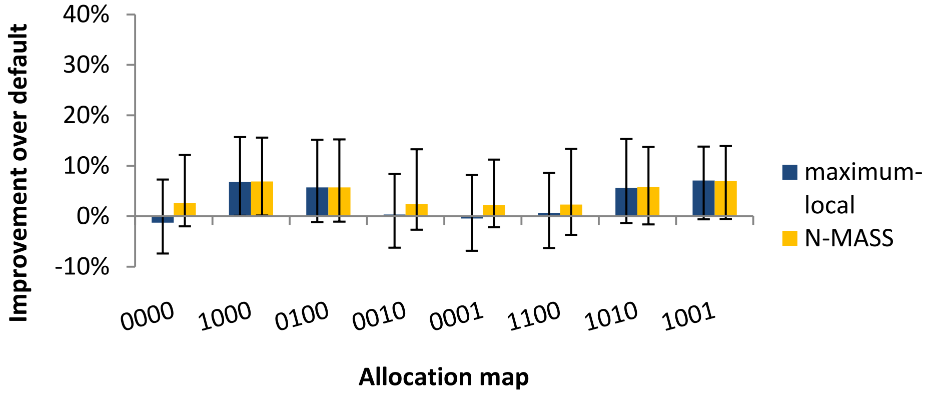
would consider. For example, in the case of 4-process workloads the default Linux scheduler always maps two processes onto each processor so that each processor is allocated half the total CPU load. For each 4-process workload there are $\binom{4}{2} = 6$ equally probable different schedules with the CPU load evenly distributed in the system. Running a single workload in all these schedules takes 6 hours execution time with our evaluation methodology, which is tolerable. Therefore, for each multiprogrammed workload we run the workload in each schedule possible for the default scheduler, and then we report the average performance degradation of the



(a) Performance degradation with default scheduling.



(b) N-MASS improvement over default scheduling.



(c) N-MASS compared to *maximum-local*. (Error bars: max/min performance improvement recorded for any constituent program of a workload.)

Figure 3.6: Performance evaluation of the *maximum-local* and N-MASS schemes with WL1.

multiprogrammed workload in all schedules as the performance of *default* scheduling.

Figure 3.5(a) (respectively Figure 3.6(a)) shows the performance degradation of the programs of WL9 (WL1) with the default scheduler. (The degradations are calculated relative to the solo mode performance of the programs.) WL9 is composed of more memory-bound programs than WL1, therefore the degradations experienced by WL9 programs are higher (up to 50% vs. 18%). 3.5(b) (respectively 3.6(b)) shows the performance improvement of N-MASS relative to default scheduling. Performance improvements of individual programs up to 32%

are possible.

An interesting question is how much improvement is due to the maximum-local scheme (i.e., accounting *only* for data locality in the system), and how much benefit is due to the final refinement step of N-MASS (i.e., taking cache contention also into account). In Figure 3.5(c) and Figure 3.6(c) we compare the average performance improvement of N-MASS versus the maximum-local scheme. The bars show the maximum and performance improvement of the constituent programs of the workloads: A multiprogrammed workload consists of several independent programs; bars show the highest and lowest performance improvement recorded for any constituent program of the workload. (Bars do not show the "standard error", a negative performance improvement means performance degradation.)

N-MASS performs approximately the same as maximum-local in most of the cases. However, when the memory allocation in the system is unbalanced (allocation map 0000 for both workloads and allocation maps 0001, 0010, and 1100 for WL1), the additional cache-balancing of the N-MASS scheme improves performance relative to maximum-local. In these cases maximum-local results in a performance degradation relative to default, because cache contention on the LLCs cancels the benefit of good data locality. There are also some cases when N-MASS performs slightly worse than maximum-local, but its average performance is never worse than the performance of default scheduling. In summary, in most cases (for most allocation maps) optimizing for data locality gives the significant part of the performance improvement measured with N-MASS; accounting for cache contention helps mostly with unbalanced allocation maps.

3.3.4 A detailed look

In the previous section we have shown that in case of unbalanced memory allocation maps, the cache-aware refinement step of N-MASS improves performance over maximum-local. In this section we look in detail at the performance of the policies N-MASS and maximum-local in case of unbalanced memory allocation maps, and extend our measurements to the 8-process workloads. Figures 3.7 and 3.11 show the performance for each of the programs in the various workload sets (Figure 3.7 considers the 4-process workloads WL1–WL9, Figure 3.11 considers the 8-process workloads WL10 and WL11).

In many cases, the maximum-local mapping scheme performs well and the final refinement step of N-MASS brings only small benefits. However, in the case of WL1, WL7, WL8, WL9, and WL11, individual programs of the workloads experience up to 10% less performance degradation with N-MASS than with maximum-local. Performance degradations relative to default scheduling are also reduced from 12% to at most 3%.

Figures 3.8 and 3.12 show the data locality (as percentage of local main memory references) of WL10–WL11 and WL1–WL9, respectively. The maximum-local scheme keeps all memory references local in case of 4-process workloads. For 8-process workloads maximum-local preserves the data locality of its memory-bound processes (P1–P3 in case of WL10 and P1–P4 in case of WL11). With N-MASS, workloads have less data locality than with maximum-local. In case of 8-process workloads, N-MASS swaps compute-intensive workloads with memory intensive workloads (relative to the maximum-local scheme) to reduce cache contention. As a result, N-MASS increases these workloads' data locality relative to maximum-local.

Figures 3.9 and 3.13 show the MPKI of WL1–WL9 and WL10–WL11, respectively. N-

MASS decreases the MPKI of workloads relative to maximum-local by accounting for cache contention. These figures contrast the figures showing the workloads' data locality (Figures 3.8 and 3.12).

3.4 Process memory behavior characterization

Scheduling decisions taken by N-MASS rely on measures that describe a process's memory behavior. N-MASS characterizes memory behavior based on three measures: (1) the process's NUMA penalty, (2) the process's cache pressure, and (3) the process's data distribution. So far we have simulated the availability of these measures by using statically collected program traces. In this section we investigate how estimates of memory behavior and information from the processor's performance monitoring unit (PMU) can be used to replace static information and thus make N-MASS more practical.

3.4.1 Estimating the NUMA penalty

Figures 3.7 and 3.11 show the effect of estimating the NUMA penalty through linear regression (the "regression-based N-MASS" data set in the figures). For this evaluation, we do not use the profile-based information about the NUMA-penalty of programs (as this number may be difficult to obtain in current NUMA-multicore systems). Instead, we estimate the NUMA penalty based on the MPKI (still obtained from statically-collected traces): we fit a simple linear model onto the data shown in Figure 3.2. Before fitting the model we remove the outlier `mcf` (data point "1" on Figure 3.2) as well as all programs with an MPKI smaller than 1.0. The resulting model's slope and intercept are 0.015 and 1.05, respectively. For the measurements, the memory of all processes of the multiprogrammed workloads was allocated on the same processor, Processor 0 (allocation maps 0000 and 00000000).

Similar to profile-based N-MASS, the regression-based version of N-MASS improves performance over the maximum-local scheme. In some cases, however, using estimates of a process's NUMA penalty results in a different amount of improvement than with profile-based N-MASS (we measure performance improvements/degradations of individual programs of up to 10% relative to profile-based N-MASS). The changes in performance are due to the imprecision of the MPKI-based estimates of the NUMA penalty. On-line techniques to estimate the NUMA penalty are difficult to construct with the PMU of current CPU models. We hope that future PMUs will provide events that can be used to estimate the NUMA penalty better. If the NUMA penalty cannot be obtained directly, the MPKI offers a reasonable approximation for this scheduler.

3.4.2 Monitoring cache pressure

When executed in a multiprogrammed configuration, the MPKI of a program can change significantly relative to the solo-mode MPKI of the program. Figures 3.14 and 3.10 show the MPKI of each constituent program of WL10–WL11 and WL1–WL9 (relative to the solo mode execution of the constituent program), respectively, when the workloads are executed with the maximum-local and N-MASS schemes. Due to cache contention, programs can experience a high MPKI increase relative to their solo-mode MPKI (up to 10X in case of memory-bound programs and up to 10'000X for compute-intensive programs).

Previous work [14, 15, 113] has shown that, despite its sensitivity to dynamic conditions, the MPKI is suitable to estimate cache pressure for scheduling algorithms. However, scheduling decisions made by the N-MASS scheduler rely on *both* the measure of a process's cache pressure and the measure of the process's NUMA penalty. We configure the N-MASS scheduler to use MPKI values measured at runtime; in this configuration the NUMA penalty of processes is estimated with the regression model previously described in Section 3.4.1. The performance results for the configuration with on-line measurement of a process MPKI are reported as the data set "N-MASS regression + dynamic MPKI" in Figures 3.7 and 3.11 for 4-process and 8-process workloads, respectively. The results are largely similar to the configuration using only static MPKI information ("N-MASS") and the configuration using static MPKI information and linear regression ("N-MASS regression").

3.4.3 Determining a process's home node

To limit the number of cases that must be evaluated, in the experiments we restrict memory allocation of a process to a single processor. As a result, the home node of processes is a priori known.

Determining a process's home node (the processor whose memory it accesses most frequently) at runtime is difficult. Current OSs provide information about the distribution of a process's pages across the processors of the system, but the distribution of a process's pages often does not correlate well with the distribution of the process's accesses to its pages. Many OSs provide an estimate of how frequently each page is accessed (e.g., by measuring how frequently accesses to a page result in a page fault) and this information can enable a process scheduler to estimate a process's preferred home processor if the process's memory is scattered around.

Determining a process's home node based on hardware performance counter feedback is not easy either because on current microarchitectures there is a limited number of counters available for monitoring DRAM accesses. For example, on the Intel Nehalem there is only one counter to monitor DRAM transfers; this counter can be programmed to monitor either local or remote DRAM transfers, but not both at the same time. Multiplexing several events onto the same counter is possible but results in somewhat decreased measurement precision [7].

On newer systems (e.g., the Intel Westmere) there are two counters available for monitoring DRAM transfers, so both local and remote memory transfers can be monitored at the same time. However, in a Westmere-based system with more than two processors it is impossible to attribute remote traffic to specific processors with the events supported by the processor (e.g., on a 4-processor Westmere-based system 3 processors are remote and DRAM transfers from/to all these 3 processors are measured by a single hardware performance monitoring event).

Newer processors have advanced features. For example, Intel processors support latency-above-threshold profiling, a sampling-based technique. Latency-above-threshold profiling can approximate the distribution of a process's data by looking at the data addresses used by sampled memory access instructions (as it demonstrated, e.g., by the standard `numatop` utility of Linux [49]). The overhead and precision of latency-above-threshold strongly depends on the sampling rate (as shown by Marathe et al. [67, 68] for past architectures). For current architectures the overhead/precision is yet to be experimentally evaluated.

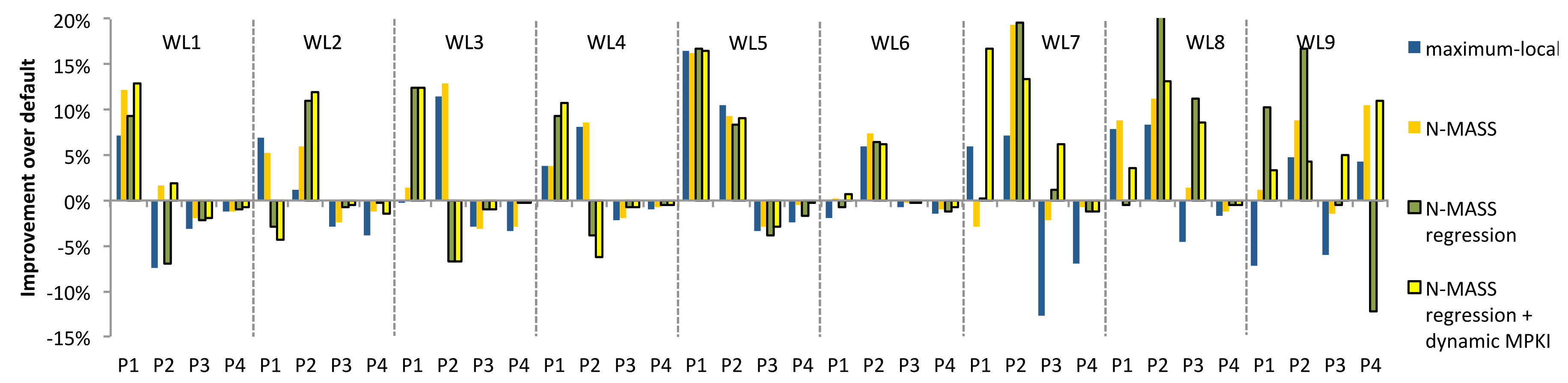


Figure 3.7: Performance improvement of 4-process workloads.

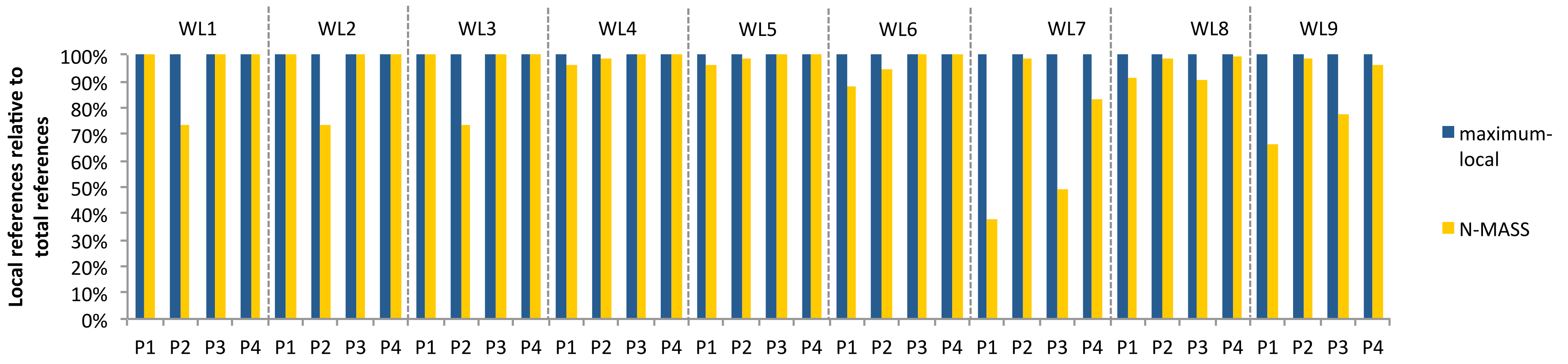


Figure 3.8: Data locality of 4-process workloads.

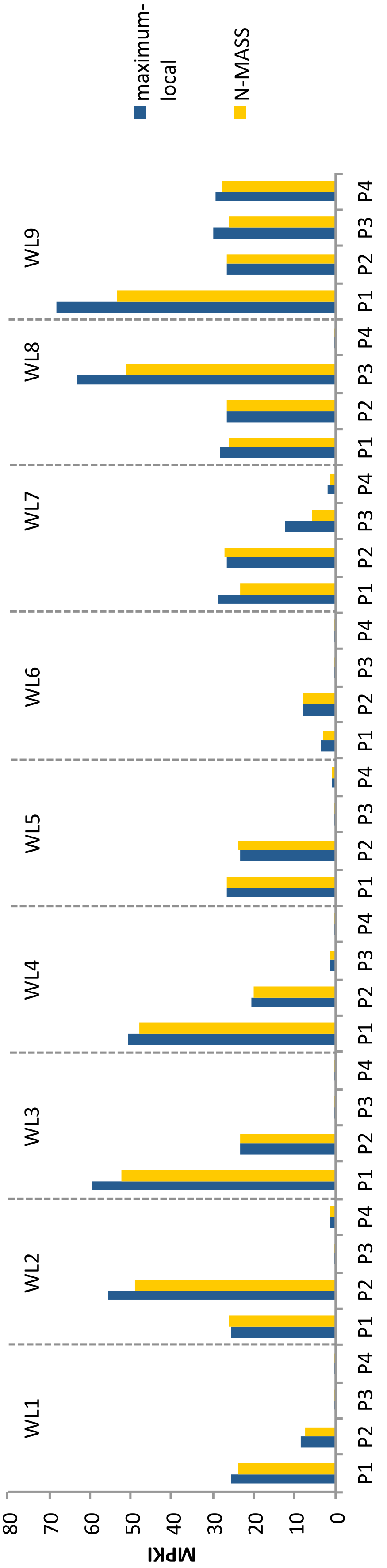


Figure 3.9: Absolute MPKI of 4-process workloads.

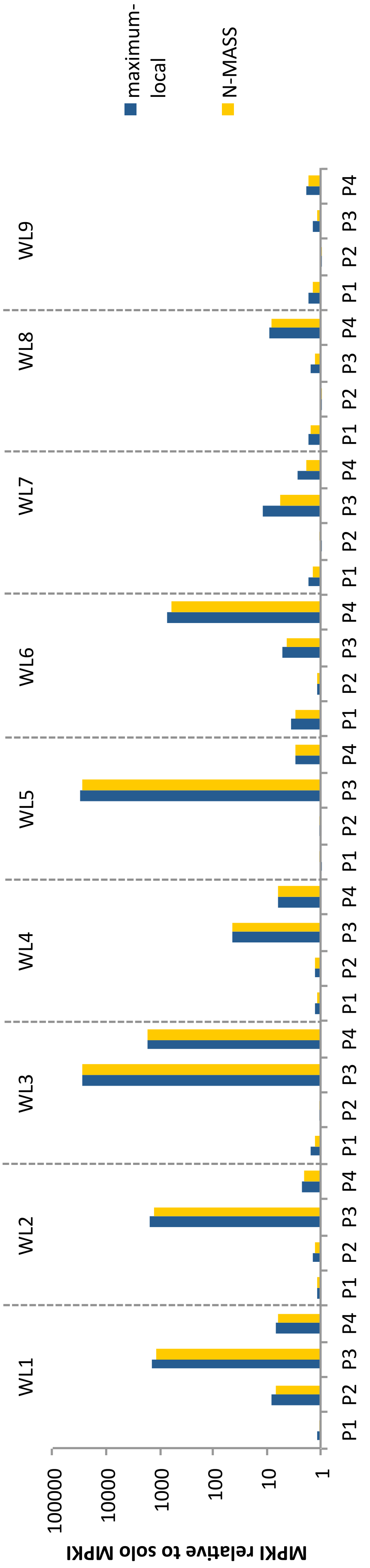


Figure 3.10: Relative MPKI of 4-process workloads.

3.5 Discussion and limitations

In summary, if the memory allocation in the system is balanced (i.e., there is an approximately equal number of processes homed at every processor), then maximum-local scheduling provides large performance benefits. If the memory allocation setup of the system is unbalanced, the mapping given by the maximum-local scheme often needs adjustment, as maximum-local mapping can result in contention for shared caches and thus a performance degradation (even relative to default scheduling). In cases with unbalanced memory allocation, the refinement step of N-MASS can re-map processes onto a different LLC to reduce cache contention.

Memory migration is an alternative technique to improve data locality in NUMA systems. We limit the discussion to process scheduling because of two issues: (1) it is difficult to estimate the cost of memory migration, and (2) memory migration is not always possible because there is not always enough free memory available on the destination processor. In these cases the process scheduler is the only part of the system software that can optimize performance.

We do not consider multithreaded programs with a shared address space. For these programs sharing caches can be beneficial, therefore finding a tradeoff between data locality and cache contention is difficult. Recent work by Dey et al. [28] extends resource-conscious scheduling to multithreaded programs as well.

The N-MASS algorithm is designed for 2-processor systems. On systems with more processors the algorithm must consider balancing between all caches/processors. That is, maximum-local mapping must distribute processes across all processors of the system (in descending order of the processes' NUMA penalties). Moreover, in cases when maximum-local mapping results in an unbalanced distribution of cache pressure between the LLCs of the system, it can be beneficial to distribute cache pressure across LLCs, but only judiciously, so that the remote execution penalty experienced by processes remains within reasonable limits. We leave the extension of the N-MASS algorithm for larger systems for future work.

Porting the N-MASS scheduler to systems with enabled simultaneous multithreading (SMT) [106] would require extending the N-MASS algorithm to consider not only the performance degradation due to remote execution and cache contention, but also the performance degradation due to sharing resources (e.g., execution units, memory ports) of a single core (e.g., in cases when processes are mapped onto hardware threads of the same processor core). Existing schedulers for SMT systems can evaluate [92] or approximate [32] on-line the performance degradation programs experience due to sharing on-core resources. The N-MASS algorithm could be extended to use any of the previously mentioned approaches to characterize the performance impact on on-core resource sharing as well. We leave the extension of N-MASS to SMT systems for future work.

3.6 Conclusions

We have shown that operating system scheduling fails to obtain good performance in NUMA-multicores if it does not consider the structure of the memory system and the allocation of physical memory in the system. If memory allocation in a NUMA-multicore system is balanced (the cumulative memory demand of processes homed on each processor in the system is approximately the same), then it is beneficial to simply map processes onto the architecture so that data locality is favored, and avoiding cache contention does not bring any benefits.

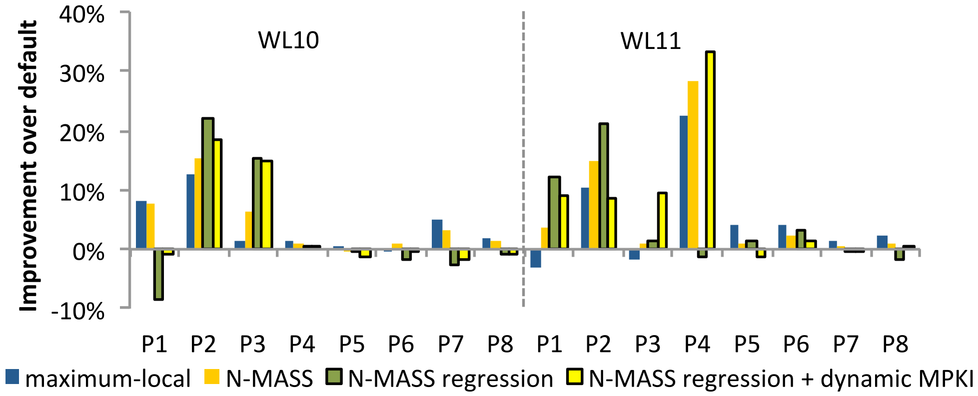


Figure 3.11: Performance improvement of 8-process workloads.

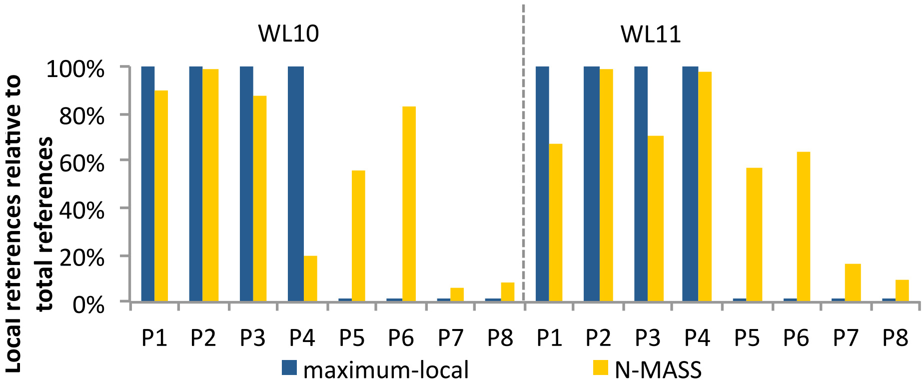


Figure 3.12: Data locality of 8-process workloads.

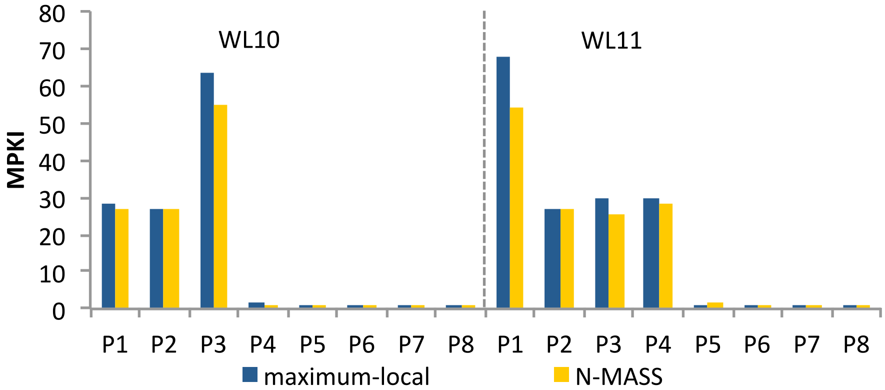


Figure 3.13: Absolute MPKI of 8-process workloads.

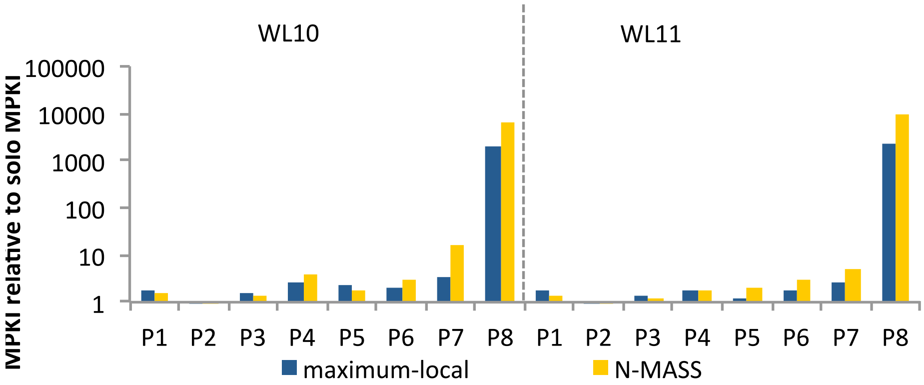


Figure 3.14: Relative MPKI of 8-process workloads.

Nonetheless, when the memory allocation in the system is unbalanced (the sum of the memory demands of processes homed on each processor in the system is different), then mapping processes so that data locality is favored can lead to severe cache contention. In these cases refining the maximum-local mapping so that cache contention is reduced improves performance, even with the cost of some processes executing remotely. The N-MASS scheme described in this chapter successfully combines memory management and process scheduling to better exploit the potential of NUMA multicore-multiprocessors.

4

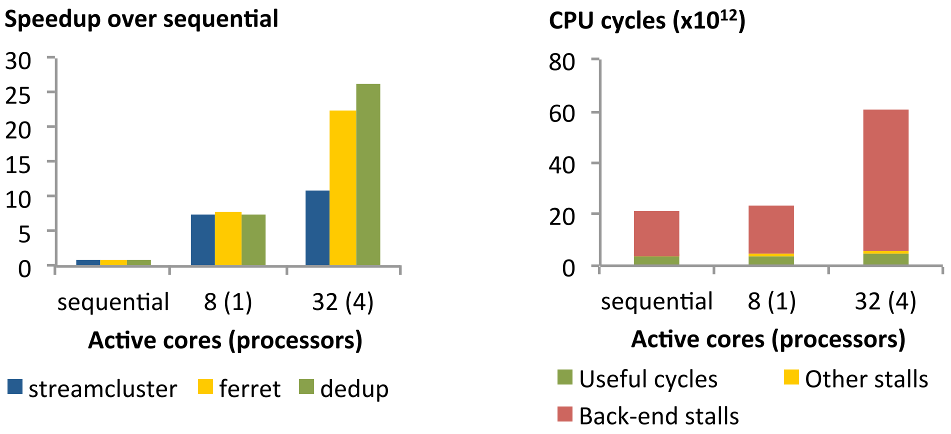
Performance analysis of multithreaded programs

A multithreaded program consist of a set of threads that operate in a single shared memory region. Multithreaded programs belong to a separate application class and are thus fundamentally different from multiprogrammed workloads, the class of applications considered in the previous chapters. Consequently, the memory system bottlenecks experienced by multithreaded programs are possibly different from the bottlenecks experienced by multiprogrammed workloads.

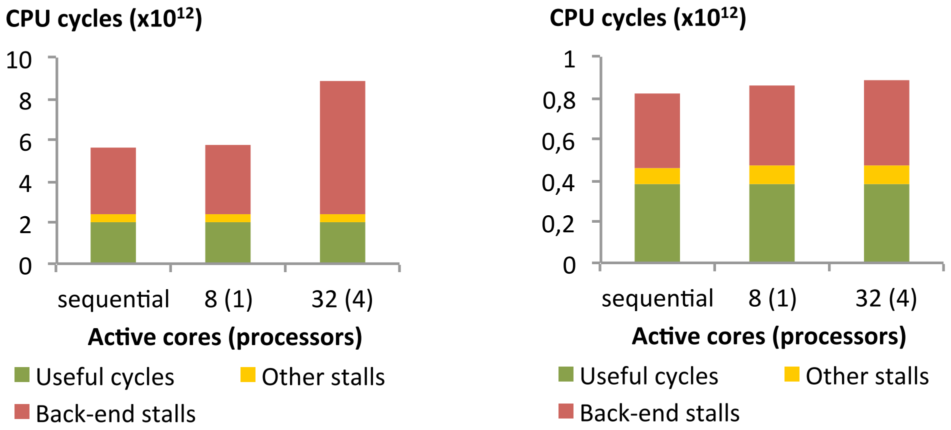
This chapter focuses on understanding the memory system performance of multithreaded programs on NUMA multicore-multiprocessors. We start the discussion by investigating the performance scaling of a set of programs from the well-known PARSEC benchmark suite [10] (Section 4.1). Then, in Section 4.2 we present the setup we use for experimental evaluation. Section 4.3 presents a detailed analysis of the memory system performance of the PARSEC benchmark programs. Based on the analysis we identify and quantify two performance-limiting factors, *program-level data sharing* and *irregular memory access patterns*. These factors result in inefficient usage of the memory system: data sharing causes bad data locality, irregular memory access patterns are difficult to predict by hardware prefetcher units and, as a result, programs experience the full access latency of their memory accesses.

To understand the performance implications of program-level data sharing and irregular memory access patterns, we restructure the benchmark programs to eliminate them so that the memory behavior of the programs matches with the requirements of the memory system. In Section 4.4 we describe three simple source-level techniques that we use: (1) *controlling the data distribution* of the program to make sure that memory regions are allocated at well-defined and distinct processors, (2) *controlling the computation distribution* of the program to guarantee that computations operate on distinct subsets of program data, and (3) *regularizing memory access patterns* so that the access patterns are easier to predict for processor prefetcher units. All techniques rely either on using existing OS functionality or on simple algorithmic changes.

As the proposed techniques effect many layers of a modern NUMA-multicore memory system, Section 4.5 reports detailed measurements of orthogonal experiments to quantify how each distinct layer is affected. Moreover, we compare the performance of the optimized programs with other optimizations proposed for NUMA-multicores (e.g., page-level interleaving, data replication), hence the experiments help to understand the implications of data locality and resource contention for the performance of multithreaded programs on NUMA multicore-multiprocessors.



(a) Performance scaling. (b) Cycles: `streamcluster`.



(c) Cycles: `ferret`. (d) Cycles: `dedup`.

Figure 4.1: Performance scaling and cycle breakdown.

4.1 Performance scaling

To understand the the memory system performance of multithreaded workloads, we select three benchmark programs, `streamcluster`, `ferret`, and `dedup`, from the PARSEC benchmark suite [9, 10] and we execute these programs on a recent 4-processor 32-core machine. The 4-processor 32-core is similar to the 2-processor 12-core machine used in Chapter 2: Both machines are based on the Westmere microarchitecture and so have a similar memory system, however, the 4-processor machine allows additional cross-chip interconnects as there can be a point-to-point interconnect between any two of the four processors of the system (see Section 4.2 for more details). In the following we refer to the 4-processor 32-core system as "Westmere-based system" (that is not to be confused with the 2-processor 12-core system used in Chapter 2).

Figure 4.1(a) shows the performance scaling of the three benchmark programs on the 4-processor 32-core Westmere-based system. For each program we report performance in three cases. The first case is *sequential execution* when the non-parallelized version of the program is executed on a single core/processor. In the second case, *8-core (1 processor)*, the parallel version of the program is executed on 8 cores, but on a single processor. In the first and second case the program is restricted to execute at a single processor, thus all program memory is allocated at that processor. As a result, there is no cross-chip traffic in the system. The third case, *32-core (4 processors)*, considers the scenario when all cores (and thus all processors) of the system are used to execute program threads. As the program is running on all cores/processors, memory allocation is not restricted to a single processor.

Figure 4.1(a) shows that the performance of all three programs scales well to 8 cores (1 processor), but the picture is different in the 32-core (4 processors) case: `streamcluster` shows bad performance scaling (11X speedup over single-threaded execution), `ferret` scales better (20X speedup), and `dedup` scales well (26X speedup).

