ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Μελέτη επιπτώσεων συνδρομολόγησης εφαρμογών σε πολυπύρηνες αρχιτεκτονικές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΟΡΕΣΤΗ Ρ. ΚΟΡΑΚΙΤΗ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2014

Page intentionally left blank.

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Μελέτη επιπτώσεων συνδρομολόγησης εφαρμογών σε πολυπύρηνες αρχιτεκτονικές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΟΡΕΣΤΗ Ρ. ΚΟΡΑΚΙΤΗ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την $26^{η}$ Σεπτεμβρίου 2014.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
....	....	....
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Νικόλαος Παπασπύρου
Καθηγητής Ε.Μ.Π.	Λέκτορας Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2014

(Υπογραφή)

....

ΟΡΕΣΤΗΣ ΚΟΡΑΚΙΤΗΣ

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Οι σύγχρονες αρχιτεκτονικές επεξεργαστών βασίζονται στην παρουσία πολλών υπολογιστικών πυρήνων πάνω στο ίδιο τσιπ, οι οποίοι μοιράζονται τη χρήση υποσυστημάτων της ιεραρχίας της μνήμης, όπως το τελευταίο επίπεδο της cache και το memory bus. Το γεγονός αυτό έχει ως αποτέλεσμα η παράλληλη εκτέλεση προγραμμάτων που κάνουν έντονη χρήση των υποσυστημάτων αυτών, σε γειτονικούς πυρήνες, να επηρεάζεται και να σημειώνεται πτώση της απόδοσης των εφαρμογών.

Ο σκοπός αυτής της διπλωματικής εργασίας ήταν η μελέτη των φαινομένων ανταγωνισμού μεταξύ των εφαρμογών για τους διαμοιραζόμενους αυτούς πόρους, που μπορεί να προκύψουν κατά τη συνεκτέλεση προγραμμάτων, και την επίδραση που έχουν στην απόδοση των εφαρμογών. Για να δημιουργηθεί ένα σύνολο εφαρμογών με ποικίλη συμπεριφορά και απαιτήσεις από τα υποσυστήματα της μνήμης, ώστε να προσομοιωθούν προγράμματα που κάνουν διαφορετική χρήση τους, αναπτύχθηκε μία εφαρμογή μέτρησης επιδόσεων μνήμης (benchmark). Το πρόγραμμα αυτό μπορεί να μετρήσει το ρυθμό μεταφοράς δεδομένων (bandwidth) στα διάφορα επίπεδα ιεραρχίας της μνήμης. Στη συνέχεια έγιναν πειράματα συνεκτέλεσης στιγμιοτύπων του benchmark, με διαφορετική συμπεριφορά και εκμεταλλευόμενων διαφορετικά υποσυστήματα. Τα πειράματα έγιναν σε δύο αρχιτεκτονικές, ώστε να μελετηθεί πώς οι ιδιαιτερότητες στη σχεδίαση και την ιεραρχία της μνήμης μπορούν να επηρεάσουν περεταίρω. Σε όλα τα πειράματα μετρήθηκαν οι επιδόσεις των εφαρμογών, ώστε να υπολογιστεί κατά πόσο μεταβάλλεται ο χρόνος εκτέλεσής τους κατά τη συνεκτέλεση, αλλά και η γενικότερη συμπεριφορά τους.

Τα αποτελέσματα μπορούν να χρησιμοποιηθούν για τον έλεγχο και την επιβεβαίωση εκτιμήσεων της συμφόρησης στα υποσυστήματα μνήμης, που υπολογίζονται από προτεινόμενα μοντέλα πρόβλεψης και αποφυγής τέτοιων φαινομένων, ώστε να γίνει πιο αποδοτική η δρομολόγηση (scheduling) των εφαρμογών σε πολυπύρηνα συστήματα. Τέλος, το μετρητικό πρόγραμμα που υλοποιήθηκε, μπορεί να χρησιμοποιηθεί ως εναλλακτική λύση τόσο για μετρήσεις επιδόσεων μνήμης, όσο και για την προσομοίωση προγραμμάτων που κάνουν έντονη χρήση των υποσυστημάτων μνήμης για πειράματα συνεκτέλεσης.

Λέξεις Κλειδιά: Πολυπύρηνες αρχιτεκτονικές, συνδρομολόγηση, συμφόρηση, ανταγωνισμός στους μοιραζόμενους πόρους, memory benchmark

Page intentionally left blank.

Abstract

Modern processor architectures have moved towards utilizing multiple cores on the same physical package, which share resources of the memory hierarchy, e.g. last-level cache, memory bus bandwidth. As a result, concurrent execution of programs that make significant use of shared memory subsystems, on cores of the same package, leads to performance degradation phenomena for co-executed applications.

The objective of this thesis was to study contention effects in shared memory resources, as a result of co-execution, and its impact on applications' performance. A memory benchmark program was developed, which can measure bandwidth in all levels of the memory hierarchy. This benchmark was used to create a set of instances with different behavior and memory usage intensity, in order to emulate a variety of memory-bound applications that utilize different memory hierarchy subsystems. Co-scheduling scenarios with all combinations of the aforementioned suite were tested on two architectures, with different characteristics. This also enabled us to observe how specific architecture features and design differences may further affect applications' interference. Performance metrics were used for all experiments in order to detect impact on execution time, as well as alterations on their general behavior.

Results of the experiments can be used to validate contention estimations based on application classification models of literature-suggested contention-aware co-scheduling approaches. Additionally, the proposed benchmark program can be further used and expanded as an alternative choice for both memory performance evaluation and emulation of various memory-intensive workloads for experiments.

Keywords: Multicore architectures, CMPs, co-scheduling, contention-aware scheduling, application classification, memory benchmark

Page intentionally left blank.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω όλο το προσωπικό του εργαστηρίου Υπολογιστικών συστημάτων και τον επιβλέποντα καθηγητή μου, κ. Ν. Κοζύρη, για την ευκαιρία να δουλέψω στο CSLab, τους Δρ. Γ. Γκούμα και Δρ. Κ. Νίκα για τις χρήσιμες συμβουλές τους. Ειδικότερα, θα ήθελα να εκφράσω τις ευχαριστίες μου στον Δρ. Νίκο Αναστόπουλο, που με καθοδήγησε από την αρχή στην εκπόνηση της παρούσας εργασίας και στον Αλέξανδρο Χαριτάτο, χωρίς τη διαρκή βοήθεια και υποστήριξη του οποίου δε θα μπορούσα να προχωρήσω. Τέλος θα ήθελα να ευχαριστήσω την οικογένειά μου και τους φίλους, οι οποίοι με την υποστήριξή τους συνέβαλαν στην περάτωση της εργασίας.

Page intentionally left blank.

Table of contents

Page intentionally left blank.

Chapter 1

Introduction

1.1 Multicore architectures concepts

Modern computer architectures' design and principles have changed in the last decade. In previous years, processors consisted of a single core which handled and executed the whole workload. Performance of uni-core systems was increased by microarchitecture improvements, progress in manufacturing methods that allowed for higher clock frequencies with lower power consumption and implementation of more complex and efficient memory hierarchy designs. Progress using this model has declined in recent years, as reaching the physical limits of semiconductor microelectronic materials and manufacturing techniques causes intractable problems, related to increased heat dissipation and data synchronization, among others. These limitations have resulted in a different approach in newer designs, using multiple cores on the same chip, sharing the workload instead of a single core running all tasks.

The concept of a computer system consisting of two or more physical processors sharing the main memory was known and used in previous decades on mainframe, server and workstation implementations (e.g. symmetric multi-processing, SMP). Demand for increased performance in general purpose systems with single processors and the previously mentioned limitations of uni-core designs led to a new trend in computer architecture, combining two or more CPU cores on the same physical processor package. All cores, operating simultaneously, can execute instructions independently, resulting in tremendous increase of the system's throughput potential. Usage of such chip multi-processors (CMPs) has seen excessive growth in recent years, being present in all types of computer systems, from servers to mobile phones.

However, contrary to systems with multiple physical processors, cores of a CMP share more resources, such as cache memory, main memory bus bandwidth available to the socket, data prefetchers. As a result, cores are not completely independent from each other. Applications executed concurrently on different cores may cause contention in these shared resources, leading to reduced hardware efficiency and subsequently performance degradation. It becomes apparent that conflicts for shared resources utilization need to be minimized to avoid such unwanted effects.

1.2 Operating system - Scheduling

The operating system is an essential component of a computer system. It is responsible for hardware and software management, providing services needed by applications, enabling them to utilize hardware resources, communicate with each other and the user. The scheduler plays a critical role in operating systems, being responsible for application execution and resource allocation. A scheduler has to make decisions about how CPU time, I/O and other resources will be shared among processes, which processes will be assigned to specific cores, starting and stopping applications in order to provide efficiency for the system. Different types of systems may require different scheduler approaches, for instance a personal computer needs optimized application responsiveness, while the objective for a server running multiple tasks is throughput maximization.

For single CPU systems, dominant in the market until recently, main concern of OS scheduling was the allocation of processor time among processes. In this context, scheduling techniques had been highly optimized over the years and became very efficient, to the point that there was no demand for further improvement. This was also the case for SMP systems, processors of which only share main memory. Notable open source examples of highly efficient schedulers are the 2 latest scheduler implementations of the Linux kernel, known as O(1) scheduler [2] and CFS [3]. With CMPs, the scheduling problem becomes much more complex. Shared memory resources among multiple cores make decisions about process execution much more difficult, as architecture specific parameters must be considered (e.g. what levels of cache hierarchy and other resources are shared, among which cores) to make the optimal choices. It becomes apparent that space-sharing of the CMP needs to be optimized along with time-sharing, since the choice of processor assigned to execute a task may have negative effects on programs' execution [1].

1.3 Problem definition

Scheduler implementations in mainstream operating systems, optimized for SMP architectures, when used with multi-core single chip processors, treat all cores as independent from each other. This simplified approach may result in concurrent execution of applications that make intense use of shared memory resources, leading to contention and causing significantly reduced efficiency and performance. Optimization of scheduling algorithms to better utilize resources in CMP context is an active field of research, due to multicore designs' massive adoption, even on handheld devices.

Many approaches suggest contention-aware scheduling techniques. It is assumed that if architecture details are known, including memory hierarchy and shared resources, then a program's behavior in co-execution and potential interference suffered or caused can be estimated by observing its memory utilization needs. Such approaches rely on various application classification schemes, based mainly on memory-associated behavior, in order to facilitate scheduling related decisions. Knowing the overall picture of how classes interact in co-scheduling scenarios on a certain architecture, concurrent execution of programs potentially harmful for the system's efficiency and throughput might be avoided.

The objective of this work is to study concurrent execution of memory intensive applications on multicore architectures and the impact it has on shared memory hierarchy and other programs' performance. Co-execution experiments are used to evaluate a literature-suggested memory contention estimation model based on an application classification scheme. To emulate different class memory-bound programs, a versatile memory pseudo-benchmark with user-controlled behavior was developed.

1.3.1 Contribution

Co-scheduling experiments were conducted on multicore architectures with different design characteristics and memory hierarchy organization. To evaluate performance of all levels in the memory hierarchy we created a benchmark program. This benchmark is extensively tested on the systems we intend to use for co-execution experimentation and results are compared with other known benchmarks to estimate its validity. Taking advantage of its versatile design, the benchmark was used to create a suite of instances with different behavior. It is suggested that this benchmark program can be used as an in-house alternative to emulate a wide range of memory-bound processes. Application performance is profiled using the aforementioned classification scheme and data collected is used to make general contention estimations for various co-execution scenarios.

All applications of the proposed set are co-executed in pairs with each other. In each experiment we observe if contention on the memory hierarchy occurs and how it affects co-running applications' execution time. Slowdown results are compared with expected behavior, to discuss the validity of the contention-based slowdown estimation model. We show that excessive application slowdown may occur in certain co-scheduling scenarios due to contention in memory subsystems. It is also demonstrated how architecture specific characteristics can drastically affect programs' performance degradation in co-execution context.

1.4 Chapter description

Chapter 2 describes the slowdown estimation model and classification algorithm used in this work, along with similar research examples.

A detailed description of the proposed benchmark program, its results and evaluation are found in chapter 3. Additionally, that chapter contains information about the specific computer systems used, including architecture details, and the co-scheduling infrastructure used for experiments

Chapter 4 contains all co-scheduling experimentation related work: workload description, preliminary evaluation, experimental procedure and results.

Conclusions, along with ideas for future work are summarized in chapter 5.

Page intentionally left blank.

Chapter 2

Motivation and current approaches

2.1 The scheduling problem on CMPs

Chip multi-processors are designed to improve performance by providing parallel computational cores to share workload, executing programs concurrently. Although there are huge potential gains from this approach, scheduling execution of threads on a CMP in an efficient way is a very complex problem. The main reason for this is that cores are not completely independent from each other, as they are sharing cache memory, access to the main memory bus, controllers, hardware prefetchers and, possibly, other resources with neighboring or all cores.

When multicore processors appeared, they were handled by existing OS schedulers similarly with cores of symmetric multi-proccessing (SMP) systems, which differ significantly as they consist of separate physical processors on different sockets. However, processors of SMP system are independent, as they only share main memory. Even with the shared memory, NUMA architectures (Non-Uniform Memory Access) provide mechanisms for avoiding memory bus related conflicts, as each socket can access its relatively closer located memory faster. Thus, the OS scheduling problem is addressed as mainly managing run-queues, ensuring that cpu-time is efficiently allocated for overall throughput. Very efficient and optimized algorithms have been developed, which led to the scheduling problem being considered as solved.

Applying the same scheduling techniques on systems with CMPs can have largely unpredictable effects. Sharing memory resources can be very beneficial or extremely destructive for running applications. Threads of an application, using the same data, can largely take advantage of shared cache, when executed concurrently on cores of a multicore processor. Similar speed-up can occur when different programs, sharing libraries, are simultaneously scheduled. On the other hand, a process streaming large amounts of data, intensely replacing the content of cache levels, can be extremely harmful for other running programs, forcing them to continuously re-fetch data from main memory and subsequently suffer from highly increased time penalties. These examples demonstrate extreme cases and help to intuitively understand how complex co-scheduling scenarios can be.

Details of the architecture, mapping of shared resources and behavior of the programs consisting the workload are all very significant factors in order to make efficient scheduling decisions. Space-sharing the CMP is equally important to time-sharing it, as the choice of which core is assigned for a programs’ execution may dramatically affect, positively or negatively, the program’s behavior, as well as other running processes. Sharing memory resources between cores running different applications can create contention in some or even all levels of the memory hierarchy that are shared. Contention is the main reason for performance degradation in CMP co-execution context. Thus, current research approaches, found in literature, try to address the problem on CMPs with contention-aware scheduling methods.

For contention-aware scheduling, it is assumed that in architectures with multiple cores, different resources are shared among core subsets, since if all cores shared all resources equally, space-sharing decisions would have no effect. Additionally, all resource sharing is considered to cause negative effects, as contention aware-schedulers try to avoid such interference. Suggested prediction schemes are based on classifying programs of the workload, using their behavior profile, attempting to keep apart applications that cause stress to the memory hierarchy [1]. These prediction mechanisms use performance metrics, such as

LLC misses and cache utilization patterns or other heuristic methods to detect potential application interference in co-execution scenarios. But even with an ideal, very accurate prediction model, finding the optimal mapping in systems with more than 2 cores is shown to be an NP-complete problem [4]. However, it is possible to suggest a much more efficient mapping, compared to initial contention-unaware scheduling.

2.2 Related work

Contention aware scheduling methodologies, for CMP and SMP or cluster systems, found in literature use classification schemes to characterize workload behavior. Many approaches have been suggested, ordering applications by performance characteristics, such as LLC miss rate, cache re-use patterns, main bus utilization. Bhadauria and McKee [6] use cache miss rate (hits/misses) or bus occupancy metrics, trying to balance resource utilization to be fairly shared among processes. Xie and Loh [8] use an approach classifying programs in animal categories. There are 4 such classes: Turtles, sheep, rabbits and Tasmanian devils. Turtles make zero or very low use of shared resources, relying in the lower level private cache of the core. Sheep and rabbits re-use LLC intensely, with the difference being that rabbits are very sensitive to the ways of the cache allocated to them, while sheep are not easily affected. Finally, devils heavily use cache, but also have a large number of misses; as a result they are very harmful for co-running applications.

Blagodurov et al.[7] proposed the Pain classification scheme. In this approach, the terms cache “sensitivity” and “intensity” are suggested. Sensitivity shows how likely it is for a program’s cached data to be replaced, using probabilities based on stack-distance profile (SDP) and reuse frequency. To calculate intensity, authors use the ratio of cache access operations per million instructions, to indicate how aggressively an application uses the shared cache. The product of sensitivity and intensity is then used to calculate the “pain” an application will suffer from and cause to co-runners, due to co-scheduling. A similar approach was proposed by Tang et al. [11], with contentiousness and sensitivity metrics for applications to estimate performance degradation.

Jaleel et al. [10] suggest a categorization model base on applications' cache utilization. There are 4 classes in this model: Core Cache Fitting (CCF), LLC Thrashing (LLCT), LLC Fitting (LLCF) and LLC Friendly (LLCFR) applications. CCF programs have small working sets and fit in private caches, without need to use the shared LLC. LLCT are applications with working sets much larger than LLC, making streaming accesses and replacing cache content and thus being very harmful for co-running programs that use the LLC. LLCF programs need a large part of the LLC, and are affected if competition for the cache occurs. LLCFR applications benefit from cache re-use, but are not so sensitive when available cache resources are reduced.

Lin et al. [12] used a color-based classification scheme to allocate cache usage by applications. Mars et al. [16] introduced "bubble", a stress test for the memory subsystem, with gradually increasing intensity, in order to characterize applications' sensitivity by their performance curve, as the "bubble" grows. They also observe how much an application stresses the subsystem itself. Being aware of contentiousness and sensitivity, co-scheduling behavior may be predicted.

Other approaches try to balance shared memory bus utilization and avoid saturation [9] as memory bus contention is seen as the major factor of performance degradation. Merkel et al. [13] try to address the issue by scheduling programs that use complementary resources for co-execution. Numerous other examples are present in literature, as research for CMP-optimized scheduling is an active and challenging field.

2.3 LCA: A memory Link and Cache-Aware approach

A recent approach suggested in literature is LCA [5]. LCA tries to address the CMP co-scheduling problem using a classification scheme based on the overall picture of memory resources utilization. Data flow in all levels of memory hierarchy is observed to predict interference problems, attempting to deal with dual contention on both memory link and shared cache.

The contention-avoidance scheme, suggested by this approach divides applications in classes, using information that can be collected by modern processors' performance monitoring mechanisms during execution time, without need for additional hardware modifications or support. The next step is contention estimation, based on the workload's classification, and corresponding decisions for time-and-space allocation of the CMP. This thesis, however, focuses only on interference caused by co-execution of memory intensive applications, thus scheduling algorithms are not discussed in detail. The following four application classes are used:

Class N: Applications that display activity on the core's private part of the memory hierarchy. This may include application with computational load, very small working sets, fitting in lower level cache, optimized data accesses, or any combination of these characteristics. Programs of this class create no contention on shared memory resources.

Class C: This class includes a wide range of applications, which benefit from shared cache (mainly LLC) reuse. Programs of this class can have different characteristics, e.g. applications intensely accessing a dataset small enough to partially fit in cache, latency-bound processes that make irregular memory accesses and benefit from cache hits. Since they rely on cache reuse, applications in this class can be affected by LLC intereference.

Class LC: Applications that require significant use of both memory link and cache, at varying levels. Again, this class contains many different applications with varied behavior, for instance programs that need to fetch large amounts of data from main memory, which are then processed, displaying intense cache reuse.

Class L: Applications included are stressing the memory bus, consuming a significant percentage of its bandwidth. Examples of programs belonging to this class are applications that use datasets much larger than cache, performing streaming memory accesses with very little or zero cache reuse. Such applications are also expected to utilize data prefetchers to achieve high levels of memory bandwidth.

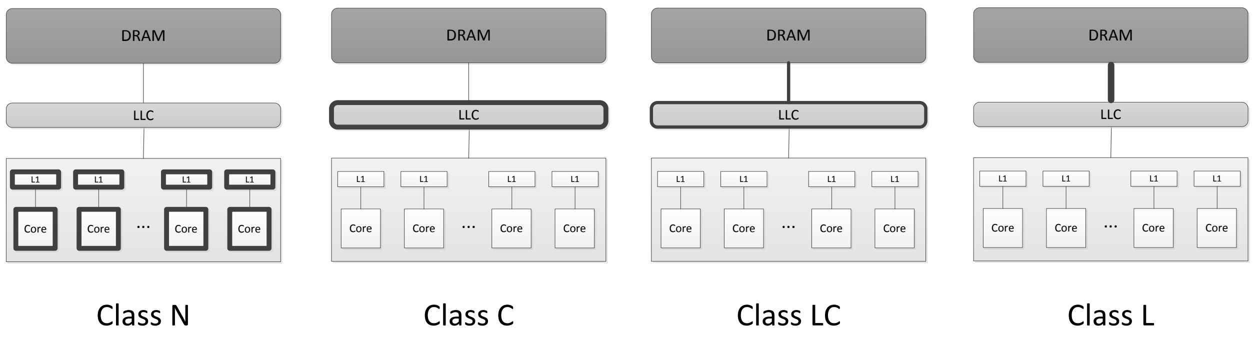


Figure 2.1: Application classes' activity [5]

Although all classes contain applications with different execution patterns and behavior, this scheme can be used to capture the overall picture of co-execution scenarios and detect potential contention situations. Co-scheduling scenarios between different classes and expected behavior, as a primary objective of this work, are discussed in more detail in chapter 4, yet a concise, per-class estimation guide follows:

Class N: No interference is expected, as application execution mostly relies on the core's private resources.

Class C: The wide variation of applications result in many difficult to predict scenarios, being affected by many factors. However, most cases of LLC-sharing (C-C combination) co-execution are expected to display minimum to moderate slowdown effects, a generally "low contention" scenario. Increased contention can occur when co-runner class shifts to a more memory intensive, LC or L, resulting even in severe slowdown effects, for instance when a L program continuously wipes cached data.

Class LC: Moderate interference is expected in the case of an LC-LC combination, as a result of medium contention in both link and cache. C class competition for the LLC could also cause low performance degradation to an LC instance, while an L co-runner can have much more impact, reducing available memory bandwidth and replacing data in the cache.

Class L: Relying solely on the memory link, L processes are expected to be affected only by other L instances competing for the bus, or LC that have increased memory link bandwidth demand as a result of interference caused by the L itself.

To classify applications in the aforementioned classes, this approach suggests a method of inspecting data flow to the core, through the levels of the memory hierarchy to detect at which levels higher utilization is noted, using runtime performance statistics. To achieve that, memory bandwidth is measured first, to determine if a program belongs in class L, or LC. If not, LLC data towards the core bandwidth is inspected to decide if LLC or other shared cache data reuse occurs, and classify the program into N or C classes accordingly. If overall data flow is low, IPC (Instructions per Cycle) and the ratio of memory micro-operations to all micro-operations metrics are used to classify an application in N and C classes.

To apply this decision scheme, five thresholds need to be set:

α: High memory link bandwidth utilization
β: Medium memory link bandwidth utilization
γ: High cache-to-core bandwidth utilization
δ: $\frac{\text{mem\_uops}}{\text{all\_uops}}$ ratio, higher than which shows a memory bound application
ε: IPC threshold, higher than which indicates a more CPU-intensive N application

The respective values, adapted for single threaded applications*, are calculated as follows: $\alpha = 0.5 \cdot B_{max}$, $\beta = 0.025 \cdot B_{max}$, $\gamma = 0.15 \cdot B_{max}$, $\delta = 0.25$, $\varepsilon = 0.25 \cdot IPC_{max}$, where B_{max} is maximum memory link bandwidth and IPC_{max} maximum theoretical IPC of the processor. This classification scheme can be described by the following decision tree:

*Note: Thresholds mentioned in the original work are calculated for 4-threaded applications.

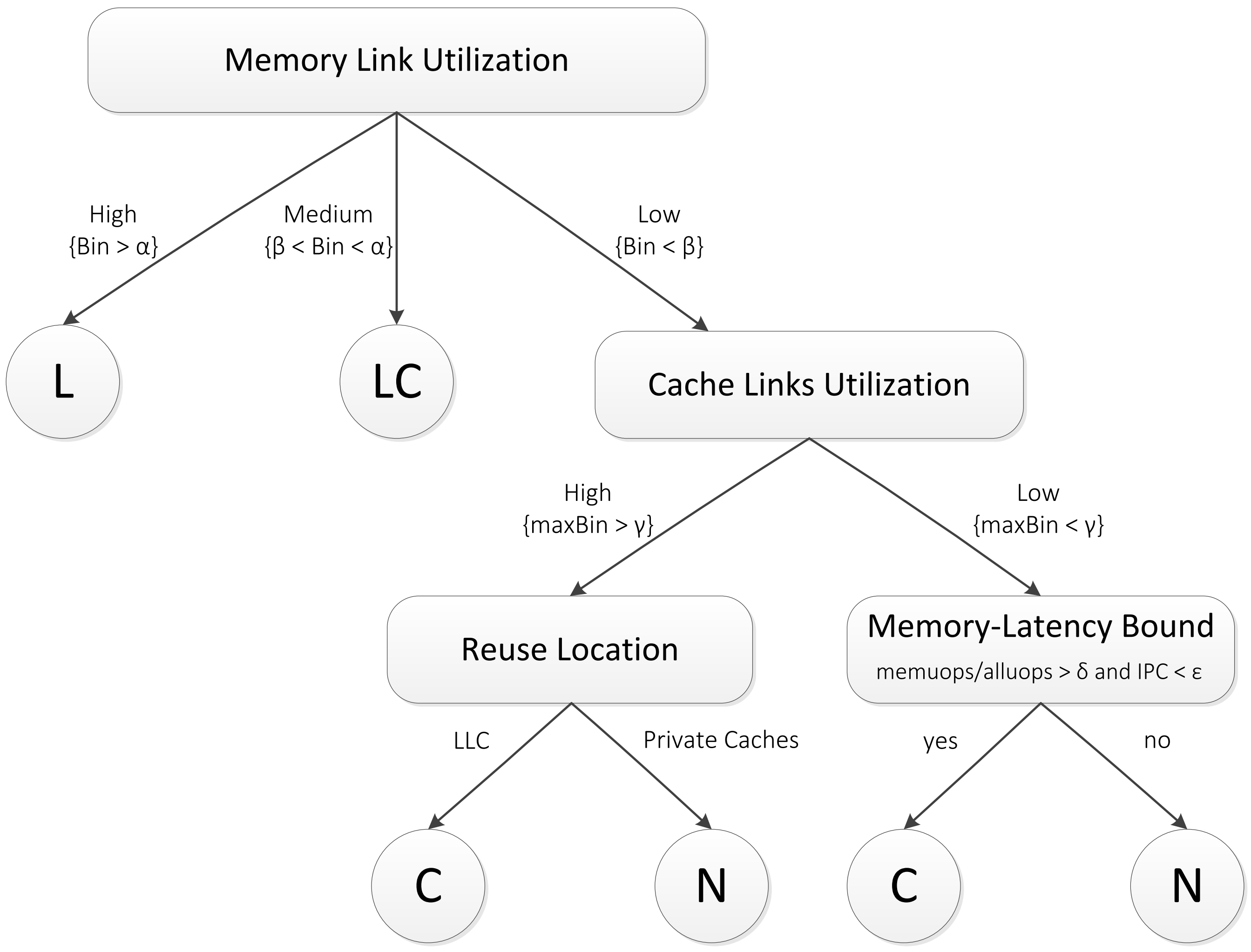


Figure 2.2: Decision tree for application classification [5]

In this work, the aforementioned scheme will be used to classify instances of a single-threaded memory pseudo-benchmark. Through extensive co-scheduling tests, the objective is to observe all classes' variations interaction with each other, and interference effects caused by contention in all levels of the memory hierarchy. Results will be compared with the estimation model and further discussed.

Chapter 3

Hardware and software systems used

In the following chapter, hardware and software infrastructure used is explained. Description of computer systems is followed by presentation of the memory benchmark program we created to study memory contention. Subsequently, the program is evaluated by comparing results with expected behavior, as well as other known benchmarks' metrics.

3.1 System characteristics

For co-scheduling tests, two processor architectures were chosen, with different characteristics, both, of course, belonging to x86_64 family: Intel® Sandy Bridge and Intel Dunnington.

3.1.1 Intel® Sandy Bridge

The first system consists of four Intel Xeon® E5-4620 processors on Intel C600 series chipset, with 256 GB of DDR3 main memory. Each physical package contains 8 cores with private 32 KB instruction and 32 KB data Level 1 cache and 256 KB L2 cache, both 8-way associative. All cores share the package's 16 MB, 16-way associative L3 cache. Cache line size is 64 bytes. It should be mentioned that this processor features hyper-threading technology with two threads per core, thus system appears to have two logical CPUs for every core; however, we will not be utilizing hyper-threading in this work, as well as other parallelization technologies featured, like QuickPath-Interconnect (QPI) bus, thus they will not be further explained. Each package communicates with main DDR3 memory through 4 channels, greatly beneficial for concurrent access requests, with maximum bandwidth being ~14GB/s per core, 18GB/s per package, and 52GB/s total maximum for all four sockets. It also features hardware data prefetching for both memory link and caches. Figure 3.1 shows a diagram of a single package, while the system's four packages (0-3) with the corresponding Linux kernel CPU numbering (e.g. cpu0) are shown in figure 3.2.

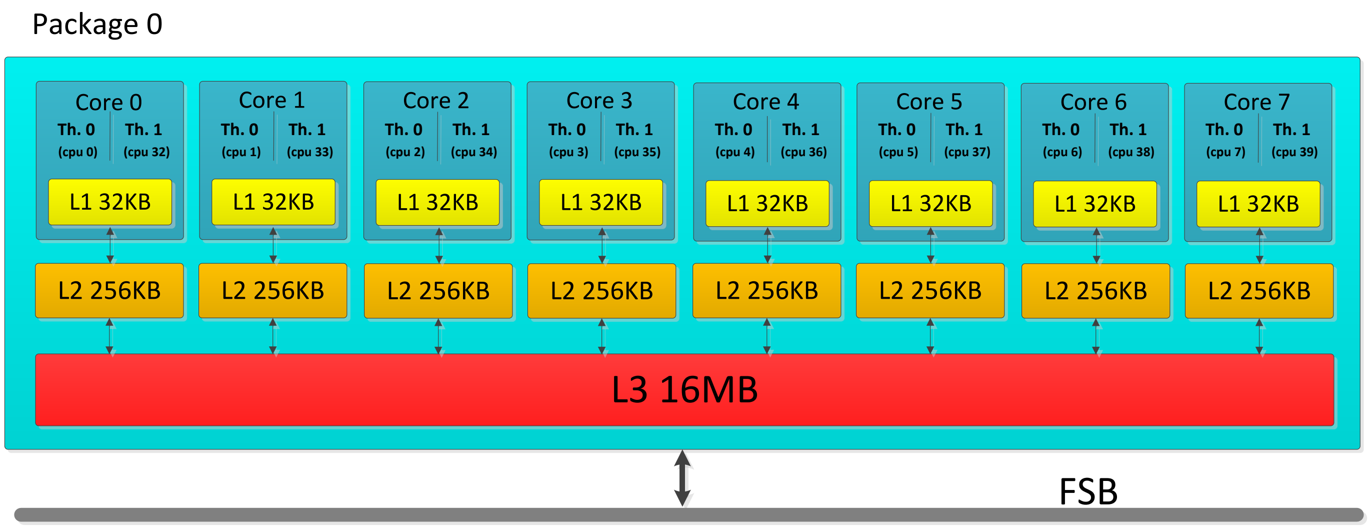


Figure 3.1: 1st of 4 Intel® Xeon® E5-4620 packages

This processor was chosen for co-execution tests since it features private L2 cache for each core, ensuring that cache level contention will only affect LLC. Additionally, its four-channel memory bus of relatively high performance can let us experiment with different behavior of applications with varied levels of memory link stressing needs and how they interact. It also enables us to observe how data prefetching affects memory intensive applications with different data access patterns.

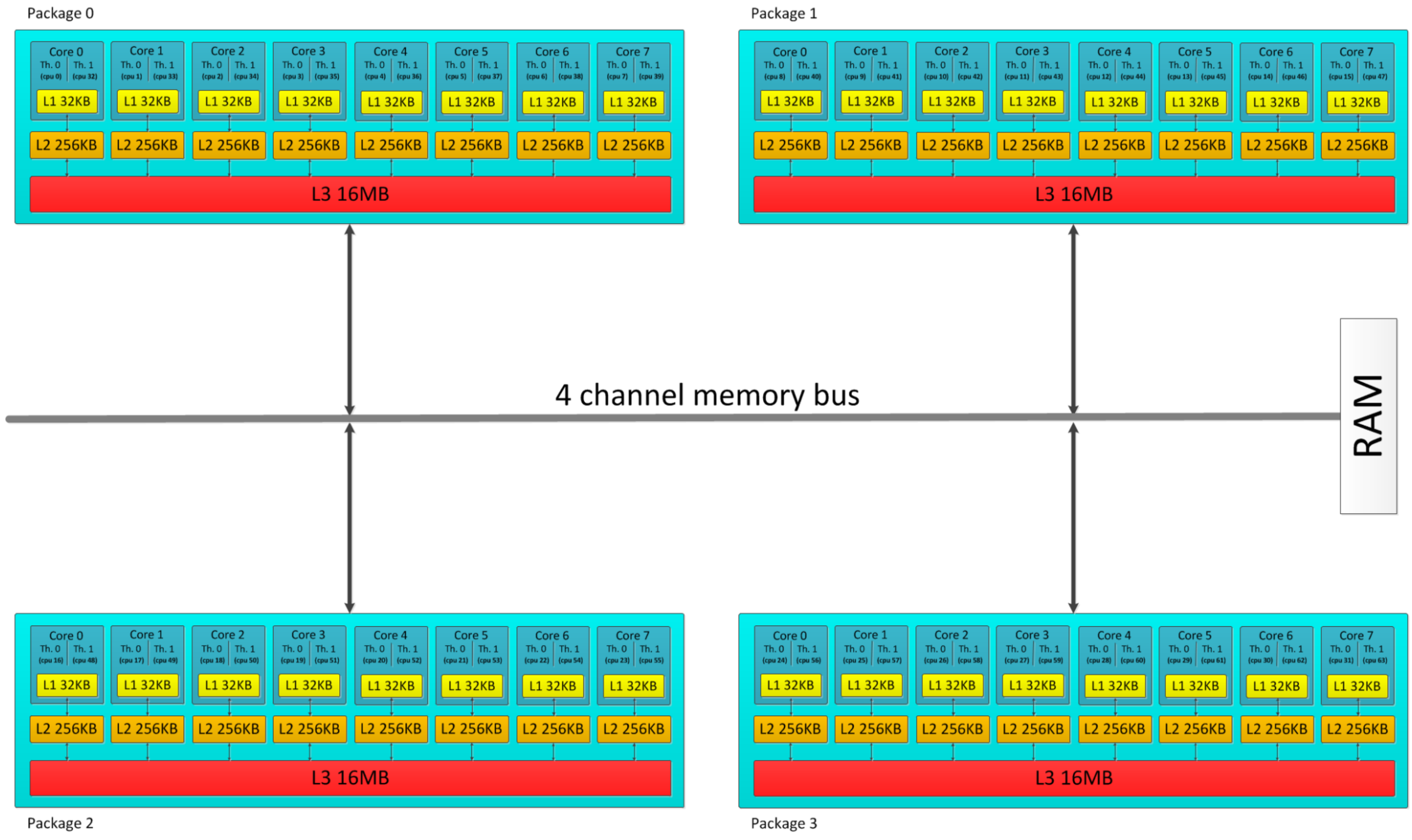
Finally, Intel also provides performance monitoring capabilities. Using performance counters extracted directly from the processor gives the ability to study application behavior and its alterations in greater depth and to better explain the effects of co-scheduling. Intel's Sandy Bridge performance monitoring infrastructure offers a very wide range of performance events counters [19]. Counters used for this work will be mentioned further in this chapter, along with the respective performance metrics.

Overall, it is a modern, high performance system, able to handle demanding workloads and is expected to be a suitable platform to evaluate the previously described contention estimation model.

This system runs Debian 6.0.9 GNU/Linux operating system, using Linux kernel 3.7.10, gcc version 4.6.3 and glibc version 2.11.3. Table 3.1 summarizes the system’s hardware characteristics:

# of packages	4
Cores/Socket	8
Threads/Core	2
CPU frequency	2.2 GHz (TurboBoost™ up to 2.6 GHz)
L1 Cache	32KB data + 32KB instr., private per core, 8-way
L2 Cache	256KB private per core, 8-way
L3 Cache	16MB shared, 16-way
RAM	256GB DDR3, 4-channel bus

Table 3.1



Package 0
Core 0 Th. 0 Th. 1 (cpu 0) (cpu 32) L1 32KB L2 256KB
Core 1 Th. 0 Th. 1 (cpu 1) (cpu 33) L1 32KB L2 256KB
Core 2 Th. 0 Th. 1 (cpu 2) (cpu 34) L1 32KB L2 256KB
Core 3 Th. 0 Th. 1 (cpu 3) (cpu 35) L1 32KB L2 256KB
Core 4 Th. 0 Th. 1 (cpu 4) (cpu 36) L1 32KB L2 256KB
Core 5 Th. 0 Th. 1 (cpu 5) (cpu 37) L1 32KB L2 256KB
Core 6 Th. 0 Th. 1 (cpu 6) (cpu 38) L1 32KB L2 256KB
Core 7 Th. 0 Th. 1 (cpu 7) (cpu 39) L1 32KB L2 256KB
L3 16MB
Package 1
Core 0 Th. 0 Th. 1 (cpu 8) (cpu 40) L1 32KB L2 256KB
Core 1 Th. 0 Th. 1 (cpu 9) (cpu 41) L1 32KB L2 256KB
Core 2 Th. 0 Th. 1 (cpu 10) (cpu 42) L1 32KB L2 256KB
Core 3 Th. 0 Th. 1 (cpu 11) (cpu 43) L1 32KB L2 256KB
Core 4 Th. 0 Th. 1 (cpu 12) (cpu 44) L1 32KB L2 256KB
Core 5 Th. 0 Th. 1 (cpu 13) (cpu 45) L1 32KB L2 256KB
Core 6 Th. 0 Th. 1 (cpu 14) (cpu 46) L1 32KB L2 256KB
Core 7 Th. 0 Th. 1 (cpu 15) (cpu 47) L1 32KB L2 256KB
L3 16MB
4 channel memory bus
RAM
Package 2
Core 0 Th. 0 Th. 1 (cpu 16) (cpu 48) L1 32KB L2 256KB
Core 1 Th. 0 Th. 1 (cpu 17) (cpu 49) L1 32KB L2 256KB
Core 2 Th. 0 Th. 1 (cpu 18) (cpu 50) L1 32KB L2 256KB
Core 3 Th. 0 Th. 1 (cpu 19) (cpu 51) L1 32KB L2 256KB
Core 4 Th. 0 Th. 1 (cpu 20) (cpu 52) L1 32KB L2 256KB
Core 5 Th. 0 Th. 1 (cpu 21) (cpu 53) L1 32KB L2 256KB
Core 6 Th. 0 Th. 1 (cpu 22) (cpu 54) L1 32KB L2 256KB
Core 7 Th. 0 Th. 1 (cpu 23) (cpu 55) L1 32KB L2 256KB
L3 16MB
Package 3
Core 0 Th. 0 Th. 1 (cpu 24) (cpu 56) L1 32KB L2 256KB
Core 1 Th. 0 Th. 1 (cpu 25) (cpu 57) L1 32KB L2 256KB
Core 2 Th. 0 Th. 1 (cpu 26) (cpu 58) L1 32KB L2 256KB
Core 3 Th. 0 Th. 1 (cpu 27) (cpu 59) L1 32KB L2 256KB
Core 4 Th. 0 Th. 1 (cpu 28) (cpu 60) L1 32KB L2 256KB
Core 5 Th. 0 Th. 1 (cpu 29) (cpu 61) L1 32KB L2 256KB
Core 6 Th. 0 Th. 1 (cpu 30) (cpu 62) L1 32KB L2 256KB
Core 7 Th. 0 Th. 1 (cpu 31) (cpu 63) L1 32KB L2 256KB
L3 16MB

Figure 3.2

3.1.2 Intel® Dunnington

This system features Intel® Xeon® X7460 "Dunnington" processors [23]. Again it is a four-socket system with Intel's 7300 chipset. Each package consists of six cores with private 32 KB instruction and 32 KB data, 8-way associative Level 1 cache. Cores in a package are in 3 pairs, each pair sharing 3 MB of 12-way associative L2. Each package comes with 16 MB, 16-way associative L3 cache, shared between all 6 cores. Again, cache line size is 64 bytes. Figure 3.3 shows the topology of a package, with the respective Linux kernel CPU numbering, while figure 3.4 shows all packages:

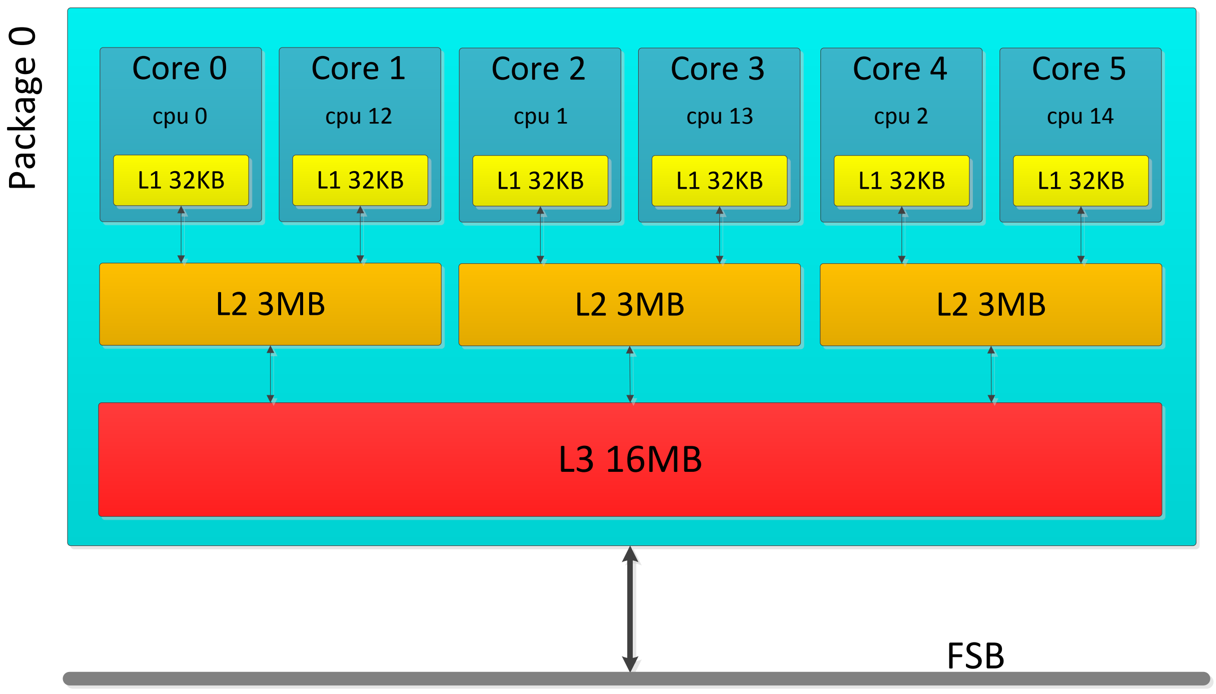


Figure 3.3: Intel® Xeon® X7460 package.

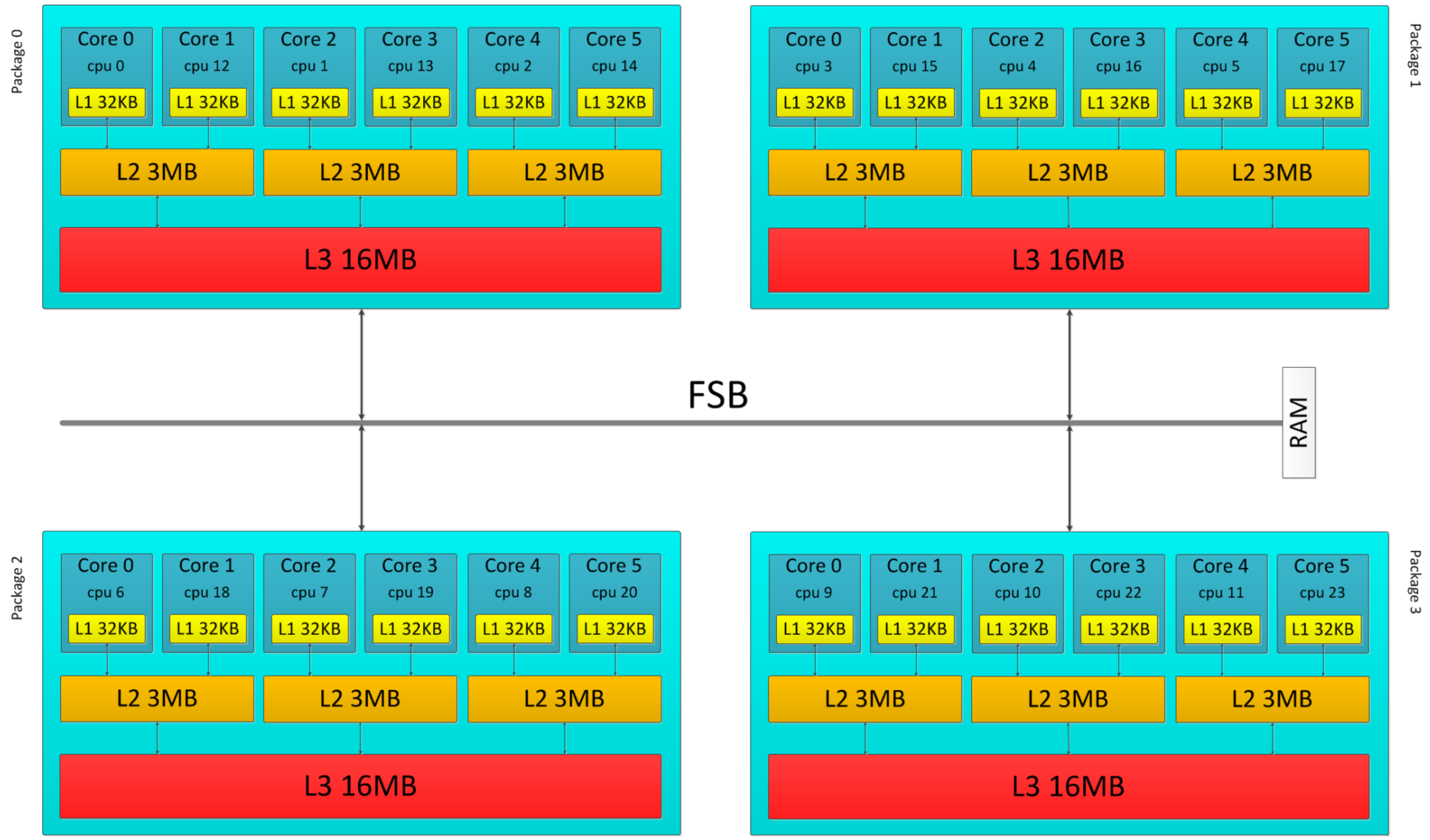
The system features 1066 MHz memory bus, connecting processors with the 27 GB DDR2 RAM. System's characteristics are summarized in table 3.2:

# of packages	4
Cores/Socket	6
Threads/Core	1
CPU frequency	2.66 GHz
L1 Cache	32KB data + 32KB instr., private per core, 8-way
L2 Cache	3MB shared per 2 cores, 12-way
L3 Cache	16MB shared, 16-way
RAM	27GB DDR2, 1066MHz

Table 3.2

This system runs Debian 6.0.7 GNU/Linux operating system, using Linux kernel 3.7.10, gcc version 4.4.7 and glibc version 2.13.

Dunnington was chosen for additional testing, since it features interesting differences in memory hierarchy, most notably the large but shared L2 cache. This gives us the opportunity to experiment with co-execution on cores sharing L2 and directly compare results with the same experiment on non-adjacent cores that will not compete for L2. It also gives potential to see the accumulated effects of contention when it occurs on all three levels of shared memory hierarchy (L2, LLC, memory link). Additionally, disabling memory bus data prefetching is expected to alter differences in performance between access patterns, comparing to the equivalent Sandy Bridge tests. However, prefetching mechanisms are present between caches and in the main memory DRAM controller. Also, contrary to the previous architecture, each socket is connected to main memory through a single memory bus (instead of four), thus concurrent memory access requests are expected to be serialized in order to be served by the memory link. This set of differences, especially L2 sharing and memory bus performance, may alter predicted behavior patterns in co-scheduling context.



Package 0
Core 0
cpu 0
L1 32KB
Core 1
cpu 12
L1 32KB
Core 2
cpu 1
L1 32KB
Core 3
cpu 13
L1 32KB
Core 4
cpu 2
L1 32KB
Core 5
cpu 14
L1 32KB
L2 3MB
L2 3MB
L2 3MB
L3 16MB
Package 1
Core 0
cpu 3
L1 32KB
Core 1
cpu 15
L1 32KB
Core 2
cpu 4
L1 32KB
Core 3
cpu 16
L1 32KB
Core 4
cpu 5
L1 32KB
Core 5
cpu 17
L1 32KB
L2 3MB
L2 3MB
L2 3MB
L3 16MB
FSB
RAM
Package 2
Core 0
cpu 6
L1 32KB
Core 1
cpu 18
L1 32KB
Core 2
cpu 7
L1 32KB
Core 3
cpu 19
L1 32KB
Core 4
cpu 8
L1 32KB
Core 5
cpu 20
L1 32KB
L2 3MB
L2 3MB
L2 3MB
L3 16MB
Package 3
Core 0
cpu 9
L1 32KB
Core 1
cpu 21
L1 32KB
Core 2
cpu 10
L1 32KB
Core 3
cpu 22
L1 32KB
Core 4
cpu 11
L1 32KB
Core 5
cpu 23
L1 32KB
L2 3MB
L2 3MB
L2 3MB
L3 16MB

Figure 3.4

3.2 Co-scheduling environment

3.2.1 Scaff

For all co-execution experiments, scaff infrastructure was used. Scaff is a runtime system used to coordinate the execution of a workload consisting of multithreaded applications on a multi-core/multi-processor system. It operates on user-level, on Linux-based systems. It is used to provide a communication mechanism between a scheduler and the programs executed. Scaff's infrastructure consists of two basic systems: the executor and the scheduler. The executor handles execution events, e.g. creation and termination of processes, while the scheduler is responsible for decisions concerning resources sharing, with the ability to implement various scheduling policies and utilize hardware performance counters. In this work, however, scheduling policy is not important since all tests conducted involve a single process on each core. Thus, there is no need for cpu time-sharing and context switching.

The executor keeps information about the programs executed and events during execution-time and stores data from the scheduler's output, programs' output, error messages and performance counters data. For each co-execution test, the executor is run, with given arguments a configuration file, an output folder, the set of CPUs and scheduler to be used. The configuration files contains the executables' paths, along with information about execution, such as the number of cores needed for each one, its place in the execution group and other scheduler-decision related parameters (e.g. starting time if a delay is desired), most of which do not concern the current work.

