Journal of Signal Processing Systems (2022) 94:721–738
https://doi.org/10.1007/s11265-021-01730-7

The Impact of Cache and Dynamic Memory Management in Static Dataflow Applications

Alemeh Ghasemi[1] · Marcelo Ruaro[1] · Rodrigo Cataldo[1] · Jean-Philippe Diguet[2] · Kevin J. M. Martin[1]

Received: 18 April 2021 / Revised: 28 September 2021 / Accepted: 30 November 2021 / Published online: 24 February 2022

Abstract
Dataflow is a parallel and generic model of computation that is agnostic of the underlying multi/many-core architecture executing it. State-of-the-art frameworks allow fast development of dataflow applications providing memory, communicating, and computing optimizations by design time exploration. However, the frameworks usually do not consider cache memory behavior when generating code. A generally accepted idea is that bigger and multi-level caches improve the performance of applications. This work evaluates such a hypothesis in a broad experiment campaign adopting different multi-core configurations related to the number of cores and cache parameters (size, sharing, controllers). The results show that bigger is not always better, and the foreseen future of more cores and bigger caches do not guarantee software-free better performance for dataflow applications. Additionally, this work investigates the adoption of two memory management strategies for dataflow applications: Copy-on-Write (CoW) and Non-Temporal Memory transfers (NTM). Experimental results addressing state-of-the-art applications show that NTM and CoW can contribute to reduce the execution time to -5.3% and −15.8%, respectively. CoW, specifically, shows improvements up to -21.8% in energy consumption with -16.8% of average among 22 different cache configurations.

Keywords Multi-core · Dataflow · Cache memory · Compilers

1 Introduction

The multi/many-core architecture is a widespread on-chip design, providing high computing power in a small silicon area. The computation power is achieved by supporting task-level parallelism, splitting the application into parallel tasks running in different cores. A generally accepted expectation is that increasing the number of cores naturally leads to better application performance. However, increasing the number of cores impacts other aspects, especially the memories subsystem. Since memories are costly in terms of area and power to be embedded on the chip, the memory hierarchy (cache memories) generally has a reduced on-chip size, making it suffer from the high pressure in systems with a high number of cores. This phenomenon is known as memory wall [1].

From a software aspect, several efforts have been made to allow the efficient development of parallel applications regarding memory footprint, communication overhead, and computing parallelism. Existing for 40+ years, the dataflow programming model may eventually stand as the ideal approach to bridge the gap between application and architecture resources. Figure 1a presents an overview of the principles of a dataflow-based application. The application is represented by a graph, where each node represents an actor having a single computing function (as exemplified by actor B1 code), and each edges representing the FIFO as a data dependency between two actors. Actors communicate via producing/consuming data tokens. An actor can start the

✉ Marcelo Ruaro
marcelo.ruaro@univ-ubs.fr

Alemeh Ghasemi
alemeh.ghasemi@univ-ubs.fr

Rodrigo Cataldo
cadorecataldo@gmail.com

Jean-Philippe Diguet
jean-philippe.diguet@cnrs.fr

Kevin J. M. Martin
kevin.martin@univ-ubs.fr

[1] Univ. Bretagne-Sud, UMR CNRS 6285, Lab-STICC, Lorient, France

[2] IRL 2010, CROSSING, Adelaide, Australia

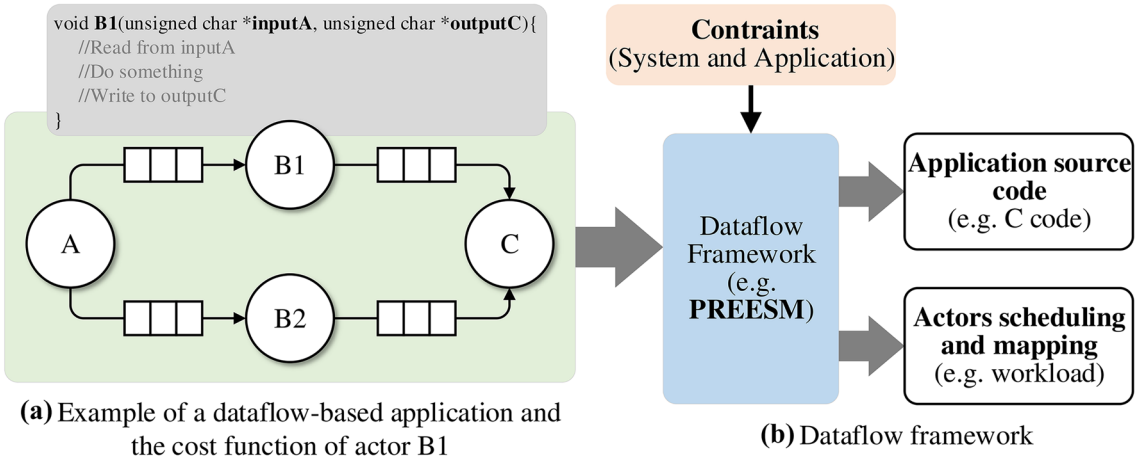


Figure 1 (**a**) Overview of a dataflow application model with four actors. (**b**) Workflow of PREESM framework [2].

execution only if required data tokens are available in the input FIFOs and if enough space is available in the output FIFOs.

Dataflow models can naturally make use of parallel resources by means of actors that run in parallel while consuming and producing tokens. Several tokens can be produced and consumed at a time, but a token is produced and consumed only once. This feature favors data spatial locality. While the cache hierarchy also exploits temporal locality, a dataflow program may benefit from the latter for instructions and spatial locality for data as consecutive tokens are usually involved. Therefore, dataflow applications performance should be improved with the increasing size of caches. However, this paper shows that such an assumption does not hold in regard to multiple cache-based architecture designs.

Taking advantage of the generic principles of dataflow applications, some rapid prototyping frameworks have been proposed. Figure 1b addresses PREESM [2], a state-of-the-art open-source framework for rapid prototyping of dataflow applications in multi/many-cores. It provides a graphical user interface for the designer to generate the application source code. Based on inputs provided by the designer including algorithm (graph of the application) and system constraints (mapping, scheduling and etc.), the framework generates a deadlock-free source code of the application (implemented in C language) and the respective actor mapping on each core, based on spatial and temporal requirements. Due to the well-defined modeling of dataflow applications, it is possible to reach design-time optimal scheduling for static applications.

Although the state-of-the-art techniques can lead to theoretical optimal schedules, this article demonstrates that even optimally scheduled applications do not scale as desired with the increasing number of cores, cache levels, size, and cache sharing factor. As expected, the memory contention is of utmost importance, and the CPU load-based actor mapping used in the experiments does not lead to the best execution time. Therefore, the first contribution of this work is to study dataflow applications according to different caches configurations, providing experimental results that demonstrate their impact on the application's execution time performance and cache miss. For this, we consider several configurations, including non-available yet platforms or non-realistic cache configurations, and use the Sniper simulator [3] to foresee the scalability of the considered dataflow applications.