There are many possible reasons for the (non-)scalability of parallel programs (e.g., serialization due to extensive synchronization, load imbalance [87]), and there exist many techniques that target these problems. A crucial and interesting issue is, however, how inefficiencies related to the memory system effect performance scaling on NUMA-multicores. To get more insight, we break down the total number of CPU cycles of each program into three categories: (1) *useful cycles*, when the CPU makes progress, (2) *back-end stalls*, when execution is stalled because the processor's back-end cannot execute instructions due to the lack of hardware resources, and (3) *other stalls*, that is, stalls related to the processor's front-end (e.g., instruction starvation). (See Section 4.2 for details about the performance monitoring setup used.)

In Intel Westmere processors the processor back-end can stall due to many reasons (including the memory system), however, due to limitations of the Westmere's performance monitoring unit we cannot distinguish the number of back-end stalls related to the memory system from back-end stalls due to other reasons. Nevertheless, as we change only the way a program is mapped onto the architecture (but not the instruction stream generated by the programs), we suspect a change in the number of back-end stalls (when comparing two setups) is related to memory system inefficiencies.

Figure 4.1(b) shows that the sequential version of `streamcluster` spends already a large fraction of its cycles on waiting for the back-end. We measure the same amount of back-end stalls for the case when the parallel version of the program is executed on a single processor (the 8-core (1-processor) case). However, when the program is executed on multiple processors, there is almost a 3X increase in the number of back-end stalls. `ferret` has similar, yet not as pronounced problems (2X more back-end stall cycles in the multiprocessor configuration than in the single-processor configurations, as shown Figure 4.1(c)). The performance of `dedup` is not affected in the multiprocessor configuration, as its number of back-end stalls increases only slightly when executed on multiple processors (Figure 4.1(d)). In summary, as long as a program uses only one processor in a NUMA-multicore system, the performance of the parallel and sequential version of the program are similar, but if all processors are used, performance can unexpectedly degrade, most likely due to a program using the memory system inefficiently. To determine the exact cause, further investigation is needed. Sections 4.2 and 4.3 present details of this investigation.

4.2 Experimental setup

4.2.1 Hardware

Two machines are used for performance evaluation: a 2-processor 8-core (Intel Nehalem microarchitecture) and a 4-processor 32-core (Intel Westmere microarchitecture) machine. Table 4.1 lists the detailed configuration of both systems. The size of per-core level 1 and level 2 caches is the same on both machines, however, the size of the level 3 cache differs between the machines. In both systems there is a point-to-point connection between every pair of processors, thus every processor is a one-hop distance from any other processor of the system. We disable frequency scaling on both machines [22]. Processor prefetcher units are on if not stated otherwise.

In this chapter, we use the same 2-processor 8-core Nehalem-based machine, as in the previous chapters, but in a configuration with different total main memory size (48 GB instead of 12 GB). Although different in size, the clock rate, throughput, and the timings of the memory modules are the same (DDR3-1066, 7-7-7), so the penalty of remote main memory accesses on the Nehalem-based system is approximately the same in both configurations.

	Nehalem	Westmere
Model number	Intel Xeon E5520	Xeon E7-4830
Number of processors	2	4
Cores per processor	4	8
Total number of cores	8	32
Clock frequency	2.26 GHz	2.13 GHz
L3 cache size	2x8 MB	4x24 MB
Main memory	2x24 GB DDR3	4x16 GB DDR3
Cross-chip interconnect	5.86 GTransfers/s	6.4 GTransfers/s

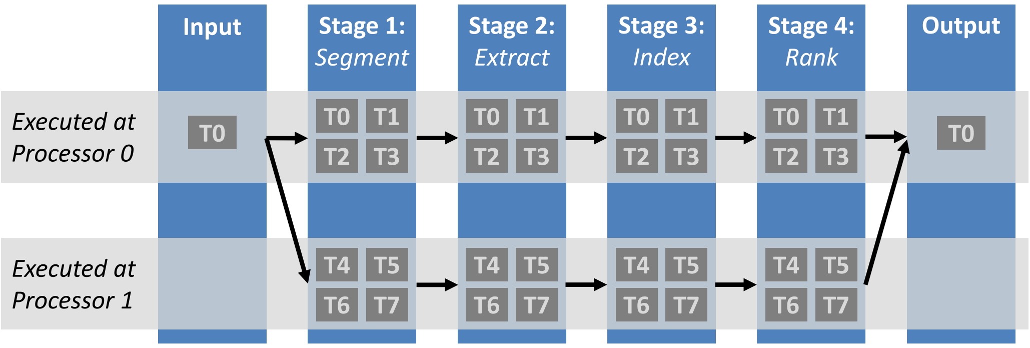
Table 4.1: Hardware configuration.

4.2.2 Benchmark programs

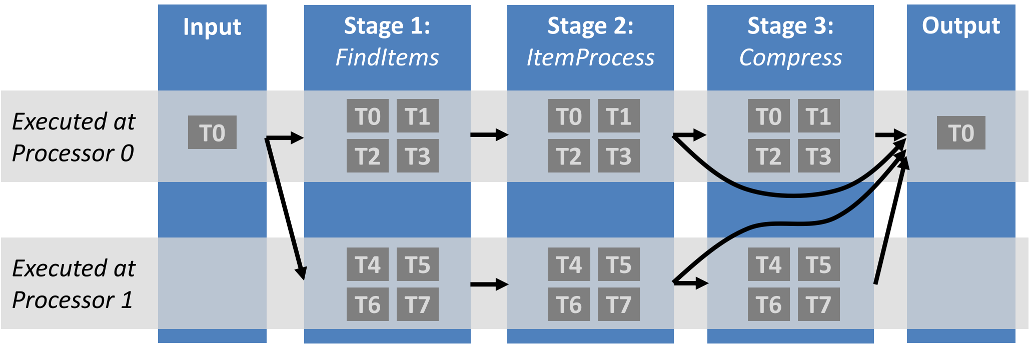
We consider three programs of the PARSEC benchmark suite [10].

- `streamcluster` is a data-parallel program that solves the on-line clustering problem for a set of input points. The program consists of a series of processing loops. Each loop is parallelized with the OpenMP `parallel for` directive with static scheduling; with static scheduling each worker thread is assigned a well-defined subset of the total iterations processed by a loop [80]. If not specified otherwise, `streamcluster` is configured with a number of worker threads equal to the number of cores of the machine it executes on (so that it makes optimal use of the computational resources offered by the architecture).

- `ferret` is a pipeline-parallel program [37, 41, 76, 89, 95] that implements content-based similarity search of images: given a set of input images, `ferret` searches in an image database for images similar to the input image. The program is structured as a set of 6 stages, as shown in Figure 4.2(a). Stages are interconnected by queues, data "flows" from the input stage through intermediary stages towards the output stage of the pipeline. Each pipeline stage of `ferret` is executed by a number of threads equal to the number of cores in the system; the input and the output stage is executed by a single thread each. Figure 4.2(a) shows the runtime configuration of `ferret` on a 2-processor 8-core machine.



(a) `ferret`.



(b) `dedup`.

Figure 4.2: Structure and configuration of pipeline programs.

- `dedup` is also a pipeline-parallel programs that consists of 5 stages. `dedup` compresses and deduplicates files given to it as input. The structure and runtime configuration of `dedup` (after cleanup) is shown in Figure 4.2(b) and it is similar to that of `ferret`.

Table 4.2 lists the benchmark programs together with the inputs considered for each program. We have grown the size of the inputs relative to *native*, the largest input size available in the PARSEC suite. Most native-sized inputs fit into the last-level caches of the systems considered (e.g., the 32-core machine has 96 MB of total last-level cache), thus growing the size of the inputs allows us to exercise the memory system of the machines.

		Execution time	
Program	Input	Nehalem (8 cores)	Westmere (32 cores)
`streamcluster`	10 M input points	1232 s	937 s
`ferret`	image database with 700 M images and 3500 input images to process	292 s	133 s
`dedup`	4.4 GB disk image	46 s	14 s

Table 4.2: Benchmark inputs and run times with default setup.

4.2.3 Scheduling and memory allocation

In NUMA systems, the placement of memory pages has a large impact on performance. Both machines that we use for evaluation run Linux, which, similar to other OSs, relies on the *first-touch page placement policy*. According to this policy each page is placed at the processor that first reads from/writes to the page after it has been allocated.

In NUMA systems not only the placement of memory pages at processors, but also the mapping of program threads to cores impacts program performance. In their default configuration OSs tend to change the thread-to-core mapping during the execution of programs. Such changes result in large performance variations. To reduce the negative effect of OS reschedules we use *affinity scheduling with identity mapping*. Affinity scheduling restricts the execution of each thread to a specific core. For `streamcluster`, worker threads that execute a parallel loop are always mapped to cores so that T0 is executed at Core 0, T1 is executed at Core 1, etc. For `ferret` and `dedup` identity mapping is used for each pipeline stage individually (i.e., for each pipeline stage the worker threads executing that stage will be mapped to cores using identity mapping).

Identity mapping defines not only thread-to-core mappings, but implicitly also the thread-to-processor mapping (in NUMA systems, memory is distributed on a per-processor basis and not on a per-core basis). By using identity affinity there is an equal number of threads executing each parallel loop (in case of `streamcluster`) and each stage (in case of the pipeline programs) at all processors of the system. For example, on the 2-processor 8-core system threads T0–T3 of each parallel loop/stage execute on Processor 0, and threads T4–T7 of each parallel loop/stage execute on Processor 1. Figure 4.2(a) and Figure 4.2(b) show how threads are mapped to processors for the `ferret` and `dedup` benchmark, respectively. Using identity mapping for mapping threads to processors/cores reduces measurement variation; furthermore, it does not affect the cache performance of the programs, as noted by Zhang et al. in [112]. Table 4.2 lists the execution time of the benchmark programs. The programs are compiled with the GCC compiler version 4.6.1 using optimization level `O3`.

4.2.4 Performance monitoring

Table 4.3 lists the performance monitoring events used to break down execution cycles into three categories (the breakdown shown in Figures 4.1(b)–4.1(d)): useful cycles, back-end stall cycles, and other stall cycles. As we disable simultaneous multithreading the cycles reported in the "other stall cycles" category are due to instruction starvation [44] and are calculated as the difference of the total number of measured stall cycles and the total number of back-end stall cycles.

Event name	Reported as
`UNHALTED_CORE_CYCLES`	Total execution cycles
`UOPS_ISSUED:STALL_CYCLES`	Total stall cycles
`RESOURCE_STALLS:ANY`	Total back-end stall cycles

Table 4.3: Performance monitoring events.

We measure the read memory traffic generated by programs on the uncore of the evaluation machines. The uncore of a processor includes the processor's last-level cache, its on-chip memory controller, and its interfaces to the cross-chip interconnect. We break down uncore memory traffic into four categories: local/remote last-level cache accesses (cache hits) and local/remote main memory accesses (cache misses that are served either from local- or from remote main memory). The four categories are illustrated on Figure 1.1 for a 2-processor system. We use the `OFFCORE_RESPONSE_0` performance monitoring event with the response types specified in Table 4.4. We use the same request types as previously shown in Table 2.1. Limitations of the performance monitoring unit restrict the kind of information that can be obtained: only read transfers can be measured on Nehalem/Westmere processors.

Response type	Reported as
`UNCORE_HIT:OTHER_CORE_HIT_SNP:OTHER_CORE_HITM`	Local L3 cache accesses
`REMOTE_CACHE_FWD`	Remote L3 cache accesses
`LOCAL_DRAM`	Local DRAM accesses
`REMOTE_DRAM`	Remote DRAM accesses

Table 4.4: Response types used with the `OFFCORE_RESPONSE_0` event.

4.3 Understanding memory system behavior

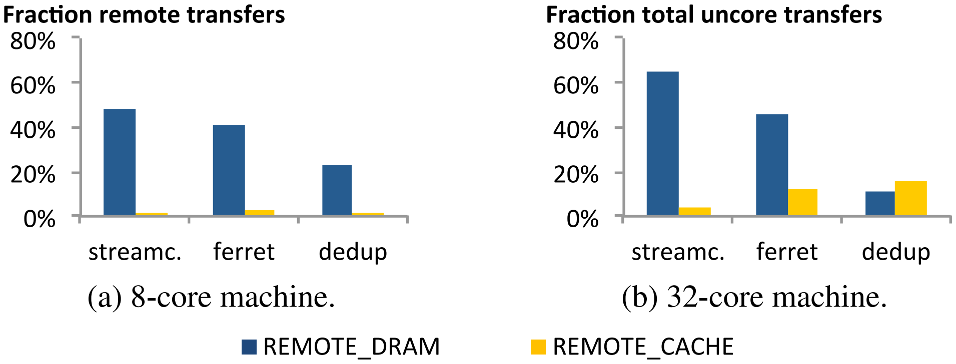
This section analyzes the memory system performance of the benchmarks. We focus on data locality (Section 4.3.1) and prefetcher effectiveness (Section 4.3.2).

4.3.1 Data locality

Figure 4.3(a) (4.3(b)) shows the fraction of remote memory transfers relative to the total number of memory transfers generated by the programs on the 8-core (32-core) machine. On both machines a large fraction (11–67%) of the program's memory transfers are handled by remote main memory. Moreover, in some configurations, the programs show a large fraction of remote cache accesses, up to 16% of the total number of uncore transfers of the program. The measurements indicate that the PARSEC programs use an ill-suited data distribution (i.e., program data frequently resides in memory that is remote to the threads accessing it). Using affinity scheduling has the consequence that OS reschedules do *not* introduce remote memory transfers (as there are none). To understand the reasons for the ill-suited data distribution, we perform data address profiling on all three programs.

Data address profiling is a sampling-based approach that records the target data address (and thus the target memory page) of each sampled memory operation [2, 13, 27, 55, 62, 68]. As data address profiling is sampling-based, it only approximates program behavior. Nonetheless, profiling data addresses still provides a good indication of where to concentrate the optimization effort. Similarly to the MemProf profiler [55], we use the processor's performance monitoring unit to gather address profiles. We use the load latency performance monitoring facility of Intel processors [45, 68]. Other microarchitectures support data-address profiling as well (e.g., AMD processors implement Instruction-Based Sampling [13, 27, 61] which offers functionalities similar to the Intel implementation of data address profiling).

Because of limitations of the Linux performance monitoring system, programs are profiled only on the 2-processor 8-core Nehalem-based machine. On Nehalem-based systems the PMU



(a) 8-core machine.

(b) 32-core machine.

Figure 4.3: Remote transfers as fraction of all uncore transfers.

gathers data address profiles for memory load operations (but not for stores) [45]. We use the `MEM_INST_RETIRED` event configured with the `LATENCY_ABOVE_THRESHOLD` mask; the sampling rate is set to 1. The PMU samples only memory load operations that have a latency greater or equal than the latency threshold. On the Nehalem microarchitecture the L1 data cache can be accessed in 4 cycles [57]. The latency threshold is therefore set to 4 cycles so that we sample accesses to all levels of the memory hierarchy.

Based on their target data address, we group memory accesses into two categories:

- *accesses to non-shared heap pages* (pages where only accesses by a single processor are recorded)
- *accesses to shared heap pages* (pages accessed by both processors of the NUMA system).

Accesses to pages that hold the program's image, dynamically-linked shared objects and the program's stack are excluded from profiles.

Exclusively accessed heap pages are placed appropriately in processor memories by the first-touch page placement policy, as these pages are accessed only by a single core/processor. Moreover, data from private and exclusively accessed pages is present in only one cache of the system, so these pages are likely not to be accessed from remote caches. However, for shared pages it is difficult to find an appropriate placement in a NUMA system: no matter at which processor of the system a shared page is placed, there will be accesses to these pages from a remote processor, as all processors access the shared data during the lifetime of the program. Likewise, the contents of shared pages is likely to be present in multiple caches of the system at the same time, thus reducing the capacity available to non-shared data and also leading to remote cache references.

Figure 4.4 shows the results of data address profiling for the PARSEC programs: a large fraction (10–70%) of the programs' memory accesses are to problematic shared heap locations. Investigation of the address profiles and of the program source code reveals that data sharing is caused by data structures being shared between all threads of the program: As program threads execute at different processors, some threads are forced to access the shared data structure through the cross-chip interconnect. In the case of `streamcluster`, data sharing is caused by accesses to the data structure holding the coordinates of processed data points. In the case of `ferret`, data sharing is caused by lookups in the image database. In the case of `dedup`, most

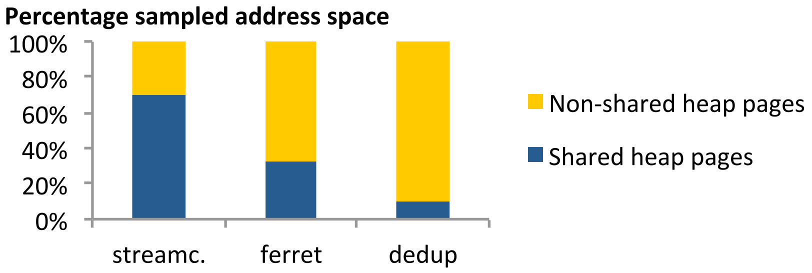


Figure 4.4: Data access characterization (8-core machine).

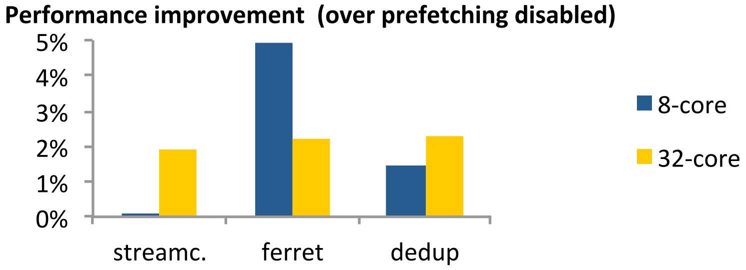


Figure 4.5: Performance gain w/ prefetching.

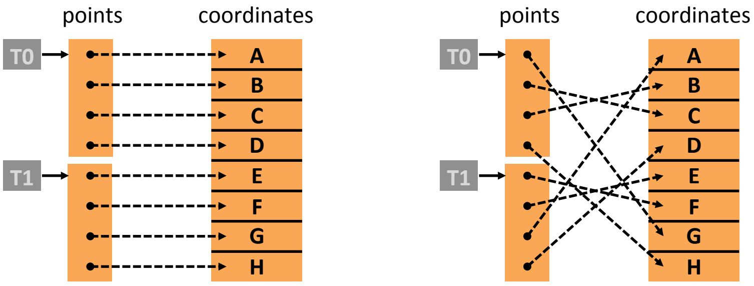
shared memory accesses are caused by accesses to a shared hash table that stores compressed chunks of the input file.

4.3.2 Prefetcher effectiveness

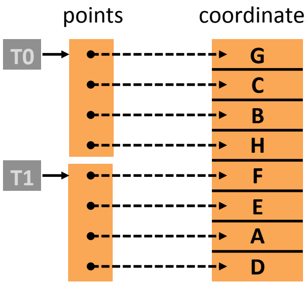
To assess the effectiveness of the hardware prefetchers, we measured the performance of each benchmark in two configurations: with disabled processor prefetcher units and with the prefetcher units enabled. Figure 4.5 shows the performance improvement gained due to prefetching. In general, prefetchers improve performance, however the amount of performance improvement is small in most cases (5% at most), and in some cases prefetchers do not effect performance at all. The ineffectiveness of hardware prefetching can be explained by the irregular access patterns of the programs considered: `streamcluster` accesses the data points randomly, and both `ferret` and `dedup` store/look up data in hash tables at locations that look random to prefetchers that are good in recognizing strided data access patterns [110].

4.4 Program transformations

The previous section shows that two factors, program-level data sharing and irregular memory access patterns, cause the poor performance scalability of the three PARSEC programs on NUMA systems. This section describes a set of simple source-level changes that allow the previously considered programs to better use the memory system of NUMA-multicores. The changes effect the mapping of programs, that is, changes effect both the data distribution and the computation scheduling of the programs. We set up the mapping by using standard OS functionalities to distribute memory pages across processors (Section 4.4.1) and by making simple algorithmic changes to the programs (Section 4.4.2).



(a) Original state. (b) Pointer-based shuffle.



(c) Copy-based shuffle.

Figure 4.6: `streamcluster`: Shuffles.

4.4.1 Distributing data

To avoid remote main memory accesses the programmer must control at which processor data is allocated. Current operating systems provide support for per-processor memory allocation. For example, Linux provides the `numa_alloc()` function that allows the programmer to allocate memory at a specific processor. Because of its high overhead, however, it is recommended to use this allocator only for the allocation of large memory regions. On Linux, once pages of a memory region are allocated, they can be migrated to a specific processor using the `move_pages()` system call. For the performance-critical memory regions of the programs considered we define a data distribution either by using `numa_alloc()` or by using memory migration. Defining the data distribution for these regions adds overhead. However, as we enforce the distribution only once, at program startup, the overhead is compensated for by improved execution times. The overhead of distributing data is included in the performance measurements (reported in Section 4.5).

4.4.2 Algorithmic changes

`streamcluster`

The performance analysis described in Section 4.3 reveals that most memory accesses of the program are to coordinates of data points processed by the program. The coordinates are stored in an array, `coordinates`. The `coordinates` array is accessed through a set of pointers stored in the `pointers` array. Figure 4.6(a) shows the two data structures, assuming that the program is executed by two threads. Because of data parallelism each thread owns a distinct subset of pointers (and thus accesses a distinct set of points and coordinates). The situation with a higher number of threads is analogous to the two-thread case.

Investigation of the program source reveals that the bad memory system performance of `streamcluster` is caused by a data shuffling operation: during program execution the set of data points processed by the program are randomly reshuffled so that clusters are discovered with higher probability. Figure 4.6(a) shows the state of the `points` and `coordinates` array when the program starts, Figure 4.6(b) shows the state of the program data after the first shuffle. After shuffling, each thread accesses the same subset of the `pointers` array as before the shuffle, but the set of coordinates accessed is different. Shuffling is performed several times during the lifetime of the program.

The shuffling operation has negative effects on caching, prefetching and also on data locality. As the set of data accessed by threads changes during program execution, the program does not have much data reuse. The presence of random pointers makes the access patterns of the program unpredictable and thus prefetching is not useful for this program. Lastly, due to the shuffles, each piece of coordinate data is possibly accessed by all threads of the program, thus coordinate data is shared between processors and it is impossible to distribute these data across the processors of the system. As a result, most accesses to the `coordinates` array are remote.

To enable memory system optimizations we change the way shuffle operations are performed. Instead of shuffling pointers so that they point to different data, we keep the pointers constant and move data instead. Figure 4.6(c) shows the data layout of `streamcluster` after a copy-based shuffle is performed on the original data shown in Figure 4.6(a). After the copy-based shuffle, the location of the coordinate data in memory accessed by each thread is the same as before the shuffling operations, however, the contents of the data locations has changed. As a result, during the lifetime of the program each thread will access the same memory locations. Because data locations are not shared between threads, they can be distributed appropriately between processors. Moreover, as each thread accesses its pointers sequentially, the access patterns of the program are regular, thus prefetchers have a better chance to predict what data will be accessed next.

Figure 4.7(a) shows the code of the original, pointer-based shuffle, Figure 4.7(b) shows the code of the copy-based shuffling operation. Copy-based shuffling involves `memcpy` operations that are more costly than switching pointers; however, shuffling is infrequent relative to data accesses, therefore this change pays off (see the performance evaluation in Section 4.5).

ferret

The performance analysis described in Section 4.3 indicates that `ferret`'s remote memory accesses are due to accesses to the shared image database. The image database is queried in Stage 3 (the indexing stage) of the program. Figure 4.8(a) shows the original pipelined implementation of this stage. Stage 3 (and other stages as well) receives the processor number *p* as parameter (line 1). As a result, each stage instance is aware of the processor as where it is executing (the processor *p* is specified by affinity scheduling with identity mapping). To keep lock contention low, two adjacent stages are usually interconnected by multiple queues, and only a subset of all threads executing the stage uses the same queue. Thus, each stage instance receives/puts data from/into queues local to that processor (lines 2 and 4). Database queries are implemented by the `index()` function (line 3).

A way to reduce data sharing for the `ferret` benchmark is to partition the image database between the processors of the system, so that each processor holds a non-overlapping subset

```
void shuffle() {
  for (i = 0; i < NUM_POINTS; i++) {
    j = random() % NUM_POINTS;
    temp = points[i].coordinates;
    points[i].coordinates = points[j].coordinates;
    points[j].coordinates = temp;
  }
}
```

(a) Shuffle (pointer-based implementation).

```
void shuffle() {
  for (i = 0; i < NUM_POINTS; i++) {
    j = random() % NUM_POINTS;
    memcpy(temp, points[i].coordinates, BLOCK_SIZE);
    memcpy(points[i].coordinates,
           points[j].coordinates,
           BLOCK_SIZE);
    memcpy(points[j].coordinates, temp, BLOCK_SIZE);
  }
}
```

(b) Shuffle (copy-based implementation).

Figure 4.7: `streamcluster`: Program transformations.

```
void index_pipelined(Processor p) {
  features = indexQueue[p].dequeue();
  candidateList = index(features);
  rankQueue[p].enqueue(candidateList);
}
```

(a) Stage 3 (original implementation).

```
void index_pipelined(Processor p) {
  features = indexQueue[p].dequeue();
  if (p == Processor.MIN) candidateList = 0;
  candidateList += index(p, features);
  if (p < Processor.MAX - 1) {
    dstProcessor = p + 1;
    indexQueue[dstProcessor].enqueue(candidateList);
  } else {
    rankQueue.enqueue(candidateList);
  }
}
```

(b) Stage 3 (optimized implementation).

Figure 4.8: `ferret`: Program transformations.

of the image database. Partitioning is performed at program startup when the image database is populated. Data distribution (described in Section 4.4.1) is used to ensure that each subset is allocated at the appropriate processor. The image database uses a hash-based index for lookup [63] that is modified as well to include information about the processor at which each image is stored.

Database queries must also be changed to support data locality; Figure 4.8(b) shows

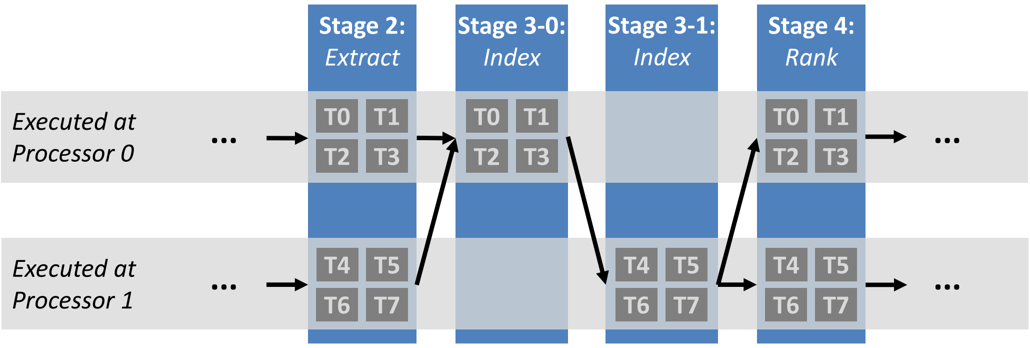


Figure 4.9: `ferret`: Stages 2, 3, and 4 (optimized implementation).

the locality-aware implementation of Stage 3. The `index()` function (line 4) is modified to operate only on a subset p of the image database ($p \in$ [0,`Processors.MAX`) where `Processors.MAX` is the total number of processors in the system). As a result, a stage instance executing at Processor p performs database queries only on partition p of the image database. Instances of Stage 3 are executed at every processor of the system (due to identity scheduling) thus all database partitions are eventually queried by the program. The final results of a query are composed from the results obtained by querying each partition of the complete image database.

To make sure that each query is performed on all database partitions, the queuing system is modified to dispatch queries to the appropriate processor. The partial result (stored in `candidateList`) of a query on partition p of the image database is forwarded to an indexing stage executing at the next processor (i.e., the candidate list is forwarded from Processor p to Processor $p+1$) (lines 6–7 of Figure 4.8(b)). When all subsets of the image database have been queried, the candidate list contains the result of the query on the complete database and it is sent to the ranking stage (line 9). All queries must start with the first partition of the image database. To ensure that each query starts with the first database partition, Stage 2 is modified to dispatch data to instances of Stage 3 executing at Processor 0. Similarly, to ensure load balance, the instances of Stage 3 querying the last partition of the image database dispatch data to all instances of Stage 4 (executing on all processors). As these changes are small, they are not shown in the code example. Figure 4.9 shows a graphical representation of Stages 2, 3, and 4 of the optimized program as they are executed on a 2-processor system. This representation indicates the change of queuing in all stages.

The example shown in Figure 4.8(b) is for a 2-processor system. On this system the proposed optimization corresponds to creating two copies of Stage 3, where one copy of this stage, Stage 3-0, executes at Processor 0, and the other copy, Stage 3-1, executes at Processor 1. This solution scales with the number of processors (i.e., on n processors n copies of Stage 3 are created).

`dedup`

Figure 4.10(a) shows the pipelined implementation of Stage 1 and Stage 2 of `dedup`. The performance analysis described in Section 4.3 indicates that accesses to the shared hash table

```
 1 // Stage 1: Divide data chunk into items.
 2 void findItems_pipelined(Processor p) {
 3   fileChunk = inputQueue[p].dequeue();
 4   items = findItems(fileChunk);
 5   for (item : items)
 6     itemProcessQueue[p].enqueue(item);
 7 }
 8 // Stage 2: Process an item.
 9 void itemProcess_pipelined(Processor p) {
10   item = itemProcessQueue[p].dequeue();
11   item.key = getSHA1Signature(item);
12   if ((entry = hashTable.get(key)) != null) {
13     entry.count++;
14     outputQueue[p].put(item.key);
15   } else {
16     compressQueue[p].put(item);
17   }
18 }
```

(a) Stages 1 and 2 (original implementation).

```
 1 // Stage 1: Divide data chunk into items.
 2 void findItems_pipelined(Processor p) {
 3   fileChunk = inputQueue[p].dequeue();
 4   items = findItems(fileChunk);
 5   for (item : items) {
 6     item.key = getSHA1Signature(item);
 7     dstProcessor = hashTable.processorForKey(key);
 8     itemProcessQueue[dstProcessor].enqueue(key);
 9   }
10 }
11 // Stage 2: Process an item.
12 void itemProcess_pipelined(Processor p) {
13   item = itemProcessQueue[p].dequeue();
14   item.key = clone(item.key);
15   if ((entry = hashTable.get(item.key)) != null) {
16     entry.count++;
17     outputQueue[p].put(item.key);
18   } else {
19     compressQueue[p].put(item);
20   }
21 }
```

(b) `ferret`: Stages 1 and 2 (optimized implementation).

Figure 4.10: `dedup`: Program transformations.

(line 12–13) are the source of data sharing for `dedup`. Not just data items, but also the key of every data item (initialized in line 11) is added to the hash table. As these keys are used frequently for hash table lookups, the processor that holds each key also plays an important role for data locality.

We change `dedup` to reduce sharing of hash table elements and also of the hash table structure itself; the changes are illustrated in Figure 4.10(b). The set of all hash keys is divided into p non-overlapping subsets, where p is the number of processors in the system. Each subset is (conceptually) mapped to a different processor. As this mapping closely depends on the

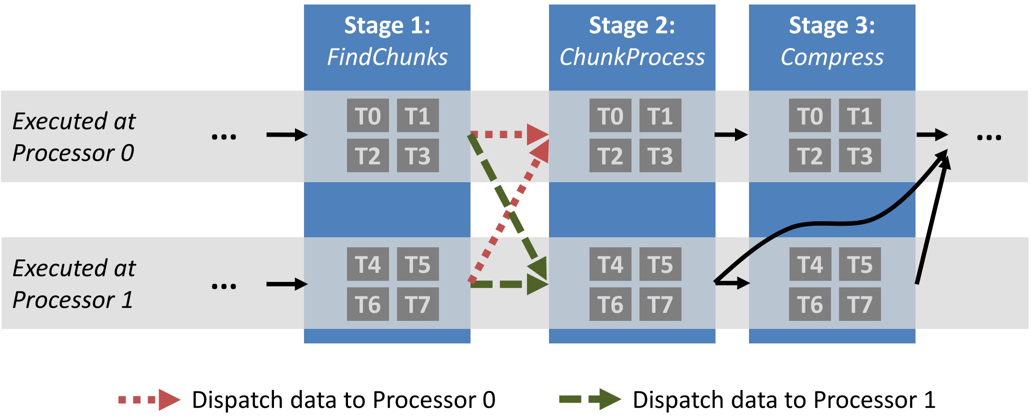


Figure 4.11: `dedup`: Stages 1, 2, and 3 (optimized implementation).

structure hash table, we add a new method to the hash table, `processorForKey()` that determines for each key the processor associated with that key (line 7).