As mentioned before, Scaff also provides the infrastructure to extract performance data for each process. Apart from a number of fixed counters (unhalted cycles, instructions retired), users can modify the scheduler code to take advantage of additional counters from the set provided by the architecture. Scaff stores performance data from the counters in approximately 1 sec. intervals in a counters file, with the corresponding PID and execution time elapsed.

3.2.2 System tools and mechanisms used

Scaff uses Linux kernel’s cgroups and cpusets subsystems. Cgroups (Control groups) provide a mechanism for creating sets of tasks with specialized behavior, in hierarchical organized groups [17,21]. A cgroup is a set of tasks with common execution parameters, such as resource limitations. User can control execution of processes belonging in a cgroup by editing the specific group’s configuration files. 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, which includes files that contain information about this cgroup instance and the subsystems associated with it. Userspace code can define behavior 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 a task is to be executed in CPU 1, we can write its PID in tasks and value 1 in cpus.

Cpusets use the generic cgroups subsystem [18,21] and constrain the execution of tasks to a set of cpus and memory nodes. Cpusets can be created and deleted from user-space, as they are using the cgroups virtual filesystem. This mechanism can restrain selected processes not only in which CPUs they are allowed to use, but also in other parameters, such as memory nodes. This is achieved by filtering system calls made by these processes; a task will not be scheduled on a CPU that is not allowed in its cpus_allowed vector, and the kernel page allocator will not allocate a page on a node that is not allowed in the requesting task's mems_allowed vector.

Scaff allocates a structure for every program (aff_prog_t) which will be used to store the program’s information during execution, also containing a pointer to the shared memory used by the executor to communicate with the process. This structure also contains a cpuset field which serves as a handler for the program’s cpuset. The executor uses the fork system call to start execution. Scaff creates a new cgroup in the cpusets filesystem and attaches the program to it.

Additionally scaff initializes signal handlers; signals important for the executor are SIGCHLD and SIGTERM. The latter indicates unexpected termination of the program and is handled as an error (e.g. stopped with a SIGKILL). A normal termination of a running task is indicated by SIGCHLD.

3.3 Benchmark program

Our goal was to create a benchmark to test the various levels of the memory hierarchy. The idea was to test both sequential access (which takes advantage of hardware data prefetchers, if any available, and cache re-use) and random access pattern, to limit the gains of prefetching and cache usage. It is based on continuously accessing the structures for a user given number of iterations. The program is implemented in C, all executables compiled with gcc's O3 optimization flags.

3.3.1 Design

<u>Data structures used:</u>

The benchmark utilizes a user-defined number of linked lists (up to 8 implemented, but very easily extendable to any desired number) for both sequential and random access. The purpose of that is to have more than one independent access requests in each loop, in order to exploit instruction level parallelism (ILP).

At this point, it may be noteworthy to mention the various structures we experimented with, some of which are used to create the ones we finally chose. The benchmark was initially using arrays instead of lists, but the latter were found to offer much more consistent performance. The basic unit used as a starting point is a sequential access array, a simple single-dimensional array of unsigned integers, in which each element contains the position of the next element to access. In our case a[i] = i + 1, imitating a pointer to the subsequent memory element. Structure initialization and access subroutines are as follows:

```
create_sequential_access_array (integer: size)

        allocate array[size] a of long integers//contains elements a[0]…a[size – 1]
        for i ← 0 to size – 2
                a[i] ← i + 1
        a[size – 1] ← 0                    //last element value points to start

        return pointer to a
        end
```

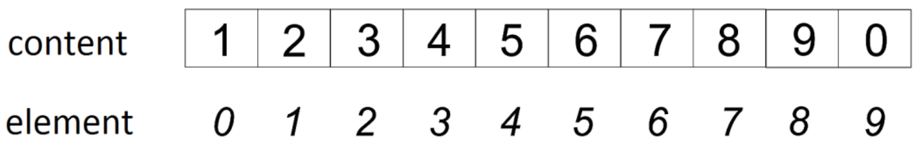


Figure 3.5: Example array with 10 elements

```
access_sequential_array (array:a)

        temp ← 0
        for i ← 1 to (size of a)
                temp ← a[temp]
        end
```

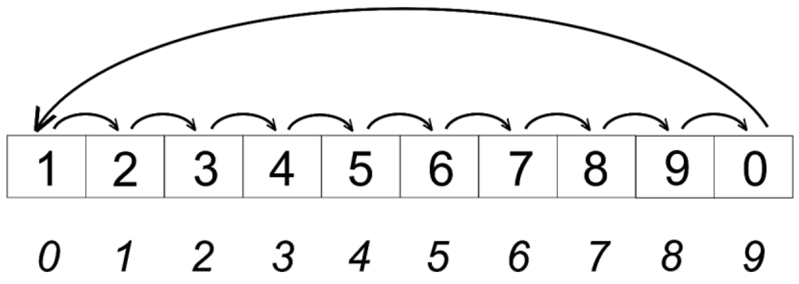


Figure 3.6: Example of sequential array access sequence

For the random access of elements, the numbers in the sequential access array need to be randomly permuted so that each element points to a random element to be accessed next. The Fisher – Yates permutation algorithm [22] is used, along with Galois linear feedback shift register (LFSR) for randomization. The LFSR produces a pseudo-random sequence with a very long cycle, making it suitable for use with large numbers of elements. Fisher – Yates shuffle algorithm, if provided with an unbiased random sequence, will produce an unbiased, random permutation of a finite set. It is ensured that each element is only repeated once and that the access is cyclic, all elements are accessed at the end of an iteration.

```
create_random_access_array(integer: size)

        allocate array a of size        //contains elements a[0]…a[size-1]
        for i ← 0 to size – 1
                a[i] ← i + 1
        fisher_yates_permute(a)

        return pointer to a
        end
```

```
fisher_yates_permute(array:a)
        for i ←((size of a) – 1) to 1      //with decrement step 1
                j ← galois_LSFR mod i
                swap (array[i], array[j])
        end
```

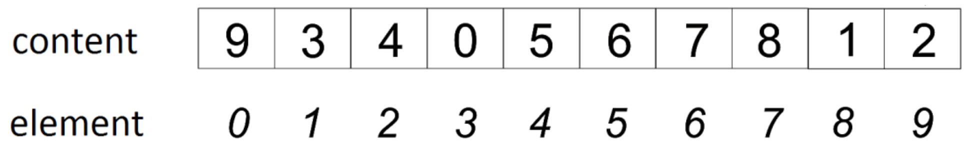


Figure 3.7: Example array after a random permutation of the elements

The array can be accessed in the same way with the sequential:

```
access_random_array(array:a)

        temp ← 0
        for i ← 1 to (size of a)
                temp ← a[temp]
        end
```

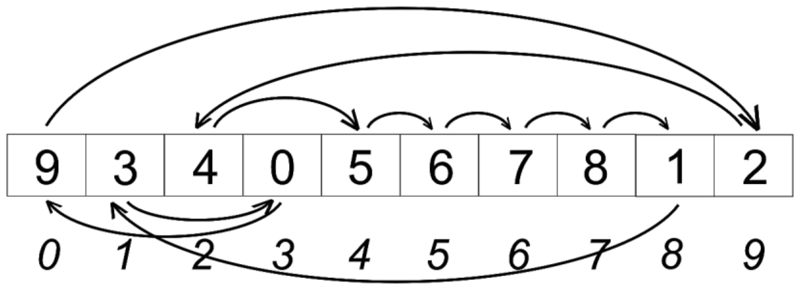


Figure 3.8: Sample of random array access sequence

The aforementioned array types are used, indirectly, for creating the linked lists the benchmark utilizes. The structure used for the list elements consists of 7 long integers forming an array of 56 bytes and a pointer (another 8 bytes) to the next element. Thus, it is ensured that each element has a total size of 64 bytes (the size of a cache line) and for every new access a new line must be loaded to the cache. The 7 long-integer elements, apparently, do not contain any useful data and are serving as the payload of the structure.

```
Struct L:

    next: pointer to struct l element
    pad: array[0..7] of long integers
```

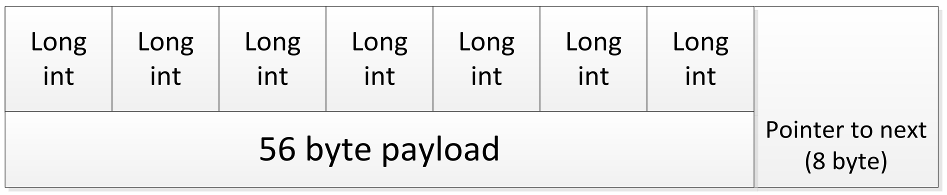


Figure 3.9: List element structure

Using this structure, the sequential access lists are created. The size of the list is variable and is given by the user, depending on which level of the cache hierarchy (or the main memory) is to be targeted.

```
create_sequential_access_list(integer: size)

    allocate array list_elements[size] of type: struct L
    for i ← 0 to size – 2
        list_elements[i].next ← list_elementsi[i + 1];      //next points at the
                                                    element //subsequent in
                                                    memory

    list_elements[i].next ← &list_elements[0];      //last element value points at
                                                    start

    return pointer to list_elements                 //first element
    end

access_sequential_list(pointer to list: a)
    s ← a
    for i ← 0 to (size – 1)
        s ← s.next
    end
```

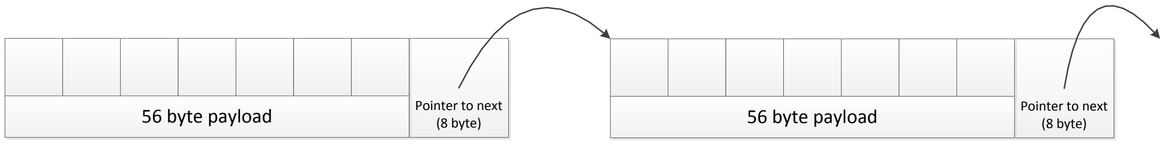


Figure 3.10: accessing consecutive memory elements.

To create the random access list, the idea is randomly permuting the pointers-to-next-element of a same type list, ensuring again that each element needs a separate cache line.

In order to implement a random access list, the same principle is used: A random access array (of size n), basically a randomly permuted list of the numbers [0, n-1], is used to assign the pointers on each element of the list:

```
create_random_access_list(integer: size)

    allocate array list_elements[size] of type: struct L
    permut ← create_random_access_array(size)
    for i ← 0 to (size – 1)
        list_elements[i].next ← list_elements[permut[i]];

    return pointer to list_elements                    //first element
    end
```

Access is the same with sequential list:

```
access_random_list(pointer to list: a)
    s ← a
    for i ← 0 to (size – 1)
        s ← s.next
    end
```

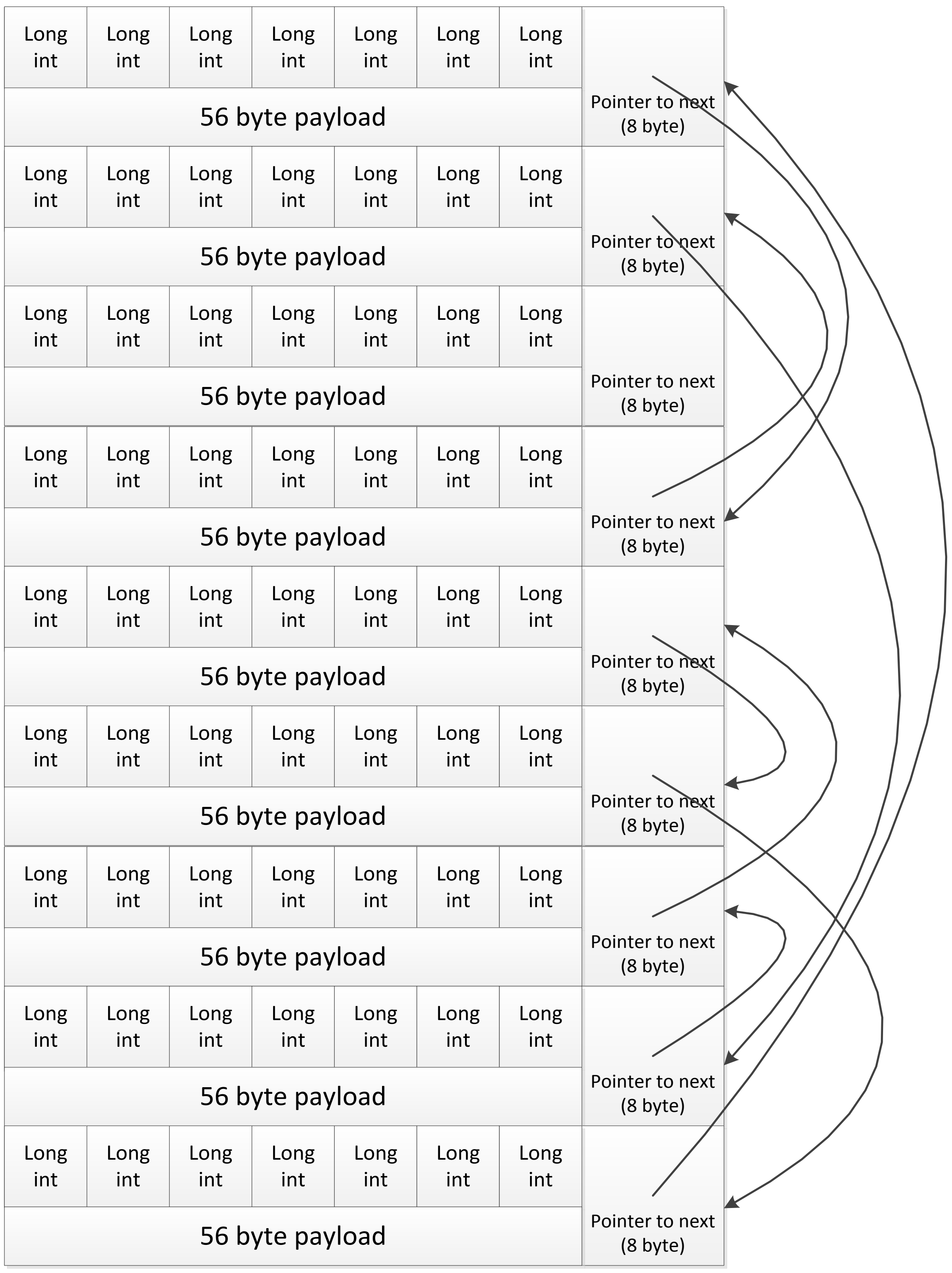


Figure 3.11: Example of a randomly accessed list.

Using the benchmark

The benchmark has a relatively simple approach: The user defines the data size (in kilobytes), number of loop iterations and number of accesses on each loop. The program first initializes sequential access structures according to user input, then it starts accessing the elements as defined previously. This repeated procedure (post initialization) is timed using system time. When finished, the number of iterations, size of data and consequently the total data accesses are known, as well as the execution time for the loop alone. Thus, it is possible to calculate the bandwidth ($\frac{\text{data}}{\text{time}}$ ratio) achieved. The procedure is repeated for random access.

```
benchmark_sequential(iterations, size_inKB)
    size ← size_inKB convert to number of elements
    seq_list ← create_sequential_access_list(size)
    start timing
    for i ← 1 to iterations
        access_sequential_list(seq_list)
    end timing
    output results //Total data accessed, Total time, Bandwidth as calculated from size
and
                    //time
    end

benchmark_random(iterations, size_inKB)
    size ← size_inKB convert to number of elements
    ran_list ← create_random_access_list(size)
    start timing
    for i ← 1 to iterations
        access_random_list(ran_list)
    end timing
    output results //Total data accessed, Total time, Bandwidth as calculated from size
                    //and time
    end
```

Example output: Benchmark with 4MB size and 20000 iterations:

```
4096 KB converted to 65536 elements.

                           Benchmark 1
       Sequential access list of 65536 elements, element size:64
bytes

Number of iterations: 20000      Total data: 80000.0000 MBytes

Total time 7.942 sec             Average rate: 10073.2831 MB/sec


                           Benchmark 2
       Random access list of 65536 elements, element size:64 bytes

Number of iterations: 20000      Total data: 80000.0000 MBytes

Total time 26.147 sec            Average rate: 3059.5720 MB/sec

=================================END==================================
==
```

When accessing more than one structures, the procedure is almost the same. The difference is that we create smaller independent structures, the total size of which is the target size, and each one’s next element is accessed independently in every loop iteration.

```
benchmark_sequential(iterations, size_inKB, no_of_streams)
    size ← size_inKB convert to number of elements
    size ← size / no_of_streams
    seq_list_1 ← create_sequential_access_list(size)
    seq_list_2 ← create_sequential_access_list(size)
    .
    .                   //up to no_of streams lists are created
    .
    start timing
    for i ← 1 to iterations
        s1 ← seq_list_1
        s2 ← seq_list_2
        .
        .
        .
        for j ← 1 to size
            s1 ← s1.next
            s2 ← s2.next
            .
            .           //in each loop no_of_streams elements are accessed
            .
    end timing
    output results //Total data accessed, Total time, Bandwidth as calculated from size
                   //and time
    end
```

The same is applied for the random counterpart. An example output with 4MB, 4 streams (thus meaning 4 streams of 1MB each) and 20000 iterations:

```
4096 KB converted to 65536 elements.

                           Benchmark 3
Parallel sequential access of 4 lists, 16384 elements each, element
size:64 bytes

Number of iterations: 20000      Total data: 80000.0000 MBytes

Total time 3.319 sec             Average rate: 24105.1418 MB/sec


                            Benchmark 4
Parallel random access of 4 lists, 16384 elements each, element
size:64 bytes

Number of iterations: 20000      Total data: 80000.0000 MBytes

Total time 7.798 sec             Average rate: 10259.5394 MB/sec


================================END=================================
==
```

3.3.2 Results and evaluation

The program was used to evaluate the performance of the different memory hierarchy levels of specific architectures, using a variety of configurations, ranging from 1 kilobyte to 128 megabytes and from 1 up to 8 independent streams for each size. This range of target sizes was chosen to demonstrate how the performance alters when gradually moving from a dataset that can fit in a fraction of the L1, to sizes much larger than the last-level-cache (LLC), where the program has to continuously access the main memory.

The number of iterations for each test was decided accordingly in order to achieve a running time of at least 5 seconds, in almost all cases more than 6 sec. The purpose of that was to minimize the effects of possible random factors during execution by providing a relatively long running time, and a large number of iterations to provide better statistical sample quality, also eliminating the impact of initial data fetching penalties, which will occur on the first cycles. Each benchmark (with the same parameters: size, iterations and number of streams) was also executed 2 to 4 times, so that possible performance inconsistencies would be easier to detect.

3.3.2.1 Intel Sandy Bridge architecture

The first processor architecture on which it was tested was Intel's Xeon E5-4620 Sandy Bridge with the following features, as mentioned before in this chapter: 8 hyper-threaded cores per package, 32 KB data + 32 KB instruction, 8-way associative Level 1 cache per core, 256 KB 8-way associative Level 2 cache per core, 16 MB, 16-way associative Level 3 cache per package and hardware data prefetching both for main memory and cache. It should be noted that the benchmark, however, only utilizes a single core (or a single thread, in hyperthreaded architectures as this one), in order to be able to measure per-core-performance and be independently executed on more cores to study the behavior when contention is caused on shared resources (e.g. LLC, Memory bus usage), which is the objective of this thesis.

Average per core data bandwidth for sequential access measured by the benchmark is shown on table 3.3, while table 3.4 contains results for random access. Actual performance, of course, may vary from 0.01% up to ~4-5% for the same tests repeated more times, but this is expected, as the tests were run in user-level within a normally running Linux OS. It is important to state that our objective is to study the order of magnitude of achievable bandwidth in normal runtime context, and not theoretical maximum or high precision. For these reasons, the presented results are considered satisfying, focusing on the 3-4 most significant digits.

The expected behavior of the tests is described below:

- For sizes less than the size of L1 cache (32 KB), bandwidth should be higher, as the data can fit in the L1 cache, which is the fastest in the hierarchy. It was also expected to be increasing even more as the number of streams is increased, utilizing ILP. As long as all data is loaded on L1 cache, no noticeable difference should be noticed between sequential and random access patterns.
- For more than 32 KB and less than 256 KB, significant performance degradation is expected, as the data can no longer fit in L1 and the slower L2 must be used. Again,

because of ILP, increasing the number of streams should increase bandwidth. As mentioned before, since all data has been fetched to the cache, no major difference between sequential and random access should occur.

- As data size gets close to and exceeds 256 KB, the L2 is not sufficient to hold it without having to use L3. Thus, a decrease in bandwidth is expected as data has to be fetched from L3, replacing higher level cache lines each time. Increasing the number of streams accessed should, again, increase throughput, at least to the limit up to which the LLC cache can perform.
- When data size verges the L3 size (16 MB), the program should start accessing data using the memory bus alongside the cache, and so slowing down even more. At larger sizes when continuous cache line replacements are expected, the sequential benchmark should very noticeably take advantage of the hardware prefetchers, while the random part should be excessively slowed down because of higher LLC-miss penalties.

Results of the benchmark for sizes corresponding to the above remarks are shown in the tables below, 3.3 for sequential access and 3.4 for random access pattern:

Sequential Bandwidth (in MB/s)									
Size (KB)	**Number of streams**								
		Single	2	3	4	5	6	7	8
16		42,257.9	79,032.8	111,928.4	141,378.1	168,441.0	192,163.0	200,749.1	225,497.5
32		40,893.9	72,536.5	103,288.7	142,248.8	153,478.4	183,356.6	184,678.0	230,070.3
38		13,168.2	26,134.6	37,675.5	38,319.8	59,253.4	59,788.2	63,556.0	68,606.7
45		13,091.2	25,948.5	37,634.7	45,587.1	55,498.9	58,037.8	62,757.9	65,374.2
128		12,810.3	25,405.3	36,676.9	44,752.7	46,074.7	57,366.0	60,664.5	58,034.6
200		11,992.1	23,380.9	29,944.9	32,800.1	38,179.0	47,494.6	49,717.1	48,382.3
256		11,089.4	21,667.7	27,333.7	23,663.3	34,515.6	38,685.5	42,360.0	45,586.5
384		10,304.4	16,665.5	21,585.7	19,678.0	27,860.9	26,681.2	28,493.6	29,630.9
3,072		10,099.3	16,095.4	20,482.3	24,226.6	26,017.0	26,692.8	27,154.0	27,317.3
10,240		9,774.7	15,953.6	20,102.7	22,911.3	24,742.0	25,227.1	25,450.1	26,582.5
13,302		9,428.1	15,224.2	18,708.2	19,954.8	20,670.3	23,442.2	23,896.6	24,252.0
16,384		7,760.6	12,880.6	16,693.7	18,122.9	18,471.1	18,191.8	18,163.4	17,991.0
20,480		6,916.9	11,802.3	14,752.6	15,111.1	15,156.8	14,889.2	14,765.5	14,703.5
32,768		6487.106	11206.05	13423.85	12957.71	13042.74	12777.82	12595.26	12525.88
131,072		6521.119	11024.08	13445.06	12820.52	12818.46	12605.11	12416.24	12368.46

Table 3.3

Random Bandwidth (in MB/s)									
Size (KB)	Number of streams								
		Single	2	3	4	5	6	7	8
16		42,259.6	79,036.2	112,915.3	141,370.5	162,941.3	192,502.7	204,920.5	225,528.0
32		40,894.8	72,175.9	106,969.6	134,536.1	152,955.7	187,909.6	202,198.3	211,157.9
38		13,938.6	27,365.4	40,808.6	39,143.1	59,489.0	66,377.3	71,454.7	73,594.3
45		13,670.3	26,685.6	38,255.9	44,201.3	53,855.8	59,221.2	63,471.4	65,960.4
128		13,338.1	25,752.9	32,380.9	45,031.1	51,441.2	57,905.7	60,786.1	60,495.0
200		10,380.3	19,810.2	26,245.2	26,315.0	41,984.5	42,492.3	47,652.4	50,005.4
256		5,159.4	12,021.0	17,085.2	15,610.3	28,707.0	27,694.8	37,751.7	33,450.9
384		3,981.8	7,385.6	10,996.7	12,149.1	17,727.0	20,901.1	23,281.7	26,585.3
3,072		3,176.7	6,141.9	8,869.7	11,299.8	13,396.5	15,162.7	16,488.7	17,100.0
10,240		2,466.5	5,178.4	6,853.5	7,141.6	7,419.7	7,672.1	7,656.8	7,589.6
13,302		2,663.9	5,052.8	6,535.2	7,061.1	7,080.6	7,105.2	6,947.2	6,981.9
16,384		1,137.5	2,169.8	3,349.6	3,947.2	4,892.8	5,460.4	5,982.0	6,235.1
20,480		810.1	1,528.6	2,296.8	2,976.0	3,670.4	4,271.7	4,830.2	5,320.4
32,768		647.8	1,267.9	1,873.9	2,449.4	3,032.1	3,582.1	4,092.2	4,542.0
131,072		589.5	1,154.6	1,691.2	2,203.5	2,625.0	2,908.9	3,032.1	3,062.2

Table 3.4

Full benchmark data for all dataset sizes tested are presented in Appendix. A graphical representation of the data can be seen in figures 3.12 (sequential) and 3.13 (random). Data size is on the horizontal axis and bandwidth on the vertical.

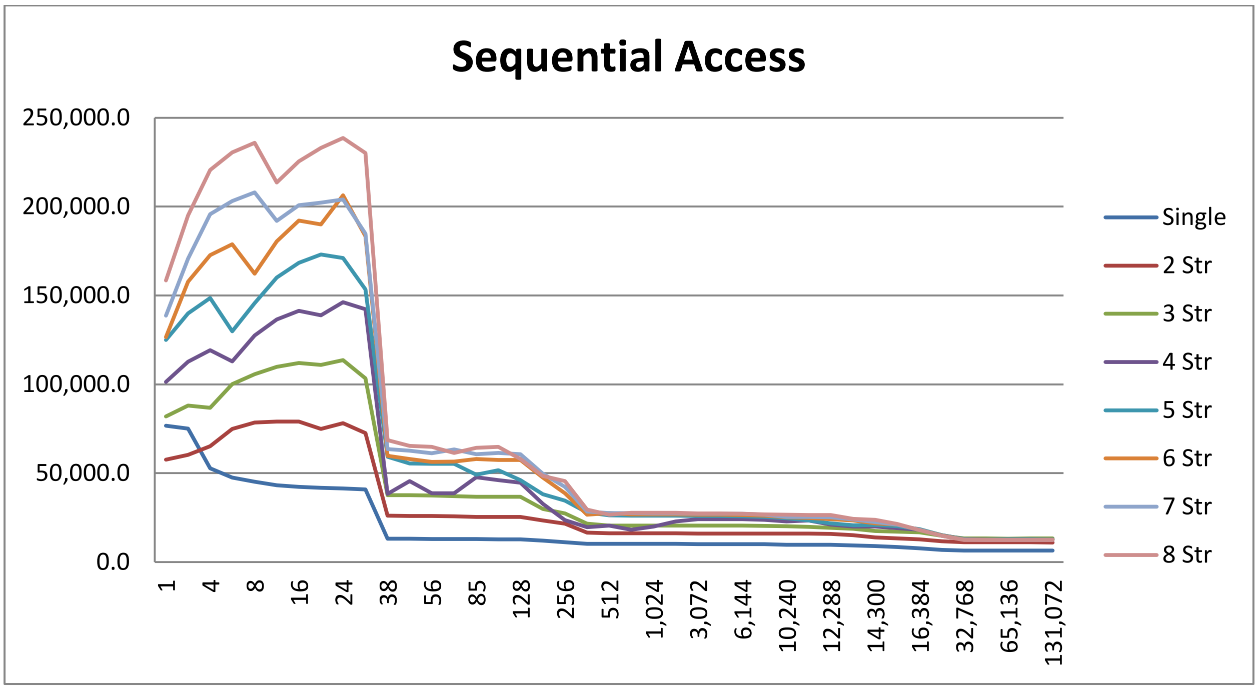


Figure 3.12

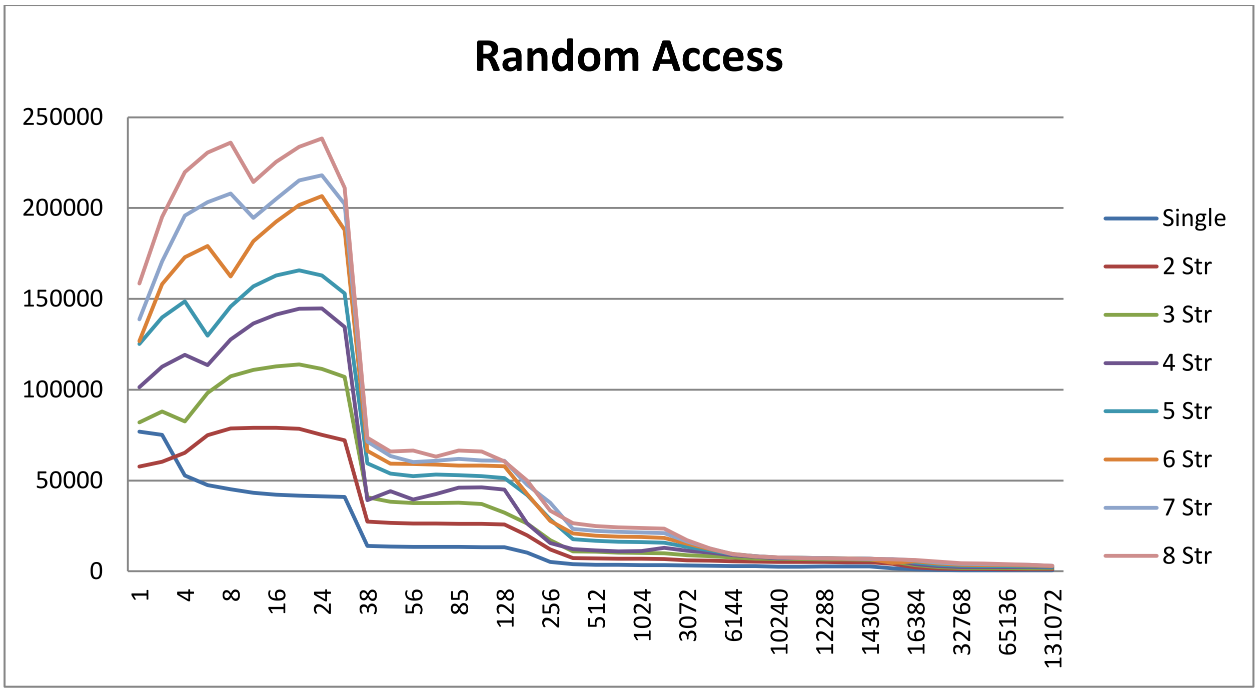


Figure 3.13

The actual results, as seen above, largely confirm the expected behavior, described previously, but also lead to additional interesting conclusions.

For tests up to 256 KB, each additional data stream increases bandwidth noticeably for both random and sequential access patterns. Also, performance is almost the same, for a given number of streams, for both patterns, with the random being marginally faster in many cases. This could be explained, given that the sequential access will make a constant number of misses, fetching and replacing data serially, while the randomized one may take advantage of elements fetched on a previous miss that weren't accessed nor replaced, and thus happen to be in the cache when the program needs to access them. However, these performance differences are extremely low.

Performance degradation, when approaching the limits of a cache level, is very noticeable when reaching 32 KB, 256 KB and 16 MB for sequential access. For random access, there is an additional point of noticeable throughput drop: when exceeding 2.5 MB and this can be explained as in the specific architecture, despite having a 16MB L3 shared among all processors of the package, maximum per core L3 is limited to 2.5 MB [20], resulting in relatively higher access times for other parts of the cache. However, the sequential counterpart is not affected, taking advantage of cache level prefetching.

For larger sizes than 128 MB, already 8 times the size of the LLC, bandwidth achieved remained constantly in the same levels, thus further results are omitted. It also becomes apparent that for large data sizes (bigger than LLC) increasing the number of streams more than 3 starts to cause slight decrease in performance for sequential pattern, indicating that contention on the memory bus starts to occur with the concurrent accesses. Between 1 and 2 streams, however, a direct doubling is noted, before achieving the maximum with 3 streams, interestingly demonstrating the performance gains of ILP. Similarly, the random pattern seems to constantly gain 0.5 GB/s for each stream up to 4, where contention on memory link seems to begin. These observations suggest the benchmark's predictable behavior, making it suitable for co-scheduling study, which is the objective of the current thesis.

To evaluate the results, "STREAM" benchmark was used for sequential access, and "pChase" benchmark for random patterns.

STREAM:

STREAM [14] is a benchmark program, designed to stretch the memory bus of a multicore system and measure its maximum sustainable bandwidth, by making streaming memory accesses. STREAM, is widely used as a standard for large-SMP systems bandwidth measurement. It implements a kernel that accesses 3 single-dimensional arrays of double-precision floating point elements, much larger than the LLC. The access pattern ensures each request has to access the main memory, eliminating cache re-use, a concept quite similar to the benchmark program described in this chapter. STREAM takes advantage of multicore architectures by running a thread on each core, in order to maximize memory bus utilization. Therefore, to compare it with the present results, it was needed for STREAM to be limited to a number of cores, in the same physical package since, as stated before, our benchmark is single threaded, running on only one core.

The best bandwidth performance achieved with STREAM running on 1, 2, 3 and 4 cores respectively is presented below, in table 3.5.

Bandwidth in MB/s				
	1 Core	2 Cores	3 Cores	4 Cores
Best	12,012.9	14,253	14,234.5	14,166
Size	2.2 GB	2.2 GB	2.2 GB	2.2 GB

Table 3.5

Maximum theoretical memory bandwidth per package on the specific system is 18 GB/s. STREAM manages to achieve up to ~14 GB/s when running on 2 or more cores of the package. When on a single core, it achieves 12 GB/s. Comparing it to our benchmark, performance is quite similar: the maximum bandwidth achieved was about 13.5 GB/s, utilizing 3 independent lists (see t. 3.3):

Sequential Bandwidth (in MB/s)									
Size (KB)	**Number of streams**								
		Single	2	3	4	5	6	7	8
131,072		6521.119	11024.08	13445.06	12820.52	12818.46	12605.11	12416.24	12368.46

It should be noted that STREAM uses 3 structures on each thread, thus making its single-threaded instance directly comparable with the 3-stream variation of the benchmark, which seems to achieve slightly increased bandwidth (13.4 GB/s instead of 12 GB/s). Further results of STREAM, as mentioned before, do not exceed 14.2 GB/s, indicating that it is close to the maximum bandwidth that can be achieved from a single process. Additionally, a final test series with STREAM utilizing all eight cores of the package, making use of hyper-threading, resulting to a 16-thread instance using 2.2 GB of main memory, achieved 13445 MB/s, a number that happens to precisely match our 3-stream benchmark instance. Also, running STREAM without thread limitations resulted on a 64-thread instance which measured 50 GB/s total bandwidth on all four packages, but this is mentioned only for the sake of completeness, since it does not fall in the context of the current work.

pChase:

pChase [15] is another memory performance benchmark, which measures performance and latency for various access patterns. It is based on pointer accessing and offers randomized access pattern and adjustable number of accessing threads, suitable to evaluate the random access results given by our benchmark. Table 3.6 contains the results of pChase using the random pattern it provides, limited to single core execution for all numbers of threads. The size of each data chain accessed by the respective thread is decided similarly, e.g. on the 2 MB experiment with 1 thread, it uses one chain of 2 MB, with 2 threads, 2 chains of 1MB etc.

Bandwidth in MB/s				
Size	1 Thread	2 Threads	3 Threads	4 Threads
128 KB	13,614	13,516	13,855	27,631
256 KB	10,490	13,530	13,661	13,650
1 MB	8,271	8,590	8,906	10,112
2 MB	8,412	8,454	8,396	8,526
3 MB	8,230	8,418	8,403	8,362
6 MB	8,109	8,219	8,375	8,371
12 MB	6,321	8,059	8,094	8,148
16 MB	4,100	7,342	7,753	7,748
32 MB	2,617	3,022	5,051	4,971
128 MB	2,582	2,561	2,570	2,571

Table 3.6

Same sizes results from table 3.4:

Random Bandwidth (in MB/s)									
Size (KB)	**Number of streams**								
		Single	2	3	4	5	6	7	8
128		13,338.1	25,752.9	32,380.9	45,031.1	51,441.2	57,905.7	60,786.1	60,495.0
256		5,159.4	12,021.0	17,085.2	15,610.3	28,707.0	27,694.8	37,751.7	33,450.9
1,024		3,511.0	6,889.5	10,096.4	11,238.6	16,100.2	18,861.7	21,470.7	23,936.6
2,048		3,438.2	6,754.8	9,902.0	12,878.3	15,765.5	18,448.3	20,999.1	23,431.1
3,072		3,176.7	6,141.9	8,869.7	11,299.8	13,396.5	15,162.7	16,488.7	17,100.0
6,144		2,934.9	5,553.1	7,601.0	8,881.5	9,404.8	9,559.7	9,490.4	9,410.6
12,288		2,711.5	5,137.0	6,614.4	7,175.2	7,262.5	7,253.5	7,220.6	7,152.4
16,384		1,137.5	2,169.8	3,349.6	3,947.2	4,892.8	5,460.4	5,982.0	6,235.1
32,768		647.8	1,267.9	1,873.9	2,449.4	3,032.1	3,582.1	4,092.2	4,542.0
131,072		589.5	1,154.6	1,691.2	2,203.5	2,625.0	2,908.9	3,032.1	3,062.2

It can be easily observed that the order of magnitude, comparing the results of the two benchmarks, is the same for the sizes tested. However, increasing the number of streams in pChase does not affect performance very noticeably for the vast majority of cases. A closer look indicates that pChase for 128 KB, up to 3 threads, performs similarly to the 1-stream variation of our benchmark, for 256 KB (for all number of threads) it is similar to the 2-stream, up to 6 MB compares with the 3-stream and for the larger sizes it performs close to the 4 or more streams. Additionally, pChase performance remains constant for all sizes in the LLC range (less than 16 MB) for all number of threads. On the contrary, our benchmark's performance constantly degrades as size is increased, and increases with additional data streams. Figure 3.14 features comparative performance of pChase and instances of our benchmark using 1 up to 5 streams. Since no significant differences are noticed between pChase thread numbers, average values were taken for the graph. It becomes apparent how similarly the two programs perform.

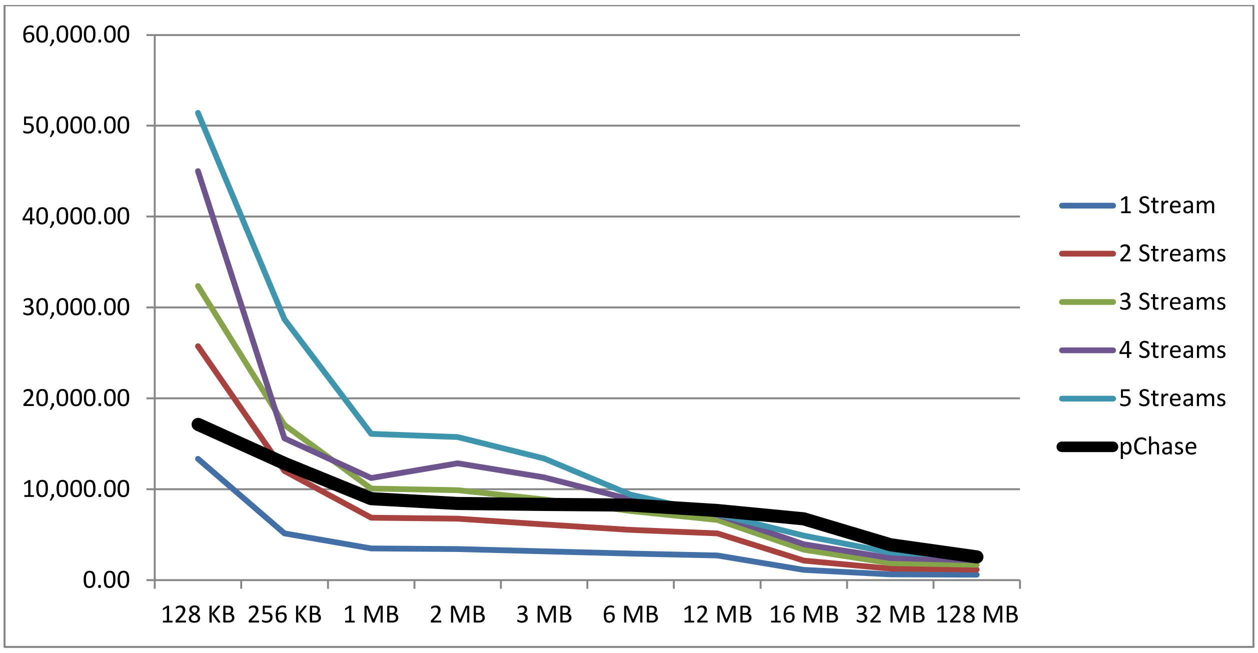


Figure 3.14: Comparison with pChase

Once more, these observations suggest that the benchmark presented in this chapter has a very predictable and consistent behavior and gives the user versatility to create a wide range of contention levels.

3.3.2.2 Intel Dunnington architecture

The same series of tests were also conducted on Intel's Xeon X7460 "Dunnington". This architecture, as mentioned before, features six cores (non hyper-threaded) per package, 32 KB instruction + 32 KB data L1 cache per core, 3 MB L2 cache, shared between core pairs, 16 MB L3 cache per package, no hardware prefetching.

The expected behavior of the benchmark remains mostly as described for the Sandy Bridge processor, with performance being decreased when reaching the sizes of the L1, L2 and L3 caches. Because of the absence of memory bus hardware prefetchers, performance of sequential pattern for larger data sizes is expected to be decreased and, accordingly, the difference between random and sequential patterns should be reduced. Results for both patterns are shown below, in tables 3.7 and 3.8, while complete results tables are in Appendix.

Sequential Bandwidth (in MB/s)									
Size (KB)	**Number of streams**								
		Single	2	3	4	5	6	7	8
16		53,282.4	77,413.5	116,444.4	127,305.5	132,916.4	136,191.7	138,461.6	140,195.9
32		53,640.4	79,141.1	93,761.1	122,190.1	114,750.3	121,475.2	118,081.8	138,596.5
38		13,168.2	26,134.6	37,675.5	35,586.3	39,896.1	42,419.5	40,808.9	38,276.1
45		12,849.7	23,435.8	31,865.3	35,543.2	39,933.6	42,358.4	37,494.2	34,623.1
256		12,838.8	23,652.3	31,918.5	35,868.0	40,236.5	42,616.5	37,531.2	33,539.3
768		12,837.7	23,635.6	31,959.5	36,366.9	41,972.8	42,868.5	37,556.2	33,580.0
1,024		12,835.8	23,624.5	25,506.2	36,377.2	41,624.3	42,893.7	37,327.5	34,569.4
2,048		11,725.2	22,623.2	28,174.8	31,647.0	37,097.4	35,768.5	34,538.6	32,772.9
3,072		9,976.5	16,047.5	18,204.5	19,364.6	20,993.3	20,661.0	21,063.4	21,384.5
4,096		8,531.1	11,846.9	11,967.0	12,442.9	12,342.4	12,805.5	12,772.7	13,015.0
6,144		7,948.8	9,570.3	9,907.9	10,057.8	10,153.1	10,098.3	10,143.3	10,146.3
12,288		5,381.6	6,317.1	6,453.9	6,432.2	6,621.7	7,577.4	7,718.1	8,130.4
16,384		3,030.4	3,486.6	3,609.2	3,791.9	3,802.9	3,760.9	3,998.7	4,193.2
20,480		2,078.5	2,629.0	2,650.4	2,737.9	2,728.3	2,745.0	2,703.3	2,691.0
131,072		1,872.7	2,322.8	2,362.5	2,386.3	2,429.0	2,422.6	2,422.9	2,363.0

Table 3.7

Random Bandwidth (in MB/s)									
Size (KB)	**Number of streams**								
		Single	2	3	4	5	6	7	8
16		53,281.5	77,555.2	115,802.4	128,106.4	132,928.9	136,191.7	138,453.9	140,197.1
32		53,634.6	79,147.2	96,329.2	114,466.4	120,876.0	122,764.6	125,908.8	128,006.2
38		13,938.6	27,365.4	40,808.6	39,606.4	47,208.8	41,838.2	41,715.1	38,784.2
45		10,894.2	21,355.7	32,016.7	40,604.9	48,372.0	45,322.8	38,097.5	36,427.4
256		9,826.0	19,558.1	29,200.6	38,639.6	43,788.0	38,544.6	37,047.8	35,683.8
768		9,625.2	19,220.3	28,775.8	37,745.9	43,024.9	38,209.8	36,693.1	35,438.9
1,024		9,599.0	19,126.4	28,421.8	37,544.5	42,429.3	36,947.8	34,475.0	33,625.4
2,048		7,167.0	11,504.5	18,523.7	21,323.7	23,443.3	22,949.5	26,250.9	25,890.6
3,072		3,778.5	7,475.8	9,132.6	11,340.2	13,203.9	14,974.1	16,037.1	16,934.3
4,096		2,515.1	4,849.1	6,773.6	8,576.0	9,836.0	10,738.8	11,383.2	11,887.9
6,144		1,931.1	3,683.4	5,285.9	6,554.0	7,512.0	8,348.9	9,107.9	9,545.6
12,288		1,495.2	2,369.9	2,691.9	5,414.4	5,813.0	6,902.7	5,891.8	6,383.1
16,384		928.3	1,582.1	2,257.1	2,481.7	2,894.9	3,314.3	3,582.2	3,777.4
20,480		643.2	1,236.3	1,560.6	1,939.2	2,190.2	2,345.3	2,463.7	2,571.3
131,072		334.1	609.9	839.2	1,061.2	1,286.5	1,497.7	1,699.4	1,852.7

Table 3.8

A graphical representation of the data can be seen in figures 3.15 (sequential) and 3.16 (random)

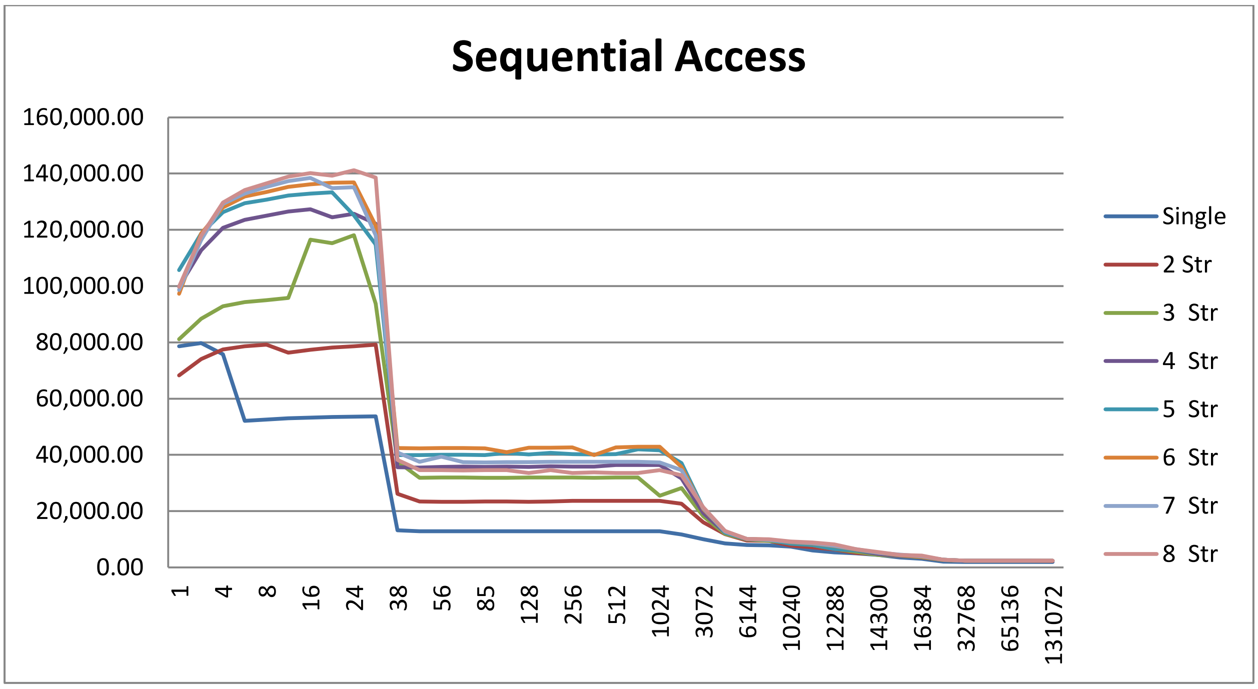


Figure 3.15

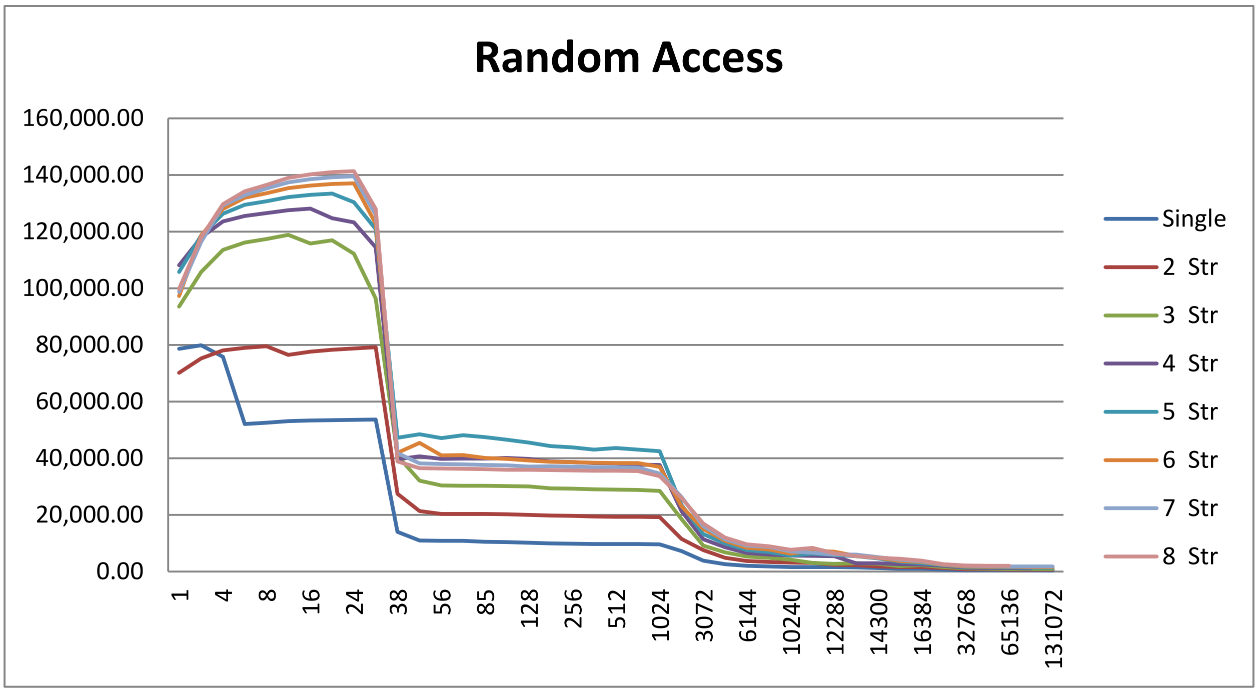


Figure 3.16

Results in this architecture are, again, as predicted above. A huge performance drop is noted when data size exceeds 32 KB (L1), followed by a second decrease while verging L2 size. While being on the LLC size range, bandwidth continues to gradually decrease, for both access patterns, until it finally gets stable for the largest size instances. As expected, the lack of hardware prefetching results in much smaller differences between sequential and random access, compared to the equivalent Sandy Bridge results.