From such analysis, the second contribution of this paper consists in the investigation of using two dynamic memory management techniques for dataflow code generation: Copy-on-Write (CoW) and Non-Temporal Memory (NTM) copying. Those techniques are not new since CoW is supported by Linux OS [11] and NTM is supported by some processor designs, including Intel [12, 13]. The novelty here is the study of the benefits and drawbacks of both approaches when applied to the dataflow programming model, evaluating whether they can contribute to speedup application's execution, reduce cache misses, and save energy. Additionally, those techniques can be used as runtime memory optimization approaches, complementary to static techniques [14]. Moreover, they are applied at the framework level and do not require changes for the application specification and code.

In summary, this work has two *contributions*:

- The evaluation of the impact of different cache parameters and number of cores over the performance of static dataflow applications;

- The evaluation of two existing memory management techniques (CoW and NTM) for three static dataflow applications.

The next section addresses related works that investigate the behaviors of dataflow applications running on systems with caches. Next, in Section 3 the multi-core model assumed in this work is presented. Section 4 presents the experiments varying cache parameters and the number of core. Section 5 details CoW and NTM techniques, and Section 6 presents the achieved results from those techniques. Finally, Section 7 concludes this work.

2 Related Work

This section highlights studies that target the behavior of dataflow applications running on systems with a memory hierarchy. In Domagala et al. [7], researchers extended the concept of tiling to the dataflow model to increase the data locality of applications for better performance by splitting iterations of nested loops. However, this type of optimization does not address the coarse-grain inter-actor (i.e., inter-tasks) relation.

In Maghazeh et al. [8], a method is proposed for GPU-based applications by splitting both the GPU kernel into sub-kernels and input data into tiles in size of GPU L2 cache. Their work is intended to accelerate applications whose performance is bound to memory latency. The method increases data locality, as the sub-kernels are scheduled in a way to have the least cache miss rate, for GPU applications over various settings. However, the method requires source code modification and does not target the dataflow model. Research about the cache effect on the performance of multiple application types is presented in Garcia et al. [6]. Garcia et al. have evaluated the impact of Last Level Cache (LLC) sharing in GPU-CPU co-design platform for heterogeneous applications. According to their study, applications with low data interaction between GPU and CPU are sped up slightly by sharing the LLC. Data sharing of LLC minimizes memory access time and dynamic power, and accelerates synchronization for fine-grained synchronization applications.

The cache behavior of multimedia workloads is evaluated by Slingerland and Smith [4]. They appraised data miss rate of applications considering data cache size, associativity, and line size parameters. The authors observed that multimedia applications benefit from longer data cache lines and have more data than instruction miss rate in comparison to other workloads. The experiment results reveal that most of the multimedia applications just need 32 KB data cache size to have less than 1% cache miss rate, while other types of applications (3D graphic, document processing) do not reach the same behavior. As the results of our work will show, sharing cache levels among more cores with larger sizes, up to 256 MB for LLC, does not help the performances of dataflow applications, but also results in data access latency overhead.

The work of Alvez et al. [5] investigates the impact of L2 sharing in order to find the best cache organization at this level. Assuming the NAS Parallel Benchmark, with heterogeneous workload set, and a 32-core SMP with two levels of caches (private L1-I and L1-D) and an L2, the work changes the sharing, size, associativity, and line size in the L2. Among the mains results, it was observed an execution time decrease when more cores share the L2 cache, even when 2 cores share the same L2. Increasing line size (64 bytes to 128 bytes) contributed to -32% in cache misses and +1.95% in speedup. The work does not address 3-level caches either dataflow applications.

Stoutchinin et al. [9] present a novel framework, called StreamDrive, for dynamic dataflow applications. StreamDrive proposes a new communication protocol, reserve-push-pop-release, for dataflow model instead of the standard send-receive. This protocol allows their solution to employ a zero-copy communication channel for actors. It employs a blocking mechanism to access FIFOs directly in shared memory; hence, no local copies are needed, which are commonly used in software dataflow model. This study is specific since it focuses on computer vision applications running on a special embedded multi-core platform (P2012) with dedicated hardware computer vision engines. Meanwhile, we propose two solutions to general-purpose architectures that do not require novel hardware components.

Fraguela et al. [10] propose the concept of a software cache with an autotuning method to configure its size according to each application. The approach is built upon Unified Parallel C++ (UPC++) library. It consists of an algorithm called in periodic intervals, which dynamically re-allocates the software cache size. Results show that the software cache can reduce the communication among actors due to the efficient cache sizing and allocation, presenting a hit rate just 0.27% lower than an optimal scenario. Similar to CoW and NTM, that proposal also implements the algorithm as part of a library, however, with a limited evaluation without varying hardware parameters and adopting just one application.

Table 1 summarizes the main characteristics of the related works addressing the cache impact in parallel applications running in SMPs. The main novelties of this work regarding the related works are twofold: (*i*) we evaluate a wide range of cache configuration in a multi-core architecture, including realistic and non-realistic configurations; and (*ii*) two existing memory management methods are proposed to be used in dataflow application, which can reduce the memory copies penalties in numbers and latency, leading to an improved application execution time and energy consumption.

Table 1 Related works studying the cache impact in parallel applications.

Author (et al.)	Proposal	Contribution	Benchmark
Slingerland [4]	N.A. (Evaluation work)	Cache profile of multimedia applications	Multimedia applications
Alvez [5]	N.A. (Evaluation work)	Evaluation of L2 properties	Heterogeneous applications
Garcia [6]	N.A. (Evaluation work)	Evaluation of impact of LLC sharing	Heterogeneous applications
Domagala [7]	Splitting nested loops	Increased Data locality	StreamIt
Maghazeh [8]	Splitting GPU kernels to sub-kernel and data input into L2 size	Increased Data locality + Decreased cache miss rate	GPU-based applications
Stoutchinin and Benini [9]	Novel framework (StreamDrive)	New communication protocol (zero-copy communication channel)	Dynamic Dataflow applications
Fraguela [10]	Strategy to improve cache usage in dataflow	Minimize communication among threads	Cholesky decomposition
This work	*Use of two dynamic memory manag. methods (CoW, NTM)*	*Cache configuration evaluation + Reduction in memory copy penalties*	*Static Dataflow applications*

Regarding contribution (*i*), works of [4–6] are also evaluation works. However, in [4] the benchmark is limited to multimedia applications, in [6] the focus is the iteration between the CPU and GPU by addressing a heterogeneous set of applications but not considering dataflow, and in [5] the Authors did not consider a 3-level cache either the dataflow application profile. Therefore, to the best of the Author's knowledge, the present research is the first to perform a comprehensive evaluation of the cache impact with 3-level and targeting dataflow applications.

Regarding contribution (*ii*), it fills different gaps from related works focused on proposals [7–10]. Specifically, we are interested in: (*i*) keeping the original dataflow modeling granularity (differently from [7, 8]); (*ii*) not making modification in the Linux-based kernel, or any part of the OS (contrary to [10]); and (*iii*), targeting generic SMP (differently from [9]). We endorse that the techniques of CoW and NTM are not new, and the goal of this study is to replace them in memcpys procedures in order to observe the impact in the cache and in the overall performance of dataflow applications, a study that is lacking in the literature.