Enforcing the mapping of keys to processors requires data elements to be dispatched to the appropriate processor. We modify Stage 1 of `dedup` to achieve this. The original version of Stage 1 dispatches each data item to the same processor that the data was dequeued from (lines 3 and 6 of Figure 4.10(a)). In the optimized version, however, each data item is dispatched to the processor its key is associated with (lines 7–8 of Figure 4.10(b)). It is sufficient to change Stage 1 to enforce the mapping of keys to processors, because by default Stage 2 and Stage 3 enqueue each data element at the same processor that the data was dequeued from, thus as soon as a data element is enqueued at the appropriate processor by Stage 1, it is processed at the same processor by all subsequent stages.

Using locality-aware dispatching of data elements guarantees data locality for structures related to the hash table but not for keys. As keys are computed before the dispatch to subsequent stages can happen (line 6 of Figure 4.10(b)), the keys are not necessarily allocated at the processor that they are conceptually mapped onto. To enforce the mapping of keys to processors, a clone of each key is created after dispatch (in Stage 2, cloning shown in line 14 of Figure 4.10(b)). As Stage 1 already enforces the previously described key-to-processor mapping, the clones created in Stage 2 are guaranteed to reside on the appropriate processor due to the first-touch page placement policy. Figure 4.11 shows a graphical representation of the locality-aware implementation of stages 1, 2, and 3 of `dedup`, including queuing.

4.5 Performance evaluation

This section evaluates the performance of the program-level transformations proposed in Section 4.4. As transformations effect both data locality and prefetcher performance, we divide the evaluation into two parts. First, in Section 4.5.1 we discuss the *cumulative effect* of the program transformations on data locality *and* prefetcher performance. Then, in Section 4.5.2 we quantify the effect of the transformations on prefetcher performance *individually*.

4.5.1 Cumulative effect of program transformations

This section evaluates the effect of program transformations on both data locality and prefetcher performance *cumulatively*, therefore the prefetcher units of the processors are turned on for the measurements presented in this section.

The proposed program transformations influence both aspects of a program's mapping, that is, transformations change both the program's schedule of computations and its data distribution. The schedule of a program's computations is defined by the way a program's threads are mapped to the processors/cores of a system and by the way computations are mapped to threads. Threads are mapped using affinity scheduling with identity mapping in all experiments. Computations are mapped to threads by the rules defined by OpenMP static scheduling [80] (in the case of `streamcluster`) or by the way pipeline stages are interconnected (in case of `ferret` and `dedup`).

The schedule of computations effects caching and prefetcher performance, the distribution of data effects the locality of main memory accesses. To be able to delimit the effect of a transformation on each level of the memory system, we consider data distribution *separately* from the rest of changes involved by a transformation. More specifically, we evaluate performance with a set of different execution scenarios. Each execution scenario defines

- the version of the program used (transformed or original), but without any consideration of the distribution of data, and
- the program's data distribution, for example, the page placement policy used by the program independent of other source-level changes.

We consider four execution scenarios:

- *original (first-touch)* This execution scenario uses the original, out-of-the-box version of the programs. The memory allocation policy used is first-touch (the default on many modern OSs). In the figures this scenario appears as *original (FT).*
- *original (interleaved)* This execution scenario also uses the original version of the programs, but memory regions are allocated using the interleaved page placement policy. The *interleaved page placement policy* distributes pages of shared memory regions across the processors of the system in a round-robin fashion. Interleaved page allocation balances the load between the memory controllers of a NUMA system and thus mitigates the performance degradation caused by shared data regions being allocated at a single processor and was thus proposed by recent work [27, 55]. Using interleaved page placement is equivalent to disabling NUMA in our systems (in the BIOS). For this scenario we use the shorthand *original (INTL)* in the figures.
- *transformed (interleaved)* This scenario evaluates the proposed program transformations with the interleaved page placement policy. As per-processor memory allocation is not used in this scenario, performance differences are only caused by the combined effect of improved caching and prefetcher performance. In the figures this scenario appears as *transformed (INTL).*

- *transformed (NUMA-alloc)* This scenario evaluates the program transformations with per-processor memory allocation enabled. This scenario has the additional benefit of local main memory accesses relative to the previous, transformed (interleaved), execution scenario. In the figures this scenario appears as *transformed (NA)*.

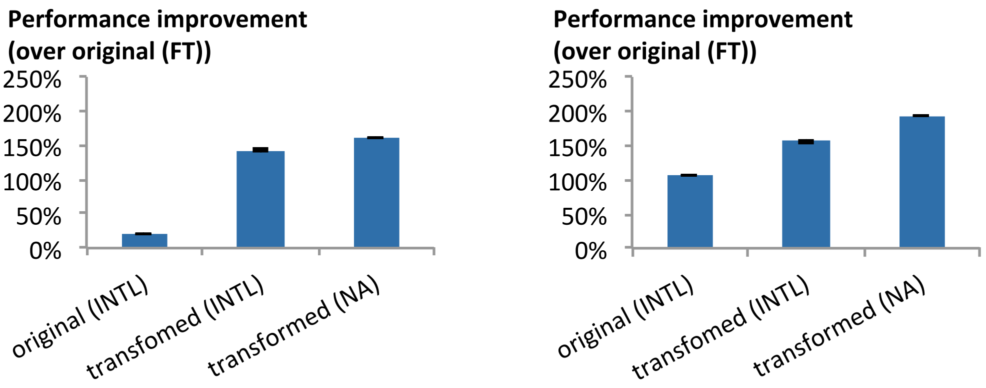
We compare the performance of the three benchmarks, `streamcluster`, `ferret` and `dedup` in all execution scenarios on both machines described in Section 4.2 (2-processor 8-core and 4-processor 32-core NUMA). Any *two adjacent execution scenarios differ in only one parameter*, thus comparing any two adjacent scenarios quantifies the exact cause of performance improvement. Besides reporting performance we quantify the effect of the transformations on the memory system by providing a breakdown of total uncore memory transfers for all configurations of the benchmark programs. (A processor's uncore includes the processor's last-level cache, its on-chip memory controller, and its interfaces to the cross-chip interconnect, thus at the uncore level we can distinguish between local/remote cache accesses and local/remote main memory accesses.) In some cases the total number of uncore transfers differs when the same program binary is executed on two different machines but in the same execution scenario. Although the two machines used for evaluation have the same microarchitecture, the sizes of the last-level caches differ between the two machines. The different size of the last-level caches explains the difference of the total number of total uncore transfers measured.

`streamcluster`

Figures 4.12(a) and 4.12(b) show the performance improvement of three scenarios, original (interleaved), transformed (interleaved), and transformed (NUMA-alloc), over the default setup (original (first-touch)). Figures 4.13(a) and 4.13(b) show the breakdown of all uncore transfers for all the scenarios considered.

By default the complete coordinates of all data points accessed by `streamcluster` are allocated at a single processor. As all threads access these data, the memory controller and cross-chip interconnect of this processor are overloaded. Using the interleaved page placement policy instead of the first-touch page placement policy reduces neither the fraction of remote cache references nor the fraction of remote main memory accesses relative to the first-touch page placement policy. However, as the interleaved policy distributes the pages of the image database across the processors of the system in a round-robin fashion, the load on the memory system is distributed between all memory controllers/cross-chip interconnects of the system. Using interleaved page placement for `streamcluster` has been previously suggested by Lachaize et al. in [55] and it results in a performance improvement of 21% (106%) relative to the default setup on the 8-core (32-core) machine.

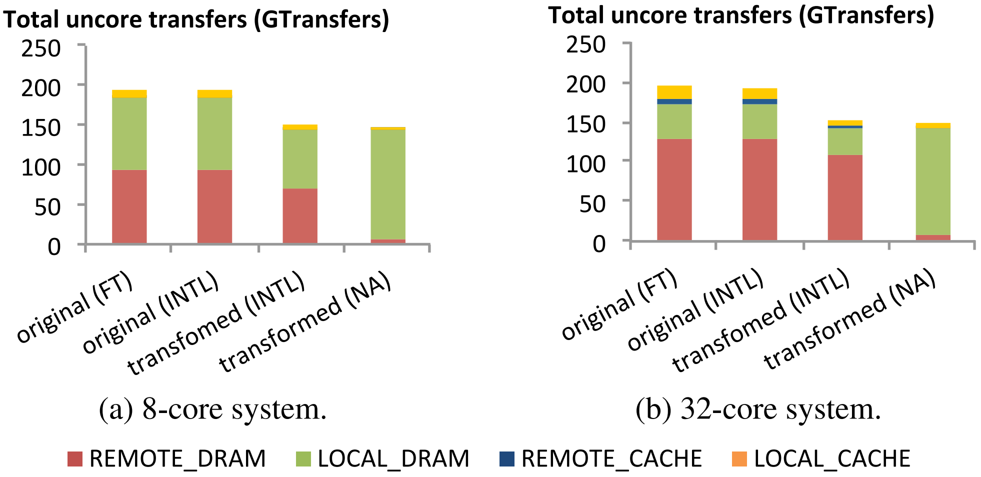
The next scenario considered is program transformations with interleaved page placement (transformed (INTL)). The total number of transfers executed by the program decreases relative to the previous scenario. This result indicates better cache usage. In addition, prefetchers are also able to better predict the memory accesses of the program (see Section 4.5.2). The combined result of these two effects yields an additional performance improvement of 121% (50%) on the 8-core (32-core) machine. If the data distribution of the program is also set appropriately (i.e., by using per-processor memory allocation), we record an almost complete elimination of remote main memory accesses. This setup is reported in the transformed (NA) execution scenario. For the transformed (NA) scenario we observe a performance improvement of 18%



(a) 8-core system.

(b) 32-core system.

Figure 4.12: `streamcluster`: Performance improvement (over *original (FT)*).



(a) 8-core system.

(b) 32-core system.

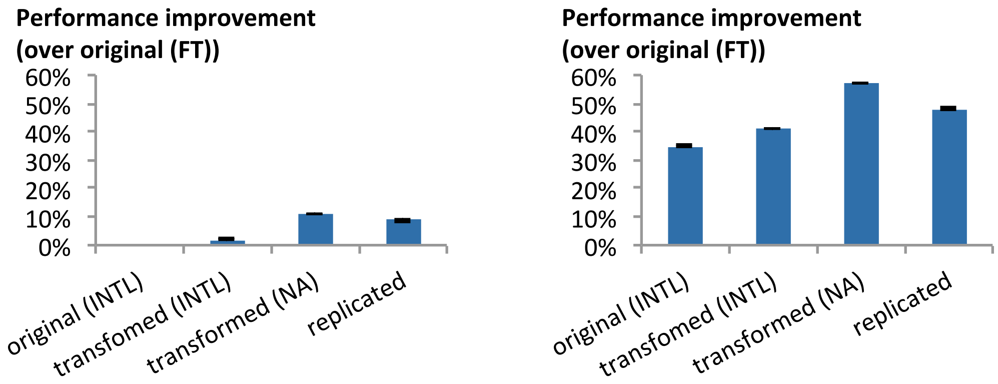
Figure 4.13: `streamcluster`: Uncore memory transfers.

(38%) on the 8-core (32-core) machine, in addition to the program transformations (interleaved) scenario. In the end, the algorithmic changes coupled with per-processor memory allocation improve performance by 2.6X (2.94X) over the out-of-the-box version of the `streamcluster` benchmark on the 8-core (32-core) machine.

`ferret`

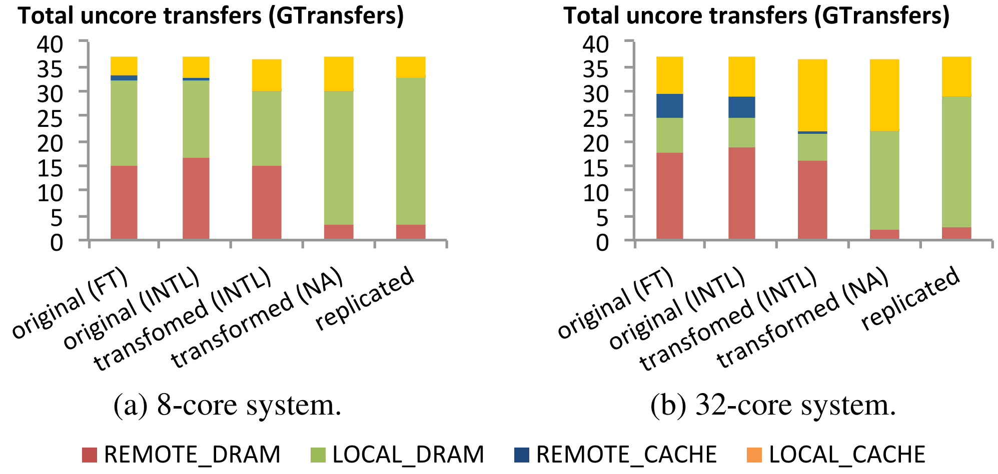
Figures 4.14(a) and 4.14(b) compare the performance of `ferret` in the four execution scenarios considered, Figures 4.15(a) and 4.15(b) present the breakdown of uncore transfers corresponding to each scenario.

With the default setup a large fraction of `ferret`'s memory accesses hit in the remote cache or are served by remote main memory, thus `ferret` experiences large performance penalties due to increased memory access latencies. By default, the complete image database is allocated at a single processor. As all threads use the database, the memory controller and cross-chip interconnect of this processor are overloaded. The interleaved policy balances the load between all memory controllers/cross-chip interconnects of the system, thus interleaved allocation results in a performance improvement of 35% relative to the default setup on the



(a) 8-core system. (b) 32-core system.

Figure 4.14: `ferret`: Performance improvement (over *original (FT)*).



(a) 8-core system. (b) 32-core system.

Figure 4.15: `ferret`: Uncore memory transfers.

32-core machine.

On the 8-core machine we do not observe any performance improvement with interleaved allocation. By default, the image database is allocated at a single processor and is therefore accessed through a single memory controller. In the 8-core machine less cores/processors (8 cores/2 processors) access the image database than on the 32-core machine (32 cores/4 processors); therefore the memory controller used to access the database must handle more load in the 32-core machine than on the 8-core machine. As a result, distributing load across all memory controllers of the system this load gives more benefit on the 32-core machine than on the 8-core machine.

The next execution scenario considered is the transformed version of the program used with interleaved memory allocation. With this execution scenario each processor accesses only a subset of the complete image database, thus costly remote cache accesses are almost completely eliminated. If the image database is shared between processors, each processor attempts to cache the complete image database, which can result in a cache line being loaded into multiple caches at the same time. If data sharing is eliminated, each processor caches a distinct subset of the image database, therefore there is more cache capacity available to the program.

Measurements show that the cache hit rate of the program increases due to the program transformations; this results in an additional 2% (6%) performance improvement relative to the original (interleaved) execution scenario. Distributing pages appropriately at the processors of the system (the transformed (NA) execution scenario) reduces the fraction of remote main memory references of the program and thus further improves performance by 9% (16%) on the 8-core (32-core) machine. In summary, the performance of `ferret` improves 11% (57%) on the 8-core (32-core) machine.

In the case of `ferret`, another optimization opportunity is to replicate the shared image database at each processor. Performance results for this optimization are reported as *replicated* in Figures 4.14(a) and 4.14(b) (uncore transfers are reported in Figures 4.15(a) and 4.15(b)). Replication of the shared image database reduces the fraction of remote memory references, but it also reduces the effectiveness of caching because each processor's last-level cache is used for caching a different replica. The results of replication are inferior to transformed (NA), which gets the benefits of both caching and few remote memory accesses.

Replication is relatively simple to implement in case of `ferret`, because the performance-critical image database is read-only. In the general case (read-write accesses to replicated data), replication would require synchronizing the replicas, which can be complicated to implement and adds overhead (to limit the overhead, recent systems disable replication after a fixed number of synchronization operations, e.g., after 5 synchronization operations in [27]). As the other two benchmark programs both read and write their critical data structures, we do not evaluate replication for benchmarks other than `ferret`.

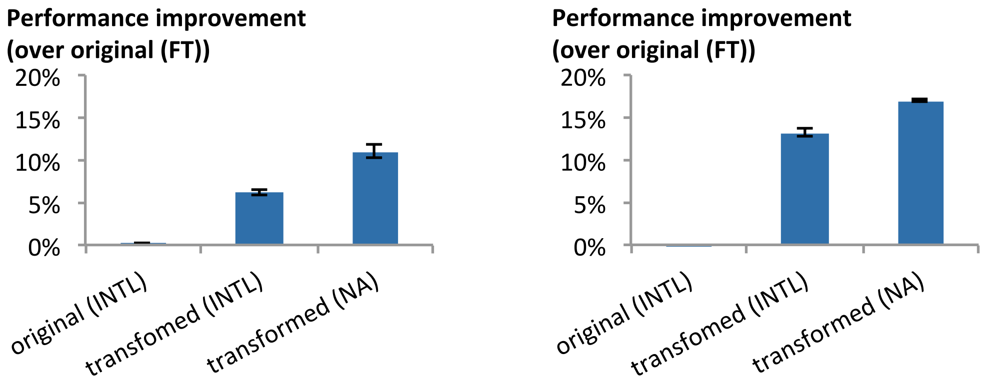
dedup

Figures 4.16(a) and 4.16(b) compare the performance of the four previously discussed execution scenarios for the `dedup` benchmark. Figures 4.17(a) and 4.17(b) show the uncore traffic breakdown for each configuration.

The shared hash table is constructed during program execution, and all threads add new file chunks to the hash table as deduplication progresses. As a result, with the first-touch allocation policy the hash table is spread across the processors of the system. Therefore, all memory controllers/cross-chip interconnects of a system are used to access these data structures and none of the interfaces to memory are overloaded. As a result, using the interleaved page placement policy does not significantly change performance over using the first-touch policy. However, the transformed version of the program uses locality-aware dispatching of data to processors and so each processor accesses only a subset of the globally shared hash table. As a result, the fraction of remote cache accesses (and also the amount of total uncore transfers) decreases on both machines. Due to better caching, performance increases by 6% (13%) on the 8-core (32-core) machine. In the last execution scenario considered (transformed with NUMA-aware allocation), `dedup` experiences fewer remote main memory transfers, which results in an additional improvement of 5% (4%) on the 8-core (32-core) machine. In summary, the performance of `dedup` improves 11% (17%) on the 8-core (32-core) machine over the default.

4.5.2 Prefetcher performance

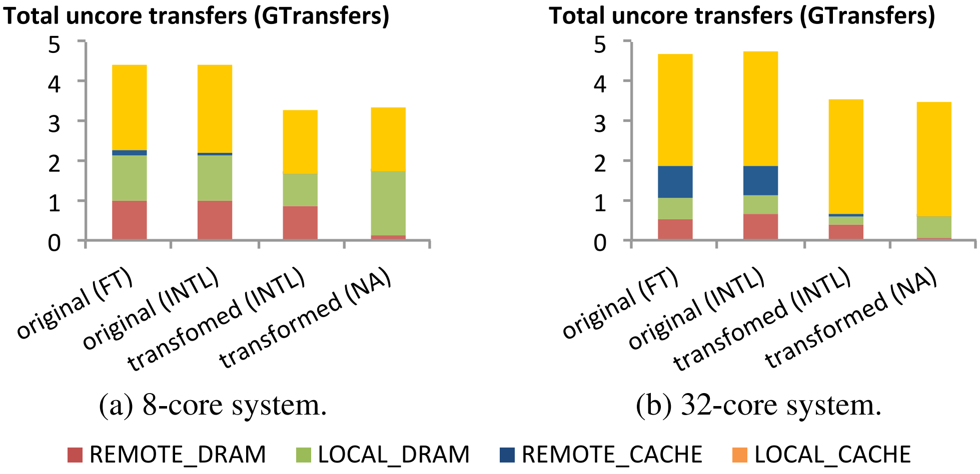
The previous section evaluated in detail the performance benefits of the proposed program transformations, however, it did not clearly identify the benefits due to prefetching effects, because



(a) 8-core system.

(b) 32-core system.

Figure 4.16: `dedup`: Performance improvement (over *original (FT)*).



(a) 8-core system.

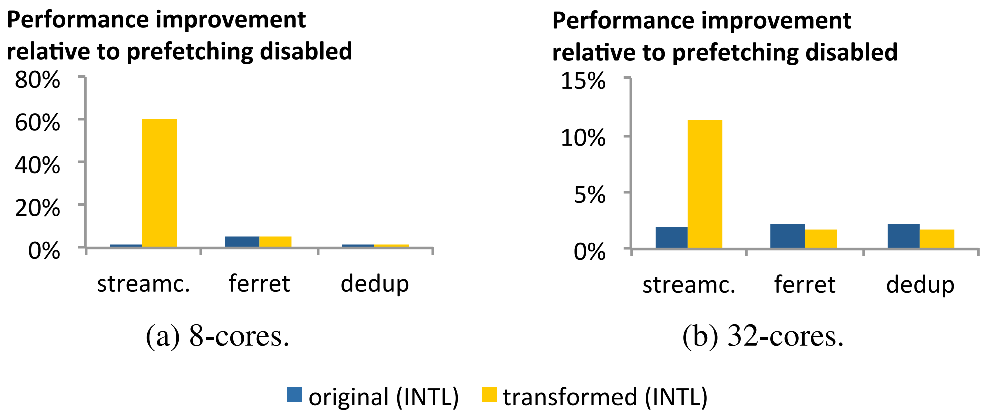
(b) 32-core system.

Figure 4.17: `dedup`: Uncore memory transfers.

the transition from the original (INTL) execution scenario to the transformed (INTL) shows improvements due to both caching and prefetching effects.

To quantify how much performance benefit is due to the more regular access patterns caused by the source-level transformations, we perform a simple experiment: we execute each benchmark with the processor prefetcher units disabled and then with prefetching enabled. In the end we compare the performance of the two cases. The results are shown in Figure 4.18(a) (4.18(b)) for the 8-core (32-core) machine.

For the original version of the program (executed in the original (INTL) scenario) there is a moderate performance improvement due to prefetching (at most 5%). The transformed version of the program (executed in the transformed (INTL) scenario), however, shows performance improvements of 60% (11%) due to prefetching for the `streamcluster` benchmark on the 8-core (32-core) machine. For the other two benchmarks the benefit of prefetching is roughly the same as with the original version of the program. As both programs use a hash table-based data structure intensively, they both have irregular memory access patterns. The hash table is key to the algorithms implemented by the programs, similarity search based on locality-sensitive hashing [63] in case of `ferret`, and data deduplication in case of `dedup`. Replacing hash tables with a different data structure would change the essence of these programs and requires



(a) 8-cores.

(b) 32-cores.

Figure 4.18: Prefetcher performance.

more significant changes than those described in this chapter.

4.6 Conclusions

4.6.1 Implications for performance evaluation

Program-level data sharing and irregular memory access patterns are two factors with a large influence on application performance on NUMA-multicore systems. The PARSEC benchmark programs, originally developed for earlier generations of symmetric multiprocessors, show sharing and access patterns that inhibit performance on modern platforms with advanced architecture features such as prefetch units. These factors must be taken into account when evaluating performance.

By using the PARSEC programs out-of-the-box on recent NUMA-multicore systems, researchers might be misled to conclude that the microprocessor design, the compiler, or the runtime system are faulty when actually the application software contains performance-limiting factors to be blamed for unsatisfactory performance. The measurement methodology presented in this chapter allows for pinpointing software-related problems detrimental to NUMA-multicore memory system performance. Furthermore, we show that in many cases minor program-level adjustments suffice to fix these problems.

There are scenarios when it is important to use an out-of-the-box version of an application for performance evaluation. But when, for example, in a research setting, new features are to be evaluated, it is essential to identify limitations of an application's implementation that stand in the way of meaningful performance measurements. Publishing not only the results of the evaluation but also the software-level changes required to achieve them is necessary to fully understand a workload's or a system's performance characteristics.

4.6.2 Implications for performance optimizations

Our experience with multithreaded programs from the PARSEC suite shows that programs can be adjusted with minimal effort to fit modern memory system architectures. The transformations result in better data locality (fewer accesses to remote data), better use of the caches, and increased effectiveness of the prefetch unit. Simple source-level changes result in high performance improvements (up to 2.9X) for three benchmark programs on recent NUMA-multicores. The chapter reports detailed measurements of orthogonal experiments to quantify how these changes individually effect each distinct layer of a modern NUMA-multicore memory system.

Most program transformations described in this chapter improve the locality of cache- and main memory accesses. We compare data locality optimizations with other optimization techniques proposed in the literature (e.g., interleaved memory allocation and data replication [27]). We find that in most cases optimizing multithreaded programs for data locality can result in better performance than any of alternative techniques we looked at. As we expect the performance gap between local and remote accesses to widen (with an increased number of processors and larger caches), it is important that software developers have the tools and insight necessary to analyze and understand problems related to data locality. Moreover, as the proposed source-level changes are specific to the PARSEC programs, more general approaches to implement data locality optimizations are worthwhile to be considered for investigation.

5

Matching memory access patterns and data placement

The locality of a multithreaded program's memory accesses can critically influence a program's performance. The previous chapter shows that in some cases minor source-level changes suffice to achieve good data locality and thus improve performance. Moreover, by applying different manual optimization techniques to a single program we are able to better understand the way each technique effects the memory system. Nevertheless, the optimization techniques proposed in the previous chapter are ad-hoc, that is, they are specific to the programs considered.

This chapter investigates more systematic approaches to achieve good data locality for loop-parallel programs. In the first part of the chapter we analyze the memory system behavior of multithreaded scientific computations of the NAS Parallel Benchmark (NPB) suite [48]. Experiments show that in many of these computations the threads access a large number of memory locations, and we characterize the applications with regard to their access patterns. A consequence of these access patterns is that it is difficult to obtain good data locality (and thus good performance) on NUMA systems. The experiments show furthermore that data migration (both profile- and program-based migration) and locality-aware iteration scheduling can in some cases soften the problems caused by global accesses, however, in many cases the overhead and/or the complexity of using these mechanisms cancels the benefits of good data locality (or worse).

The mismatch between the distribution of data across processors and the accesses to these data as performed by the processors' cores is immediately felt by users of popular parallelization directives like OpenMP. These directives allow partitioning of computations but do not allow a programmer to control either the mapping of computations to cores, or the placement of data in memory. In the second part of this chapter we present a small set of user-level directives that, combined with simple program transformations, can almost completely eliminate remote memory accesses for programs of the NPB suite and thus result in a performance improvement of up to 3.2X over the default setup in a 4-processor 32-core machine.

5.1 Memory system behavior of loop-parallel programs

5.1.1 Experimental setup

This section presents an analysis of the memory behavior of data-parallel programs of the NPB suite version 2.3 [48]. Table 5.1 shows a short characterization of the NPB programs. The programs are executed on the 2-processor 8-core Nehalem-based machine configured with 12 GB of RAM (see Section 2.1.1 for further details about the machine).

Benchmark	Class	Working set size	Run time
`bt`	B	1299 MB	125 s
`cg`	B	500 MB	26 s
`ep`	C	72 KB	85 s
`ft`	B	1766 MB	19 s
`is`	B	468 MB	10 s
`lu`	C	750 MB	368 s
`mg`	C	3503 MB	33 s
`sp`	B	423 MB	82 s

Table 5.1: Benchmark programs.

The benchmark programs are configured to execute with 8 worker threads (threads T0–T7) for all experiments described in this chapter (if not explicitly stated otherwise). The number of worker threads is equal to the total number of cores in the system to allow a program to use the full computational power of the Nehalem-based system. Similarly to the setup described in Section 4.2.3 for the PARSEC programs, the thread-to-core mapping of the worker threads is set to *identity affinity* for all experiments. Identity affinity maps each worker thread to the core with the same number as the thread's number (e.g., Thread T0 is mapped to Core 0). As threads are mapped to cores with identity affinity, the terms core and thread are used interchangeably. Fixing the thread-to-core affinity disables OS reschedules and thus measurement readings are more stable. Additionally, as the programs of the NPB suite are data-parallel, using thread-to-core affinities other than identity affinity changes neither the cache hit rate nor the performance of the programs (as noted previously by Zhang et al. [112]). In NUMA systems, memory allocation happens on a per-processor basis. Therefore, we refer to processors when we describe the distribution of data in the system: Processor 0 contains the memory local for threads T0–T3, and Processor 1 contains memory local for threads T4–T7.

Figure 5.1 shows a breakdown of the total measured memory bandwidth into local and remote memory bandwidth for all benchmark programs of the NPB suite. We use hardware performance counters to perform the measurements, the configuration is the same as in Section 4.2.4. For the measurements the *first-touch* page placement policy is in effect. This policy places every page at the processor that first reads from/writes to this page after page allocation. We refer to the combination of the first-touch policy and identity affinity as *default setup*.

As shown in Figure 5.1, the total memory bandwidth generated by `ep` is negligible. As a result, the performance of this program does not depend on the memory system, and the program is excluded from further investigation. The `is` and `mg` programs are excluded as well because both programs exhibit a low percentage of remote memory accesses with the default setup (on average 3% and 2% of the total bandwidth, respectively). Nonetheless, the other benchmarks generate a significant amount of bandwidth (up to 23 GB/s) and also show a significant contribution of remote memory accesses (11%–48% of the total bandwidth). These high percentages suggest that there is a need for approaches that reduce the number of remote memory accesses.

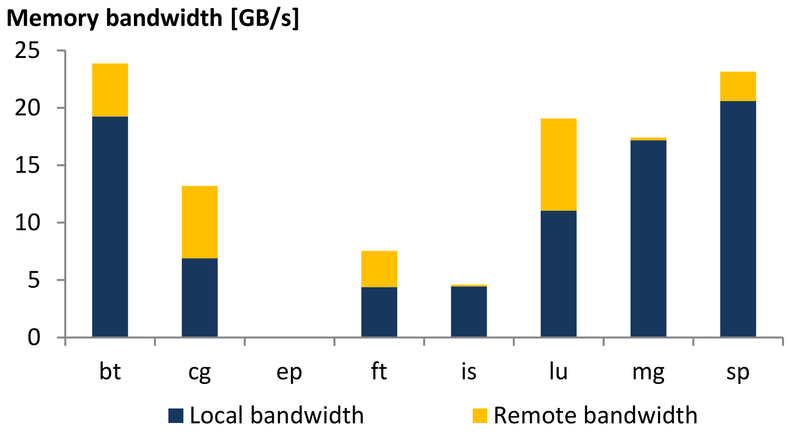


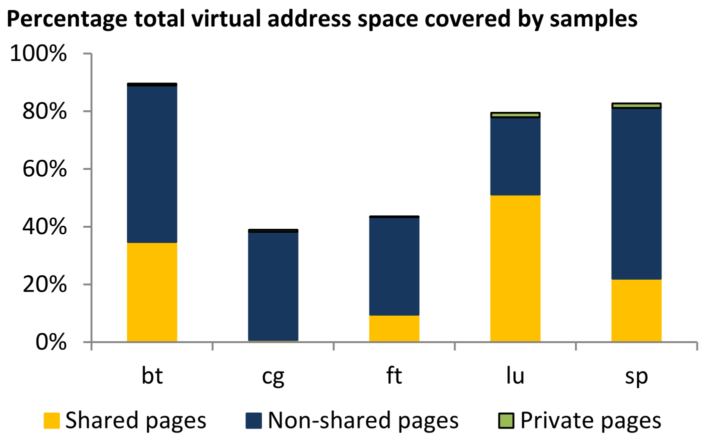
Figure 5.1: Memory bandwidth generated by the programs of the NPB suite.

5.1.2 Data address profiling

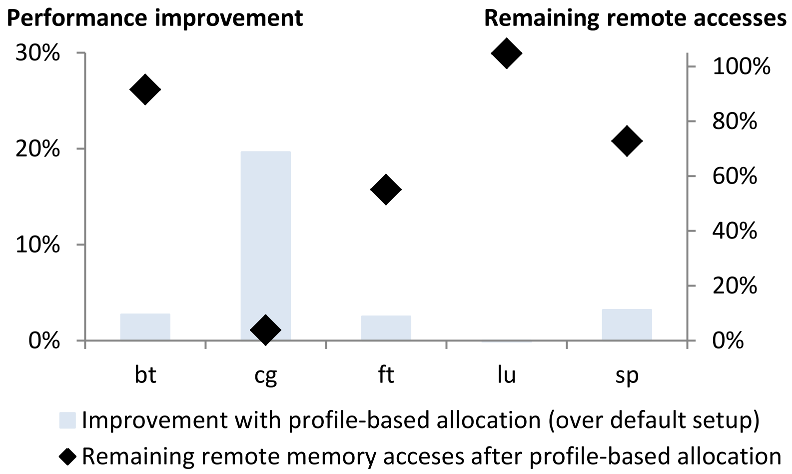
To explore the memory performance issues in a limit study, we profile the execution of the NPB programs using the latency-above-threshold profiling mechanism of the Intel Nehalem microarchitecture. As already described in Section 4.3.1, this performance-counter-based mechanism samples memory instructions with access latencies higher than a predefined threshold and provides detailed information about the data address used by each sampled instruction. Based on the sampled data addresses it is straightforward to estimate the number of times each page of the program's address space is accessed by each core of the system. To account for accesses to all levels of the memory hierarchy, we set the latency threshold to 4 cycles (accesses to the first level cache on the Nehalem-based system have a minimum latency of 4 cycles). We use a sampling rate of 1. The profiling technique is portable to many different microarchitectures, because most recent Intel microarchitectures support latency-above-threshold profiling and AMD processors support Instruction-Based Sampling [13], a profiling mechanism very similar to that of Intel's. Moreover, on architectures without hardware support for data address profiling, address profiles can be collected using software-only techniques as well [62].