Effects of ILP when increasing the number of data streams accessed are noticeable, similarly to the Sandy Bridge Xeon, with a most notable example the steady 0.2-0.25 GB/s gain for each additional stream on the largest sizes for the random access pattern.

<u>STREAM:</u>

Once again, STREAM was used to evaluate measured bandwidth and performance of the benchmark on this architecture. Bandwidth measured with STREAM is shown in table 3.9

Bandwidth in MB/s				
	1 Core	2 Cores	3 Cores	4 Cores
Best	2,336	2,580	2,602	2,613
Size	2.2 GB	2.2 GB	2.2 GB	2.2 GB

Table 3.9

As seen above, performance of the two benchmarks is very similar, both measuring maximum per-core bandwidth at 2.4 GB/s. The corresponding entries of table 3.7 follow:

Sequential Bandwidth (in MB/s)									
Size (KB)	**Number of streams**								
		Single	2	3	4	5	6	7	8
131,072		1,872.7	2,322.8	2,362.5	2,386.3	2,429.0	2,422.6	2,422.9	2,363.0

For completing the overall picture, it should be mentioned that maximum per-package bandwidth, as measured by STREAM, is 2.6 GB/s, while overall bandwidth, utilizing all four packages, was found to be 8.7 GB/s.

pChase

As before, in order to evaluate the random access pattern behavior the pChase benchmark was used, running the same series of tests:

Bandwidth in MB/s				
Size	1 Thread	2 Threads	3 Threads	4 Threads
128 KB	11,780	12,493	12,335	35,800
256 KB	13,003	12,443	12,200	11,957
1 MB	13,026	13,022	13,000	13,003
2 MB	10,850	13,013	13,019	13,000
3 MB	5,838	12,556	12,907	12,904
6 MB	3,509	5,311	9,078	11,632
12 MB	2,383	3,395	3,744	5,222
16 MB	1,237	2,795	2,880	3,020
32 MB	807	804	815	811
128 MB	797	798	793	794

Table 3.10

Same sizes results from table 3.8:

Random Bandwidth (in MB/s)									
Size (KB)	**Number of streams**								
		Single	2	3	4	5	6	7	8
128		10,130.8	19,977.3	29,953.1	39,677.8	45,465.5	39,182.1	37,060.0	35,895.1
256		9,826.0	19,558.1	29,200.6	38,639.6	43,788.0	38,544.6	37,047.8	35,683.8
1,024		9,599.0	19,126.4	28,421.8	37,544.5	42,429.3	36,947.8	34,475.0	33,625.4
2,048		7,167.0	11,504.5	18,523.7	21,323.7	23,443.3	22,949.5	26,250.9	25,890.6
3,072		3,778.5	7,475.8	9,132.6	11,340.2	13,203.9	14,974.1	16,037.1	16,934.3
6,144		1,931.1	3,683.4	5,285.9	6,554.0	7,512.0	8,348.9	9,107.9	9,545.6
12,288		1,495.2	2,369.9	2,691.9	5,414.4	5,813.0	6,902.7	5,891.8	6,383.1
16,384		928.3	1,582.1	2,257.1	2,481.7	2,894.9	3,314.3	3,582.2	3,777.4
32,768		427.1	777.3	1,077.7	1,337.7	1,554.3	1,755.2	1,934.3	2,069.0
131,072		334.1	609.9	839.2	1,061.2	1,286.5	1,497.7	1,699.4	1,852.7

Similar to the previous architecture comparison, the bandwidth's order of magnitude for the two benchmarks is the same, with pChase being much less affected by additional data streams and displaying a far narrower range of results. Figure 3.17 graphically shows pChase bandwidth compared to instances of our benchmark

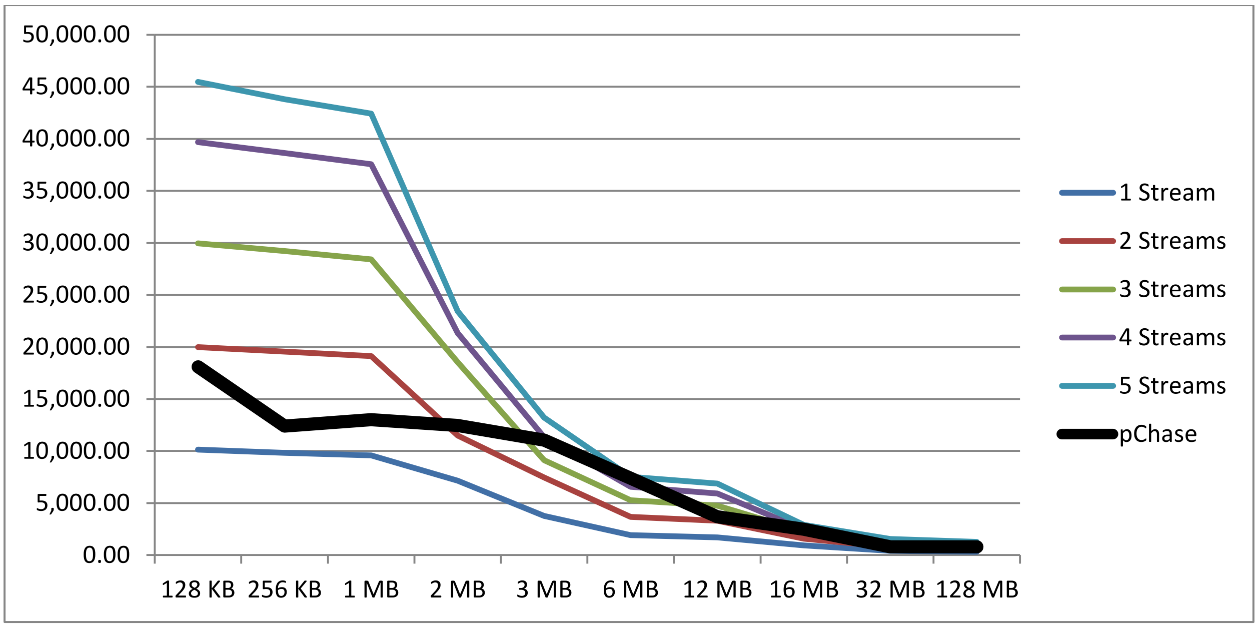


Figure 3.17

3.3.3 Class evaluation with performance counters

Finally, the hardware performance counters (provided by the Sandy Bridge architecture) were used to confirm the benchmark's application class (as described in chapter 2) for the various size instances. Performance counters used for this evaluation were unhalted clock cycles counter, instructions retired counter, per-core Bandwidth counter, L2 lines counter (data from L3 to L2), L1 lines counter (data from L2 to L1), LLC misses counter, memory micro-operations counter (mem_uops) and the total micro-operations (all_uops) counter. Mem_uops counts retired micro-operations (loads and stores) to any part of the memory hierarchy, while all_uops is total number of micro-operations. Usage of all other performance counters is explained in chapter 4. Ratio $\frac{\text{mem\_uops}}{\text{all\_uops}}$ shows if an application is memory-bound.

Table 3.11 shows the memory micro-operations ratio for a wide range of data sizes. (For the last instances, the ones with the highest bandwidth were chosen, to maximize contention, thus 3 streams instance for the sequential pattern, and 8 streams for the random respectively).

Memory micro-operations ratio							
Size (KB)	Pattern		1 Stream	2 Streams		4 Streams*	
24	seq.		0.331	0.426		0.551	
24	random		0.331	0.426		0.551	
204	seq.		0.333	0.428		0.555	
204	random		0.333	0.428		0.555	
3072	seq.		0.333	0.429		0.556	
3072	random		0.334	0.429		0.556	
13107	seq.		0.333	0.429		0.555	
13107	random		0.333	0.428		0.554	
122880	seq.		0.332	0.428		0.499*	*(3 Streams)
122880	random		0.333	0.427		0.686**	**(8 Streams)

Table 3.11

As expected, memory micro-operations ratio is only affected by the number of accesses (streams) and not the size of datasets. It can also be observed, that for all cases, ratio is over 0.25, confirming this benchmark is a memory-bound application, since all it does, after dataset initialization, is accessing datasets.

To decide the application class of each instance, the decision tree described previously was used (figure 3.18) with its parameters as follows: α = 7 GB/s, stands for 50% of maximum memory bandwidth (in our case 14 GB/s), β = 0.35 GB/s is 2.5% of maximum memory bandwidth, γ = 2 GB/s, δ = 0.25, ε = 1, as maximum IPC of this system is 4.

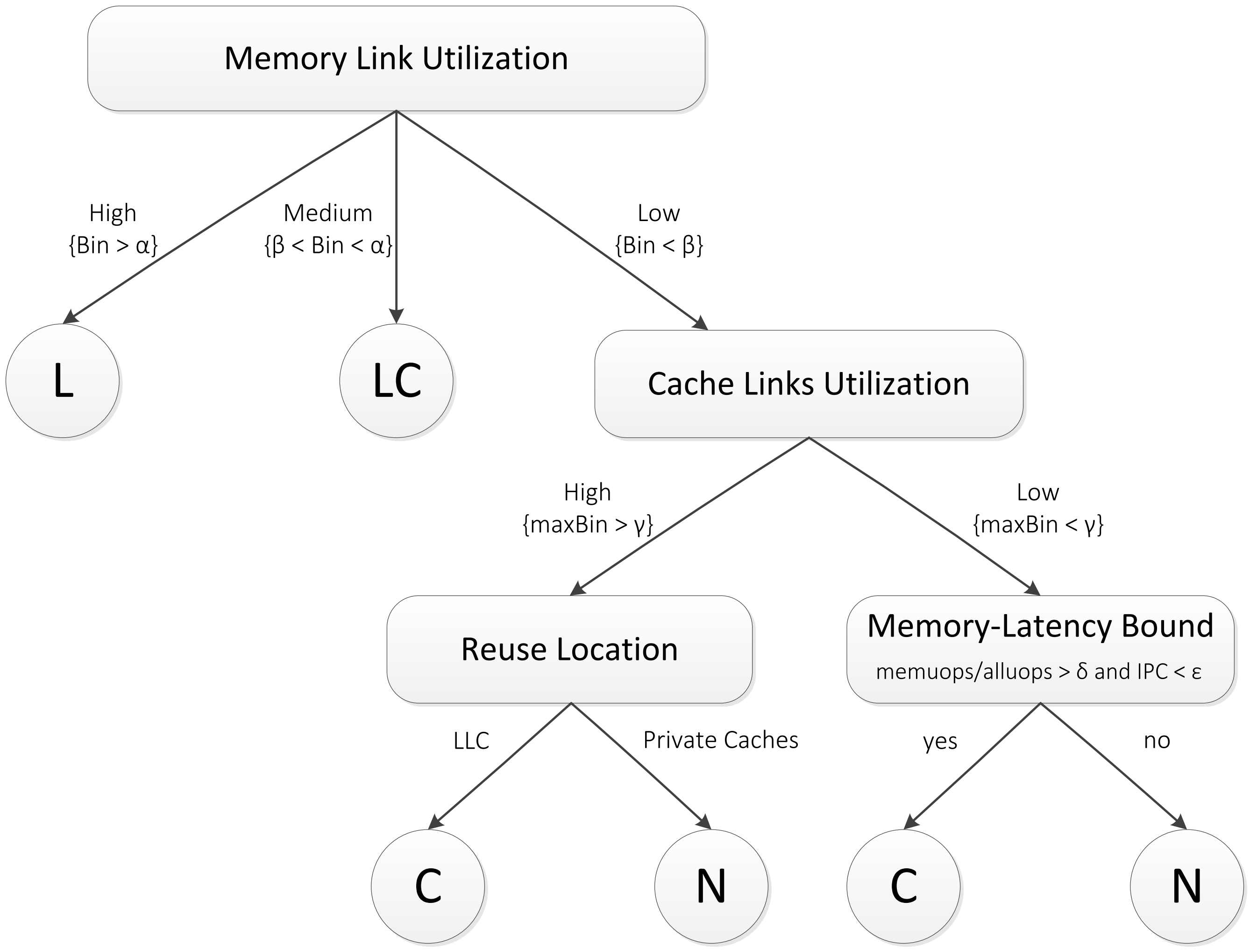


Figure 3.18

Table 3.12 contains profiling metrics for the benchmark, as extracted with the aforementioned performance counters (all bandwidth values in MB/s), for a variety of dataset sizes and number of streams, along with each instance's classification for this architecture in N, C, LC, or L application classes using the classification scheme of figure 3.18. IPC is calculated from the number of unhalted clock cycles and instructions retired, bandwidth caused by LLC misses (LLC miss BW) from LLC misses counter.

Size, No. of streams	**Pattern**	**IPC**	**per Core BW**	**L3 -> L2 BW**	**L2 -> L1 BW**	**LLC miss BW**	**Class**
204 KB, 1 Stream	sequential	0.322	0	7.64	13061.92	0	**N**
204 KB, 1 Stream	random	0.3352	0	9.77	13617.21	0	**N**
204 KB, 2 Streams	sequential	0.6335	0	10.24	25705.43	0	**N**
204 KB, 2 Streams	random	0.3594	0	8660.46	14580.69	0	**C**
204 KB, 4 Streams	sequential	0.5238	0	16027.64	33992.71	0	**C**
204 KB, 4 Streams	random	0.445	0	9268.86	28891.07	0	**C**
1.5 MB, 1 Stream	sequential	0.2624	0	10650.38	10650.29	0	**C**
1.5 MB, 1 Stream	random	0.0886	0	3594.82	3594.96	0	**C**
1.5 MB, 2 Streams	sequential	0.4112	0	16639.6	16695.15	0	**C**
1.5 MB, 2 Streams	random	0.1737	0	7049.35	7049.78	0	**C**
1.5 MB, 4 Streams	sequential	0.3873	0	22946.76	25168.58	0	**C**
1.5 MB, 6 Streams	random	0.2376	0	19288.74	19291.63	0	**C**
3 MB, 1 Stream	sequential	0.2603	0	10588.67	10587.49	0	**C**
3 MB, 1 Stream	random	0.081	0	3295.24	3544.97	0	**C**
3 MB, 2 Streams	sequential	0.4073	0	16517.03	16565.68	0	**C**
3 MB, 2 Streams	random	0.1568	0	6376.51	6865.18	0	**C**
3 MB, 4 Streams	sequential	0.3837	0	22729.61	24983.86	0	**C**
3 MB, 4 Streams	random	0.1795	0	11690.68	12575.37	0	**C**
13 MB, 1 Stream	sequential	0.2269	1862.31	9168.62	9168.46	385.48	**LC**
13 MB, 1 Stream	random	0.0428	330.38	1760.18	2536.42	330.4	**C**
13 MB, 2 Streams	sequential	0.3628	3058.99	14639.19	14697.43	524.51	**LC**
13 MB, 2 Streams	random	0.0804	621.71	3266.29	4753.77	621.71	**LC***
13 MB, 4 Streams	sequential	0.3217	3499.8	18861.3	20877.91	730.21	**LC**
13 MB, 8 Streams	random	0.0764	1214.93	7098.76	10352.99	1215.33	**LC**
128 MB, 1 Stream	sequential	0.1677	6753.77	6748.64	6754.1	1358.62	**L****
128 MB, 1 Stream	random	0.0149	604.06	943.6	1165.77	604.07	**LC**
128 MB, 2 Streams	sequential	0.2802	11270.97	11094.01	11272.11	1244.91	**L**
128 MB, 2 Streams	random	0.0293	1184.48	1831.94	2283.66	1184.51	**LC**
128 MB, 3 Streams	sequential	0.2582	13851.04	12401.27	13866.71	1136.57	**L**
128 MB, 8 Streams	random	0.0352	3245.13	5020.1	6259.44	3245.23	**LC**

Table 3.12

Notes:

* This instance could also be classified as C, strictly using the decision algorithm, but its behavior suggests it is more a link-and-cache intensive application.

** Another marginal decision, could also be classified as LC, but it is mostly a link intensive application, thus L is more appropriate.

It can be easily observed that applications of the same class can have different behavior, with the most notable example being class C, which contains a large number of benchmark instances. Applications in this class can have dataset sizes small enough to fit completely in the cache, without being much affected by other programs sharing LLC, but as memory needs increase they become more vulnerable to getting slowed down by other applications. This happens because they need to use a large part of the cache and continuous cache data replacements caused by other programs result in continuous LLC misses with the respective data fetching time penalties, while programs with smaller datasets will have smaller penalties as a result of fewer data replacements. High associativity of LLC (16-way for both architectures) in conjunction with the unpredictable way OS memory management system allocates datasets in memory pages explains why programs with datasets smaller than cache size need to additionally use the memory link. The point up to which no need for main memory utilization occurs is found (by observing and experimentation) at approximately 50% of cache size; programs with datasets smaller can practically fit entirely in the cache, while larger have an –increasing with size– need to use the memory link as well. Utilization of the main memory bus is low when there is no other program destroying cached data, but when contention occurs, program's behavior is forced to change, even into a different class. All these expected effects are to be confirmed and further discussed in chapter 4.

Usage in co-execution

Given all the above observations, the benchmark's results appear to be valid comparing it to other well-known and widely used benchmark programs and, therefore, useful to estimate the memory performance of a single core in a multi-core architecture. Additionally, it is suggested that it can be used to cause variable contention, in all levels of the memory hierarchy. Being solely a single-thread application, it gives users versatility to concurrently execute various configurations on desired cores of the architecture, observing the effects of contention for shared memory resources without being affected by synchronization or other problems of multithreaded applications. Different instances of the program cover all classes in the application classification scheme used in this work. Thus, it is possible to create a suite of programs with different behavior to emulate memory-intensive applications (with various levels of intensity) in order to study the effects of memory contention and application behavior differences caused by it in MCP co-scheduling context.

Page intentionally left blank.

Chapter 4

Co-scheduling experimental evaluation

4.1 Co-scheduling on Intel Sandy-Bridge

4.1.1 Workload profile:

For co-scheduling tests, aiming to observe memory contention and its effects in all memory hierarchy levels, a variety of memory intensive programs with different behavior was deemed necessary. To achieve that, we opted for various configurations of the proposed memory benchmark program, as described in chapter 3. In order to keep running time relatively stable and predictable, as well as enough for providing sufficient performance data using the architecture counters, we chose and tested for each instance a number of iterations that ensures running time for about 1 minute. Data set sizes should be ranging from relatively small, fitting in the private cache of a single core, to much larger than LLC. Specifically, based on observations made and explained previously, we selected five dataset sizes:

- 204 KB: Given that the Sandy-Bridge architecture we used has larger L2 caches (256 KB private), this size was selected to emulate applications reusing only small datasets, but not fitting solely in private caches as low LLC utilization may occur. It has been observed from random pattern benchmark performance that datasets can completely fit in a cache level as long as dataset is smaller than approximately half of cache size (see fig. 3.13). As a result, LLC utilization is additionally expected for these instances. We named this category L2.

- 1.5 MB: Although much larger than the L2 cache, datasets can very easily fit in the 16 MB L3 (LLC), even in the single-core dedicated 2.5 MB of the LLC. However, if another program on an adjacent core is thrashing the cache, contention should become very noticeable, making this size an interesting co-scheduling candidate. This category was named L3s (L3 *small*).
- 3 MB: Benchmarks using this dataset size are excessively reusing LLC but do not need to utilize main memory link at all. Being co-executed with another program with larger datasets, however, it is very likely that it will be noticeably affected and forced to use the memory link as well. Thus it appears as another interesting dataset size for experimentation. This size category was named L3m (L3 *medium*).
- 13 MB: Datasets of 13 MB do not fit in the LLC (due to high associativity, although being less than 16 MB), with low need to constantly use the memory link as well. Co-executing an instance of this size with any other program is expected to alter the behavior of both, and increase the demand for memory link utilization. We named this category L3l (L3 *large*).
- 128 MB: Finally, a size category not fitting in the caches, 8 times larger than LLC, demanding continuous use of the memory link and, potentially, being able to be thrashing the cache for any other program co-executed. We named it MEM (*Memory*).

Having decided different data sizes to be used with sequential and random access patterns, it was also desired to create variation in the memory resources demand for each size and pattern. It was observed, while evaluating the benchmark, that increasing the number of data streams exploits ILP and causes increased benchmark performance and, consequently, increased memory use demand, as is shown by memory micro-operations ratio in table 3.11 and benchmark results (tables A.1.a and b). Thus it was decided to create three variations for each size and pattern combination: 1 stream-, 2 stream- and max stream-instances.

Single stream instances utilize a single data structure and cause less cache replacements, but also have lower performance since only a single element access occurs in each iteration of the inner loop. Additionally 1-stream programs are expected to be experiencing longer delays since they are practically depending on a single memory access; if this access is delayed due to contention in memory resources, the program must only wait (stall) until requested data is fetched, while in more, independent accesses implementations next instructions and, consequently, new memory requests can be issued in this waiting time. Dual stream variations, as explained and shown previously, can display even double performance compared to the 1-streamed and were found suitable for increased -but not maximized- memory resources demand.

For the final set of variations, the objective was to maximize (or at least keep at a high level) memory utilization without causing “self-contention”. As seen in tables with benchmark results, when increasing the number of streams for larger datasets, performance starts to degrade because of contention caused by the programs’ data structures continuously replacing each other in the caches. This, as mentioned before, happens because of the way memory management system allocates data into memory pages, which is unpredictable and -in contrast with an ideal scenario, where all data would be allocated continuously on a page- causes frequent cache replacement conflicts, which increase as the number of structures accessed increases –since they are allocated on different areas of the memory. To avoid this phenomenon, we picked the sequential variation with maximum performance for the 128MB (MEM) category, which was the 3-stream instance. For all other sizes, using 4 streams for the sequential pattern was found sufficient (despite not having strictly maximum performance in all cases), in order to utilize the 4 memory bus channels, provided by the architecture. Four streams are also sufficient for the majority of the random pattern benchmarks max-stream instances, although the previously described “self-contention” does not occur because of the pattern’s random nature. Exception is the 128 MB (MEM), in which we used 8 streams to maximize bandwidth caused by LLC misses, and the 1.5 MB (L3s) instance, in which 6 streams were used, having the maximum combination of bandwidth and bandwidth-per-stream ratio.

Using the benchmark in the aforementioned configurations, we created a suite of 30 programs with the desired characteristics, covering all application classes (see table 3.12), shown below, in table 4.1.1:

Sequential pattern			
Category size	Category name		Variations
204 KB	L2		1, 2, 4 Streams
1.5 MB	L3s		1, 2, 4 Streams
3 MB	L3m		1, 2, 4 Streams
13 MB	L3l		1, 2, 4 Streams
128 MB	MEM		1, 2, 3 Streams

Random pattern			
Category size	Category name		Variations
204 KB	L2		1, 2, 4 Streams
1.5 MB	L3s		1, 2, 6 Streams
3 MB	L3m		1, 2, 4 Streams
13 MB	L3l		1, 2, 4 Streams
128 MB	MEM		1, 2, 8 Streams

Table 4.1.1

For convenience, a standard name formatting was chosen to name each of the instances above, consisting of three parts: category name, number of streams, and access pattern. For example, a 13 MB, 2 stream sequential benchmark is abbreviated as L3l_2Str_seq, while the 128 MB, single stream random access instance is named MEM_1Str_rdm. Workload classification according to preliminary standalone execution is shown in table 4.1.2 below:

Task	Pattern	Class		Task	Pattern	Class		Task	Pattern	Class
L2_1Str	seq	**N**		L3s_4Str	seq	**C**		L3l_2Str	seq	**LC**
L2_1Str	rdm	**N**		L3s_6Str	rdm	**C**		L3l_2Str	rdm	**LC**
L2_2Str	seq	**N**		L3m_1Str	seq	**C**		L3l_4Str	seq	**LC**
L2_2Str	rdm	**C**		L3m_1Str	rdm	**C**		L3l_4Str	rdm	**LC**
L2_4Str	seq	**C**		L3m_2Str	seq	**C**		MEM_1Str	seq	**L**
L2_4Str	rdm	**C**		L3m_2Str	rdm	**C**		MEM_1Str	rdm	**LC**
L3s_1Str	seq	**C**		L3m_4Str	seq	**C**		MEM_2Str	seq	**L**
L3s_1Str	rdm	**C**		L3m_4Str	rdm	**C**		MEM_2Str	rdm	**LC**
L3s_2Str	seq	**C**		L3l_1Str	seq	**LC**		MEM_3Str	seq	**L**
L3s_2Str	rdm	**C**		L3l_1Str	rdm	**C**		MEM_8Str	rdm	**LC**

Table 4.1.2

4.1.2 Experimental procedure

Our objective was to study how application behavior changes when contention on shared memory resources takes place. All programs of the table above were co-executed in pairs in all possible combinations, using adjacent cores of the same package. Scaff's infrastructure was used for co-execution and performance metrics, without a need for specific scheduling policy, since the only requirement was concurrent but independent execution of two applications on two cores, with no cpu-time sharing demand.

The first step was to execute each application of the suite independently on a single core (core 0, first of the package). This procedure let us store performance counters data of each benchmark instance running alone, as a reference point for comparison with the respective data from co-execution. Performance data collected shows the application's behavior in terms of execution and memory resources usage. IPC (instructions per cycle), a very important index for a program's execution, can be extracted from the performance counters as the ratio of instructions retired counter and unhalted clock cycles: $\frac{\text{instr\_ret}}{\text{unh\_clk\_cls}}$. Higher IPC generally means an application is utilizing more CPU resources, staying in private parts of the core with little or no interaction with memory resources (e.g. calculation intensive), as memory operations are slowing the program down even in the L2. Thus, all of the benchmark instances have IPCs much lower than 1, as presented already in chapter 3. Other counters used are:

- LLC misses: From this number we can measure bandwidth caused by LLC misses, as each miss means fetching a new line (64 bytes) into the cache. Bandwidth in Bytes/sec is $\frac{\text{llc\_miss}}{\text{T}}64$ where T is the period of time during which the misses were measured.
- L1 lines: This counter show how many lines were transferred from L2 to L1 cache for a time period T. Knowing line size (64 bytes) and T, we can calculate bandwidth used between L2 and L1 as $\frac{\text{l1lines}}{\text{T}}64$ (in B/s).
- L2 lines: Similarly, this is the number of lines transferred in L2 from the LLC in time T. Thus, L3 to L2 bandwidth is $\frac{\text{l2lines}}{\text{T}}64$ (in B/s).

- Per core bandwidth: Data (in 64B lines) fetched from the memory bus, from the requests and misses of the specific core in time T. It can also be noted as L3 bandwidth and is calculated as $\frac{\text{per\_core\_bw}}{\text{T}} 64$ in B/s.
- Bandwidth: This is total socket bandwidth (again in lines), for all cores of the package in time T. It is similarly converted in B/s via $\frac{\text{bandwidth}}{\text{T}} 64$.
- Power: Core power consumption in Watts, a fixed counter that we will not use in this work.

4.1.3 Preliminary evaluation

Table 4.1.3 (same as 3.12) contains performance data from all instances being executed alone with scaff on cpu0.

Instance	IPC	package BW	per Core BW	L3->L2 BW	L2->L1 BW	LLC miss BW
L2_1Str_seq	0.322	25.83	0	7.64	13061.92	0
L2_1Str_rdm	0.3352	25.82	0	9.77	13617.21	0
L2_2Str_seq	0.6335	25.82	0	10.24	25705.43	0
L2_2Str_rdm	0.3594	25.81	0	8660.46	14580.69	0
L2_4Str_seq	0.5238	25.76	0	16027.64	33992.71	0
L2_4Str_rdm	0.445	25.74	0	9268.86	28891.07	0
L3s_1Str_seq	0.2624	25.89	0	10650.38	10650.29	0
L3s_1Str_rdm	0.0886	25.99	0	3594.82	3594.96	0
L3s_2Str_seq	0.4112	25.91	0	16639.6	16695.15	0
L3s_2Str_rdm	0.1737	25.86	0	7049.35	7049.78	0
L3s_4Str_seq	0.3873	25.73	0	22946.76	25168.58	0
L3s_6Str_rdm	0.2376	26.09	0	19288.74	19291.63	0
L3m_1Str_seq	0.2603	25.84	0	10588.67	10587.49	0
L3m_1Str_rdm	0.081	26.08	0	3295.24	3544.97	0
L3m_2Str_seq	0.4073	26.31	0	16517.03	16565.68	0
L3m_2Str_rdm	0.1568	25.82	0	6376.51	6865.18	0
L3m_4Str_seq	0.3837	26.21	0	22729.61	24983.86	0
L3m_4Str_rdm	0.1795	25.91	0	11690.68	12575.37	0
L3l_1Str_seq	0.2269	1900.43	1862.31	9168.62	9168.46	385.48
L3l_1Str_rdm	0.0428	359.08	330.38	1760.18	2536.42	330.4
L3l_2Str_seq	0.3628	3096.99	3058.99	14639.19	14697.43	524.51
L3l_2Str_rdm	0.0804	657.78	621.71	3266.29	4753.77	621.71
L3l_4Str_seq	0.3217	3533.06	3499.8	18861.3	20877.91	730.21
L3l_4Str_rdm	0.0764	1247.52	1214.93	7098.76	10352.99	1215.33

MEM_1Str_seq	0.1677	6815.18	6753.77	6748.64	6754.1	1358.62
MEM_1Str_rdm	0.0149	631.25	604.06	943.6	1165.77	604.07
MEM_2Str_seq	0.2802	11340.09	11270.97	11094.01	11272.11	1244.91
MEM_2Str_rdm	0.0293	1213.4	1184.48	1831.94	2283.66	1184.51
MEM_3Str_seq	0.2582	13932.8	13851.04	12401.27	13866.71	1136.57
MEM_8Str_rdm	0.0352	3284.2	3245.13	5020.1	6259.44	3245.23

Table 4.1.3

For the co-scheduling experiment, application pairs were executed in the first pair of cores (cpu 0 and 1) in the package. Each application was co-run with every other, including itself, resulting in a number of 465 execution pairs. Performance data was stored for each application in all co-scheduled pairs. The primary index observed is application slowdown, which is calculated by the ratio of IPC of an application when executed alone and its IPC during co-execution: $\frac{\text{IPC solo}}{\text{IPC in coexec}}$. As a result, slowdown 1 means no slowdown, 2 means double execution time etc.

4.1.4 Results estimation

In the next paragraphs, we will discuss expected behavior of execution pairs, according to the contention prediction scheme presented in Ch. 2, based on application classification [5].

- N – All: As long as processes are running on different cores (which is always the case in this work), most N class applications are not expected to interfere with other programs since they are not sharing resources. However, in some cases programs with very low LLC usage, which are classified as N, may be slowed down if another program is continuously swiping cache data (e.g. from L class). This slowdown case is still expected to be far less than 2.
- C – C: Class C displays the greatest variation in application behavior. As discussed previously, some applications are more vulnerable to be slowed down by interference from other programs than others. Depending on working set size, a pair of C class processes may be using a very small part of LLC, in which case no contention occurs, up to all of the L3 and with need for continuous data replacement. In the latter case, application behavior changes from C to LC and slowdown is significant, as every replacement results in memory fetching penalty. Between those two extreme cases, a lot of intermediate combinations can occur, for which contention and its effects are

generally expected to be low. (Paper: cache organization and replacement policies are expected to handle high activity from different applications on the shared LLC).

- C – LC: In this scenario, there is potential contention on the LLC, which is expected to mostly affect the C member of the pair. This effect is expected to be maximized when a C with a relatively large working set and random pattern –which is memory latency bound, if forced to act like an LC by constantly having its data wiped out of the cache– conflicts with a streaming LC instance that keeps replacing data. Moderate slowdown may occur in this case. In the majority of cases, however, low contention with mostly unnoticeable effects for the LC processes is expected, while C's, especially the vulnerable ones using larger datasets, may be slowed down, but not dramatically.
- C – L: Another class pair in which a wide range of results is expected, as the two classes have intense activity in different levels of the hierarchy. L applications are not expected to be affected. However, an L instance streaming data in the cache at a very high rate can excessively slow down a C application with heavy cache reuse, by destroying its dataset and forcing it to continuously use the memory link as well, like an LC application. Especially if the access pattern is latency bound, as is the case for random benchmarks that cannot take advantage of prefetching, we expect to see the highest slowdown compared to all other tests.
- LC – LC: In this co-execution scenario, medium contention is caused on both levels of the hierarchy –bus and cache– on which these applications display significant activity. Moderate slowdown effects (around 2) are expected in the worst case, since instances of this class do not stress memory link to its limits, thus it can serve the augmented number of requests caused by cache contention, with processes wiping out each other's data. Other instances may not face any interference, if they are not heavily reusing cache.
- LC – L: Memory link contention may occur in this combination, affecting both sides noticeably but not significantly. Although slowdown from memory link increased demand is not dramatic, LC applications with heavy cache reuse and latency-bound pattern suffer from L instances' cache thrashing, along with memory bus competition, accumulatively causing higher slowdown. In cases were the L process does not stress memory bus at such high level, slowdown is expected to be lower.

- L – L: Worst case in this co-scheduling pair of L's competing for the memory link is slowdown about 2, as bandwidth is shared (not necessarily equally) between them. In this architecture it is expected to be even lower, as maximum package bandwidth is measured at 18 GB/s, while a single core can have 14 GB/s. These details will be further discussed along with the results from performance counters. Applications do not share any other memory resource, since they are thrashing the cache even in solo execution and don't have need for data reuse.

4.1.5 Results

Data gathered from all co-execution tests will be presented, along with more detailed analysis for some interesting examples, to evaluate and further discuss the validity of the previously described prediction model. In all tables presented below, each line contains slowdown of the program in the first cell of the row, caused by co-execution with the program on the title of each column, for instance:

	L2_1Str_seq	L2_1Str_rdm	L2_2Str_seq
L2_1Str_seq	1.038	1.000	1.000

In this example table slowdown suffered by L2_1Str_seq is shown, when it was co-scheduled with itself, L2_1Str_rdm and L2_2Str_seq respectively. Additionally, increasingly dark cell background for higher slowdown values facilitates visual detection of high contention scenarios.

For many cases, results may vary for the same experiment repeated more times, since there is dependency on factors that cannot be controlled in user-level, such as data allocation on memory pages by the OS, potentially causing more or fewer data replacements – and varying slowdown as a result – in one experiment that may not occur in a later repetition of the same test. This also explains slowdowns and cache conflicts that occur in cases with small working sets, where theoretically both programs should completely fit in a small part of the cache.

N – N

As expected, in most cases no important slowdown is noticed. According to our initial classification N-N co-execution scenarios are the ones in table 4.1.4.

	L2_1Str_seq	L2_1Str_rdm	L2_2Str_seq
L2_1Str_seq	1.038	1.000	1.000
L2_1Str_rdm	1.493	1.398	1.001
L2_2Str_seq	1.001	1.180	1.105

Table 4.1.4

If dataset sizes used were smaller (less than 128K), no slowdown at all would occur. L2_1Str_rdm appears to be more vulnerable to slowdown. Further examination of performance counters reveals that almost all co-execution scenarios force this instance to higher LLC utilization; running alone it uses much less than 1GB/s L3-to-L2 bandwidth (in the order of tens of MB/s). In almost all co-scheduling tests this number increases to several GB/s, explaining the constant –up to 1.5– slowdown displayed, as it acts more like a C class application (even with some MB/s main bus utilization in extreme cases, caused by LLC misses).

However, runtime behavior alterations in co-scheduling context also occur for other instances: L2_2Str_rdm, L2_2Str_rdm, L2_2Str_rdm benchmarks' performance counters when executed alone, indicate relatively high LLC reuse and suggest C classification. Despite that fact, in most co-execution tests –and for all repetitions of the respective experiments–, performance counters have shown that LLC-to-L2 bandwidth used has been reduced, while L2 utilization becomes more intense, as in typical N class behavior, and thus IPC increases, since L2 is much faster. This phenomenon results in application speedup as IPC slowdown ratio is lower than 1 and, consequently, execution time is lower. All these speedup cases are marked in yellow in all following result tables. It is safe to assume that for most co-execution scenarios, these applications can also be classified as N, a fact confirmed by actual performance data. Their working set sizes (2x101KB or 4x51KB) suggest that it is unlikely for cached data to suffer continuous contamination by a larger streaming application. The new N – N co-execution results are shown in table 4.1.5.

	L2_1Str_seq	L2_1Str_rdm	L2_2Str_seq	L2_2Str_rdm	L2_4Str_seq	L2_4Str_rdm
L2_1Str_seq	1.038	1.000	1.000	1.000	1.000	1.001
L2_1Str_rdm	1.493	1.398	1.001	1.001	1.001	1.001
L2_2Str_seq	1.001	1.180	1.105	1.004	1.002	1.001
L2_2Str_rdm	0.884	1.036	0.885	0.809	0.740	0.757
L2_4Str_seq	0.917	0.938	0.922	0.887	0.847	0.756
L2_4Str_rdm	0.643	0.641	1.010	0.647	0.947	0.978

Table 4.1.5 : N-N slowdown

It is apparent that, with the exception of the L2_1Str_rdm instance, no slowdown occurs in any case, with speedup effect discussed in the previous paragraph.

N – All

All L2 size programs cause no slowdown to any other program of any class, confirming preliminary estimations. Additionally they are not affected by other applications in almost all cases. L2_1Str_rdm benchmark's behavior is different, and it is classified as a C class program when co-run, hence it will not be included with N's. Speed-up phenomena for the three last applications are present in almost all experiments. In only a few cases with L3 or memory link intense streaming applications IPC ratio was slightly over 1. Slowdown effect averages for N co-execution with other classes are shown below:

Class	Co-runner class			
	N	C	LC	L
N	1.00	1.02	1.10	1.07

C – C

C class contains more applications than any other, yet in all experiments no major interference was noted between C processes. Results for the vast majority of experiments were very near or equal to 1. Only the L3l random instance, having the largest working set and which can be marginally classified also as LC, suffered minor slowdown due to competition in LLC utilization when executed with instances with more streams, accessing data more intensely. Figure 4.1.1 shows average slowdown for all other applications in the class and this instance alone. Average slowdown effect noticed was found 1.06.

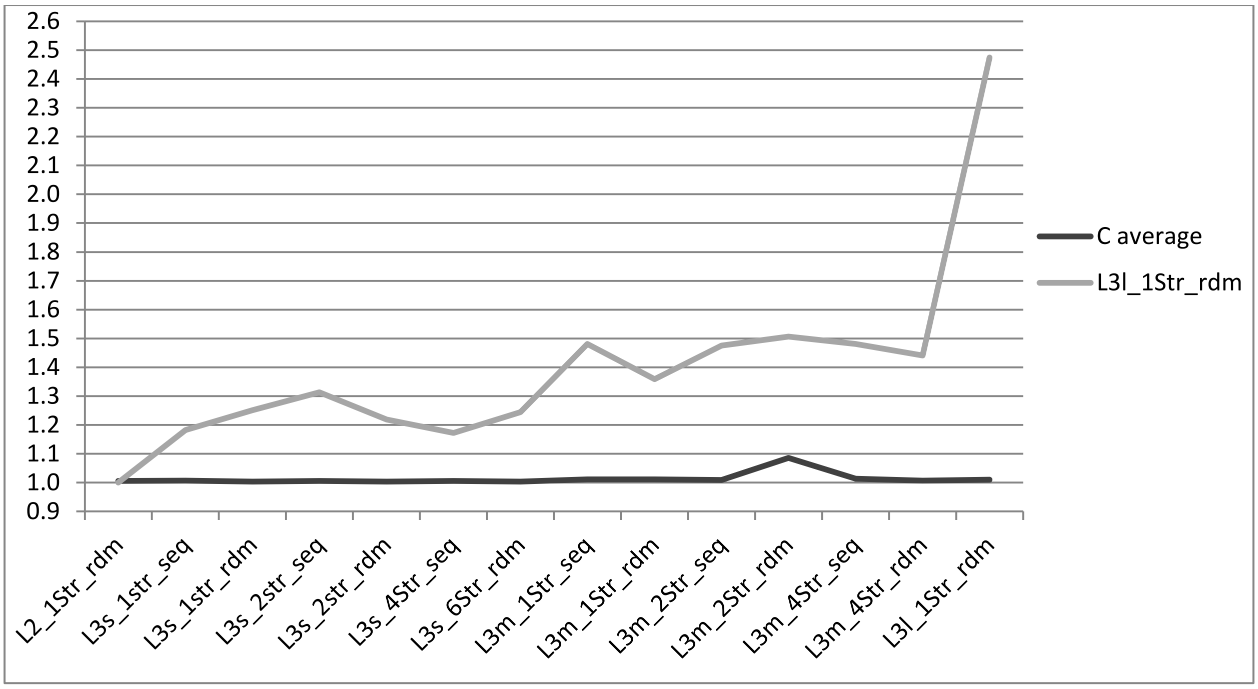


Figure 4.1.1

C – LC

Low to unnoticeable delay was observed for most tests of this class co-execution. Instances affected were random pattern benchmarks of C class, slowed down by streaming LC instances.

	L3l_1Str_seq	L3l_2Str_seq	L3l_4Str_seq
L3s_1str_rdm	1.018	1.034	2.718
L3m_1Str_rdm	2.903	2.765	2.842
L3m_2Str_rdm	1.045	2.830	2.936

Table 4.1.6: C instances slowdown by LC

As it can be seen, increasing the number of accessing streams results in more intense cache data replacements. The L3s instance has a working set small enough to avoid being wiped from LLC, when being run concurrently with the –not so intense– single and 2-stream L3l programs. Counters show ~5000 LLC misses/sec causing a very low ~300KB/s bandwidth, while LLC-to-L2 bandwidth is 3.5 GB/s, very close to its solo performance. However, the 4 stream L3l program, making more access requests, is much more destructive for cached data and forces the previously unaffected L3s application in ~8.4 million misses/sec, resulting in an additional 500MB/s main memory bandwidth, while L3 reuse bandwidth has reduced to

1.3 GB/s. As expected, since the program only does memory accesses and solely depends on them, the ratio of L3-to-L2 bandwidth in the two cases is equal to slowdown ($\frac{3.5}{1.3} = 2.7$),. This example was explained in more detail in order to better demonstrate how contention practically affects a program's execution.

Average slowdown suffered by C applications was 1.27. LC classes were mostly unaffected as well, suffering unnoticeable slowdown for the vast majority of experiments, averaging 1.06. Again, pre-experiment estimations are confirmed.

C – L

As predicted, results in this section ranged from unnoticeable effects to excessive slowdown for C applications. The 3-stream sequential MEM benchmark uses almost the maximum available bandwidth; as a result it can wipe cached data at nearly the highest possible rate. For some C programs this can be disastrous, especially for random pattern instances that can't use the prefetchers.

	MEM_1Str_seq	MEM_2Str_seq	MEM_3Str_seq
L3s_1str_rdm	1.023	1.029	8.204
L3m_1Str_rdm	5.063	5.956	7.642
L3m_2Str_rdm	1.029	5.723	6.908
L3m_4Str_rdm	1.022	1.026	6.575
L3l_1Str_rdm	2.779	3.101	3.891

Table 4.1.7: Random pattern C instances suffering excessive slowdown by L

Sequential and more intense (with more streams) C processes suffered very limited slowdown effects, taking advantage of prefetchers and the memory link's 4-channel parallelism. C class average slowdown was 2.15.

L programs, as expected, were not affected at all.

LC – LC

Medium contention occurred in this scenario resulting in varying results, from unnoticeable up to moderate slowdown. Figure 4.1.2 shows all results in this section. Average slowdown was 1.34

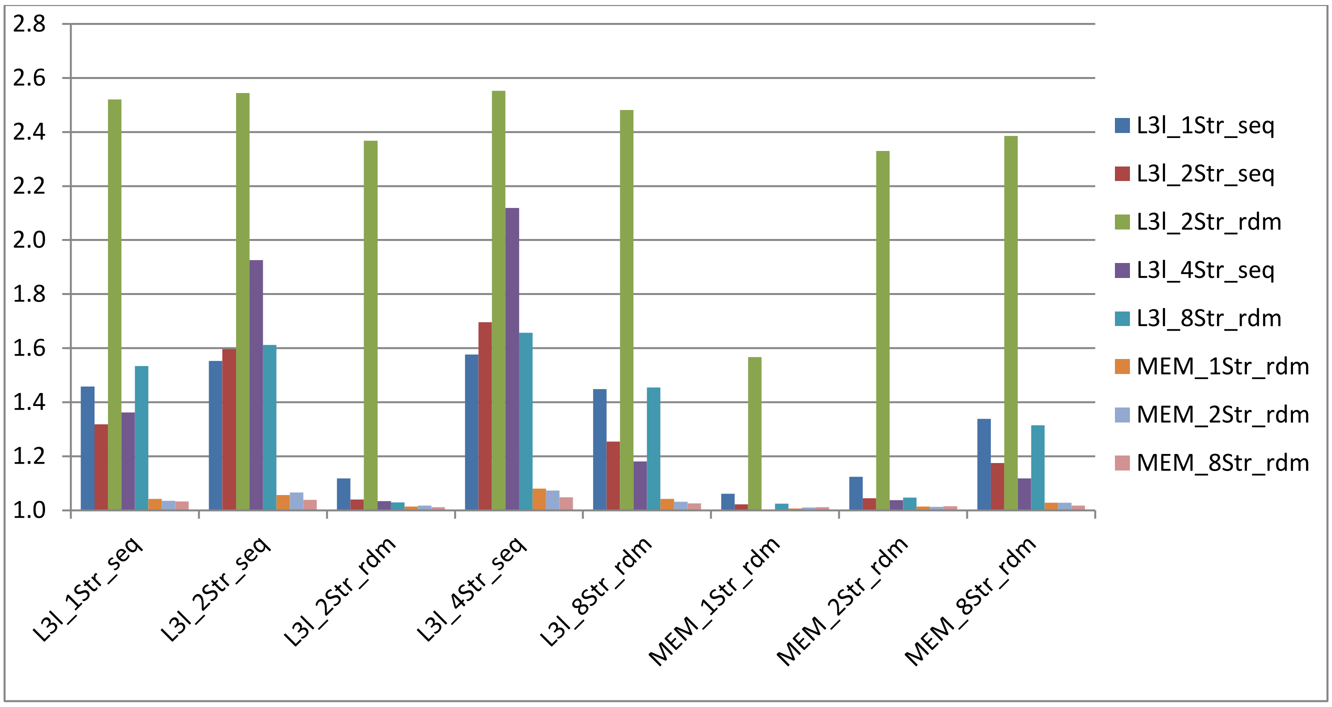


Figure 4.1.2: LC instances slowdown

<u>LC – L</u>

Predicted behavior is again confirmed here. LC programs are affected and delayed increasingly, as streams of L applications increase, and consequently the rate of cache data replacement. Figure 4.1.3 shows this effect and how slowdown increases uniformly for all instances. Average LC slowdown was 1.76.

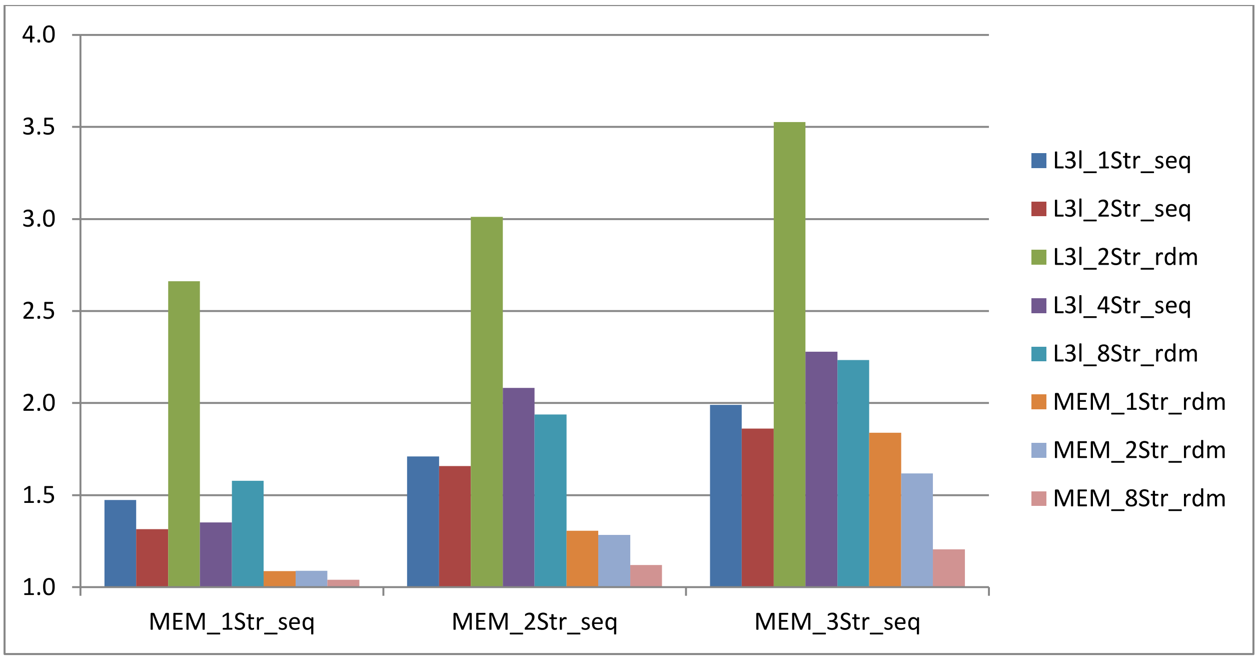


Figure 4.1.3: LC instances slowdown by L streaming programs

L applications were mostly delayed by streaming L3l instances, which were forced to utilize much more main memory bandwidth due to the same L thrashing the cache, resulting in medium memory link contention, with maximum L slowdown near 1.5 and average 1.12 Behavior of these LC instances becomes similar to that of L class. For example if we inspect performance counters data for the L3l_4Str_seq co-execution with MEM_3Str_seq, it can be seen that memory link bandwidth utilization has increased more than 2x, from 3.5GB/s to 8.5GB/s, changing the program's behavior to class L. Such class behavior switches occur in many cases, a result of contention.

