ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Δρομολόγηση Παράλληλων Εφαρμογών σε Πολυπύρηνα Συστήματα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Χαράλαμπος Κ. Χαλιός

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Αθήνα, Ιούνιος 2013

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Δρομολόγηση Παράλληλων Εφαρμογών σε Πολυπύρηνα Συστήματα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Χαράλαμπος Κ. Χαλιός

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 17η Ιουνίου 2013.

....	....	....
Νεκτάριος Κοζύρης	Νικόλαος Παπασπύρου	Δημήτριος Σούντρης
Καθηγητής ΕΜΠ	Αναπληρωτής Καθηγητής ΕΜΠ	Επίκουρος Καθηγητής ΕΜΠ

Αθήνα, Ιούνιος 2013.

....
(Χαράλαμπος Κ. Χαλιός)
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright© 2013 Εθνικό Μετσόβιο Πολυτεχνείο.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ' ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τη συγγραφέα.
Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τη συγγραφέα και δεν πρέπει να ερμηνευτεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Η στροφή σε πολυπύρηνα συστήματα, τα οποία παρέχουν αφθονία υπολογιστικών πόρων, έχουν οδηγήσει τους προγραμματιστές στην υιοθέτηση μοντέλων παράλληλου προγραμματισμού έτσι, ώστε να μπορέσουν αναπτύξουν αποδοτικές εφαρμογές. Ο παράλληλος προγραμματισμός υπόσχεται επίδοση που κλιμακώνει με την αύξηση του πλήθους των υπολογιστικών πόρων. Παρ' όλα αυτά, περιορισμοί που έχουν να κάνουν, κυρίως, με την ιεραρχία μνήμης δεν επιτρέπουν στις παράλληλες εφαρμογές να επιτύχουν την αναμενόμενη επίδοση. Σαν αποτέλεσμα οι εφαρμογές αυτές δεν είναι ικανές να αξιοποιήσουν τους διαθέσιμους πόρους. Για να μη μείνουν ανεκμετάλλευτοι αυτοί οι πόροι, υπάρχει η ανάγκη για ταυτόχρονη εκτέλεση για παραπάνω απο μία παράλληλες εφαρμογές. Σ' αυτή την περίπτωση όμως, ο ανταγωνισμός για πόρους απο διαφορετικές εφαρμογές οδηγεί σε απρόβλεπτη συμπεριφορά και επιπλέον μείωση τις επίδοσής τους. Υπάρχει, λοιπόν, η ανάγκη για αποδοτική δρομολόγηση των εφαρμογών και διανομή των πόρων του συστήματος μνήμης. Στη διπλωματική αυτή, μελετάμε τους λόγους για τους οποίους οι υπάρχοντες schedulers των λειτουργικών συστημάτων δε βοηθάνε στην εκτέλεση παράλληλων εφαρμογών στα τρέχοντα υπολογιστικά συστήματα. Εξετάζουμε τους περιορισμούς του συστήματος μνήμης που οδηγεί σε κακή επίδοση των παράλληλων εφαρμογών. Στη συνέχεια μελετάμε κάποιες απ' τις μεθόδους που έχουν προταθεί στη βιβλιογραφία για αντιμετώπιση των προβλημάτων των state-of-the-art schedulers σχετικά με τη δρομολόγηση παράλληλων εφαρμογών, καθώς και υλοποιούμε δικές μας τεχνικές δρομολόγησης. Τέλος, εξετάζουμε το θέμα της τοποθέτησης των νημάτων των πολυνηματικών εφαρμογών στους πυρήνες ενός πολυπύρηνου συστήματος. Συνοψίζοντας, τονίζουμε την αδυναμία των παράλληλων εφαρμογών να κλιμακώσουν ικανοποιητικά στα σημερινά πολυεπεξεργαστικά συστήματα και ως συνέπεια τη σημασία ανάπτυξης τεχνικών δρομολόγησης παράλληλων εφαρμογών που λαμβάνουν υπ' όψιν τους την οργάνωση τους συστήματος μνήμης της υπολογιστικής πλατφόρμας και εφαρμόζουν τεχνικές που περιορίζουν τις επιπτώσεις του ανταγωνισμού των παράλληλων εφαρμογών για μοιραζόμενους πόρους του συστήματος μνήμης. Επισημαίνουμε τη σημασία της τοποθέτησης των νημάτων μια πολυνηματικής εφαρμογής στους επεξεργαστές ενός πολυπύρηνου συστήματος και εξετάζουμε το πως διαφορετικές αποφάσεις, μπορούν να προσδώσουν διαφορετικά χαρακτηριστικά εκτέλεσης.

Λέξεις-κλειδιά: πολυεπεξεραστικά συστήματα, πολυνηματικές εφαρμογές, CMPs, gang scheduling, contention-aware scheduling, OpenMP, memory bus bandwidth, τοποθέτηση νημάτων

Abstract

The proliferation of Chip Multiprocessors (CMPs) has caused a noticeable stir in the current programming paradigm. Developers turn to parallel programming in order to take advantage of the multiple processing units found in todays computing systems. Multithreaded applications distribute the computational load in many threads that use shared memory to communicate. The shift, however, from Symmetric Multiprocessors (SMPs) to CMPs has brought up challenges concerning contention in resources that are shared among cores of the CMP which do not allow applications to achieve the expected performance.

Computing systems, from their side have been developed to adapt to the new platforms. With the advent of SMPs task schedulers have been extended to load balance the system workload to the multiple identical cpus in order to exploit parallelism. Schedulers are now obliged to use space-sharing in order to achieve maximum utilization of the system resources, since applications are incapable to exploit them on their own. The hierarchical organization of the CMPs, however, with cores sharing resources such as cache memory, memory bus, DRAM controllers etc, require much more elaborate scheduler implementations.

In this thesis, we explore the factors that lead to poor performance in cases where multiple multithreaded applications are executed on a CMP. Knowing that memory constraints are the primary reason for which multithreaded applications fail to scale with many cpus, we can expect that this phenomenon will be more intense when those applications will contend for resources of the memory subsystem. First, we investigate what effect has the limited memory bandwidth in the execution of multithreaded applications and most importantly when they are co-scheduled with other applications and extract information that could be used from a scheduler in order to make more efficient decisions. Second we study how decisions on thread affinity of multithreaded applications can affect their execution profile. The hierarchical organization of CMPs has made thread affinity an important aspect. Whereas in SMPs placement of threads is not important, since the only resource they share is the memory bus, in CMPs where cores share multiple resources, affinity of threads determine the amount of resources applications contend for. Finally we compare some scheduler implementations found in previous work, as well as, a scheduler that we implemented and the Linux scheduler.

Keywords: CMPs, multithreaded applications, CMPs, gang scheduling, contention-aware scheduling, OpenMP, memory bus bandwidth, thread placement2

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου, υπό την επίβλεψη του Καθηγητή Νεκτάριου Κοζύρη.

Καταρχάς, θα ήθελα να ευχαριστήσω τον καθηγητή μου κ. Νεκτάριο Κοζύρη, τόσο για την εποπτεία του κατά την εκπόνηση της εργασίας μου, όσο και για τις γνώσεις και την έμπνευση που μου προσέφερε με τη διδασκαλία του, καθ' όλη τη διάρκεια της φοίτησής μου.

Θα ήθελα, ιδιαιτέρως, να ευχαριστήσω το Μεταδιδακτορικό Ερευνητή Νικόλαο Αναστόπουλο για τη συνεχή καθοδήγηση και υπομονή του, η βοήθεια του οποίου για την ολοκλήρωση αυτής της διπλωματικής ήταν ανεκτίμητη.

Θα ήθελα, επίσης, να ευχαριστήσω όλους τους φίλους και συμφοιτητές μου, που με ανυπομονησία περιμέναν την ολοκλήρωση της διπλωματικής αυτής.

Τέλος, οι θερμότερες ευχαριστίες απευθύνονται στους γονείς μου, για την αγάπη τους, την αμέριστη υποστήριξή τους και την εμπιστοσύνη τους σε κάθε μου επιλογή, από τα πρώτα χρόνια της ζωής μου μέχρι σήμερα, αδιάκοπα και ακούραστα.

Μπάμπης Χαλιός

Contents

Chapter 1

Introduction

1.1 Problem definition

Until recently computing performance was provided “for free”. Architects were able to increase cpu frequencies every few months, improvements in microarchitecture and design of complex memory hierarchies provided great boost in serial execution performance. Those changes were transparent to developers who didn’t have to redesign software. The “free lunch” however, is over. Limitations related to heat dissipation, increased power consumption, the limited instruction level parallelism (ILP) that exists in software and the increasing gap between processor and memory speed caused a stir in computer architecture design from uni-processor systems with a single fat core to multicores with multiple, usually thinner, cores integrated in a single chip.

As a result, developers cannot expect, any more, for architectural advancements to improve performance of their software. They are forced to abandon serial programming paradigms and use parallel programming models that will leverage chip multi-processors (CMPs). CMPs offer tremendous opportunities for multithreaded applications which can take advantage of simultaneous thread execution and fast inter-thread communication that leads to scaling performance.

However, the cores of a CMP are not totally independent from each other. Usually, they share many components of the memory sub-system, such as last level caches (LLC), memory controllers, prefetchers, the memory bus and the underlying interconnectıon system. This sharing of resources limits the potential advantages of CMP’s. Concurrent execution of multiple applications result in contention in resources that are shared among cpu’s of the CMP. Contention leads to under-utilization of system resources and consecutively in performance degradation. It is obvious that in order to take full advantage of the potential provided by CMP’s we need to deal with their negative aspects and take advantage of the positive.

A significant class of solutions that address problems of resource contention in CMPs focuses in scheduling OS level threads. Those solutions aim to schedule threads in a resource aware manner in order to take advantage of the benefits that multithreaded applications can gain when executing in multicores and mitigate the performance degradation caused by contention in the memory subsystem. These solutions are attractive since they do not need changes to hardware and minimal changes to the OS.

1.2 Scheduling - Role of the operating system

Since the dawn of multiprogramming systems the scheduler has been an integral part of the operating systems. The scheduler, traditionally, is responsible for making decisions about the allocation of system resources to multiple applications. Its main challenge is to strive to provide efficiency, as well as fairness for every application. Depending on the nature of the platform the scheduler is designed to optimize a different objective. In desktops, for example, the challenge is application responsiveness, while in high performance computing systems schedulers try to maximize overall throughput.

Until several years ago, where uni-processor systems where the norm across the whole computing spectrum, OS scheduling was performed in terms of time-multiplexing applications. Scheduling algorithms had attracted the attention of the academic community as well as the industry. Until the advent and subsequent near ubiquitous proliferation of CMPs, schedulers had become so optimized that the need for further improvements dramatically subsided. A typical example is the latest Linux task schedulers which were considered highly efficient. O(1) scheduler used run-queues consisting of priority lists for different priority processes and implemented interactivity heuristics to decide the process with the highest priority. The scheduler kept one run-queue for every cpu of the CMP and used a load balancing algorithm to fairly distribute the load among the available cpus. Completely Fair Scheduler (CFS) [16] replaced the $O(1)$ [17] scheduler as the scheduler of Linux. Instead of keeping priority lists in every run-queue it uses a red-black tree to describe the "need" for cpu-time of every process in this runqueue. It considers the process with the highest waiting time as the next process to be chosen to execute for the next time-quantum.

When CMPs arrived schedulers were optimized for symmetric multiprocessor (SMP) systems. SMP is a multiprocessor architecture that consists of multiple identical processors that connect to a shared memory. Each processor of an SMP is independent from the others since they only share the interconnection network to the memory. Using this scheduler unmodified for CMP systems gives the OS the illusion that cpus of the CMP are isolated. This was a convenient but overly optimistic simplification that caused a great deal of problems.

CMP architectures consist of multiple cpu cores on a single chip, usually sharing the upper level of cache memory (e.g. L2, L3) in an hierarchical manner. The multiple processing units found in CMPs insert one additional level of complexity in scheduling decisions. In addition to multiplexing a single core in time, time-sharing it among threads, it becomes necessary to space-share the CMP deciding on which processor a thread will execute. The rapidly increasing number of cores provided by state-of-the-art platforms makes the need of space-sharing even more intense, since multithreaded applications are inherently incapable to presume upon this abundance of hardware resources. This is a responsibility of the OS scheduler and while on older systems space-sharing was not important, with the advent of CMP's became a very challenging issue.

Contention in resources shared by multiple cpus of a CMP is a limiting factor in the benefits that can be derived from the parallelism offered from multiprocessors. Depending on the mix of applications that execute concurrently on the cores of a multiprocessor the level of contention can vary greatly. Choosing applications that make heavy use of the memory subsystem to run concurrently leads to poor performance. Other applications blend together nicely and co-executing them is beneficial in terms of throughput. The number of threads with which an application executes is another important factor that causes the variable level of contention. Not only concurrent execution of parallel applications does not always performs as expected but to make things worse, the level of performance degradation is highly dependant from a number of factors that is difficult to predict. Unpredictable performance is evenly undesirable as reduced throughput. For example systems that provide Quality of Service (QoS) cannot tolerate unpredictable performance.

At the same time, research has shown that executing threads of the same application concurrently leads to high throughput. Frequently, multithreaded applications share data among threads, thus executing threads on cores of the CMP at the same time slot favors this sharing and at the same time some threads can make use of data prefetched by other threads of the same application.

However, modern OS's lack the ability to handle threads efficiently. OS treats threads uniformly, that is, it does not distinguish threads of different applications. As a result, it cannot enforce policies that have threads of the same application execute concurrently and does not exploit the benefits of such a scheduling scheme. Moreover the operating system does not have any kind of mechanisms to assist it to keep track of the system performance. However, that kind of mechanisms are necessary in order for the OS to implement a scheduling scheme that will be resource aware and endeavor to make schedules that will diminish the contention in the memory subsystem and avoid performance bottlenecks.

1.3 Contribution

In order to deal with the problem of scheduling multithreaded applications in CMPs there are some issues that must be addressed. First, any decision that a scheduler makes must take into consideration the underlying hardware platform. It is necessary that detailed information about the number and the organization of the available cpus, on one hand, and the way the memory subsystem is organized, on the other, is communicated to the scheduler. Second, the scheduler requires information about the execution profile of the applications that it will handle. Such information is the ability of an application to exploit the available system resources such as cpus and bandwidth of the memory bus, as exposed by characteristics such as the number of cores until which the application scales efficiently, its Last Level Cache (LLC) miss rates, memory bandwidth consumption, etc.

Apart from the knowledge concerning the system and the application characteristics that is needed for the scheduler to make efficient decisions, it is important that we investigate the effects of execution of multithreaded applications in CMPs. We must understand what are the consequences of applications sharing resources such as levels of the cache hierarchy, the memory bus or the DRAM controller, what are the characteristics of the applications that can be efficiently co-scheduled and which applications are destructive to run at the same time-quantum. We must also quantify the contribution in performance degradation of every shared resource, as well as the significance of more technical issues such as the thread affinity. This knowledge must be incorporated in the scheduling algorithm in order to be able to leverage the system resources.

The rest of the thesis deals with some of the aforementioned issues. We show that multithreaded applications lack the ability to exploit all the available system resources, since they fail to scale for large thread-counts. We demonstrate that memory constraints is the major factor for their poor scale and that contention in resources of the memory subsystem deteriorates the situation. We explore what effect has contention for the memory bus in scheduling, as well as that thread affinity is an important issue and that different decisions on that matter provide different execution characteristics. We also compare different scheduler implementations and show that implementing the previous methods significantly improve throughput comparing to the Linux scheduler.

Chapter 2

Motivation

In the last years computer architects have made a noticeable stir from uni-processor architectures to chip multi-processors (CMP). State-of-the-art computer architectures focus on integrating multiple processing units on a single chip rather than optimizing existing uni-processor systems. Until the middle of the previous decade, the exponential performance improvement of uni-processor systems was provided by advancements on the micro-architecture level. Designers were able to increase transistor density and the clock speeds, which alongside with micro-architectural optimizations provided great boost in sequential performance. However, this came with increased heat dissipation and power consumption, thus making the approach of frequency increase less tempting. Moreover, the limited instruction level parallelism (ILP) found in many applications made it difficult to exploit those improvements. For all these reasons, designers changed their strategy towards the integration of multiple cores within a chip as a means to extract more performance. And while uni-processor optimizations offered "free luch" to the programmers, requiring none or few code changes, CMP architectures require radical shifts from traditional programming approaches to novel parallel programming models. This stir in architecture design created the need for the OSs to readjust. Schedulers had to be upgraded significantly in order to utilize the multiple processors and load balance the system workload among them. Moreover, since CMPs became the common paradigm scheduler has to deal with new challenges related to contention for shared resources. This attracted the interest of researchers who strive to employ new techniques for scheduling in order to fully exploit the potential of CMPs.

2.1 Perspective and challenges

CMPs offer tremendous opportunities for high throughput in parallel applications. Parallel computing promises scalable performance when executing an application in multiple cores. At the same time, the large number of processing units allows more than one applications to execute at the same time, reducing the off-cpu time.

Research has shown that scheduling threads of the same application concurrently is more beneficial compared to scheduling them independent from each other. There are many reasons for this. First, threads often need to synchronize their execution using barriers or execute a critical section protected with locks. Synchronization might take a considerable amount of time if threads are not scheduled concurrently, e.g. when some of them wait to acquire a lock that is held by a descheduled thread. Second, some portion of data in multithreaded applications is usually shared among their threads, thus their concurrent execution potentially enables its reuse. For example, in modern CMPs where cpus share some of the levels of memory hierarchy, the data accesses of one thread might seem useful for other threads that share it, if they execute under a common cache.

Multithreaded applications very often cannot scale efficiently at large number of cores and therefore cannot take advantage of the abundance of resources in current CMPs. In order to fully utilize the unexploited resources we need to co-schedule more than one applications at the same time. This is useful to increase both application throughput and responsiveness. However, cores in a CMP are not completely independent from each other and they are designed to share multiple on- and off-chip resources. The most frequently shared components are modules of the memory hierarchy such as the memory bus and interconnects, DRAM controllers and last level caches (L2 or L3). The concurrent execution of threads leads to contention for those resources, limiting the expected performance benefits. These shared resources are managed entirely by hardware and are allocated thread-unawarely. Requests to those controllers are handled as if they were from a single source and therefore there is not any fairness or partitioning enforced when different threads make access to those resources. Moreover, modern memory designs focus on Non-Uniform Memory Access (NUMA) organizations which divide the total memory amount into memory nodes distributed to the physical packages of the CMP. The access time of memory references made from a thread executing on a cpu depends on which memory node the data that it wants to access are allocated. Accesses in local nodes are apparently faster, than accesses in remote memory nodes. Methods that manage shared resources in this thread-agnostic manner and making decisions without taking into consideration can lead system to poor scalability and performance that is variable and highly dependent from the workload that executes on the CMP.

There has been significant interest in designing solutions to address the problem of resource contention. The majority of these solutions are implemented in hardware and target mainly in performance aware cache optimizations, mostly cache-partitioning and optimization of DRAM controller. Those solutions aim to make those resources thread-aware, thus mitigate the contention among different threads sharing these modules.

Another class of solutions that is orthogonal to the above-mentioned approaches concentrates in thread scheduling. Scheduling techniques have drawn attention since the dawn of multiprogramming. Schedulers have been an integral part of the OSs, which are responsible to share resources among applications.

When uniprocessors were the norm in computing systems, the role of schedulers was to time-multiplex the cpu to applications. Scheduling policies were designed according to the nature of the computing system they were intended for. Therefore in desktops they were striving for system responsiveness, while in high performance computing systems there were optimized for high throughput. Scheduling was so important for the performance of systems that a lot of research was conducted on this subject, both from the academia and industry. This resulted in very efficient schedulers, therefore the interest on improving schedulers was diminished until the advent of the CMPs.

The multiple cores found in CMPs inserted one additional level of complexity to the scheduling problem. Schedulers not only have to share one cpu among multiple applications, but they additionally have to partition the available "space" of cpus, deciding which thread will execute in every cpu during a single time-quantum. Initially, OSs running on CMPs used schedulers designed for traditional Symmetric Multi-Processors (SMPs). These architectures consist of multiple processors that, unlike cores of modern CMPs, are completely independent to each other. In SMPs each core has its own cache hierarchy and all the only shared resource is the memory bus, which means that there are less points of contention. Apparently, SMPs use a more "flat" organization compared to CMPs, where memory organization is hierarchical.

2.2 Parallel Programming

Writing parallel programs is notoriously error prone. Concurrent execution of threads sharing data can lead to race conditions. In addition, careful design and implementation of software is necessary to achieve scalable performance. Writing parallel programs which execute efficiently on a diverse set of architectures is also a difficult challenge. Platform independent design in software development is a usual requirement, though in the multicore era this is often a significant burden. Optimizing parallel applications code requires good knowledge of the underlying system. However, multiprocessor architectures can vary significantly, presenting great differences in the number of cores they provide, their topology and the memory organization.

Parallelism is provided to programmers either as a library (Pthreads, MPI, OpenMP, etc) or as a language (Cilk, Haskel, Fortress, etc). An important part of those implementations is the run-time system, which is responsible for implementing all parallel operations during the application's execution time. The primary focus of parallel programming environments is to provide programmers with high level constructs that will allow them to express parallelism in their applications. At the same time, acknowledging the difficulty of writing efficient parallel programs, researchers and the industry struggle to remove some of the programming effort needed from the programmers and make it responsibility of the RTS, in an effort to make parallel programming more productive and efficient. This has resulted in larger and more sophisticated RTSs.

2.3 Run-time systems for parallel programming languages

Parallel programing is trying to exploit the multiple processing units found in current computing systems. Parallel application programmers decompose a large problem in parallel tasks that can execute concurrently and thus decrease execution time. When expressing parallelism with tasks it is considered a good programming technique to create a number of parallel tasks (`N`), significantly larger than the available physical cpus (`p`). This technique allows the RTS to adapt to the underlying system regardless the number of available cpus, since decomposing the problem in a large number of parallel tasks favors load-balancing which is a necessary condition to achieve good performance. Another way to utilize the available resources is to parallelize loops that do not have data dependencies among its iterations. In this case a number of threads is created (often user-defined) and each thread is responsible for the execution of a number of iterations.

When a parallel application is executed, the main responsibility of the RTS is to schedule the `N` parallel tasks to `t` available hardware threads. The number of available hardware threads is usually defined by the user or the OS. Usually the RTS implements its own scheduling policies at user space, acting as a small-scale operating system. It creates `t` OS threads and uses them to execute the parallel tasks in which the program is decomposed. RTS is responsible to create and terminate tasks, manage their memory and implement the scheduling policy with which it will assign tasks to software threads. There are several issues concerning the mapping of the `N` parallel tasks to `t` available threads. In cases where the computing load is uniform across the whole extent of the program portion to be parallelized (e.g. loop iterations), the total workload is decomposed evenly to `t` tasks, each being statically assigned to a separate hardware thread. However there are situations where the load among parallel tasks is non-uniform or the parallelism is dynamically created, and therefore it is difficult to efficiently schedule tasks to hardware threads. In those cases the RTS allows the programmer to define a number of parallel tasks significantly larger than the available hardware threads, a feature which is sometimes mentioned as parallel slack. This feature facilitates the task scheduler of the RTS to perform load balancing techniques in order to achieve higher cpu utilization and as a consequence higher throughput. Traditionally, parallel applications were not implemented employing some parallel RTS but using low-level APIs and explicit threading. In such cases parallelization was not expressed using tasks, and the number of software threads was either predefined, or defined explicitly by the user at run-time or the programer during development. The common choice in those cases was to use as many threads as possible to gain the higher possible speedup for the parallel region.

Scheduling parallel tasks in software threads from RTSs seems to be well studied and there is a number of parallel languages that implement the above-mentioned techniques. There are problems concerning performance portability for various hardware platforms, but there seem to be promising approaches. However the former techniques exhibit some significant limitations.

- The RTS is not aware of the architectural characteristics of the hardware platform. It takes decisions without taking into consideration which components hardware threads share, for example which cores share the L2 or L3 cache, or whether the system is NUMA.
- They assume that only one application is executing on the system. However, there might exist other applications concurrently on the system implemented in different languages, thus using a different RTS to manage their execution. Therefore, any decisions the RTS makes are oblivious to how they might affect execution of other applications.
- There is not the ability to alter the resources allocated to a RTS. Once a number of CPUs are assigned to a RTS it remains constant during the execution of the program. The malleability of thread numbers allocated to the RTS is an important property to achieve a resource aware scheduling policy or enforce Quality of Service (QoS).

2.4 Operating systems

Operating systems have a determinant role in application scheduling, yet the vast majority of them were designed when serial programming was the norm. Schedulers implemented in those designs were responsible for time-sharing the processor among the available cpus. To increase throughput and responsiveness they deferred execution of applications that were waiting for I/O until they were ready to execute and replaced them with others that ready to execute increasing the utilization of the cpu. This is easy to implement since whenever an I/O transaction takes place the OS is informed with a system call.

Completely Fair Scheduler (CFS) which is the current scheduler of Linux is designed to provide responsiveness for desktop systems by being equally fair to every application of the system. At the same time it uses a load balancing algorithm to distribute the workload equally among the available cores. Moreover, CFS is designed to be aware of the system topology, especially when load balancing threads.

Contention for shared resources can arise unpredictably depending on the profile of the workload being executed. Normally a workload consists of applications with different memory requirements. It has been shown [1] that executing concurrently a memory-with a cpu-bound application is beneficial, since the cpu-bound application requests a small amount of resources from the memory subsystem, and therefore does not affect the other one. The scheduler needs sufficient information either at compile or run time to make contention-aware scheduling decisions which create groups of programs that execute concurrently and their execution characteristics are complementary as far as their memory requirements is concerned. This kind of information could be acquired online by the OS using performance counters. Performance counters can be used to measure various statistics related to application execution, among of which are statistics regarding the utilization of memory hierarchy. Similarly, applications could provide significant information to the OS concerning their execution profile, e.g. whether they will make heavy use of the cache or the memory bus. Moreover, current OS schedulers do not treat threads of the same application as a whole. They handle them as if they were totally independent entities, consequently they cannot enforce policies such as gang scheduling which has been proved to increase throughput of an application.