As memory access profiling on the Intel Nehalem is sampling-based, not all pages of the program's address space appear in the profiles. Figure 5.2(a) shows for each program the percentage of the virtual address space covered by samples. A page of the address space is covered if there is at least one sample in the profiles to an address that belongs to the page in question. Samples cover a large portion of the address space (up to 90%). The coverage is low in the case of `cg` because around 50% of the program's address space is occupied by data structures that store temporary data. As these structures are used only during the initialization phase of the program, very few samples are gathered for pages that store these structures.

A program's address space can be divided into three categories. The first category contains *private* pages that belong to each thread's stack. As the stack is private for each thread, the first-touch policy (remember that we profile program execution with the default setup) can allocate the pages for thread-private stacks on each thread's processor. The first category also contains pages that are reserved for dynamically linked shared objects and the program's image in memory. These latter regions are small and are shared by all threads. As not much can be done for the first category of pages as far as data placement is concerned, these pages are excluded from further analysis and optimizations.



(a) Sampled and covered address space.



(b) Performance improvement; remaining remote accesses.

Figure 5.2: Data address profiling.

Most data accessed by the computations is allocated on the heap, which holds the second and third category of pages. Some heap pages are exclusively accessed by a single thread, thus these pages are *non-shared*. Other heap regions, however, are accessed by multiple threads, and are thus *shared*. Figure 5.2(a) shows the percentages of the address space that belong to the three categories previously described. The NPB programs we consider exhibit various degrees of sharing, ranging from almost no sharing (e.g., `cg`) to intense sharing when up to 51% of the program address space is accessed by multiple threads (e.g., `lu`).

5.1.3 Profile-based page placement

Profiles provide a great deal of information about the access patterns of a program, but the usefulness of profiles for optimizing data locality remains to be determined. To optimize data locality we adapt the profile-based page placement approach described by Marathe et al. [67, 68], and we use a simple heuristic to determine for each page the processor where the page should be allocated at: On a profile-based run of a benchmark we allocate each page on the processor whose cores accessed the page the most frequently. Figure 5.2(b) shows the performance improvement over the default setup when profile-based memory allocation is used. The performance measurements include the overhead of reading the profiles and placing data into memory. Profile-based allocation performs well for only the `cg` benchmark, which improves 20%; the remaining benchmarks show minor or no performance improvement over the default setup.

The same figure (Figure 5.2(b)) shows also the percentage of the programs' default remote memory accesses remaining after profile-based allocation. The performance improvement correlates well with the reduction of the remote memory traffic measured with the default setup. For `cg` (the program with the largest performance improvement) a small percentage of the default remote memory accesses remains after profile-based allocation. For programs with little or no performance improvement (e.g., `lu`) remote memory traffic is the same as with the default setup, because profile-based memory allocation was not able to determine a distribution of pages in memory that reduces or removes remote memory traffic (that was originally observed with the default setup).

The information about the number of shared pages (Figure 5.2(a)) provides additional insights into the performance improvement due to profile-based allocation (Figure 5.2(b)): the `cg` benchmark, which has a small fraction of shared pages, improves with profile-based memory allocation [1], but the other benchmarks, which have a high fraction of shared pages, do not. Benchmarks with high degree of sharing have the same number of remote memory accesses both with profile-based memory allocation and the default setup (Figure 5.2(b)). These measurements show that profile-based allocation cannot correctly allocate exactly those pages that are shared between multiple processors. In this chapter we describe various techniques to improve the performance of programs with a high degree of data sharing.

5.2 Memory access and distribution patterns: A detailed look

NUMA systems introduce another aspect into the problem of managing a program's data space. The program's computation determines the memory locations that are accessed; directives (e.g., OpenMP) or compilers and the runtime system determine how computations are partitioned respectively mapped onto cores. A simple characterization of memory access patterns provides a handle to understand the memory behavior of programs. We start with the memory allocation and data access patterns that frequently appear in scientific computations (and thus in the NPB programs). Then we analyze the ability of different mechanisms (in-program data migration and loop iteration distribution) to improve data locality.

5.2.1 In-memory representation of matrices

Many scientific computations process multidimensional matrices. Figure 5.3 shows a three-dimensional matrix stored in memory (in row-major order, addresses increase from left to right and from top to bottom, respectively). The matrix contains `NX = 5` rows (the size of the matrix along the x-dimension, shown vertically in the figure). Each row contains `NY = 4` columns

---

[1] The `cg` benchmark touches most of its data for the first time during the initialization phase of the benchmark. The initialization phase of the benchmark is sequential; as a result, most of the benchmark's data is allocated at a single processor (i.e., the processor where the initialization phase is executed). In the computational phase of the program, however, each thread predominantly accesses a disjoint subset of the benchmark's data. As a result, profile-based can place most pages of the benchmark at the appropriate processor.

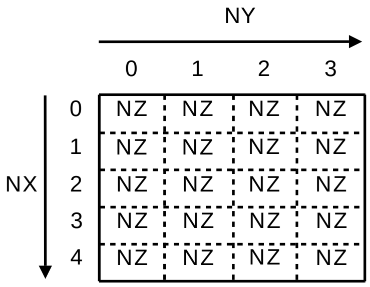


Figure 5.3: Memory layout of 3D matrix (row-major order).

(the size of the matrix along the y-dimension, shown horizontally). As the matrix has a third, z-dimension, the matrix contains a block of `NZ` items for each `(x,y)` element. The memory layout of matrices with dimensionality higher than three is analogous to the three-dimensional case, therefore we do not discuss such matrices in detail. Moreover, to keep the following discussion simple, we focus on two-dimensional matrices.

5.2.2 Matrix memory access patterns

When we study processor memories that are accessed by threads of scientific computations, we encounter two frequent data access and data distribution patterns for two-dimensional matrices. We call these the *x-wise* and *y-wise* pattern, respectively.

x-wise pattern Figure 5.4(a) shows a simple loop that accesses all elements of a two-dimensional matrix. In this example the outermost loop (the loop that iterates along the x dimension of the matrix) is parallelized with a standard OpenMP `parallel for` construct. The loop iterations are distributed across all worker threads (here: eight threads). Each thread gets a chunk of size `D = NX / 8` of the total iteration space (assuming that `NX` is an integer multiple of 8). Figure 5.4(c) shows the assignment of iterations to worker threads for the code shown in Figure 5.4(a). Each thread T0 to T7 accesses `D` non-overlapping rows of the matrix. Figure 5.4(c) shows the distribution of the matrix's memory in the system for an x-wise data access pattern. As we map threads to cores with identity affinity, the first-touch policy allocates the first `4 × D` rows at Processor 0 and the second `4 × D` rows at Processor 1. We call this distribution of the matrix in memory an x-wise data distribution.

y-wise pattern If the second `for` loop (the one that sweeps the y-dimension) is parallelized (as shown in Figure 5.4(b)), the result is a different, y-wise, data access and distribution pattern. Figure 5.4(d) shows both the allocation of the iteration space to worker threads and the distribution of the matrix memory to processors (based on a first-touch allocation). With the y-wise access pattern, the first `4 × D` columns are allocated at Processor 0, and the second `4 × D` columns are allocated at Processor 1.

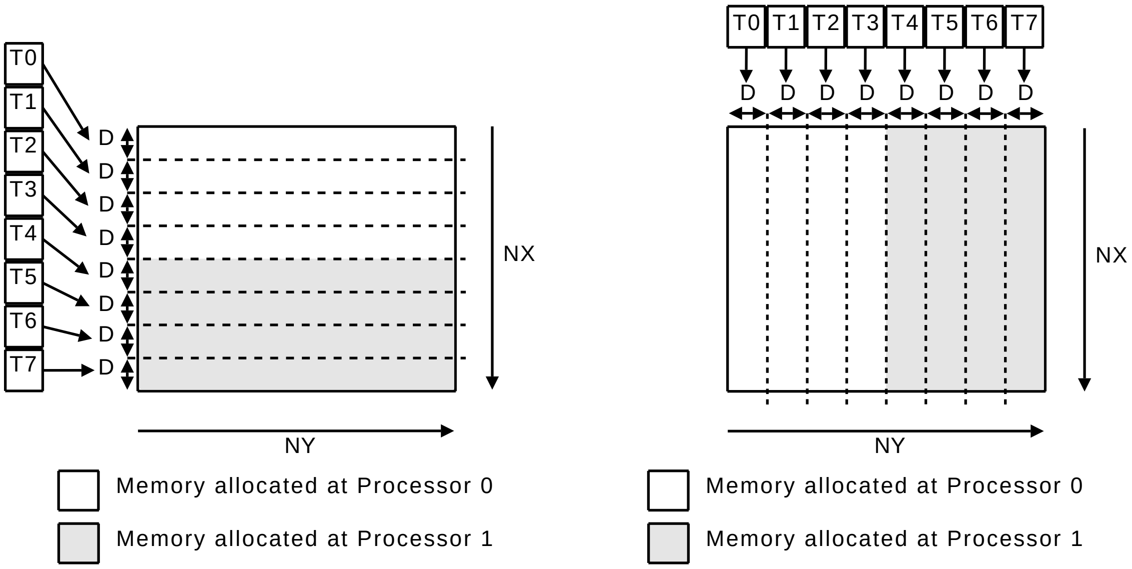
Some programs that operate on data with a dimensionality higher than two can have higher order access patterns. For example, in the case of a z-wise access pattern the iteration and data distribution is analogous to the x-wise and y-wise patterns, respectively. The only difference is that the blocks of data are distributed among the threads and processors of the system along the z-dimension.

```
#pragma omp parallel for
for (i = 0; i < NX; i++)
    for (j = 0; j < NY; j++)
        // access m[i][j]
```

(a) x-wise data access pattern.

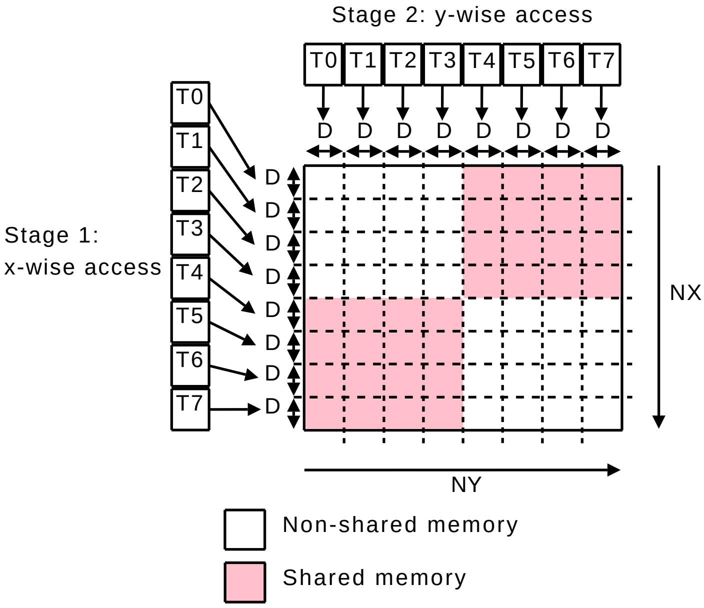
```
for (i = 0; i < NX; i++)
#pragma omp parallel for
    for (j = 0; j < NY; j++)
        // access m[i][j]
```

(b) y-wise data access pattern.



(c) x-wise iteration distribution.

(d) y-wise iteration distribution.



(e) Data sharing.

Figure 5.4: Access and distribution patterns; data sharing.

5.2.3 Data sharing

The problem of many data-parallel programs is that in many cases programs access a single memory region with multiple data access patterns. Let us assume that a two-dimensional matrix is accessed in two stages by eight worker threads (T0–T7). In the first stage (Figure 5.4(c)) the worker threads access the matrix with an x-wise access pattern, followed by a y-wise access pattern in the second stage (Figure 5.4(d)). Figure 5.4(e) shows the iteration distribution (i.e., mapping of rows respectively columns to threads for processing) for both stages: the iteration space distribution of the two stages overlaps.

If we divide the matrix into four equal-sized memory sub-regions (quadrants), then in both stages the upper left and lower right quadrants of the matrix are accessed exclusively by threads T0–T3 and threads T4–T7, respectively. Therefore, these quadrants are non-shared and allocating these quadrants at Processor 0 (Processor 1) guarantees good data locality for threads T0–T3 (T4–T7).

The upper right and lower left quadrants, however, are accessed (and thus shared) by all threads. For our example, shared quadrants constitute 50% of the memory region presented in this example. In this case the shared quadrants of the matrix are accessed equally often by the threads of all processors of the NUMA system. Therefore, the profile-based optimization technique described in Section 5.1 cannot determine the processor memory that should store these quadrants. If the processor of threads T0–T3 is preferred in allocation, then threads T4–T7 must access these regions remotely (or threads T0–T3 access their regions remotely if threads T4–T7 are preferred). For many programs that perform scientific computations data sharing is a major performance limiting factor, as we have previously seen in Section 5.1. In the next section we describe in more depth two real-world examples of data-sharing, the `bt` and `ft` benchmarks from NPB.

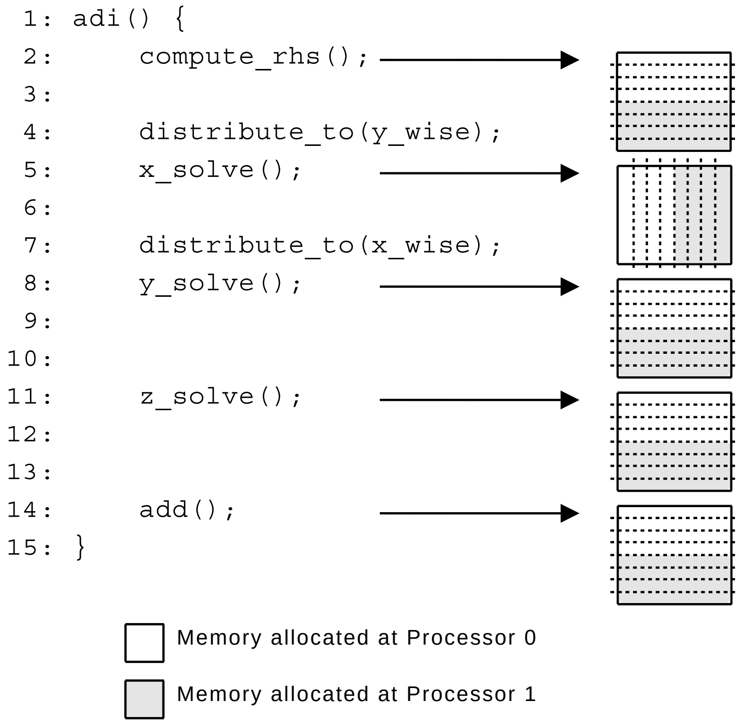
5.2.4 Two examples: `bt` and `ft`

The `bt` benchmark is a 3D computational fluid dynamics application [48] that iteratively solves the Navier-Stokes partial differential equations using the alternating direction implicit method (implemented by the `adi()` function in Figure 5.5). The `adi()` function calls several other functions (also listed in the figure). These functions operate mostly on the same data, but they have different access patterns: The functions `compute_rhs()`, `y_solve()`, `z_solve()`, and `add()` have an x-wise access pattern, but the function `x_solve()` accesses memory y-wise. Moreover, as the `adi()` function is also used for data initialization in the first stage of `bt`, some of the regions (the ones that are first touched in `compute_rhs()`) are laid out x-wise in memory, and some of the regions (first used in `x_solve()`) are laid out y-wise. As a result of this mismatch in data access and memory allocation patterns a high percentage of this program's virtual address space is shared (35%), and the program has a high percentage of remote memory accesses (19%).

The `ft` benchmark implements a spectral method that is based on a 3D Fast Fourier Transform (FFT). The memory access patterns of `ft` are similar to that of `bt`: the `ft` benchmark also alternates between accessing memory with x-wise and y-wise access patterns. `ft` shows less sharing than `bt` (9% of the program's virtual address space is shared vs. 19% sharing), but `ft` has a larger fraction of remote main memory references than `bt` (41% vs. 35%).

```
adi() {
    compute_rhs(); // x-wise pattern
    x_solve();     // y-wise pattern
    y_solve();     // x-wise pattern
    z_solve();     // x-wise pattern
    add();         // x-wise pattern
}
```

Figure 5.5: `bt` data access patterns.

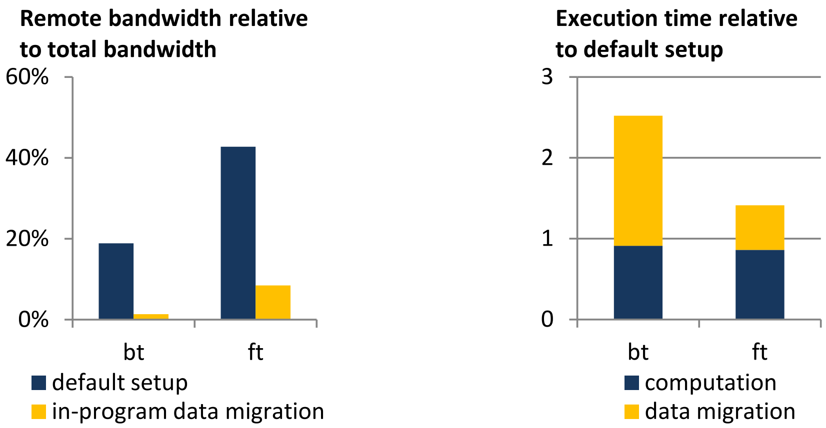


```
 1: adi() {
 2:     compute_rhs();
 3:
 4:     distribute_to(y_wise);
 5:     x_solve();
 6:
 7:     distribute_to(x_wise);
 8:     y_solve();
 9:
10:
11:     z_solve();
12:
13:
14:     add();
15: }
```

Figure 5.6: `bt` with in-program data migration.

In-program data distribution changes

A simple approach to reduce the fraction of remote memory accesses of programs with data sharing is to change the distribution of the program's data on-the-fly so that it matches each different access pattern that appears during program execution. A similar approach has been described by Bikshandi et al. for a program that calculates the FFT of 3D data in a cluster context [11]. To evaluate the effectiveness of this approach in shared-memory systems we insert into the code of the `bt` program calls to functions that change the data distribution in the system (lines 4 and 7 in Figure 5.6). These functions are based on the migration primitives offered by the OS (see Section 5.3 for more details about the implementation). The data distribution of the regions accessed by `x_solve()` is changed to y-wise data distribution before the call to `x_solve()` and then changed back to x-wise after the `x_solve()` function completes. The `adi()` function is executed in a tight loop; therefore, starting with the second iteration of the loop, the `compute_rhs()` function will encounter the correct, x-wise, data distribution it requires (due to the previous iteration of `adi()`). Similar in-program data distribution



(a) Remote bandwidth.

(b) Performance.

Figure 5.7: Evaluation of in-program data migration.

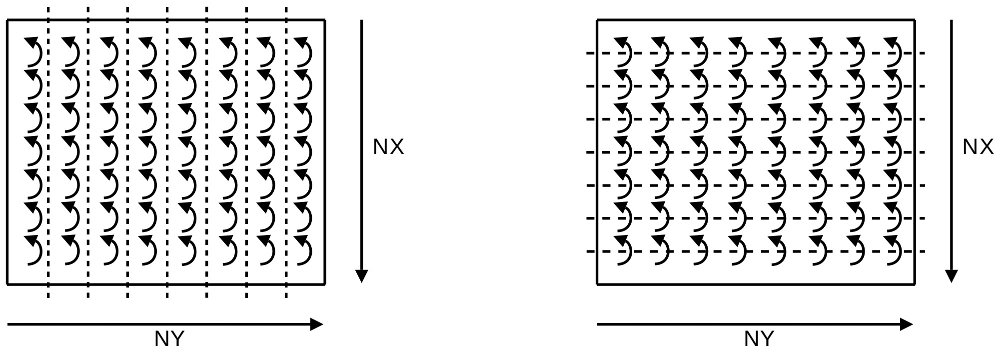
changes can used in case of the `ft` benchmark, as the memory access pattern of the `ft` program alternates between x-wise and y-wise as well.

Figure 5.7(a) compares the bandwidth distribution of the default setup (first-touch memory allocation) with dynamic in-program data distribution changes for `ft` and `bt`. The figure reports the percentage of remote accesses relative to the total number of accesses, as measured in user-mode. As data migration is performed by the kernel, the bandwidth generated by data migrations is not included in the figure. The measurements show that in-program data distribution significantly improves data locality for these two programs (`bt`: reduction of the percentage remote accesses from 19% to 1%, `ft`: reduction from 43% to 8%). This reduction of the fraction of remote accesses results in a reduction of execution time of the computations, as shown in Figure 5.7(b) (this figure shows execution time relative to the default setup, therefore lower numbers are better). For the computation part, `bt` experiences a performance improvement of around 10%, and `ft` speeds up 16%. However, if we account for the overhead of data migration, the total performance is worse than without data migration (`bt` slows down 2.5X, `ft` slows down 1.4X).

In conclusion, in-program data migration can eliminate remote memory accesses by guaranteeing the data distribution required by each access pattern of the program. However, if access patterns change frequently (as it is in the case of `bt` and `ft`), the overhead of data migration cancels the benefits of data locality. Moreover, data migrations serialize the execution of multithreaded code, because access to the page tables shared by the threads requires acquiring a shared lock (parallel migrations do not result in any improvement relative to serial migrations). As a result, in-program data distribution changes should be avoided for programs with frequent access pattern changes (but data migration could be still a viable option for coarser-grained parallel programs).

Iteration redistribution

Changing the data distribution at runtime causes too much overhead to be a viable option for programs with alternating memory access patterns. As changing the distribution of computations has far less overhead than redistributing data, in this section we explore the idea of changing the schedule of loop iterations so that the access patterns of the program match the



(a) y-wise parallelization. (b) x-wise parallelization.

Figure 5.8: Access patterns of code with y-wise dependences.

distribution of data in memory.

We start with a simple approach: As the majority of the functions executed by `bt` has an x-wise access pattern, we change the distribution of *all* program data to x-wise (at program startup). As a result, most functions experience good data locality and their performance improves. However, as the function `x_solve()` has a y-wise access pattern, the number of remote memory accesses it must perform increases due to the change of the data distribution to x-wise. At the end, the performance of the whole program (`bt`) does not improve. The same approach eliminates some of the remote memory accesses of `ft` (a reduction of the fraction of remote memory references from 43% to 18%), but results only in a small improvement (2%).

As the data distribution of all data structures is now set to x-wise, we attempt to improve performance by changing the memory access pattern of all y-wise loops to x-wise. In case of `bt`, we change the distribution of loop iterations in `x_solve()` to x-wise. Our approach is similar to those described in [51, 58]. This transformation, although it requires significant source-level changes, reduces the percentage of remote memory accesses from 19% to 2%, and improves the performance of `bt` significantly (by 14% relative to the default setup).

Transforming `ft` in the same way as `bt` would, however, require, inserting additional synchronization operations into the program code. Y-wise loops in `ft` have loop-carried dependences along the y-dimension of the processed data. Usually, iterations of loop-parallel code are distributed among worker threads so that loop-carried dependences are within the iteration space assigned to each thread for processing. The reason for that is to keep the number of inter-thread synchronization operations low: if data dependences are within the iteration space assigned to each thread, threads do not need to synchronize. For example, Figure 5.8 shows the distribution of the iteration space for two different parallelizations of a computation that has y-wise data dependences. In the first, y-wise, parallelization (Figure 5.8(a)) the loop dependences do not cross per-thread iteration space boundaries. In the second, x-wise parallelization of the code (Figure 5.8(b)) the data dependences cross iteration space boundaries, thus inter-thread synchronization is required. The overhead of inter-thread synchronization is potentially high and it can potentially cancel the benefit of data locality achieved with redistributing iterations, thus we do not explore this option for `ft`.

```
data_distr_t *create_block_cyclic_distr(
      void *m, size_t size, size_t block_size);
```

(a) Block-cyclic data layout.

```
data_distr_t *create_block_exclusive_distr(
      void *m, size_t size, size_t block_size);
```

(b) Block-exclusive data layout.

```
void distribute_to(data_distr_t *distr);
```

(c) Apply data layout.

Figure 5.9: Data distribution primitives.

5.3 Fine-grained data management and work distribution

In-program data migration can be used to match the data distribution in the system to the access patterns of programs with data sharing; however, the high cost of the data migration cancels the benefit of improved data locality. Similarly, it is also possible to redistribute computations so that the data access patterns of the program match a given data distribution, however this method can introduce potentially large synchronization overhead. To address the dual problem of distributing computations and data, we describe a simple system API that can be used to change *both* the distribution of computations and the distribution of data in the system. Using the described API together with simple program-level transformations can almost completely eliminate remote memory accesses in programs with data sharing and thus helps to better exploit the performance potential of NUMA systems. The API presented here is implemented as a library and a set of C preprocessor macros. The API offers two kinds of language primitives: primitives to manipulate the distribution of a program's data and additional schedule clauses for distribution of loop iterations.

5.3.1 Data distribution primitives

Block-cyclic data distribution

The function `create_block_cyclic_distr()` (shown in Figure 5.9(a)) is used to define a *block-cyclic data distribution* [12]. The block size given as a parameter to this function influences the data distribution. Consider the case of a memory region that stores a two-dimensional matrix of size `NX` × `NY` (as shown in Figure 5.4(c)). If the block size is `NX` × `NY / 2`, the data region will be x-wise distributed as shown in Figure 5.4(c). If, however, the size of the block is `NY / 2`, the region will be y-wise distributed (Figure 5.4(d)). By further varying the block size the function can be used to set up a block-cyclic distribution for matrices with dimensionality higher than two as well.

Applying data distributions

The primitives of the API decouple the description of the data distribution from the operations to implement a chosen data distribution. The function `create_block_cyclic_-`

`distr()` defines only a *description* of the distribution in memory of a given memory region `m` (the memory region is passed to the function as a parameter; the memory region can be, e.g., a two-dimensional matrix similar to the one described in Section 5.2). The function `distribute_to()` shown in Figure 5.9(c) takes this description and then applies it (i.e., the function migrates data to enforce the distribution). For example, the in-program data distribution described earlier for the `bt` and `ft` benchmarks uses this call to change the distribution of data in the system at runtime (see lines 4 and 7 of Figure 5.6). The implementation of the `distribute_to()` call relies on the Linux-standard `move_pages()` system call. Other OSs have similar mechanisms for user-level data migration that can be alternatively used. In the current implementation data distribution can be enforced at page-level granularity.

Block-exclusive data distribution

Access patterns cannot be easily changed, as changing the access pattern of code with loop-carried dependences can require frequent inter-thread synchronization. Inter-thread synchronization causes overhead and as a result the data locality gained from changing access patterns might not increase performance. Alternatively, the distribution of a data region can be changed to *block-exclusive* (see Figure 5.10(a)). The key idea behind using a block-exclusive distribution is that a region with a block-exclusive distribution can be swept by two different data access patterns (e.g, with an x-wise and a y-wise access pattern) with no parts of the memory region shared between the processors of the system[2]. As a result, a block-exclusive data region can be placed at processors so that all accesses to the region are local.

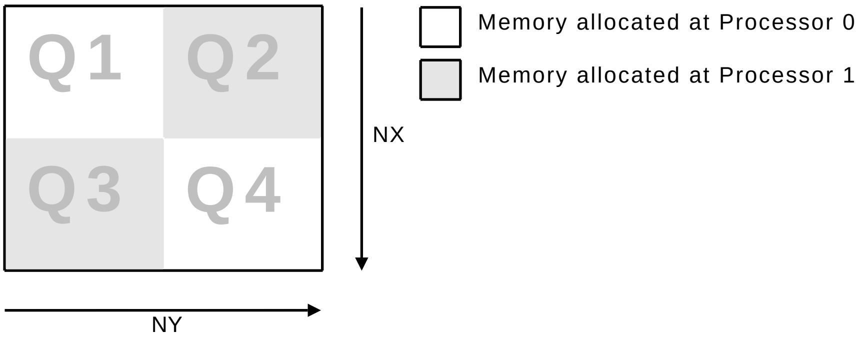
To illustrate the advantages of the block-exclusive data distribution let us consider the simple example when eight worker threads access the block-exclusive memory region in Figure 5.10(a). To simplify further discussion, we divide the memory region into four equal-sized quadrants (Q1–Q4). The eight worker threads considered in this example traverse the memory region with two different access patterns, first with an x-wise access pattern followed by a y-wise access pattern.

The x-wise traversal of the memory region must be processed in two phases due to the block-exclusive data distribution of the region. These two phases are illustrated in Figure 5.10(b) resp. Figure 5.10(c). In the first phase (Figure 5.10(b)) the worker threads process in parallel the left half of the data region (quadrants Q1 and Q3). The iteration distribution is standard OpenMP static scheduling: threads are assigned iterations in the ascending order of their respective thread number (threads T0–T3 process Q1, threads T4–T7 process Q3)[3]. As a result, all threads access memory locally. In the second phase (Figure 5.10(c)) the threads process quadrants Q2 and Q4, which are allocated to processors in a different order than quadrants Q1 and Q3. To guarantee data locality in the second phase as well, the distribution of iterations between the threads is different from the first processing phase: threads T0–T3 process quadrant Q4, and threads T4–T7 process quadrant Q2 (as shown in Figure 5.10(c)). Between the two phases of the x-wise traversal, thread synchronization is required; however, as there is only one place in the traversal

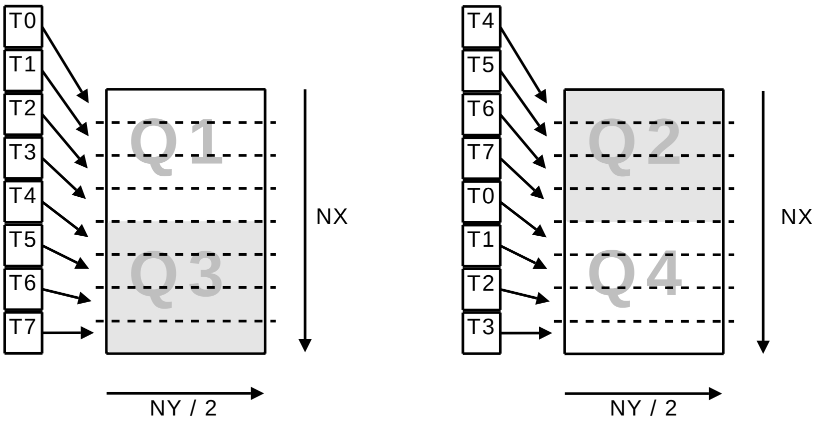
---

[2] Intuitively, a block-exclusive data distribution allows threads to sweep the data region with both and x-wise and y-wise pattern so that threads running at each processor access a part of the data region mostly exclusively (in many cases sharing between processors cannot be often completely eliminated as in practice data distributions can be enforced only at page-level granularity).

[3] The OpenMP standard [16] does not require an assignment of loop iterations to threads in ascending order; however, many OpenMP implementations (e.g., the OpenMP implementation of GCC) assign loop iterations to threads that way.

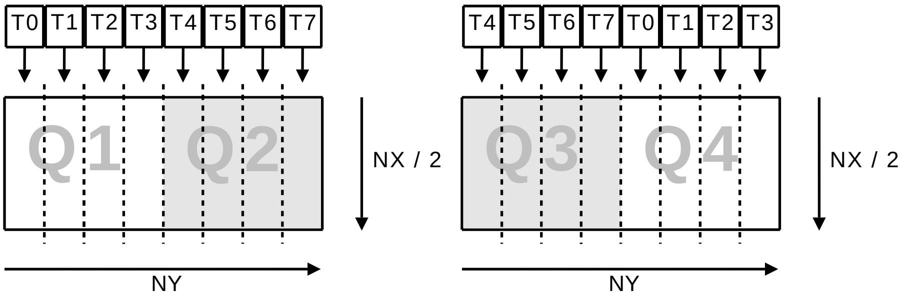


(a) Block-exclusive data distribution on two processors.



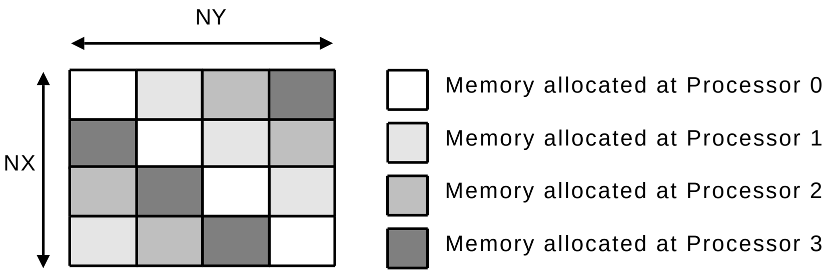
(b) x-wise traversal: phase 1.