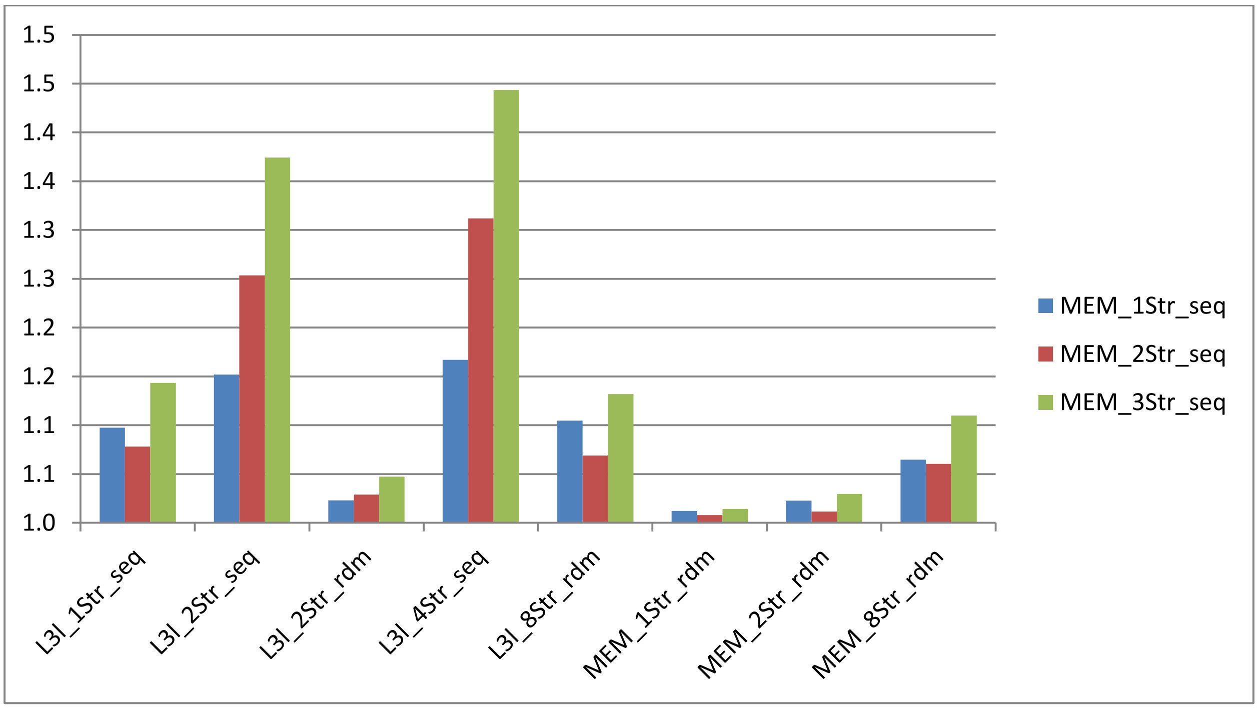


Figure 4.1.4: L instances slowdown, co-run with LC

L – L

In this scenario, memory link-only contention occurs, that results in moderate slowdown:

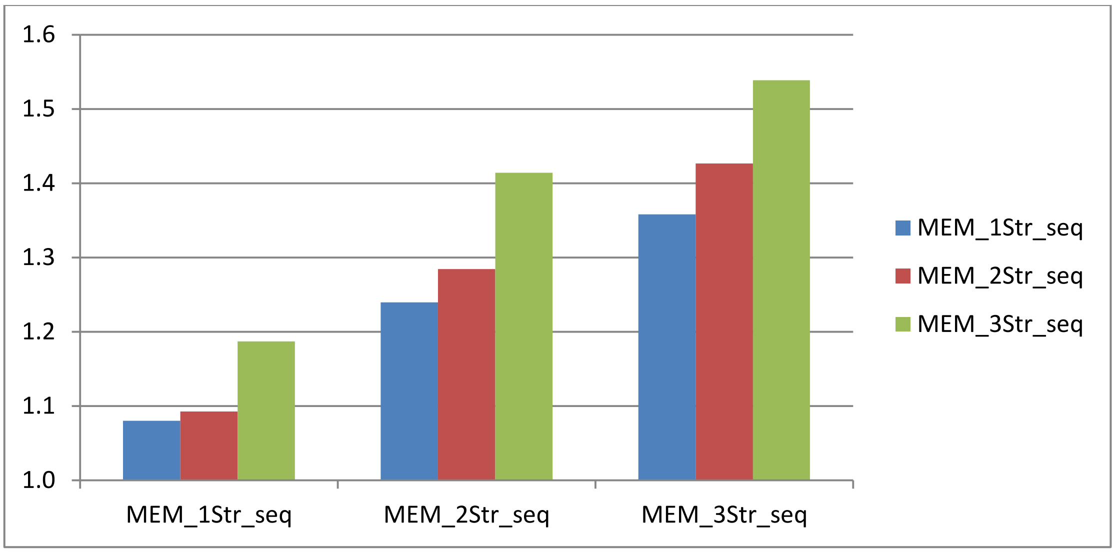


Figure 4.1.5: L-L class slowdown

Maximum slowdown is slightly over 1.5 for a pair of intensely streaming benchmarks. Data from performance counters can be used to explain this. On preliminary execution of the program (MEM_3Str_seq) maximum bandwidth noted was ~13.9 GB/s, very close to the system's per-core maximum. Performance data from co-execution indicate that one instance was using 8.4 GB/s, while the other was using 9.6 GB/s. This means maximum per-socket bandwidth can reach 18 GB/s and average bandwidth for a pair of applications competing for the memory bus is 9 GB/s. As our benchmark is exclusively memory bound, the ratio of bandwidth in solo execution and when sharing the memory link equals slowdown, in this case $\frac{13.9}{9} = 1.54$, which is exactly the result of IPC ratio as well. On a system with equal per-core and per socket maximum bandwidth, average for this situation would be 2, with worst case being slightly over 2, as applications do not share the bus equally and one may use less than half bandwidth.

<u>Overview</u>

Assumptions made in the co-scheduling estimation model described were confirmed. Application behavior was generally as predicted. It was confirmed that shared resources contention may cause change in application behavior, moving it to another class. The phenomenon, in which applications with small enough datasets to fit in private cache often performed as C class processes when executed solo, but co-scheduled with another application they shifted into N class, resulting in speedup effect, was also noticed. Apart from this exception, it was observed that contention situation in shared resources can potentially cause significant change in an application's behavior in co-execution context, moving it to a more memory demanding class, like, for example, from LC to L or from C to LC.

Overall co-scheduling scenarios interference is shown in table 4.1.8, which contains all class co-execution slowdown averages:

Class	Co-runner class			
	N	C	LC	L
N	1.00	1.02	1.10	1.07
C	1.00	1.06	1.27	2.15
LC	1.00	1.06	1.34	1.76
L	1.00	1.00	1.12	1.30

Table 4.1.8: Average class slowdown

4.2 Co-scheduling on Intel Dunnington

4.2.1 Workload profile

For experiments on this architecture we needed an altered set of applications, to expose the effects of co-execution caused shared by memory contention on this different memory hierarchy scheme. Except for the shared 16 MB LLC, pairs of cores also share 3 MB of L2. This makes the choice of cores executing the task inside the package much more significant. Again, we aimed for about 1 minute solo execution time for all instances. Dataset sizes were selected to cover the following range:

- 1 MB: Can fit in the 3MB L2, but if contention occurs caused by another application on the adjacent core, sharing the L2, utilization of LLC is also expected. The difference between co-execution scenarios, utilizing shared L2 or shared LLC only, is expected to be noticeable. This size class was named L2s (L2 *small*)
- 2 MB: Datasets cannot completely fit in L2 (because of associativity) and benchmarks of this size are expected to intensively utilize LLC in co-execution, especially when run on a pair of cores with shared L2. This size class was named L2m (L2 *medium*)
- 3 MB: This dataset class is expected to use both L2 and LLC intensively. Thus it is very likely for these benchmark instances to be largely affected by applications making extended use or thrashing the caches. We named this class L2l (L2 *large*).

- 6 MB: Benchmarks using this dataset size can rely on LLC when executed alone, but when another program also needs to use a significant part of the cache, contention is very likely to cause slowdown and the need to use the memory link as well. These instances also thrash L2 caches for other applications when executed on adjacent cores. Class name is L3s.
- 15 MB: Cannot fit in cache but makes very intense re-use, making it vulnerable to excessive slowdown, when co-scheduled with a program that also makes use of the LLC or continuously destroys its data. We named this category L3l.
- 128 MB: Similar to the previous architecture, datasets much larger than LLC cause the need for continuous memory bus use and cached data replacements, affecting all other programs being executed. Class was named MEM.

As with Sandy Bridge architecture procedure, to create varied memory requests and bandwidth utilization needs, three instances were used for each size class and pattern combination, 1- 2- and max-streams. For max-stream variations, five streams were selected for all class sizes, sequential pattern instances since results indicate that memory bandwidth utilization is maximized using this configuration. Using more datasets, previously described self-contention starts to occur. For random instances 8 streams were used, since the aforementioned effect does not occur in this access pattern.

Table 4.2.1 contains all instances used for co-scheduling experiments, a total number of 36. The same standard name formatting was followed.

Sequential pattern			
Category size	Category name		Variations
1 MB	L2s		1, 2, 5 Streams
2 MB	L2m		1, 2, 5 Streams
3 MB	L2l		1, 2, 5 Streams
6 MB	L3s		1, 2, 5 Streams
15 MB	L3l		1, 2, 5 Streams
128 MB	MEM		1, 2, 5 Streams

Random pattern			
Category size	Category name		Variations
1 MB	L2s		1, 2, 8 Streams
2 MB	L2m		1, 2, 8 Streams
3 MB	L2l		1, 2, 8 Streams
6 MB	L3s		1, 2, 8 Streams
15 MB	L3l		1, 2, 8 Streams
128 MB	MEM		1, 2, 8 Streams

Table 4.2.1

For this suite of applications on Dunnington architecture, there are no N class members, since the only private cache is L1. Classification is as follows:

- Class L: all three MEM size, sequential access instances and MEM_maxstr_rdm
- Class LC: The two remaining MEM size, random access instances, all -five, with the exception- L3l, both pattern instances, except the single stream random benchmark (L3l_1Str_rdm), which is on the margin between C and LC classes, a total of 7 -or 8 with L3l_1Str_rdm- applications.
- Class C: all other programs.

However, if co-executed on cores not sharing L2 but only LLC, some benchmark instances, mentioned in the following paragraphs, are expected to act as N and will be noted as C (N).

4.2.2 Preliminary evaluation and experimental procedure

Co-scheduling tests on this architecture aim, again, to observe slowdown effects caused by contention on the memory hierarchy. This system's characteristics may affect the general model described in 4.1.4, particularly due to low memory bus performance, that results in much lower cache wiping pace compared to the previously tested Sandy Bridge system and other, more recent implementations.

Benchmark instances were co-executed in all possible pairs. To better understand and demonstrate slowdown caused by contention on shared resources, two identical series of co-scheduling tests were conducted, the only difference being the choice of hardware cores. To determine the slowdown effect for these series of experiments, the execution time was used. All programs were timed in single execution with scaff, and time measured for each was used to divide its co-execution time in an experiment to calculate the corresponding slowdown ratio: slowdown = $\frac{\text{t\_coexec}}{\text{t\_solo}}$.

The first series of co-execution experiments were run on core 0 and core 2 of the first package (package0, cores cpu0 and cpu1, as named by the Linux kernel). This configuration enables us to observe performance degradation caused by contention on LLC and the main memory bus, as the cores selected do not share L2 and so, in the absence of additional workload, can use the 3 MB L2 without other programs replacing cached data. Due to this fact, L2s and some L2m instances are expected to act like N class processes, with minimum or zero interference.

On the next round of tests, cores 0 and 1 were chosen (cpu0 and cpu12 Linux kernel devices), which share L2. Thus, on this configuration, contention is likely to occur on 3 levels of the hierarchy, L2, LLC and main memory link, potentially causing excessive slowdown. Interference in cache utilization between programs is expected to occur even with small L2 instances and as working set sizes increase, congestion in all 3 levels of shared memory resources are expected to cumulatively cause delays in program execution.

4.2.3 Results estimation

Application class interaction in co-scheduling scenarios is generally expected as described in 4.1.3, but with some additional remarks:

- N – *: There are no strictly N classified applications in our testing suite. However, in a non-shared L2 scenario some may behave similarly with N and will be discussed further.
- C – C: As explained in 4.1.4, this scenario is probably the most complex to estimate, even when only one level of cache is shared. Moderate slowdown (generally below 2) is expected in this case. If L2 is also shared, effects are expected to be higher, even when both applications have small working sets, causing L2 conflicts and forcing each other to higher LLC utilization. With larger working sets, dual competition is likely in L2 and L3, potentially resulting in high slowdown.
- C – LC: This scenario is expected to mostly affect C instances. However because of the specific system’s single, low performance memory bus, additional contention phenomena may occur as behavior of C instances with larger working sets is forced to

become more LC-like. When sharing L2, slowdown could potentially be multiple, as an LC application can continuously replace data in both L2 and L3. Bandwidth between L2 and LLC is much higher than main memory bandwidth and also prefetching mechanisms exist in this level, thus a relatively large benchmark with streaming pattern may thrash cache data, forcing the co-executed program to constantly request data from memory.

- C – L: Streaming L instances are very likely to cause high slowdown to C applications by constantly wiping cached data and forcing them to pay highly increased number of miss penalties. Again, test results will show how low memory link performance may affect this estimation. In shared L2 execution slowdown is estimated to be much higher, as programs have their cached data continuously destroyed in 2 levels.
- LC – LC: Applications demand use of both memory link and cache, resulting in moderate contention on both levels, with significant slowdown for some scenarios. Memory bus performance could be a considerable factor. For the reasons described previously, -on C-L and C-LC cases- slowdown is expected to greatly increase when L2 becomes shared.
- LC – L: Both classes will be affected, as explained previously, since they need to share the memory link. Additionally, L applications may also replace cache data, causing more slowdown to LC instances.
- L – L: This case remains simple, even with L2 sharing, contention will only occur for memory link and programs do not rely on cache reuse.

4.2.4 Results

4.2.4.1 Co-scheduling with no L2 sharing

C (N) applications

A small subset of programs, with relatively small working sets, can be executed relying almost only on level 2 cache. These applications, judging from performance and size, are all L2s instances, while sequential L2m instances are on the margins of classification criteria. Testing behavior confirmed that these instances do not interfere with each other when executed on different cores, not sharing L2. Average slowdown was found 1.03, with the vast majority of cases found at 1.00 and maximum slowdown slightly above 1.1. Additionally, slowdown caused by this subset to all other programs was very low, as will be further explained.

A part of these tests section results is presented below:

	L2s_1Str_seq	L2s_1Str_rdm	L2s_2Str_seq	L2s_2Str_rdm
L2s_1Str_seq	1.000	1.051	1.087	1.103
L2s_1Str_rdm	1.001	1.001	1.035	1.051
L2s_2Str_seq	1.000	1.000	1.001	1.015
L2s_2Str_rdm	1.001	1.001	1.001	1.001

Table 4.2.2

C – C

Low to moderately high slowdown is noticed on this co-scheduling scenario, due to the variation of programs' behavior. C class includes benchmarks with working sets ranging from small enough to fit in cache with minor conflicts, to sizes that can marginally fit and therefore very likely to be delayed, if other programs also compete for intense LLC use, forcing each other to suffer from increased miss penalties. LLC contention -and its resulting slowdown- becomes noticeable when sizes of the co-executed instances occupy a fairly large part of the cache, belonging in the L3s (or larger) size category. Before this point slowdown is generally unnoticeable, with values very near 1, with a small number of exceptions, which do not exceed 1.3. For larger size classes, L3 contention causes gradually increasing slowdown, with a maximum value approx. 2.5. The results for these instances can be seen in table 4.2.3:

	L3s_1Str_ seq	L3s_1Str_ rdm	L3s_2Str_ seq	L3s_2Str_ rdm	L3s_max_ seq	L3s_max_ rdm	L3l_1Str_ rdm
L3s_1Str_ seq	1.590	1.344	1.341	1.463	1.364	1.687	1.606
L3s_1Str_ rdm	1.303	1.171	1.036	1.039	1.012	1.104	1.571
L3s_2Str_ seq	1.360	1.084	1.892	1.507	1.674	2.025	1.036
L3s_2Str_ rdm	1.480	1.084	1.502	1.256	1.141	1.260	1.058
L3s_max_ seq	1.384	1.059	1.675	1.145	1.770	2.153	1.034
L3s_max_ rdm	1.622	1.095	1.920	1.199	2.040	1.779	1.059
L3l_1Str_ rdm	1.809	1.824	1.151	1.178	1.148	1.240	2.378

Table 4.2.3

It can be observed that random pattern benchmarks generally interfere with other random instances, while sequential ones cause slowdown to both patterns. The way each pattern affects applications, as memory requests intensity increases can be observed in figures below:

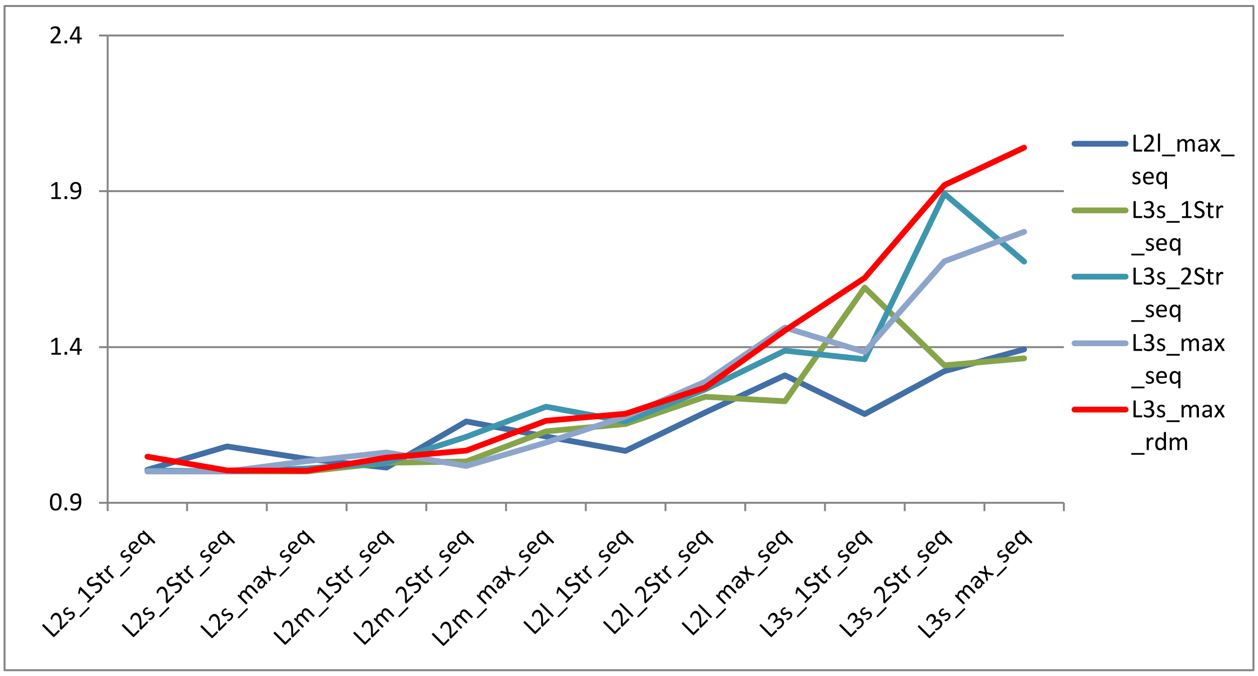


Figure 4.2.1: Slowdown caused by sequential instances

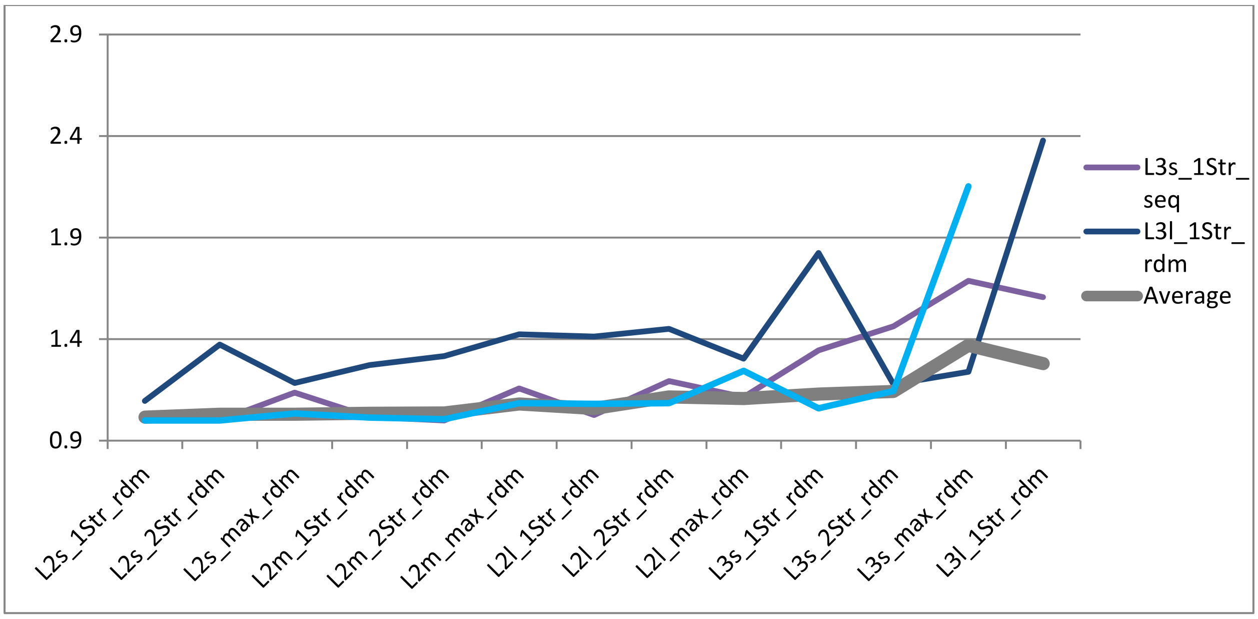


Figure 4.2.2: Slowdown caused by random instances, L3l_1Str_rdm instance is affected most

In the graphs above, moving towards the right side of the plot, it can be observed that applications racing for the cache, as competing memory requests increase with more streams, alter their behavior close to that of LC class.

Average slowdown suffered in this class' combinations was 1.12, however this includes all applications with N class behavior as well. If limited to instances that always need to use L3, average is 1.3, with worst case scenario being over 2. From this series of experiments, it is suggested that maximum slowdown caused solely by LLC contention on this system is about 2.5, and can be generally assumed less than 3.

<u>C – LC</u>

Moderate contention levels were observed in co-scheduling combinations of benchmark instances belonging in these classes. The memory bus of this system is single channel, serializing all requests and is not capable of serving more than 2.5 GB/s; there is no data prefetching either. This results in programs with large enough datasets (half the LLC size and more) to constantly force each other in LLC misses by wiping data, while in the same time competing for main memory accesses that can't be accelerated (if sequential) by prefetching mechanisms.

L3l benchmarks appear to be destroying cache data of other applications at a relatively high pace, affecting all C processes, even L2s-sized with minimum LLC utilization. Partially relying on the memory link by definition, these LC applications also face the effects of bus contention, as C applications also increase demand and their behavior verges with that of their LC competitors. They are also affected by C instances with small working sets but very intensive in cache use (2 or max streams). The 2 MEM size random instances (1 Str. and 2 Str.), however, seem to remain unaffected, as they don’t suffer from noticeable slowdown, nor cause delays in any other application. This can be explained as their random pattern doesn’t rely on cache reuse and they do not stress the memory link to a high level either, thus forcing their pair in less misses, consequently in less bus use demand and overall very low combined contention.

Table 4.2.4 contains all C instances suffered slowdown by the L3l size class, MEM instances are omitted as most results were near 1. Slowdown numbers are presented to show the variety of results noted.

	L3l_1Str_seq	L3l_2Str_seq	L3l_2Str_rdm	L3l_max_seq	L3l_max_rdm
L2s_1Str_seq	1.109	1.344	1.744	1.462	1.578
L2s_1Str_rdm	1.044	1.270	1.633	1.414	1.453
L2s_2Str_seq	1.190	1.255	1.575	1.356	1.421
L2s_2Str_rdm	1.468	1.277	1.552	1.354	1.406
L2s_max_seq	1.268	1.365	1.258	1.379	1.381
L2s_max_rdm	1.482	1.563	1.298	1.404	1.442
L2m_1Str_seq	1.179	1.447	1.919	1.644	1.723
L2m_1Str_rdm	1.037	1.288	1.653	1.398	1.489
L2m_2Str_seq	1.641	1.375	1.780	1.545	1.571
L2m_2Str_rdm	2.132	1.493	1.871	1.664	1.705
L2m_max_seq	1.661	1.843	1.419	1.577	1.554
L2m_max_rdm	1.741	2.201	1.451	1.399	1.533
L2l_1Str_seq	1.125	1.340	1.931	1.471	1.710
L2l_1Str_rdm	1.107	1.390	1.872	1.593	1.672
L2l_2Str_seq	1.497	1.385	1.928	1.521	1.696
L2l_2Str_rdm	1.698	1.263	1.887	1.434	1.617
L2l_max_seq	1.375	1.543	1.278	1.510	1.712
L2l_max_rdm	1.555	1.755	1.287	1.530	1.748
L3s_1Str_seq	1.260	1.577	2.487	1.755	2.095
L3s_1Str_rdm	1.307	1.514	2.477	1.645	1.750
L3s_2Str_seq	1.215	1.584	2.583	1.774	2.117
L3s_2Str_rdm	1.558	1.850	2.556	2.187	2.225
L3s_max_seq	1.254	1.562	1.072	1.797	2.150
L3s_max_rdm	1.331	1.738	1.167	1.686	1.995
L3l_1Str_rdm	3.248	1.722	2.895	1.889	2.023

Table 4.2.4: C slowdown caused by L3l LC instances

As observed above, most results are over 1.5 (table average is 1.6), which is a considerable effect. High slowdown was also noted in some cases (2.5 or more) with a maximum point greater than 3.2. Average slowdown, including MEM instances, was found 1.5.

LC applications also suffered considerable delay in numerous cases, some of which are presented below:

	L2s_2Str_rdm	L2m_2Str_seq	L2m_2Str_rdm	L2m_max_seq	L2m_max_rdm	L2l_2Str_rdm	L2l_max_seq	L3l_1Str_rdm
L3l_1Str_seq	1.729	1.938	2.286	1.959	2.150	2.156	1.685	3.416
L3l_2Str_seq	1.179	1.272	1.254	1.702	2.129	1.256	1.481	1.418
L3l_2Str_rdm	1.185	1.364	1.302	1.085	1.162	1.553	1.016	1.975
L3l_max_seq	1.157	1.324	1.294	1.349	1.252	1.320	1.342	1.441
L3l_max_rdm	1.174	1.316	1.296	1.299	1.342	1.455	1.487	1.508

Table 4.2.5: LC slowdown caused by C instances (selected results)

LC class, although generally less delayed than C by their interaction, also suffered high slowdown (max. 3.4). L3l_1Str_seq seems to be more vulnerable and that is because of the stalling phenomenon of a program with a single access explained in the beginning of the chapter.

Average for the LC suffered slowdown caused by C programs was found 1.4, 1.1 for the optimistic MEM cases and 1.45 for the pairs with moderate to high contention. In the next figures, average slowdown for the 2 classes is presented. It becomes apparent how increasing the competing program’s dataset and streams affects delay caused.

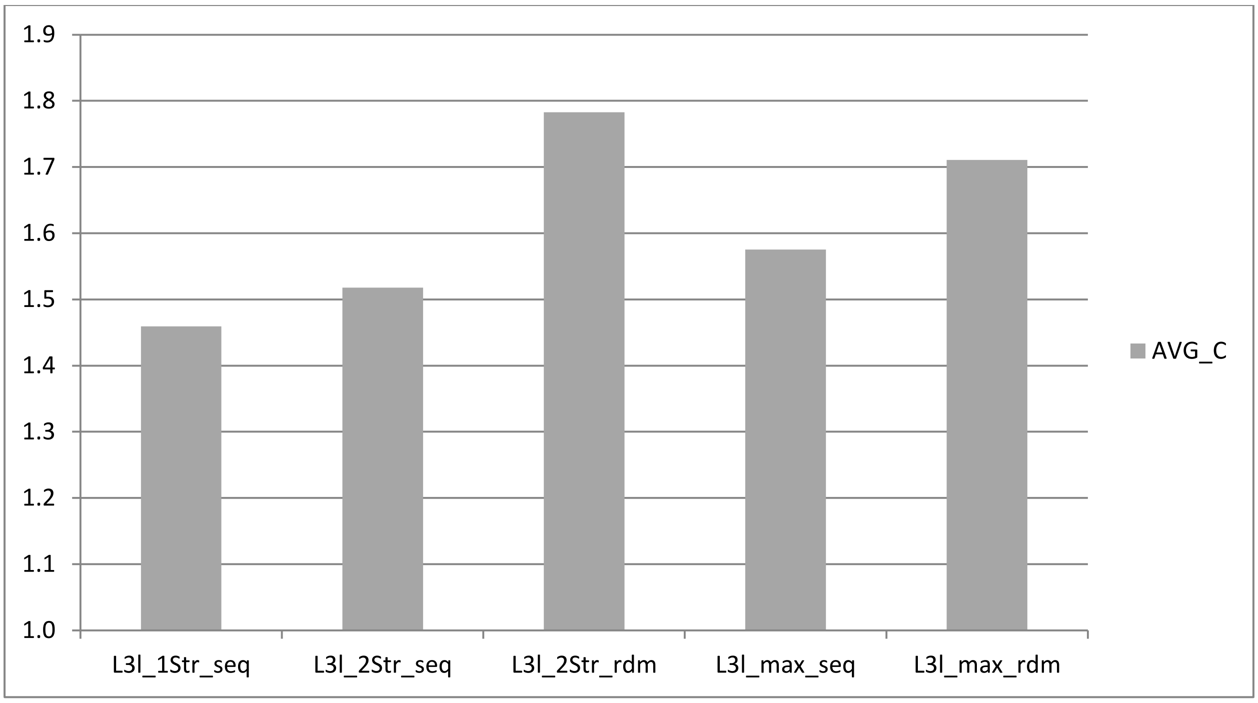


Figure 4.2.3: Average C slowdown caused by L3l LC instances

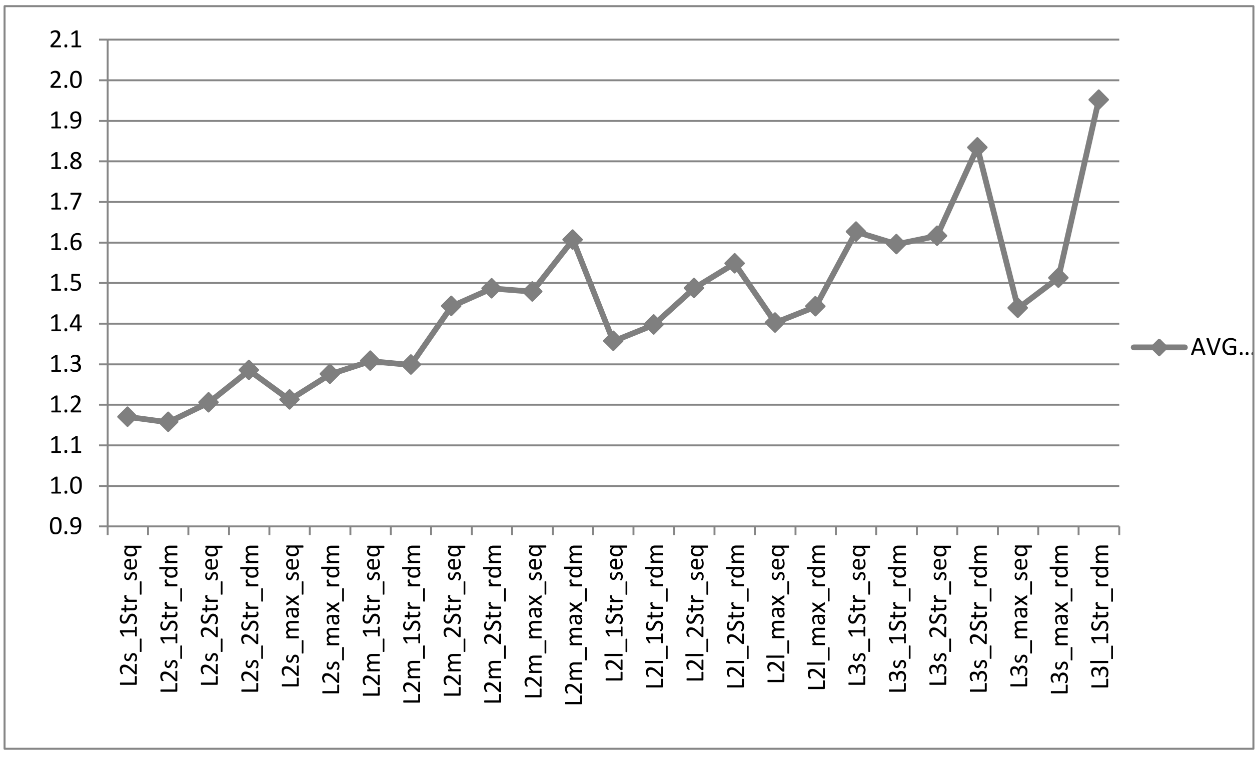


Figure 4.2.4: Average L3l LC slowdown caused by C instances

C – L

Contrary to previous estimations, MEM instances with max streams did not cause nor suffer significant slowdown in this test (around 1.1 and less in most cases). However, 1 and 2 Stream MEM sequential benchmarks caused moderate delay to C applications. Figure 4.2.5 shows slowdown caused by these 2 L instances to all C benchmarks:

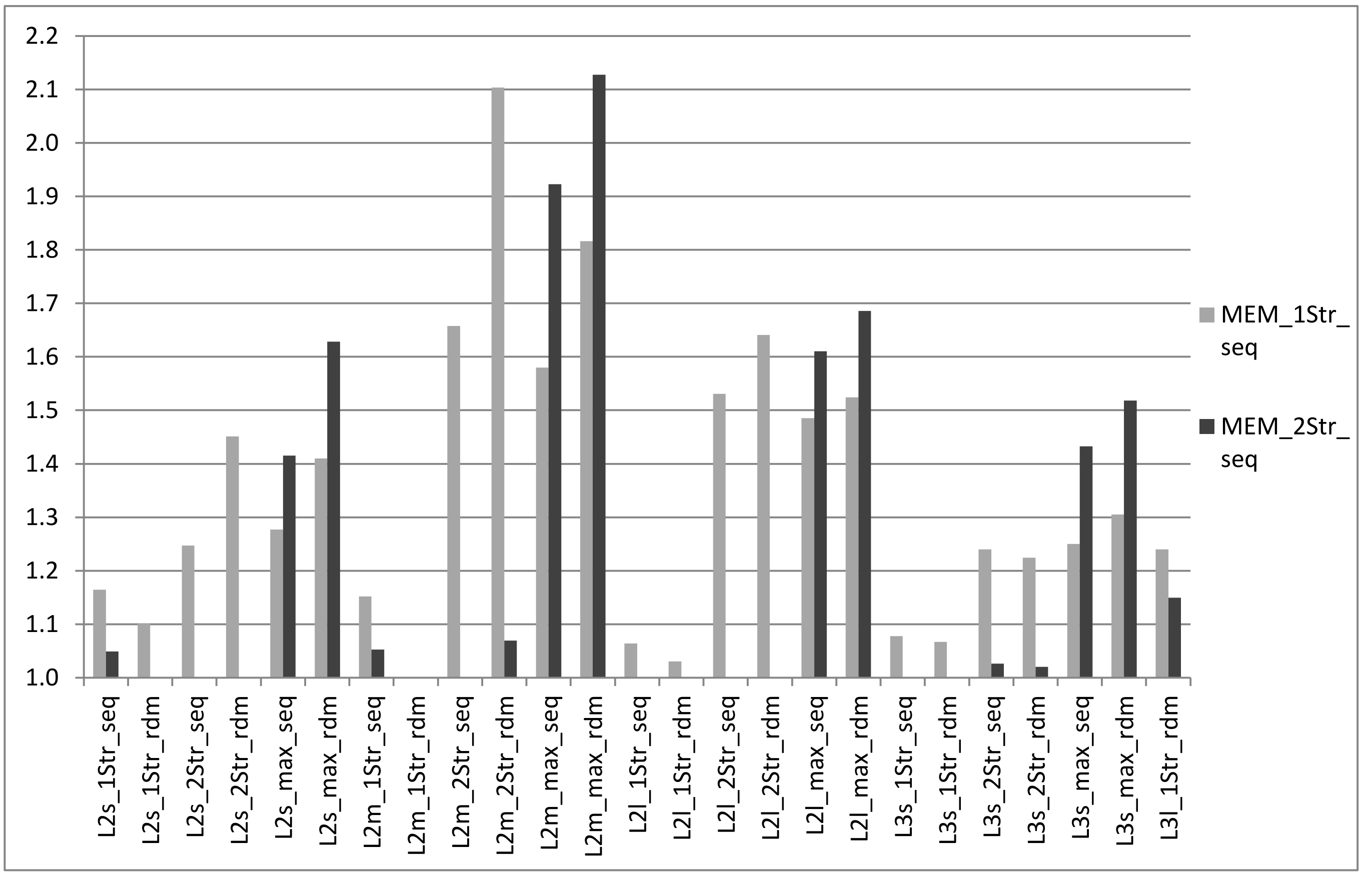


Figure 4.2.5: Average C slowdown caused by L instances
(Note that contrary to all previous graphs, “suffering” class is on horizontal axis, due to high population)

The 2 L applications were also slowed down in numerous cases, as can be seen in the following graph:

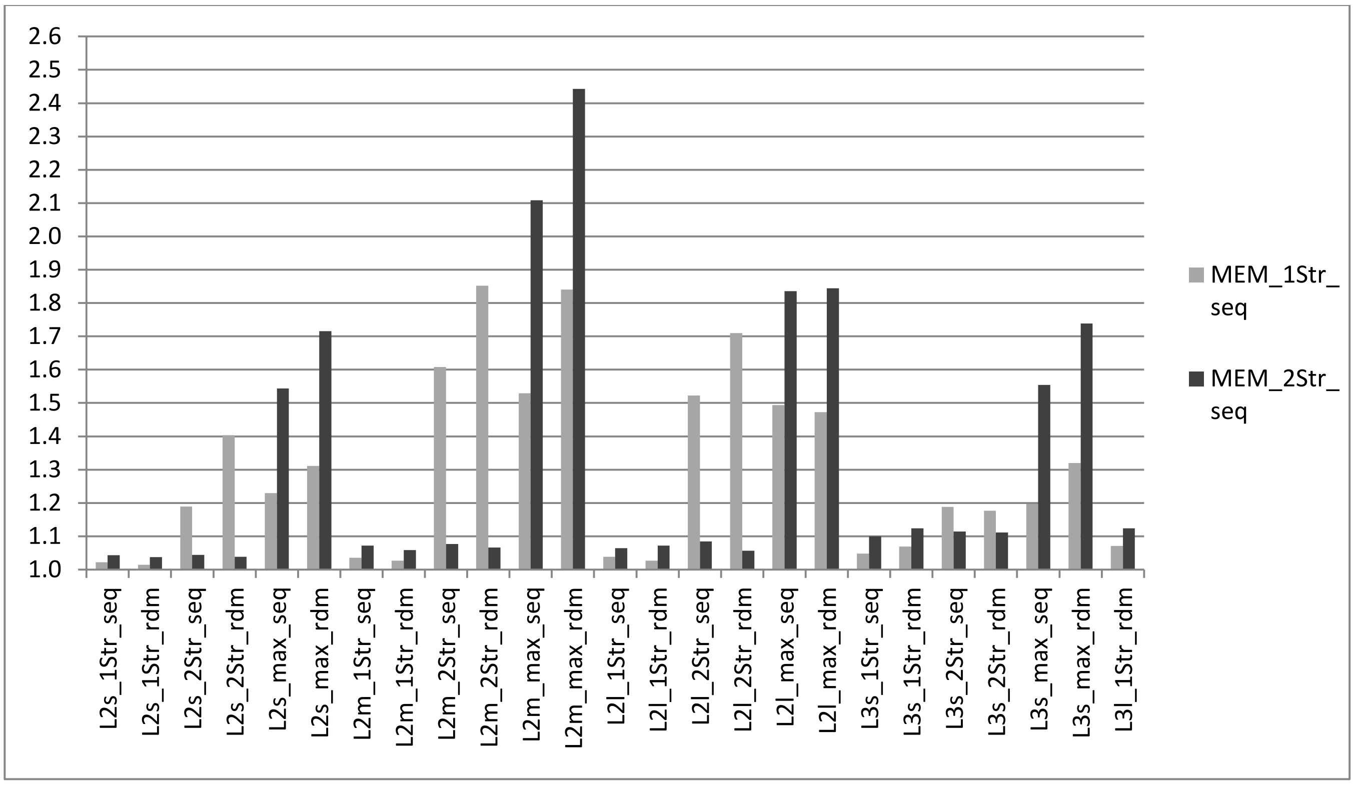


Figure 4.2.6: L suffered slowdown

It can be observed that increasing concurrent requests (by adding data streams) results in augmented memory contention, as all requests are serialized and waiting to be served by the single memory bus. However, it is noted for all cases on display that when instances utilize the same number of streams, the L application is not affected. When a C process makes more requests than the L (C streams > L streams), the L is slowed down. This consistent behavioral pattern explains why the 2 MEM_max instances are not affected at all, as all competing instances have less or equal data streams. The reason this happens is most probably the way this specific architecture handles access requests and allocates bus resources to the processes; requests from separate applications seem to be prioritized as different sets. DRAM controller-level contention and prefetching mechanisms are also likely to contribute in this effect. A relatively similar, yet less obvious pattern can be observed in the way C class processes are affected (figure 4.2.5). A C instance is affected only if it uses more streams than its L competitor (and they are both delayed). With equal streams no significant delay is noted and confirms why MEM_max instances don't affect any of the C programs tested.

LC – LC

High contention was noted on this series of experiments. A maximum of 5.2 was measured (suffered by L3l_1Str_seq co-executed with L3l_2Str_rdm), as well as many results greater than 3. Average was found to be 2.4, higher than all other class combinations.

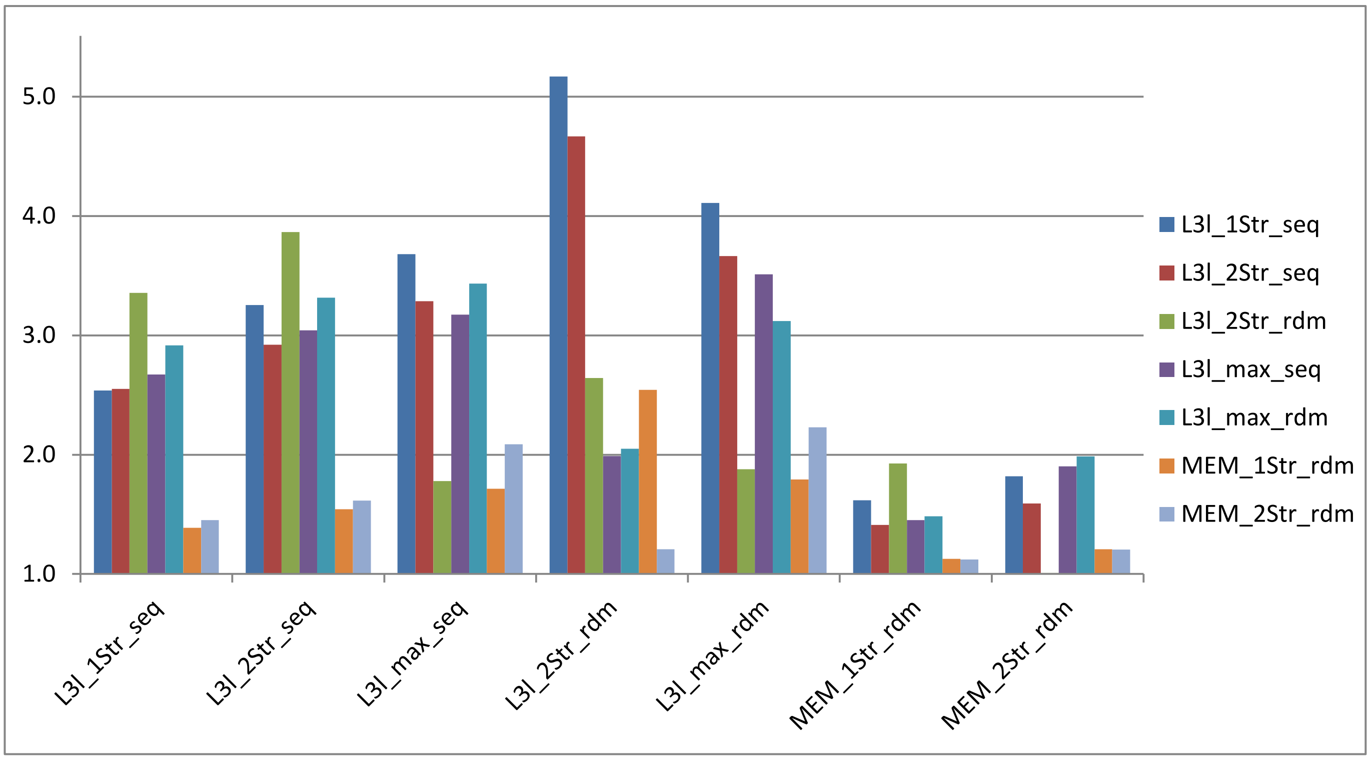


Figure 4.2.7: LC class slowdown (program on horizontal axis causes the delay)

High slowdown in all L3l instances is a result of L3 contention, forcing programs to suffer from increased number of cache miss penalties in comparison to alone execution. Increased contention also occurs on the memory bus, the low bandwidth of which seems to largely affect execution.

LC – L

Moderate to high slowdown appears for programs of both classes, as LC applications seem to both cause and suffer more slowdown than any other class. Delay ratio caused by L instances is presented in table 4.2.6:

	MEM_1Str_seq	MEM_2Str_seq	MEM_max_seq	MEM_max_rdm
L3l_1Str_seq	1.763	1.679	1.782	1.973
L3l_2Str_seq	2.549	1.471	1.569	1.741
L3l_max_seq	2.697	3.002	1.493	1.675
L3l_2Str_rdm	3.335	1.000	1.000	1.000
L3l_max_rdm	2.876	3.264	1.338	1.476
MEM_1Str_rdm	1.407	1.034	1.087	1.148
MEM_2Str_rdm	1.446	1.609	1.148	1.212

Table 4.2.6: LC class slowdown by L

The behavioral pattern noticed in C-L results discussion is present again, especially for L3l random instances: slowdown decreases when competitor's streams (L) are equal or more than LC's (see Fig. 4.2.9).

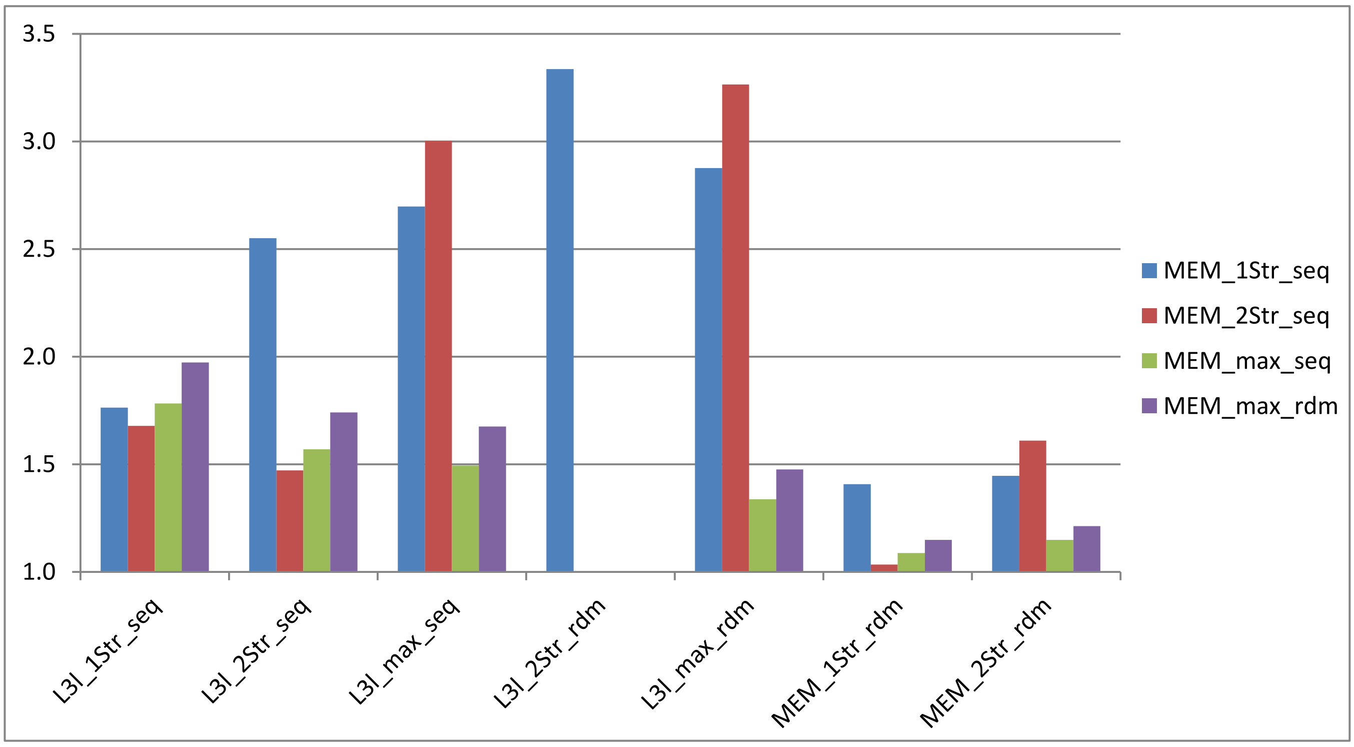


Figure 4.2.8: LC class slowdown caused by L

Average slowdown was found to be 1.75. Higher contention scenarios (2.5 and more) are very likely to occur, as the results reveal. L applications face even greater average slowdown, at 1.95, as it seems the memory bus becomes a bottleneck for both classes. Figure 4.2.9 presents slowdown each LC application caused to all L.

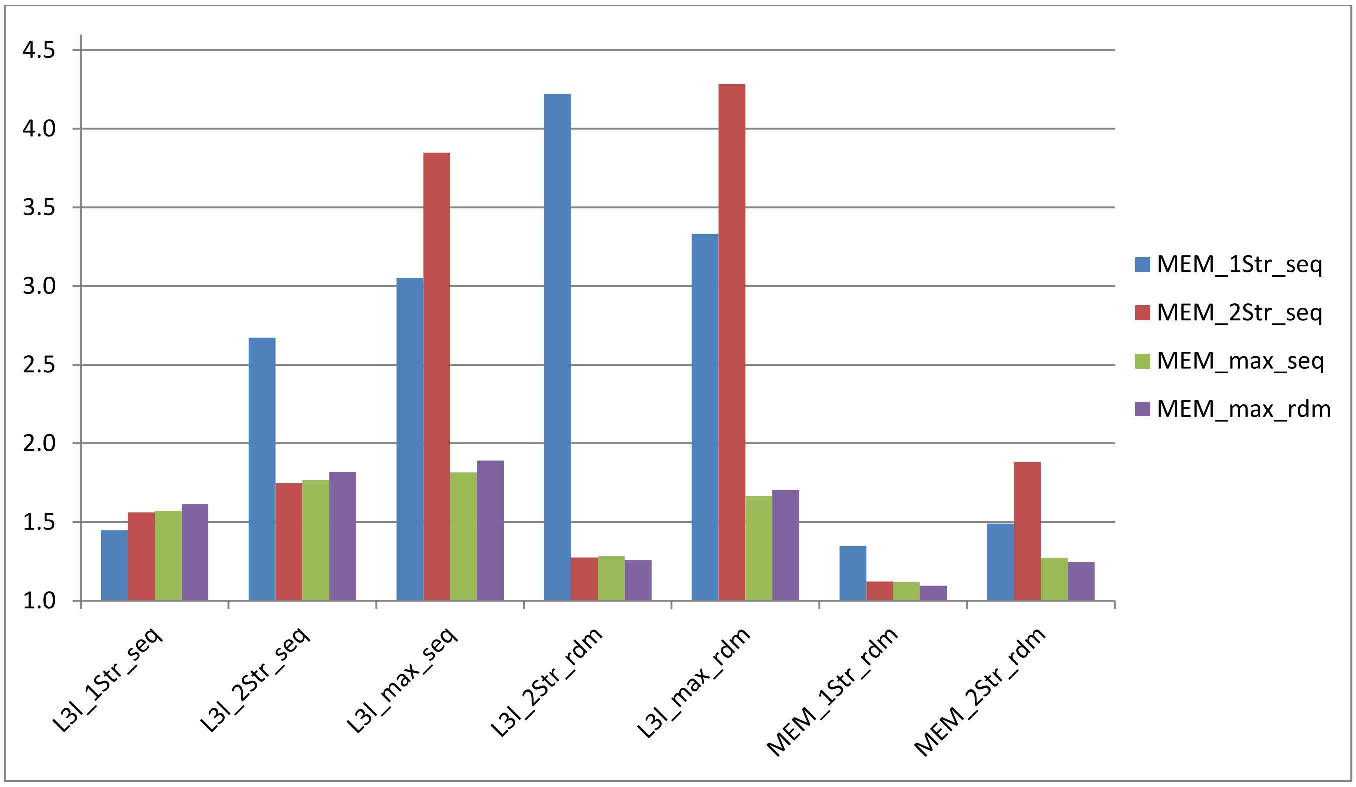


Figure 4.2.9: L class slowdown caused by LC

MEM random streams of the LC class do not cause or face high slowdown, as they only use a relatively small part of the available bandwidth.