3 Multi-Core Model

This section presents the multi-core architecture model adopted in this work.

3.1 Architecture Overview

Figure 2 presents the architecture overview. We focus on detailing the memory hierarchy since it is the target of this work. The architecture is based on the Intel Xeon X5500 chip. Each core implements the Nehalem Intel microarchitecture [15], having a private L1 cache with 32KB, a private L2 cache with 256KB, and a shared by four cores L3 cache with 8MB. The chip also includes a triple channel DRAM memory controller to interface with off-chip DRAM memories.

The interconnection is bus-based with 20-bits width, and provides 12.8 GB/s per link in each direction (25.6 GB/s total).

The architecture depicted in Fig. 2 is the reference multi-core model. The actual goal is to exploit different core counts and cache configurations by changing the following parameters:

- **C**: the number of cores (e.g., 4, 8, 16, 32)
- **L2 (xC)**: sharing of L2 cache, where *C* represents the number of cores sharing one L2 cache. For instance, in the baseline architecture, the L2 is (x1), since each core has one L2 cache. An L2(x2) indicates that two cores are sharing the L2. The size of L2 for each core is fixed in 256KB, therefore, in L2(x2), two cores are sharing an L2 with 512KB.
- **L3 (xC)**: sharing of L3 cache, where *C* represents the number of cores sharing one L3 cache. It adopts the same rule used in L2. For instance, the baseline architecture (assuming that there are 4 cores in total), adopts an L3 (x4) configuration.
- **L2 size**: the size of the L2 cache dedicated for each core. When a core shares the L2 cache with another core, i.e, L2 (x2 or more), the final size of the L2 cache will be multiplied by the number of shared cores.
- **L3 size**: same rule than L2 size.

3.2 Model Description

This work adopts the Sniper multi-core simulator [3]. Sniper includes the description of the Nehalem cores as well as cache, memory controller, and DRAM.

Sniper is a consolidated system simulator for multi-core architectures, used in many state-of-the-art works to evaluate application's performance and, mainly, power

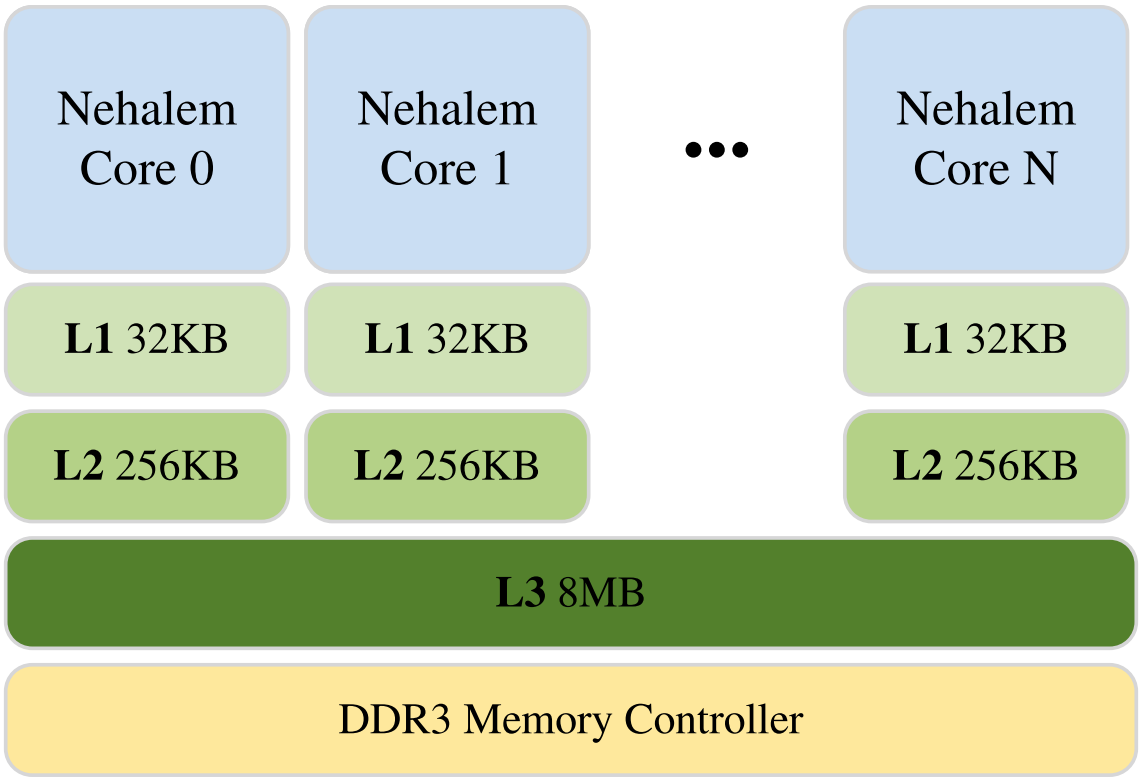


Figure 2 Architecture overview of the baseline multi-core model.

and energy consumption [16, 17]. Sniper adopts an interval-based core model simulation, which allows fast and accurate simulation. The Nehalem cores are by default provided within Sniper distribution. Sniper core model and cache hierarchy are validated against actual Xeon processor using Splash2 benchmarks. Sniper takes as input configuration files that allow the user to set parameters as cache sizes, cache sharing, number of cores, core frequency, among many others.

Next, in the experimental setup subsection, we present further details about the multi-core setup simulated on Sniper.

4 Experiments on Cache Configurations

This section addresses the first contribution of this work: experiments evaluating the cache limits for dataflow applications. The first subsection describes the experimental setup. The remaining subsections address the analyses of application's performance varying the following parameters: *C*, L2(xC), L3(xC), L2 size, and L3 size.

4.1 Experimental Setup

4.1.1 Application Set

Table 2b (1st column) lists the applications benchmark addressed in this work. We adopt three real applications named Stabilization, Stereo, and scale-invariant feature transform (SIFT), taken from PREESM repository [18]. Stabilization is used for video stabilization. Its principle is to compensate for the movements of a video recorded with a shaky camera. The main two steps of this process consist of tracking the movement of the image using image processing techniques and creating a new video where the tracked motion is compensated. The input video adopted in experiments comes from PREESM's github repository [18] and has 40.9 MB of size with a resolution of 360x202 pixels.

Stereo is a computer stereo vision application that extracts 3D information from images. Stereo matching algorithms are used in many computer vision applications to process a pair of images, taken by two separated cameras at a small distance, and produce a disparity map that corresponds to the 3rd dimension (the depth) of the captured scene. Stereo

Table 2 Experimental setup settings.