(c) x-wise traversal: phase 2.



(d) y-wise traversal: phase 1.

(e) y-wise traversal: phase 2.



(f) Block-exclusive data distribution on four processors.

Figure 5.10: Block-exclusive data distribution.

where cross-thread data is accessed (when execution transitions from processing the first two quadrants to processing the second two), the cost of this synchronization is negligible.

Data sharing is in many cases caused by a memory region being traversed with an x-wise pattern followed by a y-wise pattern. So far we have seen an x-wise traversal of a block-exclusive memory region. A y-wise traversal of the same memory region is also possible, and it proceeds in two phases. In the first phase (Figure 5.10(d)) the worker threads process quadrants Q1 and Q2, followed by quadrants Q3 and Q4 in the second phase (Figure 5.10(e)). As the distribution of loop iterations in the phases of the y-wise traversal is similar to the case of the x-wise traversal, we do not discuss it any further. Please note, however, that the y-wise traversal of the memory region guarantees data locality as well.

Figure 5.9(b) shows the programming language primitive to create a block-exclusive data distribution. So far we have discussed the block-exclusive layout only for two processors. Using the `create_block_exclusive_distr()` on a system with a number of processors higher than two results in a *latin-square* [26, 98] distribution of memory. The latin-square distribution of a two-dimensional matrix on a 4-processor system is shown in Figure 5.10(f). The data distribution primitives described in this section can be used in a system with any number of cores/processors. The primitives only require that information about the number of cores/processors is available at runtime (which is the case in many recent OSs, e.g., on Linux the `libnuma` library makes hardware-related information available at runtime).

5.3.2 Iteration distribution primitives

Well-known methods for iteration distribution, like static scheduling, are inflexible and cannot always follow the distribution of data in the system. For example, the assignment of loop iterations to data shown in Figure 5.10(c) or Figure 5.10(e) is impossible in an OpenMP-like system. To alleviate this problem we provide a directive (shown in Figure 5.11(a)) to schedule loop iterations with *inverse static scheduling*.

Similar to OpenMP static scheduling, inverse static scheduling partitions the iteration space of a loop into non-overlapping chunks and then assigns each chunk to a worker thread for processing. Figures 5.10(b) shows the assignment of loop iterations to worker threads with standard OpenMP static scheduling: Threads are assigned loop iterations in the ascending order of their respective thread number. Figure 5.10(c) shows the assignment of loop iterations to worker threads with inverse static scheduling (assuming a 2-processor system): The first half of the loop iteration space is assigned to the second half of worker threads (threads T4–T7), the second half of the iteration space is assigned to the first half of the worker threads (threads T0–T3). With static inverse scheduling, data locality can be achieved also for code accessing regions with distribution as in Figure 5.10(c) or Figure 5.10(e). To simplify the discussion and to improve the readability of code examples, Figure 5.11(a) shows a possible way to include inverse scheduling into the OpenMP standard (and not the current implementation of inverse scheduling using macros).

We have seen in the previous section that the block-exclusive data distribution can be generalized to any number of processors (see Figure 5.10(f) for the 4-processor case). As the distribution of loop iterations must match the distribution of data in the system for data locality, inverse scheduling must be generalized to any number of processors as well. We call the generalized scheduling primitive *block-exclusive scheduling* (syntax shown in Figure 5.11(b)).

```
#pragma omp parallel for schedule(static-inverse)
```

(a) Inverse static scheduling.

```
#pragma omp parallel for schedule(block-exclusive, dir, id)
```

(b) Block-exclusive static scheduling.

Figure 5.11: Loop iteration scheduling primitives.

In the general case the block exclusive data distribution partitions a data region into blocks. In a system with `p` processors the data distribution can be seen either as `p` columns of blocks, or as `p` rows of blocks (see Figure 5.10(f) for a block-exclusive data distribution on a system with `p = 4` processors). For example, on an x-wise traversal the data distribution can be viewed as `p` columns of blocks and on a y-wise distribution the data distribution can be viewed as `p` rows of blocks. For both views of a block-exclusive data distribution there are `p` different iteration-to-core distributions, one iteration distribution for each different configuration of columns of blocks respectively rows of blocks (so that the iterations are allocated to the processors that hold the block of data processed). As a result, a block-exclusive scheduling clause is defined as a function of two parameters. The first parameter specifies the view taken (columns of blocks respectively row of blocks), and the second parameter specifies the identifier `id` ($0 \leq$ `id` $<$ `p`) of the schedule used. Section 5.4 presents a detailed example of using the block-exclusive loop iteration distribution primitive.

5.4 Example program transformations

This section shows how two NPB programs can be transformed with the previously described API to execute with better data locality.

5.4.1 `bt`

Figure 5.12(a) shows code that is executed in the main computational loop of `bt` (for brevity we show only an excerpt of the complete `bt` program). The code has a y-wise access pattern and accesses two matrices, `lhs` and `rhs`, in the `matvec_sub()` function. There are loop-carried dependences in the code: The computations performed on the current row `i` depend on the result of the computation on the previous row `i - 1`.

In a 2-processor system the code is transformed in two steps. At the start of the program the distribution of both matrices, `lhs` and `rhs`, is changed to block-exclusive using the memory distribution primitives described in Section 5.3. In the second step the outermost loop of the computation (the loop that iterates through the rows of the arrays using variable `i`) is split into two halves. The transformed code is shown in Figure 5.12(b). Both of the resulting half-loops iterate using variable `i`, but the first loop processes the first `(NX - 2) / 2` rows of the matrices, and the second loop processes the remaining `(NX - 2) / 2` rows. The work distribution in the first loop is standard static block scheduling, but in the second loop inverse scheduling is required so that the affinity of the worker threads matches the data distribution. Splitting the outermost loop results in the two-phase traversal previously shown in Figure 5.10(d) and Figure 5.10(e). Note that the transformation does not break any of the data dependences of the

```
for (i = 1; i < NX - 1; i++)
#pragma omp parallel for schedule(static)
    for (j = 1; j < NY - 1; j++)
        for (k = 1; k < NZ - 1; k++)
            matvec_sub(lhs[i][j][k][0],
                       rhs[i - 1][j][k],
                       rhs[i][j][k]);
```

(a) Original `bt` code.

```
for (i = 1; i < (NX - 2) / 2; i++) {
#pragma omp parallel for schedule(static)
    for (j = 1; j < NY - 1; j++) {
        for (k = 1; k < NZ - 1; k++) {
            matvec_sub(lhs[i][j][k][0],
                       rhs[i - 1][j][k],
                       rhs[i][j][k]);
for (i = (NX - 2) / 2; i < NX - 1; i++) {
#pragma omp parallel for schedule(static-inverse)
    for (j = 1; j < NY - 1; j++) {
        for (k = 1; k < NZ - 1; k++) {
            matvec_sub(lhs[i][j][k][0],
                       rhs[i - 1][j][k],
                       rhs[i][j][k]);
```

(b) 2-processor version of `bt` code.

```
for (p = 0; p < processors; p++) {
    for (i = p * (NX - 2) / processors;
         i < (p + 1) * (NX - 2) / processors;
         i++) {
#pragma omp parallel for
        schedule(block-exclusive, X_WISE, p)
        for (j = 1; j < NY - 1; j++)
            for (k = 1; k < NZ - 1; k++) {
                matvec_sub(lhs[i][j][k][0],
                           rhs[i - 1][j][k],
                           rhs[i][j][k]);
}
```

(c) Generalized version of `bt` code.

Figure 5.12: Program transformations in `bt`.

program. Therefore, only one synchronization operation is required for correctness (at the point where the first half of the iteration space has been processed). This program transformation can be generalized to systems with an arbitrary number of processors as shown in Figure 5.12(c).

The main computational loop of `bt` executes other loops as well (not just the loop shown in Figure 5.12(a)). Many of these other loops have a different, x-wise, access pattern. To match the access patterns of these loops to the block-exclusive data distribution of the program's data, the

```
#pragma omp parallel for schedule(static)
for (i = 0; i < NX; i++)
    for (j = 0; j < NY; j++)
        // ...
```

(a) `lu` lower triangular part.

```
#pragma omp parallel for schedule(static)
for (i = NX - 1; i >= 0; i--)
    for (j = NY - 1; j >= 0; j--)
        // ...
```

(b) `lu` upper triangular part.

```
#pragma omp parallel for schedule(static-inverse)
for (i = NX - 1; i >= 0; i--)
    for (j = NY - 1; j >= 0; j--)
        // ...
```

(c) `lu` upper triangular part with inverse scheduling.

Figure 5.13: Program transformations in `lu`.

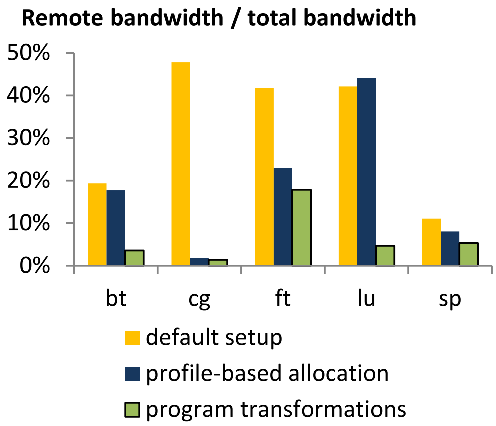
code must be transformed in a manner similar to the transformations shown in Figure 5.12(b) and 5.12(c). These transformations require programmer effort, but in the end the access patterns of all loops of the program match the blocked-exclusive distribution of shared data, and data distribution is required only once (at the start of the program).

5.4.2 `lu`

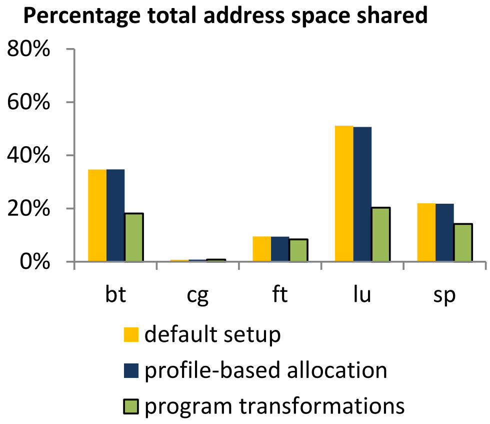
The `lu` program solves the Navier-Stokes equations in two parts: a *lower* (Figure 5.13(a)), and an *upper* part (Figure 5.13(b)). Both parts have an x-wise access pattern, therefore the memory regions used by the program are initialized x-wise. However, as the upper triangular part (Figure 5.13(b)) traverses rows in descending order of row numbers, the static scheduling clause of OpenMP distributes loop iterations between worker threads so that each worker thread operates on remote data. To increase data locality we match the access pattern of `lu` to its data distribution by using the `static-inverse` scheduling clause for the upper triangular part, as shown in Figure 5.13(c).

5.5 Evaluation

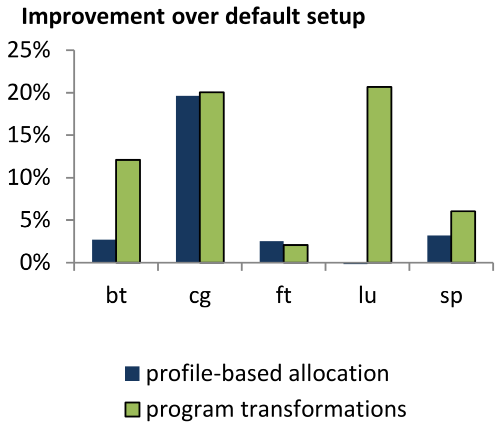
This section presents an evaluation of the previously described program transformations. In Section 5.5.1 we evaluate the effectiveness of the proposed techniques in improving data locality on the 2-processor 8-core Nehalem-based machine. In Section 5.5.2 we look at the scalability of program transformations using a larger, 4-processor 32-core machine based on the Westmere microarchitecture. In Section 5.5.3 we compare the performance of program transformations with other optimization techniques described in related work. The software and hardware configuration we use is described in Section 4.2. The variation of the measurement readings is negligible.



(a) Bandwidth distribution.



(b) Percentage shared pages.



(c) Performance improvement over default setup (first-touch page placement and identity affinity).

Figure 5.14: Performance with program transformations (2-processor 8-core machine).

5.5.1 Data locality

The original version of `sp` is synchronization-limited due to a single triple-nested loop that is parallelized on the innermost level. We eliminate the synchronization boundedness of this loop by moving the parallelization primitive to the outermost loop. We also store data accessed by the loop in multi-dimensional arrays (instead of single-dimensional ones), so that loop-carried data dependences are eliminated; using multi-dimensional arrays results in a negligible increase of the program's memory footprint. This simple change gives a 10X speedup over the out-of-the-box version of the benchmark on our 2-processor 8-core machine; as a result, `sp` is memory bound and can profit further from data locality optimizations (we only use the improved version of the benchmark in this chapter).

Figure 5.14(a) shows the percentage of remote memory bandwidth relative to the total bandwidth for each program we consider in three scenarios: (1) the *default setup* uses the combination of first-touch page placement and thread mapping with identity affinity, as described in Section 5.1.1, (2) *profile-based allocation* (as described in Section 5.1.3), and (3) the *program transformations* we propose. Programs are executed with 8 threads in all scenarios.

For `bt`, `lu` and `sp`, program transformations reduce the percentage of remote accesses significantly (to around 4% on average). For the `cg` benchmark, profile-based memory allocation almost completely eliminates remote memory accesses (see Figure 5.2(b)). However, we are able to achieve the same effect without profiling by inserting data distribution primitives into the program. In `ft` most remote memory accesses are caused by a single loop. Transforming this loop is possible by inserting fine-grained inter-thread synchronization, which could result in high overhead that can cancel the benefits of the improved data locality. Nonetheless, we obtain a reduction of the fraction of remote memory accesses relative to profile-based memory allocation (from 23% to 18%) by distributing memory so that the distribution matches the access patterns of the most frequently executing loop of the program.

Figure 5.14(b) shows for each program we consider the percentage of program address space shared with the default setup, with profile-based memory allocation, and with program transformations (the figures consider only the sharing of heap pages). For three programs (`bt`, `lu`, and `sp`) program transformations reduce the fraction of shared pages, it is thus possible to place more pages appropriately than with the original version of the program (either by using the proposed data distribution primitives, or by profiling the program). Finally, Figure 5.14(c) shows the performance improvement with profile-based memory allocation and program transformations relative to the default setup. By eliminating sharing we obtain performance improvements also when profile-based allocation does not. Moreover, for `cg` (the program that improves with profile-based allocation) we are able to match the performance of profile-based allocation by distributing memory so that the distribution of memory matches the program's access patterns.

5.5.2 Scalability

Adding more processors to a NUMA system increases the complexity of the system. Standard OSs and runtime systems handle the increase of the number of processors gracefully, however, if good program performance is also desired, action must be taken (e.g., by the programmer). This section evaluates the scalability of the proposed program transformations on the 4-processor 32-core Westmere-based machine described in Section 4.2 that is larger than the 2-processor machine previously examined. For all benchmark programs size C (the largest size) is used.

To look at the scalability of the benchmark programs each program is executed in four different configurations (i.e., with 4, 8, 16, and 32 threads, respectively). As cache contention can cause significant performance degradations in NUMA systems [13, 65], we fix the thread-to-core affinities so that the same number of threads is executed on each processor of the system

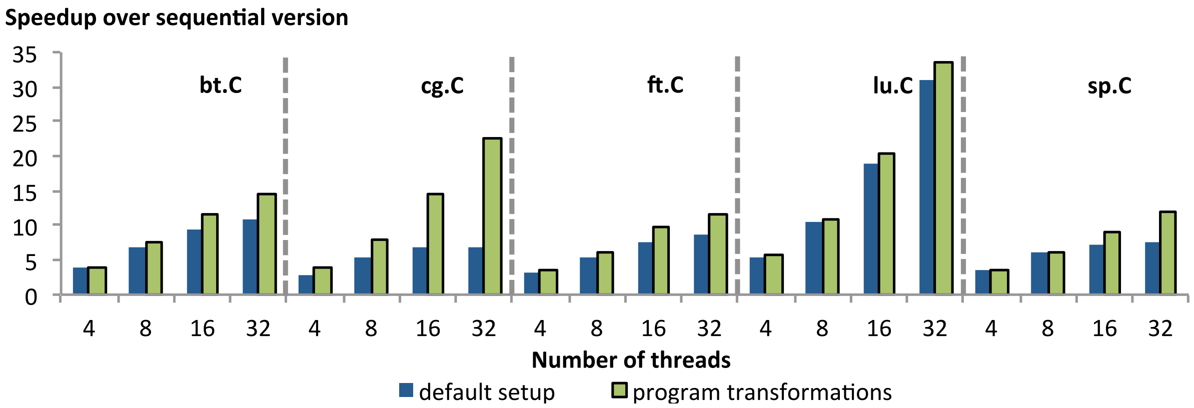


Figure 5.15: Performance with program transformations (4-processor 32-core machine).

and thus the cache capacity available to each thread is maximized (remember that each processor has a single last-level cache shared between all processor cores). For example, in the 4-thread configuration one thread is executed on each processor of the 4-processor system; in the 32-thread configuration each processor runs eight threads.

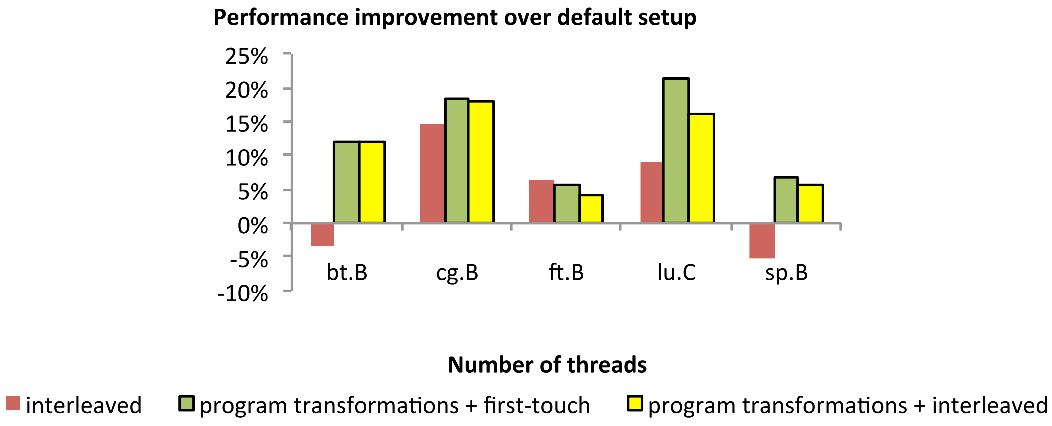
Figure 5.15 shows the speedup of the benchmark programs over their respective sequential version. The figure compares two versions of the benchmark for each runtime configuration. The default setup relies on the first-touch memory allocation policy and uses the previously discussed thread-to-core assignment (which corresponds to identity affinity in the configuration with 32-threads). With program transformations the memory distribution and the scheduling of loop iterations is changed so that data locality is maximized. Performance improves with program transformations, with results that are similar to those obtained on a 2-processor 8-core system. In the 32-thread configuration we measure a performance improvement of up to 3.2X (and 1.7X on average) over the default setup. Program performance also scales better with program transformations than with the default setup. (For the programs of the NAS suite studied here, the programs obtain a speedup of up to 33.6X over single-core execution, with a mean speedup of 18.8X.)

5.5.3 Comparison with other optimization techniques

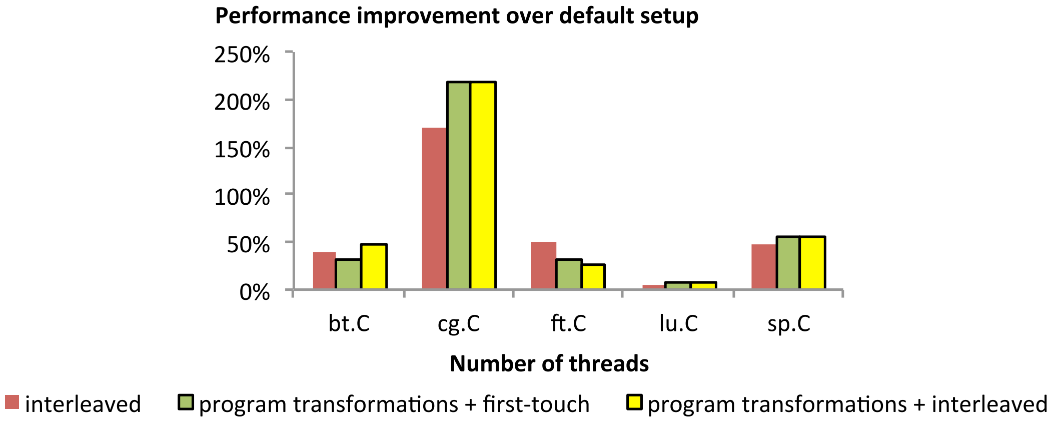
The approach presented in this chapter allows the programmer to control the distribution of data across the processor of a NUMA-multicore system. There exist, however, alternative techniques for managing a program's data distribution. Two frequently used techniques are memory replication and interleaved page placement.

State-of-the-art software systems allow replication of memory regions. However, to limit the overhead of synchronizing replicas, memory replication is enabled only for memory regions that are read-only or are written infrequently (e.g., in [27] only pages written less than 5 times during the lifetime of the program are replicated). The programs of the NAS PB suite we study in this chapter frequently read/write shared memory regions. We believe, thus, that replication would cause high overhead for these programs (or it would result in replication to be disabled), therefore we do not compare our approach to data replication.

In some cases interleaved page placement (i.e., distributing memory pages round-robin across processors) reduces contention on memory controllers relative to first-touch page place-



(a) Comparison with interleaved page placement (8-core machine).



(b) Comparison with interleaved page placement (32-core machine).

ment (the default policy of many OSs, including Linux). Figures 5.16(a) and 5.16(b) report the performance of the NAS programs with interleaved page placement relative to the default setup for the 2-processor 8-core and the 4-processor 32-core system, respectively. In some cases interleaved page placement results in a performance improvement of up to 170% relative to first-touch page placement (the default setup). In some cases, however, interleaved placement can decrease performance by at most 5%.

To allow a comparison with interleaved page placement, Figures 5.16(a) and 5.16(b) also report the performance of the program transformations described earlier in this chapter. Transformations target only a subset of a program's total data, that is, the migration primitives are applied only to data structures identified as performance-critical by the programmer. The data distribution of other program data (i.e., data structures *not* considered by the programmer) can effect performance as well, thus we report the performance of program transformations in two cases:

program transformations + first-touch In this case memory regions not targeted by programmer-controlled data migration are managed using the default first-touch policy. This case is the same as the "program transformations" earlier in this chapter.

program transformations + interleaved In this case memory regions not targeted by programmer-controlled data migration are managed using interleaved page placement. In this case a program is started up with the page placement policy set to interleaved. Then the program applies the data distribution specified by the programmer to performance-critical memory regions, but leaves the data distribution of other regions at interleaved to reduce memory controller contention caused by accesses to memory regions not considered for data migration.

In most cases program transformations (both in combination with first-touch and with interleaved page placement) result in better performance than using interleaved page placement only (up to 50% more performance improvement). The reason is that interleaved page placement reduces contention whereas transformations achieve good data locality. In one case (the `bt` benchmark on the 4-processor 32-core machine), program transformations improve performance over interleaving only when program transformations are used in combination with interleaved page allocation. In this case the memory controller contention caused by accesses to data structures not targeted by programmer-controlled data migration is a significant factor for the program's performance; using interleaved page placement for these pages reduces contention and enables program transformations to improve over using interleaved page placement for all program data.

For the `ft` benchmark interleaving results in better performance than program transformations, mostly due to the overhead of data migration (encountered by program transformations but not in the case with interleaved page placement).

5.6 Conclusions

In NUMA systems the performance of many multithreaded scientific computations is limited by data sharing. In-program memory migration, a conventional method in large cluster environments, does not help because of its high overhead. In some cases it is possible to redistribute loop iterations so that the access patterns of a loop match the data distribution. However, in the presence of loop-carried dependences redistribution of loop iterations is a complex task, moreover, redistribution can require extensive inter-thread synchronization that may reduce (or completely eliminate) the benefits of data locality.

This chapter presents a simple system API that is powerful enough to allow for programming scientific applications in a more architecture-aware manner, yet simple enough to be used by programmers with a reasonable amount of knowledge about the underlying architecture. Using this API together with program transformations reduces the number of shared pages and remote memory accesses for many programs of the NPB suite. This API allows a programmer to control the mapping of data and computation and realizes a performance improvement of up to 3.2X (1.7X on average) compared to the first-touch policy for NAS benchmark programs. Future multiprocessors are likely to see a widening performance gap between local and remote memory accesses. For these NUMA systems, compilers and programmer must collaborate to exploit the performance potential of such parallel systems. The techniques described here provide an approach that attempts to strike a balance between complexity and performance.

6

A parallel library for locality-aware programming

The previous chapter presented a simple API that allows adjusting the data distribution and computation scheduling of multithreaded programs. Source-level optimization can increase the data locality (and thus the performance) of multithreaded programs. Moreover, in case of programs with complex memory access patterns and/or data sharing, source-level data locality optimizations are an attractive alternative to automatic performance optimization techniques (e.g., profile-based data placement).

The API presented in the previous chapter is simple, but it has a number of limitations. First, the API supports only loop-parallel computations. Furthermore, implementing data locality optimizations in practice requires additional aspects to be considered. For example, with current parallel programming frameworks it is difficult to guarantee that data locality optimizations are portable. Moreover, optimizations implemented with current parallel programming frameworks are often not composable (e.g., they can lose their favorable properties in runtime systems that support composable parallel software).

This chapter presents TBB-NUMA, a parallel programming library based on Intel Threading Building Blocks (TBB) that supports portable and composable NUMA-aware programming. TBB-NUMA is based on task parallelism, but it supports loop-, pipeline-, and also other forms of parallelism through high-level algorithm templates. TBB-NUMA provides a model of task affinity that captures a programmer's insights on mapping tasks to resources. NUMA-awareness affects all layers of the library (i.e., resource management, task scheduling, and high-level parallel algorithm templates) and requires close coupling between all these layers. Data locality optimizations implemented with TBB-NUMA (for a set of standard benchmark programs) result in up to 57% performance improvement over standard TBB, but, more importantly, optimized programs are portable across different NUMA architectures and preserve data locality also when composed with other parallel computations.

6.1 Practical aspects of implementing data locality optimizations

6.1.1 Introduction

Due to the large performance penalty of cross-chip memory accesses, performance optimizations for NUMA systems typically target improving data locality, that is, the reduction (or even elimination) of remote memory accesses [13, 17, 27, 58, 68, 79, 104, 107, 107]. Optimizations are often automatic, that is, the runtime system (e.g., the OS or the VM) profiles the memory accesses of programs and then, based on the profiles, it automatically adjusts the distribution of data and/or the scheduling of computations to more efficiently use the memory system.

Automatic optimizations for NUMA systems can be highly effective; however, for some programs (e.g., programs with complex memory access patterns) profiles do not convey enough information to enable the runtime system to carry out optimizations successfully. In these cases, high-level information about programs (e.g., program data dependences) is needed. As this type of information is likely to be available to the programmer, several projects consider making the development toolchain NUMA-aware. E.g., recent profilers like MemProf [55], Memphis [72], and DProf [82] present information about a program's memory behavior to the developer, who can then change the code to improve performance.

Profilers pinpoint code locations with inefficient usage of the memory system. In practice, however, programs are rarely optimized for NUMA systems as commonly used parallel languages and libraries like OpenMP or Intel Threading Building Blocks (TBB) are geared towards exploiting the lower levels of the memory system (e.g., L1 and L2 caches), if at all, and have no support for NUMA systems. More specifically, existing parallel programming frameworks have three main limitations.

First, existing frameworks usually require memory-system-aware code to consider the layout of the memory system in full detail, thus optimized programs are *not portable*. Second, NUMA-aware code is *not composable*. Mapping data and computations depends on the hardware resources (i.e., cores/processors) available to the program. However, in frameworks with support for composable parallel software (i.e., parallel software composed of multiple, concurrently executing parallel computations [81]) the amount of resources available to a computation can change over time, which requires memory-system-aware programs to adapt the mapping at runtime. Existing frameworks provide the programmer little information about the program's runtime configuration, thus optimizations often simply assume that all hardware resources (i.e., all cores/processors) are continuously available. As a result, the advantages of memory system optimizations are annulled as soon as the optimized computation is composed with other parallel computations. Finally, existing parallel programming frameworks have *no support for explicit mapping* of data and computations, i.e., the programmer is required to be aware of runtime/compiler/library internals to be able to set up a mapping. Thus, even if a programmer conceptually knows how to optimize a program, implementing optimizations is difficult (or even impossible) with existing frameworks.

Due to the previously mentioned limitations, source-level optimizations for NUMA systems are rare in practice and the performance potential of NUMA systems is often unexploited. To fill the gap in the development toolchain for NUMA systems, this chapter presents TBB-NUMA, a parallel programming library for programming NUMA systems. We first discuss theoretical and practical considerations for implementing data locality optimizations (Sections 6.1.2 and 6.1.3) and then we articulate the goals of TBB-NUMA library (Section 6.1.4).

6.1.2 Principles of data locality optimizations

Data locality optimizations for NUMA systems have traditionally targeted the co-location of data and computations. To understand the principles of these optimizations, let us consider an example multithreaded program that is parallelized for a 2-processor 8-core NUMA system (the

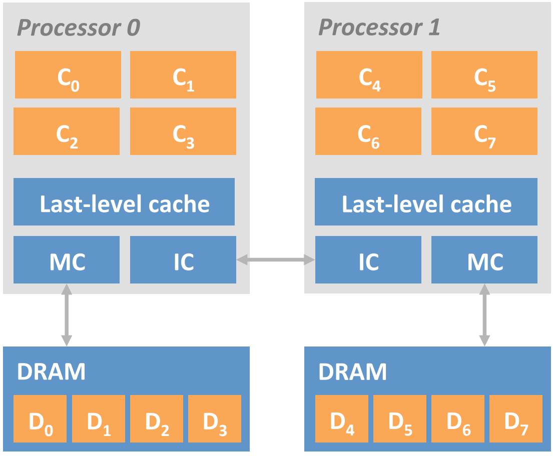


Figure 6.1: Computation optimized for data locality.

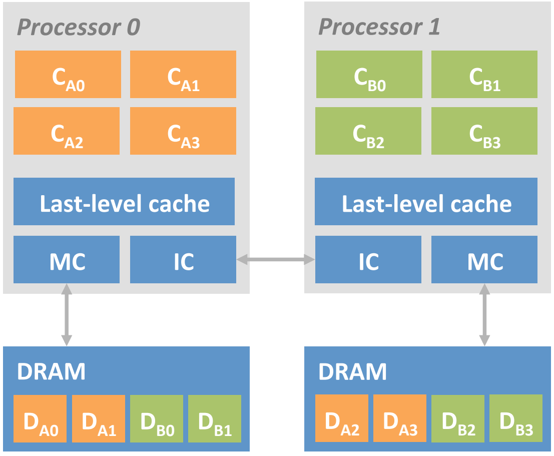


Figure 6.2: Shared threads: Unfortunate mapping.

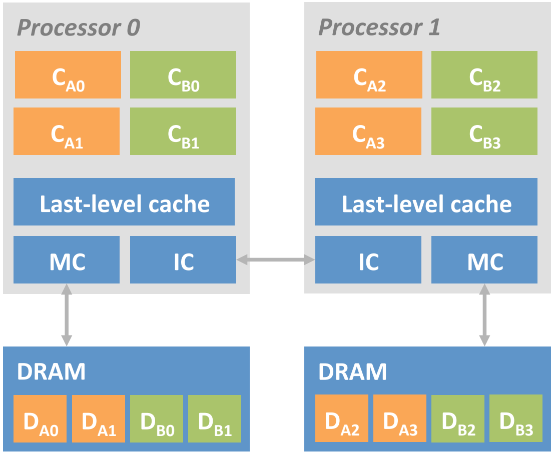


Figure 6.3: Shared threads: Appropriate mapping.

system in Figure 6.1). The program consists of a set of concurrently executing computations $C_0, C_1, \ldots, C_7$; each computation accesses a subset $D_0, D_1, \ldots, D_7$ of the total data used by the program.

To achieve good data locality, the programmer must go through a series of steps. First, the programmer must parallelize the algorithm (i.e., define the computations C_i) so that the data subsets D_i overlap as little as possible. Second, the programmer must distribute data subsets among processors. The final step is to schedule computations so that each computation C_i executes at the same processor as where its data D_i is placed at.