<u>L – L</u>

In this, quite simpler, scenario, our estimation model is confirmed. Even when competing programs can potentially stress the memory bus to its limits, bandwidth is divided and a worst case average slowdown will slightly exceed 2. Average for all tests was found to be 1.7

	MEM_1Str_seq	MEM_2Str_seq	MEM_max_seq	MEM_max_rdm
MEM_1Str_seq	1.447	1.378	1.464	1.594
MEM_2Str_seq	1.561	1.749	1.847	2.057
MEM_max_seq	1.573	1.752	1.821	2.006
MEM_max_rdm	1.590	1.811	1.862	1.676

Table 4.2.7: L-L class slowdown

Overview

Having separate L2 cache on the cores utilized for this series of experiments let us observe the effects of memory bus and LLC contention. A combination of system-specific characteristics (low memory link performance in conjunction with the way it serially prioritizes memory access requests) resulted in a slightly altered overall behavior, compared to preliminary estimations.

Class	Co-runner class			
	N	C	LC	L
N	(1.03)	-	-	-
C	-	1.15	1.47	1.2
LC	-	1.34	2.38	1.75
L	-	1.2	1.95	1.7

Table 4.2.8: Slowdown average overview, N-N value is the C(N) applications when co-executed.

Class LC appears to cause most slowdown to other classes, as well as being the most vulnerable to be delayed. Taking advantage of not sharing the large L2, C programs also are less affected.

4.2.4.2 Co-scheduling with L2 sharing

Results of the second round of tests, conducted on a pair of cores with shared L2, are presented. N class instances do not exist in our suite, thus all testing is between C, LC and L classes.

C – C

Moving into co-execution context with shared L2 cache, C class application interaction changes dramatically. The set of applications in the suite belonging to this class is quite large (25), so results will be presented in sections.

Starting with L2s and L2m series, tables 4.2.9a and 4.2.9b contain slowdown caused by sequential and random instances of the same size classes respectively:

	L2s_1Str_seq	L2s_2Str_seq	L2s_max_seq	L2m_1Str_seq	L2m_2Str_seq	L2m_max_seq
L2s_1Str_seq	1.098	1.268	1.219	1.228	1.935	2.234
L2s_1Str_rdm	1.267	1.088	1.325	2.720	3.566	3.543
L2s_2Str_seq	1.154	1.265	1.356	1.199	1.401	2.480
L2s_2Str_rdm	1.181	1.478	1.466	1.667	3.502	3.156
L2s_max_seq	1.283	1.334	1.672	1.217	1.421	2.012
L2s_max_rdm	1.131	1.240	1.401	1.128	1.244	1.470
L2m_1Str_seq	1.367	1.466	1.608	2.172	2.316	2.641
L2m_1Str_rdm	1.830	1.908	1.797	2.970	3.115	3.280
L2m_2Str_seq	1.905	1.636	1.997	2.071	3.551	4.016
L2m_2Str_rdm	1.882	1.837	1.975	3.308	3.664	3.734
L2m_max_seq	1.912	2.318	2.157	1.783	2.710	5.503
L2m_max_rdm	1.499	1.730	1.663	1.673	2.474	3.421

Table 4.2.9a

	L2s_1Str_rdm	L2s_2Str_rdm	L2s_max_rdm	L2m_1Str_rdm	L2m_2Str_rdm	L2m_max_rdm
L2s_1Str_seq	1.063	1.096	1.174	1.054	1.117	1.683
L2s_1Str_rdm	1.261	1.170	1.438	1.463	1.905	2.800
L2s_2Str_seq	1.042	1.135	1.238	1.032	1.104	1.790
L2s_2Str_rdm	1.085	1.218	1.309	1.060	1.350	3.015
L2s_max_seq	1.214	1.324	1.357	1.155	1.285	1.796
L2s_max_rdm	1.109	1.180	1.338	1.061	1.114	1.441
L2m_1Str_seq	1.309	1.359	1.588	1.320	1.495	2.325
L2m_1Str_rdm	1.776	1.750	1.727	2.427	2.753	2.940
L2m_2Str_seq	1.321	1.537	1.952	1.247	1.513	3.718
L2m_2Str_rdm	1.799	1.859	1.791	2.081	2.690	3.434
L2m_max_seq	1.502	1.699	2.300	1.308	1.569	5.718
L2m_max_rdm	1.226	1.420	1.684	1.116	1.350	3.971

Table 4.2.9b

Increasing the working set size and number of requests noticeably increases slowdown. Contrary to the memory bus, in-cache bandwidth is sufficient to intensively wipe data if an application uses a streaming pattern. Sequential instances cause noticeably more slowdown than random ones, despite the lower number of accesses (5 instead of 8), taking advantage of cache-level prefetching. Observing L2s instances' interaction with each other (1+1 MB working sets), confirms that cache contention starts to occur when working sets' size exceeds approximately half the size of this cache. Slowdown starts from 1.05, reaching more than 1.5 as the number of requests increases.

Moving to L2m sized instances, slowdown is constantly at least close to 2, in many cases greatly exceeding 3. Without loss of generality, it can be assumed that co-running L2m working sets (total 4MB) can fit in the LLC with no need for extensive main memory accesses. Given this assumption it can be estimated that L2-only contention can excessively delay a program. From the results above, a slowdown factor of 4 seems realistic for high contention situations and potentially even close to 5 in extreme cases, although in such cases it is very difficult to estimate if it's only L2-caused or in conjunction with possible LLC competition, without detailed performance information from counters.

Increasing the competing instances size, slowdown is even higher (presented in tables 4.2.11 and 4.2.12) caused by sequential and random instances respectively:

	L2l_1Str_seq	L2l_2Str_seq	L2l_max_seq	L3s_1Str_seq	L3s_2Str_seq	L3s_max_seq
L2s_1Str_seq	1.235	1.920	4.419	1.208	1.978	5.586
L2s_1Str_rdm	4.495	5.870	6.843	5.066	6.645	8.346
L2s_2Str_seq	1.220	1.407	4.486	1.226	1.479	4.048
L2s_2Str_rdm	1.259	5.646	7.634	1.249	2.139	9.127
L2s_max_seq	1.269	1.356	1.858	1.278	1.405	1.671
L2s_max_rdm	1.171	1.258	1.461	1.121	1.303	1.464
L2m_1Str_seq	2.238	3.391	4.925	2.174	3.586	5.543
L2m_1Str_rdm	3.734	4.698	5.239	4.427	5.226	5.925
L2m_2Str_seq	2.059	3.727	6.506	2.230	3.784	6.967
L2m_2Str_rdm	4.455	5.039	5.993	4.933	5.126	6.670
L2m_max_seq	1.747	2.917	5.964	1.628	2.755	5.585
L2m_max_rdm	1.623	2.539	3.368	1.586	2.508	3.222

Table 4.2.10a: L2s-L2m slowdown induced by sequential L2l-L3s instances

	L2l_1Str_rdm	L2l_2Str_rdm	L2l_max_rdm	L3s_1Str_rdm	L3s_2Str_rdm	L3s_max_rdm
L2s_1Str_seq	1.031	1.109	2.359	1.033	1.119	2.820
L2s_1Str_rdm	1.090	2.259	5.139	1.098	1.671	7.228
L2s_2Str_seq	1.039	1.074	1.545	1.033	1.086	1.450
L2s_2Str_rdm	1.068	1.143	5.962	1.030	1.132	7.061
L2s_max_seq	1.119	1.239	1.560	1.124	1.183	1.391
L2s_max_rdm	1.055	1.076	1.268	1.050	1.065	1.166
L2m_1Str_seq	1.279	1.520	3.763	1.260	1.508	3.850
L2m_1Str_rdm	2.472	3.050	4.307	2.477	3.108	5.099
L2m_2Str_seq	1.205	1.486	5.202	1.243	1.471	4.962
L2m_2Str_rdm	2.061	2.811	4.959	2.032	2.869	5.601
L2m_max_seq	1.290	1.674	6.848	1.258	1.510	6.147
L2m_max_rdm	1.127	1.393	3.708	1.064	1.316	3.474

Table 4.2.10b: L2s-L2m slowdown induced by random pattern L2l-L3s instances

Results show that slowdown is increasing to very high levels. It can be observed that induced slowdown increases depending –in order of significance- on number of streams, dataset size and access pattern, with sequential instances being more aggressive to smaller competitors' cached data, as a result of their streaming nature. Slowdown noted was often much higher than 5, in some cases even close to 10, as moderate contention starts to take place also on L3, forcing small L2 instances to highly increased LLC and main memory accesses, the access time of which is much higher. This relative difference is reflected in slowdown ratio. Figures 4.2.10 and 4.2.11 show average L2s and L2m slowdown caused by L2l and L3s benchmarks.

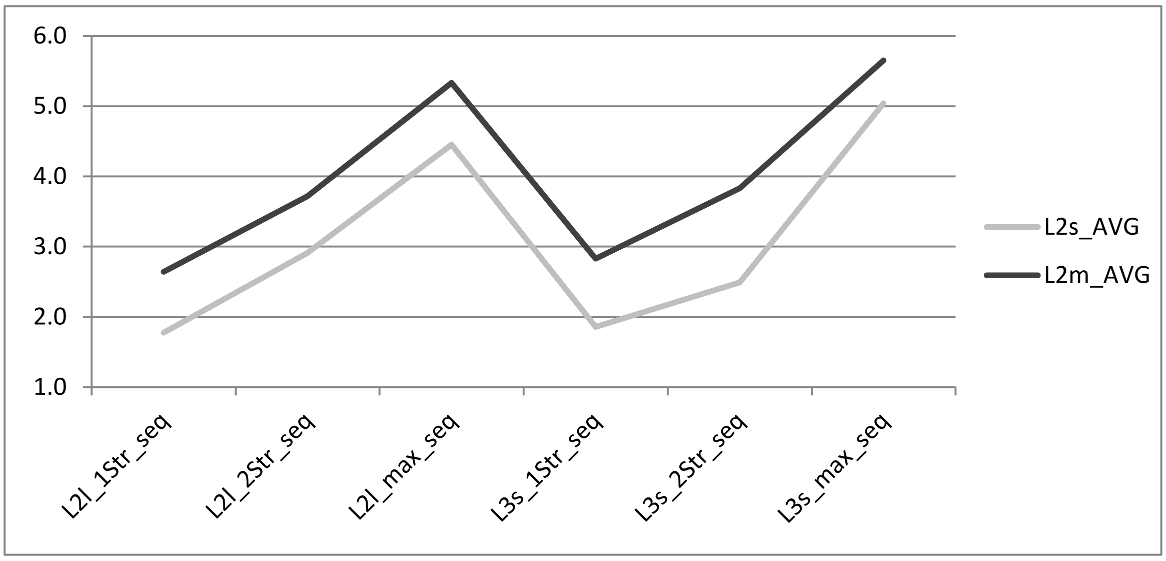


Figure 4.2.10a: L2l and L3s sequential induced slowdown

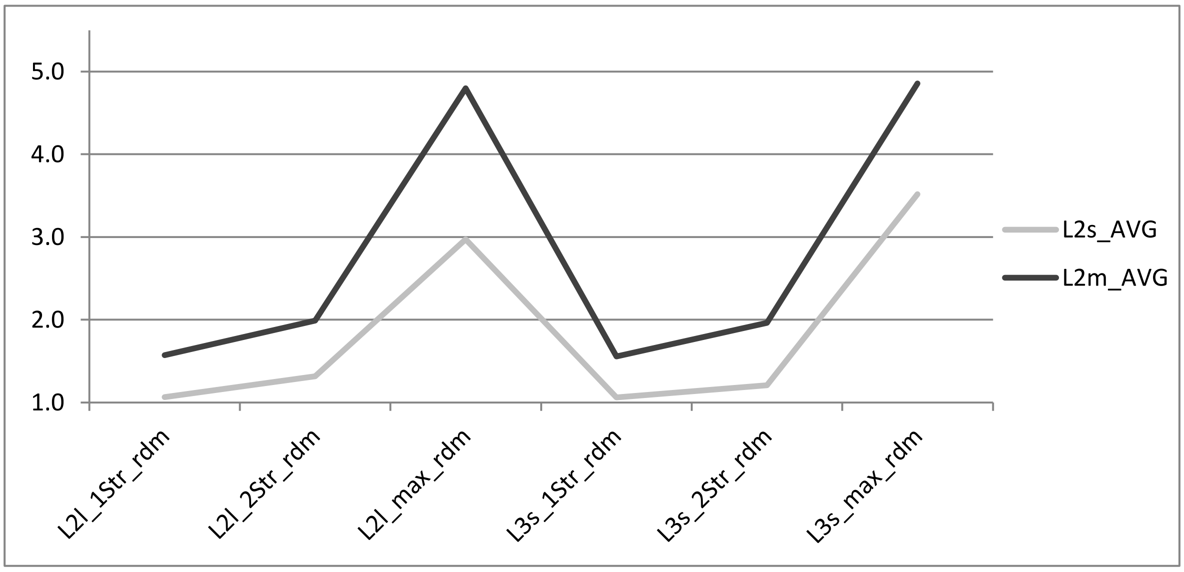


Figure 4.2.10b: L2l and L3s random induced slowdown

L2s instances are not so destructive for the larger L2l and L3s working sets, while L2m interfere more, as expected, causing slowdown up to more than 2.5

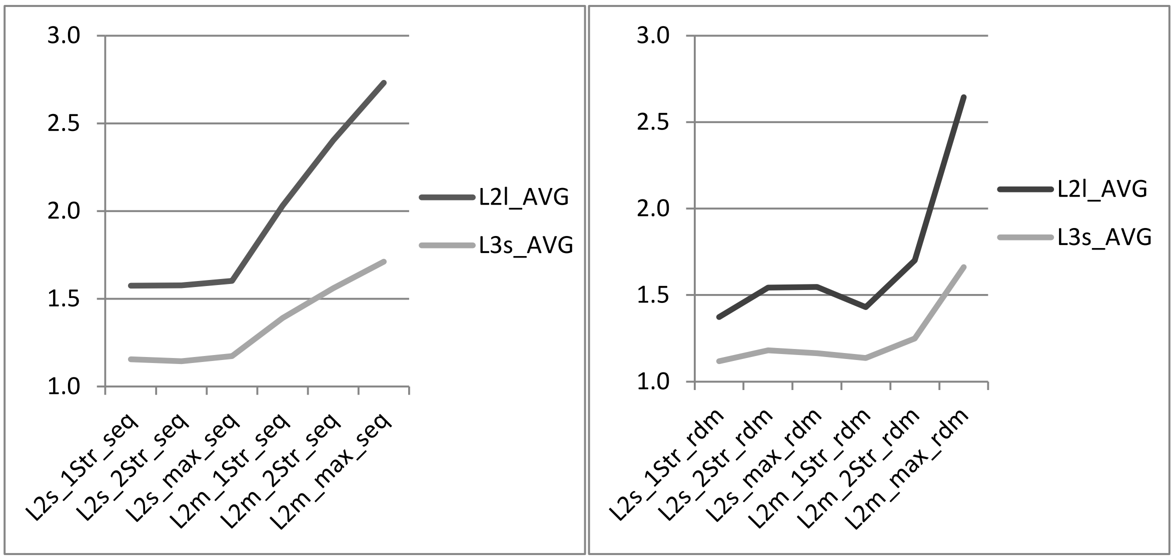


Figure 4.2.11: L2l and L3s average slowdown caused by L2s and L2m interference

As observed previously, intense slowdown effects are present when a streaming instance with large number of concurrent requests destroys cached data of a smaller instance with lower requests. This behavior is also present when larger benchmarks of C class are co-executed.

	L2l_1Str_seq	L2l_2Str_seq	L2l_max_seq	L3s_1Str_seq	L3s_2Str_seq	L3s_max_seq
L2l_1Str_seq	1.997	2.750	3.990	1.977	2.970	4.712
L2l_1Str_rdm	2.567	2.958	3.281	3.088	3.313	3.674
L2l_2Str_seq	2.182	3.012	4.642	2.181	2.956	4.717
L2l_2Str_rdm	2.708	3.038	3.528	2.980	3.013	3.824
L2l_max_seq	1.865	2.512	3.931	1.860	2.509	3.803
L2l_max_rdm	2.035	2.568	2.848	1.989	2.468	2.742
L3s_1Str_seq	1.691	2.241	3.357	2.403	2.859	4.375
L3s_1Str_rdm	1.547	1.735	1.924	1.943	2.122	2.369
L3s_2Str_seq	1.492	1.885	2.868	1.504	2.642	3.463
L3s_2Str_rdm	1.599	1.764	2.082	2.023	2.115	2.349
L3s_max_seq	1.234	1.503	2.013	1.208	1.374	2.396
L3s_max_rdm	1.356	1.603	1.712	1.327	1.571	1.923

Table 4.2.11: L2l – L3s slowdown caused by sequential instances

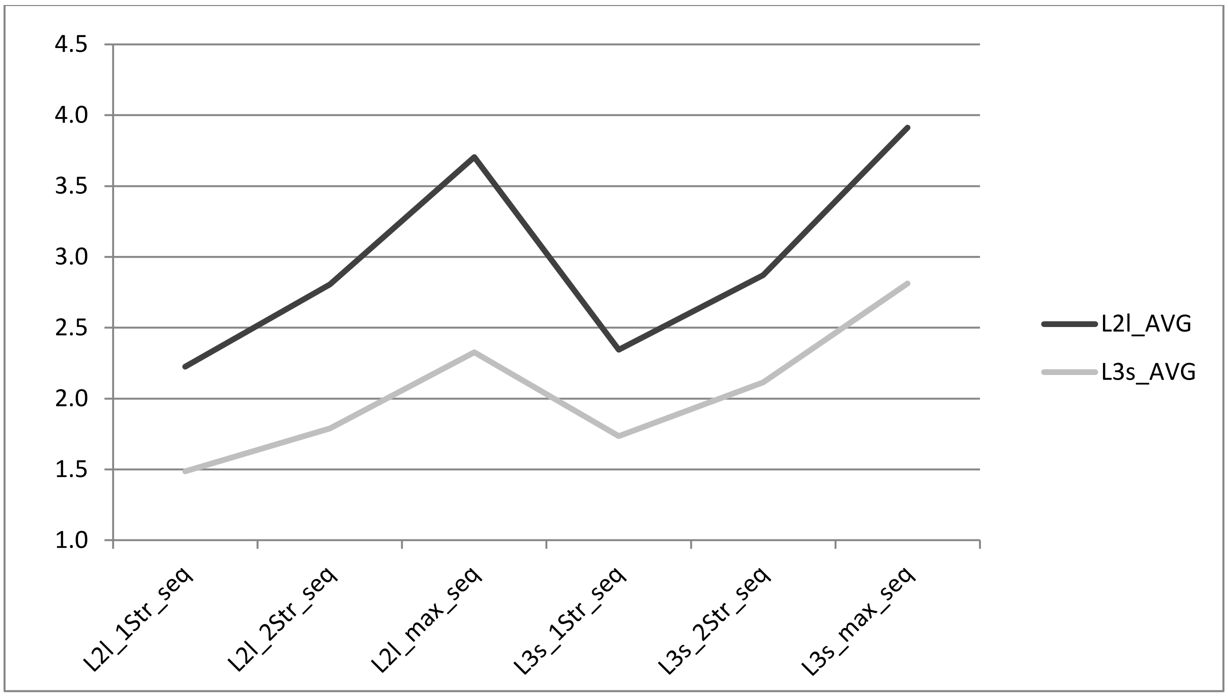


Figure 4.2.12a: L2l and L3s average slowdown caused by same classes sequential interference

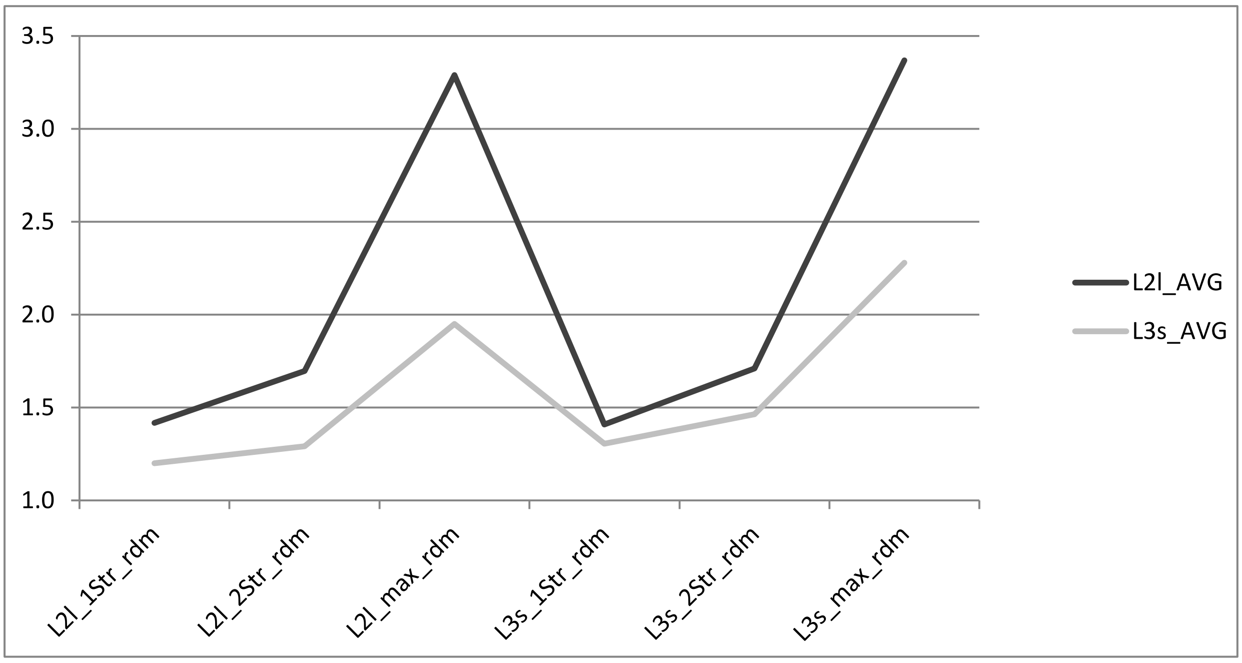


Figure 4.2.12b: L2l and L3s average slowdown caused by same classes random interference

In the latter cases, moderate contention occurs in both L3 and L2 caches. Finally, L3l_1Str_rdm behavior is different; its classification in C category was marginal and results show that in this context it acts as an LC application, not following the patterns displayed by all other applications, but the LC behavior, as will be seen in next sections:

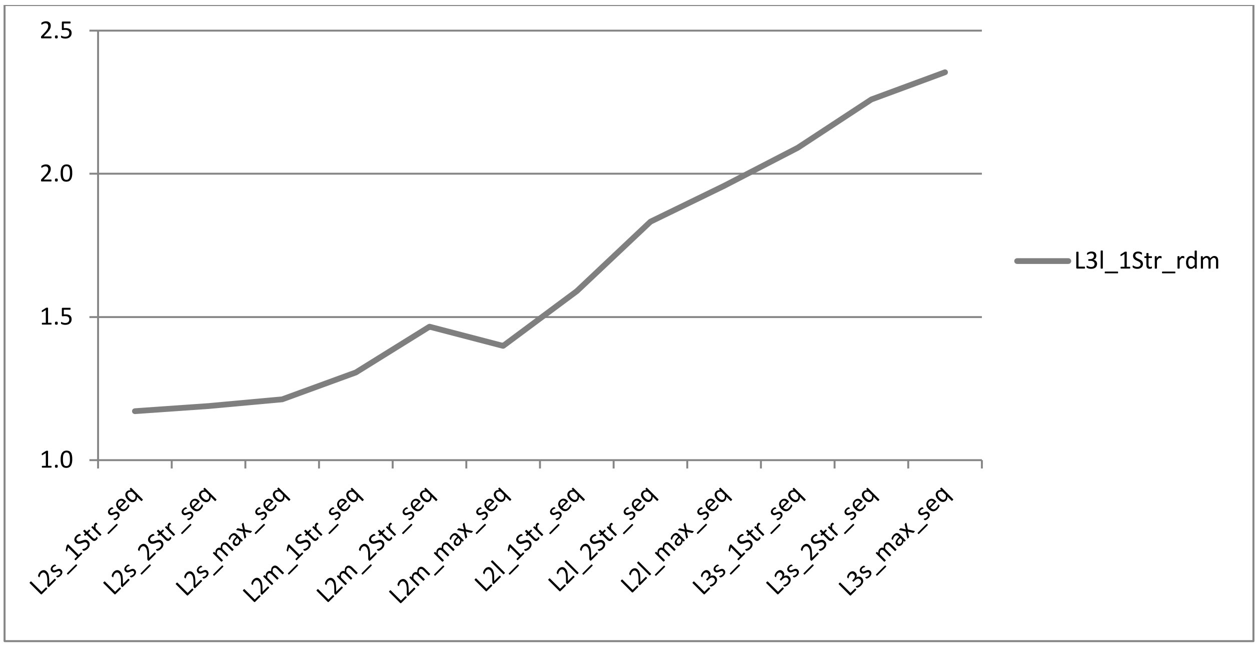


Figure 4.2.13a: L3l_1Str_rdm slowdown with all C sequential benchmarks

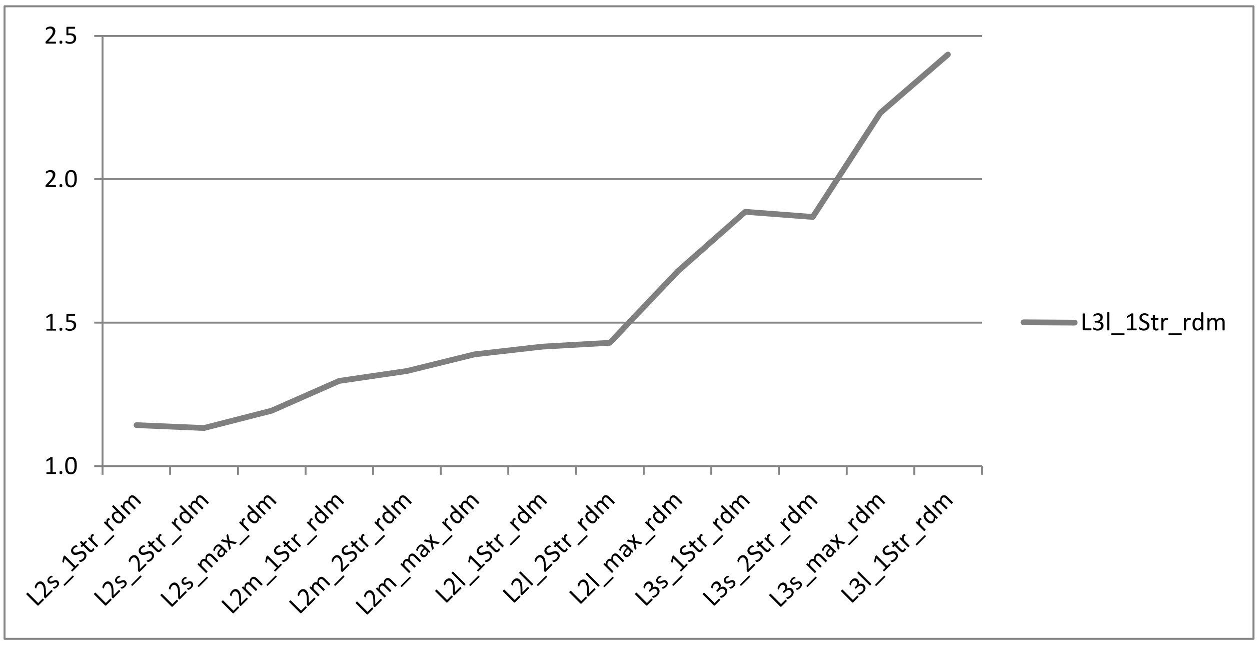


Figure 4.2.13b: L3l_1Str_rdm slowdown with all C random benchmarks

In summary, it has been confirmed that shared L2 contention may lead to high slowdown effect, which can escalate to excessive levels in conjunction with possible concurrent LLC competition. Graphs in figures 4.2.14a and 14b show overall C slowdown progression as size and requests increase, as caused by sequential and random instances respectively:

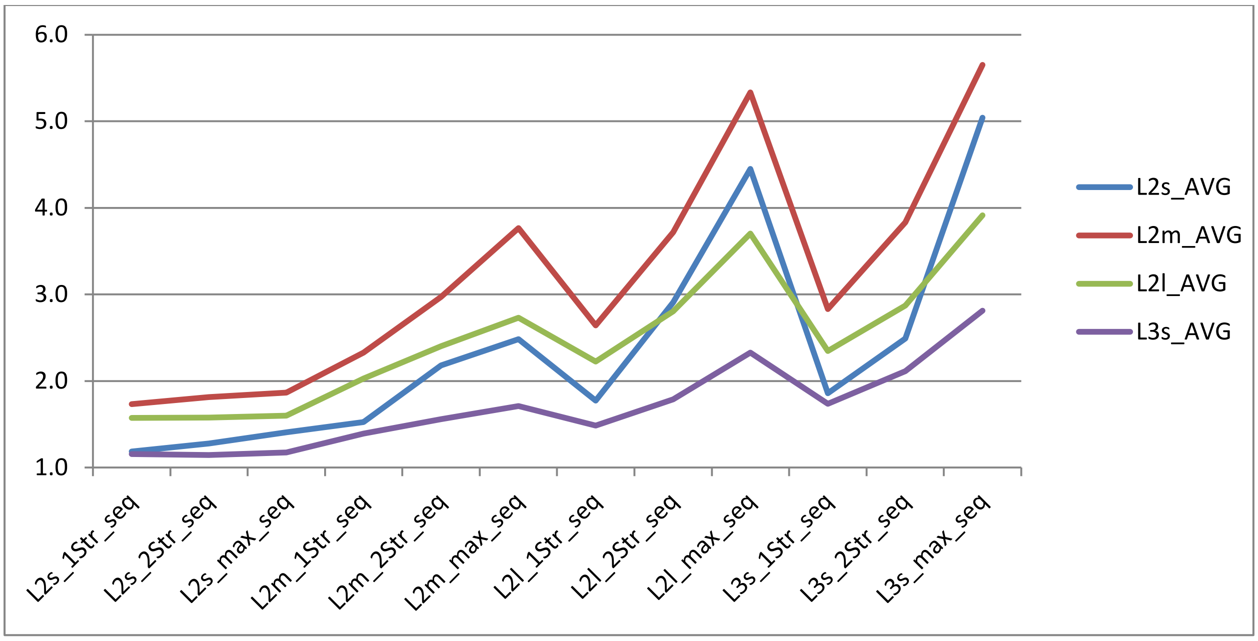


Figure 4.2.14a: Size class average slowdown with all C sequential benchmarks

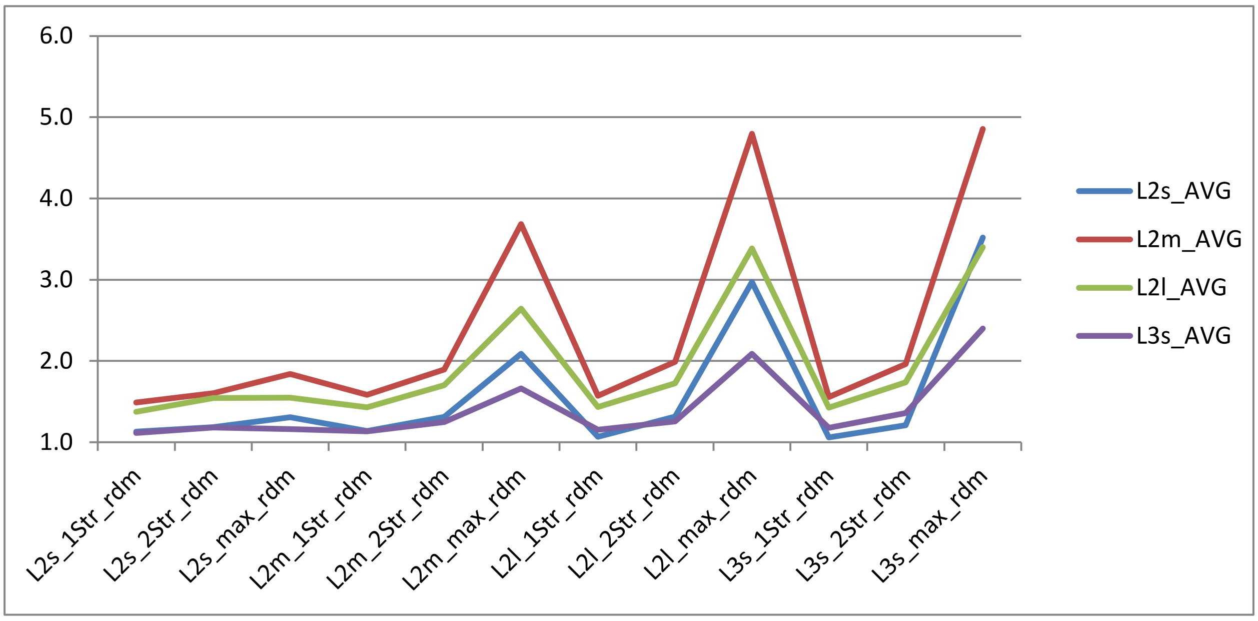


Figure 4.2.14b: Size class average slowdown with all C random benchmarks

Overall C – C co-scheduling average was found significantly increased, at a value of 2.08.

C – LC

This series of tests confirmed observed class C behavior. Table 4.2.12 contains some of the results, ranging from noticeable up to very high slowdown effect:

	L3l_1Str_seq	L3l_2Str_seq	L3l_max_seq	L3l_1Str_rdm	L3l_2Str_rdm	L3l_max_rdm
L2s_1Str_seq	1.427	1.685	2.283	1.124	1.228	1.546
L2s_1Str_rdm	1.599	2.179	3.113	1.192	1.382	3.021
L2m_1Str_seq	1.795	2.478	4.483	1.231	1.422	2.906
L2m_1Str_rdm	3.632	4.066	6.575	1.921	2.687	4.371
L2m_2Str_seq	1.734	2.338	4.499	1.220	1.355	2.550
L2m_2Str_rdm	2.821	3.452	4.588	1.671	2.065	3.992
L2m_max_seq	1.585	2.029	3.594	1.176	1.442	2.193
L2m_max_rdm	1.403	1.635	1.935	1.185	1.267	1.715
L2l_1Str_seq	1.752	2.402	4.052	1.160	1.268	2.700
L2l_1Str_rdm	2.490	8.344	9.372	1.620	1.942	8.511
L2l_max_seq	1.537	1.895	2.794	1.171	1.300	2.685
L2l_max_rdm	1.429	1.782	1.925	1.175	1.327	2.128
L3s_1Str_seq	1.676	2.753	10.615	1.064	1.189	8.411
L3s_1Str_rdm	5.261	5.888	6.070	1.275	2.364	5.700
L3s_2Str_seq	1.356	1.695	9.402	1.108	1.215	2.210
L3s_2Str_rdm	5.510	6.115	6.655	1.144	1.366	6.098

Table 4.2.12: C slowdown caused by LC applications

It is interestingly observed that results are similar to those presented in tables 4.2.10a and b, for corresponding working set size relevance: For instance L2s instances in C class co-scheduling were excessively delayed by L2l competitors, and even more from L3s. Similarly, in this test series, L2l and L3s instances suffer from combined contention on L2, L3 caches as well as moderate memory link competition, caused by L3l. Again, MEM random instances, with moderate memory bandwidth demand and low cache reuse, do not interfere significantly with the rest of the applications. Graph below (figure 4.2.15) shows average slowdown for each size category in C class:

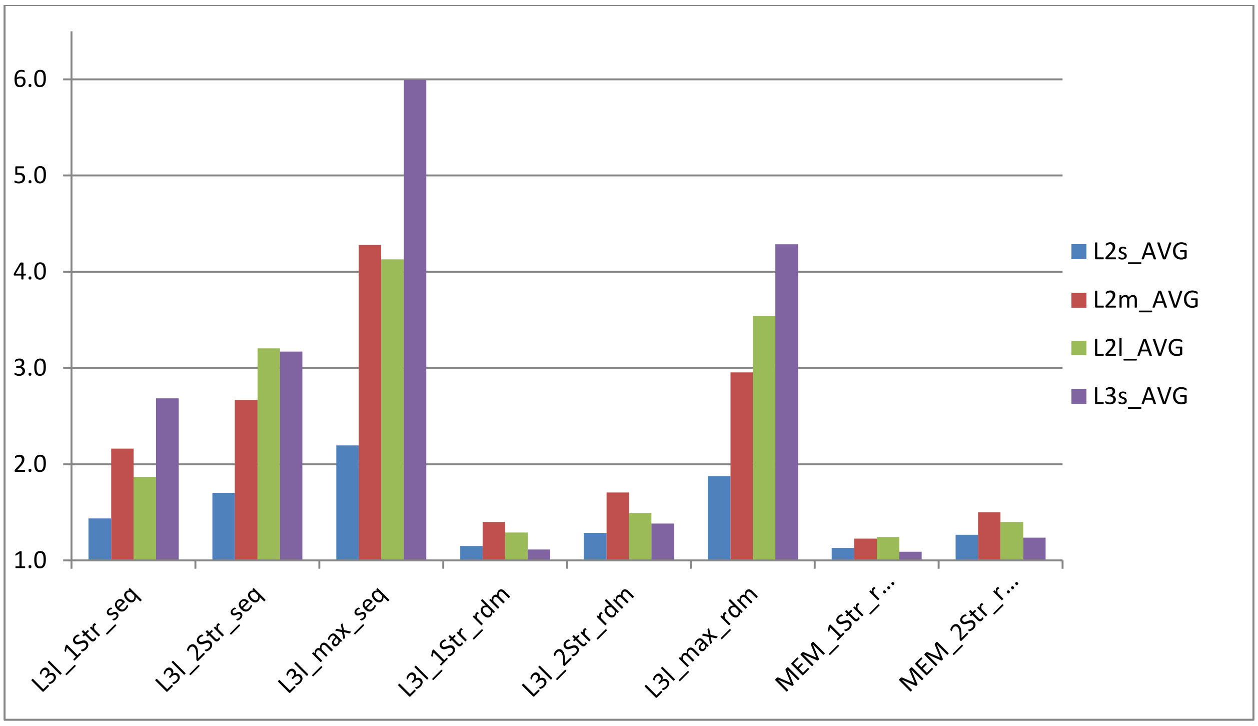


Figure 4.2.15: Size class average slowdown with all LC benchmarks

LC class is also affected, mainly due to LLC competition, with a maximum of 3.1 caused by an intensely streaming L3s benchmark. All LC instances behavior is uniform, slowdown progressing with increasing co-runner working set and number of streams:

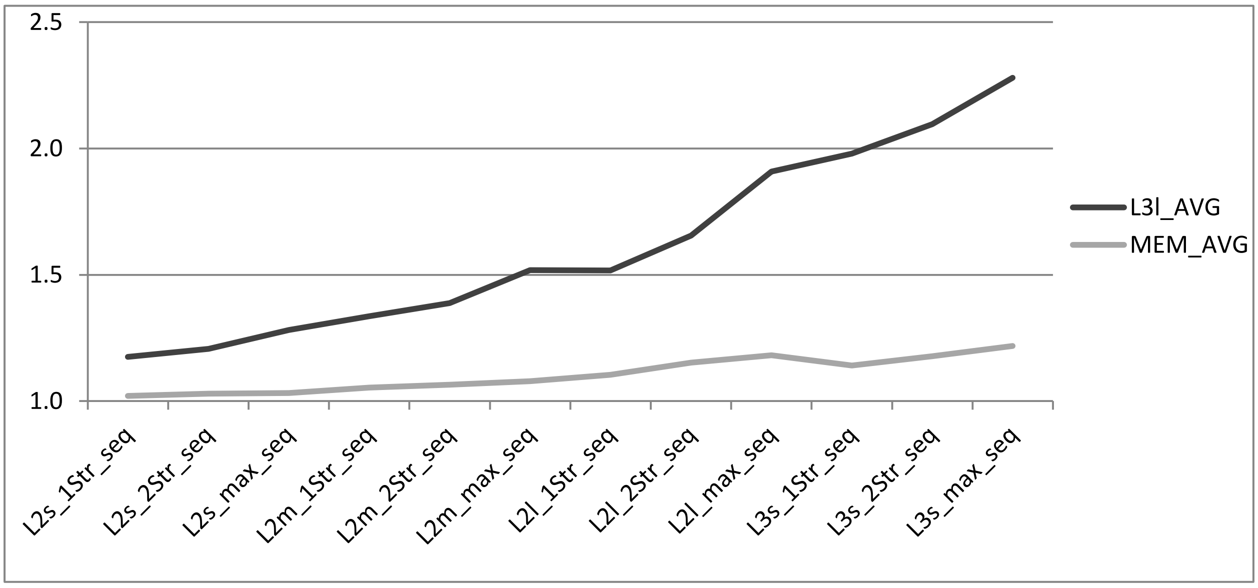


Figure 4.2.16a: Average LC slowdown with C sequential instances

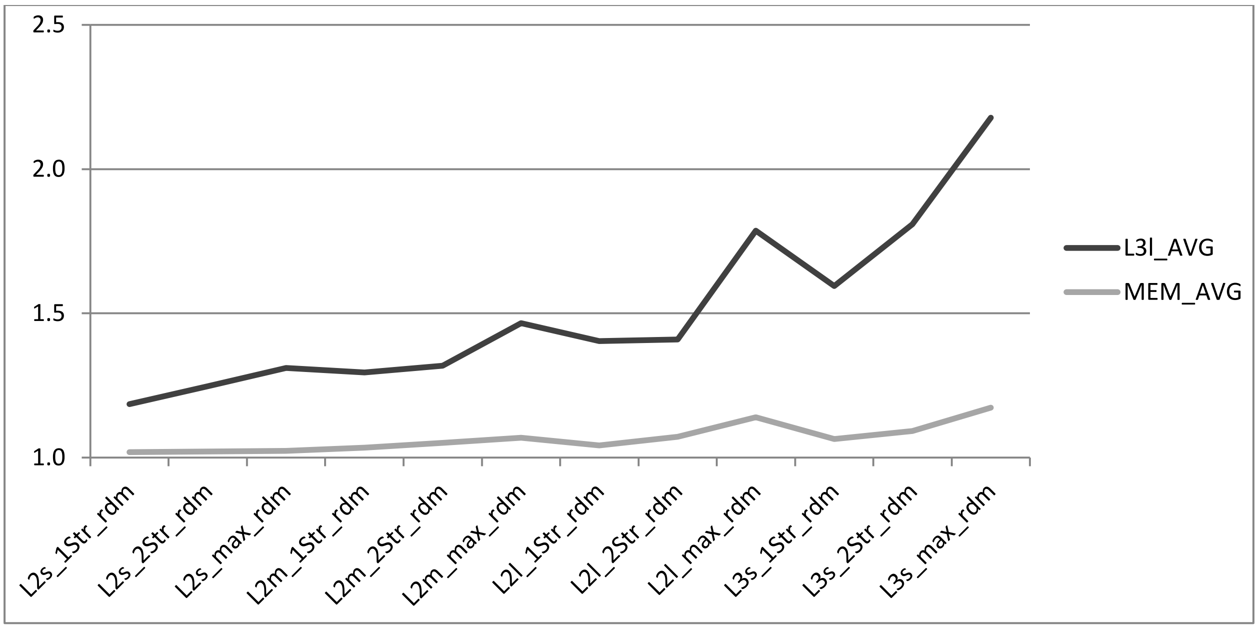


Figure 4.2.16b: Average LC slowdown with C random instances

<u>C – L</u>

Excessive slowdown scenarios were observed, confirming the estimation model. As expected, a streaming application with large working set can be very destructive for applications that normally rely on cache re-use, by massively wiping their cached data. L2s instances were the least affected with maximum slowdown noted 2.6, as their small datasets, although displaced from L2, are less vulnerable to be completely wiped of the much larger LLC; re-fetching data from LLC has greatly smaller time cost than from main memory. Increasing working set sizes of C applications, co-executed with Streaming L programs, creates simultaneous competition for all 3 shared memory levels, L2, L3 and memory Link. This results in extreme slowdown for programs with limited number of streams, as they have fewer requests, examples of which can be seen in table 4.2.13:

	MEM_1Str_seq	MEM_2Str_seq	MEM_max_seq	MEM_max_rdm
L2m_1Str_seq	1.754	2.448	5.087	2.758
L2m_1Str_rdm	3.228	3.682	4.343	3.921
L2m_2Str_seq	1.803	2.434	4.646	2.520
L2l_1Str_seq	1.851	2.526	4.500	2.638
L2l_1Str_rdm	2.348	3.164	11.157	2.697
L2l_2Str_seq	1.790	2.489	4.378	2.632
L3s_1Str_seq	1.744	2.458	12.114	6.730
L3s_1Str_rdm	6.205	6.563	6.926	6.481
L3s_2Str_seq	1.377	1.871	9.183	2.161
L3s_2Str_rdm	5.494	6.328	7.121	6.391

Table 4.2.13: C class slowdown with L instances

Overall slowdown caused by L instances is presented in figure 4.2.17. Fig. 4.2.18 shows average class C slowdown:.

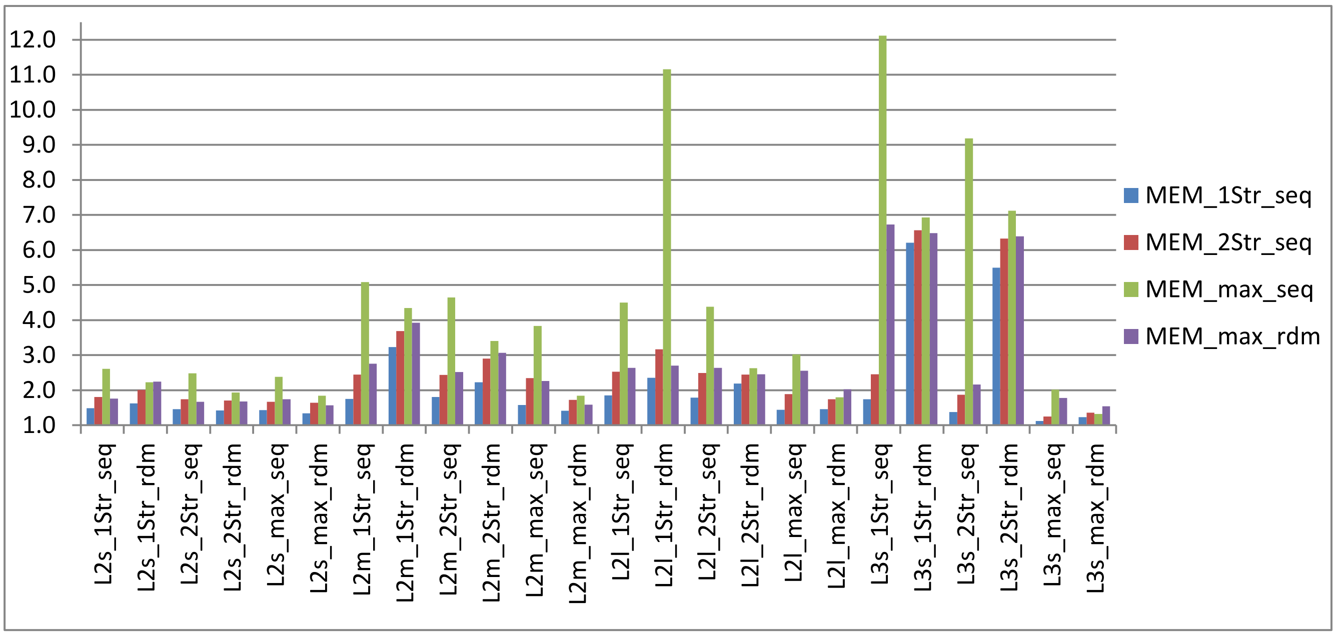


Figure 4.2.17: C class slowdown caused by L instances interference

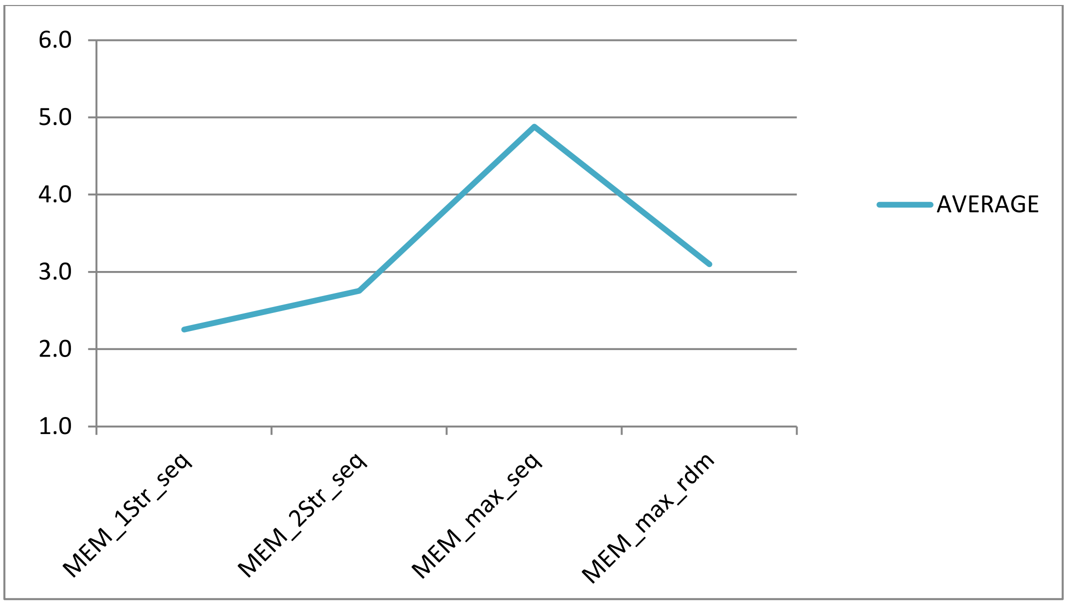
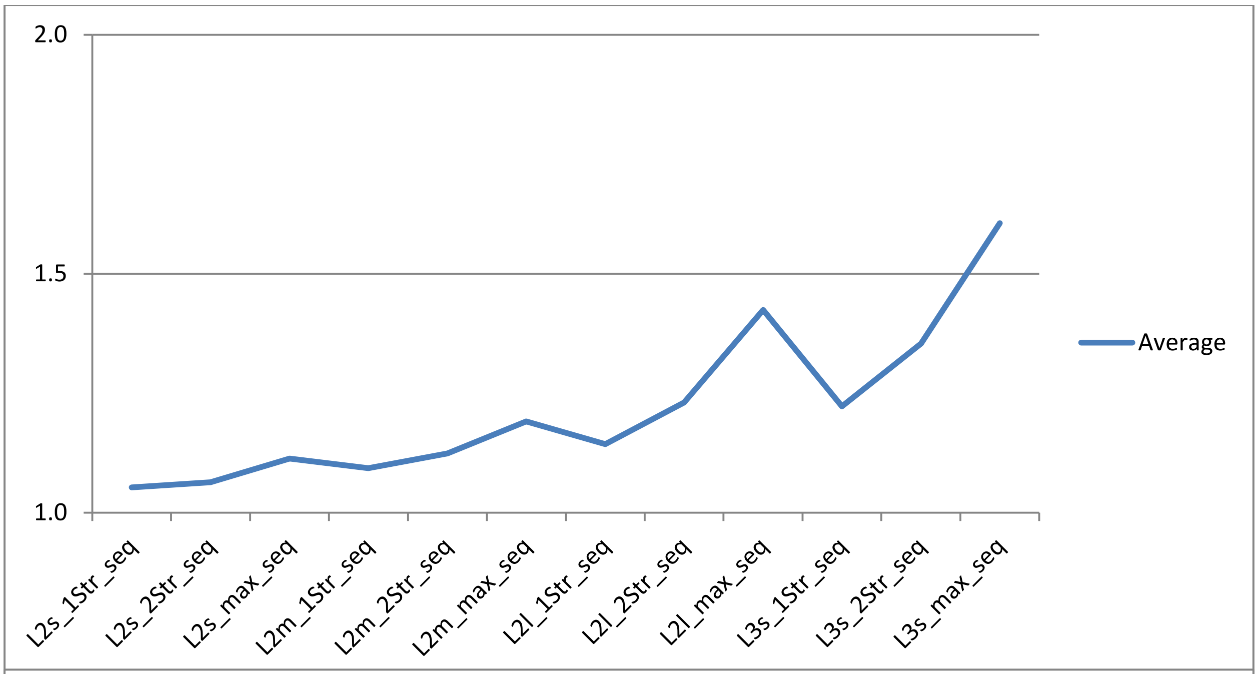


Figure 4.2.18: Average C class slowdown

L applications may also be affected by C instances with increased requests, due to serialization for memory bus usage, but, as expected, in a much lesser degree.