(a) Hardware model settings

Core Model	centering Intel Xeon X5550 4/8/16/32 @ 2.66 GHz (base clock)				
L1-I Cache	32KB	8way	1 cyc. tag lat.	4 cyc. data lat.	LRU
L1-D Cache	32KB	8way	1 cyc. tag lat.	4 cyc. data lat.	LRU
L2 Cache	256KB	8way	3 cyc. tag lat.	8 cyc. data lat.	LRU
L3 Cache (LLC)	8MB	16way	10 cyc. tag lat.	30 cyc. data lat.	LRU

(b) Dataflow applications benchmark profile

Application	Actors	PREESM # FIFOs	PREESM FIFOs size	Memory copying[a]	
				PREESM	Actors
Stabilization	30	607	0.92 MB	21 MB	0.2 MB
Stereo	36	811	29.09 MB	5 MB	13 MB
SIFT	77	2183	188.6 MB	12 MB	308.6 MB

cyc = cycles

lat = latency

LRU = Least Recently Used

[a] sum of all copied memory using the `memcpy` procedure

matching algorithms and their implementations are still heavily studied as they raise important research problems [19]. The two input images [18] adopted in experiments have the size of 506.3 KB with a resolution of 405x375 pixels.

SIFT is used to object recognition in cluttered real-world 3D scenes [20]. The extracted features are invariant to image scaling, translation, and rotation, and partially invariant to illumination changes and affine or 3D projection. The application behavior shares a number of properties in common with the responses of neurons in the inferior temporal cortex in primate vision. The input image [18] used in SIFT has a size of 512 KB with a resolution of 800x640 pixels, with 4 levels of parallelism and 1400 number of keypoints.

These three applications are specified through the PREESM framework, which is responsible for the code generation, actors scheduling and mapping, as shown in Fig. 1b.

Table 2b highlights that the applications have heterogeneous memory requirements. Specifically, the 4th column details the sum of PREESM FIFOs size, which can be understood as the memory footprint of inter-actor communication. SIFT is memory bounded and has high synchronization demands (high number of actors and FIFOs), Stereo is computational and memory bounded, and Stabilization is computational bounded but with low memory and synchronization demands. The heterogeneous memory requirements lead to different cache locality and memory footprints, making such applications appropriated candidates for the evaluation of cache impact intended in this work.

We use the optimal scheduling and mapping decision provided by PREESM [2], which is focused on workload balancing. The memory allocation adopts advanced memory optimization proposed in [14], which considerably reduces the applications' memory overhead. The selected memory allocation uses the *FirstFit* algorithm with *MixedMerged* distributions and *none* data alignment. These features were selected because they have presented the lowest memory footprint at the same time that they are suitable to the target multi-core architecture used in this work. After the generation of C code by PREESM, the applications were compiled using GCC v7.5.0 optimization -O2 (default optimization adopted by PREESM), and simulated on Sniper.

4.1.2 Hardware Setup

The experimental setup adopts the multi-core model described in Section 3, configured on Sniper. Table 2a presents the hardware setup. These parameters are based on the real Xeon X5500 multi-core.

To evaluate the number of cores and cache sharing we created 22 multi-core cache configurations, varying the parameters C, L2(xC), and L3(xC). Figure 3 express graphically the reasoning behinds these configurations. Each configuration is a black spot in the figure. The configurations can be divided into 4 groups (different background color on the figure) according to the number of cores ($C = 4, 8, 16, 32$) in which a given configuration was simulated. Note that the 22 configurations were not simulated for each C configuration. The minimal C evaluated for each configuration is dictated according to the sharing factor of the LLC. For instance, we do not evaluate a system with 4 cores for config. 9 (which have L3(x8) as LLC), since it is unfeasible because the L3 sharing (LLC sharing) requires at least 8 cores to meet the sharing factors of L3(x8).

The L2 sharing comprises configurations from L2(x1) up to L2(x32), with most of them (36%) addressing a private L2 cache (since this L2 design choice is found in real architectures like Xeon Nehalem and AMD K10). Some configurations are unrealistic, specially those that have a big L2, as the case of configurations 8, 16, 21, where L2 = 2MB; configurations 12, 22, where L2 = 4MB; and configuration 17, where L2 = 8MB.

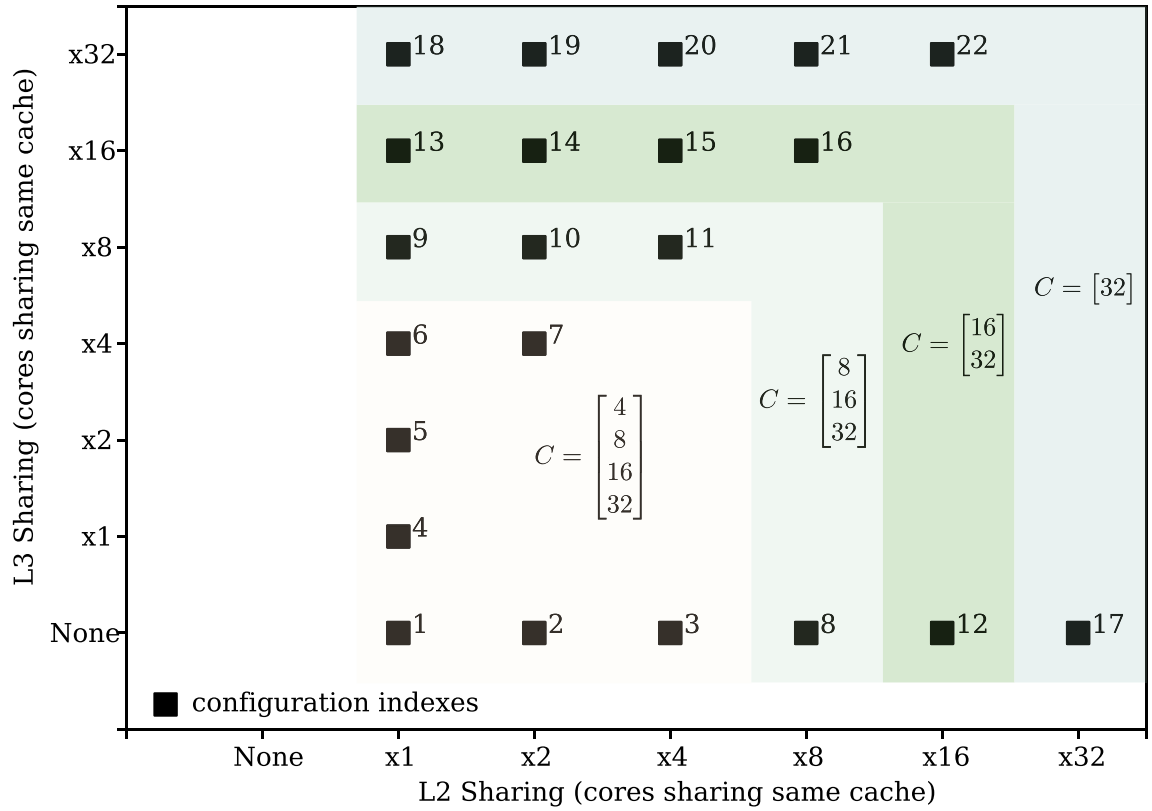


Figure 3 Overview of the reasoning behind the 22 cache configuration adopted in the experiments. **C** = number of cores simulated for each configuration.

However, our goal is to address the trend in multi-core processor design, which features always bigger L2 caches.

The L3 sharing also adopts a very heterogeneous configuration set, including no L3 (e.g. configuration 1), one private L3 cache (e.g. configuration 4), up to 32 cores sharing the same L3 (configuration 18-22).