Unfortunately, current state-of-the-art schedulers do not use such techniques to help them make optimal decisions about scheduling. Information provided by performance counters is not utilized at all and the flow of information between the OS and the applications is only towards one direction. The scheduler should be upgraded to receive feedback from applications and performance counters which, along with the knowledge of the underlying hardware platform, would lead to more efficient scheduling and, as a consequence, increased performance.

2.5 Current approaches

There are many approaches employing thread scheduling to deal with contention for shared resources in CMPs. Scheduling is a promising solution since it does not require any modification of the underlying platforms, in contrast to other methods that target at redesigning hardware (e.g. cache partitioning, modification of DRAM controllers) [2, 3, 4, 5].

Many approaches focus on techniques for scheduling workloads of multiple parallel applications to improve overall throughput. Corbalan, Martorell and Labarta [6] implemented Performance-Driven Processor Allocation (PDPA) scheduler to decide at runtime the number of threads per application based on the online measured performance of every application, though they do not take into consideration the contention on shared resources. Time-sliced gang-scheduling is a time-sharing scheme which executes concurrently in a time-quantum all threads of the same parallel application. This method improves response time of applications and provides mutual prefetching in cache hierarchy among threads using shared working sets, while reducing thrashing at the same time. This scheduling policy provides significant benefits for multithreaded applications:

1. Executing all threads of the parallel application at the same time-quantum prevents phenomena of threads halting at synchronization points (such as barriers and critical sections) waiting for suspended threads. This provides higher throughput for multithreaded applications such as HPC applications which typically benefit from inter-thread communication and require many synchronization operations.

2. The previous feature leads to applications being more responsive which is a common demand in desktop applications.

3. When scheduling threads of a multithreaded application concurrently their execution acts as mutual prefetching of shared data in shared caches among them.

This way some of the latency of accessing the main memory is hidden and throughput for memory bounded applications is increased.

Other approaches [1, 7, 8] use hybrid methods of space-time scheduling and make use of performance counters to determine which applications can be co-scheduled efficiently together. They monitor application performance while creating groups of applications to co-schedule, trying to converge to a solution that improves a metric. Apart from deciding the applications that will execute concurrently they also determine the concurrency level of each of them. Those methods try to deduce, again using performance counters, the best thread count for each application based on how well they can exploit the resources that are allocated to them, as well as the interference that they cause to other applications with which they co-execute. Bhadauria and McKee [1] propose time-space sharing techniques to increase throughput and energy efficiency in multithreaded workloads. Their implementations make scheduling decisions based on information that results from quantifying contention in shared resources and energy consumption. They use metrics that represent LLC misses, which is an indicator of the percentage of accesses that result in off-chip traffic and data bus usage. Their methods, then, classify applications based on these metrics and create co-schedule groups, trying to put in the same group applications with complementary characteristics, e.g. applications with low miss rates with others having high miss rates. Apart from throughput, the proposed methods target also at energy efficiency, using power meters to take energy consumption measurements. Zhuralev et al. [9] showed that LLC miss rate is very indicative of the contention for shared resources. They tried to measure the ratio of performance degradation due to contention in different shared resources (LLC, interconnects, DRAM controller and prefetchers) and showed that contribution of every contention point in performance degradation is interdependent and all those factors work in conjunction to result in the observed performance degradation. Although the dominant points of contention seem to be the interconnections and the memory controllers, since LLC is the last on-chip shared resource, LLC miss rate is an accurate indicator of the pressure put on higher level shared resources. The techniques proposed by Knauerhase [11], Zhuralev [9], Merkel [10] approximate contention measuring LLC miss rate online and use this information as input for their schedulers. Their primary concern is to match applications that are complementary in their demands from the memory subsystem. However, their experiments focus on multiple single-threaded applications that execute concurrently on a multicore system.

The aforementioned methods are designed to work over traditional OS's that do not implement contention-aware scheduling techniques, since it was not considered critical until now. A different approach is to design an OS from scratch, employing principles that are derived from the modern computer organizations. The hierarchical organization of shared resources needs to be accounted when designing scheduling policies. Barrelfish [12] is organized as a distributed micro-kernel system trying to adapt to multicore and probably heterogeneous systems. Similarly, Helios [13] introduces satellite kernels to address the challenge of heterogeneity. Fos [14] focuses on space-sharing and Tessellation [15] argues for time-space partitioning the shared resources with each application handling its own partition.

In this work we evaluate some of the approaches mentioned before and try to explore our own scheduling methodology. We demonstrate that current state-of-the-art scheduling policies are not scalable. We evaluate alternate proposed approaches that are shown to be more scalable and efficient. We focus on workloads with multiple multithreaded applications and employ time-space sharing scheduling techniques, trying to provide high throughput for the whole workload while not compromising performance of individual applications. Multithreaded applications usually consist of large parallel regions, e.g. large parallel loops with many iterations which are assigned to multiple threads to execute them. It is highly possible that those parallel regions scale differently. Acknowledging this, we consider current implementations that allocate a fixed number of resources throughout the whole execution of the application restrictive. We believe that a more fine-grain, dynamic allocation of resources that would take into consideration the actual needs of every parallel region of the application can lead to higher throughput and better resource allocation. Previous implementations strive to detect the level of concurrency at which a parallel application is most benefited and employ hybrid time-space sharing techniques to schedule a workload of multithreaded applications. Effective implementations try to quantify the contention on shared resources caused from concurrent execution of applications and create groups of them to be co-scheduled. Furthermore, our experiments concluded that, apart from finding effective co-schedules that diminish contention, it is important how the threads of each application will be placed on the available cores based on the architectural characteristics of the execution platform, such as which cores share a LLC, which cores belong to different NUMA nodes, etc.

Chapter 3

Preliminary evaluation and results

In our effort to explore the problem of scheduling multithreaded applications in modern CMPs, we conducted a series of initial experiments that highlight its multiple aspects and different implications. We came up with some interesting results which we present in the sections that follow.

3.1 Multithreaded applications scalability

Multithreaded applications can exhibit performance degradation not only when co-scheduled with other applications but also when executing solo. Typical reasons for this are the memory contention, load imbalance, synchronization overhead, intense true/-false sharing, etc.. While CMPs provide opportunities for parallelism for a single multi-threaded application, memory accesses can be a bottleneck for parallel execution. This can be true even if threads of a parallel application work on private data, without incurring notable interprocessor traffic. This is particularly true for memory-intensive parallel applications with large working sets (e.g. streaming applications), which may suffer from memory bandwidth saturation as more threads are introduced. In such cases, the application will not probably scale as expected, however efficiently parallelized it is, and the performance will be rather poor.

The above are a result of the memory subsystem design of modern architectures. Each socket in the platform has a maximum bandwidth to memory which is shared by all processing elements it encompasses (cores, hardware threads, etc.). Depending on the architecture, even multiple sockets might share access to main memory through a common bus. Given that even under perfect conditions (e.g. full software optimizations) the memory subsystem cannot fulfill a single thread's requests without having its core stalled, we can imagine the amount of pressure put on memory bus and how quickly it can be saturated when the number of cores increases. This is why the processor industry strives to provide improved bus speeds or alternate memory paths on each new processor generation (e.g. through Non-Uniform Memory Access designs), but unfortunately these enhancements have never been enough to make the memory subsystem keep pace with

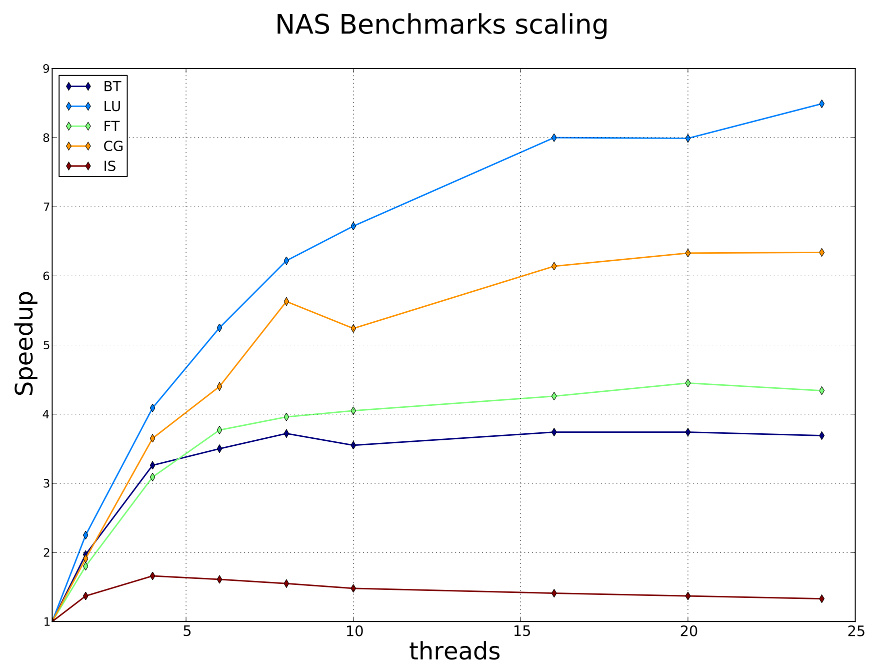


Figure 3.1: Scaling of NAS benchmarks

the increasing core counts.

As a result multithreaded applications cannot exploit the abundance of computing resources found in current CMPs and and this forces them to execute with a smaller number of threads. This makes space scheduling an one-way solution to address the problem of under-utilization of resources.

Sublinear scalability of multithreaded applications poses a limit on the number of resources they can be benefited from. In Figure 3.1 we show the speedup of some of the NAS benchmarks on a 24-core CMP. It is obvious that almost all applications fail to gain significant speedup when executing with more than 8 cores. While LU scales better than others it only exhibits a speedup of below 8 when executing with 16 threads. Even in this case it would be wise to assign less cpus to LU, and use the rest to co-execute more applications. This can lead to better utilization of the available resources.

3.2 Performance of state-of-the-art scheduling policies

Current state-of-the-art scheduler implementations, such as the Linux CFS scheduler [16], are designed to distribute threads across available cpus in a balanced fashion. In this way, they enforce some kind of space sharing among applications, yet this approach is defective since they treat threads of the same application as separate entities and not

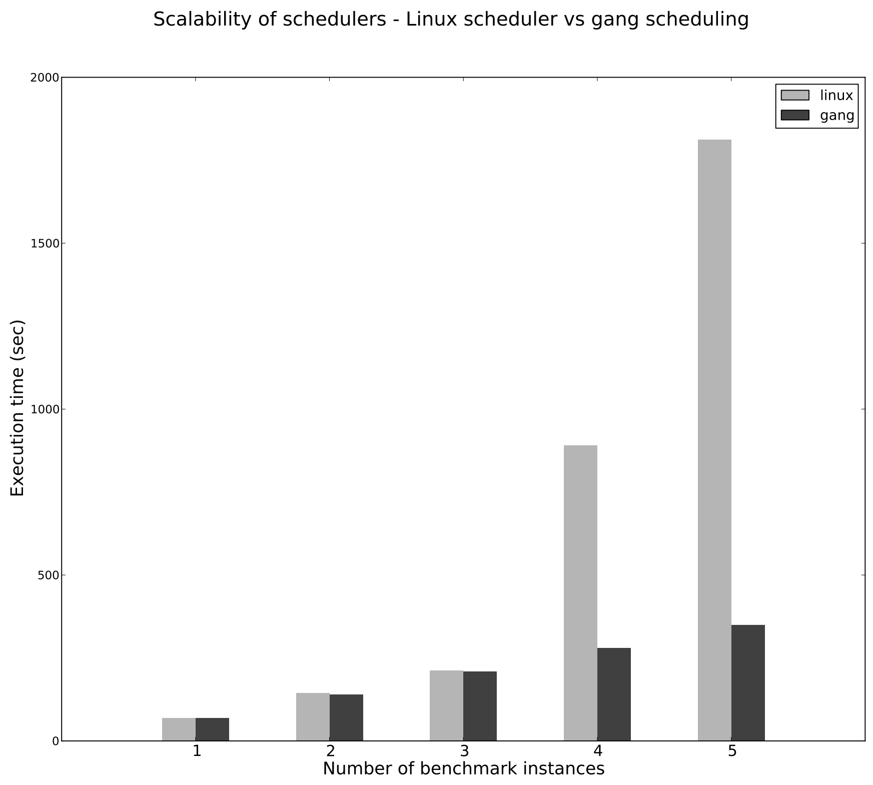


Figure 3.2: Failure of linux to scale when handling many threads

as components of a larger, single entity. This results in threads of the same application being scheduled in different time-quanta which undermines the progression of the application. Gang scheduling acknowledges this observation and dictates that threads of the same application must be scheduled in the same time-quantum. Therefore, threads of the same application are benefited from all the advantages of being concurrently scheduled, such as avoiding large waiting periods in synchronization events (critical sections, barriers), better exploitation of data locality under shared-cache configurations, etc.

The Linux scheduler aims at distributing all threads of the workload across the available cpus in a balanced way, thus there are not any cpus left idle at any time-quantum. Gang scheduling, obviously, does not achieve this level of utilization of cpus. However, scheduling all threads of one application at the same time achieves better performance.

The fact that Linux scheduler does not schedule threads of the same multithreaded application at the same time-quantum, causes their performance to degredate. This phenomenon becomes more intense when the total number of threads of the applications executing on the system exceed the number of the available cpus. When that happens threads of the same application are scheduled in different time-quanta. At the same time the contention in shared resources among different applications is increased and the

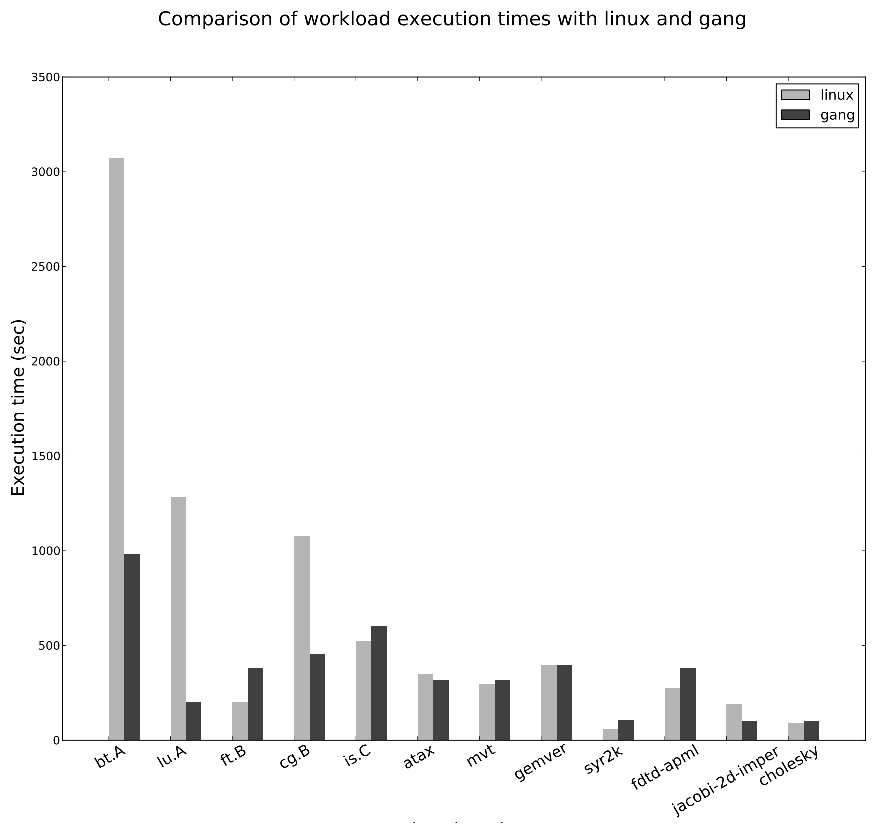


Figure 3.3: Comparison of executing a workload with Linux and a gang scheduler

conjuction of these lead to poor performance. Figure 3.2 shows that the Linux scheduler fails to scale when the number of software threads is increased, comparing with gang scheduling techniques. Gang scheduling avoids the impact of scheduling threads of one application to different time-quanta, which also reduces the contention for shared resources, resulting in higher throughput

Figure 3.3 presents the execution time of every program of our workload when executing using the Linux scheduler and a gang scheduler. Gang scheduling chooses all threads of an application (gang of threads) to execute them in the same time-quantum. This implementation of gang scheduler does not enforce space-sharing and as a result the gangs being created consist of one application (one program executes every time-quantum). Not many applications of the workload benefit significantly from gang scheduling. It is obvious, however, that the defective scheduling of Linux for multithreaded applications can result in poor throughput for some applications. Using gang scheduling for this workload, we achieved a speedup at the completion time of the workload of three times without decelerating significantly any application of the workload.

Even if gang scheduling is proved to provide higher throughput for multithreaded applications, this policy leaves many cpus unutilized because applications do not scale perfectly and are unable to exploit all the cores of a CMP. We therefore need to employ gang scheduling techniques that also enforce space-sharing policies, in order to fully utilize the available resources. Space-sharing, however, introduces contention for shared resources among co-scheduled applications. There are many points of contention in the memory subsystem, with the memory bus being one of the most prominent. Competition for it, can cause significant performance degradation, especially for memory-bound applications.

3.3 Studying effects of memory contention in scheduling

We use STREAM [18], a pseudo-benchmark that is designed to make streaming memory accesses without any cache reuse, in order to stretch the memory bus of a multicore system and measure its maximum sustainable bandwidth. We experimented in a 24-core Core-based CMP system in order to determine the effects in performance of applications that have different memory intensity. Apart from exploring the effects imposed from memory limitations to the execution of applications individually, we also conducted experiments to observe what implications does scheduling applications with different needs in memory bandwidth have in their performance. We created five different variants of STREAM, each having a different memory intensity level ranging from maximum to small. The maximum case (referred as stream-100) is the original version of STREAM, used to calculate the maximum sustainable memory bandwidth of our multicore system. STREAM implements a computational kernel (Code 3.1) that accesses three one-dimensional arrays, with double-precision floating point elements, which do not fit in the LLC. The access pattern is such that they do not make any reuse of data, thus all memory requests result in off-chip requests to DRAM.

```
void tuned_STREAM_Triad(STREAM_TYPE scalar)
{
    ssize_t j,k;
    #pragma omp parallel for shared(val)
    for (j=0; j<STREAM_ARRAY_SIZE; j++) {
        a[j] = b[j]+scalar*c[j];
    }
}
```

Code 3.1: STREAM computational kernel

The small case (stream-20) is a cpu-bound variant. It has been derived from the original version by inserting the appropriate amount of register-to-register arithmetic operations (Code 3.2) so that the program consumes 20% of the maximum memory bandwidth. The three intermediate cases (stream-40, stream-60, stream-80) have been produced in the same way, each consuming 40%, 60% and 80%, respectively, of the maximum bandwidth. To insert the number of arithmetic operations that will decrease the memory bandwidth consumed by stream, which is computed as the amount of data transferred divided by the amount of time for the transfer, we inserted a second loop that performs those operations, adjusting the number of iterations of the loop to achieve the desired percentage of used bandwidth.

```
void tuned_STREAM_Triad(STREAM_TYPE scalar)
{
    ssize_t j,k;
    #pragma omp parallel for shared(val)
    for (j=0; j<STREAM_ARRAY_SIZE; j++) {
        a[j] = b[j]+scalar*c[j];


        for (k=0; k<ARITHMETIC_OPS; k++) {
             val = val + scalar;
        }
    }
}
```

Code 3.2: STREAM computational kernel with arithmetic operations

For each variant we measured its standalone performance and the way their performance scales with the number of threads used to execute it. To gain some insight on the effects of memory bandwidth consumed by applications when co-scheduled, we also executed concurrently the versions of STREAM in pairs and triples and watched the way each version of STREAM is slowed-down.

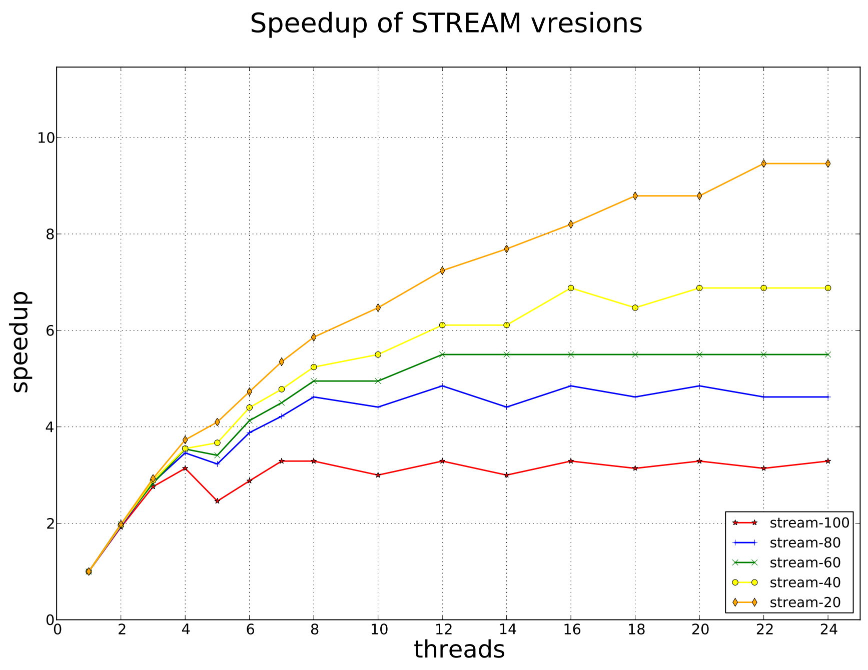


Figure 3.4: Scaling of STREAM versions

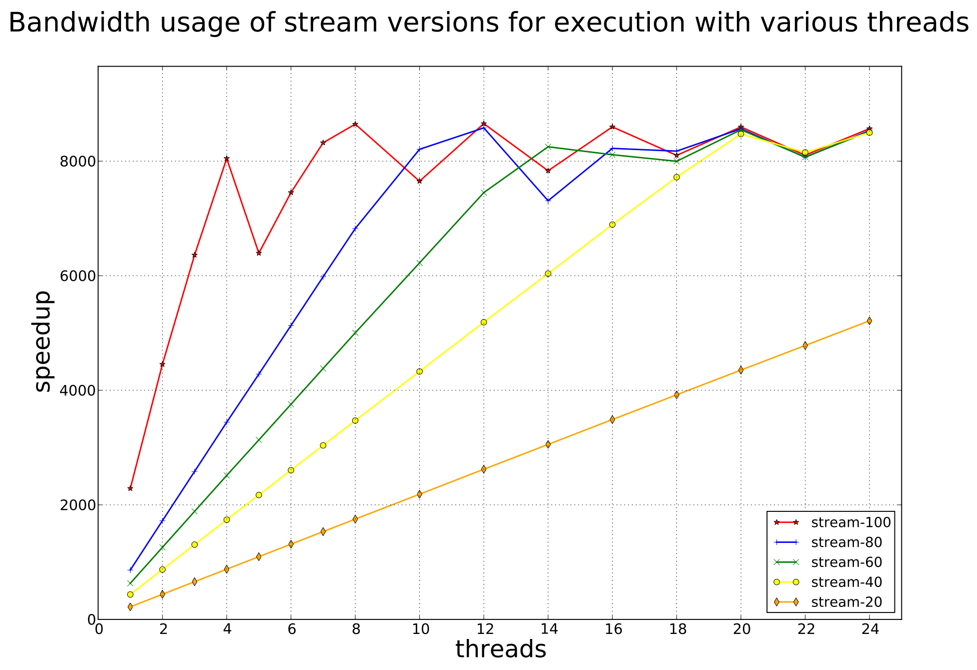


Figure 3.5: Bandwidth of STREAM versions for different threadcounts

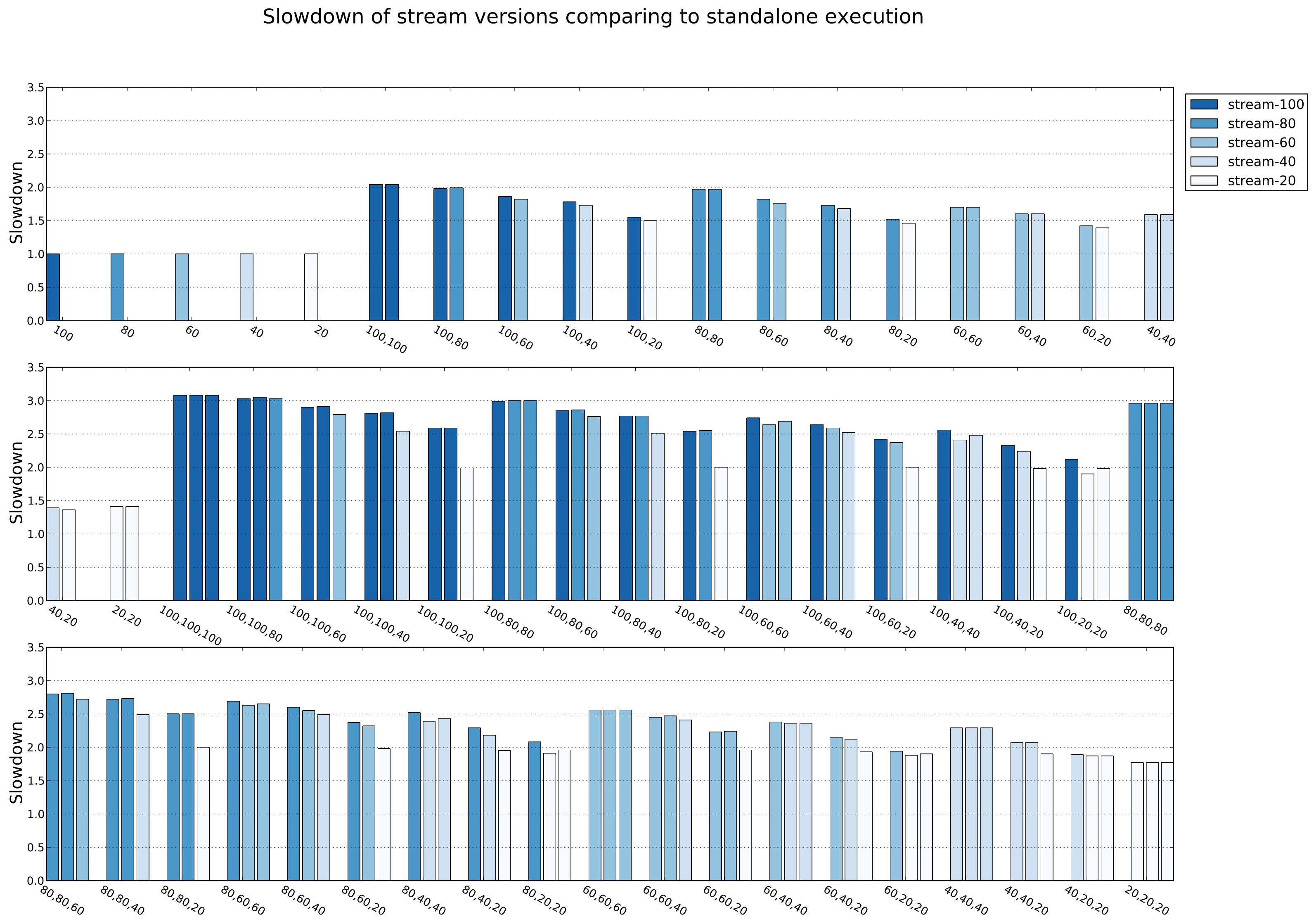


Figure 3.6: Slowdown of pairs and triples of STREAM versions

In Figure 3.4, we can see how the memory bandwidth usage of an application affects its performance. The figure depicts the speedup of the versions of STREAM that we created comparing to its serial execution. Applications that use less memory bandwidth scale better when the number of cores is increased.