Figure 6.1 shows the mapping of the example program onto the 2-processor 8-core NUMA system. In the figure each computation executes at the processor where its data is allocated. If data subsets do not overlap (small adjustments often suffice to adjust overlap in multi-threaded computations, as shown by previous work [112]), this mapping is beneficial for both caching and DRAM performance: (1) The cache capacity available to the computation is maximized (data subsets are disjoint; as a result, each piece of data can be present in only one cache), (2) as data is placed at all DRAM modules, all paths to memory are utilized, which increases the bandwidth available to the program and reduces contention on memory interfaces, and (3) as each computation accesses locally placed data, the program does not encounter any remote (cache or DRAM) memory accesses.

6.1.3 Enforcing data locality in practice

Data locality optimizations are simple in theory but difficult to implement in practice. Although data distribution is well supported in recent OSs (e.g., Linux supports per-processor memory allocation through the `libnuma` library and memory migration through the `move_pages()` system call), scheduling computations at appropriate processors is problematic in today's parallel languages and libraries. In commonly used parallel frameworks scheduling computations at processors depends on two components. Most parallel frameworks operate with thread pools, thus computations must be first mapped to threads in the pool; this chapter concerns task-based parallelism, we thus use the terminology of *setting task-to-thread affinities* for mapping computations to threads. Second, threads from the pool must be *pinned to processors* of the system to ensure that computations execute where intended. If both mappings are set up right, the system guarantees data locality. In the following we discuss problems related to both components.

Component 1: Setting task-to-thread affinities

In commonly used parallel frameworks task-to-thread affinities are *implicit*, that is, the programmer has no direct control on how to map computations to threads. OpenMP static loop partitioning is an example of implicit computation scheduling. For statically partitioned parallel loops the OpenMP runtime assigns a well-defined chunk of the iteration space to each thread. If the programmer is aware of the internals of static partitioning and knows which pieces of data are touched by each loop iteration, she can distribute data among processors so that each thread accesses its data locally. With other OpenMP work-division schemes (e.g., dynamic partitioning), however, the distribution of loop iterations between threads is not deterministic [80], thus the programmer cannot assume much about the data accesses of the program and data locality is, as a result, not controllable.

Setting up task-to-thread affinities is not easy in case of systems based on task parallelism either. For example, in Intel TBB each task can be assigned a special value; the value defines the affinity of that task to a thread in the pool. The TBB Reference Manual [43] states the following about the values of a task's affinity:

> "A value of 0 indicates no affinity. Other values represent affinity to a particular thread. Do not assume anything about non-zero values. The mapping of non-zero values to threads is internal to the Intel TBB implementation."

This would require the programmer to reverse-engineer the TBB implementation to be able to set up a mapping between tasks and threads. Due to this limitation, it is difficult to implement NUMA data locality optimizations in TBB.

Finally, defining task-to-thread affinities depends on the number of threads available to the program (a value that can change at runtime) but the distribution of data is expressed depending on the number of processors in the system. To ensure data locality on any system and in any runtime configuration the programmer must consider both parameters, which makes writing NUMA-aware programs with current systems even more cumbersome.

Component 2: Pinning threads to processors

The second component of mapping computations to processors is pinning threads to processors. Unless threads are pinned, the OS scheduler is allowed to freely move threads around in the system. OS re-schedules can result in remote memory accesses (or costly data migrations if data follows the computations using it, e.g., in systems with automatic data migration [13]; therefore, this chapter assumes a standard OS without automatic data migration).

Some OpenMP implementations allow (although not required by the OpenMP standard [80]) pinning thread pool threads to processors. Pinning threads requires understanding the memory system for every new machine the program is to be run on. If threads are pinned to processors, moreover, the programmer has distributed data and she has also set up task-to-thread affinities (e.g., by relying on the properties of static loop scheduling, as discussed before), each piece of data will be accessed at a well-defined processor and the program has thereby good data locality.

Pinning threads to processors works well as long as only one parallel computation uses a thread pool at a time. Modern runtime systems, however, support composable parallel software, that is, programs that contain nested parallelism, programs that reuse functionality from parallelized libraries, or programs that are parallelized using different parallel languages/libraries [81]. For these programs the thread pool of the runtime is shared by multiple parallel computations and the runtime distributes threads between all computations registered with it.

To illustrate the problems composability causes for programs optimized for NUMA systems, Figure 6.2 shows an example where two parallel computations, C_A and C_B, execute in parallel on the example 2-processor 8-core system. Computation C_A is composed of subcomputations $C_{A_0} \ldots C_{A_3}$; each subcomputation C_{A_i} accesses a different data subset D_{A_i}. Similarly, each subcomputation C_{B_i} of C_B accesses a distinct data subset D_{B_i}. The programmer optimized both computations for NUMA, thus data used by the computations is distributed across processors (according to the principles discussed in Section 6.1.2). The programmer has set up task-to-thread affinities as well, but as the runtime is not aware of the programmer's intentions, it can allocate threads to computations in several ways. Figure 6.2 shows an unfortunate allocation that cancels the optimization intended by the programmer: C_A is mapped to threads executing at Processor 0 and C_B is mapped to threads executing at Processor 1. Thus, both computations access some of their data remotely ($D_{A_2}, D_{A_3}, D_{B_0}, D_{B_1}$). Figure 6.3 shows an appropriate assignment of threads to computations: In this case each computation is assigned threads from both processors so that each computation can access data locally and exploits all caches of the system.

6.1.4 Goals of TBB-NUMA

TBB-NUMA aims to support programming NUMAs by:

Explicit mapping The programmer can define the distribution of data among processors and, in addition, can also express the preferred schedule of computations in the form of hints to the library's work-stealing scheduler without being required to understand runtime system internals. The scheduler honors these hints unless there are idle resources; in this case a task may be moved by the scheduler to a different processor in an attempt to balance the load (as in current systems, incurring the overhead of remote execution is preferable to idling processing resources). To hide the complexity of the work-stealing scheduler and therefore make writing NUMA-aware code easier, TBB-NUMA defines a set of parallel algorithm templates that programmers can adapt and reuse.

Portability Programmers are not required to have information about the exact hardware layout: TBB-NUMA programs are written for a generic NUMA system with P processors, the library automatically determines the remaining details of pinning threads to appropriate processors. As a result, optimized programs are portable.

Composability The runtime manages its thread pool so that the advantages of data locality optimizations are preserved even if only a fraction of all system resources are available for an optimized computation. This setup allows optimized programs to be included as part of libraries (or reuse functionalities from libraries already parallelized) and to utilize the memory system appropriately at the same time.

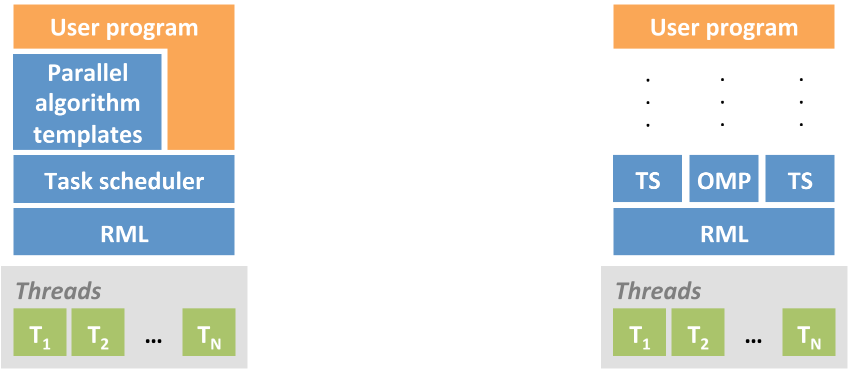
TBB-NUMA extends Intel TBB, so the chapter first focuses on the architecture of standard TBB (Section 6.2). Then, Section 6.3 highlights the differences between standard TBB and TBB-NUMA in terms of locality-aware programming. Finally, Section 6.4 presents an evaluation of the performance, composability, and portability of data locality optimizations implemented with TBB and TBB-NUMA for a set of well-known benchmark programs.

6.2 Anatomy of TBB

Standard TBB has a layered architecture (shown in Figure 6.4(a)[1]). This section describes the layers top-down, that is, the discussion starts with the layer closest to the programmer and ends with the farthest layer (the layer closest to the hardware).

---

[1] TBB has an additional layer, the *task arena*, below the task scheduler layer. Threads are registered with the task arena, the task scheduler implements only scheduling. To simplify the discussion we refer to the task scheduler and task arena layers as one layer (task scheduler), see [43] for exact details of TBB's implementation.



(a) TBB layers. (b) Composed program.

Figure 6.4: TBB architecture.

6.2.1 User programs

There are two ways to implement parallelism with standard TBB: programmers can either use the library's Cilk-style work-stealing scheduler [35] directly or they can reuse parallel algorithm templates from a set of templates defined by the library. Templates hide the complexity of the work-stealing scheduler from the programmer, but they still use the work-stealing scheduler internally.

6.2.2 Parallel algorithm templates

TBB supports loop parallelism through the `parallel_for` algorithm template (and variations of it, e.g., `parallel_reduce`, `parallel_do`). TBB supports pipeline-parallelism as well (through the `pipeline` template). In this chapter we concentrate on two algorithm templates, `parallel_for` and `pipeline`, because they are widely used and they represent two significantly different ways of approaching parallelism. In standard TBB both templates are optimized for better utilizing L1 and L2 caches. The `parallel_for` template preserves cache locality if it is given an `affinity_partitioner` object as a parameter. We briefly discuss this optimization in Section 6.3.3 (see [3] and [86] for details about the principles and implementation of this optimization, respectively). Parallel pipelines are optimized for better L1 and L2 cache locality through the way they generate the task tree corresponding to a pipeline computation (see Section 6.3.4 for details).

6.2.3 Task scheduler

Similar to Cilk [35], the TBB task scheduler interface exposes library functions to spawn and join tasks (implemented in TBB by the `spawn()` and `wait_for_all()` methods and variations of them). TBB allows but does not guarantee parallelism, thus the task scheduler can have multiple threads (but must have at least one thread) at any given point of time. Each thread has a local deque where spawned tasks are inserted. A thread removes tasks for execution from its local deque in LIFO order and, if the local deque is empty, steals tasks from other threads' deques in FIFO order.

The task scheduler has a set of mailboxes. Each thread in the task scheduler is connected

1. The task returned by the current task `t`.
2. The successor of `t` (if all predecessors of `t` have completed).
3. The task removed from the thread's own deque (LIFO).
4. The task removed from the mailbox this thread is currently connected to.
5. The task removed from the task scheduler's shared queue.
6. The task removed from an other randomly chosen thread's deque (FIFO) (steals).

Figure 6.5: Standard TBB: Rules to fetch next task.

to a (different) *mailbox*. A task can be assigned a special value that specifies the affinity of that task to a mailbox.

The definition of task affinities provided by the TBB Reference Manual (see Section 6.1.3 and [43]) is misleading. According to the manual, a task affinity implies that a task is associated with a *particular thread*, yet an affinity value associates a task *only with a mailbox*. During the lifetime of a program possibly different threads (but only one thread at a time) can be connected to a mailbox. Therefore, affinity values provided by the standard TBB implementation guarantee only that a task is associated with a mailbox, but not with any particular thread. We further discuss the implications of task-to-mailbox affinities in Section 6.3.3.

An affinitized task (a task with a non-zero affinity value) is *sent* to the thread currently connected to the mailbox. Sending is realized by *inserting* the task into the mailbox. The thread connected to the mailbox *receives the task* by *removing* it from the mailbox. Furthermore, an affinitized task is inserted not only into the mailbox it is sent to, but into the local deque of the thread that created it as well. Lastly, task affinities are internal to the TBB implementation by definition and only the `affinity_partitioner` uses them internally.

TBB supports asynchronous operations through the `enqueue()` call. Tasks to be executed asynchronously are inserted into a FIFO queue shared between all threads (but are not inserted into any thread's local deque). Due to the multiple types of queues in the task scheduler, TBB defines a set of rules (Figure 6.5) that specify from where thread is supposed to fetch the next task to be executed. The rules are listed in order of decreasing priority. If a high priority rule is unsuccessful, the scheduler tries the next rule.

6.2.4 Resource Management Layer

The number of threads in a task scheduler is determined by the Resource Management Layer (RML). (To avoid excessive OS scheduler overhead, the RML limits the number of threads available.) TBB is interoperable with other parallel frameworks (e.g., Intel OpenMP): If a program is composed of multiple computations (parallelized with possibly different parallel frameworks), all computations *register* with the same RML instance that assigns a subset of the available threads to each computation. Moreover, if the number of computations registered with a RML changes, threads are redistributed between computations. As a result, the number of threads assigned to a computation can vary over time.

Figure 6.4(b) shows a program composed of two TBB task scheduler-based computations (TS) and one OpenMP-based (OMP) computation (computations are registered with the same RML that has N threads). (The example omits higher-level details about the program, e.g., the parallel algorithm templates it uses.) Upcoming examples consider only TBB task schedulers but not OpenMP runtimes to be registered with an RML. This simplifies the discussion but is not a real restriction of either standard TBB nor TBB-NUMA.

6.2.5 Threads

In addition to task schedulers, threads are registered and managed by the RML as well. The RML manages two types of threads: (1) The RML automatically creates $N - 1$ worker threads (N is the number of cores of the system); (2) master threads are created by the user program with a suitable system library (e.g., `pthreads`) and are registered the first time they use a parallel construct.

6.3 Implementing NUMA support

Implementing support for NUMA-aware programming involves all layers of TBB-NUMA's architecture, and, in some cases, it requires tight coupling between the layers. The discussion in this section follows the layers bottom to top. This section focuses on aspects specific to TBB-NUMA, contrasting it to standard TBB where interesting.

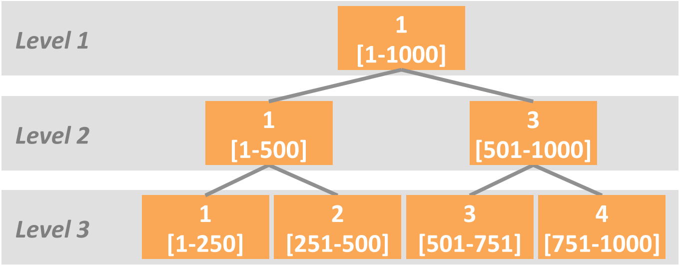
6.3.1 Threads

Previous parallel frameworks (e.g., Intel OpenMP) allow the user to pin thread to cores, i.e., each thread is allowed to execute only at one *specific core*. TBB-NUMA automatically pins each thread to a *specific processor*. If a thread is pinned to a processor, the thread is allowed to execute at *any core of that specific processor*, but not at cores of any other processor. Threads are pinned to processors when they are created by the RML (worker threads) or when they register with the RML (master threads). Threads are distributed round-robin across the processors of a system (the first thread registered/created is pinned to Processor 1, the second to Processor 2, and so on); we assume all processors are identical with regard to number and capabilities of cores, thus the RML guarantees that there is an approximately equal number of threads pinned to each processor at any given point of time. TBB-NUMA is aware of the memory system's layout and threads are pinned to processors without user intervention.

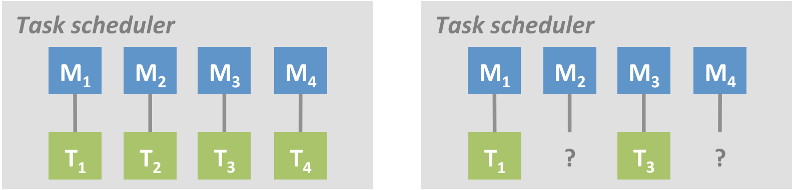
The OS scheduler has fewer constraints with per-processor pinning than with per-core pinning, thus it can possibly balance load better if there are external (non-TBB) threads running on the system. If threads are not pinned, the TBB-NUMA runtime cannot give any guarantees to the layers above the threading layer, hence per-processor pinning is the minimal constraint that must be imposed on the OS scheduler to support NUMA-awareness.

6.3.2 Resource Management Layer

Similar to the RML in standard TBB, the TBB-NUMA RML distributes threads between all registered task schedulers. In addition to the standard TBB, the TBB-NUMA RML is aware



(a) Task tree with task-to-thread affinities (top) and partitioning (bottom).



(b) TS fully populated.

(c) TS partially populated.

Figure 6.6: Mailboxing (standard TBB).

of which processor each registered thread is pinned to and it distributes threads so that in each registered task scheduler there is an approximately equal number of threads from each processor. Let us assume an example program with two task schedulers running on a 2-processor 8-core system; there are 8 threads registered with the RML. In this case the RML assigns four threads to each task scheduler, with two threads pinned to Processor 1 and with two threads pinned to Processor 2. Distributing threads this way guarantees that each task scheduler has access to all memory system resources (i.e., last-level caches, memory controllers, and cross-chip interconnects) and unfortunate assignments like that in Figure 6.2 are avoided.

6.3.3 Standard TBB task scheduler

In standard TBB each task scheduler has a set of mailboxes, the number of mailboxes is usually set to the number of cores of the machine. When the RML assigns a thread to a task scheduler, the thread connects to a *randomly* chosen mailbox.

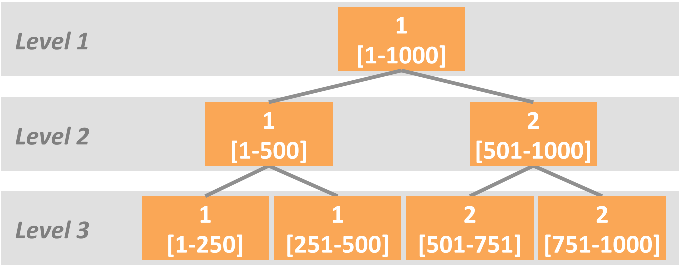
Figure 6.6(b) shows a task scheduler with four mailboxes ($M_1 \ldots M_4$); the task scheduler is allocated four threads by the RML ($T_1 \ldots T_4$). During its lifetime, a task scheduler can be allocated different numbers of threads. Moreover, even if the same set of threads are allocated to a task scheduler, each thread can be connected to a different mailbox during the lifetime of the task scheduler (e.g., if a thread leaves and then re-joins a scheduler, the thread can be assigned to a randomly chosen mailbox, and thus possibly not to the same mailbox as it was connected to before it left the task scheduler). We refer to the combination of the number of threads in a task scheduler and the mailboxes used by these threads as a *task scheduler configuration*. Figure 6.6(b) shows the task scheduler in a fully populated configuration in which there is a thread connected to each mailbox; in this configuration each thread T_i is connected to mailbox M_i. In contrast, Figure 6.6(c) shows the task scheduler in a partially populated configuration in which only mailboxes M_1 and M_3 are used (by threads T_1 and T_3, respectively).

If a thread creates a task tree and then submits it to the task scheduler for execution, moreover, tasks in the tree have affinities to mailboxes, these tasks are inserted into the creator thread's local deque as well as into the mailbox corresponding to the task's affinity value. Threads in the task scheduler attempt to obtain tasks to execute. First, a thread tries to receive a task from the mailbox the thread is connected to (Rule 4 in Figure 6.5). If the mailbox is empty, the thread falls back trying to remove a task from the shared queue (Rule 5) or to randomly stealing a task (Rule 6).

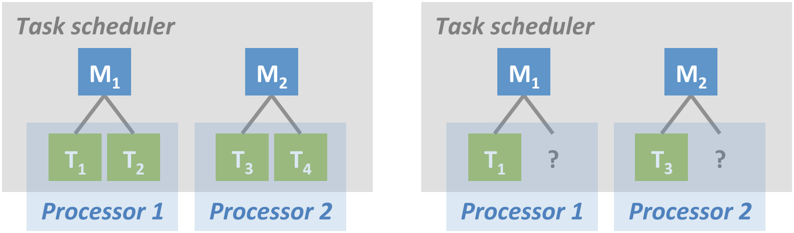
Figure 6.6(a) shows a task tree with three levels; tasks in the tree have affinities specified for the task scheduler configuration shown in Figure 6.6(b). The example corresponds to a possible partitioning of an iteration space of 1000 iterations (as done, e.g., by the `parallel_for` pattern). If the task tree is repeatedly executed with the same set of affinities and with the same task scheduler configuration, the same subset of the iteration space is sent to the same mailbox. Thus, each thread processes the same subset of the iteration space. As a result, the computation has good cache locality. In standard TBB the `parallel_for` algorithm template is based on this principle: If used with an `affinity_partitioner` object, the `parallel_for` template stores task affinities into the partitioner object and reuses them on future executions.

In standard TBB affinities are only a *hint* on the preferred place of a task's execution, that is, the task scheduler is allowed to *ignore* task affinities to better balance the load. More specifically, tasks are not executed by the thread specified by affinities for three main reasons: (1) *steals*, (2) *revokes*, and (3) *changes in the task scheduler configuration*. First, affinitized tasks are also inserted into the local deque of the thread that creates them, thus they can be *stolen* before they are received at the mailbox they have affinity to. Second, if the thread that created tasks executes Rule 3, it can *revoke* tasks from mailboxes and execute them locally itself. Third, the task scheduler configuration *changes at runtime*; Figure 6.6(c) shows the task scheduler partially populated with only two threads. For this task scheduler configuration the affinities of the task tree do not make sense (because there is no thread connected to mailboxes M_2 and M_4), therefore all tasks of the tree (including those with affinity to threads T_2 and T_4) will be executed either by thread T_1 or by thread T_3.

Programmers cannot foresee dynamically changing runtime conditions, thus TBB does not encourage programmers to specify affinities for tasks. Instead, TBB keeps task affinities internal to the library's implementation. The only case where TBB uses affinities, even internally, is the `parallel_for` template used in combination with an `affinity_partitioner` object. The `parallel_for` algorithm template automatically and internally adapts the affinities of task trees to match the effective place of execution. E.g., let us assume a task tree generated by a `parallel_for` partitioned with an `affinity_partitioner`. Let us furthermore assume that, when unfolding the task tree, the partitioner sets the affinity value of a task `A` in the tree to value 1 (indicating that `A` is preferably executed by the thread connected to mailbox M_1). If task `A` is executed by a thread connected to a different mailbox M_i (because of any of the previously mentioned three reasons), the runtime overwrites the partitioner's record about the task's affinity; after the update the task has its affinity value set to i and this value is used when the task is re-executed. This strategy is beneficial assuming that the affinities specified by the partitioner match the configuration of the task scheduler for some time in the future. However, updating task affinities is not acceptable in NUMA systems: *In NUMA systems task affinities must stay constant because each task must execute at the processor where its data is located.*



(a) Task tree with task-to-processor affinities (top) and partitioning (bottom).



(b) TS fully populated. (c) TS partially populated.

Figure 6.7: Mailboxing (TBB-NUMA).

$4'$ The task removed from mailbox M_i, where the current thread is pinned to Processor i.

$5'$ The task removed from the task scheduler's shared queue i, where the current thread is pinned to Processor i.

$5''$ The task removed from the task scheduler's shared queue 0 (queue w/o affinity for any processor).

$5'''$ The task removed from the task scheduler's shared queue k, where the current thread is pinned to Processor i and $i \neq k$.

Figure 6.8: Rules substituted by TBB-NUMA to fetch next task (relative to standard TBB).

6.3.4 TBB-NUMA task scheduler

Unlike in standard TBB, in TBB-NUMA the programmer can specify task affinities explicitly. Task affinities are *hints* in TBB-NUMA as well, but, unlike in standard TBB, affinities are *sticky* in TBB-NUMA. That is, the TBB-NUMA runtime is not allowed to modify a task's affinity when the task is not executed on a different processor (i.e., a processor not originally intended by the programmer). To help the TBB-NUMA task scheduler still honor affinities (and balance load at the same time), the TBB-NUMA runtime implements a set of optimizations in addition to standard TBB. We first define the semantics of task affinities in TBB-NUMA, then we describe the optimization to handle scheduler configuration changes, steals, and revokes.

Task-to-processor affinities

In TBB-NUMA tasks have affinity to a processor (instead of a mailbox as in standard TBB). That is, a task with an affinity value equal to i is not meant to be executed by the *single thread* connected to mailbox M_i as in TBB, but by *any thread* running at Processor i. Thus, TBB-NUMA replaces Rule 4 of standard TBB (Figure 6.5) by Rule 4' (Figure 6.8). Because affinity values have a clear meaning backed by the TBB-NUMA runtime system, the programmer is allowed to use them (either directly by using the task scheduler interface or indirectly by reusing parallel algorithm templates). To support per-processor task affinities, the number of mailboxes of a TBB-NUMA task scheduler is equal to the number of processors of the machine (i.e., on a P-processor system there are P mailboxes). A task with an affinity value of i is inserted into mailbox M_i and, as the RML allows only threads pinned to Processor i to use this mailbox, the task is slated to be executed at the appropriate processor. Figure 6.7(b) shows the layout of a task scheduler populated with 4 threads (on a 2-processor system); two threads are pinned to each processor.

Figure 6.7(a) shows a NUMA-aware affinitization of a task tree (also for a 2-processor system). In the example the first half of the iteration space (iterations `[1-500]`) is mapped to Processor 1, the second half (iterations `[501-1000]`) is mapped to Processor 2. If data accessed by iterations `[1-500]` (`[501-1000]`) is allocated at Processor 1 (Processor 2), the computation has good data locality and thus good performance with TBB-NUMA.

Handling configuration changes (Cause 1)

TBB-NUMA handles the problem of changing task scheduler configurations by hardware-aware resource management. The TBB-NUMA RML allocates threads to task schedulers so that each scheduler has an approximately equal number of threads pinned to each processor. As a result, in every task scheduler the number of threads using each per-processor mailbox is approximately the same. Thus, every task scheduler has approximately the same share of each processor's computational and memory system resources. Figure 6.7(c) shows a task scheduler populated with two threads (two threads less than in Figure 6.7(b)). Each mailbox is served by one thread pinned to each processor and the affinitized task tree will execute with good data locality, just as when the task scheduler is fully populated (Figure 6.7(b)).

In some scenarios (when the number of thread schedulers registered with the RML is close to or is larger than the total number of threads registered with the RML) threads cannot be allocated to schedulers so that each mailbox is served by an equal number of threads. But as long as the number of registered schedulers is low (which is frequently the case in practice), it is possible to evenly distribute threads between task schedulers.

Handling steals (Cause 2)

An affinitized task is present at two places: in the local deque of the thread that created it and in the mailbox it is sent to. Affinitization is successful if the task is removed from the mailbox by the thread connected to the mailbox. Affinitization is unsuccessful if the task is stolen by a thread that has no work to do and it has fallen back to random stealing (Rule 6) (the stealing thread that has fallen through Rules 1–5 and is obtaining work according to Rule 6).

Standard TBB prevents a stealing thread from obtaining an affinitized task if there is a good chance that the task is going to be removed from the destination mailbox soon. Before stealing an affinitized task, each thread checks if the destination mailbox of the task is *idle* (by calling the `is_idle()` function shown in Figure 6.9(a)). If the mailbox is marked as idle, the stealing

```
void set_idle(affinity id, bool flag) {
    mailbox[id].flag = flag;
}
bool is_idle(affinity id) {
    return mailbox[id].flag;
}
```

(a) Standard TBB.

```
void set_idle(affinity id, bool flag) {
    mailbox[id].counter += flag ? 1 : -1;
}
bool is_idle(affinity id) {
    bool threads_expected = num_threads_active[id]
                           < num_threads_allotted[id];
    return mailbox[id].counter > 0 || threads_expected;
}
```

(b) TBB-NUMA.

Figure 6.9: Indicating idleness.

thread *bypasses* the mailbox and tries to obtain a task from some other thread.

A mailbox is marked as idle (by the `set_idle()` function in Figure 6.9(a)) in two cases: (1) when a thread falls through dequeuing from its local deque (Rule 3), but has not yet peeked at its mailbox yet (Rule 4), and (2) the thread connected to the mailbox has left the task scheduler. In the first case bypassing is well-justified because the task will be shortly received by the thread connected to the mailbox. The second case needs more explanation. A thread leaves the task scheduler when there is no work available for it. Alternatively, the RML can revoke the thread from the current task scheduler and then assign it to some other task scheduler. If the thread is associated with the current task scheduler again, it will empty its mailbox and the task will be executed at the intended location. But if the thread is permanently assigned to some other task scheduler, the task will be revoked (executed locally) by the thread spawning it and the task's affinity will be updated (which is in conformity with the TBB principle of non-constant task affinities).

In TBB-NUMA task affinities are constant (by design), thus the idling mechanism of standard TBB must be revised. If a task is executed repeatedly, due to the constant affinities, a task with affinity value i will be submitted to the same mailbox M_i over and over again. If there are no threads connected to mailbox M_i, other threads in the scheduler will bypass mailbox M_i (assuming the idling mechanism of standard TBB). Bypasses reject work thus they can result in a high performance penalty. The idling mechanism of standard TBB must be updated also because TBB-NUMA allows multiple threads attached to a single mailbox.

TBB-NUMA uses an idling mechanism that is tightly coupled with resource management. The idling mechanism of TBB-NUMA is based on incrementing/decrementing a counter, as implemented by the `set_idle()` function shown in Figure 6.9(b). A thread increments the counter before receiving from its mailbox and decrements it after it has received a task. Unlike with standard TBB, with TBB-NUMA a thread does not indicate idleness when it leaves the task scheduler. To allow stealing those tasks that are not likely to be picked up at their destination mailbox and thus provide good load balance, the `is_idle()` function inspects both the counter and the number of threads allocated/active in the destination's mailbox, which avoids

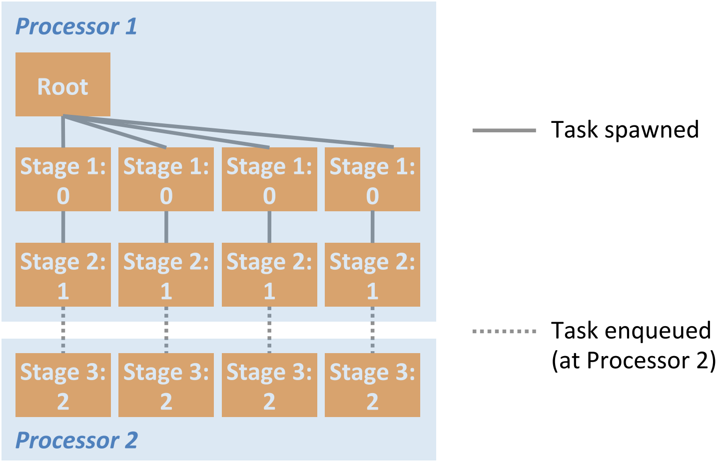


Figure 6.10: Shallow task tree: 2-stage pipeline with affinities.

unnecessary bypasses.

Handling revokes (Cause 3)

Affinitized tasks are inserted into the local deque of the thread that creates them. An affinitized task is *revoked* if the creator thread retrieves it (Rule 3) before it can be received at the destination processor. Unlike Rule 6, Rule 3 does not bypass affinitized tasks (to guarantee that each task is eventually executed before the program terminates). TBB-NUMA attempts to avoid revokes in two ways: by controlling task submission order via reflection (in case of wide task trees) and by detaching subtrees (in case of shallow task trees).

Controlling task submission order In wide task trees each task (except leafs) has at least two children tasks. When unfolding wide task trees, it is beneficial to submit tasks affinitized for the current processor last (the processor where the thread unfolding the tree executes). As Rule 3 retrieves tasks in LIFO order, tasks enqueued earlier have a chance to be picked up for execution at their destination thread before the creator thread revokes them.

For example, when unfolding level 2 of the task tree shown in Figure 6.7(a), the `parallel_for` template spawns the right subtree and continues executing the left subtree, if the current processor is Processor 1; otherwise it spawns the left subtree and continues executing the right subtree (assuming a 2-processor system). To control task submission order the creator thread must determine its current processor. The library supports this kind of reflection through the `task_scheduler_init::get_current_cpu()` call. This call is used internally by the `parallel_for` template. Code using the task scheduler directly can also rely on this reflection-based capability to control submission order.

Detaching subtrees Figure 6.10 shows a shallow task tree. E.g., the pipeline algorithm template of TBB generates shallow subtrees: The pipeline template generates a distinct subtree for each input element processed by the pipeline; each task in a subtree corresponds to a different pipeline stage. The task tree shown in Figure 6.10 corresponds to a 3-stage pipeline computation.

In a shallow task tree each task (except the root task) has only one child task that is executed next by the task scheduler (according to Rule 3). If the memory accesses of a pipeline computation are dominated by accesses to input elements, shallow task trees can be beneficial for L1/L2 cache locality, because tree shallowness guarantees that each input element is processed by the same thread (thus the input element is in the cache used by this thread). In some cases, however, a child task predominantly accesses data other than the input element it processes. Moreover, in some cases the child task's accesses do not hit in the L1/L2 cache and are served by last-level caches (or even by DRAM). In these cases it can be beneficial to schedule the child tasks at threads executing at well-defined processors to achieve good last-level cache/DRAM data locality.

E.g., in the pipeline computation in Figure 6.10, Stage 1 of the pipeline is not associated with any processor (its task-to-processor affinity is 0), but Stage 2 accesses data associated with Processor 1 and Stage 3 accesses data associated with Processor 2 (Stage 2 and Stage 3 have a task-to-processor affinity value of 1 and 2, respectively). Spawning affinitized tasks and sending them to the mailbox of the appropriate thread does not help in this case because the affinitized task will be revoked (Rule 3 has priority over Rule 4). To allow a child task to execute at the processor it is associated with, the child task must be *detached* from its parent task, that is, it must be sent to the destination processor without inserting it into the local queue.