The number of memory controllers is equal to the number of LLC. For instance, configuration 6 executed for 8 cores has two L3 shared by 4 cores (L3(x4)). Therefore, this configuration has two memory controllers (one for each L3).

Although the results achieved are based on Xeon architecture, the presence of 22 different hardware configurations, varying the core count and cache sharing and size, helps to project the behavior of the benchmarks in architectures different from Xeon, especially those that adopt similar cache organizations.

4.2 Number of Cores – *C*

Figure 4 shows the application iteration time (time for the application to complete the execution of one loop), for Stabilization (a), Stereo (b), and SIFT (c). The x-axis contains groups of bars, where each group represents one configuration (only the ones that support *C* varying from 4 to 32 were shown), and each bar represents a different *C* to that configuration.

The main evaluation to be extracted from these results is related to scalability with the number of cores *C*. It is possible to observe that Stabilization presents a continuous reduction in the execution time according to a higher *C*, reducing its execution time on average -46% from 4 to 8 cores, -43% from 8 to 16 cores, and -39% from 16 to 32 cores. However, the same does not occurs to Stereo and SIFT, which have a moderate or even worst improvement in $C \geq 16$, with Stereo presenting an execution time of -22%, -1.3%, +2.6%, for an increase in *C* of 4 to 8, 8 to 16, and 16 to 32, respectively.

Observing Table 2b, it is possible to note that Stereo and SIFT have a higher FIFOs size compared to Stabilization, which puts more pressure on the cache subsystem and does not allow the application to entirely benefit from a higher core count (reaching a memory wall).

It is also possible to observe that there are different performances among the configurations of the x-axis. Such performance is impacted due to the different L2

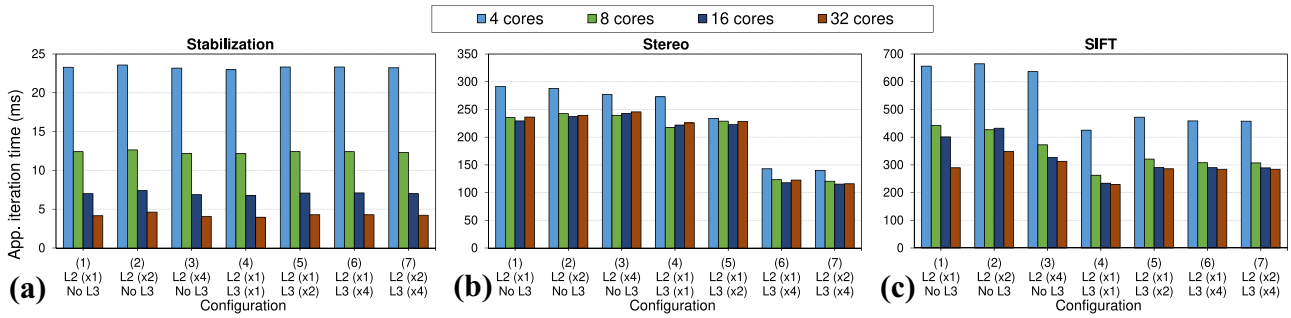


Figure 4 Application iteration time over different number of cores for three applications: (**a**) Stabilization, (**b**) Stereo, (**c**) SIFT.

and L3 sharing configurations. The next two subsections enter into details about the impact of L2 and L3 sharing.

4.3 L2 Sharing

Figure 5 presents a comprehensive evaluation of the L2 sharing impact over the execution time, L2 miss rate, and L2 miss rate for the three applications. The left y-axis of each plot represents the application iteration time, the right y-axis represents the miss rate, and the x-axis represents the configurations.

Each application has 4 plots, one for each simulated *C*. As the purpose is to evaluate the results only varying L2 sharing, the plots have the L3 sharing fixed according to the maximum number of cores (as well as in the Xeon architecture).

The L2 miss rate decreases for all applications, more sharply for Stabilization (-59%), and less significantly for SIFT (-23%), and Stereo (-22%), considering the average

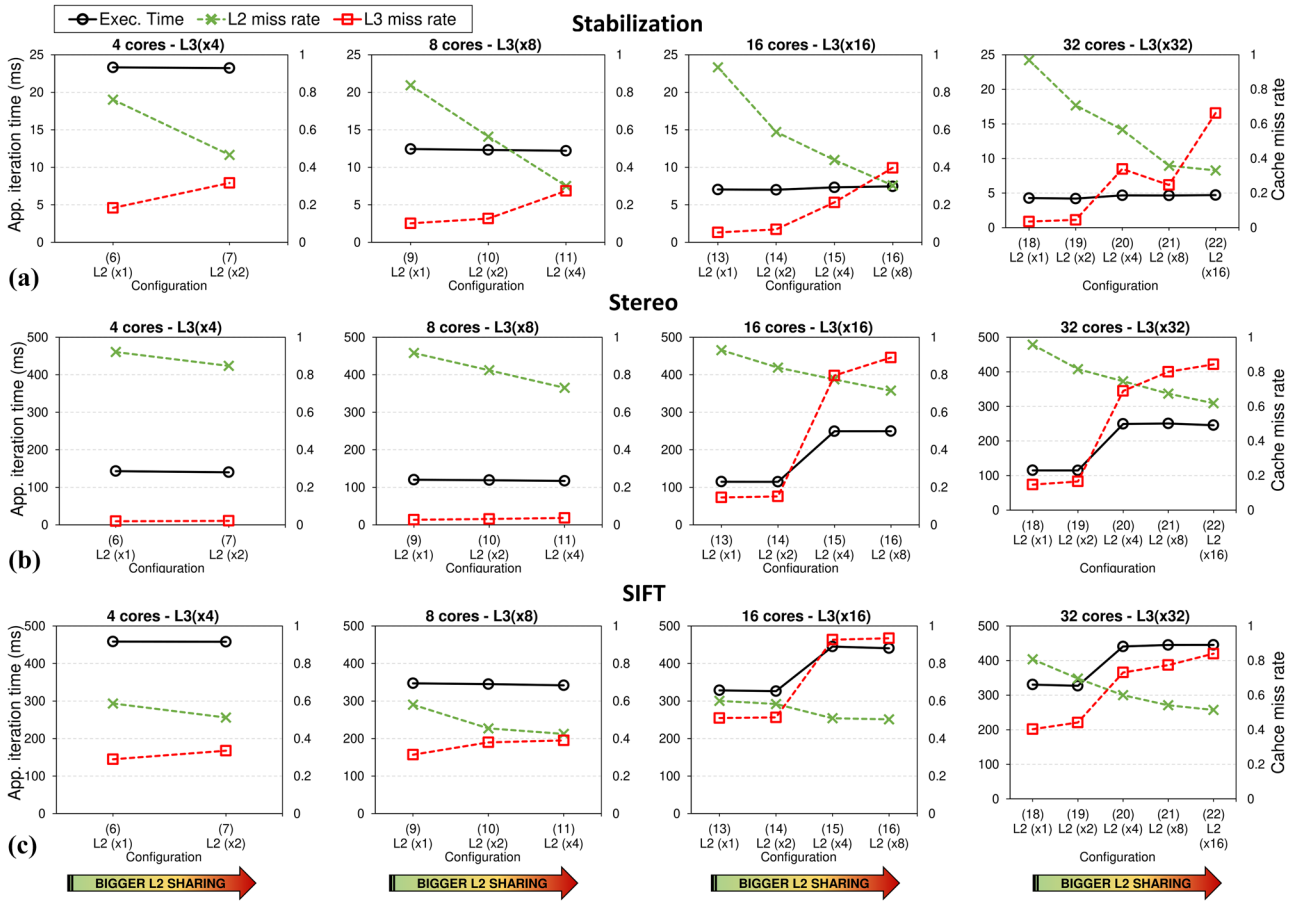


Figure 5 L2 sharing evaluation for three applications. (**a**) Stabilization, (**b**) Stereo, (**c**) SIFT.