This conclusion can be derived from Figure 3.5 as well. The stream-20 version is not significantly bound from bandwidth usage, resulting in linear scaling. This figure, also reveals the upper bound of memory bandwidth provided by the memory bus of our system. This is consumed by the stream-100 version when executing with 8 threads. This threadcount was used to create the rest of STREAM versions. Considering the bandwidth consumed by the default STREAM version (stream-100) when executing with eight threads, the versions of STREAM we created consume a percentage of this bandwidth when executing with eight threads. For example, stream-60 utilizes the 60% of the maximum bandwidth when executing with eight threads. We chose this threadcount to characterize the STREAM versions (rather than 24 threads which indicates the maximum memory usage of the version) because we wanted to use those versions and create groups (pairs and triples) to co-execute. In our 24-core system this was the appropriate value in order to not oversubscribe the system when executing triples of applications.

Figure 3.6 presents the results of co-scheduling applications of various levels of memory bandwidth usage and how their throughput is affected depending on the applications being executed with. Observing the most memory consuming pair and triple we see that co-scheduling two or three memory bound applications is not worse than executing them sequentially. Moreover, the level of performance degradation of one application is not depending only on the sum of the memory bandwidth usage of the rest of the applications running concurrently. The number of the co-scheduled applications is important. For example, stream-100 when executed in a pair with stream-80 it presents a slowdown of two times. However, when it is executed in a triple with stream-60 and stream-20 (which have a total memory bandwidth consumption of 80% of the total bandwidth) it is delayed 2.4 times. stream-20 when executing with stream-100 is delayed 1.50 times, whereas when executed with stream-40 and stream-20 it receives a slowdown of 1.93. stream-20 receives larger slowdown when executing with two other instances of stream-20 comparing to when executing with one instance of stream-100. We can conclude that the number of co-runners of an application is more important than the total bandwidth they use.

3.4 Thread placement issues

Apart from employing gang-scheduling based policies and deciding the co-runners of an application, we found that the policy with which a scheduler decides where (i.e. on which cores) the application threads should run is important. In traditional flat organized multiprocessors, cores share components of the memory subsystem uniformly. As a result, decisions concerning affinity of applications are unnecessary for the scheduler. CMPs, in the contrary, are organized hierarchically and as a result placement can be considered as an additional level of scheduling decisions. We explore two different placement policies, packed and spreaded. Packed placement tries to allocate to an application cores that are as close as possible to each other, therefore sharing many resources. Spreaded placement maps application threads as distant as possible from each other, which implies minimal resource sharing.

Each placement introduces different performance characteristics for every application of the workload being executed. Spreaded placement maps the threads of an application with the trend of different physical packages first. This means that threads fetch their data in different LLC, and as a result the multithreaded application utilizes larger effective LLC. This also, enables an application to use a larger ratio of the sustainable memory bandwidth. Consequently, the application suffers less from restricted memory resources and achieves higher throughput. On the contrary, packed placement policy places threads of an application as "closely" as possible hoping that they will be benefited by shared data fetched in shared caches. At the same time, this prevents sharing resources, such as LLC, with other applications in an effort to provide isolation to applications. The less sharing of resources among applications occur, the less likely will an application's performance suffer from contention.

Figure 3.7 below shows a typical CMP which consists of four physical packages, each having four cores. Each core has its own L1 cache, two cores of a physical package share a L2 cache and all cores of one package share a L3 cache. All packages connect to the memory bus with the Front-Side Bus (FSB).

Figure 3.8 depicts the spreaded placement for five threads. First, threads are placed on cores of different physical packages. When all packages have been filled with a thread, the fifth thread is placed round-robin to the first package again, but on a core that does not share the same L2 cache as the first thread. This placement tries to create the least possible contention for shared resources, such as L2 and L3 caches. For a multithreaded application, spreading its threads in this way provides larger effective LLC space to it.

Unlike spreaded, packed placement (Figure 3.9) maps threads trying to initially fill cores of one physical package before using other packages. In this way, it can provide performance isolation for multithreaded applications since different applications are allocated cores that do not share many resources.

Figure 3.10 presents the execution of a workload of multithreaded applications when executing on their own in our CMP machine when placing their threads packed or spreaded. It shows that spreaded placement provides higher throughput for multithreaded applications. Larger effective cache along with the optimal utilization of the available

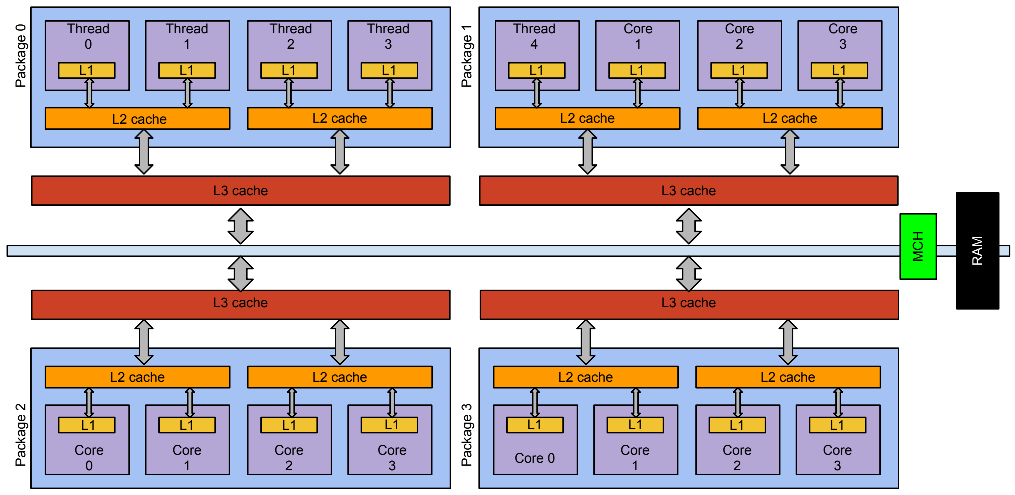


Figure 3.7: Typical organization of a CMP machine

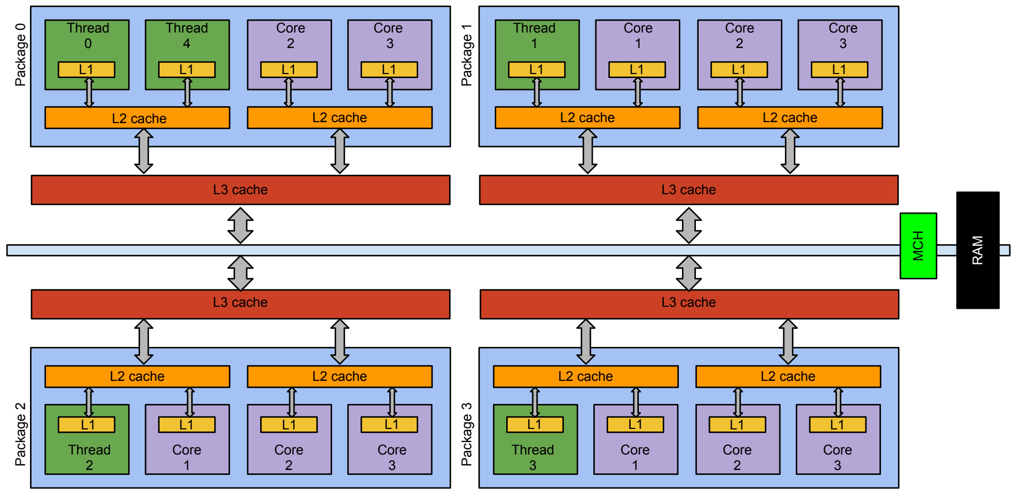


Figure 3.8: Spreaded placement of threads on a CMP machine

memory bandwidth benefits execution, significantly, for most of the programs of our workload.

In Figure 3.11, we see the results of executing our workload with a scheduler that implements gang-scheduling with space sharing. This scheduler uses a bin-packing algorithm to decide which programs will co-schedule in a time-quantum. After deciding co-schedules, the scheduler decides for every application the mapping of its threads to cores. Figure 3.11 depicts, for the two placements, the slowdown for every application when executed with this scheduler, comparing to the standalone execution with the corresponding placement. Packed placement provides isolation for the programs of

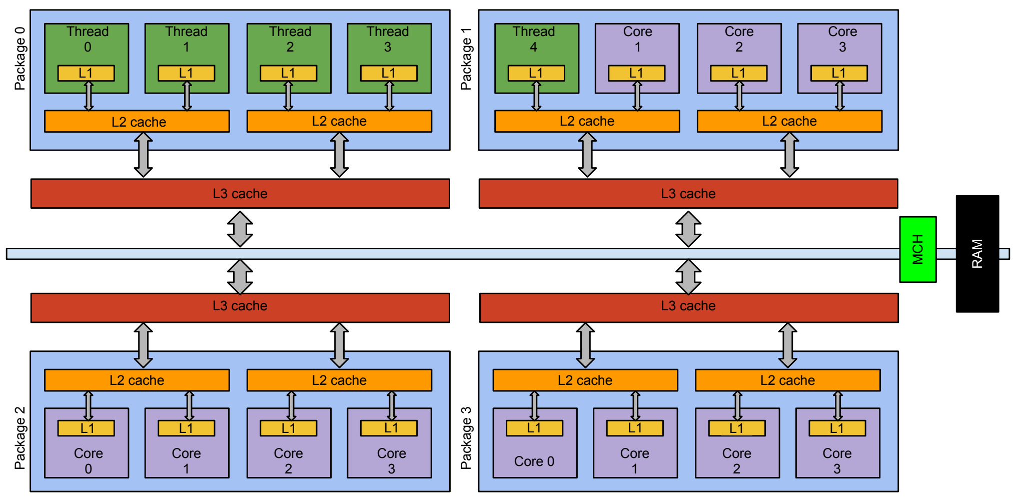


Figure 3.9: Packed placement of threads on a CMP machine

the workload when executing in a environment with contention (e.g. when co-scheduled with other applications). Almost all applications of the workload display larger slowdown when executed with spreaded placement.

Figure 3.12 describes the significance of the utilization of memory bandwidth for multithreaded applications. We used the original STREAM benchmark and executed it with various numbers of threads for both packed and spreaded placement policies. Each physical package utilizes a proportion of the total available memory bandwidth. As a result, spreaded placement has an advantage over packed, since threads fetch data using larger bandwidth. For example, execution with spreaded placement scales almost linearly from one to four threads. When threads are incremented, the additional thread is placed in a different physical package, thus using more effective bandwidth. On the contrary, packed placement maps additional threads to the same physical package and share the same amount of memory bandwidth.

This is depicted in the figure for memory bandwidth usage for the packed placement for the first six threads where the bandwidth usage is stable, because of the saturation of the memory bandwidth. The limited bandwidth usage is reflected to the execution time of STREAM which is stably high and only starts to decrease when executing with more than six threads when the seventh thread is placed to a new physical package. On the other side, spreaded placement uses linearly increasing bandwidth for the first four threads, which is reflected to its execution time.

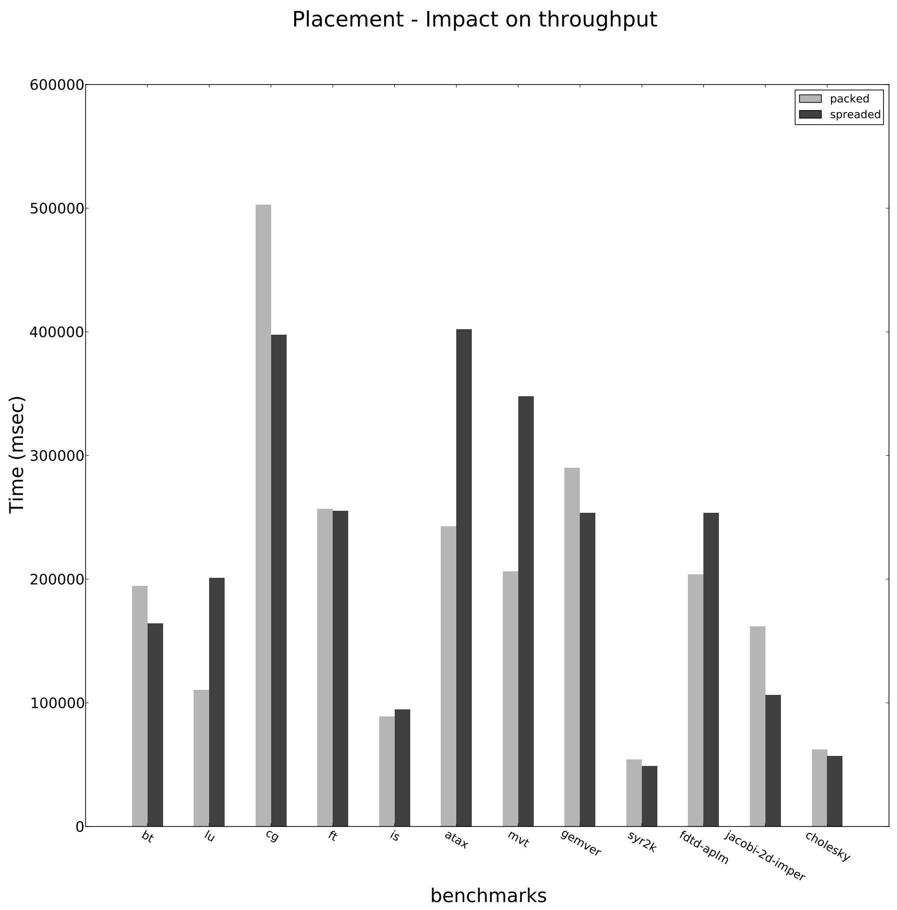


Figure 3.10: Execution with spreaded placement

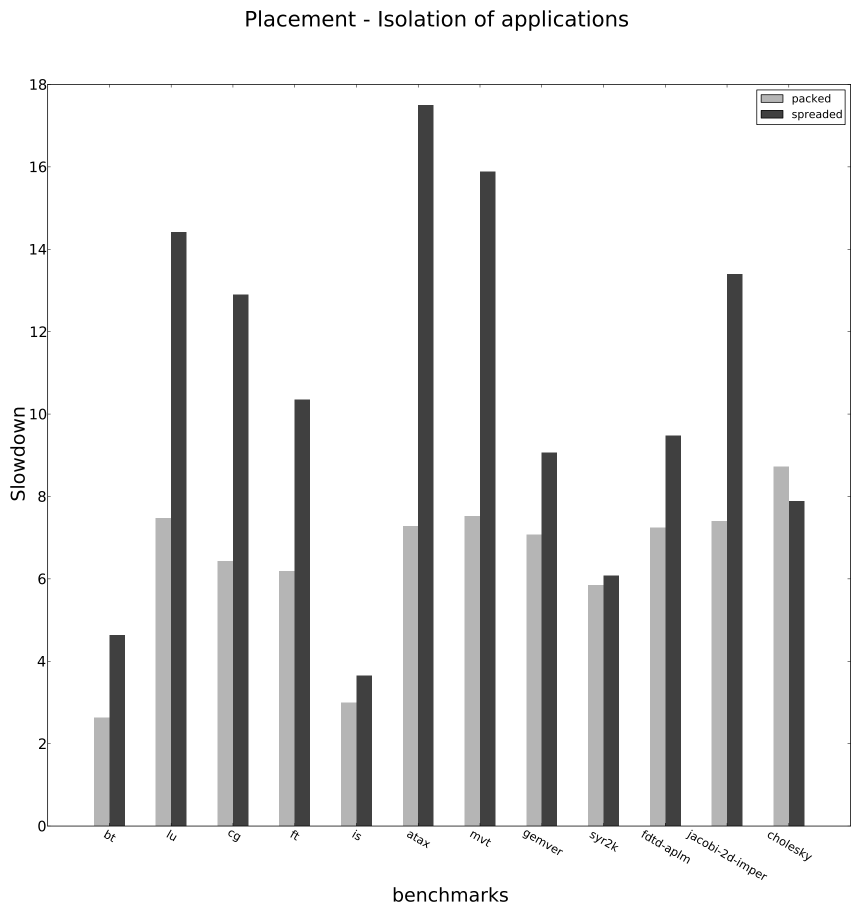


Figure 3.11: Execution with packed placement

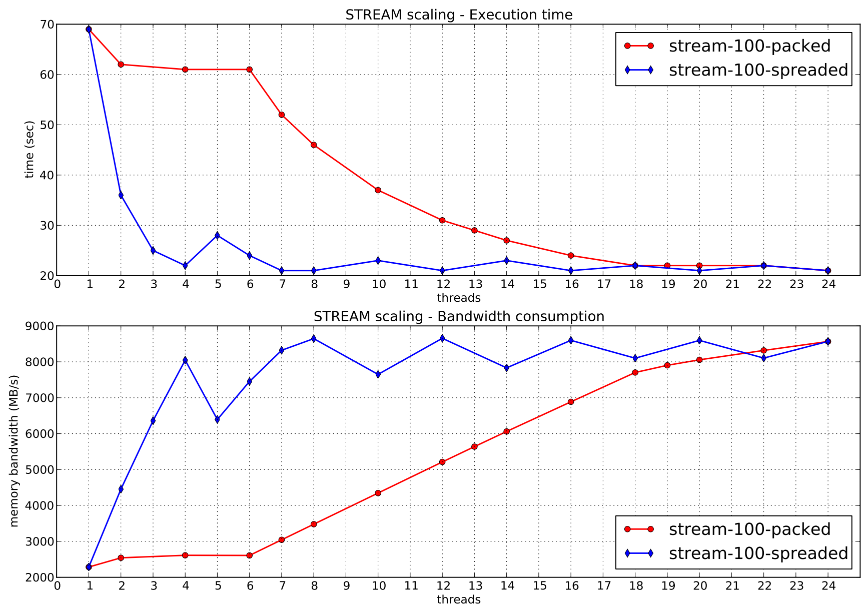


Figure 3.12: Speedup and bandwidth usage of STREAM with packed and spreaded placement

Chapter 4

Proposed scheduling methodologies

This section describes our approach on scheduling of multithreaded applications. Fist, describes the methodologies that we used in order to develop our schedulers, along with scheduling algorithms that we implemented. Second, it presents the metrics we used to evaluate the efficiency of our schedulers and finally describes the infrastructure we used to implement our schedulers.

4.1 Performance metrics

We define some metrics which will help us compare schedulers and evaluate their characteristics. Our goal is to provide high throughput for the whole workload without compromising the performance of any program of the workload. Moreover we want our schedulers to be fair towards all programs of the workload. Finally we examine the responsiveness of the programs, as we do not desire to let programs waiting for cpu resources for a long time.

We define the total execution time of a program i as t(i). This time breaks down into three portions:

$$t(i) = t_{alone}(i) + t_{waiting}(i) + t_{interference}(i) \tag{4.1}$$

- $t_{alone}(i)$ is the time for which a program would execute if it ran alone on the system, with the same configuration as in the co-scheduled case (threadcount, placement).
- $t_{waiting}(i)$ is the total time program i was waiting for the scheduler to execute it. $t_{waiting}(i) = sum_{j=1}^{m} T_j$ where T_j represents a period of time between two consecutive time-quanta in which i was scheduled to run.
- $t_{interference}(i)$ is the additional time a program executes due to interference introduced from co-scheduling it with other applications.

We define the fairness of a program as

$$f(i) = t_{alone}(i)/t(i) \tag{4.2}$$

This is actually the performance degradation that a program i suffers due to co-executing it with other programs, and as a result of the policy enforced by the scheduler. Global fairness is a metric of the overall fairness achieved by a scheduler, and is given by the following formula:

$$F = \sigma(f(i)) \qquad \text{for all programs } i \tag{4.3}$$

where $\sigma(f(i))$ represents the standard deviation of individual application fairness values. This metric indicates how equally the scheduler treats every program of the workload. We also measure responsiveness R for program i as

$$R_i = average(Ti, j) \qquad \text{for all quanta } j \tag{4.4}$$

which is the average period of time application i was in `FROZEN` state. Similarly, to estimate the responsiveness that a scheduler provides to the workload we use

$$max(R_i) \qquad \text{for every program } i \tag{4.5}$$

which is the maximum period of time the scheduler left an application waiting, during the execution of the workload.

4.2 Scheduling policies

We explored two classes of schedulers. Uniform schedulers use for every application a fixed thread-count throughout the execution. Adaptive schedulers execute programs that require different number of threads for different phases of their execution. Before an application enters a new phase of execution it informs the scheduler about the new number of threads it requires. Adaptive schedulers make a new scheduling decision every constant period of time called round, based on the updated required thread-counts of the applications. Round is the number of time-quanta in which all programs of the workload have executed once.

Schedulers implement time-sharing or hybrid time-space sharing techniques. They treat threads of an application as a whole, so any decisions they make concern the application (and as a result every thread of the application). Every scheduler creates gangs of programs and then performs a time-scheduling policy. Gang is a group of programs of the workload that will be scheduled to execute concurrently sharing the available cores of the CMP.

Programs of the workload do not have the same execution time. Programs that take longer time to execute will suffer less from contention than programs that finish their execution early, since, as some programs finish their execution the level of contention drops for the programs still running. However, we want the system to be in conditions of full workload throughout the whole execution so that our evaluation will be fair towards all programs of the workload. Thus, we re-execute programs that finish until all programs of the workload will have finished at least once.

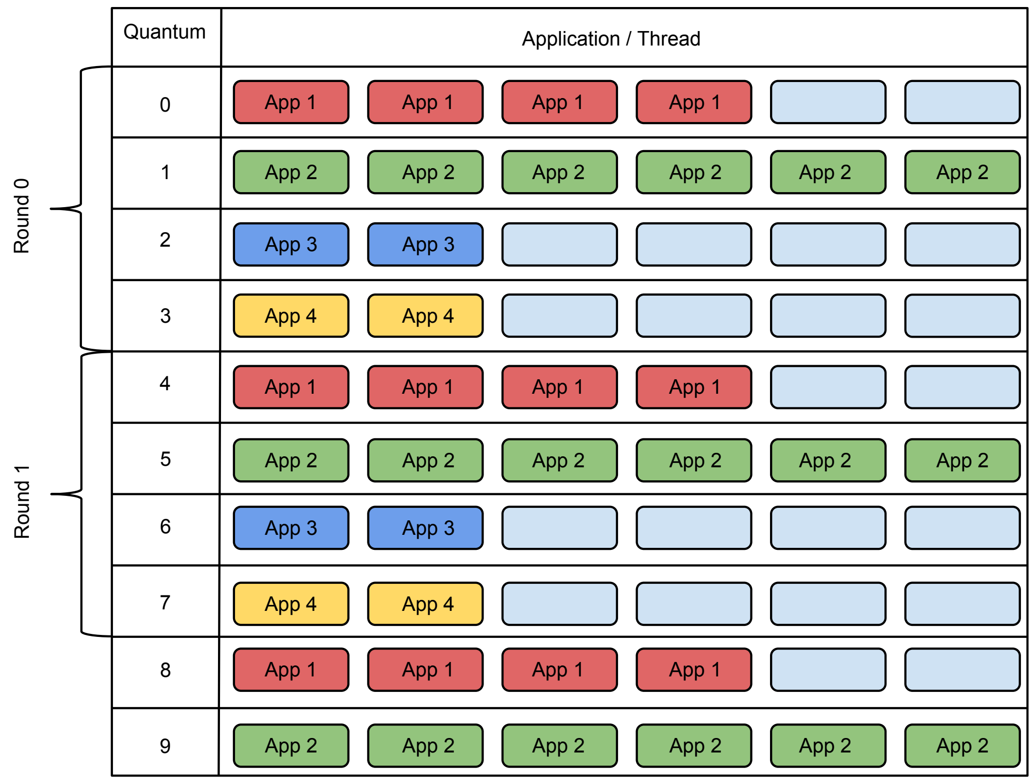

Quantum
Application / Thread
Round 0
Round 1
0 App 1 App 1 App 1 App 1
1 App 2 App 2 App 2 App 2 App 2 App 2
2 App 3 App 3
3 App 4 App 4
4 App 1 App 1 App 1 App 1
5 App 2 App 2 App 2 App 2 App 2 App 2
6 App 3 App 3
7 App 4 App 4
8 App 1 App 1 App 1 App 1
9 App 2 App 2 App 2 App 2 App 2 App 2

Figure 4.1: Placement of threads with GANG scheduler

4.2.1 Gang scheduler (GANG)

This is a uniform scheduler which does not co-schedule applications at the same time-quantum. Gang scheduling is the scheduling policy that executes all threads of the same multithreaded application at the same time-quantum. This policy is orthogonal to whether or not a scheduler will employ space sharing. When both gang scheduling and space sharing is enforced a number of applications is chosen to execute on the same time-quantum and their threads will execute simultaneously. The applications chosen to execute on the same time-quantum form a gang. The scheduler is responsible to create gangs. GANG scheduler does not enforce space sharing. As a result, GANG creates gangs that contain one application. Uniform means that applications execute with constant thread-count during their execution. The scheduler keeps a list of the programs that belong to the workload and time-multiplexes them on the available cpus of the CMP in a round robin manner. This scheduler provides isolated execution for every application, since it schedules one application at a time. In this way the available system cores are not fully utilized. A direct consequence of this is that the time needed for a whole scheduling round to complete (i.e. all applications to get a time-quantum of execution) is large, and therefore the time an application is swapped out is big.