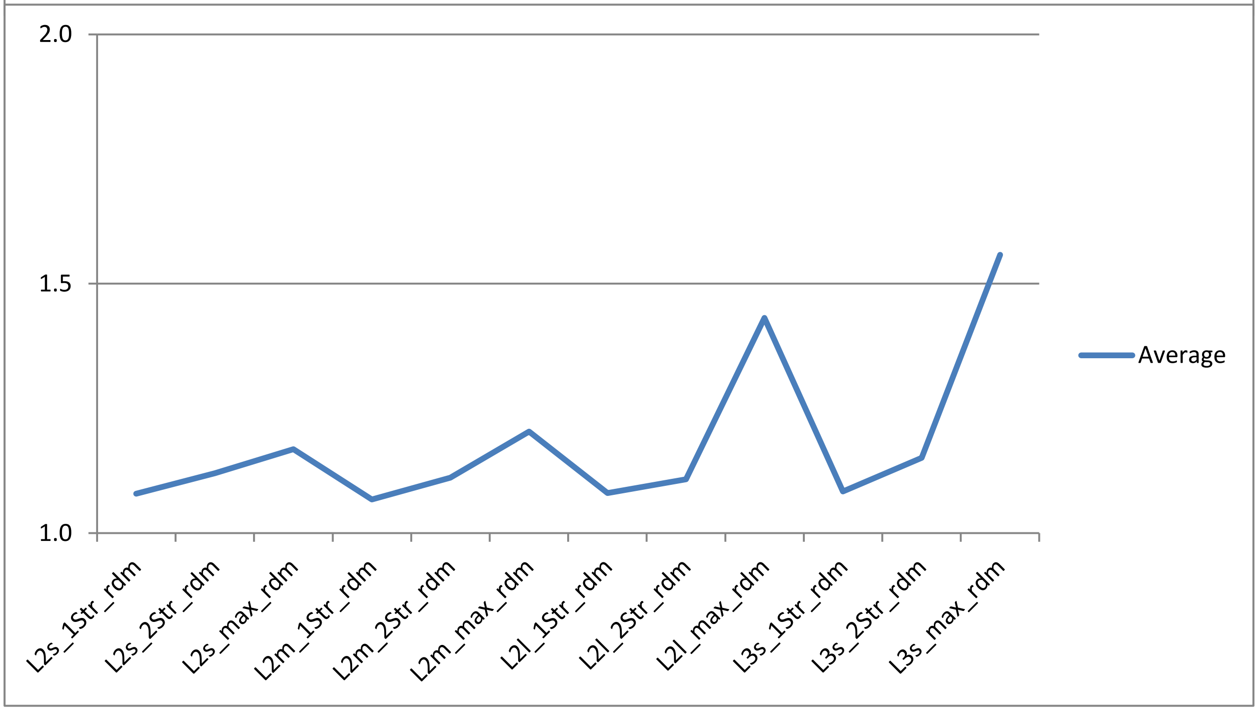


Figure 4.2.19: Average L class slowdown, by C sequential and random instances respectively

LC – LC

With both co-running applications in LC class, performance dependence on L2 utilization is insignificant. This is suggested by the results being similar to the non-shared L2 scenario:

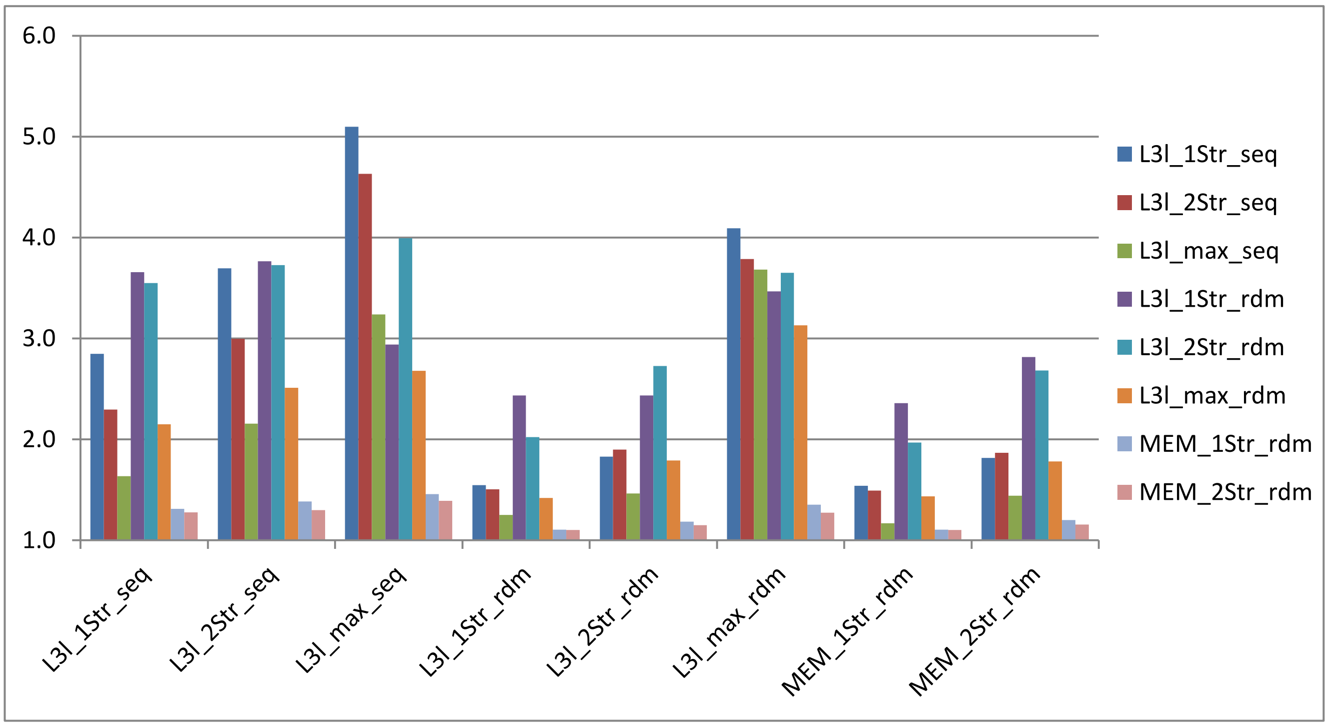


Figure 4.2.20: LC class co-execution slowdown

The effect of shared L2 is increased contention when we have streaming pattern applications, but overall behavior is same. Average slowdown is 2.3

LC – L

Exactly as estimated in our model, LCs suffered from cache contamination by the L instances in conjunction with bus contention caused by their increased requests. Increased memory access due to cache misses also affects L programs. LC instances of the same class category and pattern behave similarly, figure 4.2.21 shows slowdown per size-and-pattern:

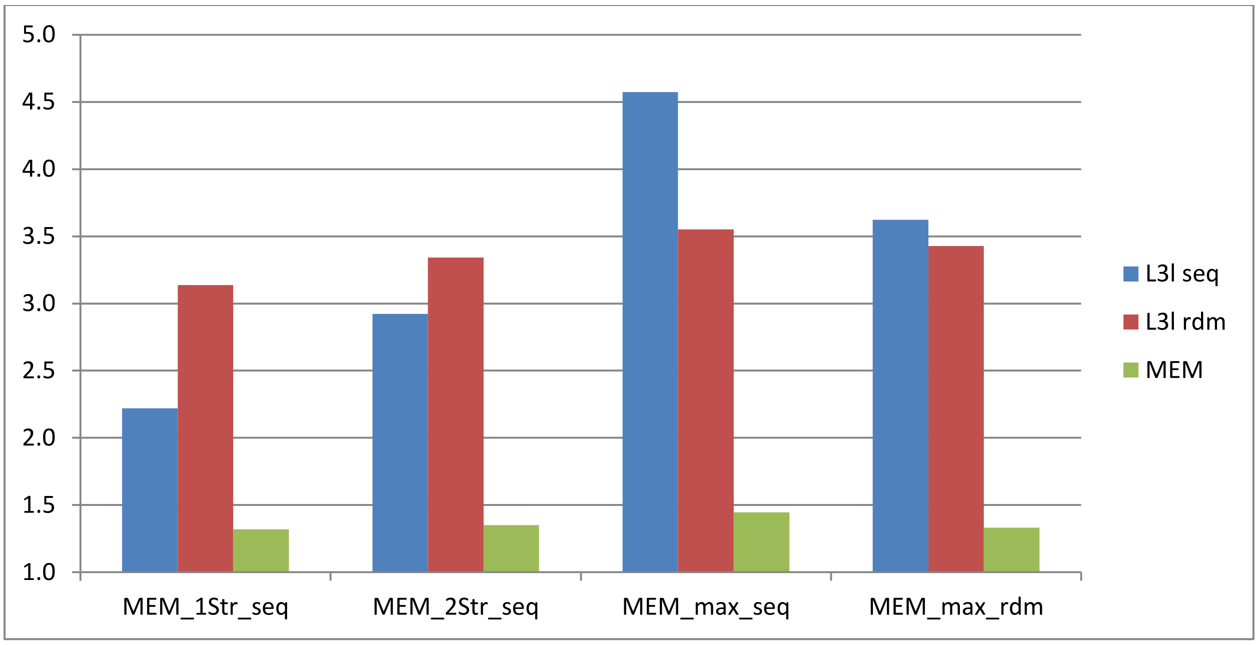


Figure 4.2.21: LC slowdown by L instances

L application slowdown was maximized by L3l sequential instances, reaching its maximum value at 3. Average slowdown of L applications co-executing with LC:

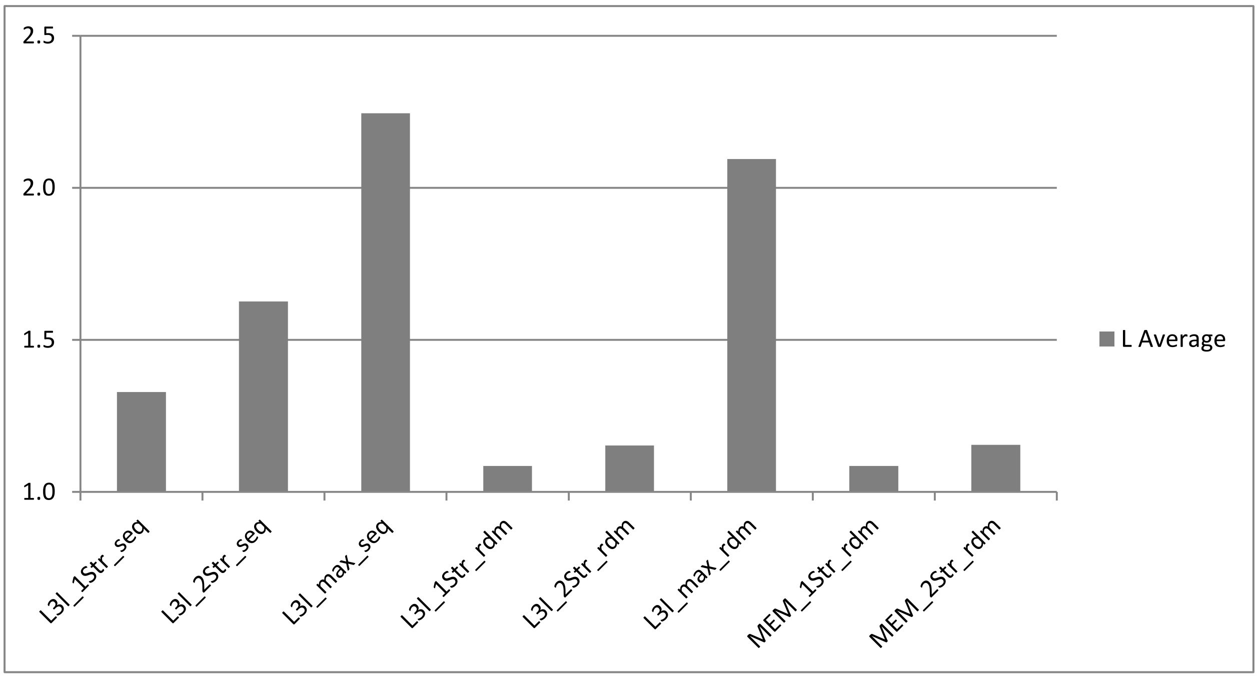


Figure 4.2.22: Average L slowdown by LC instances

L – L

This co-execution scenario's experimental results have also confirmed the predicted behavior. Maximum bandwidth streaming instances are sharing the bus and slowdown in this case does not exceed 2 (was found to be 1.85). Again programs with fewer streams are more vulnerable to higher slowdown, as requests are handled serially.

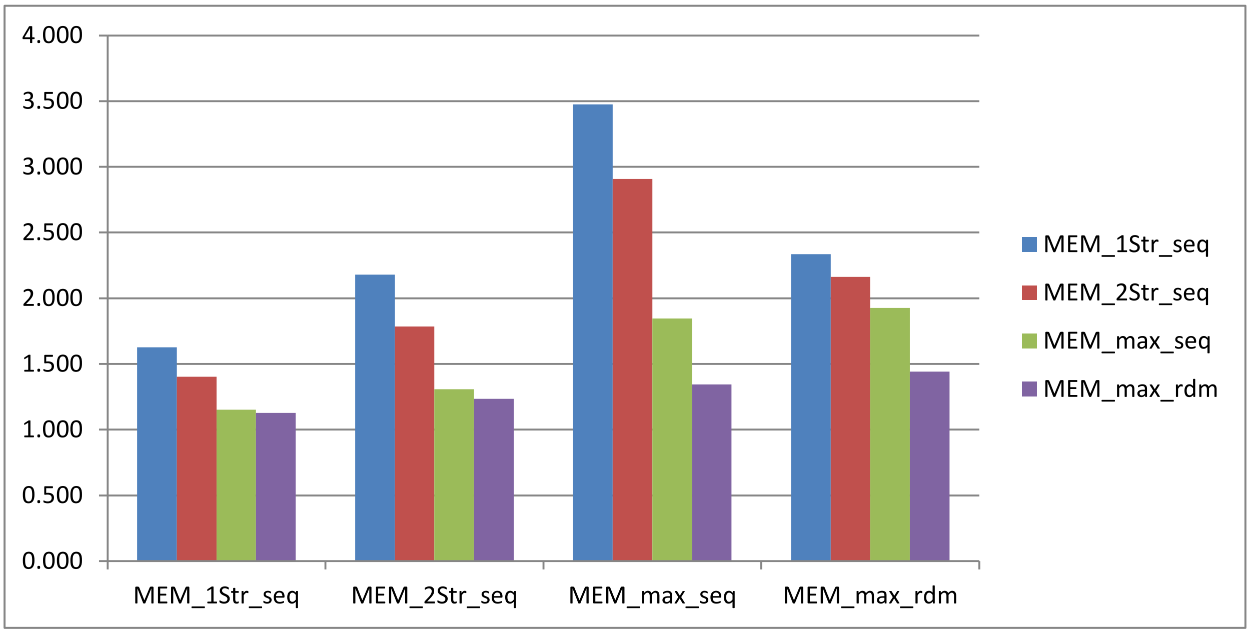


Figure 4.2.23: L co-execution slowdown

Overview

Experimenting with shared L2 cache has shown that application behavior changes drastically. Average co-scheduling induced slowdown for class pairs was found:

Class	Co-runner class		
	C	LC	L
C	2.08	2.20	3.00
LC	1.44	2.30	2.85
L	1.20	1.50	1.83

Table 4.2.14: Class average slowdown in shared L2 execution

Additionally, the contention estimation model presented has been mostly confirmed, with the system's peculiar characteristics playing a less significant role, compared to non-shared L2 experiments. It has been shown that combined contention in memory hierarchy levels can cause multiplied slowdown. As expected, C class suffers the most from this effect, with some extreme case programs having comparative slowdown over 11. To calculate this relative slowdown, we simply divide slowdown in shared L2 context, with the respective number from non-shared L2 experiments: $\frac{\text{slowdown}_{\text{sharedL2}}}{\text{slowdown}_{\text{indivL2}}}$. Indicative co-executed pairs that displayed escalating slowdown effect with shared L2 are shown:

	L2l_max_seq	L2l_max_rdm	L3s_1Str_seq	L3s_2Str_seq	L3s_max_seq	L3s_max_rdm
L2s_1Str_seq	3.836	2.230	1.090	1.804	5.134	2.337
L2s_1Str_rdm	6.330	5.139	4.806	6.376	8.121	6.575
L2s_2Str_seq	3.937	1.528	1.226	1.479	4.042	1.363
L2s_2Str_rdm	7.631	5.866	1.245	2.139	9.127	6.598
L2s_max_seq	1.708	1.540	1.271	1.397	1.624	1.322
L2s_max_rdm	1.288	1.258	0.943	1.289	1.374	1.072
L2m_1Str_seq	4.345	3.474	1.955	3.267	4.904	3.277
L2m_1Str_rdm	5.211	4.307	4.427	5.226	5.925	5.099
L2m_2Str_seq	5.400	5.202	2.155	3.444	6.928	4.459
L2m_2Str_rdm	5.143	4.453	4.478	4.462	6.094	4.735
L2m_max_seq	5.156	6.775	1.436	2.302	5.163	5.060

Table 4.2.15a: Relative slowdown ratio for selected C class pairs. The number indicates how many times slower execution was with the pair sharing L2 than with separate L2 per core.

	MEM_2Str_seq	MEM_max_seq	MEM_max_rdm
L2l_1Str_rdm	3.164	11.157	2.516
L3s_1Str_seq	2.458	11.556	5.964
L3s_1Str_rdm	6.563	6.669	5.882
L3s_2Str_seq	1.824	8.558	1.855
L3s_2Str_rdm	6.202	6.789	5.420

Table 4.2.15b: Maximum/worst-case relative slowdown ratio noted.

Finally, table 4.2.16 shows how class C and LC co-execution averages were affected from change to L2 sharing, while L was generally not affected:

Class	Co-runner class		
	C	LC	L
C	1.81	1.50	2.50
LC	1.07	1.00	1.63
L	1.00	1.00	1.08

Table 4.2.16: Overall relative slowdown.

Chapter 5

Conclusions and future work

5.1 Results evaluation

Memory contention in co-execution scenarios is very hard to predict. Results can largely differ with slight alterations in the executed applications and on different architectures. However, a well-designed prediction estimation model can be used to indicate at which scenarios the possibilities for increased interference are higher. The classification and prediction scheme described in chapter 2, according to experimental results, appears being able to capture the big picture, as in most cases behavior and interaction between processes when co-scheduled were found close to that predicted. Overall average slowdown for the systems used was found:

Class	Co-runner class			
	N	C	LC	L
N	1.00	1.02	1.10	1.07
C	1.00	1.06	1.27	2.15
LC	1.00	1.06	1.34	1.76
L	1.00	1.00	1.12	1.30

Overall slowdown map for Sandy Bridge system

Class	Co-runner class			
	N	C	LC	L
N	*(1.03)*	-	-	-
C	-	1.48	1.48	1.85
LC	-	1.21	1.67	1.69
L	-	1.10	1.36	1.39

Overall slowdown map for Dunnington system, average of L2 shared and non-shared results

Thus, avoidance of specific class combinations' co-execution, as suggested by this approach, may result in significant efficiency improvement. That, of course, does not mean that such scenarios will always have negative results because, as it was seen in the experiments, each application class includes programs with varying behavior, and such differences can be further affected by architecture features. Further division of large classes (e.g. C) to subclasses could possibly be a future choice.

It was observed that competition for shared memory resources, apart from the obvious slowdown effect, caused affected applications' general behavior to shift to that of a "higher" class (if we consider the order from smaller to higher being N, C, LC, L).

Additionally, it is confirmed that applying increasing pressure to the memory hierarchy causes slowdown effects to all applications. Figure 5.1 shows average slowdown (all programs for all classes) caused as the streaming co-running application becomes more intense and with increasing dataset size. Similar phenomena have been noted in other research works, e.g. BubbleUp [16].

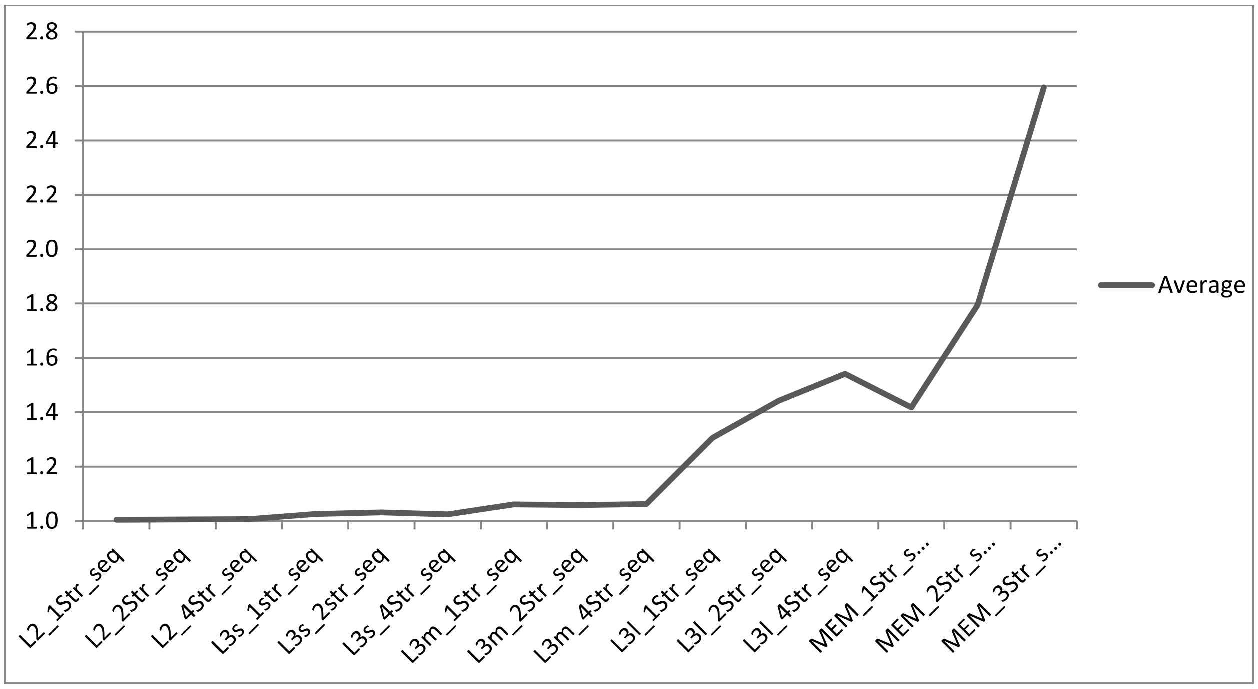


Figure 5.1: Overall average slowdown by sequential applications on Sandy Bridge system

The benchmark program created was found to be sufficiently accurate for bandwidth measurement. Its versatility, single-threaded execution and predictable behavior make it suitable for use as an in-house alternative to complement testing workloads, along with other known benchmarks and programs used for this purpose.

5.2 Future work

As an expansion of this work, more co-scheduling experiments could be conducted, using more concurrent instances on more cores and observe contention caused by multiple applications. Using class "shifting" phenomena, also observed in this work, it would be interesting to see how multi-core variations of the experiment, with more than 2 cores and instances, could be reduced, for instance to a 2- or more- class generalized scenario.

Additionally, the benchmark program could be quite easily extended to include an adjustable dummy computational module, in order to emulate CPU-and-memory intensive application for further experimentation. Such a modification could also result in being able to control, if desired, the amount of bandwidth used by the program by adding computational load when needed to limit memory requests and subsequently the contention it causes.

Bibliography & references

[1] Zhuravlev, Sergey, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. "Survey of scheduling techniques for addressing shared resources in multicore processors." *ACM Computing Surveys (CSUR)*45, no. 1 (2012): 4.

[2] Jones, Tim. Inside the Linux 2.6 Completely Fair Scheduler, http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/

[3] The Linux kernel Documentation, CFS Scheduler https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt

[4] Jiang, Yunlian, Xipeng Shen, Jie Chen, and Rahul Tripathi. "Analysis and approximation of optimal co-scheduling on chip multiprocessors." In*Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pp. 220-229. ACM, 2008.

[5] Haritatos, Alexandros-Herodotos, Georgios Goumas, Nikos Anastopoulos, Konstantinos Nikas, Kornilios Kourtis, and Nectarios Koziris. "LCA: a memory link and cache-aware co-scheduling approach for CMPs." In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 469-470. ACM, 2014.

[6] Bhadauria, Major, and Sally A. McKee. "An approach to resource-aware co-scheduling for cmps. " In *Proceedings of the 24th ACM International Conference on Supercomputing*, pp. 189-199. ACM, 2010.

[7] Blagodurov, Sergey, Sergey Zhuravlev, and Alexandra Fedorova. "Contention-aware scheduling on multicore systems." *ACM Transactions on Computer Systems (TOCS)* 28, no. 4 (2010): 8.

[8] Xie, Yuejian, and Gabriel Loh. "Dynamic classification of program memory behaviors in CMPs." In *the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*. 2008.

[9] Koukis, Evangelos, and Nectarios Koziris. "Memory bandwidth aware scheduling for SMP cluster nodes." In *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on*, pp. 187-196. IEEE, 2005.

[10] Jaleel, Aamer, Hashem H. Najaf-Abadi, Samantika Subramaniam, Simon C. Steely, and Joel Emer. "Cruise: cache replacement and utility-aware scheduling." In *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 249-260. ACM, 2012.

[11] Tang, Lingjia, Jason Mars, and Mary Lou Soffa. "Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures." In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, pp. 12-21. ACM, 2011.

[12] Lin, Jiang, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems." In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pp. 367-378. IEEE, 2008.

[13] Merkel, Andreas, Jan Stoess, and Frank Bellosa. "Resource-conscious scheduling for energy efficiency on multicore processors." In *Proceedings of the 5th European conference on Computer systems*, pp. 153-166. ACM, 2010.

[14] McCalpin, John D. "A survey of memory bandwidth and machine balance in current high performance computers." *IEEE TCCA Newsletter* (1995): 19-25.

[15] pChase benchmark. https://github.com/maleadt/pChase

[16] Mars, Jason, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations." In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 248-259. ACM, 2011.

[17] The Linux kernel Documentation, Cgroups https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt

[18] The Linux kernel Documentation, Cpusets https://www.kernel.org/doc/Documentation/cgroups/cpusets.txt

[19] Intel® Performance Counter Monitor - A better way to measure CPU utilization. http://software.intel.com/en-us/articles/intel-performance-counter-monitor

[20] Intel® Xeon® Processor E5-1600/E5-2600/E5-4600 Product Families Datasheet - Volume One http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-1600-2600-vol-1-datasheet.pdf

[21] Χαλιός, Χαράλαμπος. *Δρομολόγηση Παράλληλων Εφαρμογών σε Πολυπύρηνα Συστήματα*. Diploma thesis, School of Electrical and Computer Engineering, N.T.U.A., 2013.

[22] Knuth, Donald E. (1969). *Seminumerical algorithms*. The Art of Computer Programming 2. Reading, MA: Addison–Wesley. pp. 124–125.

[23] Intel® Xeon® Processor 7400 Series Datasheet, 2008 http://www.intel.com/Assets/en_US/PDF/datasheet/320335.pdf

Appendix A

Benchmark test results

All results in MB/s

A.1 Intel™ Xeon™ E5-4620 (Sandy Bridge)

Sequential Bandwidth (in MB/s)

Size (KB)	Number of streams								
		Single	2	3	4	5	6	7	8
1		76,684.0	57,617.8	81,926.7	101,361.4	124,983.8	126,620.6	138,612.8	158,452.0
2		75,088.9	60,402.4	88,060.8	112,735.1	139,852.2	157,666.7	170,685.8	195,100.0
4		52,744.3	65,186.1	86,840.4	119,170.0	148,604.7	172,729.0	195,819.7	220,528.8
6		47,512.5	74,974.7	100,108.8	112,983.8	129,828.2	178,846.1	203,166.9	230,476.3
8		45,260.0	78,442.8	105,630.6	127,552.8	145,710.5	162,208.3	208,051.2	235,944.6
12		43,217.6	79,014.0	109,878.8	136,454.7	160,035.3	180,407.7	192,043.1	213,608.2
16		42,257.9	79,032.8	111,928.4	141,378.1	168,441.0	192,163.0	200,749.1	225,497.5
20		41,706.1	74,932.6	110,882.3	138,851.3	173,135.3	189,925.6	202,191.4	232,973.0
24		41,346.2	78,111.0	113,649.0	146,224.7	171,001.9	206,384.5	203,931.2	238,594.9
32		40,893.9	72,536.5	103,288.7	142,248.8	153,478.4	183,356.6	184,678.0	230,070.3
38		13,168.2	26,134.6	37,675.5	38,319.8	59,253.4	59,788.2	63,556.0	68,606.7
45		13,091.2	25,948.5	37,634.7	45,587.1	55,498.9	58,037.8	62,757.9	65,374.2
56		13,008.9	25,871.2	37,413.6	38,775.9	55,295.1	56,380.5	61,229.5	64,913.0
64		12,979.9	25,722.4	37,120.6	38,694.0	55,374.8	56,586.6	63,378.9	61,492.7
85		12,894.8	25,402.3	36,696.2	47,655.4	49,255.8	58,068.3	60,651.5	64,232.3
100		12,861.5	25,485.3	36,822.4	46,099.3	51,632.6	57,445.8	61,472.4	64,867.1
128		12,810.3	25,405.3	36,676.9	44,752.7	46,074.7	57,366.0	60,664.5	58,034.6
200		11,992.1	23,380.9	29,944.9	32,800.1	38,179.0	47,494.6	49,717.1	48,382.3
256		11,089.4	21,667.7	27,333.7	23,663.3	34,515.6	38,685.5	42,360.0	45,586.5
384		10,304.4	16,665.5	21,585.7	19,678.0	27,860.9	26,681.2	28,493.6	29,630.9
512		10,261.1	16,253.8	20,590.5	20,570.7	26,362.5	27,522.5	27,581.0	26,594.4
768		10,213.4	16,260.0	20,592.1	18,142.5	26,172.9	26,895.2	27,497.8	27,708.2
1,024		10,219.2	16,256.5	20,592.5	20,047.0	26,178.5	26,879.5	27,348.1	27,709.0
2,048		10,214.6	16,244.1	20,563.5	22,956.2	26,138.7	26,832.1	27,313.6	27,689.4
3,072		10,099.3	16,095.4	20,482.3	24,226.6	26,017.0	26,692.8	27,154.0	27,317.3
4,096		10,094.8	16,094.2	20,479.1	24,225.1	26,020.7	26,616.0	27,158.5	27,318.7
6,144		10,050.9	16,073.4	20,468.0	24,210.3	25,828.2	26,489.4	27,157.8	27,231.5
8,192		10,060.9	16,083.9	20,446.8	23,845.5	25,376.7	26,070.8	26,635.6	26,755.8

Size (KB)		Number of streams							
		Single	2	3	4	5	6	7	8
10,240		9,774.7	15,953.6	20,102.7	22,911.3	24,742.0	25,227.1	25,450.1	26,582.5
11,264		9,729.3	15,969.9	19,765.4	23,373.5	23,482.6	24,778.1	25,401.3	26,427.4
12,288		9,669.6	15,804.9	19,275.5	21,104.4	21,861.0	24,205.3	25,276.1	26,563.8
13,302		9,428.1	15,224.2	18,708.2	19,954.8	20,670.3	23,442.2	23,896.6	24,252.0
14,300		9,021.5	13,946.0	17,482.9	19,931.2	20,903.4	21,886.0	22,537.3	23,851.4
15,400		8,516.4	13,316.0	17,180.8	19,066.3	20,162.6	21,158.4	21,034.4	21,487.0
16,384		7,760.6	12,880.6	16,693.7	18,122.9	18,471.1	18,191.8	18,163.4	17,991.0
20,480		6,916.9	11,802.3	14,752.6	15,111.1	15,156.8	14,889.2	14,765.5	14,703.5
32,768		6487.106	11206.05	13423.85	12957.71	13042.74	12777.82	12595.26	12525.88
45,000		6484.498	11223.3	13316.38	12828.89	12870.51	12583.86	12396.58	12388.12
65,136		6492.599	11115.37	13224.01	12706.62	12907.63	12620.56	12441.67	12366.19
78,453		6497.135	11182.61	13312.1	12657.7	12890.7	12631.37	12446.39	12368.64
131,072		6521.119	11024.08	13445.06	12820.52	12818.46	12605.11	12416.24	12368.46

Table A.1.a

Random Bandwidth (in MB/s)									
Size (KB)	Number of streams								
		Single	2	3	4	5	6	7	8
1		76,847.4	57,651.3	81,995.4	101,444.4	125,142.9	126,790.2	138,709.6	158,573.6
2		75,154.0	60,403.6	88,071.3	112,730.1	139,865.1	158,198.6	170,702.7	195,138.4
4		52,753.9	65,312.7	82,538.3	119,277.9	148,618.5	172,938.8	195,835.8	219,836.3
6		47,514.0	74,968.1	98,184.4	113,555.5	129,847.2	179,021.4	203,177.4	230,632.2
8		45,267.3	78,587.9	107,343.7	127,595.5	145,714.7	162,373.3	208,081.8	235,950.6
12		43,219.3	79,015.4	110,916.2	136,471.0	156,892.0	181,761.6	194,578.1	214,413.4
16		42,259.6	79,036.2	112,915.3	141,370.5	162,941.3	192,502.7	204,920.5	225,528.0
20		41,708.0	78,550.6	113,882.0	144,528.3	165,796.3	201,603.8	215,296.6	233,791.0
24		41,347.0	75,218.9	111,363.1	144,796.9	162,971.1	206,633.4	217,964.0	238,262.9
32		40,894.8	72,175.9	106,969.6	134,536.1	152,955.7	187,909.6	202,198.3	211,157.9
38		13,938.6	27,365.4	40,808.6	39,143.1	59,489.0	66,377.3	71,454.7	73,594.3
45		13,670.3	26,685.6	38,255.9	44,201.3	53,855.8	59,221.2	63,471.4	65,960.4
56		13,533.7	26,273.2	37,666.6	39,498.7	52,433.7	59,025.4	60,082.8	66,538.8
64		13,496.5	26,300.5	37,667.5	42,601.9	53,264.1	58,681.8	60,932.7	63,113.8
85		13,405.0	26,190.0	37,778.3	46,154.7	52,990.5	58,260.0	61,943.7	66,531.4
100		13,380.7	26,143.0	37,083.8	46,220.6	52,355.2	58,179.9	61,017.0	66,065.6
128		13,338.1	25,752.9	32,380.9	45,031.1	51,441.2	57,905.7	60,786.1	60,495.0
200		10,380.3	19,810.2	26,245.2	26,315.0	41,984.5	42,492.3	47,652.4	50,005.4
256		5,159.4	12,021.0	17,085.2	15,610.3	28,707.0	27,694.8	37,751.7	33,450.9
384		3,981.8	7,385.6	10,996.7	12,149.1	17,727.0	20,901.1	23,281.7	26,585.3
512		3,667.0	7,175.4	10,784.2	11,528.4	16,756.6	19,636.0	22,330.5	24,861.5

Size (KB)	Number of streams								
		Single	2	3	4	5	6	7	8
768		3,563.0	6,975.7	10,245.9	11,064.5	16,310.3	19,112.2	21,794.4	24,238.5
1,024		3,511.0	6,889.5	10,096.4	11,238.6	16,100.2	18,861.7	21,470.7	23,936.6
2,048		3,438.2	6,754.8	9,902.0	12,878.3	15,765.5	18,448.3	20,999.1	23,431.1
3,072		3,176.7	6,141.9	8,869.7	11,299.8	13,396.5	15,162.7	16,488.7	17,100.0
4,096		3,060.8	5,855.2	8,283.6	10,255.2	11,570.7	12,303.6	12,501.6	12,529.6
6,144		2,934.9	5,553.1	7,601.0	8,881.5	9,404.8	9,559.7	9,490.4	9,410.6
8,192		2,815.9	5,308.4	6,976.2	8,074.7	8,338.4	8,358.0	8,320.6	8,229.0
10,240		2,466.5	5,178.4	6,853.5	7,141.6	7,419.7	7,672.1	7,656.8	7,589.6
11,264		2,506.8	5,160.2	6,729.7	7,330.3	7,461.2	7,436.8	7,421.5	7,316.9
12,288		2,711.5	5,137.0	6,614.4	7,175.2	7,262.5	7,253.5	7,220.6	7,152.4
13,302		2,663.9	5,052.8	6,535.2	7,061.1	7,080.6	7,105.2	6,947.2	6,981.9
14,300		2,657.1	5,032.9	6,380.3	6,888.2	6,974.5	6,851.4	6,781.3	6,872.6
15,400		1,642.8	4,143.6	5,423.6	6,347.5	6,488.2	4,692.7	6,695.0	6,682.1
16,384		1,137.5	2,169.8	3,349.6	3,947.2	4,892.8	5,460.4	5,982.0	6,235.1
20,480		810.1	1,528.6	2,296.8	2,976.0	3,670.4	4,271.7	4,830.2	5,320.4
32,768		647.8	1,267.9	1,873.9	2,449.4	3,032.1	3,582.1	4,092.2	4,542.0
45,000		629.0	1,240.8	1,819.3	2,396.3	2,943.9	3,444.0	3,929.5	4,285.4
65,136		617.5	1,209.1	1,779.5	2,330.2	2,866.1	3,340.3	3,708.8	3,928.2
78,453		608.3	1,189.3	1,748.6	2,290.6	2,790.6	3,215.5	3,523.2	3,644.9
131,072		589.5	1,154.6	1,691.2	2,203.5	2,625.0	2,908.9	3,032.1	3,062.2