Standard TBB facilitates detaching tasks through the `enqueue()` call. *Enqueued tasks* are inserted into a queue shared by all threads in a task scheduler, threads receive enqueued tasks according to Rule 5. In standard TBB enqueued tasks are not allowed to have affinities to threads. TBB-NUMA extends standard TBB by allowing enqueued tasks to have affinities as well: the TBB-NUMA task scheduler has $P + 1$ shared queues (assuming a P-processor system), tasks with affinity for Processor i are enqueued at Q_i, $1 \leq i \leq P$, tasks with no task-to-processor affinity defined are enqueued at Q_0, thus Rule 5 of TBB is replaced by a set of rules in TBB-NUMA (Rule $5'$-Rule $5'''$ in Figure 6.8).

To illustrate how enqueuing handles revokes, let us consider the 3-stage pipeline example again (Figure 6.10). Let us assume that the root task runs at Processor 1 (as shown in the figure). Stage 1 has no task-to-processor affinity, thus the root task unfolds Stage 1 tasks using spawns. Let us assume that Stage 1 tasks are then also executed at Processor 1. The next stage (Stage 2) has affinity for Processor 1, but as all Stage 1 tasks are already running at Processor 1, Stage 2 tasks do not have to be detached (thus they are spawned). When, however, the task tree is unfolded further (i.e., Stage 3 tasks are created), these tasks have affinity for Processor 2 (a processor different from the current processor), thus Stage 3 tasks are not spawned but enqueued (with affinity for Processor 2). Threads at Processor 2 will then dequeue these tasks (Rule $5'$) and each stage is executed where the programmer originally intended. Threads at Processor 1 (the threads that originally unfolded the upper levels of the task tree) in the meantime unfold new subtrees to process any remaining input elements.

The decision whether to spawn or to enqueue tasks when unfolding a task tree depends on the processor the current thread is pinned to. The TBB-NUMA pipeline template uses reflection to determine the current thread's processor. E.g., if the root task of the example in Figure 6.10 executes at Processor 2 instead of Processor 1, enqueuing is used already when unfolding Stage 2. Finally, similar to the affinity of mailboxed tasks, the affinity of enqueued tasks is a *hint* on the preferred place of execution, that is, if a thread cannot get a task from the shared queue associated with its processor (i.e., Rule $5'$ fails), the thread will try all other queues in the task scheduler (i.e., it will fall back to Rules $5''$ and $5'''$).

6.3.5 Programming with TBB-NUMA

TBB-NUMA extends TBB, that is, the programmer can define at runtime which rules the task scheduler uses, the rules of standard TBB or rules specific to TBB-NUMA. If TBB-NUMA is enabled, the `parallel_for` algorithm template can be used with an additional parameter, a data distribution object that specifies the distribution of data for the iteration space processed by the loop. TBB-NUMA includes a set of predefined data distributions (e.g., the block-cyclic distribution [12] shown in Figure 6.7(a) for a 2-processor system). If needed, the programmer can define custom data distributions: The `parallel_for` template interfaces with data distributions through a single method; this method is used to determine the affinity of a subrange of the iteration space when the task tree corresponding to the iteration space is unfolded. The `pipeline` template allows specifying the per-processor affinity of pipeline stages. Finally, with TBB-NUMA the semantics of task affinities is clearly defined (task-to-processor affinity), thus the programmer can use task affinities directly with the task scheduler interface.

In addition to specifying hints on the schedule of computations, TBB-NUMA defines helper functions to enforce data distributions on memory regions as well. Data distributions are enforced through memory migrations (e.g., through the `move_pages()` system call in Linux). Both data distributions and computation schedules depend on the actual hardware configuration. TBB-NUMA determines the number of processors at runtime and passes on this information to user programs. As a result, programs can be parametrized for a generic NUMA system and are thus portable.

6.4 Evaluation

The evaluation presented in this section attempts to answer three questions: (1) do optimizations improve data locality and performance (Section 6.4.2), (2) are optimizations composable (Section 6.4.3), and (3) are optimizations portable (Section 6.4.4).

6.4.1 Experimental setup

Three machines are used to run experiments (see Table 6.1).

	Intel E7-4830	Intel E5520	AMD 6212
Microarchitecture	Westmere	Nehalem	Bulldozer
# of processors/cores	4/32	2/8	4/16
Main memory	4x16 GB	2x24 GB	4x32 GB
Cross-chip interconnect	QPI	QPI	HT
Last-level cache	4x24 MB	2x8 MB	4x8 MB

Table 6.1: Hardware configuration (as reported by Linux).

We use five programs for performance evaluation, three programs use parallel algorithm templates and two programs use the task scheduler interface directly (see Table 6.2).

Program	Input	Benchmark suite	Parallel algorithm template used
`cg`	input size C	NAS PB	`parallel_for`
`mg`	input size C	NAS PB	`parallel_for`
`streamc.`	10M input points	PARSEC	none
`fluida.`	500K particles/500 frames	PARSEC	none
`ferret`	database (700M images)/ 3500 input images	PARSEC	`pipeline`

Table 6.2: Benchmark programs.

6.4.2 Data locality optimizations

Performance in NUMA systems depends on two aspects: the data distribution policy used and the policy used to schedule computations. We evaluate performance for a series of different execution scenarios; an execution scenario is defined by the pair (data distribution policy, computation schedule policy) used. Two execution scenarios consecutively listed in the evaluation differ in only one aspect, that is, they differ either in the data distribution policy used or the computation schedule policy used, but not both. Performance optimizations for loop-parallel programs are evaluated in five execution scenarios:

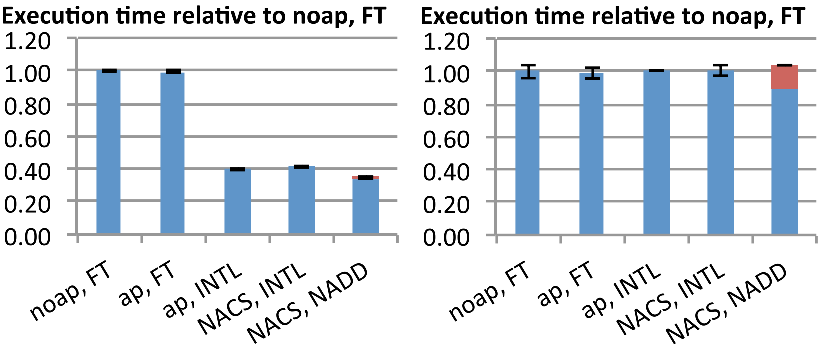
(noap, FT) Default version of the program that does not use task affinitization and uses the **f**irst-**t**ouch page placement policy (default policy in many OSs, e.g., Linux).

(noap, INTL) Default version of the program used with the **int**er**l**eaved page placement policy. The interleaved page placement policy distributes pages across processors in a round-robin fashion. Interleaved page placement is recommended for source-level optimizations by [55] and is used in automatic systems as well [27]. Interleaved page placement improves performance by reducing contention on memory interfaces, but it does not reduce the number of remote memory accesses. In many systems (including the systems in Table 6.1) interleaved placement is equivalent to disabling NUMA (in the BIOS).

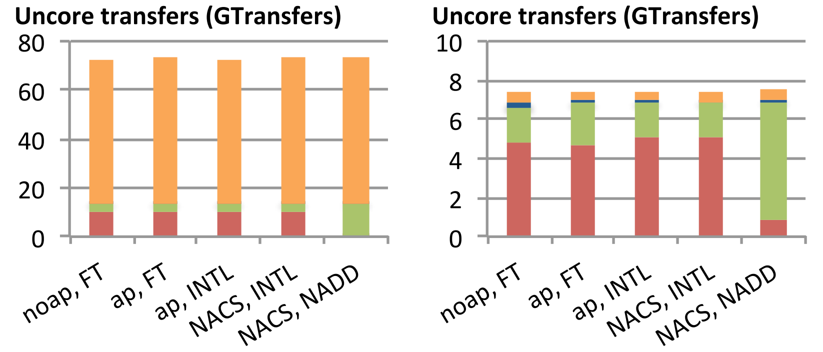
(ap, INTL) The parallel loops of the program are affinitized with the TBB-standard `affinity_partitioner` [86]; pages are placed interleaved, as in the previous configuration. The `affinity_partitioner` is designed to improve cache performance. This configuration shows the benefits of using this partitioner.

(NACS, INTL) While the previous configurations can be achieved with standard TBB, this configuration is achievable only with TBB-NUMA. This configuration uses **N**UMA-**A**ware **C**onfiguration **S**cheduling (NACS), that is, the task scheduler is given hints about the distribution of data in memory. Normally, NACS effects both caching and DRAM data locality. However, to assess how NACS effects *caching only*, data distribution is not enforced in this configuration. Instead, the interleaved page placement policy is used, thus this configuration differs only in one parameter (the schedule of computations) from the previous configuration.

(NACS, NADD) **N**UMA-**A**ware **D**ata **D**istribution is enforced (in addition to NACS in the previous configuration). The performance results in this configuration show the benefits due to both cache and DRAM data locality.



(a) `cg` performance. (b) `mg` performance.



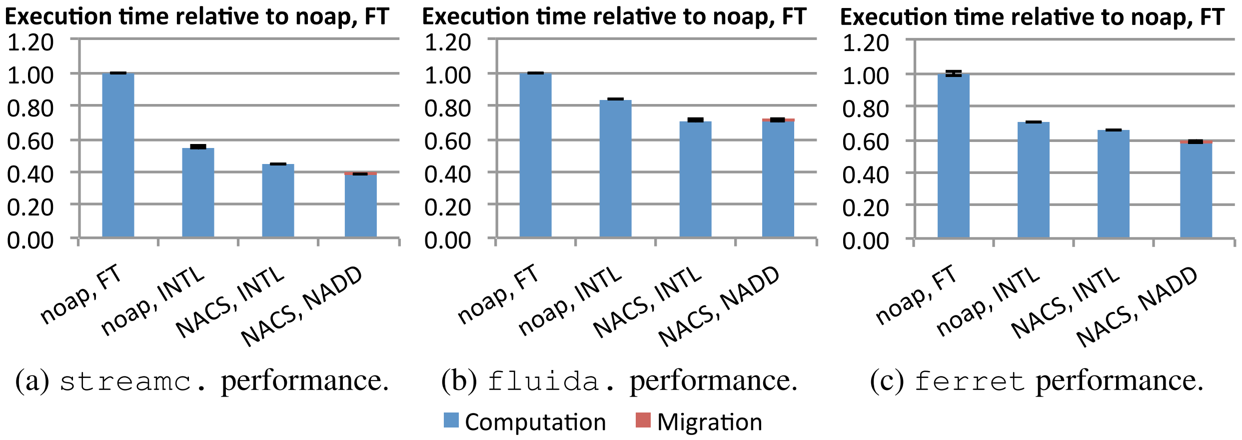
(c) `cg` uncore traffic. (d) `mg` uncore traffic.

Figure 6.11: Performance and uncore traffic of loop-parallel programs w/o contention (Westmere).

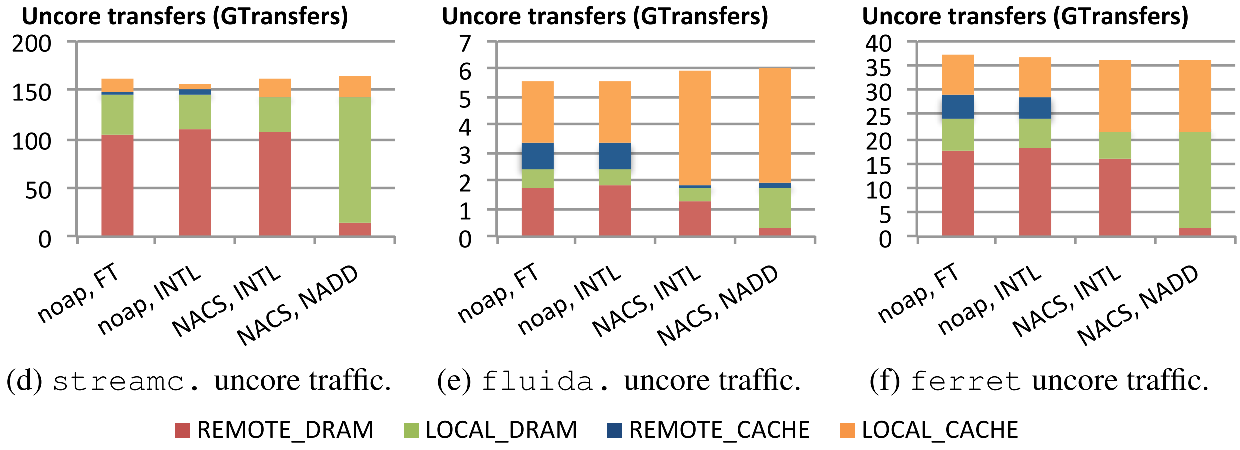
The evaluation described in this section uses the 4-processor 32-core Westmere system (see Section 6.4.4 for evaluation on the other systems). Figure 6.11(a) and 6.11(b) show the relative execution time of `cg` and `mg` in all five configurations. The execution times are relative to the `(noap, FT)` configuration, which has a relative execution time of 1.0. The lower the relative execution time measured in a configuration, the better the performance of the program in that configuration.

Relative to the best-performing configuration achievable with standard TBB, `(ap, INTL)`, `cg` improves 18% (relative execution time of 0.34 with `(NACS,NADD)` vs. relative execution time of 0.4 with `(ap, INTL)`, the best configuration that can be realized with standard TBB). The computation time of `mg` improves around 12%, but its overall performance becomes slightly worse than the best configuration achievable with standard TBB because the cost of data migration (distributing data) cancels the improvement in computation time.

To show that performance optimizations improve both cache- and DRAM locality, we measure the number of uncore transfers a program generates in each of the five examined configurations. There are four types of uncore transfers: local cache/DRAM accesses and remote cache/DRAM accesses; the latencies of the different types of transfers are listed in Figure 1.1 for a 2-processor system. Figures 6.11(c) and 6.11(d) show the uncore traffic breakdown of `cg` and



(a) `streamc.` performance. (b) `fluida.` performance. (c) `ferret` performance.

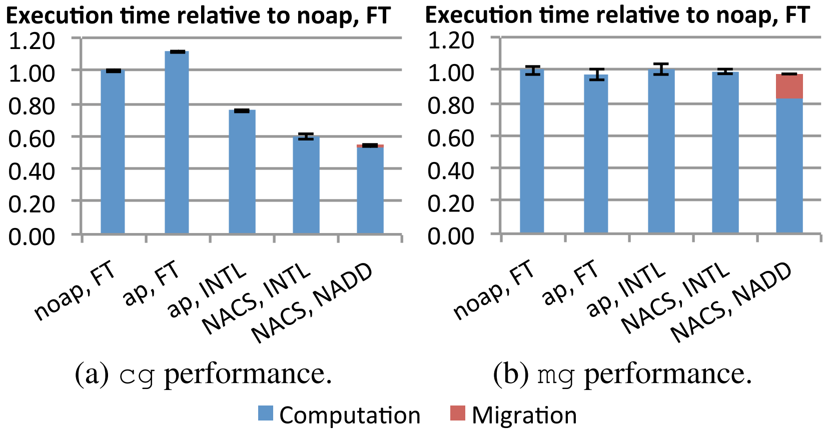


(d) `streamc.` uncore traffic. (e) `fluida.` uncore traffic. (f) `ferret` uncore traffic.

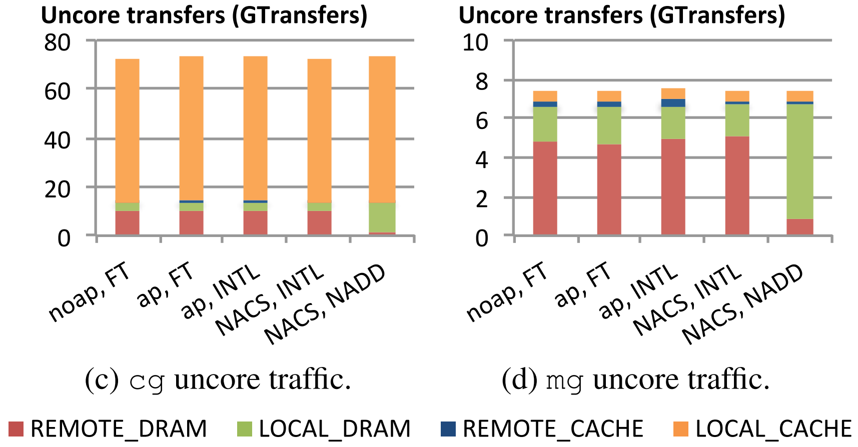
Figure 6.12: Performance and uncore traffic of non–loop-based programs w/o contention (Westmere).

`mg`, respectively. In the `(NACS, NADD)` configuration almost all remote memory accesses are eliminated (relative to both the baseline `(noap, FT)` and the `(ap, INTL)` configuration).

Figures 6.12(a), 6.12(b), and 6.12(c) show the performance of the remaining three, non–loop-based programs. The `affinity_partitioner` can be used only with loop-based programs, thus the `(ap, FT)` and `(ap, INTL)` configurations are invalid for non–loop-based programs. As a result, non-loop-based programs are evaluated in four instead of five configurations: the invalid configurations are replaced by the `(noap, INTL)` configuration (version of the program with no affinities specified, interleaved page placement policy). The principle that two subsequently listed configurations change only in a single parameter still holds after this change. Similar to loop-based programs, the two last configuration scenarios shown in the figures can be realized only with TBB-NUMA. For the non–loop-based programs NUMA-aware memory system optimizations result in performance improvements between 16–44% over the best possible configuration achievable with standard TBB (e.g., in case of `streamcluster` relative execution time of 0.38 with `(NACS, NADD)` vs. relative execution time of 0.55 with `(noap, INTL)`, the best configuration that can be realized with standard TBB). Figures 6.12(d), 6.12(e), and 6.12(f) show the breakdown of uncore traffic for all three programs in all configurations. Memory system optimizations reduce the number of remote accesses for these programs, too.



(a) `cg` performance. (b) `mg` performance.



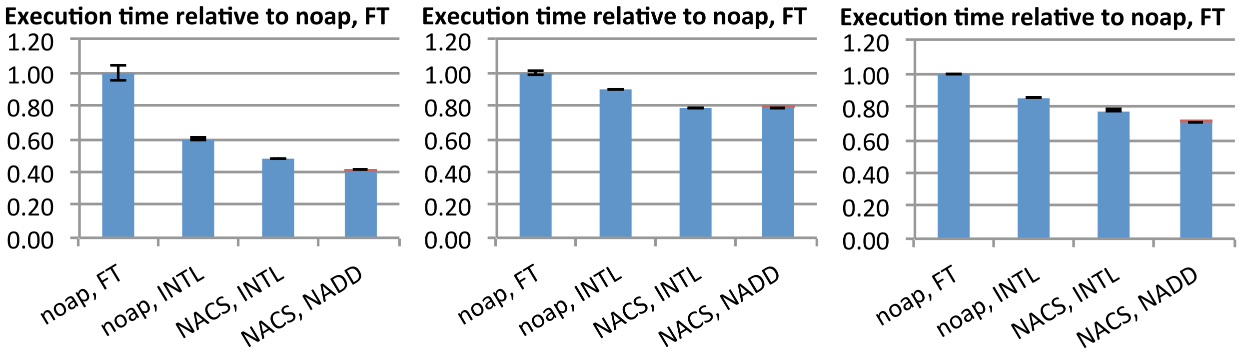
(c) `cg` uncore traffic. (d) `mg` uncore traffic.

Figure 6.13: Performance and uncore traffic of loop-parallel programs w/ contention (Westmere).

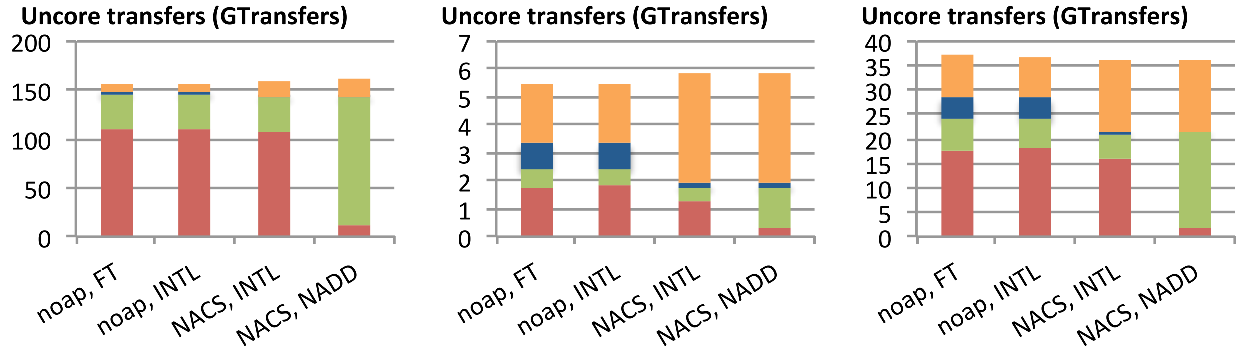
6.4.3 Composability

This section evaluates how the properties of memory system optimizations are preserved when only a part of the hardware is available for executing optimized computations. Each benchmark program is executed concurrently with a contender computation. The contender computation is parallelized and demands all hardware resources (just as the benchmark program it is co-run with). The RML divides threads between the benchmark program and the contender program. This setup is similar to the scenario shown in Figure 6.4(b) with the difference that only two task schedulers (TS) but no OpenMP runtime (OMP) use the RML. The contender computation is floating-point intensive and its working set fits into the private L1/L2 caches of the cores. As a result, uncore transfers measured are predominantly caused by the benchmark program (and not by the contender computation).

Figures 6.13(a), 6.13(b), 6.14(a), 6.14(b), and 6.14(c), show the performance of all benchmark programs in all relevant configurations, Figures 6.13(c), 6.13(d), 6.14(d), 6.14(e), and 6.14(f) show the breakdown of uncore traffic corresponding to each program/configuration. Performance is reported as execution time relative to the `(noap, FT)` configuration with enabled contender. For each program/configuration performance results and the breakdown of uncore traffic is similar to the corresponding case with no contention. We record minor dif-



(a) `streamc.` performance. (b) `fluida.` performance. (c) `ferret` performance.



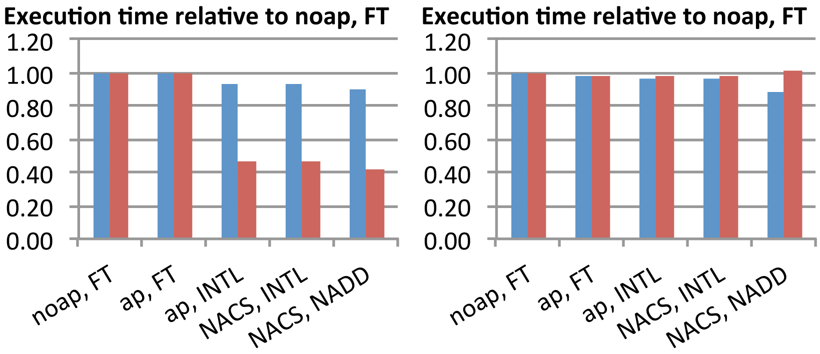
(d) `streamc.` uncore traffic. (e) `fluida.` uncore traffic. (f) `ferret` uncore traffic.

Figure 6.14: Performance and uncore traffic of non–loop-based programs w/ contention (Westmere).

ferences in relative performance numbers and uncore traffic because with contention there is a different amount of per thread cache capacity available to programs than without contention. In conclusion, the TBB-NUMA runtime preserves the properties of memory system optimizations even if only a part of the hardware is available.

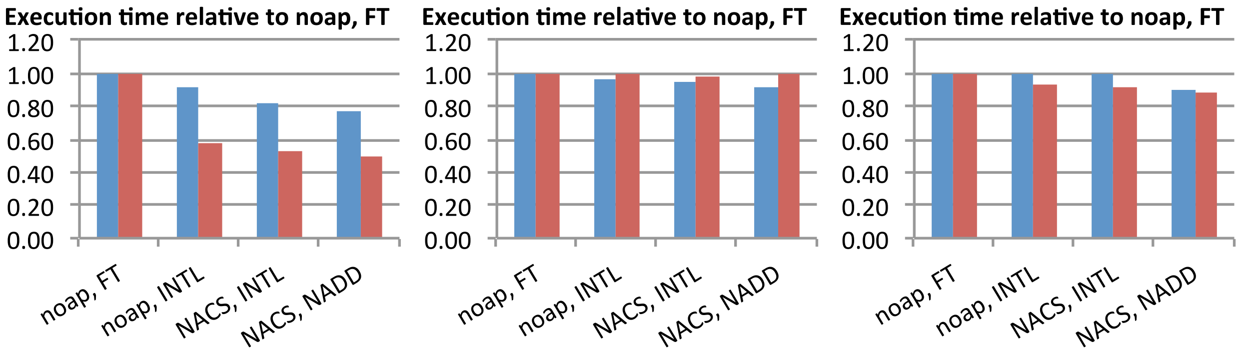
An interesting aspect is that using the `affinity_partitioner` with `cg` causes a slowdown under contention. The RML is shared with the contender computation and threads frequently "migrate" between the two computations (i.e., thread previously assigned to the task scheduler running `cg` are frequently reassigned to the task scheduler running the contender computation and vice versa). When a thread leaves a task scheduler (because the RML reassigned the thread to an other task scheduler), the mailbox of the thread (i.e., the mailbox in the task scheduler the thread was previously connected to) is marked as idle. Affinitized tasks present in a mailbox that is marked idle are not removed by stealing threads (i.e., treads looking for work; see Section 6.3.4 for further details). Instead, the tasks are kept in the mailbox until the thread that created them becomes available and can process them.

In summary, if threads frequently migrate between task schedulers, stealing threads often reject work (i.e., reject to execute tasks) by bypassing mailboxes in which work (i.e., affinitized tasks) is present. As a result, processor cores are often idle, which results in a slowdown in the



(a) `cg` performance.

(b) `mg` performance.



(c) `streamc.` performance.

(d) `fluida.` performance.

(e) `ferret` performance.

Figure 6.15: Performance w/o contention (Nehalem and Bulldozer).

case of the `cg` benchmark. The TBB-NUMA task scheduler is coupled with resource allocation that helps avoid unnecessary bypasses.

6.4.4 Portability

To show that memory system optimizations are portable, we run the same set of programs on two additional systems. As memory system optimizations are implemented for a generic NUMA system, the programs are executed on these systems without modification. Performance results are shown in Figures 6.15(a)–6.15(e); the variation of the measurement readings is negligible. On the Nehalem, optimizations result in 3–18% performance improvement over the the best configuration that can be realized with standard TBB. On the Bulldozer we measure 6–18% improvement (and no improvement (`fluidanimate`) resp. a 3% slowdown (`mg`)).

6.5 Conclusions

The work presented in this chapter contributes to the set of NUMA-aware development tools that aid programmers in finding and fixing NUMA-related performance problems. TBB-NUMA is an extension to standard TBB and the programmer can decide when to use NUMA-specific functionalities. TBB-NUMA supports *portable* and *composable* software for NUMA systems by providing a unified interface to the runtime system as well as by implementing memory-system-aware resource management.

There are several tools to provide information about NUMA performance bottlenecks, but programmers so far lack a unified way to control the execution of parallel programs on NUMA systems. TBB-NUMA allows the programmer to pass directives (based on insights and/or performance monitoring information) about computation *and* data placement to the runtime system. With NUMA systems increasing in size we expect the gap between local and remote memory accesses to increase as well, thus we expect data locality optimizations to be even more important in the future.

7

Related work

The contributions of this thesis can be grouped into two main areas: (1) analyzing application memory system performance and (2) performance optimization techniques that target the memory system. The following sections present work related to each area.

7.1 Memory system performance analysis

7.1.1 Memory controller performance

Molka et al. [73] analyze in detail the memory system performance of the Intel Nehalem: They use sophisticated synthetic benchmarks to determine the bandwidth and latency of memory accesses to different levels of the memory hierarchy (including local/remote DRAM accesses). In later work, Hackenberg et al. [39] compare the performance of the Intel Nehalem with the AMD Shanghai by using the same methodology. Their methodology can be used to measure memory bandwidth and memory access latency and also to analyze the impact of cache coherency. However, they consider the properties of the different interconnects of multicore chips only in isolation, and not the interaction between them. The measurements presented in Chapter 2 consider only cache lines in the E, M and I states, but the measurement framework can be extended to measure the sharing of bandwidth to cache lines in other coherency states as well. We leave this investigation for future work.

Yang et al. [111] examine the dependence of application performance on memory and thread placement in an AMD Opteron-based NUMA machine. They quantify execution time, but do not measure low-level hardware issues (e.g., cache coherency traffic that is significant in some execution configuration of their benchmarks). Mandal et al. [66] model the memory bandwidth and memory access latency of commercially available systems (among those also the Nehalem) as a function of concurrent memory references in the system. However, it is difficult to extend their model to include sharing between multiple types of memory controllers, because requests can be produced at different rates through the on-chip memory controller and the QPI. Their pointer-chasing benchmark also encounters inter-core misses, so the values reported are slightly dependent on the cache sharing behavior of the evaluated systems.

Tuduce et al. [105] describe the asymmetries of the memory system of a multicore multiprocessor with a shared off-chip memory controller. The authors argue that the hardware performance measurement unit of modern CPUs should be improved to allow system software to detect and to avoid the performance bottlenecks of the underlying architecture. In this thesis we analyze a different, Nehalem-based architecture with multiple types of memory controllers and find that asymmetries are present in this system as well, however, they are of a different nature. In the Nehalem-based system analyzed in this thesis the fairness of the arbitration mechanism (the Global Queue) is crucial for performance, therefore better monitoring of this subsystem should be made possible in successors of the Nehalem microarchitecture.

Awasthi et al. [6] investigate the problem of data placement in a system with multiple memory controllers. They identify the performance degradation caused by overloading a single memory controller in the system, and attribute the costs to increased queuing delays and decreased DRAM row-buffer hit rates. However, their evaluation focuses more on future architectures and less on present and near-future systems. Blagodurov et al. [14] describe the sources of performance degradation that cause slowdowns to programs co-executing on NUMA systems (the remote latency and interconnect degradation). The NUMA penalty proposed in this thesis quantifies the slowdown that a single program experiences due to both factors. Moreover, this thesis also considers issues related to the fairness of the memory controller's queuing system.

7.1.2 Shared resource contention

Chandra et al. [20] use analytical models to predict the inter-thread cache contention of co-executing programs. Jiang et al. [47] prove that the complexity of optimal co-scheduling on chip multiprocessor systems is NP-complete. Mars et al. [69, 70] describe several systems that characterize resource contention at runtime.

Zhuravlev et al. [113] quantify the performance impact of contention for different types of shared resources in recent multicore-multiprocessor architectures. Furthermore, they compare the accuracy of different metrics used to characterize the interference of co-executing programs. They find that the MPKI is reasonably accurate, thus we also use this metric in our work. The work of Blagodurov et al. [13, 14] extends Zhuravlev's work and considers contention for shared resources on NUMA systems as well. Tang et al. [100] analyze the impact of contention for shared resources in NUMA-multicore systems on the performance of Google workloads.

7.1.3 Data sharing

Several authors have noticed the problem of data sharing in NUMA systems. Thekkath et al. [101] show that clustering threads based on the level of data sharing introduces load imbalances and cancels the benefits of data locality. Tam et al. [96] schedule threads with a high degree of data sharing onto the same last-level cache. Verghese et al. [107] describe OS-level dynamic page migration that migrates thread-private pages but does not consider shared pages. Therefore, their results show a significant amount of remaining remote memory accesses.

An approach similar to the work of Verghese et al. is described by Nikolopoulos et al. [77]. Remote memory accesses are not eliminated by this approach either. Marathe et al. [67] describe the profile-based memory placement methodology we use, but they do not discuss how data sharing influences the effectiveness of their method. Tikir et al. [103] present a profile-based page placement scheme. Although successful in reducing the percentage of remote memory accesses for many benchmark programs, their method is not able to eliminate a significant portion of remote memory accesses for some programs, possibly due to data sharing.

7.2 Performance optimizations

7.2.1 Reducing shared resource contention

There are several methods to mitigate shared resource contention. Qureshi et al. [84] partition caches between concurrently executing processes. Tam et al. [97] identify the size of cache partitions on runtime. Mars et al. [70] halt low priority processes when contention is detected. Herdrich et al. [40] analyze the effectiveness of frequency scaling and clock modulation to reduce shared resource contention. Awasthi et al. [6] show that data migration and adaptive memory allocation can be used to reduce memory controller overhead in systems with multiple memory controllers (such as NUMAs). OS process scheduling is also well suited for reducing contention on shared caches, as described by Fedorova et al. [33].