between the leftmost configuration and the rightmost configuration. This decrease in L2 miss rate happens because a high L2 sharing increases the probability of an actor to share a FIFO inside the same L2 that is being shared with another actor (without the need to retrieve the data at the L3 cache level). The decrease is less significant in high memory demand applications – as SIFT and Stereo – since they naturally require more memory than Stabilization.

The L3 miss rate increases for all applications according to the higher L2 sharing. Such increase makes the L3 reach high miss rates of 84.3% for SIFT, 84% for Stereo, and 66.32% for Stabilization in configuration 22. Again, the memory demands of each application play an important role to stress the cache. The number of L3 accesses helps to justify this L3 miss rate increase. With a more shared L2, the L3 accesses consequently decreases, reaching, on average of -39.7% for Stabilization, -32.3% for Stereo, and -17.6% for SIFT. This makes the L3 lose temporal and spatial locality and increasing its miss rate, which transfers the data access to DRAM level and delays the execution time.

The execution time remains constant for Stabilization regardless of higher L2 sharing. For Stereo and SIFT, it remains constant for $C = 4, 8$, but for $C \geq 16$, the execution time starts to increase from L2(x2), reaching up to +56% of increase for Stereo and to +17% for SIFT L2(x32). This increase in execution time is attributable to the significant increase of the L3 miss rate compared to a not-so-high decrease of the L2 miss rate, which generates miss penalties from both sides (L2 and L3 caches).

In summary, increasing L2 cache sharing is not beneficial to dataflow applications, specifically those that demand more memory as in the case of Stereo and SIFT. This is in compliance with the cache design choices of some processor architectures as Intel Nehalem and AMD K10, which use private L2 caches. As can be observed from the results, assigning to each core a private L2 reduces the execution time since this allows a more balanced rate of L2 and L3 misses, which reduces cache contention earlier avoiding data to be fetched in a higher level of caches or even DRAM.

4.4 L3 Sharing

Figure 6 presents a similar set of plots of L2 sharing analysis, but now varying L3 sharing. The L2 sharing is fixed in L2(x1) since the previous subsection has shown that this is the best L2 sharing configuration.

The results show three trends: (*i*) L2 miss rate remains constant; (*ii*) L3 miss rate decreases significantly according to the increasing of L3 sharing; and (*iii*) the execution time can benefit from a higher L3 sharing.

Regarding the L2 miss rate, it is expected that it remains constant since the L2 was not changed. Regarding the L3 miss rate, it decreases significantly for all applications according to higher L3 sharing, reaching a miss rate in the L3(xC) of, on average, 9.3% for Stabilization (-87.34%), 8.4% for Stereo (-87%), and 37.8% for SIFT (-38%). This result is expected since a higher L3 sharing allows all application data to fit on the L3 cache (note that SIFT presented the lowest improvement due to its higher memory demands). Consequently, the execution time also benefits from this L3 miss rate decrease, specifically for the applications with higher memory demands such as Stereo and SIFT.

In summary, increasing L3 cache sharing is beneficial to dataflow applications, specifically those that demand more memory. A single L3 cache is slower but larger, allowing it to store all application data on it.

4.5 Cache Size

In the previous L2 and L3 sharing analysis, it was possible to conclude that an private L2 and an L3 shared by all cores presents the best results related to application speedup and L2/L3 miss rate. To the cache size evaluation, we keep this sharing configuration, and changed only the size of L2 or L3 per core, creating 15 new cache configurations (3 varying L2 size × 5 varying L3 sizes). Besides, the evaluation only addresses configurations with 32 cores, since lower core count have presented the same trend and are not interesting in terms of a state-of-the-art analysis.

Figure 7 shows the results varying the L2 size (256KB, 512KB, and 1MB) at x-axis. The left y-axis represents the application iteration time, and the right y-axis represents the cache miss rate. Each plot represents one application, with each one having 3 sets of results representing different L3 sizes.

It is possible to observe that the increase in L2 and L3 size has a low influence on the L2 and L3 miss rate for all applications. The execution time has a small reduction according to higher L2 sizes, however, this value is insignificant, representing an average reduction from the lower L2 size (256KB) to the higher L2 size (1MB), of -0,49% for Stereo, -1.76% for SIFT, and -4.62% for Stabilization.

The results varying the L3 sizes follows the same trend observed for L2. Figure 8 shows an example with the L2 size fixed in 512KB (other L2 sizes present very similar behavior). It is possible to see that both L2 and L3 cache misses remains stable, and with an insignificant reduction in the execution time (not better than -0.26% for all applications).

In summary, increasing the L2 and L3 sizes does not guarantee an automatic improvement for dataflow applications. In such a case, when a higher amount of hardware resources cannot provide speedup to the application, other aspects must be taken into consideration, specifically at the software level, by allowing the mapping and scheduling algorithms to make better use of such availability of resources and improving the parallel workload of the application.