Figure 4.1 depicts a scenario of four multithreaded applications scheduled with GANG in a system with six cores. Threads of one application are selected to execute at every time-quantum in a round-robin fashion. In this case the length of one round is fixed (four time-quanta).

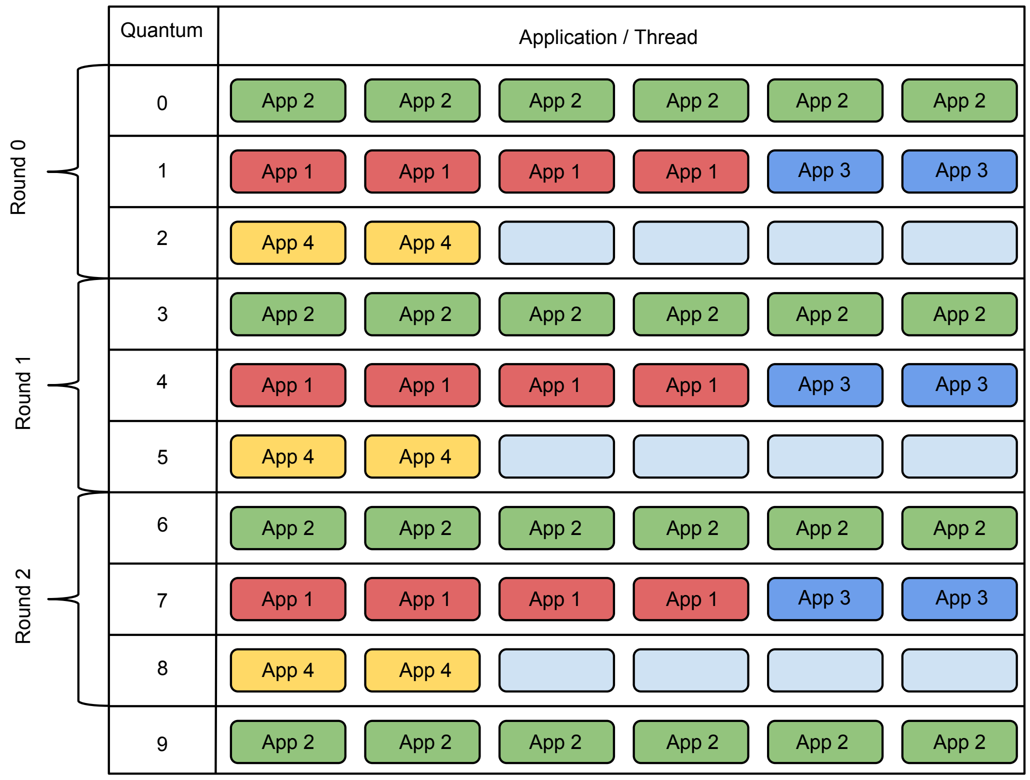


Figure 4.2: Placement of threads with GREEDY scheduler

4.2.2 Greedy scheduler (GREEDY)

This is a scheduler that tries to utilize the available cores at the maximum degree. It implements a bin-packing scheme to greedily place applications into gangs, so that the percentage of unutilized cpus in one round is minimized. In this case, as well, the applications execute with a uniform thread count.

Figure 4.2 depicts the choices that GREEDY scheduler will made when executing with the previous scenario with the four applications in a six-core system. GREEDY enforces space-sharing using a bin packing policy that tries to create gangs utilizing as many as possible cores. Space sharing results in smaller rounds (three time-quanta), which results in applications being scheduled more often.

When execution begins, GREEDY creates a descending sorted list of the programs based on their requested thread count. Afterwards, it creates gangs using the algorithm 4.1. In order to create the first gang, the algorithm chooses the application with the highest thread count from the top of the list, removes it from the list and adds it to the gang. Next, it traverses the list and chooses the first program with the highest thread count that fits in the remaining cores of the gang being created, until either there are no other available cores in the gang, or there is not a program in the remaining list that fits in the gang. The same procedure is repeated until every program of the workload is added in a gang.

```
List<Prog>  progs;  //initially populated with all
                    //programs  of the workload
List<Gang>  gangs;  //initially empty

sort(progs);        //sort programs based on their
                    //threadcounts
foreach  (Prog p: progs):
  progs.remove(p);

  flag = false;

  //find suitable gang in the 'gangs list
  //to add the program to
  foreach(Gang g: gangs):
    if (g.cores + p.cores <= total_cores):

      //program fits in this gang. Add it.
      g.add(p);
      flag = true;
      break;

  //flag will be false when there was not found
  //a gang in which the program fitted.
  if (not flag):
    //create a new gang to add program
    g = Gang.new()
    g.add(p);
```

Code 4.1: Algorithm creating gangs in GREEDY scheduler

GREEDY provides high utilization level of the CMP's cores, however its decisions are contention-oblivious, since the only factor used to make its decisions is the level of parallelism of each application.

4.2.3 Contention aware greedy scheduler (CGREEDY)

This is a scheduler that tries to enforce a contention-aware policy in the selection of applications to schedule. It uses LLC misses collected from previous profiling, and aims to match applications with complementary cache behaviour, i.e. those having high miss rates with others with few misses. The insight that led to the selection of LLC miss rate as an indicator of the contention an application causes is that, because LLC is the last on-chip shared resource, misses on that level reflect the contention caused in every off-chip subsystem beyond that, e.g. the memory bus and the DRAM controller. Moreover, a miss in that level incurs several cycles of penalty (typically, a few hundreds), and therefore can notably hurt performance. Matching an application that produces high miss rates with one that does not, limits the contention in shared resources and the performance degradation this phenomenon would cause.

CGREEDY sorts applications based on their LLC miss rate. Based on this sorted list, it creates gangs with the algorithm 4.2. It selects the application with the highest miss rate and removes it from the list. Next, it traverses the list from bottom to top (from the application with the lowest miss rate towards the applications with higher miss rates) and chooses the first application which fits in the gang being created (the sum of its needed cores added to cores required by the highest miss rate application fit the available cores of the system). The algorithm is repeated until the sorted list is empty and all gangs have been shaped.

```
List<Prog>  progs;  //initially populated with all
                    //programs  of the workload
List<Gang>  gangs;  //initially empty

progs.sort();  //sort programs based
               //on their LLC misses

while (not progs.empty()):
  //add program of the list with the highest
  //LLC miss rate to a gang
  p = progs.first();
  progs.remove(p);
  Gang g = Gang.new();
  g.add(p);
  g.cores = p.cores;

  //search if there is a suitable program in the list
  //beginning from those with the lowest LLC misses
  foreach_reverse(Prog q: progs):
    if (q.cores + p.cores <= total_cores):
    progs.remove(q);
    g.add(q);
    break;
```

Code 4.2: Algorithm creating gangs in CGREEDY scheduler

4.2.4 MINWAIT scheduler

MINWAIT scheduler is inspired by the CFS Linux scheduler, in the sense that it strives to minimize the waiting time of applications. It tracks the current waiting time of every application, which is the time elapsed since the last time an application was scheduled to run on the CMP. The scheduler keeps one gang to execute programs which is recreated at every time-quantum. Whenever a time-quantum is expired, the scheduler sorts the applications in descending order, according to their current waiting time. After that, MINWAIT chooses programs from the top of the sorted list until the cores of the gang are filled or there is not any other program that can fit in the remaining cores.

```
List<Prog>  progs;  //list of programs.
Gang  g;            //initially empty.

sort(progs);  //sort programs based on the
              //time they are waiting

//start creating gang from programs
//with higher waiting time
foreach (Prog p: progs):
  if (gang.cores + p.cores <= total_cores):
    progs.remove(p);
    gang.add(p);

//continue until no other program fits in the gang
    if (gang.cores_allocated == total_cores) :
      break;
```

Code 4.3: Algorithm creating gang in MINWAIT scheduler

MINWAIT scheduler tries to make maximum utilization of the available cpus. With GREEDY scheduler it is possible to create gangs that will leave some cores unutilized in some time-quanta while there are applications that could fit in the remaing cores. To overcome that MINWAIT does not use the concept of rounds, it creates one gang every time-quantum. As a result, an application is not restricted to execute one time per round. If there are available cores the same application may be scheduled in subsequent time-quanta.

4.3 Scheduling infrastructure

4.3.1 Basic components

All scheduler implementations are based on the scaff infrastructure. Scaff is a run-time system that orchestrates the execution of a workload of multithreaded programs. Scaff operates at user-level, on top of Linux-based systems. Its primary role is to provide a communication mechanism between a scheduler implementation and the programs being executed. The operation of scaff is relied on the collaboration of two subsystems, the executor and the scheduler. The executor is responsible for handling events regarding the execution, such as creation or termination of programs and handling the commu-

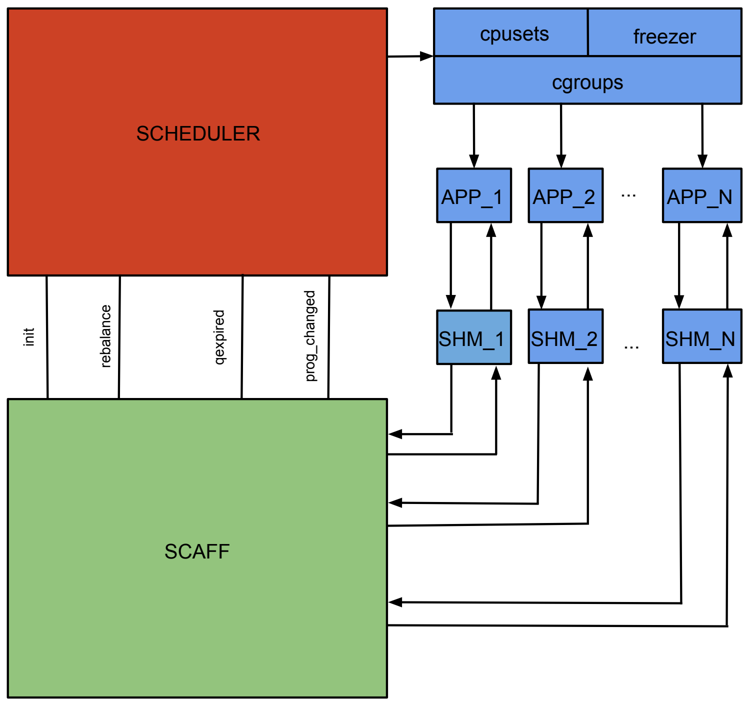


Figure 4.3: Run-time system architecture

nication between the scheduler and programs. The main responsibility of the scheduler is to make decisions on how the available resources should be shared among programs. Moreover, the executor provides an interface to assist schedulers manage the workload. Managing the workload means choosing which programs will execute in the current time-quantum, changing affinity of programs, and allocating resources to them.

The executor awaits for various kinds of events to come up and, for each event, it triggers the appropriate function of the scheduler. The scheduler, in turn, decides on the scheduling of the workload. The events that the executor is responsible for include creation and termination of programs, notifications that originate from programs and concern requests to the scheduler, as well as responses from the scheduler to those requests. The scheduler implements a policy based on which it takes decisions about when a program will execute, on which cores and whether some programs will co-execute. In some cases, it gets feedback from applications concerning their resource needs and decides the amount of resources that will be allocated to every program of the workload.

In the following sections we describe in detail the operation and the internals of each component.

Executor

The executor keeps information about the programs of the workload, certain kinds of events during execution and the current scheduler implementation that is used to execute the programs. Each program can be in four states:

1. WAITING: waiting for its time to come to be executed.
2. NEW: ready to be picked up
3. RUNNING: executing
4. FINISHED: program has finished its execution

Programs of the workload are given as an argument to the executor, along with the time that their execution must start. Until this time comes, they are in WAITING state. In NEW state are all programs that are about to execute for the first time. A list (`pnew_l`) of these programs is given to the scheduler, so that it will add them to the set of programs that handles. Programs that are handled by the scheduler are in RUNNING state. A program is in FINISHED state, either when it has finished its execution, or it got stopped. The executor adds all programs in FINISHED state to a list (`pfinished_l`) that is, again, given to the scheduler. Whenever a program changes to FINISHED, the scheduler removes it from the set of programs it handles and informs the executor so it will clean-up every structure concerning this program.

The executor keeps for every program of the workload information about its cpuset, the number of cores that the program has requested and a shared memory area, that is used for communication between the program and the scheduler. The cpuset (See 4.3.3) of a program is used to decide in which cores and memory nodes it is allowed to execute and allocate memory pages from, respectively. While a program is in RUNNING state, it can be either `FROZEN` (that is, stopped for this time-quantum) of `THAWED` (that is, running for this time-quantum).

The executor handles two kinds of events:

- `EVNT_NEWPROG`
- `EVNT_QEXPIRED`

`EVNT_NEWPROG` is an event associated with the start of the execution of a program. `EVNT_QEXPIRED` indicates the expiration of a time-quantum of scheduling. Each event is associated with a timestamp, at which it must be processed by the executor. The executor uses a priority heap to keep events. During execution, the scheduler checks for events that their time has come to be processed, and depending from the type of the event, calls the appropriate scheduler hook.

Scheduler

Scaff uses a scheduler to be able to manage the execution of programs. The scheduler is implementing a scheduling policy and is responsible for allocating resources among programs. It takes into consideration resource demands from programs and information provided by the executor about the executing system, and makes decision concerning "when" (if it implements a time-sharing policy) and "where" (if it implements a space-sharing scheduling policy) will a program execute.

A scheduler must implement the following functions:

- `void *init(void)`: This function is called at the beginning of execution by the executor in order to initialize the scheduler. Whatever `init` returns is stored by the executor, and passed to subsequent invocations of the scheduler.

- `void rebalance(void *sched_data)`: `rebalance` is called either when one or more programs are ready to be executed (NEW state) or, one or more programs have been terminated (FINISHED state). A struct describing a program in NEW state is kept in `pnew_l`. The scheduler must remove it from this list, and begin its execution. Respectively, when a program is in FINISHED state a struct describing it is kept in `pfinished_l`, from which the scheduler must remove and deallocate it.

- `void qexpired(void *sched_data, struct timeval *now)`: this function is called whenever a time-quantum has finished. The scheduler must, itself, add `EVNT_QEXPIRED` event in the priority heap, if it wants to implement time-sharing.

- `int prog_changed(void *sched_data, aff_prog_t *prog, int nr_threads)`: Whenever there is a change in a program e.g. when a program enters a new phase of execution for which its needs for resources is changed, `prog_changed` is called. Its purpose is to inform the scheduler that a program's requirements for resources have changed, so that it will reconsider its scheduling decisions. `prog_changed` must return 1 if `rebalance` should be subsequently called.

4.3.2 Design and architecture of scaff

The design of scaff aims to assist a scheduler implementation that will, effectively, interact with the workload of programs it is handling during execution time. It provides both the means of communication between the two ends, as well as the necessary mechanisms that a scheduler implementation can use to control the execution of programs.

Scheduler-applications communication

As far as the communication is concerned, scaff keeps a per program portion of shared memory to transfer data between itself and the programs. The most significant piece of information that is kept is the number of cores a program requires in order to execute and the number of cores that a scheduler, ultimately, allocated. Another piece of information is the number of available cores of the system. Moreover, there is the need for programs to trigger the executor to serve their requests, as well as the need for synchronous communication. To implement this, scaff uses two pipes. Whenever a program wants to make a request to the executor, it sends from the write-end of the pipe an identifier. This identifier informs the executor that this specific application has made a request. This pipe is unique and is kept in scaff's structure describing the execution. Scaff polls the reading end of this pipe, waiting for requests from programs. Every program has a pipe stored in the structure describing it, that is used as a barrier where programs synchronize with the executor while waiting the last to satisfy their requests for resources. Programs read from the read-end of the pipe waiting for scaff to write some arbitrary value to the write-end, after the scheduler has processed the request of the program.

Initialization and execution control

When execution begins, there is a phase of initialization scaff must carry out. First, it initializes all the data structures to keep track of the execution. These are the lists `pnew_l` and `pfinished_l` that are used to keep new and finished programs, respectively, the priority heap where it keeps events during execution and a hash table which maps the structures that describe programs with their process id's (pids).

The workload that must be executed is given as a command line argument in a configuration file. Scaff parses this configuration file and begins execution of every program of the workload. For every program, it allocates and initializes a structure (`aff_prog_t`) that will be used to describe it throughout the execution. This structure contains a `cpuset_t` field that is used as a handler for the program's cpuset. Scaff uses `cpuset_create()` to create a new cgroup to the cpuset virtual filesystem instance. cgroup which is a utility of Linux kernel that allows to cluster a bunch of processes and assign them specific characteristics, they are described in detail in subsection 4.3.3. During execution this `cpuset_t` handler will be used to modify program's affinity. The `aff_prog_t` also contains a pointer to the shared memory that the program will use to communicate with the executor. Scaff initializes this portion of shared memory and sets some of the fields. These fields are the pipe that a program will use to notify executor about a new request, the number of cpus available in the current execution system, the cores allocated initially to the application and a pointer to the application's `aff_prog_t` structure that will be used as an identifier, when the program will make a request to the executor, so that the last will distinguish requests from different programs.

After initializing the new program's `aff_prog_t` structure, the executor will `fork()` a new process. The new process will in turn use `execl()` to begin the program's execution on a new shell. The executor will wait for the program to freeze itself, and then it attaches it to its new cpuset which, at this moment, contains all the cpus of the system. Finally it pushes on the heap an `EVNT_NEWPROG` event. The new program will remain in `FROZEN` state until it is time for this `EVNT_NEWPROG` to be handled. The scheduler will then take over responsibility for its execution.

After parsing the configuration file, scaff initializes the scheduler that will be used, which is given as a command line argument, as well, and installs signal handlers. Signals that matter to scaff are `SIGCHLD` and `SIGTERM`. The first one corresponds to normal termination of a program, and will cause a call to the scheduler, which will handle its internal structures and, probably, make a new scheduling decision. The second signal indicates an unexpected termination of a program (with a `SIGKILL` for example) and is handled as an error, that causes execution to abort.

Additionally, there is the initialization that has to be performed by each program, separately, before its execution begins. We do not want to make great changes to the source of workload programs, so scaff provides a pre-load dynamic library with which is linked every program of the workload. This library provides functionality that takes care of the initialization, just before the program's execution begins and cleanup after the program finishes its execution. The preload library interacts with shared memory's fields and takes care of communication between the program and the executor. As for the initialization, it opens a file descriptor for the shared memory area that has already been created by the executor. Every program has a distinct integer value as an id. Shared memory objects use a name of the form `/somename` to be identified. The same semantics are used to distinguish different cgroups of a Cgroups' instance, such as the freezer and cpusets that we also use in our implementation. Freezer is a cgroup used to start and stop processes without signals, and is described in detail in subsection 4.3.3. So every program is associated with a name of the form `/AFF_ID`, where ID is the id of the program. In this way we can separate among different programs' freezer, cpusets and shared memory objects. Using this name, the preload library takes care of the creation of a new freezer cgroup and attaching the program to it.

Setting the number of threads of applications

After the initialization phase, the library sets the program to `FROZEN` state. The program will remain in this state, until its time will come to go to RUNNING state. Then, the scheduler undertakes the responsibility to decide when it will "freeze" or "thaw" it. The preload library also provides a C function (`affhook_region_notify()`) as an interface, which applications can use to make requests to the executor. Providing this function relieves the application from having to deal with executor's implementation specific functions and data. Applications use this function whenever they want to request a new thread count, for their next phase of execution. `affhook_region_notify()` writes in shared memory the thread count and then sends through the write-end of the executor's pipe the application's handle (a pointer to `aff_prog_t` structure). The executor identifies which application has made the request and handles it. After the request is served, the executor sends an arbitrary value to the write-end of the application's pipe. The application that is blocked, waiting to receive from the read-end, is now unblocked. The function reads the number of thread the executor allocated for this phase of execution and sets the thread number with a call to the runtime of OpenMP.

Executor's cotrol flow

Scaff's duties during execution reduce to handling execution events, programs notifications and signals. As mentioned, signals that scaff handles are `SIGCHLD` and `SIGSTOP`. `SIGCHLD` indicates a program of the workload has finished its execution. `SIGSTOP` means a program has been forced to stop (due to an error during its execution, or it was sent a `SIGKILL`, etc.) and it is considered erroneous behavior. Execution finishes, when no programs have been left, in any state of execution. Until then, handles events and programs notifications iteratively.

The executor pops events from the priority heap until it becomes empty. Every event is associated with a timestamp that indicates when it must be handled by the executor. The executor tests the event with the highest priority and compares the current timestamp with this event's timestamp and if its time to be processed has come the executor pops it from the heap and handles it. Handling events requires different actions to be taken, depending on the kind of the event. When the executor handles an `EVNT_NEWPROG` event, the program it refers to is added to the `pnew_l`. On the other hand, when the executor handles an `EVNT_QEXPIRED` event, a time-quantum has expired, and therefore if the scheduler has implemented the `qexpired()` function, this function is called. When no other events are to be handled for the time being, if programs have been added to `pnew_l` then the `rebalance()` function of the scheduler is called. This procedure returns the amount of time until the next event in the heap must be handled. Until then, the executor waits for program notifications.

Scaff uses the `select()` system call to check if there is a program notification. `select()` receives as an argument three sets of file descriptors: `readfds` will be watched to see if characters become available for reading, `writefds` will be watched to see if writing to one of them will not block and `exceptfds` will be watched for exceptions. Moreover, `select()` receives a `struct timeval`. When the time indicated by this struct elapses without a change occurring to any of the file descriptors, `select()` returns with return value 0. When a number of file descriptors become ready for the I/O operation corresponding to the set they belong to, `select()` returns with return value this number, having updated the `struct timeval` to the time left from the initial timeout. In our case, `select()` is called with the read-end of the executor pipe in the `readfds` set, and the `struct timeval` that the handling of events returned. In other words, the executor is polling for program notifications, until the next event in the heap must be handled. When a program notification has arrived, the executor reads from the read-end of its pipe the handle of the program that sent the notification.

As mentioned, this handle is a pointer to the `aff_prog_t` structure of the program. Before sending the handle, the program has already set in the shared memory area the number of threads it requests. If implemented, the `prog_changed()` function is called, so that the scheduler takes into account the program's request. When the initial amount of time passed to `select()` has elapsed, the executor checks if there are any other programs left (in any state) and if not, execution stops. Otherwise, the procedure goes back to handling events.

4.3.3 Underlying tools and mechanisms employed

Implementing a scheduler requires several tools, in order to be able to manage execution of programs. When it comes to scheduling on multicore systems, apart from deciding when will a program execute, it is also necessary to make a decision about how will the scheduler distribute the available cpus among programs. We can achieve these goals using cpusets and freezer, a couple of tools provided by linux. Cpusets is a system that allows the scheduler to bind a program on a set of cpus and restrict its execution in this set, while freezer is used to start and stop execution of running programs. Both of these are part of a feature provided by Linux called Control Groups (Cgroups).

Cgroups

Cgroups are a feature of Linux kernel which allows aggregating or partitioning tasks (processes) in hierarchical organized groups. We can affect the way tasks execute by configuring the control group they belong to. There are several concepts associated with Cgroups:

- A *cgroup* is a group of tasks associated with common execution characteristics.
- A *subsystem* is a module which uses the grouping of tasks provided by Cgroups, so it can handle different sets of tasks in certain manner. It is usually a resource allocator which schedules, or sets per cgroup limits on the use of system resources. However, it can be any module operating in a group of processes.
- An *hierarchy* is a set of cgroups arranged in a tree and a set of subsystems associated with it. Every task in the system may be in exactly one cgroup in the hierarchy. However, there can be more than one hierarchies, at the same time, in the system. Each one of these is a partition of the tasks in the system. Different subsystems can be associated with different hierarchies and determine execution characteristics for sets of tasks. It is useful to be able to keep different hierarchies, so we can use each to control different aspects of execution.

Cgroups are exported as a virtual filesystem and can be easily handled from userspace. User level code can create, handle and destroy cgroups by name in an instance of the cgroup virtual filesystem. This filesystem includes files that contain information about this cgroup instance and the subsystems associated with it. Besides, userspace code can define behaviour of a cgroup by changing values of those files. For example, when using cpusets, every cgroup of the cpuset filesystem contains the files `cpus` and `tasks`. If we want a task to be executed in cpus 0 and 1 we can write its pid in tasks and values 0,1 in cpus.