Fedorova et al. [34] present an OS scheduling algorithm that reduces the performance degradation of programs co-executed on multicore systems. Banikazemi et al. [8] describe a cache model for a process scheduler that estimates the performance impact of program-to-core mapping in multicore systems. The process scheduler mechanism described by Knauerhase et al. [52] and Zhuravlev et al. [113] is most closely related to the N-MASS scheme presented in this dissertation. The schemes presented by both groups schedule processes so that each LLC must handle approximately equal cache pressure. These approaches were evaluated on SMPs with uniform memory access times. We show that cache balancing algorithms do not work well in NUMA systems if the memory allocation setup of the system is not considered. Many approaches on resource-aware scheduling target mostly the application class of multiprogrammed workloads, but the recent work of Dey et al. describes a resource-aware scheduler that can handle multithreaded programs as well [28].

Recent research proposed performance-asymmetric multicore processors (AMPs). In contrast to AMPs, the cores of a NUMA system have the same performance, but the memory system is asymmetric and programs have different performance on remote execution. Li et al. present an OS scheduler for AMPs [59]. They evaluate their system also on NUMA systems, but they do not account for cache contention. Saez et al. [88] and Koufaty et al. [54] independently describe a scheduler for AMPS based on the efficiency specialization principle. Their schedulers implement a strategy similar to the maximum-local policy presented in this thesis, but their system targets performance asymmetry instead of memory system asymmetry (compute-bound processes are scheduled onto high performance cores with larger priority than memory-bound processes).

7.2.2 Improving data locality

Many projects (including the `auto_partitioner` approach of standard TBB) target improving the cache hit rate of programs [24, 50, 51, 86, 101, 112]. There are, however, some programs whose cache hit rate cannot be easily improved (e.g., programs that process data sets larger than the size of the caches available on the machine used to execute the program) and in NUMA systems the locality of a program's cache misses is often critical to performance. Data locality optimizations are beneficial for programs that have a bad cache behavior and can thus be considered orthogonal to cache locality optimizations.

In the following we present related work in the area of data locality optimizations.

Automatic approaches

There exist many automatic approaches to improve the data locality of multithreaded programs. OSs are in a good position to increase data locality by migrating data close to the threads using them [13] and by replicating data across processors [27, 107]. These approaches have large performance benefits, but they face limitations in case of programs with data sharing, because no matter how threads/data are placed in the system, there will be some remote memory accesses to shared data. Tang et al. [100] show that scheduling threads close to the data they access can be beneficial for data locality, but in some cases cache contention counteracts the benefits of data locality.

Data placement based on data access profiles gathered on separate profiling runs can improve the performance of multithreaded programs, as shown by Marathe et al. [68]. Page placement can also be performed at runtime based on dynamic data access profiles [77, 79, 104]. Although these approaches are beneficial for performance, they work well only for programs that have little data shared between threads.

Su et al. [94] show that dynamic data access profiles can guide not only data placement, but also thread placement to improve data locality. Tam et al. [96] cluster the threads of a multithreaded program based on the amount of data they share. Threads of the same cluster are then scheduled onto cores connected to the same last-level cache to reduce the number of remote cache accesses. The programs considered by Tam et al. exhibit non-uniform data sharing (i.e., each thread shares data with only a subset of the threads of the program), but data sharing can hinder thread clustering, as noted by Thekkath et al. [101]. We expect information additional to data access profiles (e.g., information about a program's control flow [46]) can enable a profiler to better handle data and/or thread placement of programs with data sharing.

Tandir et al. [98] present an automatic compiler-based technique to improve data placement. It is not clear how their tool distributes loop iterations that access shared memory regions and have loop-carried dependences at the same time. Kandemir et al. [51] describe an automatic loop iteration distribution method based on a polyhedral model. Their method can optimize for data locality, however, in the presence of loop-carried dependences it inserts synchronization operations. We show that the number of additional synchronization operations can be kept low if the memory distribution of the program is carefully adjusted. Reddy et al. [85] describe an approach based on the polyhedral model that can handle both data placement and computation mapping for programs with affine loop nest sequences.

Language-based approaches

All previously mentioned approaches are fully automatic, that is, the runtime system, the OS, or the compiler is able to perform the optimizations without programmer intervention. Unfortunately, however, data locality optimizations cannot be automated in many cases because they require high-level semantic information about the program.

Previous work has shown that the memory system performance (and thus the execution time) of many multithreaded programs can be improved on the source-code level by using compiler directives available in today's commercially available state-of-the-art compilers [91], by making simple algorithmic changes to the program [112], or by using a combination of the two [55]. Although all previously mentioned papers use recent multicore machines for evaluation, none of them considers all aspects related to cross-processor accesses on NUMAs: [91] and [112] do not consider NUMA at all; [55] does not take cache performance into account and in many cases it recommends disabling NUMA (i.e., using page-level interleaving) for problematic memory regions.

In [78] Nikolopoulos et al. claim that data distribution primitives are not necessary for languages like OpenMP, because dynamic profiling can place pages correctly. We find that even with perfect information available, runtime data migrations have too much overhead to be beneficial for programs with data sharing. Bikshandi et al. [11] describe in-program data migrations in a cluster environment, but their language primitives are too heavyweight in small-scale NUMA machines. Darte et al. [26] present generalized multipartitioning of multi-dimensional arrays, a data distribution similar to the block-exclusive data distribution. Multipartitioning, however, relies on message-passing, while we consider direct accesses to shared memory. Zhang et al. [112] take an approach similar to the one described in the thesis: They show that simple program transformations that introduce non-uniformities into inter-thread data sharing improve performance on multicore architectures with shared caches. Chandra et al. [21] describe language primitives similar to ours, but they do not show how programs must be transformed to eliminate data sharing. McCurdy et al. [72] show that adjusting the initialization phase of programs (so that data access patterns of the initialization phase are similar to that of the computations) gives good performance with the first-touch policy. We find that in programs with multiple phases that have conflicting access patterns, first-touch cannot place pages so that each program phase experiences good data locality.

Several runtime systems (e.g., Microsoft's ConcRT, Lithe [81], and Poli-C [5]), support composable parallel software, but none of these systems is designed to preserve the data locality of NUMA-optimized code. There are several approaches to improve the cache locality of work stealing [24, 38]. However, [24] focuses only on improving cache utilization but not on reducing the number of remote memory accesses and [38] supports data locality optimizations, but balances load individually within the scope of each processor (and not between all processors of a system, as the TBB-NUMA system presented in this thesis does). TBB-NUMA is similar to existing PGAS languages [19, 23, 25] in that it gives the programmer explicit control of the mapping of data and computations. However, while PGAS languages are geared towards large cluster-based systems and consequently require the programmer to design applications with a mindset towards large machines, TBB-NUMA targets smaller machines with a single shared memory domain and it allows (but does not require) the developer to custom-tailor existing shared memory-based programs to NUMA architectures.

TBB-NUMA uses the `concurrent_queue` of standard TBB to implement per-processor mailboxes in the task scheduler. Recent work proposed NUMA-aware queuing and locking techniques [29, 36, 74], and wait-free queues have also been developed [53]. Although the TBB `concurrent_queue` is highly optimized, it is neither NUMA-aware, nor wait-free, thus TBB-NUMA could profit from the previously mentioned techniques by using them to enqueue/dequeue tasks more efficiently. The goal of TBB-NUMA is, however, to optimize the memory system performance of the tasks executed by the work-stealing system and not the queuing itself; therefore, we leave the investigation of using NUMA-aware queues with TBB-NUMA to future work.

8

Conclusions

This thesis presents a performance-oriented model for the memory system of NUMA multicore-multiprocessors, a type of shared-memory computers that appeared recently on the market. The thesis investigates two application classes, multiprogrammed workloads and multithreaded programs, and describes memory system bottlenecks experienced by each application class on NUMA multicore-multiprocessors. Furthermore, the thesis presents techniques to reduce negative impact of these bottlenecks on application performance.

Using experimental analysis we show that the performance of multiprogrammed workloads strongly depends on contention for processor-local resources, a factor that is specific to multicore processors (the building blocks of a NUMA multicore-multiprocessor). The non-uniformity of memory access times (a factor specific to NUMA systems) can have a large effect on application performance as well. Thus, system software must consider both factors to achieve good application performance. The N-MASS process scheduler presented in this thesis accounts for two performance-degrading factors, contention for shared caches and locality of main memory accesses, and improves performance by up to 32% and 7% on average over the default setup in current Linux implementations.

We show (also based on experimental analysis) that the locality of cache and DRAM memory accesses is crucial for the performance of many multithreaded programs, the second application class we consider. Unfortunately, however, for some programs data locality can be hard to achieve with today's standard parallel programming frameworks. The thesis proposes extensions to two well-known parallel programming frameworks, OpenMP and TBB; the extensions allow programmers to convey task and data affinity information to the runtime system. We show that, based on affinity information provided by the programmer, a NUMA-aware runtime can realize good data locality that results in up to 220% performance improvement over a NUMA-unaware runtime running on top of a standard implementation of Linux. Moreover, a NUMA-aware runtime can also preserve data locality when a program is ported to a different machine or is composed with other parallel computations.

In summary, this thesis shows the importance of taking the layout of the memory system into account when developing (system) software. We focus on a particular class of machines, NUMA multicore-multiprocessors, and we describe strategies to enhance the memory system-awareness of both OS process scheduling and parallel programming frameworks. We think that in general there is a need to integrate better memory system support into compilers, runtime systems, and operating systems if the performance potential of systems is to be exploited.

Currently, it seems likely that the number of cores per processor chip will continue to increase in the (near) future; a good example of this trend are manycore systems, e.g., the recent generation of Intel Xeon Phi processors with up to 61 cores per processor. More cores can place more demand on the memory system, thus we expect the memory system of future (manycore) systems to be even more elaborate than that of today's multicore-based systems. Furthermore, we expect that future systems will exhibit non-uniform memory access times as well. It is, therefore, likely that on future architectures the performance penalty of inappropriately using the memory system will be as high (or possibly even higher) than on current systems; as a result, memory system-awareness will be even more important for future (system) software.

Producing memory system-aware software (and architecture-aware software in general) requires a thorough and detailed understanding of the interaction between applications and the underlying hardware. Understanding hardware-software interaction is, however, difficult for two main reasons.

First, in practical systems there are usually many layers (e.g., the runtime system, the memory allocator, the linker, or the OS) between the application and the hardware; the performance of an application can closely depend on actions taken by intermediary layers of a system's software stack. In NUMA multicore-multiprocessors, for example, the OS's decisions on page placement and on process/thread scheduling can influence application performance. A programmer seeking to understand and to improve the application's performance must control both aspects to achieve meaningful measurement readings; furthermore, an OS must provide the programmer ways to control both aspects.

The state of the layers of the system's software stack can also have a significant effect on an application's performance. For example, in recent Linux distributions a seemingly innocuous aspect of the system's state, the size and data distribution of the OS's I/O buffer cache, has a large influence on the data distribution of a program: If the OS's buffer cache occupies a high fraction of a processor's local memory, the OS may satisfy requests for new memory at that processor from the memory of other (remote) processors. The OS's decision to allocate pages at a remote processor can cause remote memory accesses, but, more importantly, it can contradict the page distribution policy (e.g., first-touch page placement) that is assumed to be valid in the system. OS support for monitoring and controlling the distribution of the buffer cache across processors and/or OS-based notifications of violations of the system's page placement policy could help a software developer to better and easier understand an application's interaction with a NUMA-multicore system.

In summary, in current systems it is often difficult (or even impossible) to control and to monitor many performance-relevant aspects of the system's software stack. We hope that future systems will provide better means to diagnose and control the layers of the software stack to allow software developers better understand the software stack's influence onto application performance.

A second aspect of understanding hardware-software interaction is to understand how an application interacts with a given processor's microarchitecture. Understanding the performance characteristics of a machine is, however, not always easy. Processor manufacturer's manuals often do no include all details about the characteristics of a given system; in these cases careful (and sometimes tedious) experimental analysis is needed to explore the characteristics of the system.

Feedback from the performance monitoring units of the processor gives insight into the internals of a processor's architecture and can thus also help to understand hardware performance. Moreover, as information from processor performance monitoring units is typically available at runtime, hardware performance monitoring can be used to understand and optimize application performance on the fly. Unfortunately, however, the performance monitoring units of today's processors have a number of limitations.

First, on today's hardware only a low number of performance monitoring events can be counted simultaneously (recent systems have typically around 10 programmable hardware performance monitoring registers) and often not all types of events can be counted at the same time. As understanding an application's performance often requires looking at several hardware-related aspects at the same time, it is important that a large number of events can be measured simultaneously. On current architectures, multiplexing performance monitoring events allows several (even mutually exclusive) performance monitoring events to be measured together (with the cost of somewhat reduced precision). It would be, however, beneficial, if future systems allowed simultaneous monitoring of a large number of events without imposing restrictions on the events that can be measured together.

Second, both the design and the implementation of the hardware performance monitoring units of today's systems lack stability: Monitoring functionality can vary among different microarchitectures, moreover, performance monitoring unit functionality is often complemented, reduced, or disabled as new processor models (even based on the same microarchitecture) are released. With unstable performance monitoring units software developers must first spend time on understanding the monitoring unit of a system before being able to focus on actual performance-related aspects of a system. Thus, stable performance monitoring units could potentially speed up the development of architecture-aware software.

Finally, the level of abstraction of most hardware events in current systems is close to the microarchitecture. As hardware manufacturer's do not usually describe all details of microarchitectures they release, some performance monitoring events are difficult to understand. Moreover, as software often requires (at least in our experience) a level of abstraction above the abstraction level of the microarchitecture, supporting high-level events in the performance monitoring unit (in addition to low-level ones) might be beneficial in the future.

We hope that this thesis provides supporting evidence to include appropriate performance monitoring, debugging, and profiling support into future architectures to allow (easier) development of architecture-aware software.

Bibliography

[1] AMD HyperTransport technology-based system architecture. 2002.

[2] Using Intel Vtune performance analyzer to optimize software on Intel Core i7 Processors, 2010.

[3] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, SPAA '00, pages 1–12, New York, NY, USA, 2000. ACM.

[4] Milena Milenkovic Aleksandar, Ar Milenkovic, and Jeffrey Kulick. Demystifying Intel branch predictors. In *Proceedings of the annual workshop on duplicating, deconstructing, and debunking*, WDDD '02, pages 52–61. John Wiley & Sons, 2002.

[5] Zachary Anderson. Efficiently combining parallel software using fine-grained, language-level, hierarchical resource management policies. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 717–736, New York, NY, USA, 2012. ACM.

[6] Manu Awasthi, David W. Nellans, Kshitij Sudan, Rajeev Balasubramonian, and Al Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 319–330, New York, NY, USA, 2010. ACM.

[7] Reza Azimi, Michael Stumm, and Robert W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 101–110, New York, NY, USA, 2005. ACM.

[8] Mohammad Banikazemi, Dan Poff, and Bulent Abali. PAM: A novel performance/power aware meta-scheduler for multi-core systems. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 39:1–39:12, Piscataway, NJ, USA, 2008. IEEE Press.

[9] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[10] 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.

[11] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, María J. Garzarán, David Padua, and Christoph von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 48–57, New York, NY, USA, 2006. ACM.

[12] John Bircsak, Peter Craig, RaeLyn Crowell, Zarka Cvetanovic, Jonathan Harris, C. Alexander Nelson, and Carl D. Offner. Extending OpenMP for NUMA machines. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.

[13] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A case for NUMA-aware contention management on multicore processors. In *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC '11, Berkeley, CA, USA, 2011. USENIX Association.

[14] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28:8:1–8:45, December 2010.

[15] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A case for NUMA-aware contention management on multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 557–558, New York, NY, USA, 2010. ACM.

[16] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. 2008.

[17] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for NUMA memory management. In *Proceedings of the twelfth ACM symposium on Operating systems principles*, SOSP '89, pages 19–31, New York, NY, USA, 1989. ACM.

[18] James R. Bulpin and Ian A. Pratt. Multiprogramming performance of the Pentium 4 with hyper-threading. In *Proceedings of the annual workshop on Duplicating, deconstructing and debunking*, WDDD '04, pages 53–62, June 2004.

[19] Bradford L. Chamberlain. A brief overview of Chapel (revision 1.0). 2013.

[20] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th international symposium on High-performance computer architecture*, HPCA '05, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[21] Rohit Chandra, Ding-Kai Chen, Robert Cox, Dror E. Maydan, Nenad Nedeljkovic, and Jennifer M. Anderson. Data distribution support on distributed shared memory multiprocessors. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, pages 334–345, New York, NY, USA, 1997. ACM.

[22] James Charles, Preet Jassi, Narayan S. Ananth, Abbas Sadat, and Alexandra Fedorova. Evaluation of the Intel Core i7 Turbo Boost feature. In *Proceedings of the 2009 IEEE*

international symposium on Workload characterization, IISWC '09, pages 188–197, Washington, DC, USA, 2009. IEEE Computer Society.

[23] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[24] Quan Chen, Minyi Guo, and Zhiyi Huang. CATS: cache aware task-stealing based on online profiling in multi-socket multi-core architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 163–172, New York, NY, USA, 2012. ACM.

[25] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: Co-Array Fortran and Unified Parallel C. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 36–47, New York, NY, USA, 2005. ACM.

[26] Alain Darte, John Mellor-Crummey, Robert Fowler, and Daniel Chavarría-Miranda. Generalized multipartitioning of multi-dimensional arrays for parallelizing line-sweep computations. *J. Parallel Distrib. Comput.*, 63(9):887–911, September 2003.

[27] 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.

[28] Tanima Dey, Wei Wang, Jack W. Davidson, and Mary Lou Soffa. Resense: Mapping dynamic workloads of colocated multithreaded applications using resource sensitivity. *ACM Trans. Archit. Code Optim.*, 10(4):41:1–41:25, December 2013.

[29] Dave Dice, Virendra J. Marathe, and Nir Shavit. Flat-combining NUMA locks. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 65–74, New York, NY, USA, 2011. ACM.

[30] Stéphane Eranian. What can performance counters do for memory subsystem analysis? In *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness*, MSPC '08, pages 26–30, New York, NY, USA, 2008. ACM.

[31] Stijn Eyerman and Lieven Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, 2008.

[32] Stijn Eyerman and Lieven Eeckhout. Probabilistic job symbiosis modeling for SMT processor scheduling. In *Proceedings of the fifteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XV, pages 91–102, New York, NY, USA, 2010. ACM.

[33] Alexandra Fedorova, Margo Seltzer, Christoper Small, and Daniel Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *Proceedings of the USENIX Annual Technical Conference*, ATEC '05, pages 26–26, Berkeley, CA, USA, 2005. USENIX Association.

[34] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 25–38, Washington, DC, USA, 2007. IEEE Computer Society.

[35] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[36] Elad Gidron, Idit Keidar, Dmitri Perelman, and Yonathan Perez. SALSA: scalable and low synchronization NUMA-aware algorithm for producer-consumer pools. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, SPAA '12, pages 151–160, New York, NY, USA, 2012. ACM.

[37] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 151–162, New York, NY, USA, 2006. ACM.

[38] Yi Guo, Jisheng Zhao, Vincent Cav, and Vivek Sarkar. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In *Proceedings of the 24th IEEE international symposium on Parallel and distributed processing*, IPDPS '10, pages 1–12. IEEE, 2010.

[39] Daniel Hackenberg, Daniel Molka, and Wolfgang E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore smp systems. In *Proceedings of the 42nd Annual IEEE/ACM international symposium on Microarchitecture*, MICRO 42, pages 413–422, New York, NY, USA, 2009. ACM.

[40] Andrew Herdrich, Ramesh Illikkal, Ravi Iyer, Don Newell, Vineet Chadha, and Jaideep Moses. Rate-based QoS techniques for cache/memory in CMP platforms. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 479–488, New York, NY, USA, 2009. ACM.

[41] Amir H. Hormati, Yoonseo Choi, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Proceedings of the 2009 18th international conference on Parallel architectures and compilation techniques*, PACT '09, pages 214–223, Washington, DC, USA, 2009. IEEE Computer Society.

[42] Intel Corporation. *Intel Xeon Processor 7500 Series Uncore Programming Guide*, March 2010.

[43] Intel Corporation. *Intel(R) Threading Building Blocks Reference Manual*, 2012.

[44] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, April 2012.

[45] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual (Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, and 3C*, September 2013.

[46] Marty Itzkowitz, Oleg Mazurov, Nawal Copty nad, and Yuan Lin. An OpenMP runtime API for profiling. Technical report, Sun Microsystems, Inc., 2007.

[47] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 220–229, New York, NY, USA, 2008. ACM.

[48] H. Jin, H. Jin, M. Frumkin, M. Frumkin, J. Yan, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical report, NASA Ames Research Center, Moffett Field, CA, 1999.

[49] Yao Jin. numatop: A tool for memory access locality characterization and analysis. Intel Open Source Technology Center, 2013.

[50] Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. Inter-core prefetching for multicore processors using migrating helper threads. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, pages 393–404, New York, NY, USA, 2011. ACM.

[51] Mahmut Kandemir, Taylan Yemliha, SaiPrashanth Muralidhara, Shekhar Srikantaiah, Mary Jane Irwin, and Yuanrui Zhang. Cache topology aware computation mapping for multicores. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 74–85, New York, NY, USA, 2010. ACM.

[52] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, May 2008.

[53] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 223–234, New York, NY, USA, 2011. ACM.

[54] David Koufaty, Dheeraj Reddy, and Scott Hahn. Bias scheduling in heterogeneous multicore architectures. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 125–138, New York, NY, USA, 2010. ACM.

[55] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. Memprof: A memory profiler for numa multicore systems. In *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC'12, pages 5–5, Berkeley, CA, USA, 2012. USENIX Association.

[56] Christoph Lameter. Local and remote remory: Memory in a Linux/NUMA system. SGI, 2006.

[57] David Levinthal. *Performance analysis guide for the Intel Core i7 Processor and the Intel Xeon 5500 processors*. Intel Corporation, 2009.

[58] Hui Li, Hui Li Sudarsan, Michael Stumm, and Kenneth C. Sevcik. Locality and loop scheduling on NUMA multiprocessors. In *Proceedings of the 1993 international conference on Parallel processing*, pages 140–147. CRC Press, Inc, 1993.

[59] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 1–11, New York, NY, USA, 2007. ACM.

[60] Wei Li and Keshav Pingali. Access normalization: Loop restructuring for NUMA compilers. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 285–295, New York, NY, USA, 1992. ACM.

[61] Xu Liu and John Mellor-Crummey. Pinpointing data locality problems using data-centric analysis. In *Proceedings of the 9th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '11, pages 171–180, Washington, DC, USA, 2011. IEEE Computer Society.

[62] 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.

[63] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 950–961. VLDB Endowment, 2007.

[64] Robert A. Maddox, Gurbir Singh, and Robert J. Safranek. A first look at the Intel QuickPath Interconnect. 2009.

[65] Zoltan Majo and Thomas R. Gross. Memory management in NUMA multicore systems: Trapped between cache contention and interconnect overhead. In *Proceedings of the international symposium on Memory management*, ISMM '11, pages 11–20, New York, NY, USA, 2011. ACM.

[66] Anirban Mandal, Rob Fowler, and Allan Porterfield. Modeling memory concurrency for multi-socket multi-core systems. In *Proceedings of the 2010 IEEE international symposium on Performance analysis of systems and software*, ISPASS '10, pages 66–75, March 2010.

[67] Jaydeep Marathe and Frank Mueller. Hardware profile-guided automatic page placement for ccNUMA systems. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 90–99, New York, NY, USA, 2006. ACM.

[68] Jaydeep Marathe, Vivek Thakkar, and Frank Mueller. Feedback-directed page placement for ccNUMA via hardware-generated memory traces. *J. Parallel Distrib. Comput.*, 70:1204–1219, December 2010.

[69] Jason Mars, Lingjia Tang, and Mary Lou Soffa. Directly characterizing cross core interference through contention synthesis. In *Proceedings of the 6th international conference on High performance and embedded architectures and compilers*, HiPEAC '11, pages 167–176, New York, NY, USA, 2011. ACM.

[70] Jason Mars, Neil Vachharajani, Mary Lou Soffa, and Robert Hundt. Contention aware execution: Online contention detection and response. In *Proceedings of the 2010 international symposium on Code generation and optimization*, CGO '10, pages 257–265, New York, NY, USA, 2010. ACM.

[71] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[72] Collin McCurdy and Jeffrey S. Vetter. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *Proceedings of the IEEE international symposium on Performance analysis of systems and software*, ISPASS '10, pages 87–96, 2010.

[73] Daniel Molka, Daniel Hackenberg, Robert Schne, and Matthias S. Müller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *Proceedings of the 2009 18th international conference on Parallel architectures and compilation techniques*, PACT '09, pages 261–270, Washington, DC, USA, 2009. IEEE Computer Society.

[74] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, pages 103–112, New York, NY, USA, 2013. ACM.

[75] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 265–276, New York, NY, USA, 2009. ACM.

[76] Angeles Navarro, Rafael Asenjo, Siham Tabik, and Calin Cascaval. Analytical modeling of pipeline parallelism. In *Proceedings of the 2009 18th international conference on Parallel architectures and compilation techniques*, PACT '09, pages 281–290, Washington, DC, USA, 2009. IEEE Computer Society.

[77] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. A case for user-level dynamic page migration. In *Proceedings of the 2000 International Conference on Parallel Processing*, ICPP '00, pages 95–, Washington, DC, USA, 2000. IEEE Computer Society.

[78] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesus Labarta, and Eduard Ayguade. Is data distribution necessary in OpenMP? In

Proceedings of the 2000 ACM/IEEE conference on Supercomputing, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[79] Takeshi Ogasawara. NUMA-aware memory manager with dominant-thread-based copying GC. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 377–390, New York, NY, USA, 2009. ACM.

[80] OpenMP Architecture Review Board. *OpenMP Application Programming Interface, Version 3.1*, July 2011.

[81] Heidi Pan, Benjamin Hindman, and Krste Asanović. Composing parallel software efficiently with lithe. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 376–387, New York, NY, USA, 2010. ACM.

[82] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 335–348, New York, NY, USA, 2010. ACM.

[83] Kishore Kumar Pusukuri, David Vengerov, Alexandra Fedorova, and Vana Kalogeraki. FACT: A framework for adaptive contention-aware thread migrations. In *Proceedings of the 8th ACM international conference on Computing frontiers*, CF '11, pages 35:1–35:10, New York, NY, USA, 2011. ACM.

[84] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM international symposium on Microarchitecture*, MICRO 39, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[85] Chandan Reddy and Uday Bondhugula. Effective automatic computation placement and dataallocation for parallelization of regular programs. In *Proceedings of the 28th ACM international conference on Supercomputing*, ICS '14, pages 13–22, New York, NY, USA, 2014. ACM.

[86] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in TBB. In *Proceedings of the IEEE international symposium on Parallel and distributed processing*, IPDPS'08, pages 1 –8, April 2008.

[87] M. Roth, M.J. Best, C. Mustard, and A. Fedorova. Deconstructing the overhead in parallel applications. In *Proceedings of the IEEE international symposium on Workload characterization*, IISWC'12, pages 59 –68, nov. 2012.

[88] Juan Carlos Saez, Manuel Prieto, Alexandra Fedorova, and Sergey Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 139–152, New York, NY, USA, 2010. ACM.

[89] Daniel Sanchez, David Lo, Richard M. Yoo, Jeremy Sugerman, and Christos Kozyrakis. Dynamic fine-grain scheduling of pipeline parallelism. In *Proceedings of the 2011 International conference on Parallel architectures and compilation techniques*, PACT '11, pages 22–32, Washington, DC, USA, 2011. IEEE Computer Society.

[90] Andreas Sandberg, David Eklöv, and Erik Hagersten. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In *Proceedings of the 2010 ACM/IEEE international conference for High performance computing, networking, storage and analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[91] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can traditional programming bridge the Ninja performance gap for parallel computing applications? In *Proceedings of the 39th annual international symposium on Computer architecture*, ISCA'12, pages 440 –451, June 2012.

[92] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, ASPLOS IX, pages 234–244, New York, NY, USA, 2000. ACM.

[93] Shekhar Srikantaiah, Mahmut Kandemir, and Mary Jane Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 135–144, New York, NY, USA, 2008. ACM.

[94] ChunYi Su, Dong Li, Dimitrios S. Nikolopoulos, Matthew Grove, Kirk Cameron, and Bronis R. de Supinski. Critical path-based thread placement for NUMA systems. *SIGMETRICS Perform. Eval. Rev.*, 40(2):106–112, October 2012.

[95] M. Aater Suleman, Moinuddin K. Qureshi, Khubaib, and Yale N. Patt. Feedback-directed pipeline parallelism. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 147–156, New York, NY, USA, 2010. ACM.

[96] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European conference on Computer systems*, EuroSys '07, pages 47–58, New York, NY, USA, 2007. ACM.

[97] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 121–132, New York, NY, USA, 2009. ACM.

[98] S. Tandri and T.S. Abdelrahman. Automatic partitioning of data and computations on scalable shared memory multiprocessors. In *Proceedings of the international conference on Parallel processing*, pages 64 –73, August 1997.

[99] Lingjia Tang, Jason Mars, and Mary Lou Soffa. Contentiousness vs. sensitivity: Improving contention aware runtime systems on multicore architectures. In *Proceedings of the 1st international workshop on Adaptive self-tuning computing systems for the exaflop era*, EXADAPT '11, pages 12–21, New York, NY, USA, 2011. ACM.

[100] Lingjia Tang, Jason Mars, Xiao Zhang, Robert Hagmann, Robert Hundt, and Eric Tune. Optimizing Google's warehouse scale computers: The NUMA experience. In *Proceedings of the 2013 IEEE 19th international symposium on High performance computer architecture*, HPCA '13, pages 188–197, Washington, DC, USA, 2013. IEEE Computer Society.

[101] R. Thekkath and S. J. Eggers. Impact of sharing-based thread placement on multithreaded architectures. In *Proceedings of the 21st annual international symposium on Computer architecture*, ISCA '94, pages 176–186, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[102] Mustafa M. Tikir and Jeffery K. Hollingsworth. NUMA-aware Java heaps for server applications. In *Proceedings of the 19th IEEE international symposium on Parallel and distributed processing*, IPDPS '05, Washington, DC, USA, 2005. IEEE Computer Society.

[103] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Using hardware counters to automatically improve memory performance. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, page 46, Washington, DC, USA, 2004. IEEE Computer Society.

[104] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Hardware monitors for dynamic page migration. *J. Parallel Distrib. Comput.*, 68(9):1186–1200, 2008.

[105] Irina Tuduce, Zoltan Majo, Adrian Gauch, Brad Chen, and Thomas R. Gross. Asymmetries in multi-core systems – or why we need better performance measurement units. The Exascale Evaluation and Research Techniques Workshop (EXERT) at ASPLOS 2010, 2010.

[106] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ISCA '98, pages 533–544, New York, NY, USA, 1998. ACM.

[107] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ASPLOS '96, pages 279–289, New York, NY, USA, 1996. ACM.

[108] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

[109] Kenneth M. Wilson and Bob B. Aglietti. Dynamic page placement to improve locality in CC-NUMA multiprocessors for TPC-C. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, Supercomputing '01, pages 33–33, New York, NY, USA, 2001. ACM.

[110] Carole-Jean Wu and Margaret Martonosi. Characterization and dynamic mitigation of intra-application cache interference. In *Proceedings of the IEEE international symposium*

on Performance analysis of systems and software, ISPASS '11, pages 2–11, Washington, DC, USA, 2011. IEEE Computer Society.

[111] R. Yang, J. Antony, P. P. Janes, and A. P. Rendell. Memory and thread placement effects as a function of cache usage: A study of the gaussian chemistry code on the SunFire X4600 M2. In *Proceedings of the the international symposium on Parallel architectures, algorithms, and networks*, ISPAN '08, pages 31–36, Washington, DC, USA, 2008. IEEE Computer Society.

[112] Eddy Z. Zhang, Yunlian Jiang, and Xipeng Shen. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 203–212, New York, NY, USA, 2010. ACM.

[113] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the fifteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XV, pages 129–142, New York, NY, USA, 2010. ACM.

List of Figures

Curriculum Vitae

Zoltán Majó

June 18, 1983	Born in Cluj-Napoca/Klausenburg/Kolozsvár/Kloiznburg, Romania
1998–2002	“Apáczai Csere János” High School, Cluj-Napoca, Romania
2002	High school exit exam (maturity exam)
2002–2007	Studies in Computer Science, Technical University of Cluj-Napoca, Romania
2007	Diploma of Engineer (“Inginer diplomat”) in Computer Science, Technical University of Cluj-Napoca, Romania
2007–2008	Teaching Assistant and System Engineer, Technical University of Cluj-Napoca, Romania
since 2008	Research and Teaching Assistant Laboratory for Software Technology, ETH Zurich, Switzerland