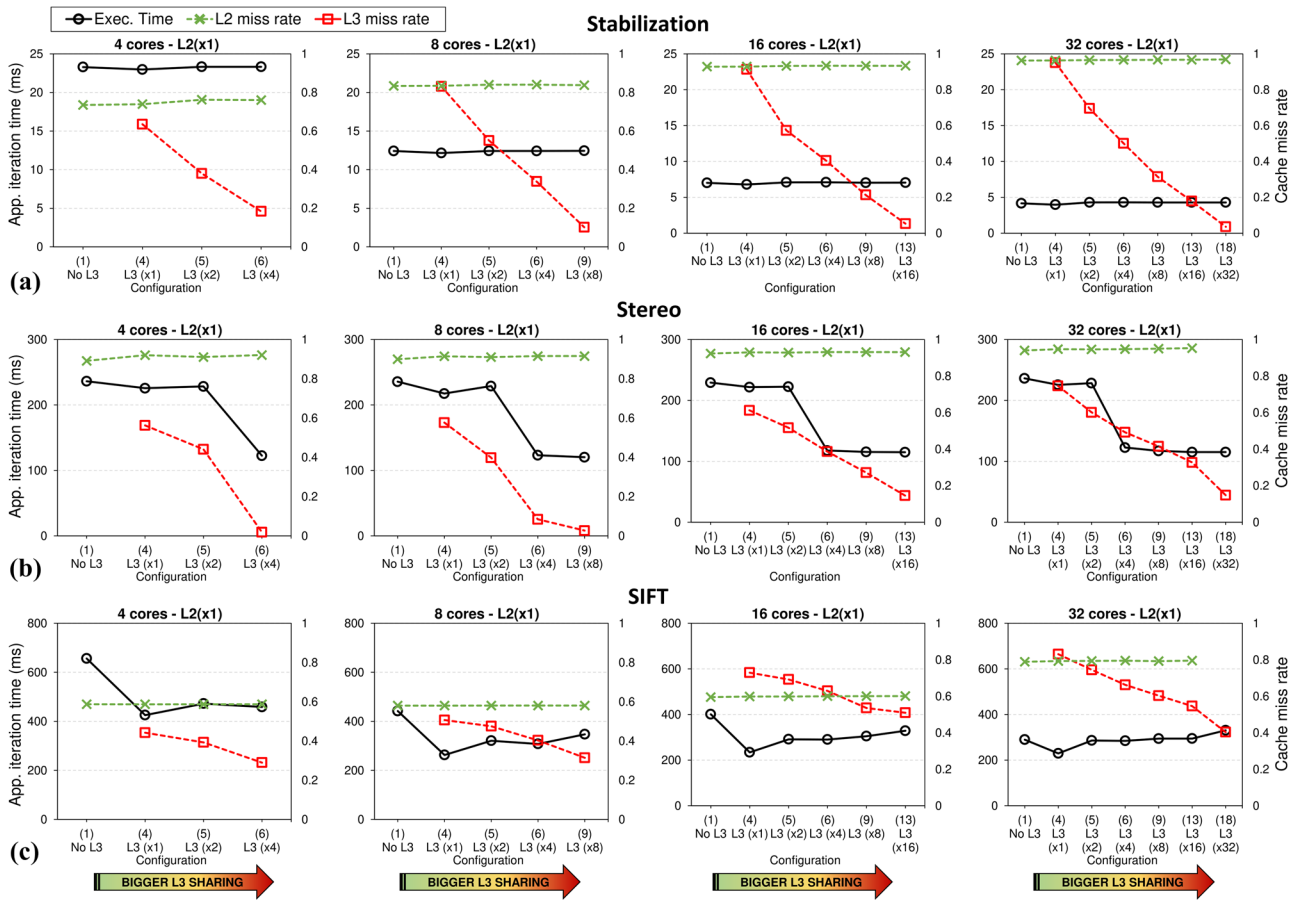


Figure 6 L3 sharing evaluation for three applications. (**a**) Stabilization, (**b**) Stereo, (**c**) SIFT.

4.6 Summary of Findings

Bigger is not always better with dataflow; increasing the number of cores, cache levels, size, does not guarantee a faster application execution. This finding is especially significant for working sets that demand more than the total cache size.

The next items summarize the finding for each analysis:

- **Number of cores**: increasing the number of cores does not guarantee automatic improvement in the execution time, since the overhead of cache protocols and required synchronization does not allow applications to increasingly speed-up, specifically the ones with more memory demands.
- **Cache sharing**: reducing L2 sharing and increasing L3 sharing was the most beneficial configuration for the addressed dataflow applications.

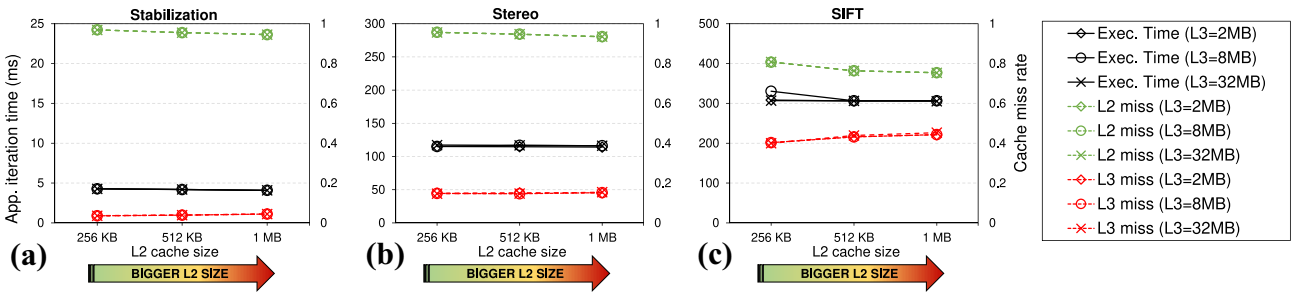


Figure 7 L2 cache size comparison varying L2 size over multiples L3 sizes. (**a**) Stabilization, (**b**) Stereo, (**c**) SIFT.

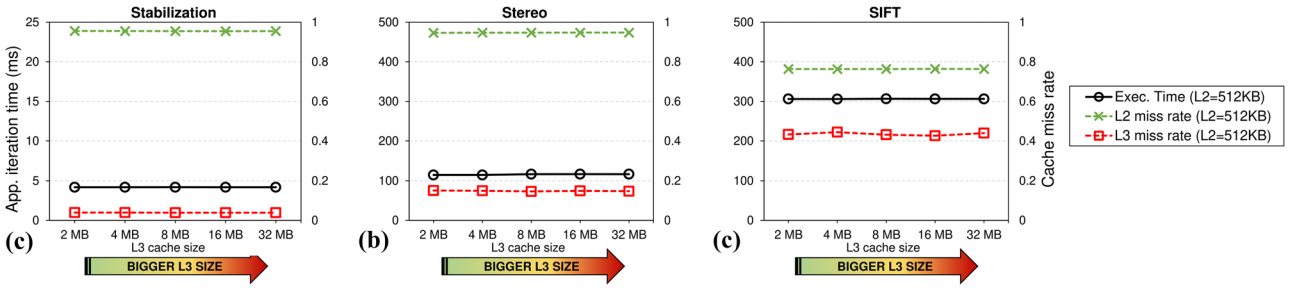


Figure 8 L3 cache size comparison varying L3 size with an private L2 of 512KB. (**a**) Stabilization, (**b**) Stereo, (**c**) SIFT.

- **Cache size**: increasing the L2 and L3 sizes have an insignificant effect on the adopted dataflow benchmarks.

One interesting finding is that private L2 and L3 shared by all cores was the configuration that presented the best results related to application speedup and L2/L3 miss rate. While this conclusion can sound similar as Intel had reached some years ago, justifying its current cache organization with L3 shared by all cores, it was not so apparent from our point of view. First, our focus was to evaluate the impact specifically for dataflow applications, research that, to the best of our knowledge, was not addressed yet. Secondly, our initial hypothesis was that when two actors – sharing the same FIFO – are mapped on different processors that share an L2 cache (increased sharing factor), this will improve performance due to the reduction in the coherence traffic and the L2 miss rate reduction. This behavior is supported by the results (Fig. 5). However, this leads to a higher miss rate for L3, which has higher penalties than L2, and consequently, has a higher influence on the execution time, as shown in the case of the three applications studied (Fig. 6).