Cgroups facilitate applications management. Every cgroup instance is associated with a set of subsystems that provide certain execution characteristics. A task is executing with the constraints enforced by the cgroup of the hierarchy it belongs. Those constraints are inherited by all descendants of that task. So all children (tasks created using `fork()` for example) belong to the same cgroup with their father, until they are placed in another cgroup of the hierarchy. This is especially useful when we want to schedule multithreaded applications. Since gang scheduling threads of the same application has proven to be beneficial for its execution, we can schedule all threads of this application at the same time, without having to take special action for every thread separately. Since all threads of the application inherit the same cgroup, we can ,for example, start or stop it (using freezer) by just starting or stopping the cgroup the application belongs to.

Freezer subsystem

In order to enforce a time scheduling policy we have to choose which programs of the workload will be running and which will be waiting for the next time quantum. The freezer subsystem is the Linux facility that helps us achieve this.

We have seen that Cgroups provide a grouping of all the tasks of the system. Those groups are hierarchically organized and different subsystems can determine common execution characteristics for tasks belonging to the same group. When using the Freezer, we can "freeze" or "thaw" groups, meaning that we can have tasks of the group running or waiting, respectively. At first, every task of the system belongs to the root cgroup. We can then create different cgroups and organize them, putting those that should run concurrently at the same cgroup.

The Freezer is used like every other subsystem, by writing values in files of the virtual filesystem. The most interesting files for us are `tasks` and `freezer.state`, which are present in every cgroup of the hierarchy. `tasks` contain the pids of all processes that are member of this particular cgroup. `freezer.state` contains a string describing the state of the cgroup. There is no `freezer.state` file in the root cgroup. Since all processes of the system belong, initially, to that cgroup, we cannot have the whole system `FROZEN`. When freezer is used every cgroup can be in one of the following states:

- `THAWED`: Every task of the cgroup is allowed to execute.
- `FROZEN`: Every task of the cgroup is waiting.
- FREEZING: The cgroup is being `FROZEN`.

Therefore, to manipulate tasks, someone has to write either `THAWED` or WAITING to the `freezer.state` file and every task belonging to this cgroup will be handled accordingly by freezer.

Using signals `SIGSTOP` and `SIGCONT` is usually not sufficient[cb] when we want to handle processes from userspace. Those signals can b[cc]e caught by the processes we want to handle, or their parent who might be waiting or ptracing them. If a process that we want to handle is designed to watch for `SIGSTOP` or `SIGCONT`, it could be broken by attempting to stop it or resume it using signals. This is because, if we send for a example a `SIGSTOP` signal to a process designed to catch this signal, this process won't be able to distinguish whether the signal is meant to send the process in `FROZEN` state, or is regarded with some other process-interior function.

Sometimes freezing a task might be unsuccessful. In those cases, writing `FROZEN` to `freezer.state` will return `EBUSY` and the cgroup will remain in FREEZING state until one of the following occurs:

- Someone writes `THAWED` in `freezer.state`
- Someone re-tries to write `FROZEN` in `freezer.state`
- The task that caused the cgroup to block is removed from this cgroup.

In our implementation we create one freezer cgroup for every program of the workload. All descendants of those processes belong to the same cgroup as their father. In this way we can freeze or thaw at once all threads belonging to the same multithreaded applications, behaviour that is mostly desired.

Cpusets

Cpusets is a mechanism provided by Linux to constrain the execution of tasks to a set of cpus and memory nodes. Cpusets extend the existing mechanisms of Linux kernel that specify in which cpus a process is allowed to execute (`sched_setaffinity`) and from which memory nodes it is allowed to allocate memory pages (`mbind`, `set_mempolicy`). Calls to those mechanisms are now filtered through cpusets. For example, a call to `sched_setaffinity` will only add to the affinity mask of the process the cpus that are set in the cpuset of that process. In a similar fashion, calls to `mbind` and `set_mempolicy` will be filtered to memory nodes allowed by the task's cpuset.

Cpusets form a hierarchy in which every cpuset is allowed to include a subset of cpus of its direct ancestor. This hierarchy becomes visible to user-level through a virtual filesystem. User-level code can create cpusets in this filesystem and attach processes to them. After system boot, one cpuset is created that contains all resources of the system (cpus, memory nodes) and all the processes are attached to this cpuset.

Cpusets can be created and destroyed using `mkdir` and `rmdir` system calls or shell commands. Attributes of each cpuset are determined, as with every Cgroup, by writing values in certain files of the filesystem. The most important files for Cpusets are:

- `cpus` contains a list of cpus currently belonging to this cpuset
- `mems` contains a list of memory nodes currently belonging to this cpuset
- `tasks` contains a list of task pid's attached to this cpuset
- `cpu_exclusive` defines if this cpu placement is exclusive for this cpuset
- `mem_exclusive` defines if this memory nodes placement is exclusive for this cpuset

When created, tasks inherit the cpuset attachment of their father. Later, tasks can be moved in any cpuset, given that the permissions of that cpuset directory allow it. We create one cpuset for every process of the workload. When executing parallel regions the threads that are being created inherit the application's cpuset, so that their execution is restricted to the proper set of cpus.

Chapter 5

Experimental evaluation

5.1 Evaluation platform

Table 5.1 describes the characteristics of the platform on which we conducted our experiments. It is a CMP with four sockets, six cores each. Every core has its own private L1 cache, two cores of a socket share a L2 cache and all cores of the socket share the L3 cache. All sockets are connected with the main memory through a FSB. In Figure 5.1 there is a graphical representation of the system organization.

Platform	
# of packages	4
Cores/Socket	6
Threads/Core	1
CPU frequency	2.66 GHz
Chipset interface	FSB 1066MT/s
L1 Cache	L1D,L1I: 32KB
L2 Cache	3 MB, shared per 2 cores
L3 Cache	16 MB, shared
RAM	28 GB

Table 5.1: Platform Characteristics

We executed our experiments on a Debian GNU/Linux, with version 3.7.10 kernel. All benchmarks, the infrastructure and tools we implemented were compiled with gcc 4.4.8 (which implements version 3.0 of the OpenMP standard) with O2 optimizations and used glibc 2.11.3.

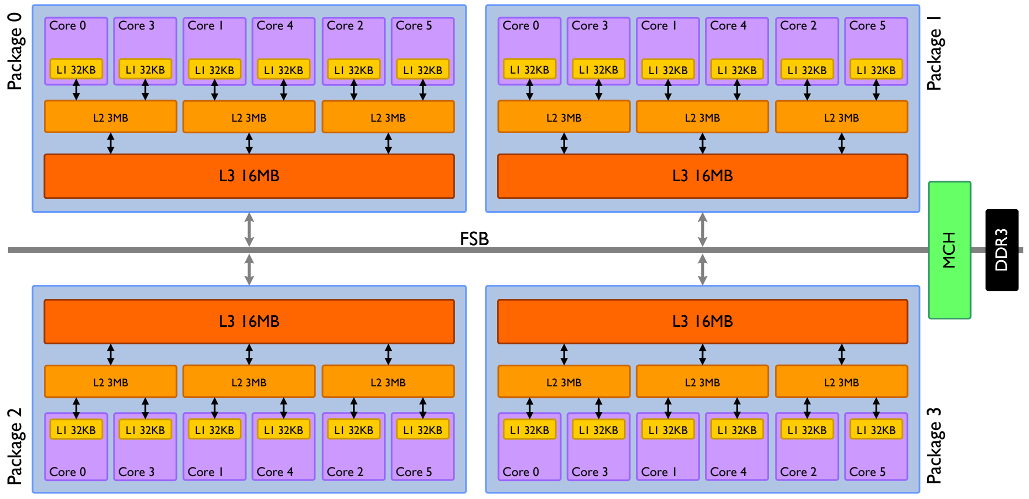


Figure 5.1: Dunnington organization

5.2 Workload profile

We chose a mix of benchmarks parallelized with OpenMP [21] from NAS [19] and polybench [20] suites. The benchmarks consist of multiple parallel-for loops, and among them there are some programs that are memory bound, consuming many computing cycles for memory operations, and others that are cpu intensive. Initially we performed a series of experiments to determine the execution profile of the benchmarks and to discover the way each program scales. This information gave us a view about the ability of a program to take advantage of the available cores of the system. We determined the ideal thread-count for every application as the thread-count for which the application achieves the 80% of its maximum speedup. We used these thread-counts for most of our experiments.

Benchmarks	Dataset size	Ideal threadcount	speedup	LLC miss rate (misses/1K instructions)
BT	A	4	3.26	2.5
LU	A	12	7.00	0.6
CG	B	8	5.6	3.0
FT	B	6	3.8	1.3
IS	C	2	1.4	44.8
atax	LARGE	8	3.1	30.6
mvt	LARGE	12	3.0	20.8
gemver	LARGE	8	3.3	33.5
syr2k	LARGE	22	10.0	0.2
fdtd-apml	LARGE	12	4.0	0.3
jacobi-2d-imper	LARGE	12	7.0	0.8
cholesky	LARGE	20	10.1	0.5

Table 5.2: Workload characteristics

Figuer 5.2 shows that our workload consists of applications with various execution

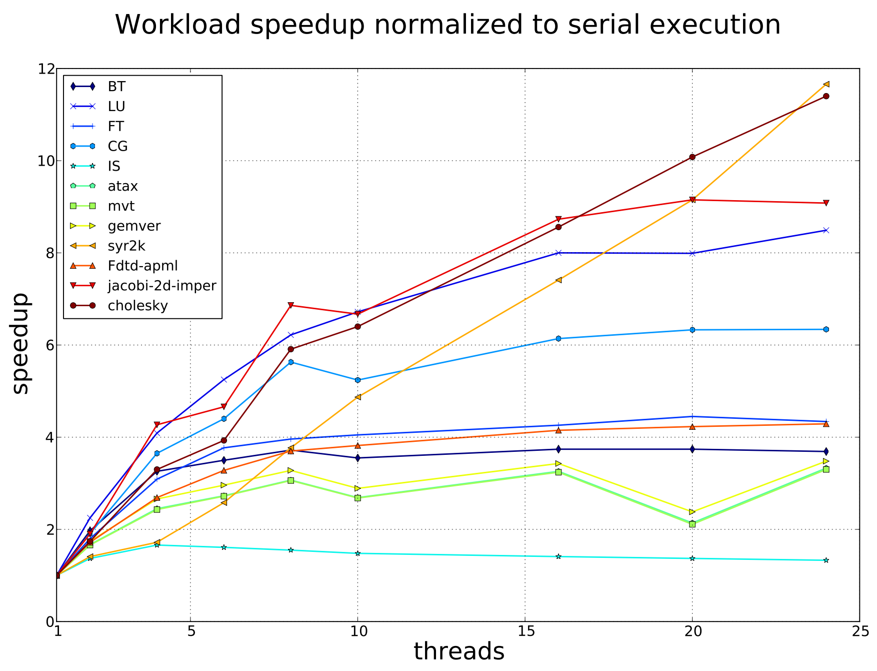


Figure 5.2: Speedup of benchmarks of our workload

profiles. None of them scales perfectly, which is an indication of why space-sharing among applications is important, since none of them is able to fully leverage all the cores of the system. We also needed to acquire information about the cache behaviour of each program. We were interested mostly in the last level cache (LLC) misses, since LLC is a significant point of contention when executing on CMPs, and a miss in that level usually costs tens of cycles.

Figure 5.3 shows the LLC miss rate (misses per 1K instructions) of every application of our workload when executing with the thread-count we selected for it. Applications present varying LLC usage profile, similarly to their varying performance scaling with the number of cores. For example, IS, atax, mvt and gemver, that suffer large miss rates, scale quite poorly as Figure 5.2 shows, while cholesky, jacobi-2d-imper and fdtd-apml with few misses per 1K instructions, perform much better than any other application in the workload.

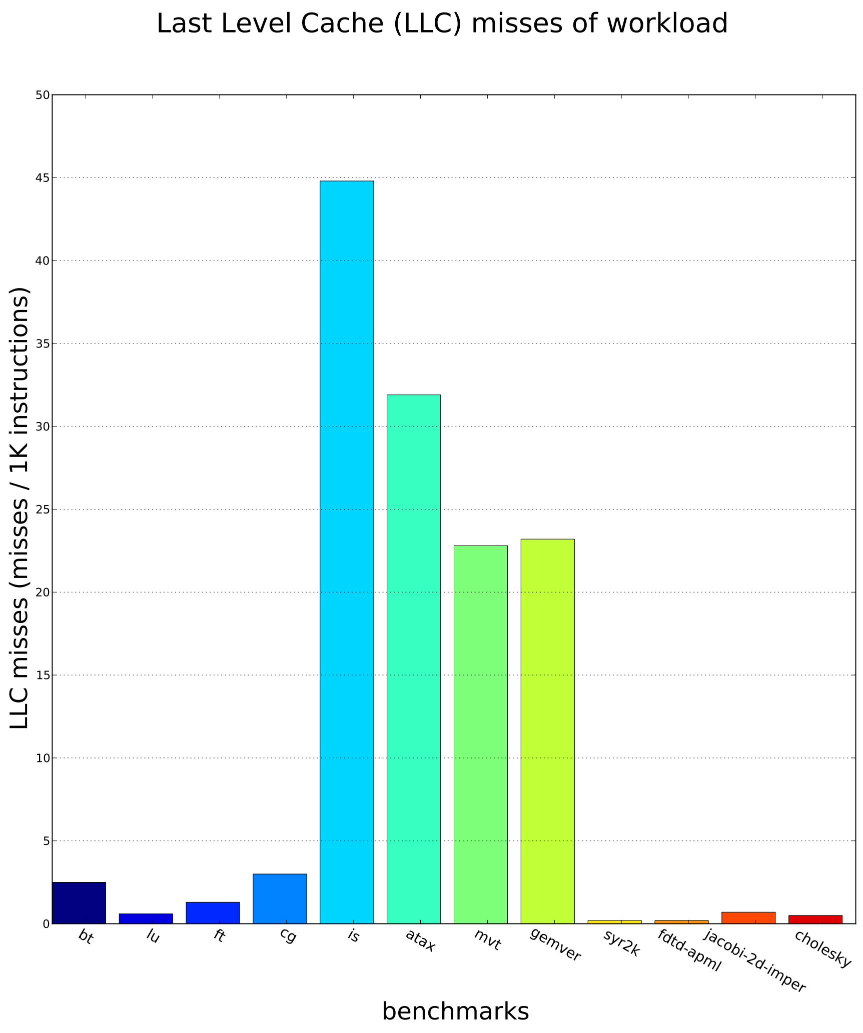


Figure 5.3: Miss rate of benchmarks of our workload

Benchmarks	speedup	LLC miss rate (misses/1K instructions)
IS	1.4	44.8
gemver	3.3	33.5
atax	3.1	30.6
mvt	3.0	20.8
CG	5.6	3.0
BT	3.26	2.5
FT	3.8	1.3
jacobi-2d-imper	7.0	0.8
LU	7.00	0.6
cholesky	10.1	0.5
fdtd-apml	4.0	0.3
syr2k	10.0	0.2

Table 5.3: Applications sorted by LLC misses

Actually, the LLC misses of an application significantly determine the way it scales. As we can see in Table 5.3, if we sort the list of the applications of the workload based on which scales better and then sort it based on their LLC misses we will take two lists which are very similar, indicating the significance of LLC misses to performance.

5.3 Evaluation of scheduling policies

In order to be able to evaluate the implemented scheduling policies, we defined some metrics to quantify their efficiency. We measure throughput, which is our main objective, as the overall execution time of the workload, which is the time until the last program of the workload is finished. Execution time of a benchmark i can be decomposed in three parts:

$$t(i) = t_{alone}(i) + t_{waiting}(i) + t_{interference}(i) \tag{5.1}$$

A scheduler implementation tries to minimize $t_{waiting}$, $t_{interference}$ or both.

Another desired feature of the scheduling policies we implement is fairness. We want the schedulers to treat all applications uniformly, meaning that we want to improve or deteriorate their throughput at a nearly equal factor. To derive this metric we measure the ratio of the standalone execution time of an application (t_{alone}) divided by its execution time when executed with a scheduler, with the same thread count. The standalone execution of an application provides its optimal throughput since there is no interference from other applications or the scheduler and the only restrictions are posed by the platform on which it is executed. Furthermore, a strong requirement from schedulers designed for desktops is to provide responsiveness for the system. We measure the

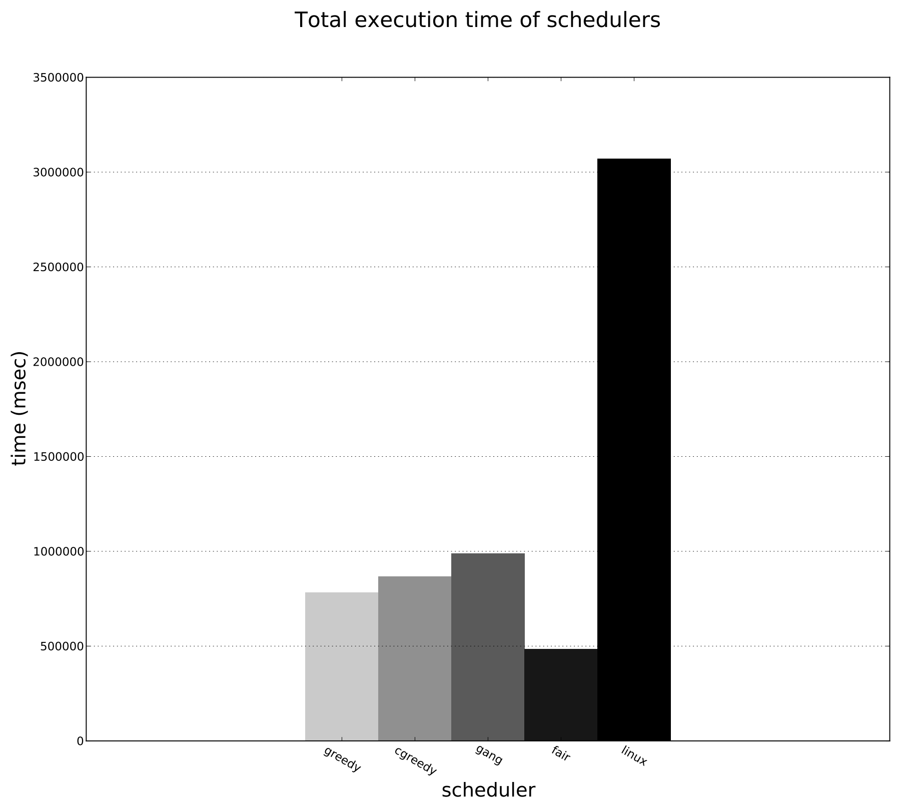


Figure 5.4: Total execution time of the workload with each scheduler

responsiveness achieved by a scheduler as the maximum average waiting time of the applications of the workload. We also measure the utilization of the available cpus achieved by each scheduler to use it as a tool in our effort to interpret the experimental results.

Figure 5.4 depicts the execution time of our workload when executing with each one of the four implemented scheduling algorithms, as well as when no scheduling policy is implemented and all decisions are left to the Linux scheduler. It is obvious that the Linux scheduler fails to handle a workload with multiple multithreaded applications efficiently. Even the GANG policy, which is the worst performing amongst our implemented schedulers, outperforms the Linux scheduler by a factor of three. We achieve a six-fold improvement of throughput with MINWAIT scheduler. We observe that CGREEDY scheduler does not outperform GREEDY scheduler even though it enforces a contention aware policy. This may be due to the fact that CGREEDY creates gangs with two programs at maximum, while there is not such a restriction in GREEDY.

MINWAIT scheduler does not use rounds. It chooses which programs it will execute, creating one gang every time-quantum. The more an application is left waiting, the more its priority is increasing, eliminating the possibility of starvation phenomena. It is possible for an application to be selected to execute in subsequent time-quanta if

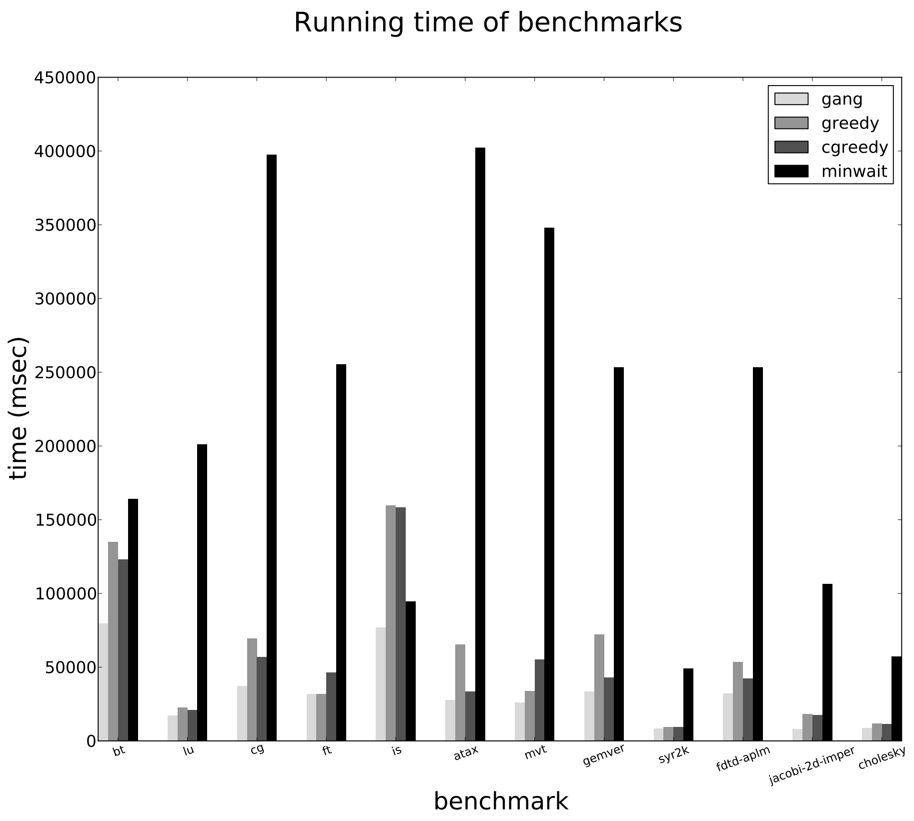


Figure 5.5: Running time of applications

there are not other applications with higher priority that can be scheduled (because the cores they request are not available). On one hand, this lowers the average waiting time of applications, as can be seen from Figure 5.7, and on the other, it leads to an interesting side-effect, applications requesting small thread-counts usually fit in the gang being created and are scheduled more often than others. This results in the scheduler not being fair as can be seen in Figure 5.9, since it favours applications with lower thread-counts. However, favouring these applications results in, overally, better performance. This observation repeats in Figure 5.6 from which we can see that BT and IS which execute with four and two threads, respectively, benefit greatly when executing with MINWAIT scheduler, as well as in Figure 5.8, where we can see that BT and IS present significantly lower average waiting time. Some of the benchmarks e.g. atax, mvt are significantly slowed-down, however, most of the applications seem to retain their throughput or even improve it, compared to GREEDY scheduler.

The way MINWAIT creates the gang to execute in every time quantum achieves high cpu usage (Figure 5.10) and as we saw reduces significantly the waiting of applications. However, at the same time it creates gangs that consist of many applications (Figure 5.12β) which results in high contention for shared resources among bench-

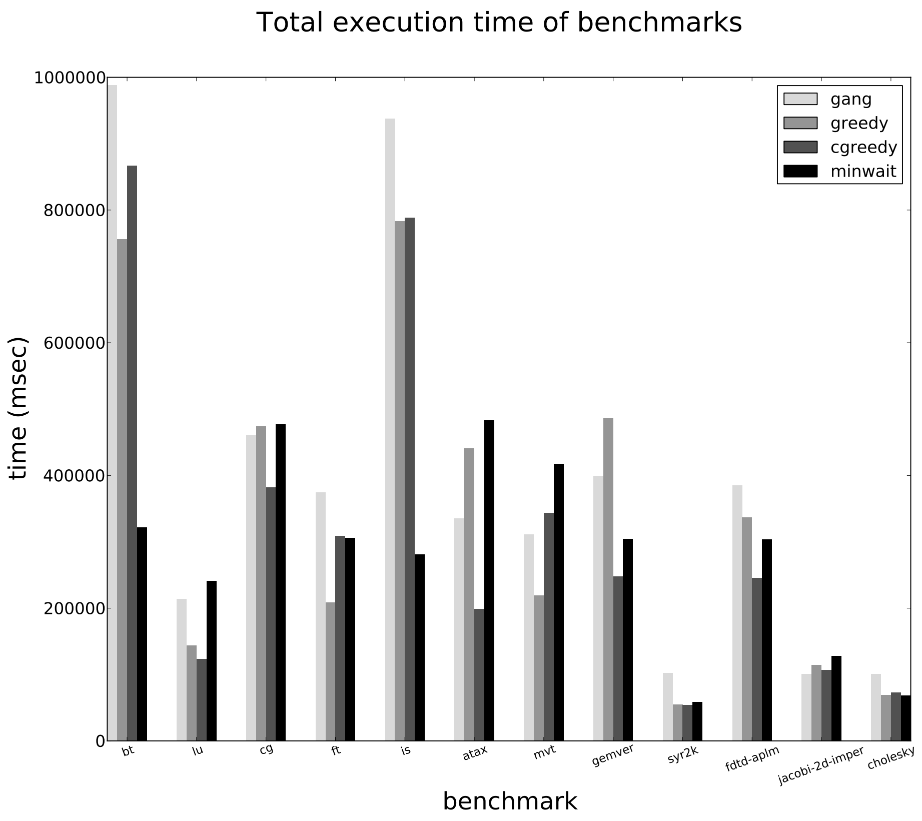


Figure 5.6: Total execution time of applications

marks. As a result, while MINWAIT reduces dramatically the waiting time of applications it increases its running time, which is the total time an application was THAWED ($t_{alone} + t_{interference}$). In Figure 5.5 we can see that the MINWAIT can increase the running time of an application up to 14 times (e.g. atax) comparing to GANG scheduler (which achieves the lowest running time for the applications). The reason that MINWAIT achieves the better total execution time is the significat reduction of the waiting time for every application of the workload. GREEDY scheduler creates gangs using a greedy bin packing algorithm in an effort to achieve maximum utilization of the cpus. GREEDY does not pack as many applications in a gang as MINWAIT does. This results in higher waiting time for the applications of the workload but, GREEDY does not create as much contention as MINWAIT does. This is reflected in the running time of applications in Figure 5.5. GREEDY achieves much better execution time for the applications.