Table A.1.b

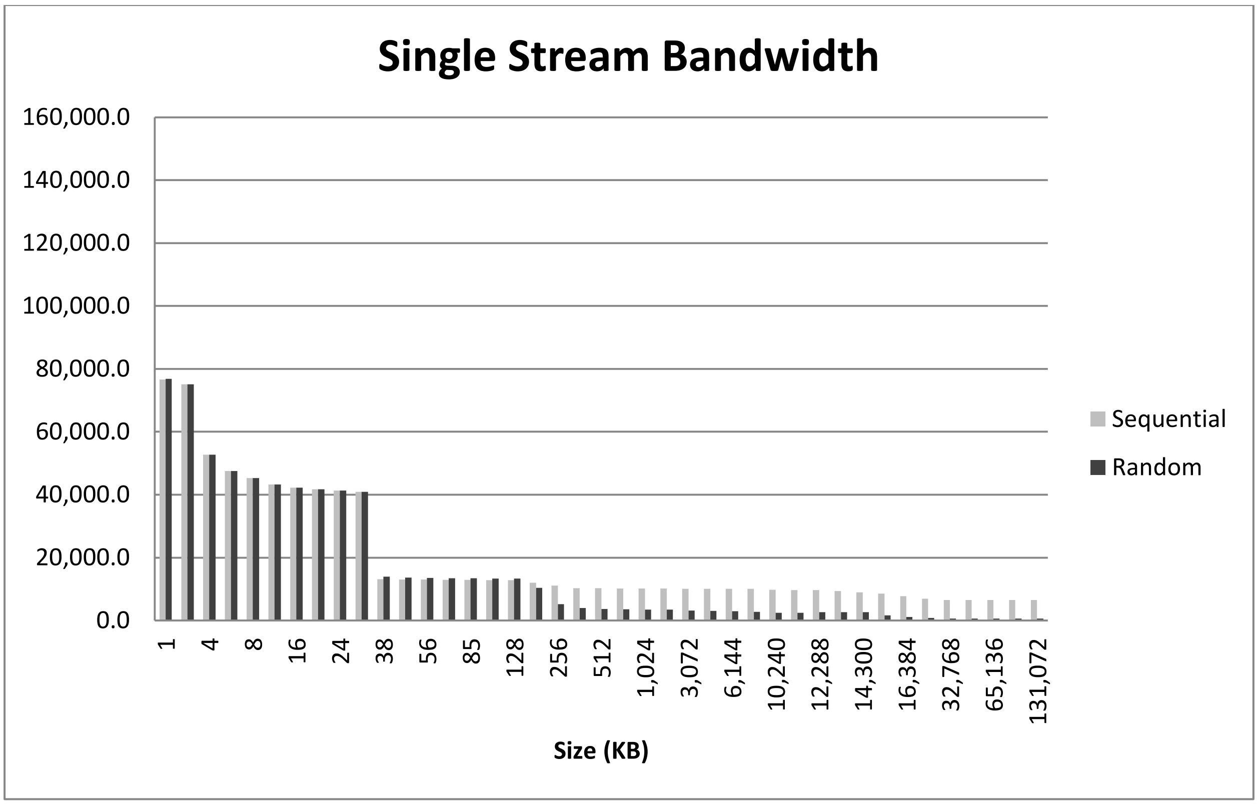


Figure A.1.1

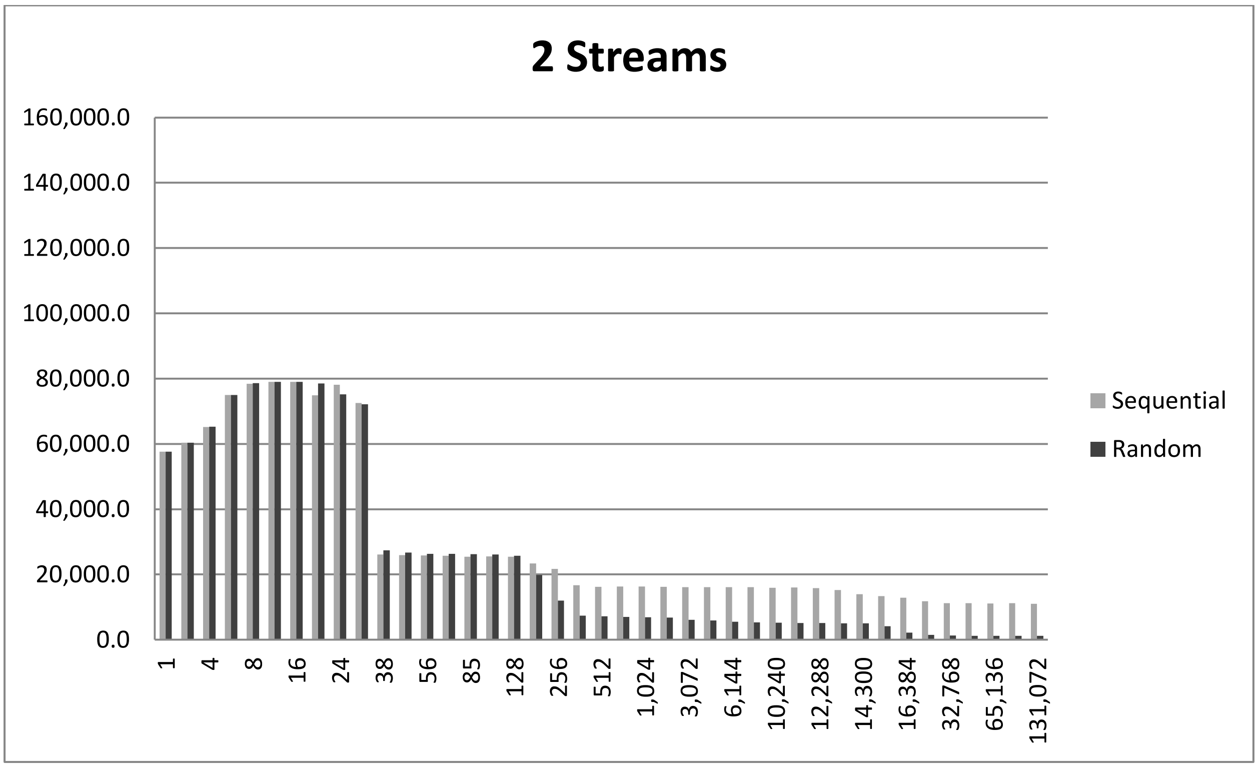


Figure A.1.2

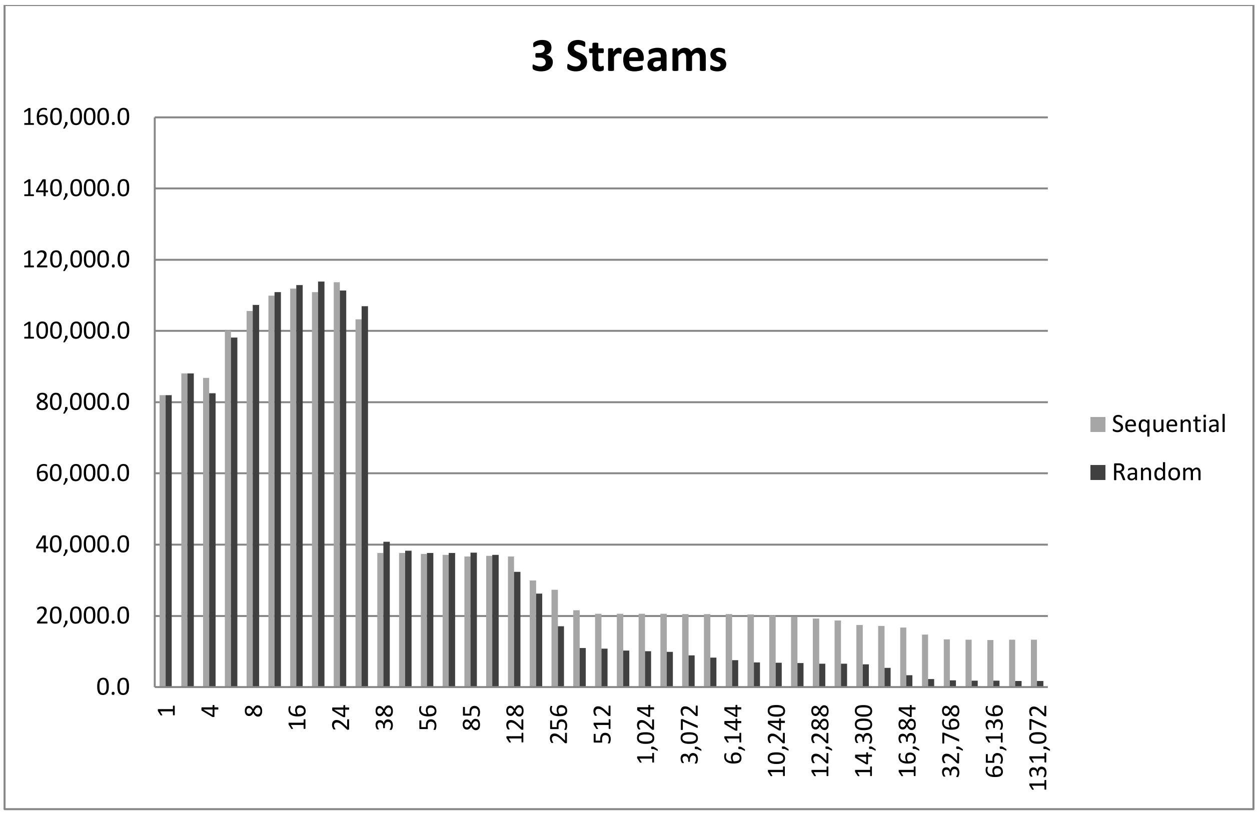


Figure A.1.3

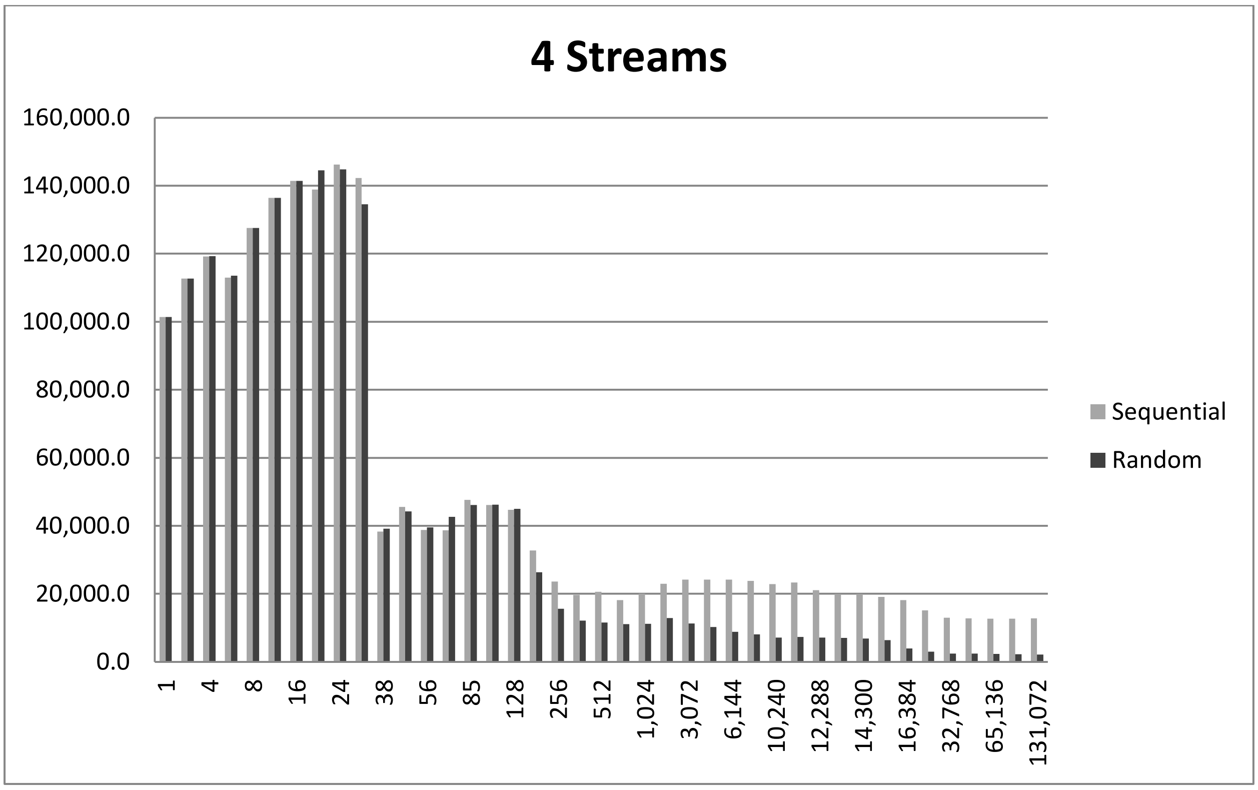


Figure A.1.4

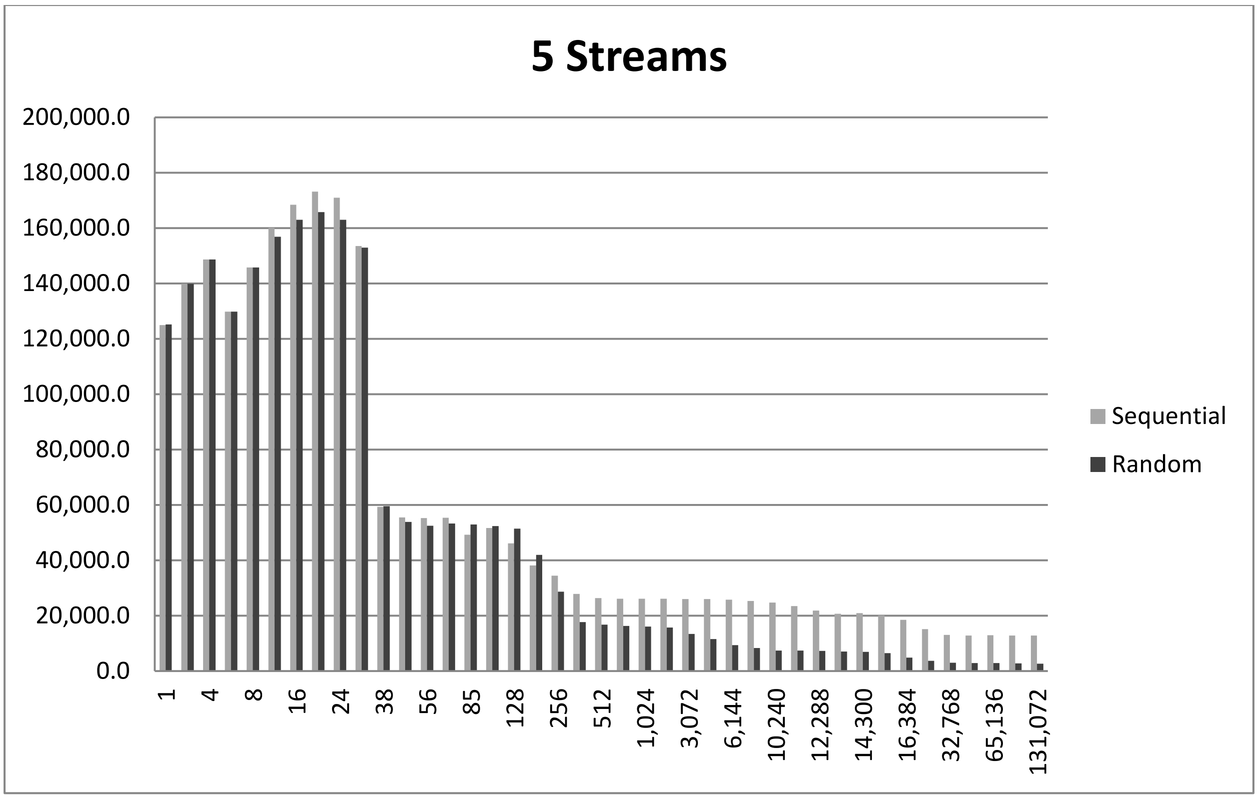


Figure A.1.5

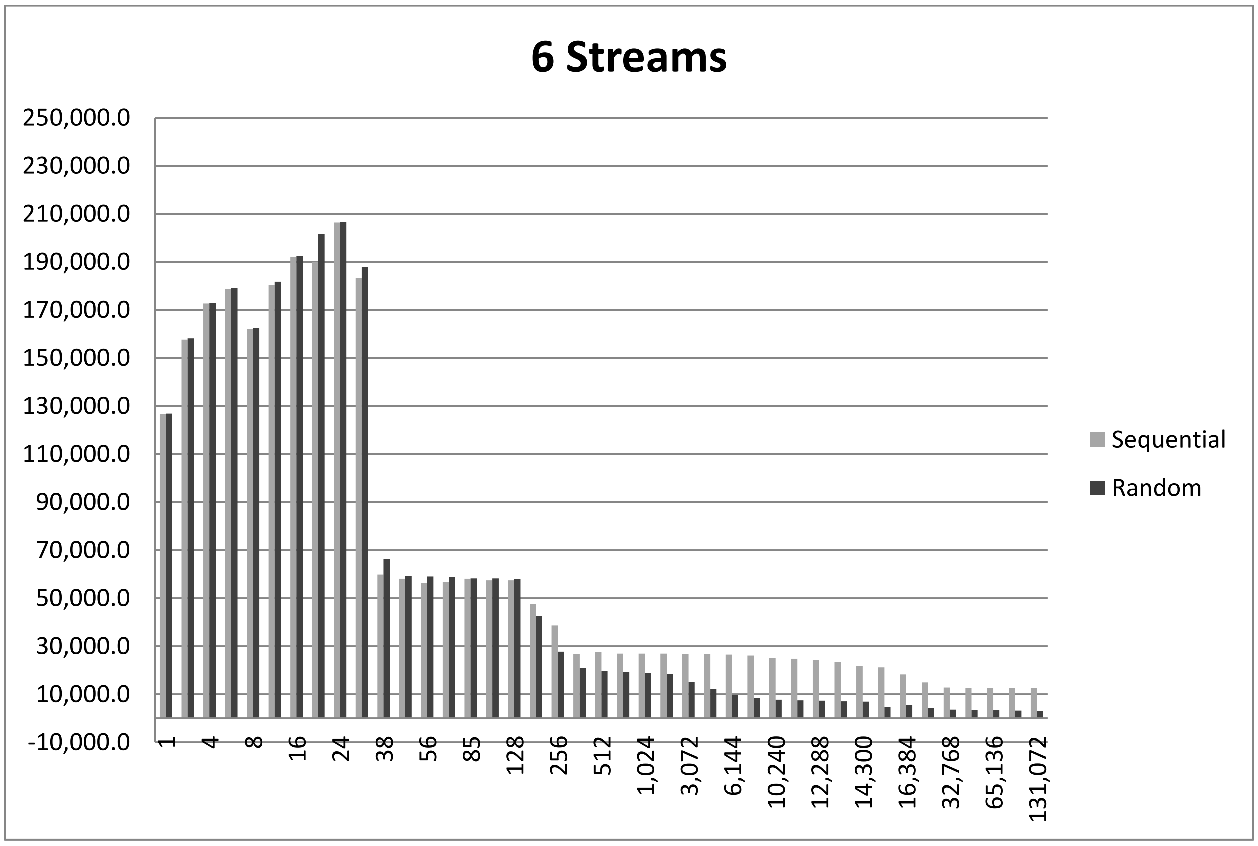


Figure A.1.6

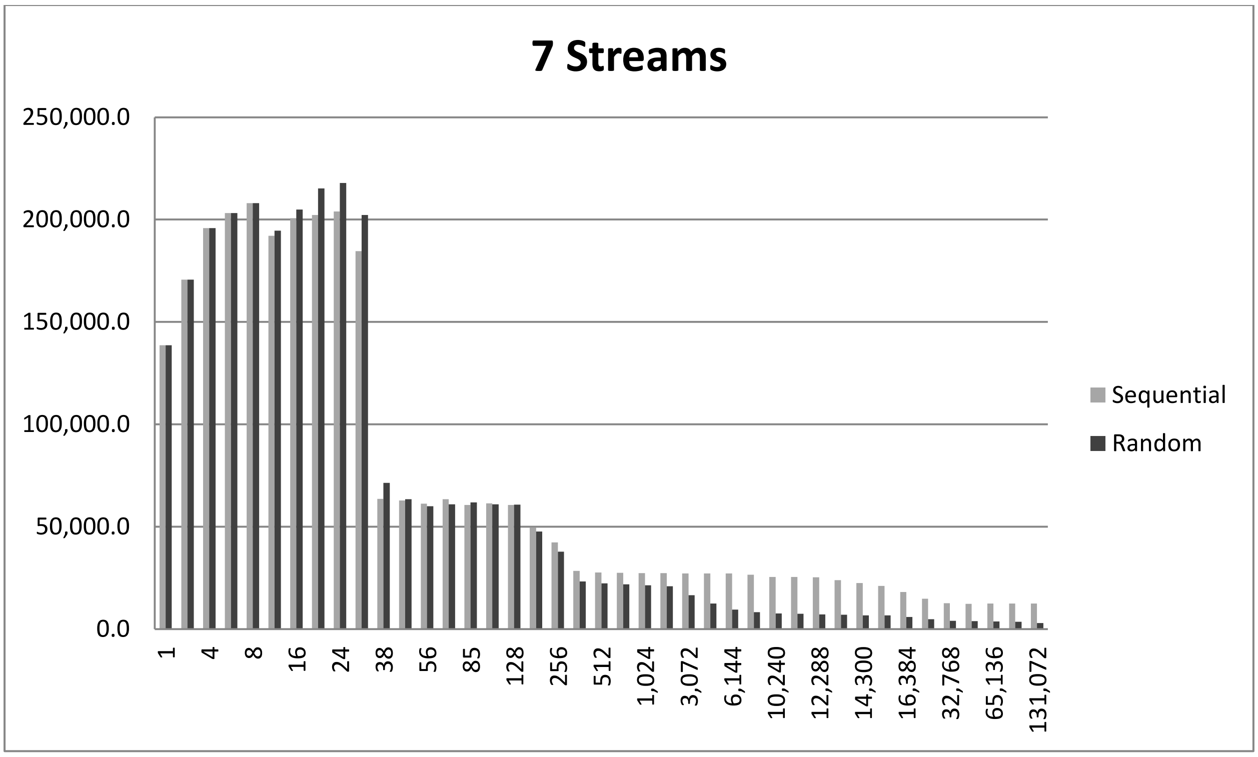


Figure A.1.7

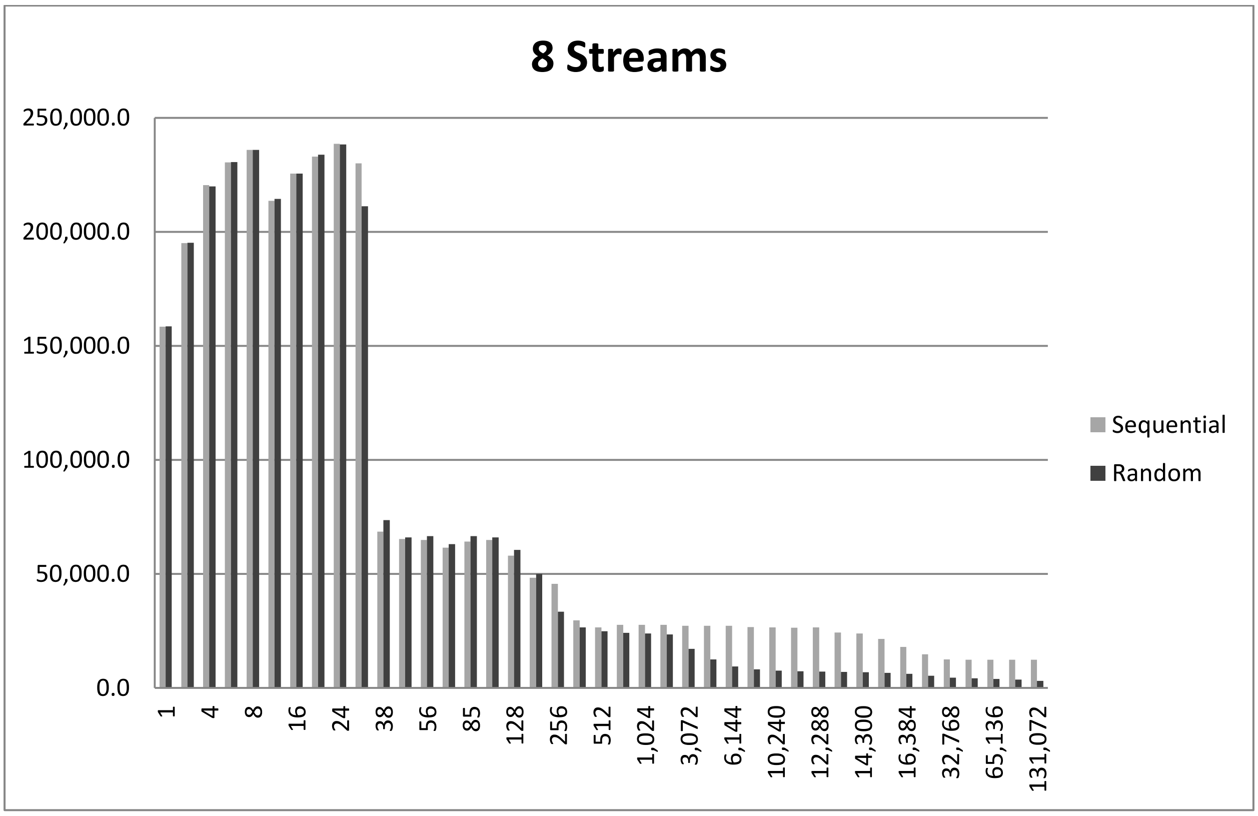


Figure A.1.8

A.2 Intel™ Xeon™ X-7460 (Dunnington)

Sequential Bandwidth (in MB/s)									
Size (KB)	Number of streams								
		Single	2	3	4	5	6	7	8
1		78,596.7	68,256.8	81,065.2	99,766.1	105,716.7	97,262.8	98,668.2	99,760.1
2		79,801.2	74,105.4	88,429.6	112,767.6	118,623.6	118,612.5	116,381.2	117,881.0
4		75,727.4	77,420.5	92,852.0	120,664.9	126,313.4	127,998.1	129,250.4	129,670.3
6		52,045.4	78,592.3	94,330.6	123,527.7	129,420.5	131,894.7	132,880.5	134,140.9
8		52,530.2	79,193.7	95,014.4	125,001.6	130,722.1	133,496.2	135,235.3	136,506.0
12		53,033.3	76,283.4	95,771.8	126,530.8	132,171.0	135,322.5	137,362.0	138,939.3
16		53,282.4	77,413.5	116,444.4	127,305.5	132,916.4	136,191.7	138,461.6	140,195.9
20		53,426.6	78,112.3	115,258.3	124,493.2	133,351.9	136,750.7	134,850.8	139,198.8
24		53,526.9	78,596.5	118,114.5	125,753.0	125,195.8	136,910.7	135,114.9	141,145.3
32		53,640.4	79,141.1	93,761.1	122,190.1	114,750.3	121,475.2	118,081.8	138,596.5
38		13,168.2	26,134.6	37,675.5	35,586.3	39,896.1	42,419.5	40,808.9	38,276.1
45		12,849.7	23,435.8	31,865.3	35,543.2	39,933.6	42,358.4	37,494.2	34,623.1
56		12,858.6	23,265.6	31,956.7	35,740.2	40,046.2	42,398.7	39,348.5	34,628.0
64		12,861.8	23,367.5	31,962.2	35,778.7	40,028.5	42,430.1	37,471.2	34,422.0
85		12,829.7	23,445.7	31,844.7	35,739.7	39,959.8	42,309.7	37,341.8	34,592.2
100		12,836.8	23,436.3	31,873.7	35,813.5	40,668.7	40,898.4	37,424.6	34,591.5
128		12,836.4	23,353.8	31,987.7	35,758.5	40,145.5	42,571.8	37,471.8	33,510.3
200		12,838.8	23,419.5	31,957.4	35,891.9	40,678.0	42,558.1	37,502.1	34,620.9
256		12,838.8	23,652.3	31,918.5	35,868.0	40,236.5	42,616.5	37,531.2	33,539.3
384		12,837.9	23,630.9	31,894.6	35,880.3	40,202.7	39,945.4	37,547.6	33,757.7
512		12,838.2	23,653.4	31,934.2	36,392.7	40,248.6	42,679.7	37,554.5	33,563.8
768		12,837.7	23,635.6	31,959.5	36,366.9	41,972.8	42,868.5	37,556.2	33,580.0
1,024		12,835.8	23,624.5	25,506.2	36,377.2	41,624.3	42,893.7	37,327.5	34,569.4
2,048		11,725.2	22,623.2	28,174.8	31,647.0	37,097.4	35,768.5	34,538.6	32,772.9
3,072		9,976.5	16,047.5	18,204.5	19,364.6	20,993.3	20,661.0	21,063.4	21,384.5
4,096		8,531.1	11,846.9	11,967.0	12,442.9	12,342.4	12,805.5	12,772.7	13,015.0
6,144		7,948.8	9,570.3	9,907.9	10,057.8	10,153.1	10,098.3	10,143.3	10,146.3
8,192		7,890.5	9,416.3	9,481.3	9,896.9	9,826.8	9,856.5	9,785.9	10,020.3
10,240		7,353.5	7,734.3	8,643.4	8,780.5	8,378.2	8,946.7	9,060.2	9,215.7
11,264		6,036.4	7,133.6	7,985.4	8,022.3	7,997.0	8,493.4	8,595.9	8,817.0
12,288		5,381.6	6,317.1	6,453.9	6,432.2	6,621.7	7,577.4	7,718.1	8,130.4
13,302		4,996.6	5,129.0	5,572.1	5,968.3	5,983.9	6,315.4	6,528.1	6,505.7
14,300		4,383.0	4,640.7	4,442.9	4,774.2	5,173.2	5,270.0	5,207.8	5,410.4
15,400		3,546.8	4,273.3	4,005.0	3,978.7	4,202.2	4,273.3	4,504.3	4,448.6
16,384		3,030.4	3,486.6	3,609.2	3,791.9	3,802.9	3,760.9	3,998.7	4,193.2
20,480		2,078.5	2,629.0	2,650.4	2,737.9	2,728.3	2,745.0	2,703.3	2,691.0

Size (KB)	Number of streams								
		Single	2	3	4	5	6	7	8
32,768		1,873.9	2,330.1	2,359.7	2,400.1	2,420.3	2,410.6	2,412.3	2,380.0
45,000		1,872.8	2,333.1	2,358.8	2,414.7	2,427.8	2,429.0	2,404.9	2,416.2
65,136		1,874.8	2,331.9	2,358.2	2,414.1	2,428.3	2,421.4	2,422.7	2,402.6
78,453		1,873.3	2,331.6	2,360.6	2,397.4	2,427.8	2,422.0	2,418.1	2,398.9
131,072		1,872.7	2,322.8	2,362.5	2,386.3	2,429.0	2,422.6	2,422.9	2,363.0

Table A.2.a

Random Bandwidth (in MB/s)									
Size (KB)	Number of streams								
		Single	2	3	4	5	6	7	8
1		78,602.3	70,102.3	93,538.2	108,083.8	105,710.5	97,269.5	98,676.1	99,757.3
2		79,806.0	75,182.6	105,738.7	117,891.0	118,629.1	118,600.9	116,373.9	117,882.7
4		75,725.7	78,006.6	113,493.9	123,547.9	126,330.5	128,010.7	129,262.8	129,674.9
6		52,049.2	78,989.4	116,164.9	125,510.6	129,411.0	131,894.1	132,887.3	134,158.3
8		52,536.7	79,492.3	117,401.1	126,543.2	130,721.1	133,504.0	135,250.5	136,497.9
12		53,036.6	76,468.4	118,798.4	127,572.1	132,205.1	135,317.5	137,369.1	138,928.2
16		53,281.5	77,555.2	115,802.4	128,106.4	132,928.9	136,191.7	138,453.9	140,197.1
20		53,425.8	78,229.6	116,909.2	124,703.9	133,359.6	136,756.9	139,123.2	140,966.8
24		53,529.8	78,691.5	112,167.8	123,195.3	130,356.8	137,074.0	139,547.3	141,284.6
32		53,634.6	79,147.2	96,329.2	114,466.4	120,876.0	122,764.6	125,908.8	128,006.2
38		13,938.6	27,365.4	40,808.6	39,606.4	47,208.8	41,838.2	41,715.1	38,784.2
45		10,894.2	21,355.7	32,016.7	40,604.9	48,372.0	45,322.8	38,097.5	36,427.4
56		10,798.3	20,338.0	30,400.1	39,720.2	47,034.1	40,999.2	37,902.5	36,338.3
64		10,800.2	20,274.8	30,179.1	39,890.4	48,091.6	41,039.5	37,787.4	36,240.7
85		10,439.5	20,243.6	30,192.7	39,864.6	47,421.3	40,027.4	37,525.8	36,077.1
100		10,333.1	20,176.4	30,102.8	40,048.5	46,533.5	39,714.9	37,414.0	35,928.6
128		10,130.8	19,977.3	29,953.1	39,677.8	45,465.5	39,182.1	37,060.0	35,895.1
200		9,903.4	19,671.8	29,371.0	38,849.2	44,243.9	38,668.4	37,074.2	35,719.0
256		9,826.0	19,558.1	29,200.6	38,639.6	43,788.0	38,544.6	37,047.8	35,683.8
384		9,718.7	19,383.5	28,973.2	38,278.8	42,964.3	38,347.8	36,833.9	35,543.2
512		9,676.0	19,305.1	28,852.3	38,121.3	43,517.2	38,199.9	36,839.3	35,527.0
768		9,625.2	19,220.3	28,775.8	37,745.9	43,024.9	38,209.8	36,693.1	35,438.9
1,024		9,599.0	19,126.4	28,421.8	37,544.5	42,429.3	36,947.8	34,475.0	33,625.4
2,048		7,167.0	11,504.5	18,523.7	21,323.7	23,443.3	22,949.5	26,250.9	25,890.6
3,072		3,778.5	7,475.8	9,132.6	11,340.2	13,203.9	14,974.1	16,037.1	16,934.3
4,096		2,515.1	4,849.1	6,773.6	8,576.0	9,836.0	10,738.8	11,383.2	11,887.9
6,144		1,931.1	3,683.4	5,285.9	6,554.0	7,512.0	8,348.9	9,107.9	9,545.6
8,192		1,698.4	3,308.3	4,754.7	5,929.0	6,884.9	7,768.4	8,602.4	8,873.0
10,240		1,585.0	3,068.6	4,077.9	5,628.9	5,614.1	6,302.3	7,013.4	7,612.6

Size (KB)	**Number of streams**							
		Single	2	3	4	5	6	7	8
11,264		1,537.2	3,033.7	2,974.3	5,512.5	6,322.6	7,375.0	6,604.0	8,302.3
12,288		1,495.2	2,369.9	2,691.9	5,414.4	5,813.0	6,902.7	5,891.8	6,383.1
13,302		1,415.9	2,160.7	2,835.5	2,894.0	5,357.8	5,473.6	5,889.9	5,330.4
14,300		1,233.3	1,713.0	2,635.8	2,885.1	4,576.3	4,814.0	5,036.5	4,844.5
15,400		965.1	1,508.2	1,912.2	2,516.5	3,430.2	3,661.8	4,157.7	4,502.3
16,384		928.3	1,582.1	2,257.1	2,481.7	2,894.9	3,314.3	3,582.2	3,777.4
20,480		643.2	1,236.3	1,560.6	1,939.2	2,190.2	2,345.3	2,463.7	2,571.3
32,768		427.1	777.3	1,077.7	1,337.7	1,554.3	1,755.2	1,934.3	2,069.0
45,000		384.7	705.5	973.7	1,221.6	1,450.2	1,659.3	1,854.6	2,006.4
65,136		359.8	659.1	907.0	1,144.8	1,377.5	1,594.3	1,804.5	1,958.9
78,453		350.8	640.7	884.3	1,117.8	1,346.4	1,567.7	1,774.5	1,931.9
131,072		334.1	609.9	839.2	1,061.2	1,286.5	1,497.7	1,699.4	1,852.7

Table A.2.b

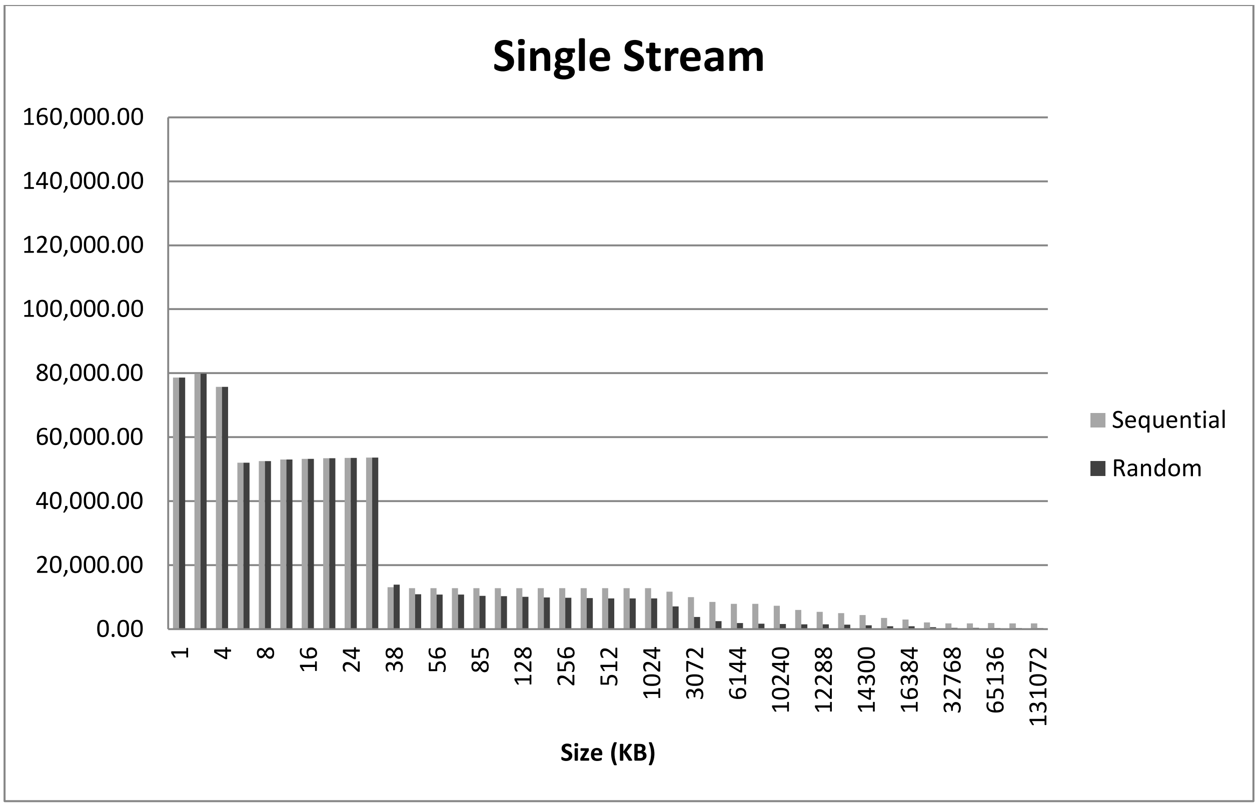


Figure A.2.1

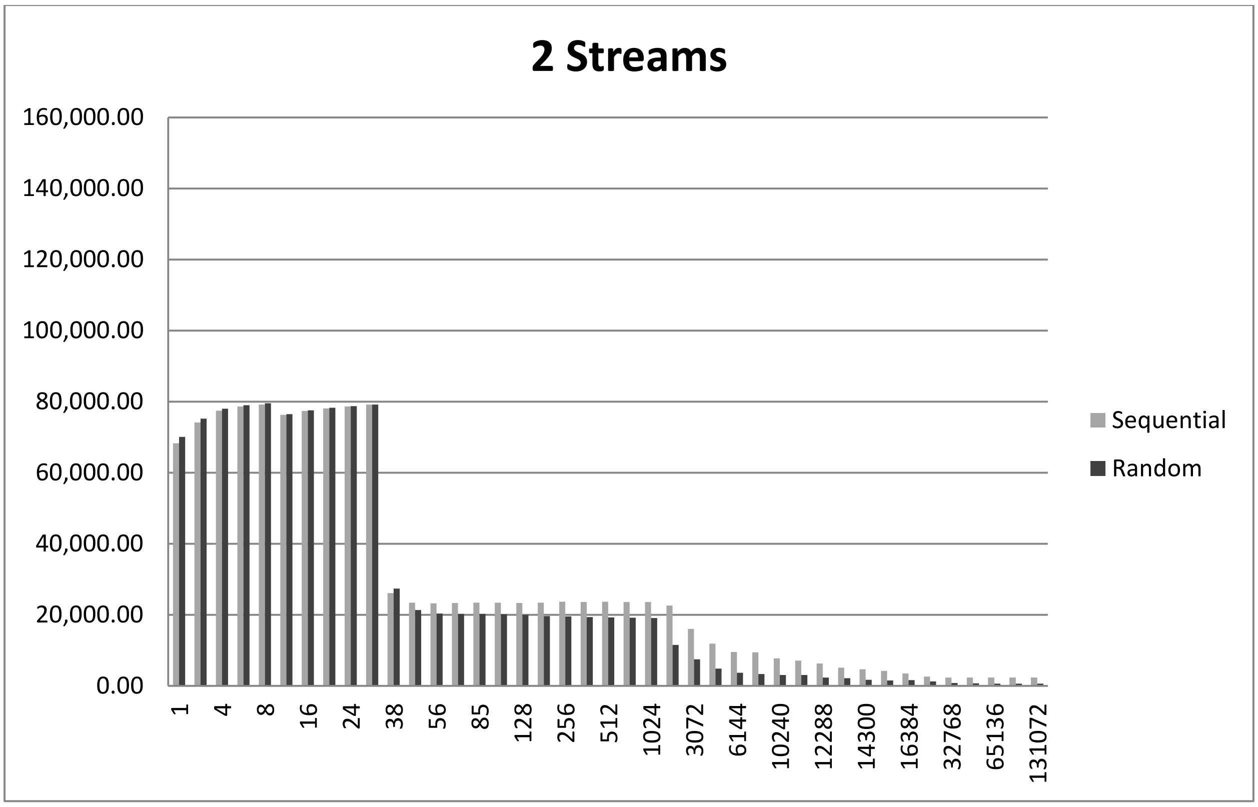


Figure A.2.2

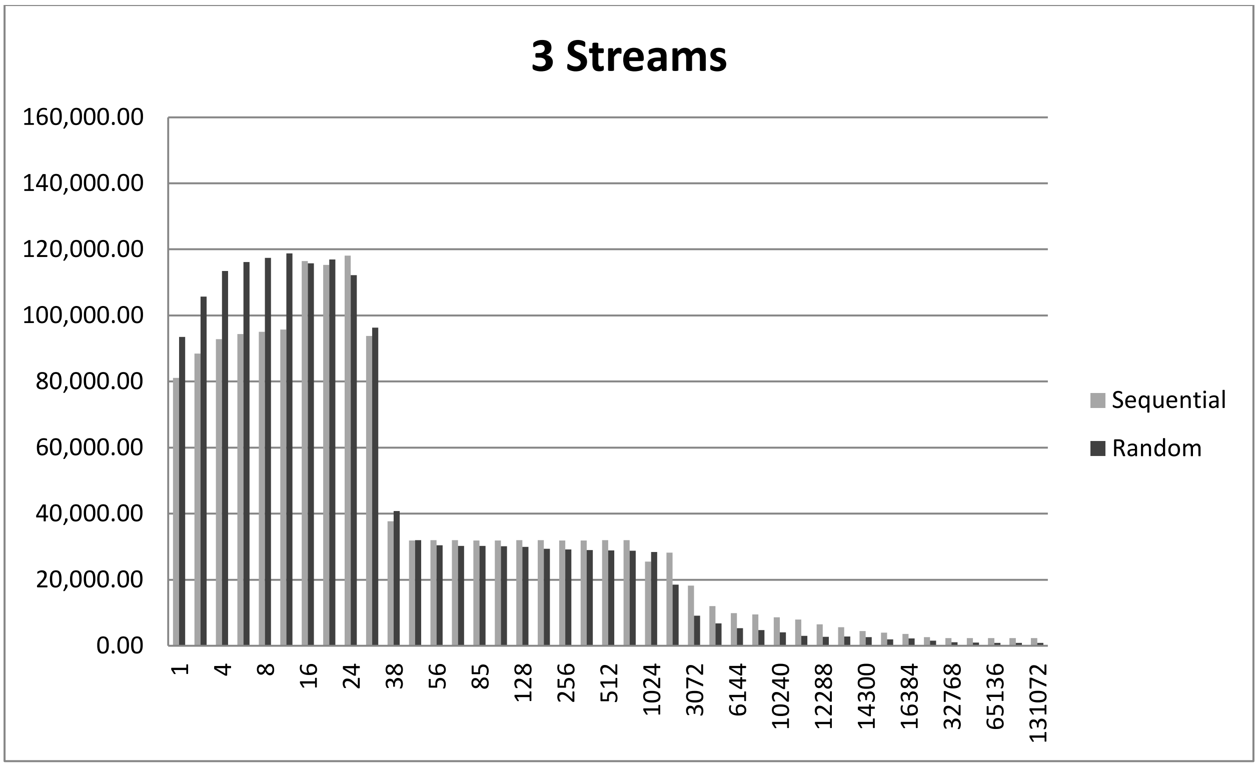


Figure A.2.3

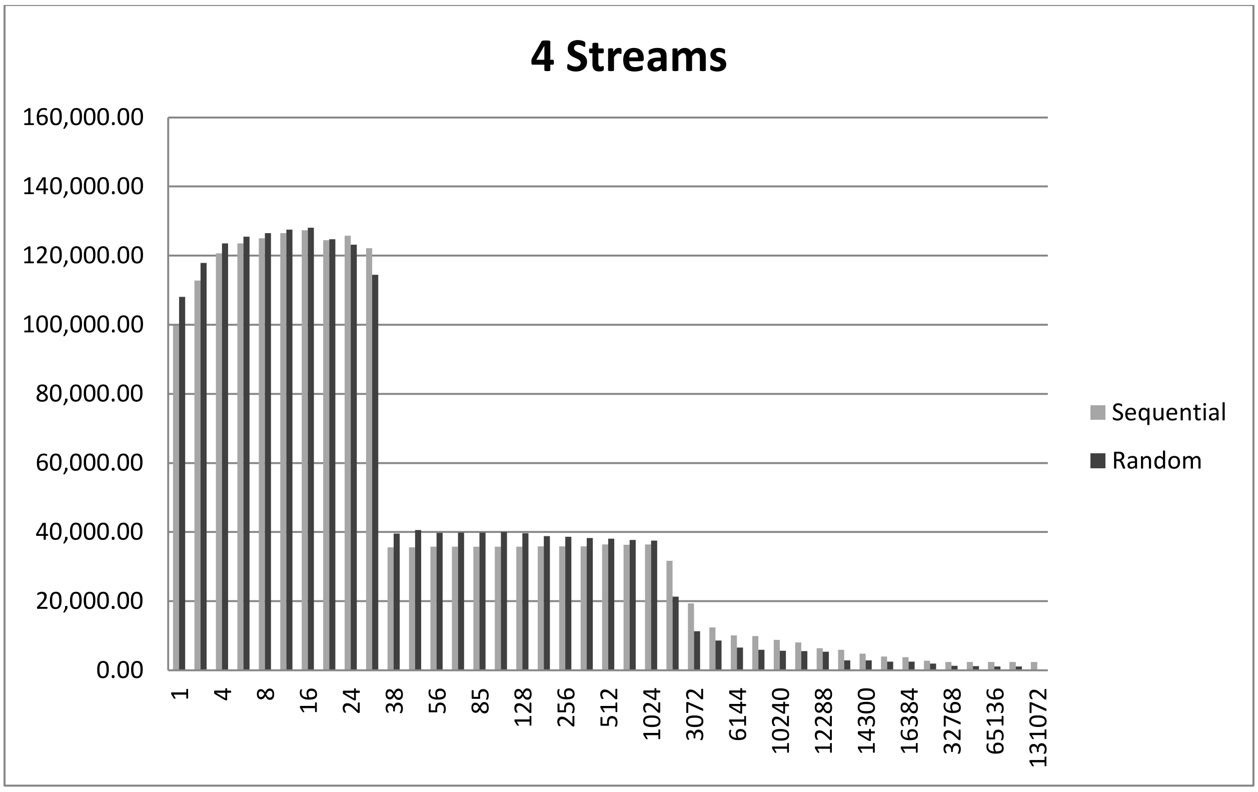


Figure A.2.4

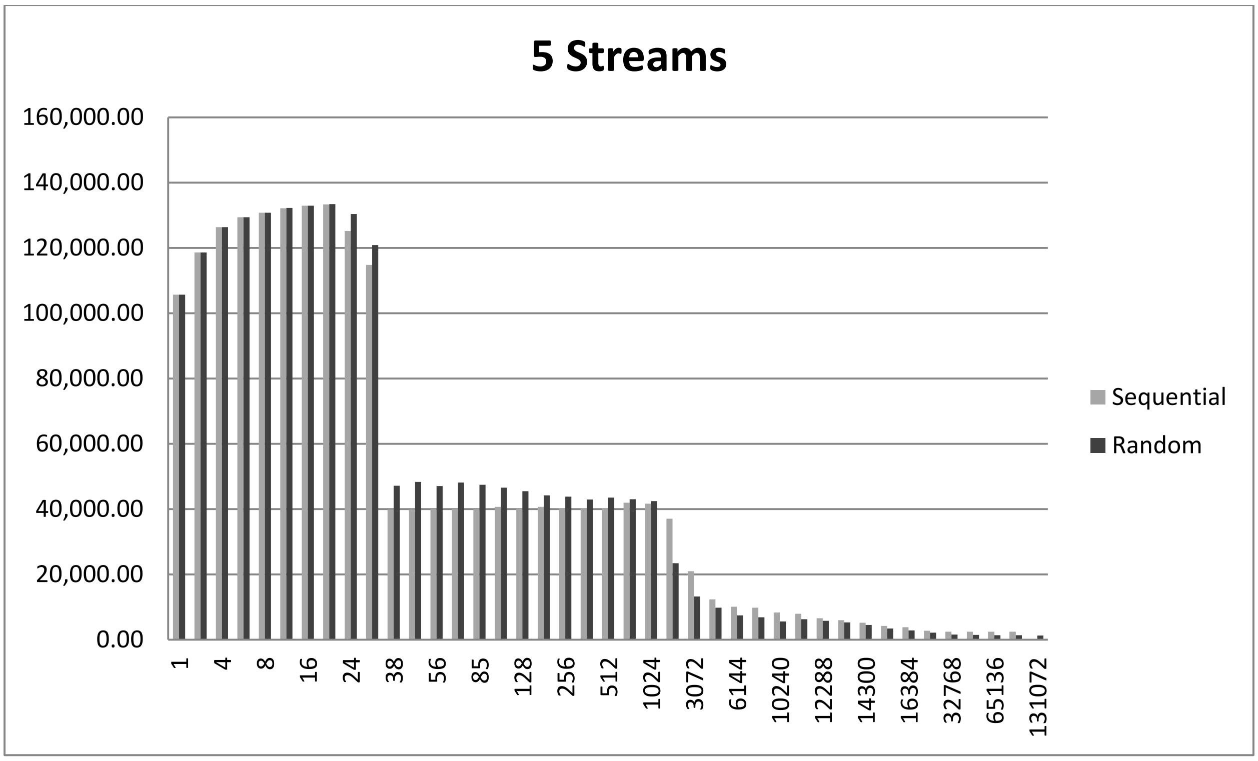


Figure A.2.5

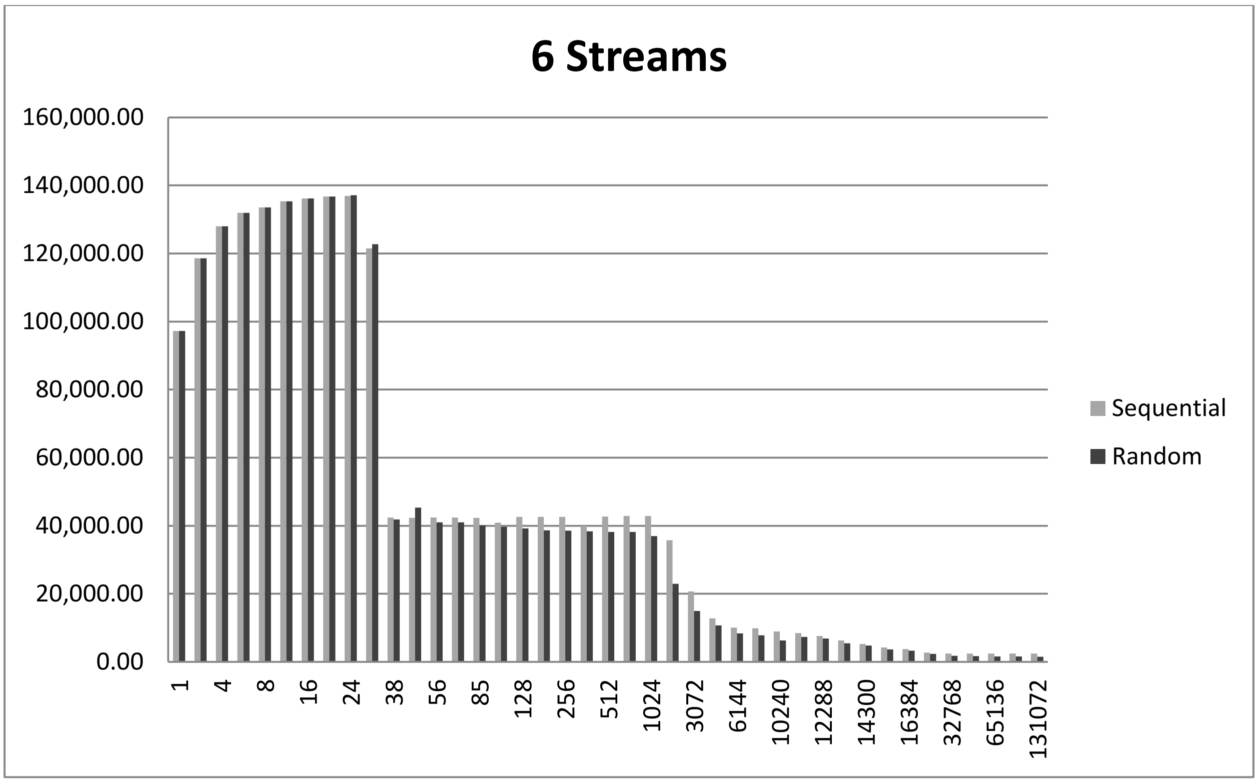


Figure A.2.6

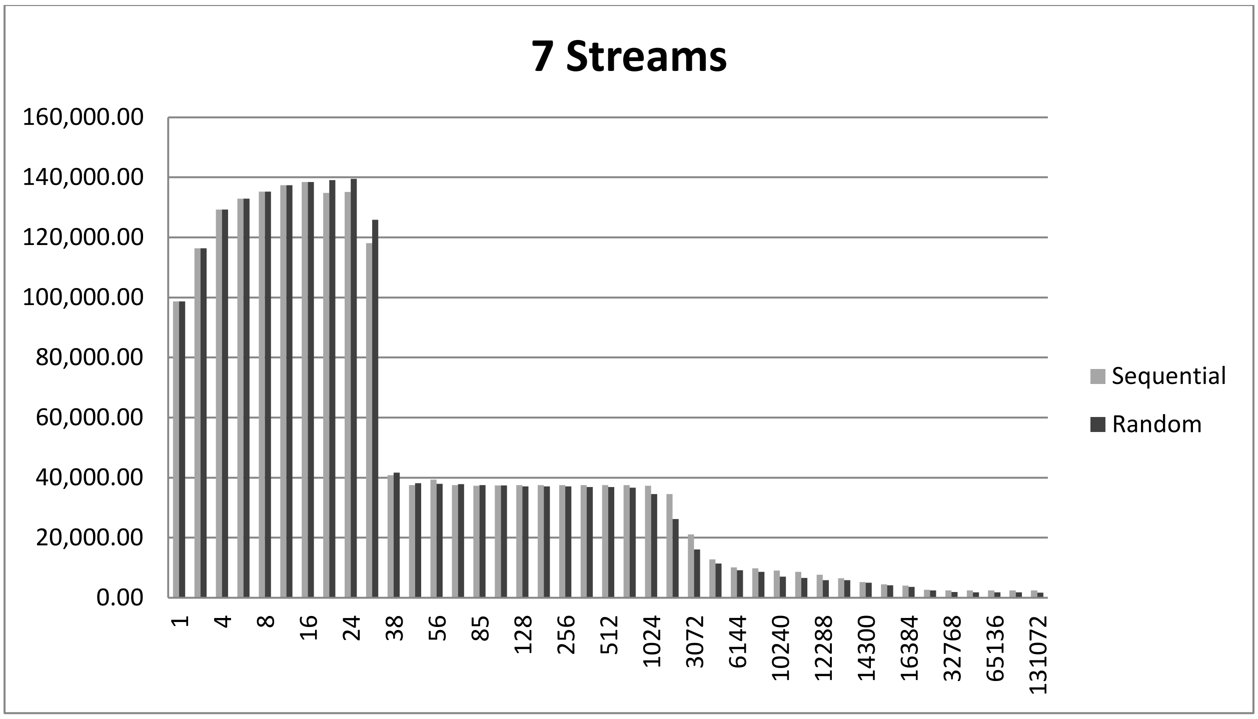


Figure A.2.7

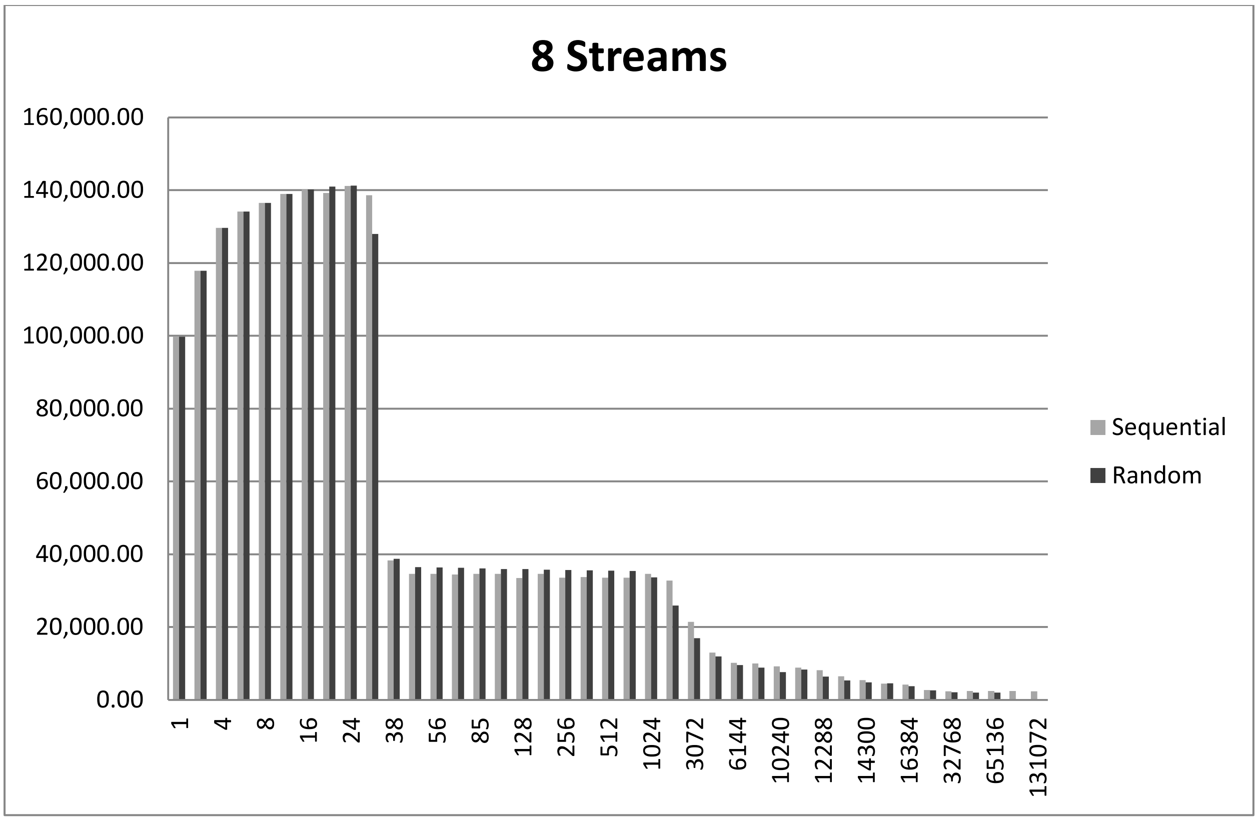


Figure A.2.8

Appendix B

Co-execution test results

The following pages contain full tables, with all slowdown results from co-execution experiments.