CGREEDY similarly with GREEDY scheduler does not cause high contention among applications. Moreover, it creates gangs using information about the memory profile of applications and creates gangs with applications that have complementary cache usage characteristics. This results in CGREEDY achieving better running time for applications comparing to GREEDY. In Figure 5.5 we can see that CGREEDY improves running time

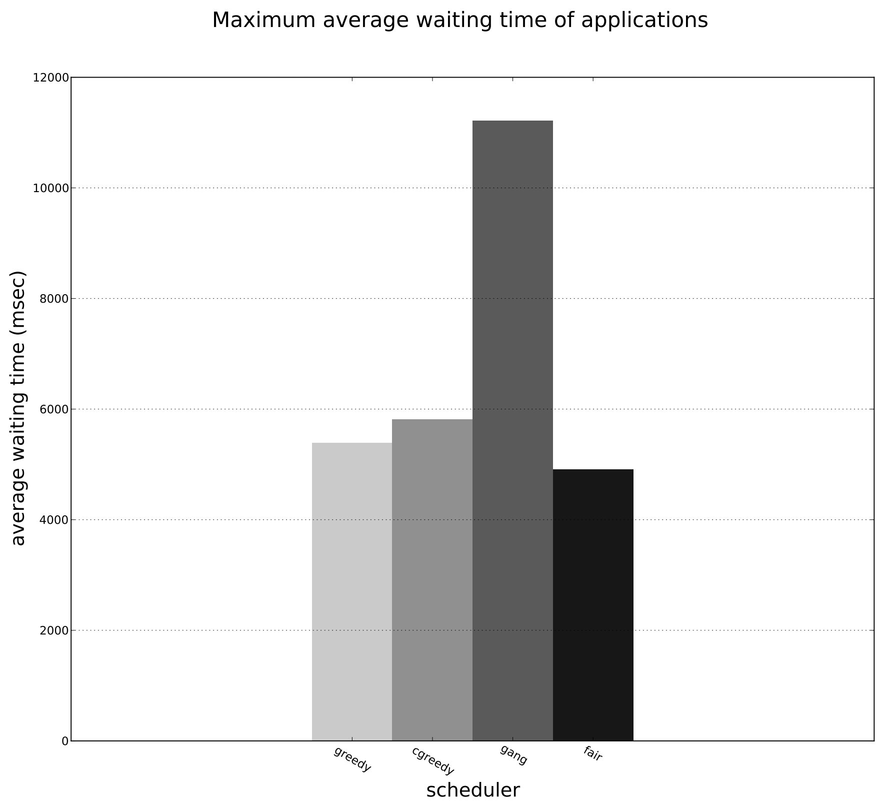


Figure 5.7: Maximum average waiting time of schedulers

of almost every application. For some applications this improvement is up to 96% for atax and 68% for gemver.

GANG scheduler creates gangs with one application per gang, which results in rounds being long (many gangs per round), and thus programs waiting a long time until their turn to be executed comes (Figure 5.8. On the other hand, the fact GANG does not use space-sharing results in contention for shared resoures such as cache memory and memory bus bandwidth being dramatically reduced. This is depicted in Figure 5.5 in which we can see that with GANG every program of the workload achieves the lowest running time.

GANG scheduler provides the better isolated execution environment for the applications. As a result this scheduler provides higher fairness for the applications (see Figure 5.9). GANG scheduler treats applications uniformly. Since all applications execute alone at their time-quantum $t_{interference}$ is minized, thus any degradation in performance (due to an application being executed with the rest of the workload) is equal among applications.

The rest of the schedulers, enforce space-sharing schemes, which in conjunction with the varying contention that occurs among applications based on the schedulers decisions

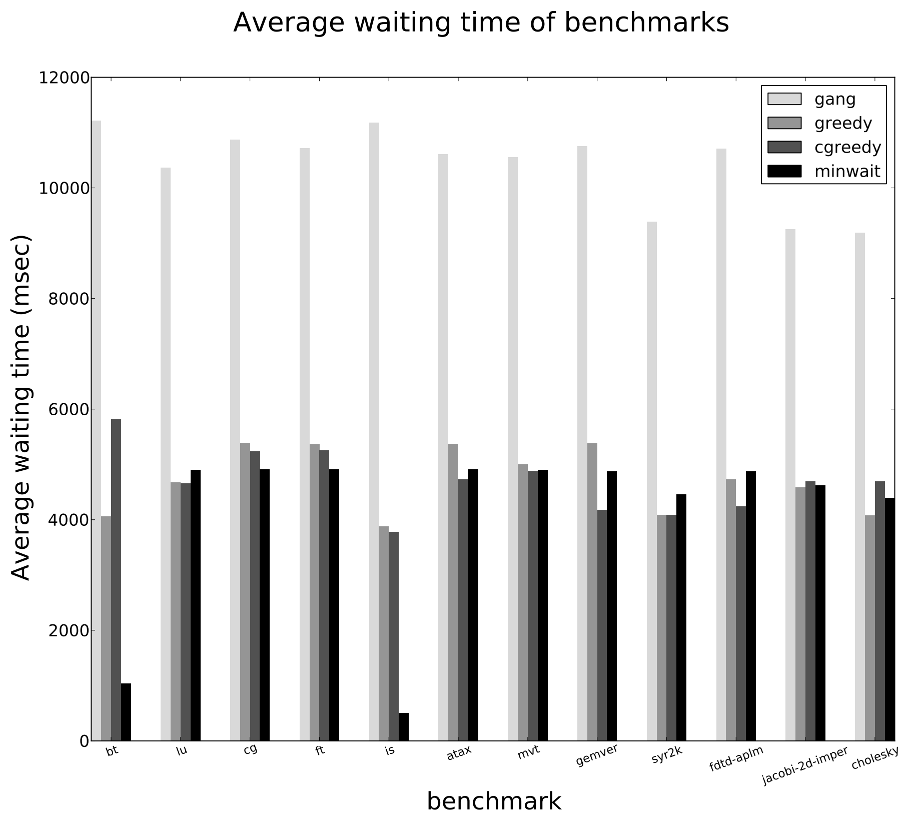


Figure 5.8: Average waiting of benchmarks of every scheduler

results in varying degradation of performance among them. As a result, fairness of schedulers that enforce space-share is expected to be lower than fairness of GANG scheduler (since fairness is defined as the standard deviation of the fairness of applications lower values indicate higher fairness).

CGREEDY scheduler creates gangs with maximum two applications, thus limited contention is inserted in comparison with GREEDY scheduler. At the same time, CGREEDY is designed to be contention-aware while GREEDY scheduler is not which also leads to lower degradation in applications throughput. This is why GREEDY scheduler presents lower fairness than cgreedy.

MINWAIT scheduler presents the lowest fairness of all. This is a result of the side effect of the scheduling algorithm which favours applications that request small thread-counts. Throughput of those applications is affected less and this leads in greatly varying fairness for the applications and consequently very low fairness for MINWAIT scheduler. In general, there is a tradeoff between GANG and schedulers that enforce space-sharing. GANG provides an isolated execution environment, reducing to minimum the level of contention among applications, leaving, however, many of the system resources unutilized. On the other hand, the other schedulers that use space-sharing endeavor to

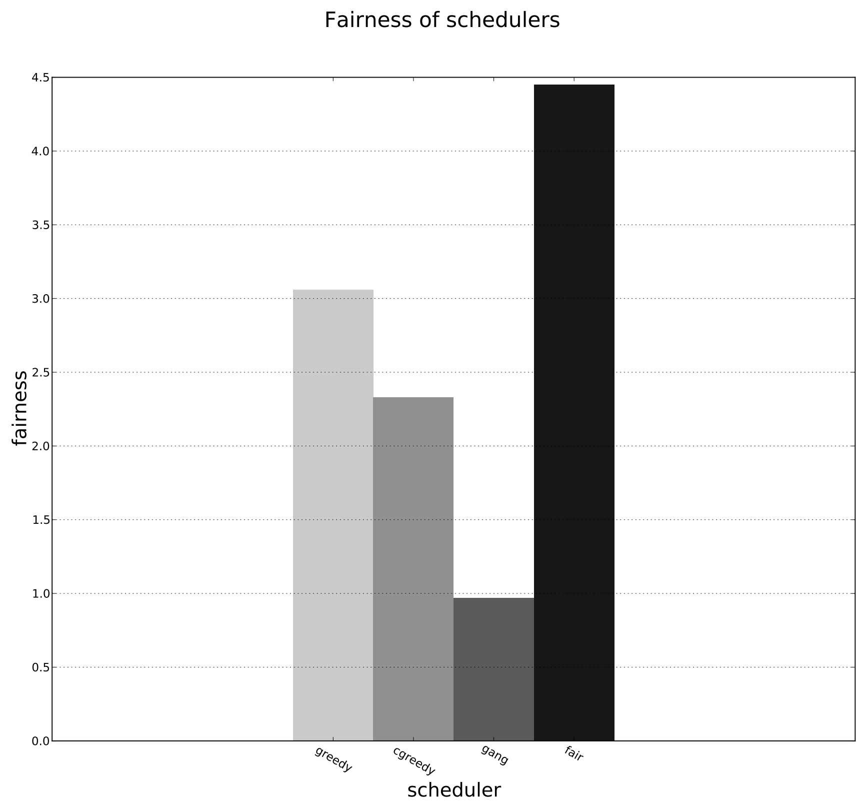


Figure 5.9: Fairness achieved by schedulers

maximize the utilization of system resources. However, when executing applications at the same time-quantum they suffer from contention. It seems that exploiting system resources at a maximum level, outweighs the contention introduced due to space-sharing, because the waiting time of applications is minimized. The fact that CGREEDY which employs space-sharing but not as aggressively as GREEDY and MINWAIT, decreases the running time of applications and in some cases (CG, atax, fdtd-apml) as we can see in Figure 5.6 decreases the total execution time, implies that exploiting system resources as much as possible is not panacea, and contention-aware policies can be employed to further improve throughput.

We created snapshots of the workload execution with the four schedulers, in order to depict the way that schedulers create gangs and in which way this affects their performance. The horizontal axis represents the time-quantum of the execution and the vertical axis the cpus of the system.

The cpu usage achieved by each scheduler is related to the number of applications it manages to pack in each gang. GANG scheduler that creates gangs with only one application presents significantly low cpu usage (Figure 5.11α⊠). CGREEDY scheduler which creates gangs with up to two applications achieves higher cpu usage than GANG but

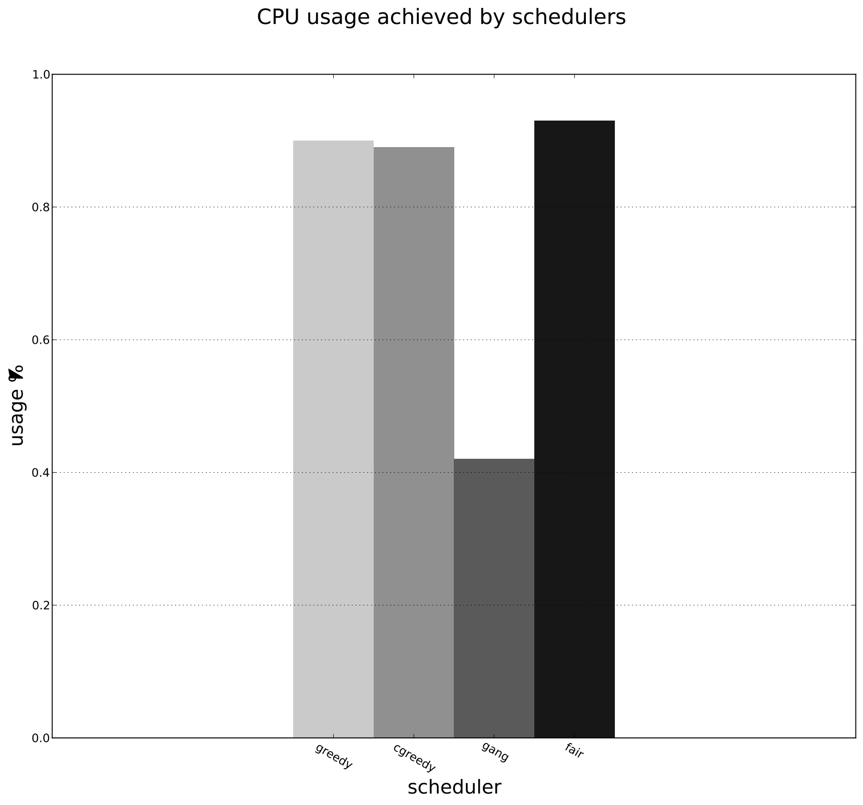
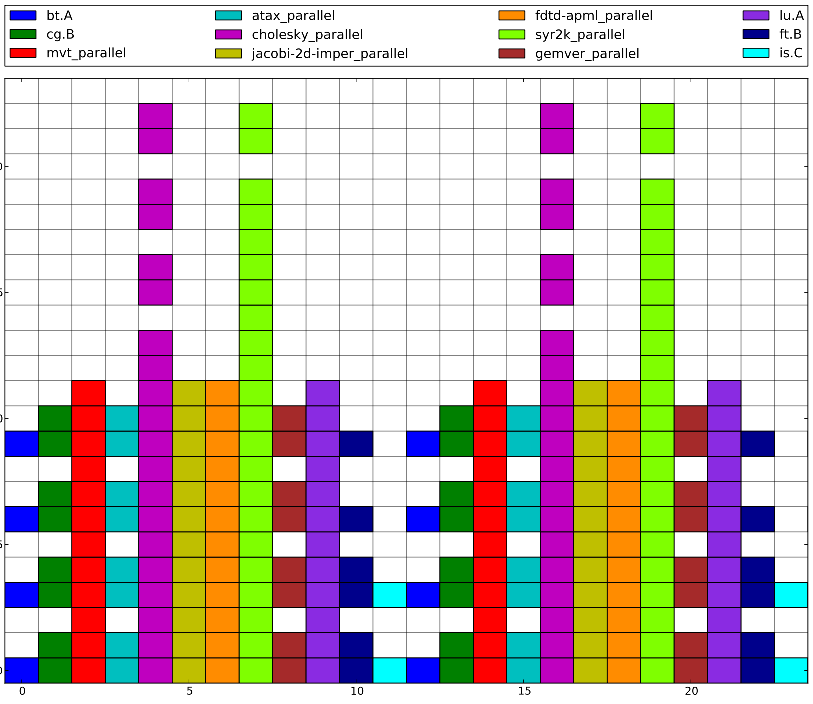
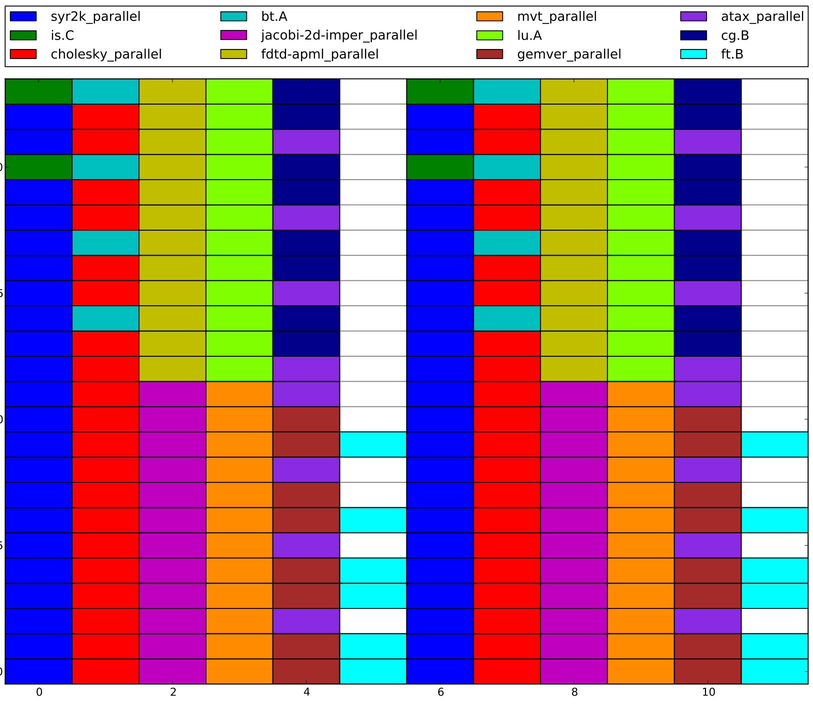


Figure 5.10: CPU usage of schedulers

lower than GREEDY. It creates gangs based on LLC misses and cpus needed from each application (Figure 5.12α). GREEDY which does not put restrictions on the number of applications per gang achieves higher utilization. Gangs created by GREEDY are fixed in every round (IS - syr2k, cholesky - BT, jacobi - fdtd-apml, mvt - LU, gemver - atax - CG, ft). MINWAIT scheduler achieves the highest cpu usage. This is a result of the gang to be executed is created in every time-quantum as we can see in Figure 5.12β. Whenever the gang is created the scheduling algorithm chooses the programs that were waiting for a long time, but also traverses the whole list of programs trying to find any application fitting in the remaining cores even if it was recently scheduled to execute. On the other hand, the rest of schedulers that enforce space-sharing create gangs in the beginning of a round and as a result there are time-quanta that there are unutilized cores that could be used by other applications and yet they do not because those applications are scheduled to execute in a different time-quantum of this round. The cpu usage statistics of the algorithms are depicted, at Figure 5.10, which clearly describes the previous conclusions.

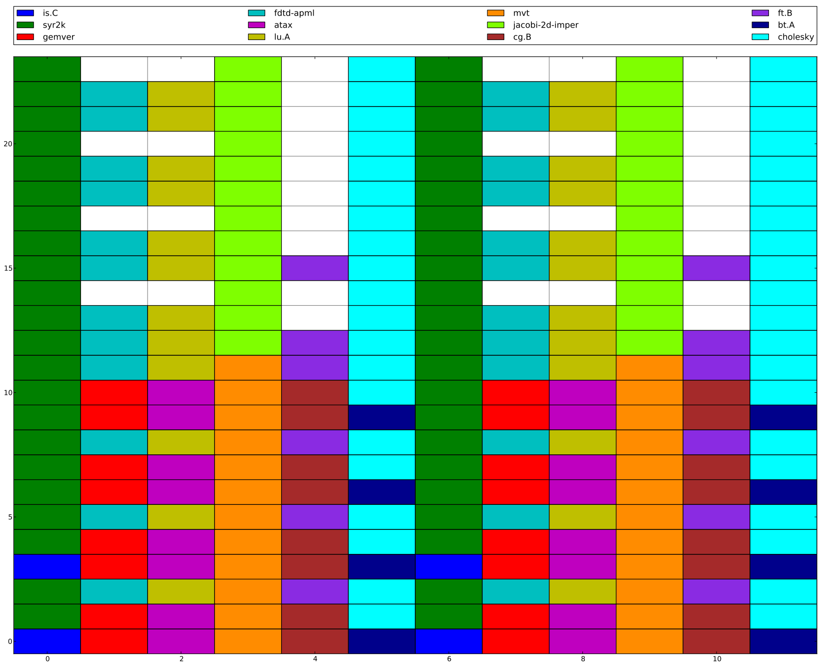


(α)

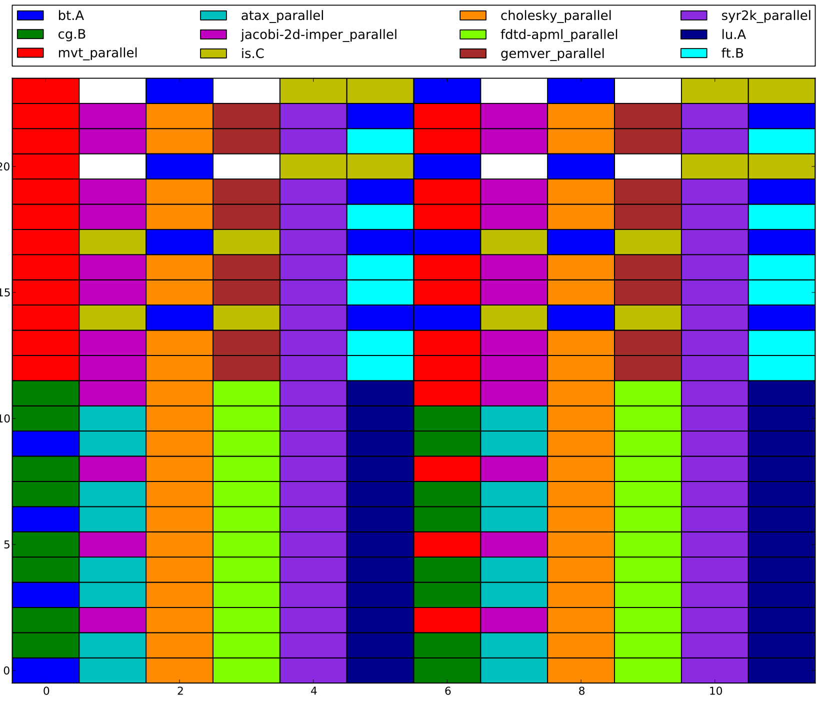


(β)

Figure 5.11: Snapshots of GANG (5.11α) and GREEDY (5.11β) execution



(α)



(β)

Figure 5.12: Snapshots of CGREEDY (5.12α) and MINWAIT (5.12β) execution

In order to better understand the way that the schedulers we implemented work, we used a subset of the benchmarks of the workload we used for the initial experiments to create two different workloads. One consists of multiple instances of cpu-intensive applications which produce few LLC misses and scale well. These are syr2k, jacobi-2d-imper, lu and cholesky. Similarly we created a workload that consist of is, mvt, atax and gemver which have many LLC misses and scale poorly. In both workloads we use three instances of every benchmark in order to have the same number of applications with the original workload.

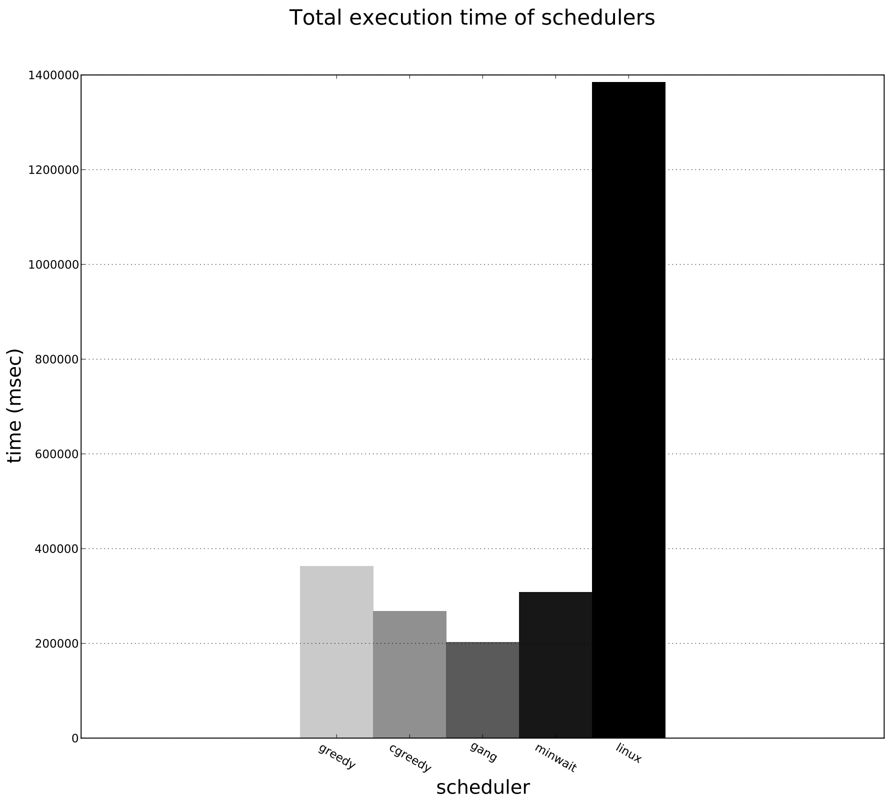


Figure 5.13: Total execution time of cpu-intensive workload

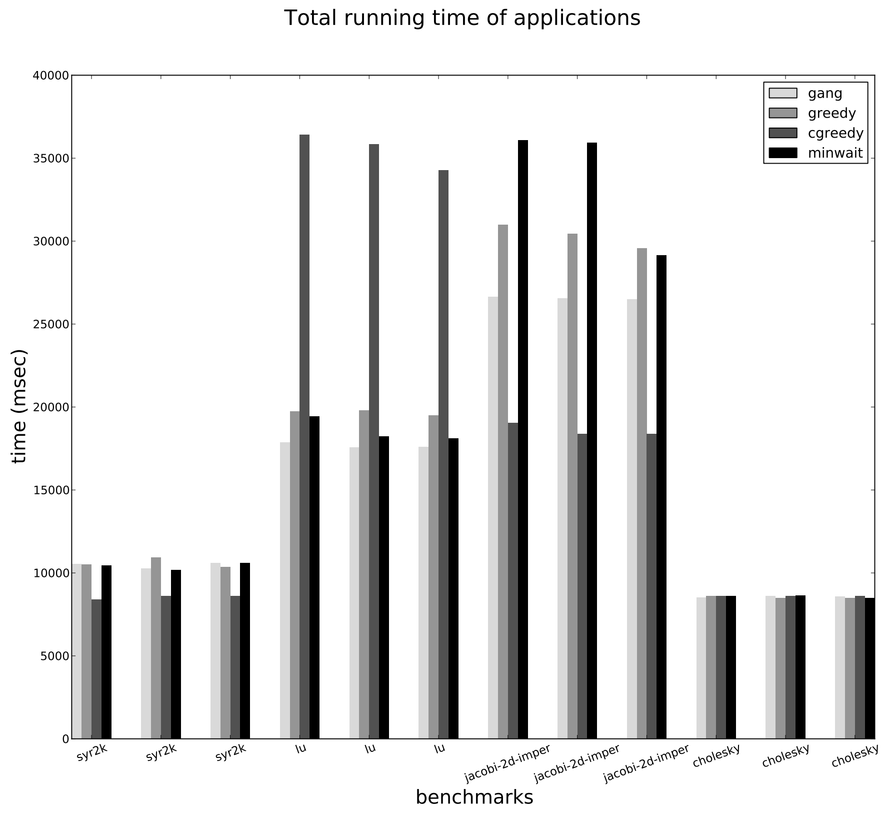


Figure 5.14: Total running time of applications

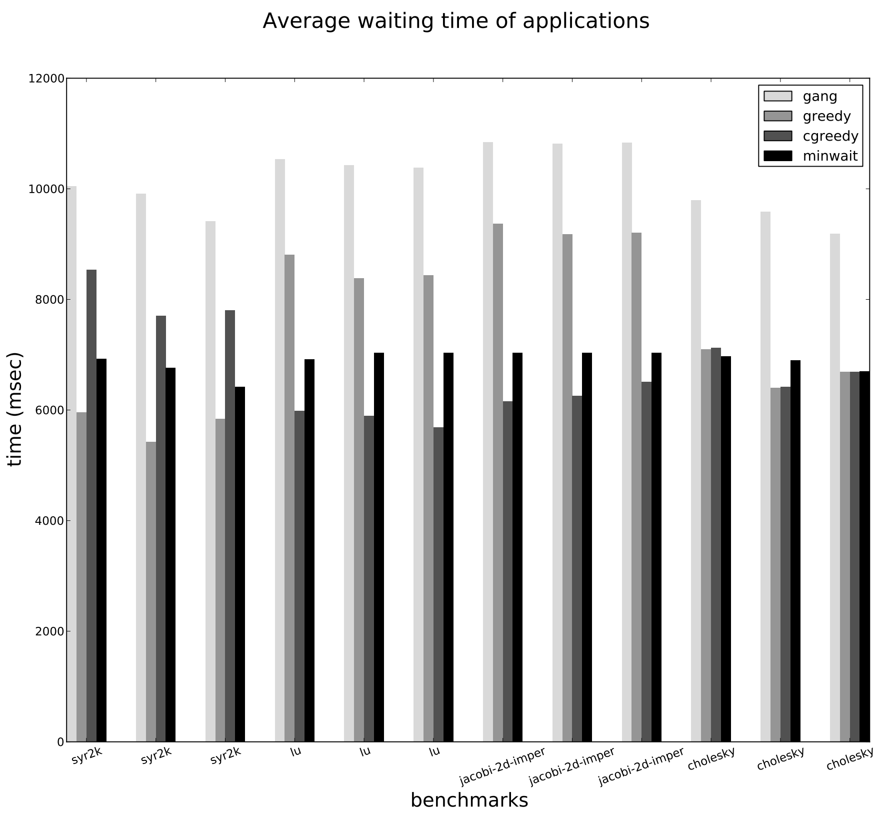


Figure 5.15: Average waiting time of applications of the cpu-intensive workload

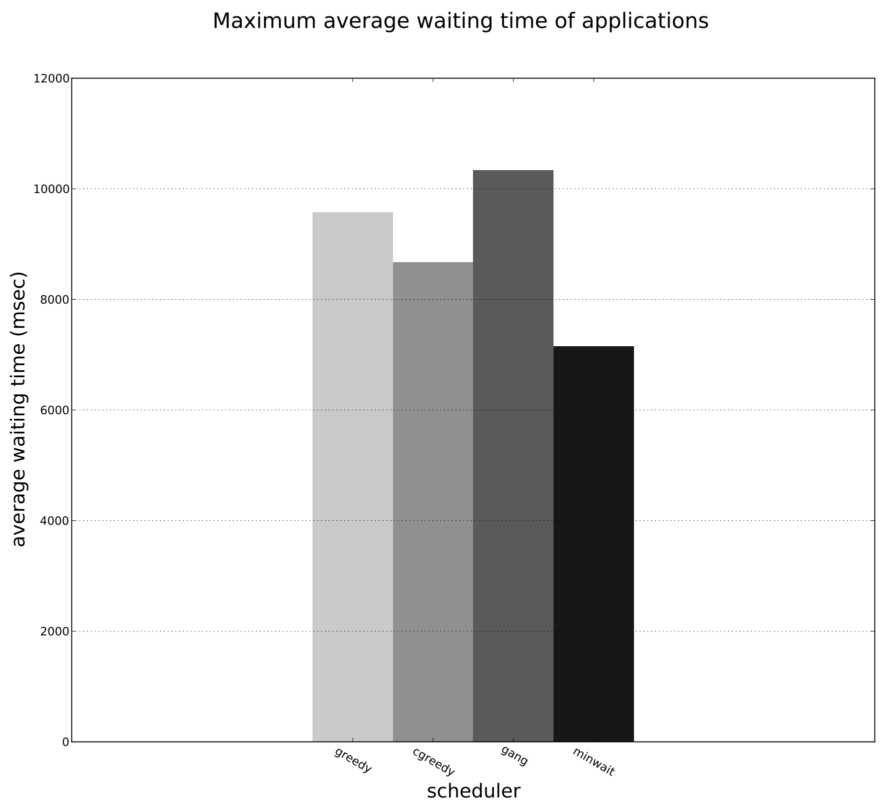


Figure 5.16: Maximum average waiting time of cpu-intensive workload

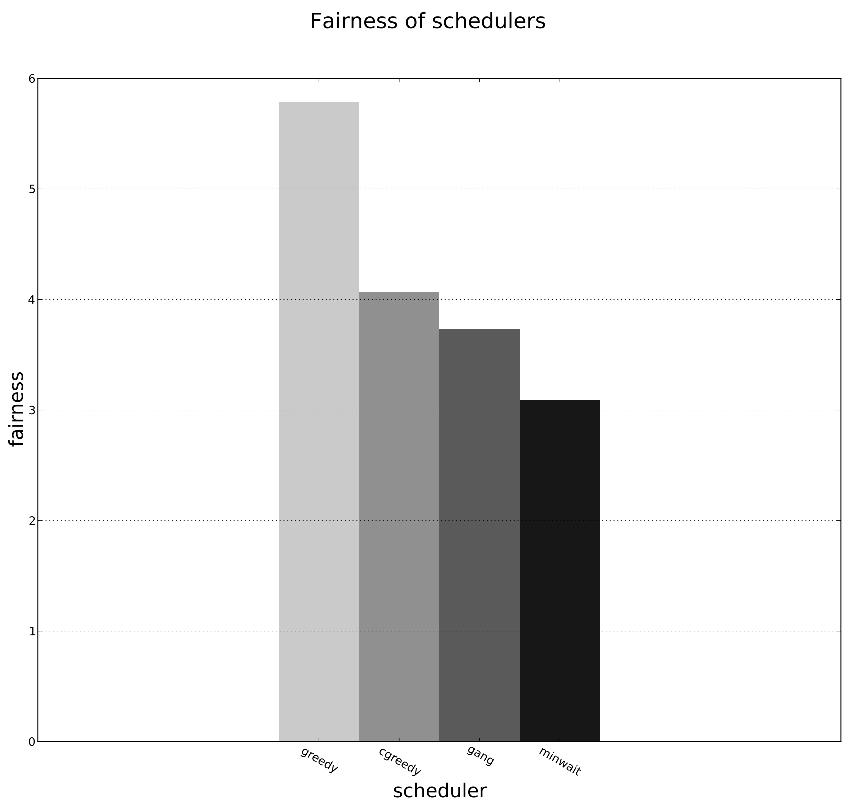
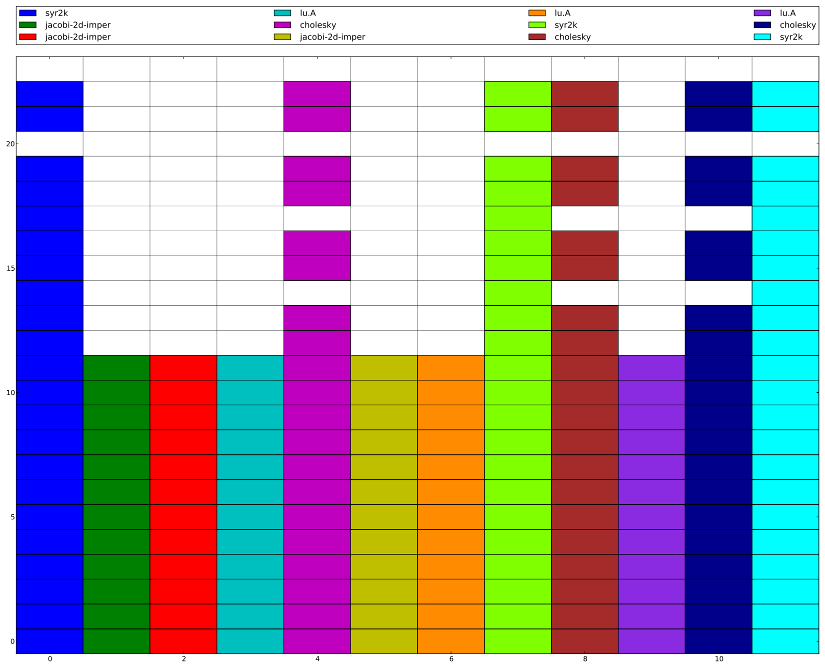
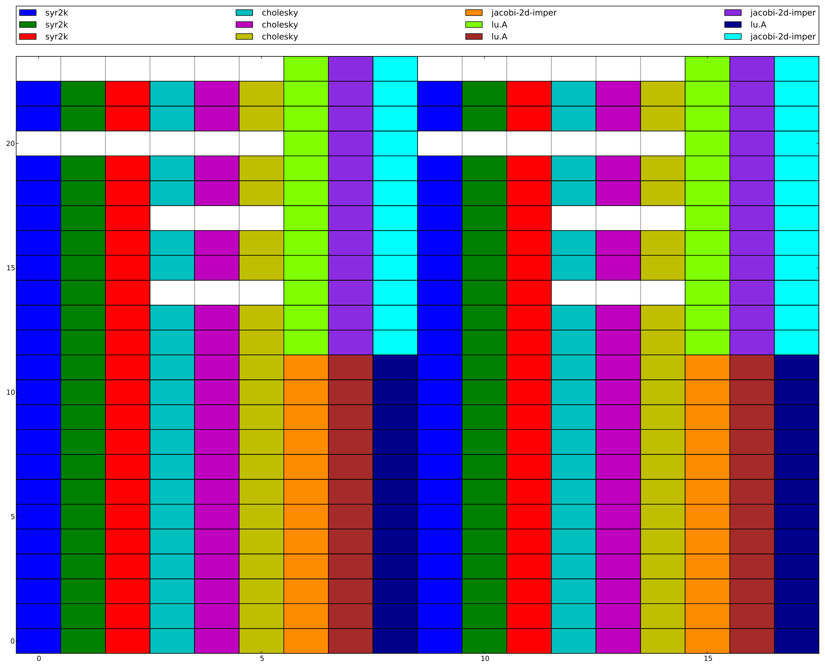


Figure 5.17: Fairness of schedulers with cpu-intensive workload

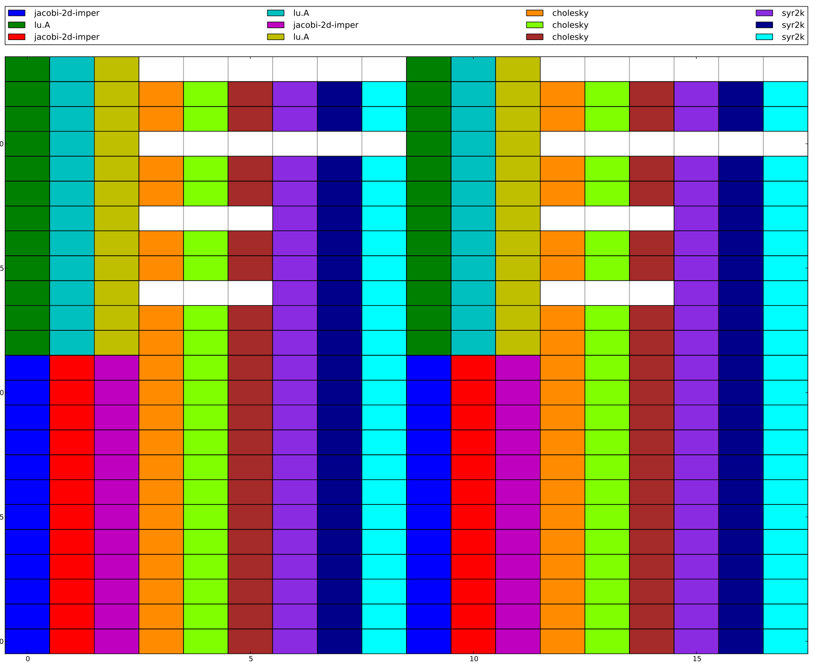


(α)

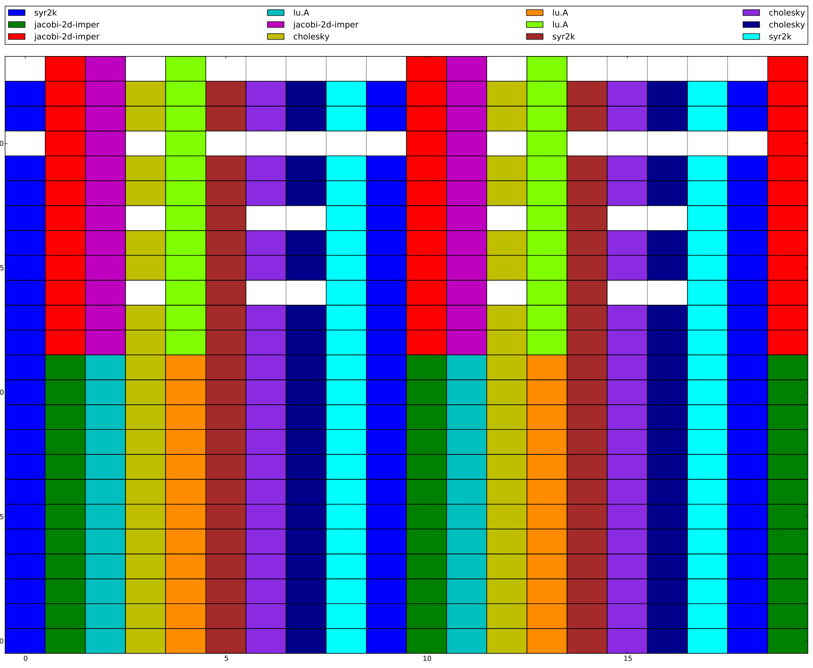


(β)

Figure 5.18: Snapshots of GANG (5.11α) and GREEDY (5.11β) for execution cpu-intensive workload



(α)



(β)

Figure 5.19: Snapshots of CGREEDY (5.12α) and MINWAIT (5.12β) for execution of cpu-intensive workload

Applications that consist the memory-bound workload do not scale well and as a result they use relatively low thread-counts. This, favours the schedulers that enforce space-sharing and packing as many applications as possible in a gang. The interesting result shown in Figure 5.20, is the good performance of the Linux scheduler. It is only outperformed by the MINWAIT scheduler. This is partially because of the fact that the total number of threads of this workload is significantly smaller than this of other schedulers. We have seen (Figure 3.2) that Linux does not scale well when the number of threads it handles is much larger than the available cores.

We can see that MINWAIT scheduler performs better than the Linux scheduler. MINWAIT leverages the fact that the number of threads of every application is small and achieves to make full utilization of the available cpus, similar to the utilization of the Linux scheduler. At the same time, MINWAIT uses gang scheduling which benefits the execution comparing to the thrashing of threads to different time-quanta that Linux scheduler causes.

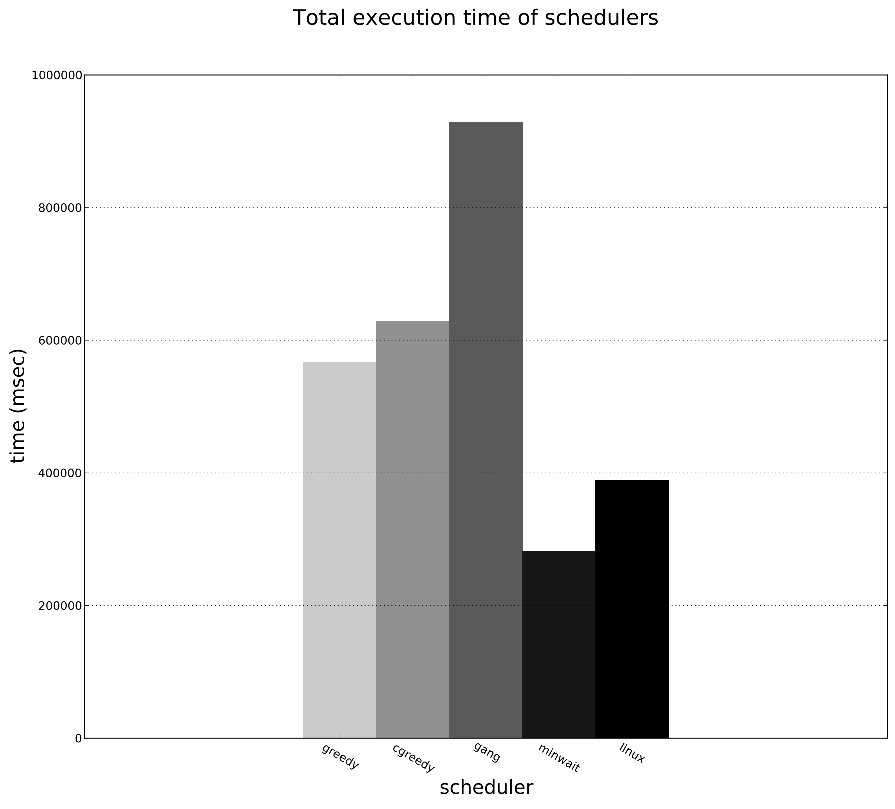


Figure 5.20: Total execution time of memory-bound workload

The benchmarks of this workload make heavy use of the memory subsystem. The poor scaling is inherent to their execution. Thus any contention aware scheduling policy is difficult to benefit their execution. This is can be seen in Figure 5.21 where CGREEDY scheduler fails to reduce significantly the running time of applications. Since, all applications encounter high LLC miss rates there are not good pairs to create gangs. GREEDY scheduler does not achieve to utilize the available cores as much, as MINWAIT does. This is reflected in the larger average waiting time (Figure 5.22), and this is why GREEDY scheduler fails to perform as well, as MINWAIT.

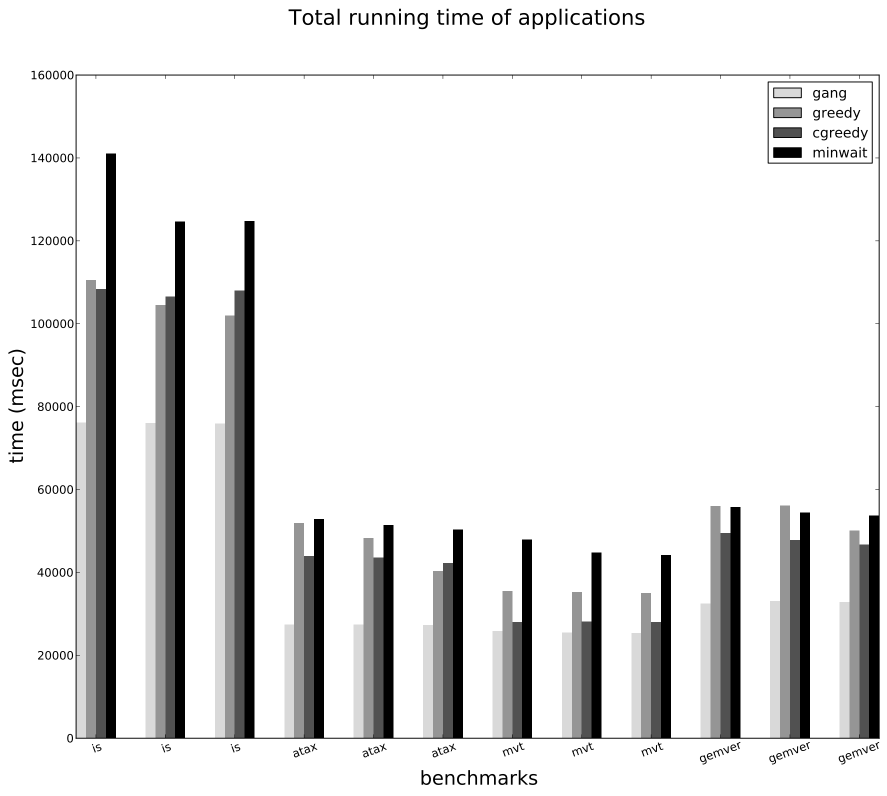


Figure 5.21: Total running time of applications

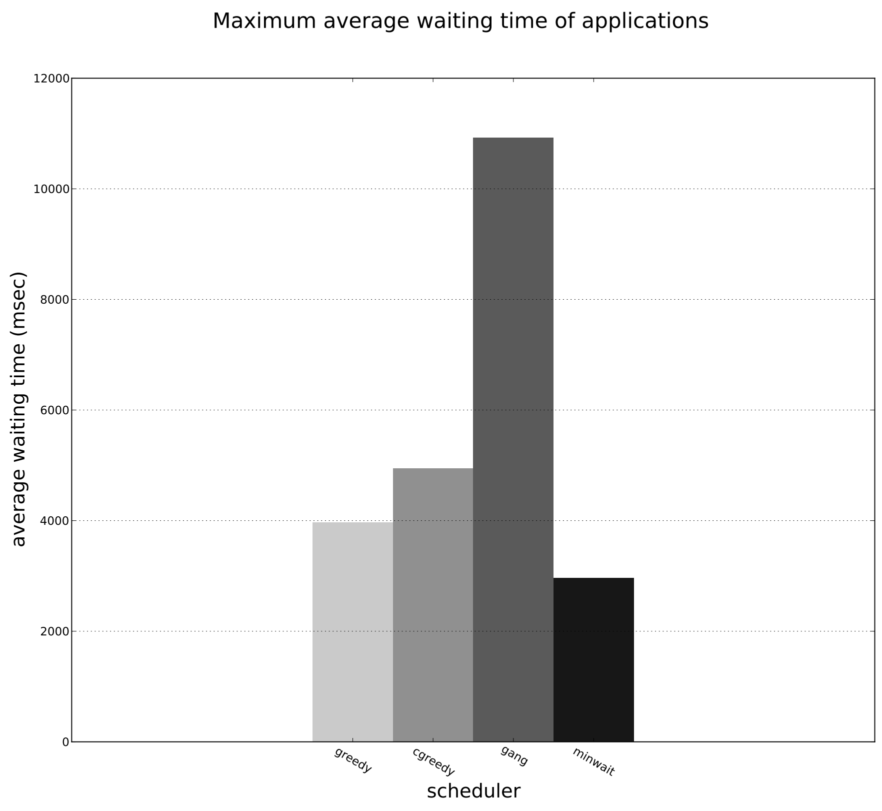
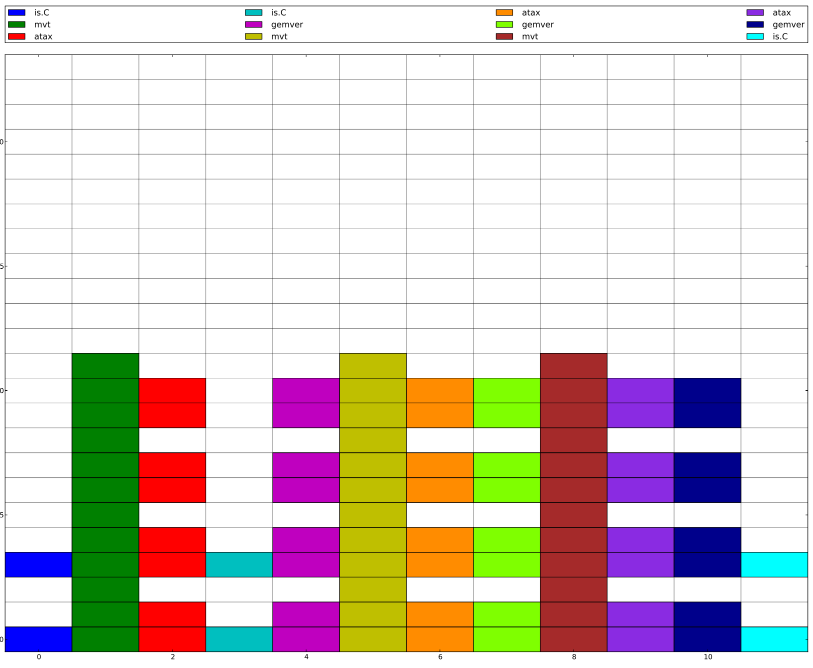
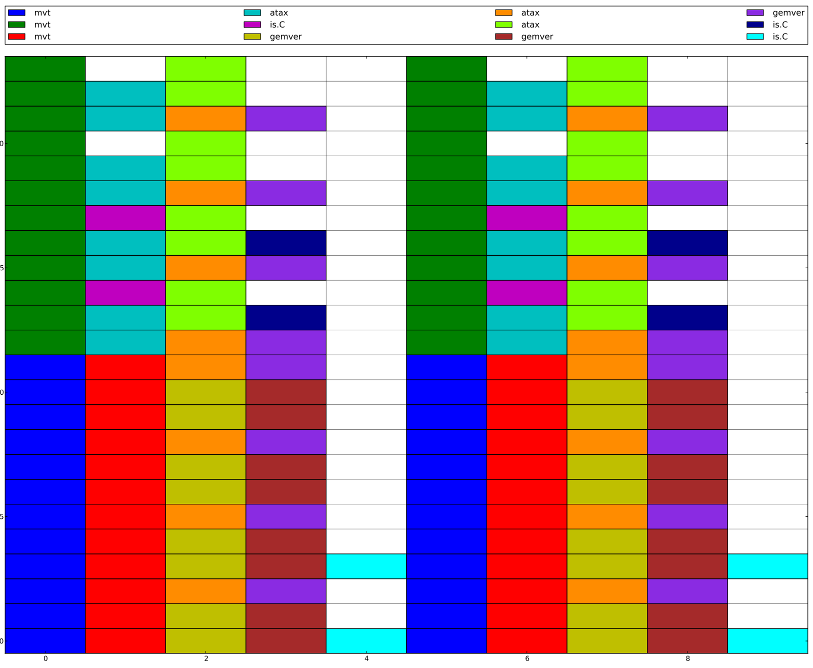