		L2_1Str_seq	L2_2Str_seq	L2_2Str_rdm	L2_4Str_seq	L2_4Str_rdm	L2_1Str_rdm	L3s_1str_seq	L3s_1str_rdm	L3s_2str_seq	L3s_2str_rdm	L3s_maxstr_seq	L3s_maxstr_rdm	L3m_1Str_seq	L3m_1Str_rdm	L3m_2Str_seq	L3m_2Str_rdm	L3m_4Str_seq	L3m_4Str_rdm	L3l_1Str_seq	L3l_1Str_rdm	L3l_2Str_seq	L3l_2Str_rdm	L3l_4Str_seq	L3l_8Str_rdm	MEM_1Str_rdm	MEM_2Str_rdm	MEM_8Str_rdm	MEM_1Str_seq	MEM_2Str_seq	MEM_3Str_seq
	Class	N	N	C (N)	C (N)	C (N)	C	C	C	C	C	C	C	C	C	C	C	C	C	LC	LC	LC	LC	LC	LC	LC	LC	LC	L	L	L
L2_1Str_seq	N	1.04	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.09	1.05	1.05	1.00	1.14	1.13	1.06	1.05	1.00	1.09	1.08	1.01	1.14	1.06
L2_2Str_seq	N	1.00	1.10	1.00	1.00	1.00	1.18	1.05	1.06	1.00	1.00	1.00	1.00	1.18	1.07	1.00	1.00	1.08	1.07	1.00	1.00	1.16	1.13	1.14	1.14	1.17	1.11	1.13	1.07	1.14	1.11
L2_2Str_rdm	C (N)	0.88	0.88	0.81	0.74	0.76	1.04	0.74	0.89	0.74	0.76	0.71	0.97	0.84	0.87	0.56	0.81	0.78	0.90	0.87	0.57	0.73	0.73	1.18	0.57	0.72	0.56	0.78	0.76	0.75	0.90
L2_4Str_seq	C (N)	0.92	0.92	0.89	0.85	0.76	0.94	0.90	0.95	0.82	0.92	0.74	0.79	0.98	0.87	0.80	0.94	0.86	0.88	0.95	0.85	0.96	0.81	0.91	0.94	0.87	0.75	0.98	1.02	0.85	0.98
L2_4Str_rdm	C (N)	0.64	1.01	0.65	0.95	0.98	0.64	0.95	1.09	0.64	0.95	0.65	0.65	0.90	1.07	1.13	1.01	1.06	0.86	1.69	0.86	1.06	1.01	1.01	0.97	1.02	0.87	0.64	1.04	1.01	1.11
L2_1Str_rdm	C	1.49	1.00	1.00	1.00	1.00	1.40	1.00	1.00	1.00	1.00	1.00	1.00	1.59	1.29	1.51	1.23	1.23	1.24	1.24	1.88	1.54	1.47	1.26	1.30	1.29	1.53	1.52	1.60	1.56	1.37
L3s_1str_seq	C	1.02	1.02	1.02	1.02	1.02	1.02	1.01	1.00	1.00	1.00	1.00	1.00	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.03	1.02	1.03	1.02	1.02	1.02	1.02	1.02	1.03	1.03
L3s_1str_rdm	C	1.01	1.01	1.02	1.02	1.02	1.01	1.02	1.01	1.01	1.00	1.01	1.00	1.02	1.01	1.02	1.01	1.02	1.01	1.02	1.01	1.03	1.02	2.72	1.02	1.02	1.02	1.02	1.02	1.03	8.20
L3s_2str_seq	C	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.00	1.01	1.00	1.01	1.00	1.01	1.01	1.00	1.01	1.01	1.01	1.02	1.02
L3s_2str_rdm	C	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.02	1.01	1.01	1.00	1.01	1.01	1.02	1.01	1.02	1.01	1.02	1.02	1.03	1.01	1.02	1.02	1.02	1.02	1.02	1.03	1.03	1.03
L3s_maxstr_seq	C	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.00	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.02	1.02
L3s_maxstr_rdm	C	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.03	1.01	1.02	1.01	1.02	1.02	1.01	1.01	1.02	1.02	1.03	1.03
L3m_1Str_seq	C	1.00	1.01	1.00	1.01	1.00	1.00	1.01	1.00	1.01	1.00	1.01	1.01	1.01	1.00	1.01	1.00	1.01	1.01	1.01	1.00	1.01	1.01	1.02	1.01	1.00	1.00	1.01	1.01	1.02	1.03
L3m_1Str_rdm	C	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.00	1.01	1.00	1.02	1.04	1.01	1.01	1.01	1.00	2.90	1.00	2.76	1.01	2.84	3.01	1.00	1.00	1.01	5.06	5.96	7.64
L3m_2Str_seq	C	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.01	1.01	1.00	1.01	1.01	1.01	1.00	1.01	1.02	1.02	1.02
L3m_2Str_rdm	C	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.00	1.01	1.01	1.01	1.01	1.01	1.00	1.05	1.01	2.83	1.01	2.94	1.01	1.01	1.00	1.01	1.03	5.72	6.91
L3m_4Str_seq	C	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.00	1.01	1.00	1.00	1.00	1.02	1.01	1.01	1.01	1.01	1.00	1.01	1.01	1.01	1.02	1.02	1.02
L3m_4Str_rdm	C	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.01	1.01	1.01	1.06	1.01	1.02	1.01	1.02	1.01	1.02	1.01	1.01	1.01	1.01	1.02	1.03	6.58
L3l_1Str_seq	LC	1.00	1.00	1.00	1.00	1.00	1.00	1.05	1.05	1.07	1.05	1.05	1.06	1.13	1.07	1.12	1.12	1.12	1.14	1.46	1.05	1.55	1.12	1.58	1.45	1.06	1.12	1.34	1.47	1.71	1.99
L3l_1Str_rdm	LC	1.01	1.05	1.01	1.05	1.00	1.00	1.18	1.25	1.31	1.22	1.17	1.24	1.48	1.36	1.48	1.51	1.48	1.44	2.61	2.47	2.68	2.61	2.63	2.59	2.43	2.63	2.50	2.78	3.10	3.89
L3l_2Str_seq	LC	1.00	1.00	1.00	1.00	1.00	1.00	1.03	1.04	1.03	1.04	1.04	1.05	1.08	1.04	1.07	1.08	1.09	1.08	1.32	1.02	1.60	1.04	1.70	1.25	1.02	1.04	1.18	1.32	1.66	1.86
L3l_2Str_rdm	LC	1.01	1.00	1.00	1.00	1.00	1.00	1.20	1.19	1.18	1.17	1.19	1.26	1.43	1.35	1.39	1.34	1.41	1.47	2.52	1.58	2.54	2.37	2.55	2.48	1.57	2.33	2.39	2.66	3.01	3.53
L3l_4Str_seq	LC	1.00	1.00	1.00	1.00	1.00	1.00	1.04	1.00	1.03	1.04	1.04	1.04	1.11	1.02	1.12	1.05	1.14	1.13	1.36	1.03	1.93	1.03	2.12	1.18	1.00	1.04	1.12	1.35	2.08	2.28
L3l_8Str_rdm	LC	1.00	1.00	1.00	1.00	1.00	1.01	1.02	1.02	1.01	1.01	1.01	1.00	1.05	1.02	1.06	1.04	1.04	1.05	1.53	1.00	1.61	1.03	1.66	1.45	1.02	1.05	1.32	1.58	1.94	2.23
MEM_1Str_rdm	LC	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.04	1.01	1.06	1.01	1.08	1.04	1.01	1.01	1.03	1.09	1.31	1.84
MEM_2Str_rdm	LC	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.04	1.01	1.07	1.02	1.07	1.03	1.01	1.01	1.03	1.09	1.29	1.62
MEM_8Str_rdm	LC	1.01	1.00	1.00	1.00	1.01	1.01	1.01	1.01	1.02	1.01	1.01	1.01	1.01	1.01	1.02	1.02	1.01	1.01	1.03	1.01	1.04	1.01	1.05	1.03	1.01	1.01	1.02	1.04	1.12	1.21
MEM_1Str_seq	L	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.00	1.00	1.00	1.00	1.10	1.01	1.15	1.02	1.17	1.10	1.01	1.02	1.06	1.08	1.24	1.36
MEM_2Str_seq	L	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.02	1.01	1.03	1.00	1.00	1.08	1.02	1.25	1.03	1.31	1.07	1.01	1.01	1.06	1.09	1.28	1.43
MEM_3Str_seq	L	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.00	1.00	1.00	1.00	1.01	1.02	1.01	1.05	1.01	1.06	1.14	1.03	1.37	1.05	1.44	1.13	1.01	1.03	1.11	1.19	1.41	1.54

Table B.1: Total slowdown results for Sandy Bridge (sandman)

		L2s_1Str_seq	L2s_1Str_rdm	L2s_2Str_seq	L2s_2Str_rdm	L2s_max_seq	L2s_max_rdm	L2m_1Str_seq	L2m_2Str_seq	L2m_1Str_rdm	L2m_2Str_rdm	L2m_max_seq	L2m_max_rdm	L2l_1Str_seq	L2l_1Str_rdm	L2l_2Str_seq	L2l_2Str_rdm	L2l_max_seq	L2l_max_rdm	L3s_1Str_seq	L3s_1Str_rdm	L3s_2Str_seq	L3s_2Str_rdm	L3s_max_seq	L3s_max_rdm	L3l_1Str_seq	L3l_1Str_rdm	L3l_2Str_seq	L3l_2Str_rdm	L3l_max_seq	L3l_max_rdm	MEM_1Str_rdm	MEM_2Str_rdm	MEM_1Str_seq	MEM_2Str_seq	MEM_max_seq	MEM_max_rdm
	Class	C (N)	C (N)	C (N)	C (N)	C (N)	C (N)	C (N)	C (N)	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	LC	LC	LC	LC	LC	LC	LC	LC	L	L	L	L
L2s_1Str_seq	C (N)	1.00	1.05	1.09	1.10	1.10	1.09	1.04	1.19	1.24	1.00	1.09	1.13	1.15	1.15	1.14	1.15	1.15	1.06	1.11	1.14	1.10	1.10	1.09	1.21	1.11	1.09	1.34	1.74	1.46	1.58	1.10	1.20	1.16	1.05	1.11	1.17
L2s_1Str_rdm	C (N)	1.00	1.00	1.03	1.05	1.04	1.04	1.01	1.14	1.16	1.00	1.00	1.16	1.09	1.09	1.02	1.15	1.08	1.00	1.05	1.09	1.04	1.05	1.03	1.10	1.04	1.03	1.27	1.63	1.41	1.45	1.05	1.14	1.10	1.00	1.05	1.12
L2s_2Str_seq	C (N)	1.00	1.00	1.00	1.02	1.01	1.02	1.00	1.02	1.00	1.00	1.04	1.14	1.00	1.00	1.01	1.08	1.14	1.01	1.00	1.00	1.00	1.02	1.00	1.06	1.19	1.09	1.25	1.57	1.36	1.42	1.09	1.10	1.25	1.00	1.02	1.08
L2s_2Str_rdm	C (N)	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.06	1.00	1.00	1.06	1.11	1.00	1.02	1.00	1.00	1.00	1.00	1.00	1.07	1.47	1.23	1.28	1.55	1.35	1.41	1.11	1.09	1.45	1.00	1.00	1.06
L2s_max_seq	C (N)	1.00	1.00	1.01	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.03	1.00	1.00	1.00	1.00	1.09	1.01	1.01	1.00	1.01	1.01	1.03	1.05	1.27	1.12	1.37	1.26	1.38	1.38	1.10	1.20	1.28	1.42	1.01	1.08
L2s_max_rdm	C (N)	1.03	1.03	1.04	1.02	1.00	1.01	1.03	1.02	1.04	1.02	1.11	1.08	1.00	1.00	1.03	1.02	1.13	1.01	1.19	1.04	1.01	1.03	1.07	1.09	1.48	1.10	1.56	1.30	1.40	1.44	1.10	1.25	1.41	1.63	1.04	1.11
L2m_1Str_seq	C (N)	1.02	1.04	1.06	1.08	1.07	1.07	1.02	1.13	1.28	1.00	1.16	1.12	1.09	1.06	1.12	1.16	1.13	1.08	1.11	1.14	1.10	1.08	1.13	1.17	1.18	1.21	1.45	1.92	1.64	1.72	1.09	1.19	1.15	1.05	1.11	1.17
L2m_2Str_seq	C (N)	1.08	1.08	1.01	1.00	1.00	1.00	1.05	1.04	1.05	1.01	1.07	1.06	1.07	1.04	1.05	1.12	1.20	1.00	1.03	1.07	1.10	1.03	1.01	1.11	1.64	1.30	1.37	1.78	1.55	1.57	1.25	1.11	1.66	1.00	1.03	1.09
L2m_1Str_rdm	C	1.06	1.04	1.00	1.00	1.00	1.00	1.12	1.00	1.00	1.00	1.00	1.04	1.00	1.00	1.00	1.05	1.01	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.04	1.07	1.29	1.65	1.40	1.49	1.00	1.04	1.00	1.00	1.00	1.02
L2m_2Str_rdm	C	1.00	1.00	1.03	1.01	1.10	1.07	1.00	1.12	1.08	1.08	1.15	1.16	1.00	1.10	1.16	1.13	1.17	1.11	1.10	1.08	1.15	1.12	1.09	1.18	2.13	1.29	1.49	1.87	1.66	1.70	1.22	1.21	2.10	1.07	1.13	1.19
L2m_max_seq	C	1.00	1.00	1.02	1.00	1.00	1.07	1.08	1.07	1.03	1.05	1.14	1.09	1.05	1.03	1.11	1.19	1.16	1.01	1.13	1.02	1.20	1.17	1.08	1.21	1.66	1.24	1.84	1.42	1.58	1.55	1.12	1.41	1.58	1.92	1.03	1.10
L2m_max_rdm	C	1.00	1.06	1.07	1.01	1.00	1.00	1.00	1.01	1.05	1.01	1.04	1.02	1.03	1.00	1.04	1.02	1.06	1.00	1.11	1.04	1.02	1.00	1.03	1.03	1.74	1.21	2.20	1.45	1.40	1.53	1.16	1.37	1.82	2.13	1.00	1.05
L2l_1Str_seq	C	1.03	1.03	1.00	1.00	1.00	1.00	1.01	1.07	1.01	1.00	1.04	1.07	1.06	1.01	1.07	1.14	1.10	1.06	1.15	1.10	1.14	1.10	1.15	1.23	1.13	1.25	1.34	1.93	1.47	1.71	1.01	1.11	1.06	1.00	1.02	1.08
L2l_1Str_rdm	C	1.01	1.01	1.00	1.00	1.00	1.00	1.00	1.02	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.07	1.03	1.00	1.00	1.02	1.03	1.01	1.04	1.10	1.11	1.22	1.39	1.87	1.59	1.67	1.00	1.08	1.03	1.00	1.00	1.07
L2l_2Str_seq	C	1.00	1.00	1.00	1.03	1.00	1.00	1.01	1.03	1.00	1.03	1.08	1.06	1.05	1.00	1.08	1.13	1.20	1.18	1.21	1.00	1.22	1.08	1.24	1.29	1.50	1.14	1.39	1.93	1.52	1.70	1.10	1.11	1.53	1.00	1.01	1.08
L2l_2Str_rdm	C	1.00	1.02	1.00	1.03	1.00	1.00	1.00	1.04	1.03	1.00	1.11	1.00	1.06	1.02	1.07	1.03	1.14	1.00	1.11	1.02	1.00	1.00	1.00	1.11	1.70	1.20	1.26	1.89	1.43	1.62	1.13	1.04	1.64	1.00	1.00	1.01
L2l_max_seq	C	1.00	1.00	1.08	1.00	1.04	1.05	1.01	1.16	1.03	1.02	1.11	1.07	1.07	1.03	1.19	1.18	1.31	1.15	1.18	1.11	1.32	1.11	1.39	1.46	1.37	1.17	1.54	1.28	1.51	1.71	1.09	1.15	1.49	1.61	1.02	1.08
L2l_max_rdm	C	1.00	1.00	1.00	1.02	1.01	1.00	1.01	1.00	1.01	1.02	1.01	1.01	1.07	1.03	1.22	1.07	1.20	1.10	1.12	1.04	1.18	1.07	1.23	1.26	1.55	1.17	1.76	1.29	1.53	1.75	1.07	1.21	1.52	1.69	1.03	1.11
L3s_1Str_seq	C	1.00	1.00	1.00	1.00	1.00	1.14	1.03	1.03	1.02	1.00	1.13	1.16	1.15	1.03	1.24	1.19	1.23	1.11	1.59	1.34	1.34	1.46	1.36	1.69	1.26	1.61	1.58	2.49	1.76	2.10	1.04	1.14	1.08	1.00	1.05	1.13
L3s_1Str_rdm	C	1.00	1.00	1.00	1.00	1.00	1.00	1.02	1.03	1.01	1.00	1.00	1.05	1.07	1.02	1.00	1.06	1.12	1.00	1.30	1.17	1.04	1.04	1.01	1.10	1.31	1.57	1.51	2.48	1.64	1.75	1.00	1.11	1.07	1.00	1.04	1.10
L3s_2Str_seq	C	1.00	1.00	1.00	1.00	1.01	1.00	1.03	1.11	1.01	1.05	1.21	1.07	1.16	1.08	1.26	1.07	1.39	1.19	1.36	1.08	1.89	1.51	1.67	2.02	1.22	1.04	1.58	2.58	1.77	2.12	1.03	1.17	1.24	1.03	1.07	1.16
L3s_2Str_rdm	C	1.00	1.00	1.01	1.00	1.01	1.00	1.01	1.04	1.01	1.03	1.18	1.05	1.11	1.05	1.12	1.04	1.16	1.07	1.48	1.08	1.50	1.26	1.14	1.26	1.56	1.06	1.85	2.56	2.19	2.22	1.05	1.17	1.22	1.02	1.05	1.18
L3s_max_seq	C	1.00	1.00	1.00	1.00	1.03	1.03	1.06	1.02	1.01	1.01	1.09	1.09	1.18	1.08	1.29	1.08	1.46	1.24	1.38	1.06	1.68	1.15	1.77	2.15	1.25	1.03	1.56	1.07	1.80	2.15	1.02	1.04	1.25	1.43	1.09	1.17
L3s_max_rdm	C	1.05	1.00	1.00	1.02	1.00	1.00	1.04	1.07	1.02	1.03	1.16	1.03	1.19	1.09	1.27	1.14	1.45	1.20	1.62	1.09	1.92	1.20	2.04	1.78	1.33	1.06	1.74	1.17	1.69	1.99	1.05	1.10	1.31	1.52	1.01	1.11
L3l_1Str_seq	LC	1.19	1.17	1.38	1.73	1.49	1.68	1.29	1.94	1.30	2.29	1.96	2.15	1.34	1.35	1.81	2.16	1.69	1.83	1.49	1.60	1.42	1.83	1.46	1.64	2.53	3.42	3.24	5.16	3.67	4.10	1.61	1.81	1.76	1.68	1.78	1.97
L3l_1Str_rdm	LC	1.11	1.10	1.21	1.37	1.24	1.18	1.26	1.46	1.27	1.32	1.39	1.42	1.42	1.41	1.31	1.45	1.36	1.30	1.81	1.82	1.15	1.18	1.15	1.24	3.25	2.38	1.72	2.90	1.89	2.02	1.24	1.33	1.24	1.15	1.21	1.28
L3l_2Str_seq	LC	1.13	1.12	1.14	1.18	1.25	1.39	1.24	1.27	1.26	1.25	1.70	2.13	1.25	1.32	1.31	1.26	1.48	1.62	1.46	1.45	1.45	1.70	1.43	1.68	2.54	1.42	2.91	4.66	3.28	3.65	1.40	1.58	2.55	1.47	1.57	1.74
L3l_2Str_rdm	LC	1.21	1.19	1.19	1.19	1.00	1.00	1.36	1.36	1.34	1.30	1.09	1.16	1.49	1.47	1.51	1.55	1.02	1.00	1.91	1.96	1.96	1.94	1.00	1.00	3.35	1.97	3.86	2.63	1.77	1.87	1.92	1.00	3.34	1.00	1.00	1.00
L3l_max_seq	LC	1.13	1.15	1.14	1.16	1.17	1.15	1.31	1.32	1.27	1.29	1.35	1.25	1.27	1.40	1.34	1.32	1.34	1.31	1.51	1.46	1.50	1.86	1.52	1.51	2.66	1.44	3.03	1.98	3.16	3.50	1.44	1.89	2.70	3.00	1.49	1.67
L3l_max_rdm	LC	1.20	1.16	1.17	1.17	1.15	1.16	1.34	1.32	1.32	1.30	1.30	1.34	1.44	1.44	1.46	1.46	1.49	1.46	1.76	1.51	1.75	1.85	1.78	1.74	2.90	1.51	3.31	2.04	3.42	3.11	1.47	1.98	2.88	3.26	1.34	1.48
MEM_1Str_rdm	LC	1.01	1.01	1.08	1.12	1.11	1.06	1.02	1.26	1.02	1.12	1.13	1.23	1.03	1.03	1.15	1.23	1.15	1.08	1.06	1.05	1.03	1.06	1.02	1.10	1.38	1.12	1.53	2.53	1.70	1.78	1.12	1.20	1.41	1.03	1.09	1.15
MEM_2Str_rdm	LC	1.02	1.02	1.02	1.02	1.12	1.13	1.04	1.04	1.03	1.03	1.32	1.34	1.05	1.04	1.07	1.05	1.12	1.13	1.08	1.08	1.09	1.09	1.00	1.08	1.44	1.12	1.61	1.20	2.08	2.22	1.11	1.19	1.45	1.61	1.15	1.21
MEM_1Str_seq	L	1.02	1.01	1.19	1.40	1.23	1.31	1.04	1.61	1.03	1.85	1.53	1.84	1.04	1.03	1.52	1.71	1.49	1.47	1.05	1.07	1.19	1.18	1.20	1.32	1.45	1.07	2.67	4.22	3.05	3.33	1.35	1.49	1.45	1.38	1.46	1.59
MEM_2Str_seq	L	1.04	1.04	1.04	1.04	1.54	1.72	1.07	1.08	1.06	1.07	2.11	2.44	1.06	1.07	1.08	1.06	1.84	1.84	1.10	1.12	1.11	1.11	1.55	1.74	1.56	1.12	1.75	1.28	3.85	4.28	1.12	1.88	1.56	1.75	1.85	2.06
MEM_max_seq	L	1.04	1.04	1.04	1.04	1.04	1.04	1.08	1.08	1.06	1.07	1.07	1.07	1.07	1.07	1.08	1.06	1.11	1.07	1.09	1.12	1.11	1.08	1.12	1.10	1.57	1.12	1.77	1.28	1.82	1.66	1.12	1.27	1.57	1.75	1.82	2.01
MEM_max_rdm	L	1.02	1.03	1.02	1.03	1.03	1.03	1.05	1.05	1.04	1.05	1.06	1.06	1.05	1.07	1.07	1.05	1.09	1.06	1.09	1.10	1.11	1.13	1.12	1.12	1.61	1.10	1.82	1.26	1.89	1.70	1.10	1.25	1.59	1.81	1.86	1.68

Table B.2: Total slowdown for Dunnington, non-shared L2

		L2s_1Str_seq	L2s_1Str_rdm	L2s_2Str_seq	L2s_2Str_rdm	L2s_max_seq	L2s_max_rdm	L2m_1Str_seq	L2m_2Str_seq	L2m_1Str_rdm	L2m_2Str_rdm	L2m_max_seq	L2m_max_rdm	L2l_1Str_seq	L2l_1Str_rdm	L2l_2Str_seq	L2l_2Str_rdm	L2l_max_seq	L2l_max_rdm	L3s_1Str_seq	L3s_1Str_rdm	L3s_2Str_seq	L3s_2Str_rdm	L3s_max_seq	L3s_max_rdm	L3l_1Str_seq	L3l_1Str_rdm	L3l_2Str_seq	L3l_2Str_rdm	L3l_max_seq	L3l_max_rdm	MEM_1Str_rdm	MEM_2Str_rdm	MEM_1Str_seq	MEM_2Str_seq	MEM_max_seq	MEM_max_rdm
	Class	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	LC	LC	LC	LC	LC	LC	LC	LC	L	L	L	L
L2s_1Str_seq	C	1.10	1.06	1.27	1.10	1.22	1.17	1.23	1.94	1.05	1.12	2.23	1.68	1.24	1.03	1.92	1.11	4.42	2.36	1.21	1.03	1.98	1.12	5.59	2.82	1.43	1.12	1.68	1.23	2.28	1.55	1.12	1.24	1.49	1.81	2.61	1.76
L2s_1Str_rdm	C	1.27	1.26	1.09	1.17	1.32	1.44	2.72	3.57	1.46	1.90	3.54	2.80	4.50	1.09	5.87	2.26	6.84	5.14	5.07	1.10	6.64	1.67	8.35	7.23	1.60	1.19	2.18	1.38	3.11	3.02	1.14	1.36	1.63	2.00	2.23	2.24
L2s_2Str_seq	C	1.15	1.04	1.27	1.14	1.36	1.24	1.20	1.40	1.03	1.10	2.48	1.79	1.22	1.04	1.41	1.07	4.49	1.55	1.23	1.03	1.48	1.09	4.05	1.45	1.41	1.11	1.63	1.22	2.20	1.59	1.10	1.21	1.46	1.74	2.48	1.67
L2s_2Str_rdm	C	1.18	1.09	1.48	1.22	1.47	1.31	1.67	3.50	1.06	1.35	3.16	3.02	1.26	1.07	5.65	1.14	7.63	5.96	1.25	1.03	2.14	1.13	9.13	7.06	1.40	1.15	1.64	1.30	1.91	1.72	1.12	1.25	1.43	1.71	1.94	1.68
L2s_max_seq	C	1.28	1.21	1.33	1.32	1.67	1.36	1.22	1.42	1.16	1.29	2.01	1.80	1.27	1.12	1.36	1.24	1.86	1.56	1.28	1.12	1.40	1.18	1.67	1.39	1.40	1.19	1.56	1.33	2.04	1.77	1.16	1.28	1.43	1.67	2.38	1.74
L2s_max_rdm	C	1.13	1.11	1.24	1.18	1.40	1.34	1.13	1.24	1.06	1.11	1.47	1.44	1.17	1.06	1.26	1.08	1.46	1.27	1.12	1.05	1.30	1.07	1.46	1.17	1.37	1.14	1.52	1.26	1.64	1.61	1.13	1.27	1.34	1.64	1.84	1.56
L2m_1Str_seq	C	1.37	1.31	1.47	1.36	1.61	1.59	2.17	2.32	1.32	1.49	2.64	2.33	2.24	1.28	3.39	1.52	4.92	3.76	2.17	1.26	3.59	1.51	5.54	3.85	1.80	1.23	2.48	1.42	4.48	2.91	1.21	1.35	1.75	2.45	5.09	2.76
L2m_2Str_seq	C	1.91	1.32	1.64	1.54	2.00	1.95	2.07	3.55	1.25	1.51	4.02	3.72	2.06	1.21	3.73	1.49	6.51	5.20	2.23	1.24	3.78	1.47	6.97	4.96	1.73	1.22	2.34	1.35	4.50	2.55	1.15	1.28	1.80	2.43	4.65	2.52
L2m_1Str_rdm	C	1.83	1.78	1.91	1.75	1.80	1.73	2.97	3.12	2.43	2.75	3.28	2.94	3.73	2.47	4.70	3.05	5.24	4.31	4.43	2.48	5.23	3.11	5.93	5.10	3.63	1.92	4.07	2.69	6.57	4.37	1.36	2.04	3.23	3.68	4.34	3.92
L2m_2Str_rdm	C	1.88	1.80	1.84	1.86	1.97	1.79	3.31	3.66	2.08	2.69	3.73	3.43	4.45	2.06	5.04	2.81	5.99	4.96	4.93	2.03	5.13	2.87	6.67	5.60	2.82	1.67	3.45	2.07	4.59	3.99	1.29	1.71	2.22	2.90	3.40	3.07
L2m_max_seq	C	1.91	1.50	2.32	1.70	2.16	2.30	1.78	2.71	1.31	1.57	5.50	5.72	1.75	1.29	2.92	1.67	5.96	6.85	1.63	1.26	2.76	1.51	5.58	6.15	1.59	1.18	2.03	1.44	3.59	2.19	1.20	1.38	1.57	2.34	3.83	2.26
L2m_max_rdm	C	1.50	1.23	1.73	1.42	1.66	1.68	1.67	2.47	1.12	1.35	3.42	3.97	1.62	1.13	2.54	1.39	3.37	3.71	1.59	1.06	2.51	1.32	3.22	3.47	1.40	1.19	1.63	1.27	1.94	1.72	1.15	1.25	1.42	1.73	1.85	1.59
L2l_1Str_seq	C	1.19	1.20	1.30	1.36	1.61	1.53	1.85	2.03	1.23	1.38	2.25	1.98	2.00	1.23	2.75	1.39	3.99	3.08	1.98	1.22	2.97	1.39	4.71	3.14	1.75	1.16	2.40	1.27	4.05	2.70	1.14	1.24	1.85	2.53	4.50	2.64
L2l_1Str_rdm	C	1.55	1.53	1.53	1.48	1.57	1.49	2.11	2.20	1.79	1.96	2.27	2.09	2.57	1.85	2.96	2.14	3.28	2.77	3.09	1.85	3.31	2.17	3.67	3.08	2.49	1.62	8.34	1.94	9.37	8.51	1.47	1.71	2.35	3.16	11.16	2.70
L2l_2Str_seq	C	1.62	1.33	1.47	1.64	1.63	1.63	2.16	2.70	1.27	1.69	3.07	2.87	2.18	1.29	3.01	1.68	4.64	3.85	2.18	1.26	2.96	1.65	4.72	3.62	1.74	1.16	2.27	1.35	3.79	2.62	1.16	1.33	1.79	2.49	4.38	2.63
L2l_2Str_rdm	C	1.54	1.51	1.55	1.51	1.59	1.52	2.25	2.37	1.63	1.92	2.47	2.25	2.71	1.64	3.04	1.98	3.53	2.96	2.98	1.61	3.01	1.97	3.82	3.24	2.26	1.46	2.53	1.78	2.84	2.60	1.34	1.56	2.19	2.44	2.63	2.45
L2l_max_seq	C	1.86	1.37	2.04	1.74	1.67	1.66	1.83	2.53	1.36	1.67	3.51	3.63	1.86	1.32	2.51	1.60	3.93	4.39	1.86	1.32	2.51	1.67	3.80	4.19	1.54	1.17	1.90	1.30	2.79	2.68	1.16	1.30	1.44	1.89	3.02	2.55
L2l_max_rdm	C	1.69	1.30	1.57	1.53	1.54	1.46	1.98	2.59	1.30	1.58	2.82	3.05	2.03	1.28	2.57	1.56	2.85	3.28	1.99	1.29	2.47	1.56	2.74	3.14	1.43	1.17	1.78	1.33	1.93	2.13	1.18	1.27	1.46	1.74	1.79	2.02
L3s_1Str_seq	C	1.04	1.11	1.13	1.29	1.39	1.45	1.57	1.80	1.09	1.19	1.84	1.65	1.69	1.13	2.24	1.22	3.36	2.50	2.40	1.13	2.86	1.24	4.37	2.95	1.68	1.06	2.75	1.19	10.61	8.41	1.06	1.20	1.74	2.46	12.11	6.73
L3s_1Str_rdm	C	1.13	1.13	1.15	1.13	1.16	1.12	1.35	1.37	1.24	1.28	1.39	1.28	1.55	1.29	1.74	1.32	1.92	1.60	1.94	1.44	2.12	1.57	2.37	2.09	5.26	1.27	5.89	2.36	6.07	5.70	1.18	1.56	6.21	6.56	6.93	6.48
L3s_2Str_seq	C	1.18	1.15	1.09	1.15	1.18	1.16	1.43	1.75	1.14	1.28	1.96	1.92	1.49	1.14	1.88	1.28	2.87	2.50	1.50	1.12	2.64	1.30	3.46	2.66	1.36	1.11	1.69	1.22	9.40	2.21	1.09	1.24	1.38	1.87	9.18	2.16
L3s_2Str_rdm	C	1.14	1.14	1.15	1.12	1.16	1.12	1.40	1.47	1.17	1.27	1.48	1.39	1.60	1.17	1.76	1.34	2.08	1.68	2.02	1.21	2.12	1.62	2.35	2.16	5.51	1.14	6.12	1.37	6.66	6.10	1.11	1.26	5.49	6.33	7.12	6.39
L3s_max_seq	C	1.23	1.10	1.22	1.21	1.03	1.06	1.25	1.41	1.10	1.20	1.85	1.94	1.23	1.10	1.50	1.20	2.01	2.33	1.21	1.08	1.37	1.23	2.40	2.45	1.13	1.04	1.25	1.09	1.87	1.71	1.05	1.09	1.12	1.25	2.01	1.78
L3s_max_rdm	C	1.22	1.08	1.13	1.19	1.11	1.07	1.36	1.55	1.08	1.27	1.74	1.79	1.36	1.08	1.60	1.19	1.71	1.92	1.33	1.08	1.57	1.19	1.92	2.09	1.18	1.04	1.31	1.07	1.35	1.59	1.04	1.07	1.23	1.36	1.32	1.54
L3l_1Str_seq	LC	1.20	1.26	1.21	1.37	1.36	1.55	1.37	1.47	1.25	1.33	1.71	1.57	1.61	1.35	1.89	1.40	2.46	2.04	1.94	1.53	2.43	1.72	3.12	2.58	2.85	1.55	3.69	1.83	5.10	4.09	1.54	1.82	2.77	3.68	5.84	4.01
L3l_1Str_rdm	LC	1.17	1.14	1.19	1.13	1.21	1.19	1.31	1.47	1.30	1.33	1.40	1.39	1.59	1.42	1.83	1.43	1.96	1.68	2.09	1.89	2.26	1.87	2.36	2.23	3.37	2.43	3.66	2.94	3.77	3.47	2.36	2.82	3.66	3.79	3.91	3.66
L3l_2Str_seq	LC	1.22	1.24	1.28	1.39	1.46	1.45	1.41	1.46	1.31	1.37	1.65	1.59	1.52	1.49	1.68	1.42	2.03	1.90	1.90	1.49	1.88	1.91	2.46	2.20	2.30	1.50	3.00	1.90	4.63	3.79	1.49	1.87	2.27	2.95	4.74	3.59
L3l_2Str_rdm	LC	1.20	1.17	1.19	1.17	1.24	1.18	1.45	1.43	1.36	1.41	1.59	1.44	1.72	1.52	1.84	1.53	2.01	1.82	2.15	2.01	2.20	2.05	2.30	2.19	3.55	2.02	3.73	2.73	3.99	3.65	1.97	2.68	3.62	3.73	4.07	3.69
L3l_max_seq	LC	1.08	1.11	1.19	1.20	1.18	1.26	1.11	1.12	1.23	1.17	1.31	1.38	1.22	1.21	1.24	1.25	1.51	1.76	1.69	1.29	2.04	1.52	1.76	2.10	1.64	1.25	2.16	1.47	3.24	3.68	1.17	1.44	1.62	2.14	3.14	3.27
L3l_max_rdm	LC	1.18	1.20	1.18	1.22	1.23	1.22	1.37	1.38	1.33	1.31	1.45	1.43	1.45	1.43	1.45	1.43	1.49	1.52	2.11	1.38	1.76	1.78	1.69	1.76	2.15	1.42	2.51	1.79	2.68	3.13	1.44	1.78	2.13	2.51	2.67	2.93
MEM_1Str_rdm	LC	1.02	1.02	1.03	1.02	1.03	1.02	1.06	1.07	1.04	1.05	1.08	1.07	1.12	1.05	1.18	1.08	1.22	1.15	1.18	1.07	1.23	1.10	1.26	1.22	1.31	1.10	1.38	1.19	1.46	1.35	1.11	1.20	1.35	1.39	1.48	1.37
MEM_2Str_rdm	LC	1.02	1.02	1.03	1.02	1.03	1.03	1.05	1.06	1.03	1.05	1.08	1.07	1.09	1.04	1.12	1.06	1.14	1.13	1.10	1.06	1.13	1.08	1.18	1.13	1.28	1.10	1.30	1.15	1.39	1.27	1.10	1.16	1.29	1.31	1.41	1.29
MEM_1Str_seq	L	1.05	1.13	1.07	1.23	1.19	1.34	1.14	1.22	1.08	1.18	1.35	1.33	1.30	1.07	1.56	1.17	1.88	1.64	1.35	1.06	1.77	1.13	2.22	1.85	1.63	1.06	2.18	1.13	3.03	2.40	1.06	1.14	1.63	2.18	3.48	2.34
MEM_2Str_seq	L	1.13	1.15	1.15	1.18	1.21	1.23	1.22	1.25	1.18	1.21	1.33	1.31	1.26	1.20	1.32	1.22	1.56	1.56	1.29	1.21	1.35	1.29	1.87	1.67	1.40	1.21	1.79	1.30	2.76	2.33	1.21	1.30	1.40	1.79	2.91	2.16
MEM_max_seq	L	1.03	1.04	1.03	1.07	1.04	1.11	1.01	1.03	1.01	1.05	1.07	1.16	1.02	1.05	1.04	1.04	1.22	1.46	1.15	1.05	1.27	1.13	1.28	1.64	1.15	1.05	1.31	1.13	1.84	2.15	1.05	1.13	1.15	1.31	1.85	1.93
MEM_max_rdm	L	1.00	1.00	1.00	1.00	1.01	1.00	1.00	1.00	1.00	1.00	1.01	1.02	1.00	1.00	1.00	1.00	1.04	1.07	1.10	1.02	1.02	1.05	1.05	1.07	1.13	1.02	1.23	1.06	1.35	1.50	1.02	1.05	1.13	1.24	1.34	1.44

Table B.3: Total slowdown for Dunnington, shared L2

		L2s_1Str_seq	L2s_1Str_rdm	L2s_2Str_seq	L2s_2Str_rdm	L2s_max_seq	L2s_max_rdm	L2m_1Str_seq	L2m_2Str_seq	L2m_1Str_rdm	L2m_2Str_rdm	L2m_max_seq	L2m_max_rdm	L2l_1Str_seq	L2l_1Str_rdm	L2l_2Str_seq	L2l_2Str_rdm	L2l_max_seq	L2l_max_rdm	L3s_1Str_seq	L3s_1Str_rdm	L3s_2Str_seq	L3s_2Str_rdm	L3s_max_seq	L3s_max_rdm	L3l_1Str_seq	L3l_1Str_rdm	L3l_2Str_seq	L3l_2Str_rdm	L3l_max_seq	L3l_max_rdm	MEM_1Str_rdm	MEM_2Str_rdm	MEM_1Str_seq	MEM_2Str_seq	MEM_max_seq	MEM_max_rdm
	Class	C(N)	C(N)	C(N)	C(N)	C(N)	C(N)	C(N)	C(N)	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	LC	LC	LC	LC	LC	LC	LC	LC	L	L	L	L
L2s_1Str_seq	C(N)	1.10	1.01	1.17	0.99	1.11	1.07	1.18	1.63	0.85	1.12	2.05	1.49	1.08	0.90	1.69	0.96	3.84	2.23	1.09	0.90	1.80	1.02	5.13	2.34	1.29	1.03	1.25	0.70	1.56	0.98	1.02	1.03	1.28	1.72	2.36	1.50
L2s_1Str_rdm	C(N)	1.27	1.26	1.05	1.11	1.28	1.39	2.68	3.13	1.26	1.90	3.54	2.41	4.14	1.00	5.78	1.96	6.33	5.14	4.81	1.01	6.38	1.60	8.12	6.58	1.53	1.16	1.72	0.85	2.20	2.08	1.09	1.19	1.48	2.00	2.13	2.01
L2s_2Str_seq	C(N)	1.15	1.04	1.26	1.12	1.34	1.22	1.20	1.37	1.03	1.10	2.39	1.58	1.22	1.04	1.39	1.00	3.94	1.53	1.23	1.03	1.48	1.07	4.04	1.36	1.18	1.02	1.30	0.78	1.62	1.12	1.01	1.09	1.17	1.74	2.44	1.55
L2s_2Str_rdm	C(N)	1.18	1.08	1.48	1.22	1.47	1.31	1.66	3.49	1.06	1.35	3.16	2.84	1.26	1.07	5.31	1.03	7.63	5.87	1.25	1.03	2.14	1.13	9.13	6.60	0.96	0.94	1.28	0.84	1.41	1.22	1.01	1.15	0.98	1.71	1.94	1.58
L2s_max_seq	C(N)	1.28	1.21	1.33	1.32	1.67	1.36	1.22	1.42	1.15	1.28	2.00	1.75	1.27	1.12	1.36	1.24	1.71	1.54	1.27	1.12	1.40	1.18	1.62	1.32	1.11	1.06	1.15	1.05	1.48	1.28	1.05	1.06	1.12	1.18	2.36	1.62
L2s_max_rdm	C(N)	1.10	1.08	1.19	1.16	1.40	1.32	1.09	1.21	1.02	1.10	1.32	1.33	1.17	1.06	1.22	1.05	1.29	1.26	0.94	1.01	1.29	1.04	1.37	1.07	0.92	1.04	0.97	0.97	1.17	1.12	1.03	1.01	0.95	1.01	1.77	1.41
L2m_1Str_seq	C(N)	1.35	1.26	1.38	1.26	1.50	1.48	2.12	2.05	1.04	1.49	2.28	2.08	2.05	1.21	3.02	1.31	4.34	3.47	1.95	1.11	3.27	1.40	4.90	3.28	1.52	1.02	1.71	0.74	2.73	1.69	1.11	1.13	1.52	2.32	4.57	2.35
L2m_2Str_seq	C(N)	1.77	1.22	1.63	1.54	2.00	1.95	1.97	3.43	1.18	1.49	3.76	3.51	1.92	1.16	3.54	1.33	5.40	5.20	2.16	1.16	3.44	1.42	6.93	4.46	1.06	0.94	1.70	0.76	2.91	1.62	0.92	1.16	1.09	2.43	4.50	2.32
L2m_1Str_rdm	C	1.73	1.71	1.91	1.75	1.80	1.73	2.66	3.12	2.43	2.75	3.28	2.83	3.73	2.47	4.70	2.91	5.21	4.31	4.43	2.48	5.23	3.11	5.93	5.10	3.50	1.80	3.16	1.63	4.70	2.94	1.36	1.96	3.23	3.68	4.34	3.85
L2m_2Str_rdm	C	1.88	1.80	1.79	1.85	1.80	1.67	3.31	3.28	1.92	2.50	3.24	2.96	4.45	1.88	4.35	2.49	5.14	4.45	4.48	1.87	4.46	2.56	6.09	4.73	1.32	1.29	2.31	1.10	2.76	2.34	1.06	1.41	1.06	2.71	3.02	2.57
L2m_max_seq	C	1.91	1.50	2.27	1.70	2.16	2.16	1.65	2.53	1.27	1.50	4.84	5.24	1.66	1.25	2.63	1.40	5.16	6.77	1.44	1.23	2.30	1.29	5.16	5.06	0.95	0.95	1.10	1.02	2.28	1.41	1.07	0.98	0.99	1.22	3.71	2.06
L2m_max_rdm	C	1.50	1.16	1.62	1.40	1.66	1.68	1.67	2.44	1.06	1.34	3.28	3.90	1.58	1.13	2.43	1.36	3.19	3.71	1.43	1.03	2.47	1.32	3.14	3.37	0.81	0.98	0.74	0.87	1.38	1.12	0.99	0.91	0.78	0.81	1.85	1.52
L2l_1Str_seq	C	1.16	1.17	1.30	1.36	1.61	1.53	1.84	1.90	1.22	1.38	2.16	1.86	1.89	1.21	2.58	1.22	3.63	2.91	1.72	1.11	2.60	1.27	4.08	2.56	1.56	0.93	1.79	0.66	2.75	1.58	1.13	1.12	1.74	2.53	4.43	2.45
L2l_1Str_rdm	C	1.53	1.52	1.53	1.48	1.57	1.49	2.11	2.17	1.79	1.96	2.27	2.09	2.57	1.85	2.96	2.00	3.17	2.77	3.09	1.81	3.20	2.14	3.54	2.80	2.25	1.32	6.00	1.04	5.88	5.09	1.47	1.58	2.28	3.16	11.16	2.52
L2l_2Str_seq	C	1.61	1.33	1.47	1.59	1.63	1.63	2.13	2.63	1.27	1.65	2.85	2.69	2.08	1.29	2.79	1.50	3.86	3.26	1.80	1.26	2.43	1.53	3.80	2.80	1.16	1.02	1.64	0.70	2.49	1.54	1.05	1.20	1.17	2.49	4.32	2.44
L2l_2Str_rdm	C	1.54	1.48	1.55	1.47	1.59	1.52	2.24	2.29	1.58	1.92	2.23	2.25	2.54	1.60	2.83	1.92	3.09	2.96	2.68	1.57	3.01	1.97	3.82	2.92	1.33	1.21	2.00	0.94	1.98	1.61	1.18	1.50	1.34	2.44	2.63	2.42
L2l_max_seq	C	1.85	1.37	1.88	1.74	1.60	1.58	1.80	2.18	1.32	1.64	3.16	3.41	1.75	1.29	2.11	1.35	3.00	3.81	1.57	1.19	1.90	1.51	2.73	2.87	1.12	1.00	1.23	1.02	1.85	1.57	1.06	1.13	0.97	1.17	2.95	2.35
L2l_max_rdm	C	1.69	1.30	1.57	1.50	1.53	1.46	1.96	2.59	1.28	1.55	2.79	3.01	1.91	1.25	2.11	1.46	2.37	2.99	1.77	1.25	2.09	1.46	2.22	2.49	0.92	1.01	1.02	1.03	1.26	1.22	1.11	1.05	0.96	1.03	1.74	1.83
L3s_1Str_seq	C	1.04	1.11	1.13	1.29	1.39	1.27	1.53	1.75	1.07	1.19	1.63	1.43	1.47	1.11	1.81	1.02	2.74	2.24	1.51	0.84	2.13	0.85	3.21	1.75	1.33	0.66	1.75	0.48	6.05	4.01	1.02	1.05	1.62	2.46	11.56	5.96
L3s_1Str_rdm	C	1.13	1.13	1.15	1.13	1.16	1.12	1.32	1.33	1.23	1.28	1.39	1.22	1.45	1.27	1.74	1.24	1.72	1.60	1.49	1.23	2.05	1.51	2.34	1.90	4.03	0.81	3.89	0.95	3.69	3.26	1.18	1.41	5.82	6.56	6.67	5.88
L3s_2Str_seq	C	1.17	1.14	1.09	1.15	1.17	1.16	1.38	1.58	1.12	1.22	1.63	1.79	1.29	1.06	1.49	1.20	2.07	2.10	1.11	1.03	1.40	0.86	2.07	1.31	1.12	1.07	1.07	0.47	5.30	1.04	1.06	1.05	1.11	1.82	8.56	1.86
L3s_2Str_rdm	C	1.14	1.13	1.14	1.12	1.15	1.12	1.38	1.41	1.15	1.23	1.25	1.32	1.44	1.12	1.58	1.28	1.80	1.56	1.37	1.12	1.41	1.29	2.06	1.71	3.54	1.08	3.31	0.53	3.04	2.74	1.05	1.08	4.49	6.20	6.79	5.42
L3s_max_seq	C	1.23	1.10	1.22	1.21	1.00	1.03	1.18	1.39	1.08	1.20	1.70	1.79	1.05	1.02	1.17	1.10	1.38	1.88	0.87	1.02	0.82	1.08	1.35	1.14	0.90	1.01	0.80	1.01	1.04	0.79	1.03	1.05	0.89	0.87	1.84	1.52
L3s_max_rdm	C	1.16	1.08	1.12	1.16	1.11	1.07	1.30	1.45	1.06	1.24	1.50	1.73	1.14	1.00	1.26	1.04	1.18	1.60	0.82	0.99	0.82	1.00	0.94	1.17	0.88	0.99	0.75	0.92	0.80	0.80	1.00	0.98	0.94	0.89	1.30	1.38
L3l_1Str_seq	LC	1.01	1.07	0.88	0.79	0.92	0.92	1.06	0.76	0.97	0.58	0.87	0.73	1.21	1.00	1.04	0.65	1.46	1.11	1.30	0.96	1.71	0.94	2.13	1.57	1.13	0.45	1.14	0.35	1.39	1.00	0.96	1.00	1.57	2.19	3.28	2.03
L3l_1Str_rdm	LC	1.05	1.04	0.98	0.83	0.97	1.01	1.04	1.00	1.02	1.01	1.01	0.98	1.12	1.00	1.40	0.99	1.44	1.29	1.16	1.03	1.96	1.59	2.05	1.80	1.04	1.02	2.12	1.02	1.99	1.71	1.90	2.11	2.95	3.29	3.24	2.86
L3l_2Str_seq	LC	1.08	1.11	1.12	1.18	1.16	1.04	1.13	1.15	1.04	1.09	0.97	0.75	1.22	1.13	1.28	1.13	1.37	1.17	1.30	1.03	1.30	1.12	1.72	1.31	0.90	1.06	1.03	0.41	1.41	1.04	1.07	1.18	0.89	2.00	3.02	2.06
L3l_2Str_rdm	LC	0.99	0.99	1.00	0.99	1.24	1.18	1.07	1.05	1.01	1.08	1.47	1.24	1.15	1.03	1.22	0.98	1.98	1.82	1.13	1.02	1.12	1.06	2.30	2.19	1.06	1.02	0.97	1.04	2.26	1.95	1.03	2.68	1.09	3.73	4.07	3.69
L3l_max_seq	LC	0.95	0.96	1.04	1.04	1.01	1.09	0.85	0.85	0.97	0.90	0.97	1.10	0.96	0.86	0.93	0.94	1.13	1.34	1.12	0.88	1.36	0.82	1.16	1.40	0.61	0.87	0.71	0.74	1.02	1.05	0.81	0.76	0.60	0.71	2.10	1.95
L3l_max_rdm	LC	0.99	1.03	1.01	1.04	1.07	1.06	1.02	1.05	1.00	1.01	1.11	1.07	1.00	1.00	0.99	0.98	1.00	1.04	1.20	0.91	1.01	0.96	0.95	1.01	0.74	0.94	0.76	0.88	0.78	1.01	0.97	0.90	0.74	0.77	2.00	1.98
MEM_1Str_rdm	LC	1.01	1.01	0.95	0.91	0.94	0.96	1.04	0.84	1.01	0.94	0.95	0.87	1.09	1.02	1.03	0.87	1.06	1.07	1.12	1.02	1.19	1.04	1.24	1.10	0.95	0.99	0.90	0.47	0.86	0.76	0.99	1.00	0.96	1.34	1.36	1.19
MEM_2Str_rdm	LC	1.00	1.00	1.01	1.00	0.92	0.91	1.01	1.02	1.00	1.02	0.81	0.79	1.03	0.99	1.05	1.01	1.02	1.00	1.03	0.98	1.04	0.99	1.18	1.05	0.89	0.99	0.81	0.96	0.67	0.57	0.99	0.97	0.89	0.82	1.23	1.07
MEM_1Str_seq	L	1.03	1.11	0.90	0.87	0.97	1.02	1.10	0.76	1.05	0.64	0.88	0.72	1.25	1.04	1.03	0.68	1.26	1.11	1.29	0.99	1.49	0.96	1.86	1.40	1.12	0.99	0.82	0.27	0.99	0.72	0.79	0.76	1.12	1.58	2.37	1.47
MEM_2Str_seq	L	1.08	1.11	1.10	1.14	0.78	0.72	1.13	1.16	1.11	1.14	0.63	0.54	1.18	1.12	1.22	1.16	0.85	0.85	1.17	1.07	1.21	1.16	1.20	0.96	0.90	1.07	1.02	1.02	0.72	0.54	1.08	0.69	0.90	1.02	1.57	1.05
MEM_max_seq	L	0.98	1.00	0.99	1.03	1.00	1.07	0.94	0.96	0.95	0.99	1.00	1.08	0.95	0.98	0.96	0.99	1.10	1.36	1.05	0.94	1.15	1.04	1.14	1.49	0.73	0.94	0.74	0.88	1.02	1.29	0.94	0.89	0.73	0.75	1.01	0.96
MEM_max_rdm	L	0.98	0.97	0.98	0.97	0.98	0.97	0.95	0.95	0.96	0.95	0.96	0.96	0.95	0.94	0.93	0.95	0.95	1.01	1.00	0.92	0.92	0.93	0.94	0.95	0.70	0.93	0.68	0.84	0.71	0.88	0.93	0.85	0.71	0.68	0.72	0.86

Table B.4: Total relative slowdown for Dunnington, ratio of tables B.3/B.2