Figure 5.22: Maximum average waiting time of memory-bound workload

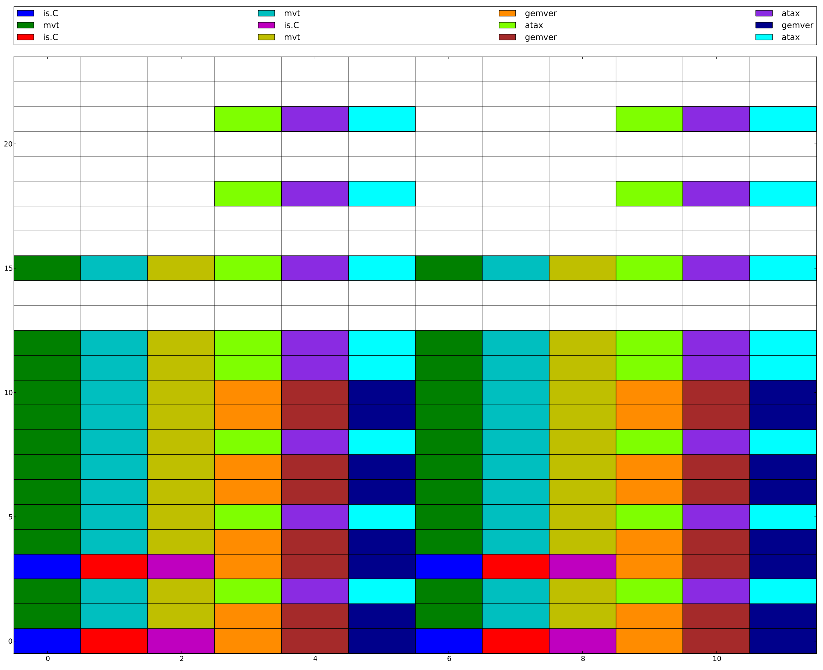


(α΄)

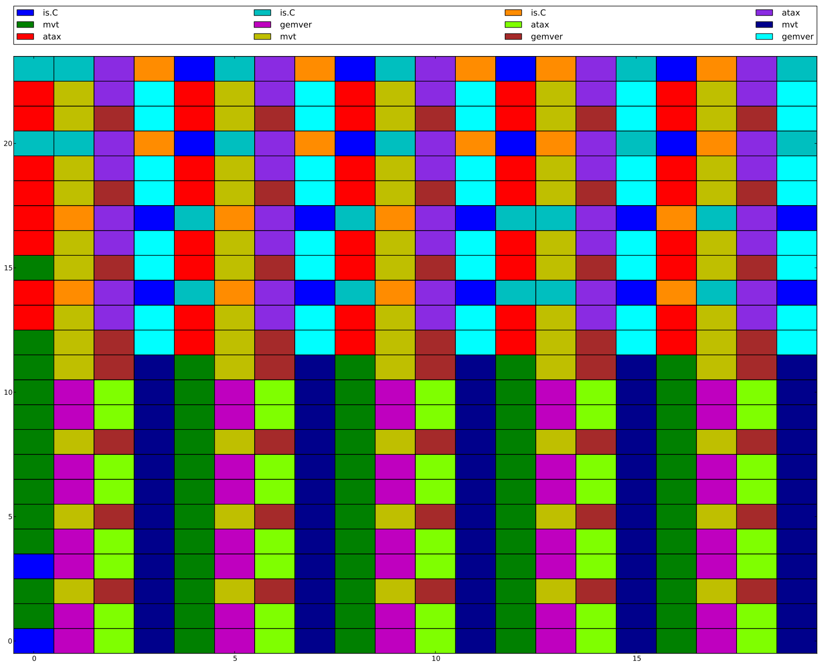


(β΄)

Figure 5.23: Snapshots of GANG (5.11α΄) and GREEDY (5.11β΄) for execution of memory-bound workload



(α)



(β)

Figure 5.24: Snapshots of CGREEDY (5.12α) and MINWAIT (5.12β) for execution of memory-bound workload

5.4 Placement significance in scheduling

Our preliminary experiments demonstrate clearly that the placement of threads of multithreaded applications is a factor that significantly affects the execution characteristics. We defined two different placements packed and spreaded which place threads on the-same-package-first and on different-packages-first respectively and lead to different execution profiles. Packed placement provide isolation for applications, that is they are affected less from executing with other applications concurrently. On the other hand, spreaded placement provides applications with bigger effective LLC and memory bandwidth, which leads to higher throughput for every application and the workload overall.

We used those placements to experiment when executing the whole workload using a scheduler. We conducted two experiments for every scheduler. One is assigning cores to applications in packed manner and the other in spreaded manner. Figure 5.25 depicts that spreaded placement provides higher throughput for the whole workload for all schedulers. Throughput of gang scheduler doubles when executing with spreaded placement comparing to packed placement.

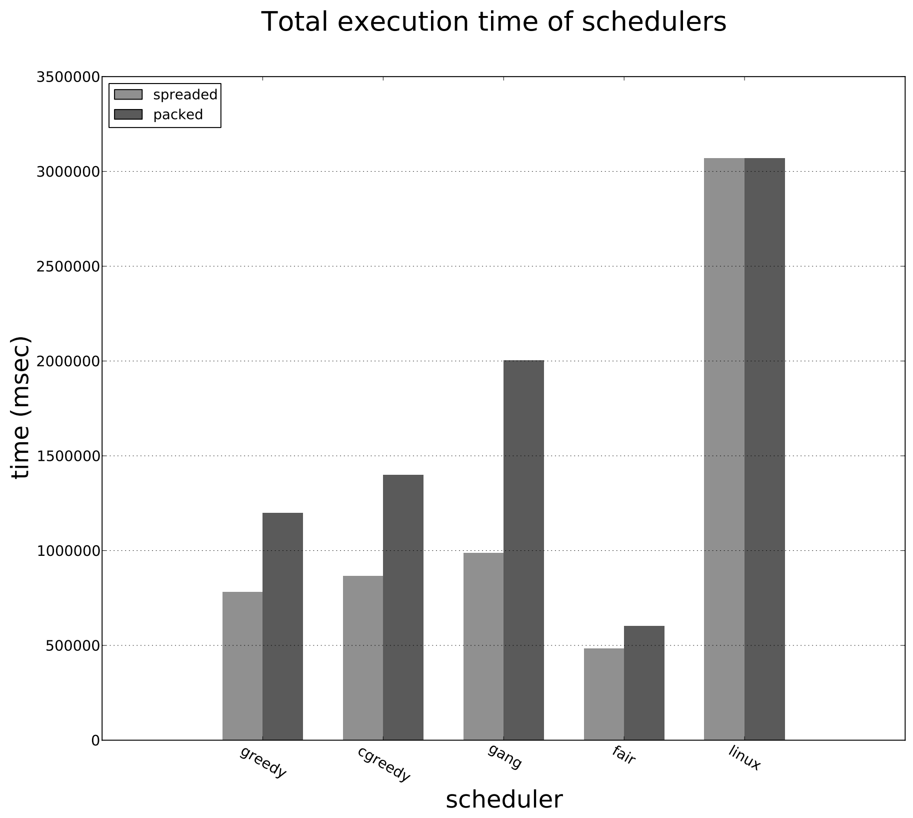


Figure 5.25: Total execution time of schedulers for packed and spreaded placement

The same results were taken for the other workloads. In Figure 5.26 we observe the results for the cpu-intensive workload. The difference of throughput between the two placements is minimal for this workload. Since, this workload is consisted of cpu-intensive applications which do not make heavy use of the LLC or the memory bus bandwdith, larger effective LLC or bandwidth cannot benefit them, thus spreaded placement does not improve their throughput.

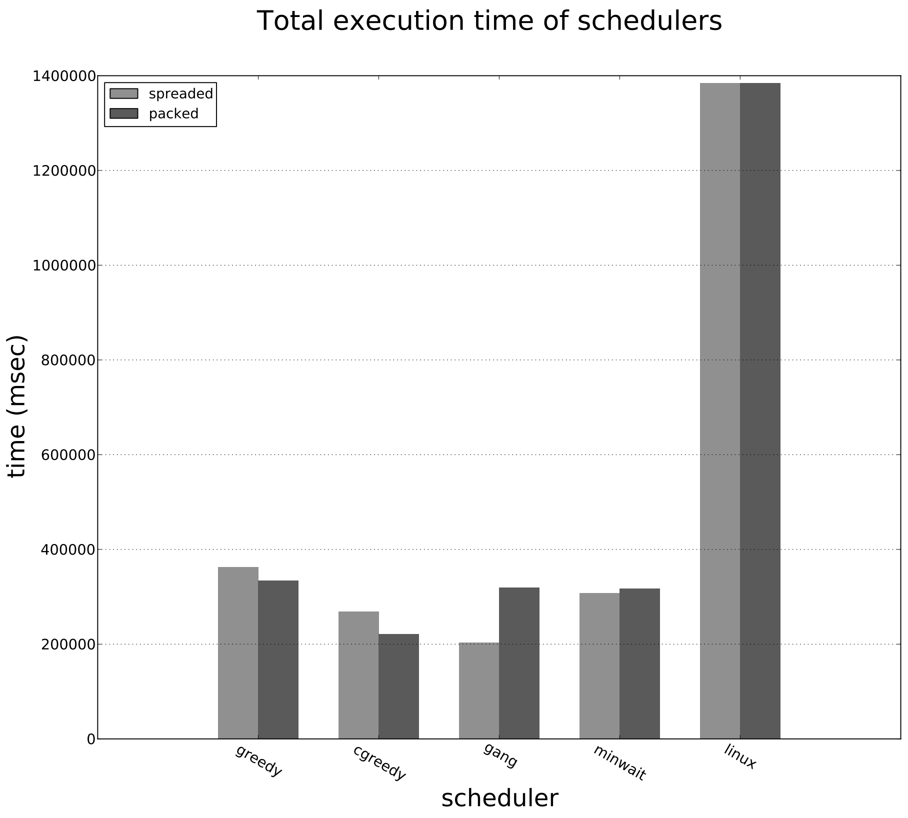


Figure 5.26: Total execution time of schedulers for packed and spreaded placement - cpu-intensive workload

Figure 5.27 shows results corresponding to the memory-bound workload. For this workload, we see that spreaded placement is important for throughput. The improvement is not as big as for the original workload. This is due to the fact, that the usage of the memory subsystem from the applications of the workload is very intensive and the margin for improvement is little.

At the same time, observing Figures 5.28, 5.29, 5.30 and 5.31 packed, placement provides higher isolation for most of the benchmarks of our workload with all schedulers. Fairness is defined as the total execution time of the application normalized to the standalone execution time. Both times are measured with the same placement, so even if packed placement leads to lower throughput when executing standalone, execution with

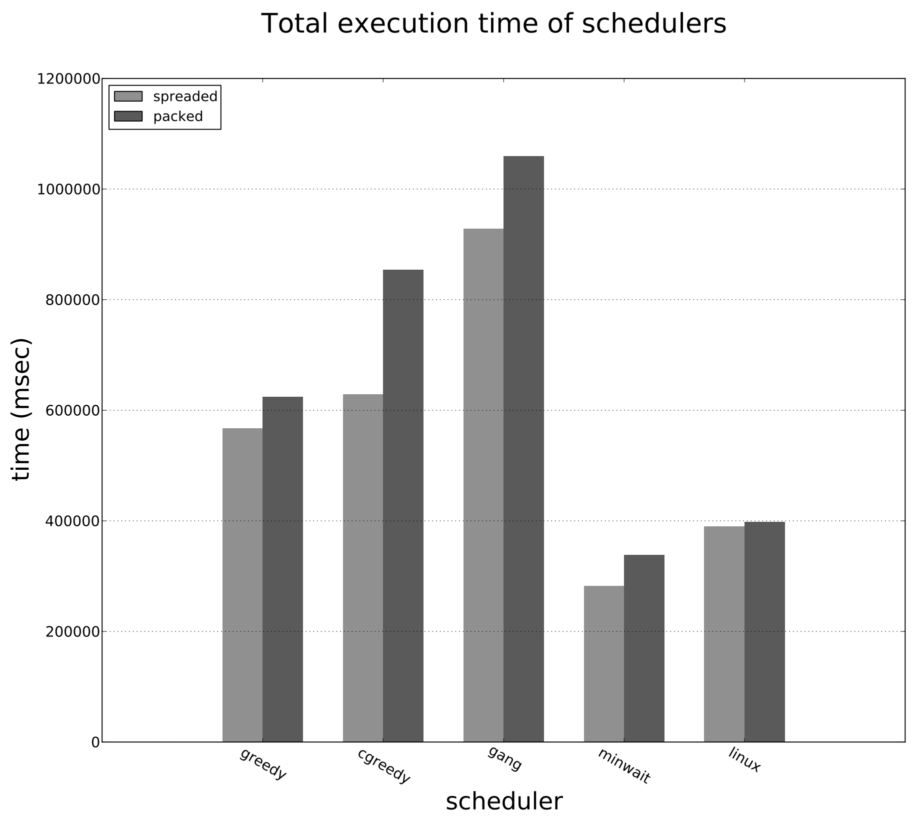


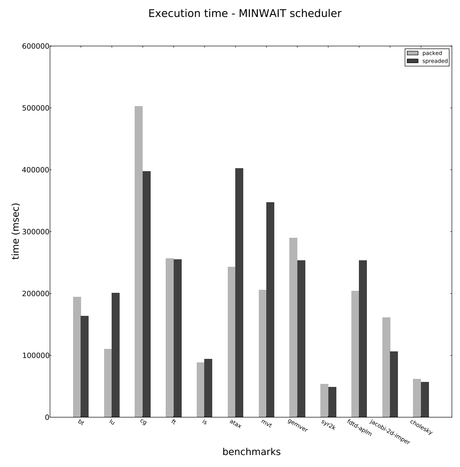
Figure 5.27: Total execution time of schedulers for packed and spreaded placement - memory-bounded workload

a scheduler degrades the performance of an application less than spreaded placement.

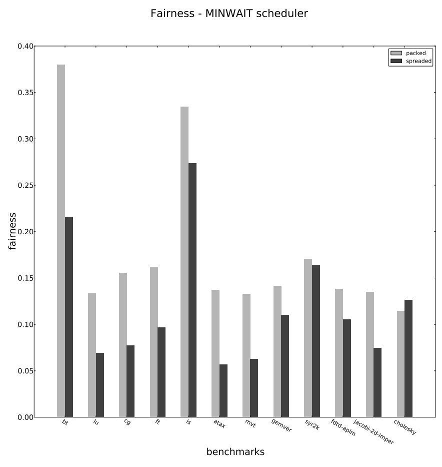
MINWAIT scheduler, when executed with spreaded placement, provides higher throughput for most of the benchmarks, comparing to packed placement, except from the applications with high miss rates such as is, atax, mvt and gemver. Packed placement on the other side provides isolation for every benchmark of the workload. Especially atax and mvt, which suffer from LLC misses, benefit from packed placement.

MINWAIT and CGREEDY, provide higher isolation that GREEDY, with packed placement. This configuration could be used in cases where we want to enforce QoS policies. When applications are scheduled with packed affinity are not significantly affected from applications with which they are co-scheduled. In platforms organized in NUMA nodes, where each physical package has its own memory bus, applications packed into different physical packages, will not contend neither for LLC, or memory bus bandwidth. In those systems, packed placement will provide isolation and moreover will not suffer from limited resources, and as a result will not suffer from reduced throughput, as in our platform.

GANG scheduler does not present difference between packed and spreaded placement, as far as fairness is concerned, which is expected. Since there is no space-sharing, there is no contention among applications, so packed placement cannot contribute in applications' isolation.

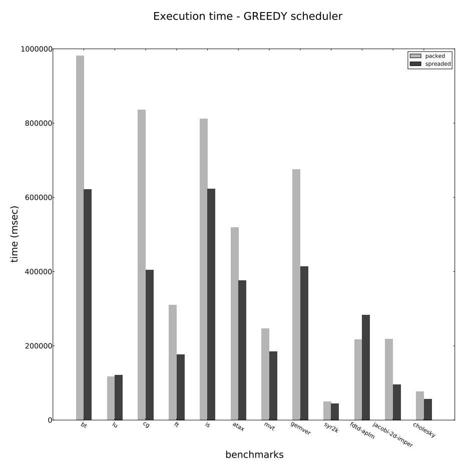


(α)

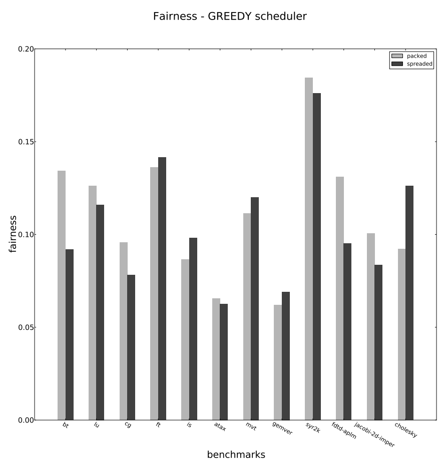


(β)

Figure 5.28: Total execution time and fairness of applications - MINWAIT scheduler

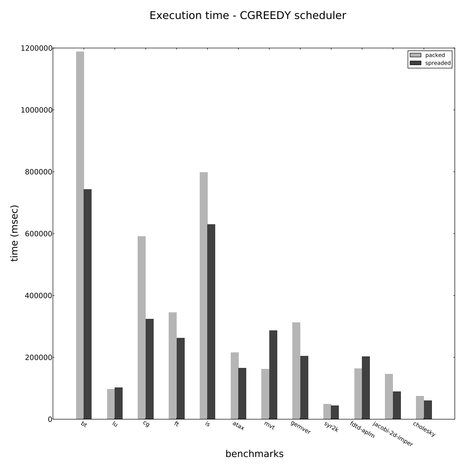


(α)

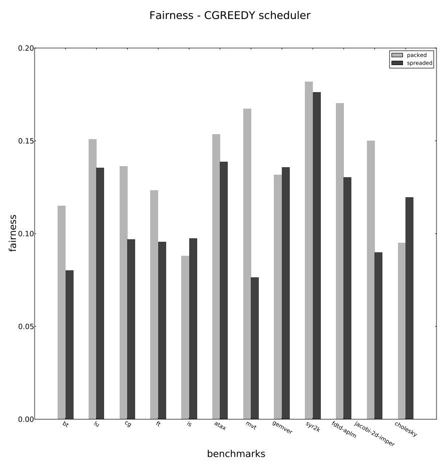


(β)

Figure 5.29: Total execution time and fairness of applications - GREEDY scheduler

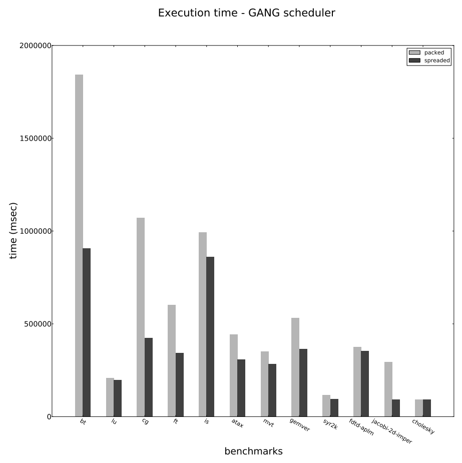


(α)

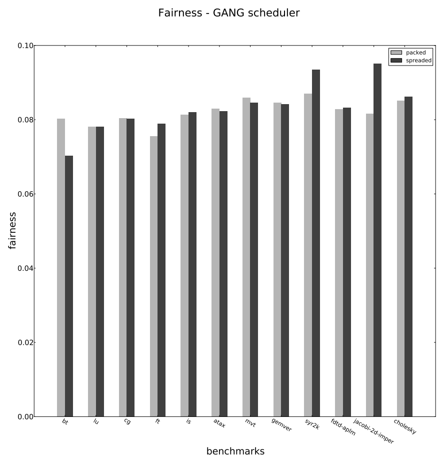


(β)

Figure 5.30: Total execution time and fairness of applications - CGREEDY scheduler



(α)



(β)

Figure 5.31: Total execution time and fairness of applications - GANG scheduler

Chapter 6

Conclusions and future work

Scheduling of multithreaded applications is an emerging issue in current platforms. Scheduling techniques that do not take into consideration the nature of multithreaded applications are not efficient. Real multithreaded applications, on their side, fail to scale well and exploit the abundance of the available resources, found in current computing systems. In this thesis we have explored many of the aforementioned issues regarding the scheduling of multithreaded applications on multicore systems. We have found that gang scheduling is necessary so that multithreaded applications achieve acceptable throughput. Moreover, space-sharing can be employed to utilize efficiently the available cores, since multithreaded applications fail to scale well for large threadcounts. Even space-sharing, though, will not be efficient unless scheduling algorithms incorporate contention aware techniques. We found that the limited memory bandwidth is a constraining factor that must be taken into account from scheduling algorithms. The scheduling methodologies we devised, based on gang scheduling and greedy packing of applications on available cores, were able to outperfom the Linux scheduler by a large factor. Finally, thread placement determines the level of contention for shared resources among applications. Different configurations, lead to different execution characteristics. Spreaded placement can be used to provide high throughput to multithreaded applications, while packed placement reduce the affection among applications and could be employed to enforce QoS techniques.

As an expansion of this work, we could study ways to allow us incorporate the results of contention for memory resources such as the memory bus bandwidth, in a scheduler implementation. If we can realiably measure the memory bandwidth consumption of applications, and exploit the results we exported from the experiments we conducted with STREAM, we could predict the slowdown of an application when co-scheduled with others. Moreover, we could expand our research in NUMA machines, which memory subsystem is organized differently and produce corresponding results. Those results would be significant since the current computer designs are NUMA.

Bibliography

[1] M. Bhadauria, S. A. McKee. 2010. An Approach to Resource-Aware Co-scheduling for CMPs. In ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing.

[2] Moinuddin K. Qureshi, Yale N. Patt. 2006. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. 423-432

[3] Yuejian Xie, Gabriel H. Loh. 2009. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches. In ISCA '09: Proceedings of the 36th Intl. Symp. on Computer Architecture. 174-183

[4] G. H. Loh. 2008. 3d-stacked-memory architectures for multi-core processors. In ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture. IEEE Computer Society, Washington, DC, USA, 453-464.

[5] Suh, G. E., Devadas, S., and Rudolph, L. 2002. A new memory monitoring scheme for memory- aware scheduling and partitioning. In HPCA '02: Proceedings of the 8th International Sympo- sium on High-Performance Computer Architecture. 117.

[6] J. Corbalan, X. Martorell, J. Labarta. 2000. Performance-driven processor allocation. In OSDI '00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4.

[7] J. Corbalan, X. Martorell, J. Labarta. 2001. Improving Gang Scheduling through Job Performance Analysis and Malleability. In ICS '01: Proceedings of the 15th international conference on Supercomputing.

[8] L. Tang, J. Mars, M. L. Soffa. 2010. Contentiousness vs. Sensitivity: Improving Contention Aware Runtime Systems on Multicore Architectures. In EXADAPT '11: Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era

[9] S. Zhuralev, S. Blagodurov, A. Fedorova. 2010. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In ASPLOS XV: Proceedings of the

fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems.

[10] A. Merkel, J. Stoess, F. Belossa. 2010. Resource-conscious Scheduling for Energy Efficiency on Multicore Processors. In EuroSys '10: Proceedings of the 5th European conference on Computer systems.

[11] Knauerhase, R., Brett, P., Hohlt, B., Li, T., and Hahn, S. 2008. Using OS observations to improve performance in multicore systems. IEEE Micro 28, 3, 54–66.

[12] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schupbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[13] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, SOSP '09, pages 221–234, New York, NY, USA, 2009. ACM.

[14] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. SIGOPS Oper. Syst. Rev., 43:76–85, April 2009.

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

[16] https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt

[17] http://www.ibm.com/developerworks/linux/library/l-scheduler/

[18] http://www.cs.virginia.edu/stream/

[19] http://www.nas.nasa.gov/publications/npb.html

[20] http://www.cse.ohio-state.edu/ pouchet/software/polybench/

[21] http://openmp.org/wp/