Table 2b shows that PREESM uses memory copying mechanisms extensively for FIFO handling. Some memory copying is expected in a dataflow design; however, memory copying is done to the degree that negates the cache hierarchy benefits. Therefore, alternative approaches must be investigated to allow reducing memory copies penalties at runtime. The next sections detail the research made in this sense.

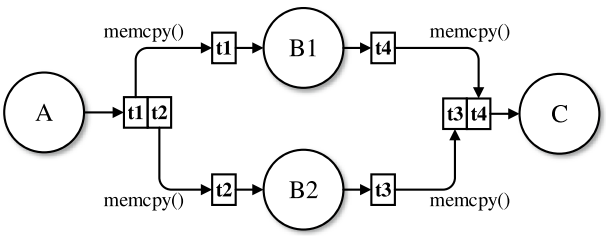
5 Dynamic Memory Management Techniques

This section presents the second contribution of this work: the evaluation of two dynamic memory management techniques and its impact when used in the context of static dataflow applications. These techniques are Copy-on-Write (CoW) and Non-Temporal Memory (NTM) copying. They are not novel in their principle, CoW is a well-known approach supported by Linux OS by the mmap() syscall [11], and NTM is essentially a direct RAM-to-RAM copy, supported in some Intel processors [12]. The novelty here is to exploit opportunities of using such techniques in dataflow frameworks, and quantify how much they can improve the applications execution time and system energy by saving memory transfers.

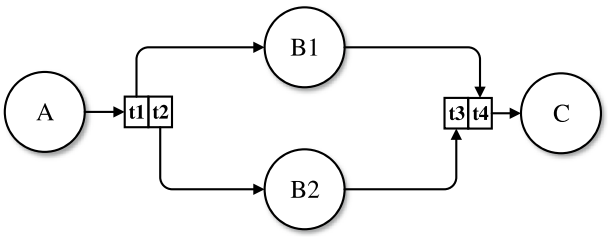
First, Subsection 5.1 presents the motivation to employ these techniques in dataflow applications. In sequence, the Subsection 5.2 presents CoW, and Subsection 5.3 presents NTM.

5.1 Motivation

Figure 9 shows a fork-join actor graph implemented as a dataflow application. Actor *A* produces data tokens *t1* and *t2* to actors *B1* and *B2*, which access those data, process, and generate a token to actor *C*, which merges the data and generates the application output.



(a) Classical dataflow fork-join application



(b) Memory optimization proposed by [14]

Figure 9 Communication overview among actors of a dataflow-based fork-join application.

A memcpy is used to transfer a data token between actors. A non-overlapping memory space (dedicated buffer), is required for each actor, since producer actors (for instance, actor *A*), can, during its processing, modify the data produced but not consumed yet in its output buffer. And the other way round, a consumer actor can use the input buffer for temporary values. Besides this fork and join pattern, dataflow applications can also have broadcast and round-buffer actors which also assumes non-overlapping memory spaces between actors [14].

Taking advantage of this high waste of memory and time by applying memory copies, Desnos et al. [14] proposed memory reuse techniques over those dataflow applications. In summary, the designer can inform the framework which buffers can be merged in the same memory space, resulting in the graph presented by Fig. 9b. Thus, the memcpy are avoided, helping to significantly save memory footprint and execution time.

However, such design-time approach only works assuming two conditions: (*i*) the designer must know the framework and the application very well in order to extract applications behavior and to model into the framework the desired memory reuse; (*ii*) it only works for buffers that are known to be read-only over all the actor lifetime, as the case of buffer *t1* of actor *B1* of Fig. 9a. If the actor – due to a branch in its algorithm flow – chooses to write in *t1* buffer space, these memory reuses cannot be adopted. Even if the actor has a probability of less than 1% to write in this buffer, the memory reuse cannot be applied since it is fundamentally a design-time exploration technique.

Thinking about how to fulfill this lack, our idea is to investigate two dynamic memory management techniques which are CoW and NTM. Differently from static memory management, they were designed to be used at runtime, and have the potential to avoid unnecessary memory copies (CoW), and cache trashing (NTM). In the next subsections, we present the details of each one and how they were implemented in our multi-core model, as well as, the evaluation of its drawbacks and benefits.

5.2 Copy-On-Write (CoW)

Figure 10 details the CoW concept. The principle of CoW is simple. It consists in allowing two or more threads (actors in our case) to share to the same memory space. When one thread attempts to write in that space a new memory space is dynamically created, bringing the data with it (a copy on write). It thus prevents the writing thread from overwriting the data in the first memory space [11]. Figure 10 depicts at time $t1$ thread *A* and thread *B* pointing to the same memory space 1. At the time $t2$, thread *B* writes in the memory space 1. At the time $t3$, the OS detects this write and makes a copy of the memory space, creating the memory space 2 and making thread *B* point to it. Now, any data written/read by thread *B* will be placed/accessed in memory space 2.

This functionality can be implemented in a dataflow application by making the buffers involved in a given operation (like a fork, join, and broadcast) point to the same memory space after the producer actor writes data in this space. If the destination buffer receives a write attempt by any actor, a CoW happens, preserving the original values of the buffer to the consumer actor.

The core code change to support CoW is shown on the right side of Table 3 as a single line. Initialization and termination has been omitted for this example. The CoW mechanism is achieved by mapping all destination buffers (named `dst_buffer`) into the same physical address, which is referenced by file descriptor `shm_open_fd`. This latter address was initialized by the `shm_open` procedure. Besides, we map the region as private (`MAP_PRIVATE`) with read and write permissions (`PROT_READ | PROT_WRITE`). The combination of these two flags will create a new copy of the physical address when a write has been made to the memory area (in other words, a copy-on-write). Finally, we allow the OS to decide the virtual address of this new buffer by passing nil (`NULL`) as the first parameter to `mmap`.

The CoW procedure is typically handled by the OS kernel. Unfortunately, the Sniper simulator has limited operating system modeling capabilities to evaluate kernel-based strategies [3]. Thus, for our experiments, we used a combination of user- and kernel-space interaction so that Sniper can account OS overhead accurately. Specifically, in our implementation a buffer is mapped to CoW without the `PROT_WRITE` flag. Any future attempts to write in the buffer will trigger an exception (`SIGSEGV` signal[1]), which interrupts the causing thread. Then, an exception handler implemented at user space changes the offending memory page to use CoW. In regard to the code presented in Table 3, we change the capability of the memory regions to read-only (removing flag `PROT_WRITE`), and install a signal handler to re-enable the write capability for this mapping.

5.3 Non-Temporal Memory (NTM) Copying

NTM copying is ideal for memory spaces known to be write-mostly or rarely used (i.e., poor temporal locality). This approach uses either (*i*) instructions that bypass the cache hierarchy or (*ii*) userspace RAM-to-RAM DMA. For the x86 architecture, (*i*) is available using SSE extensions [12], and (*ii*) through the I/OAT DMA engine available in some processor designs [13]. In any case, the `memcpy` procedure is replaced by another procedure that uses either technique specialized for memory transfer and therefore, avoiding CPU to be executing instruction of data transfers. Another

[1] `SIGSEGV` is a synchronously-generated signal and is guaranteed to be delivered to the causing POSIX thread [22].

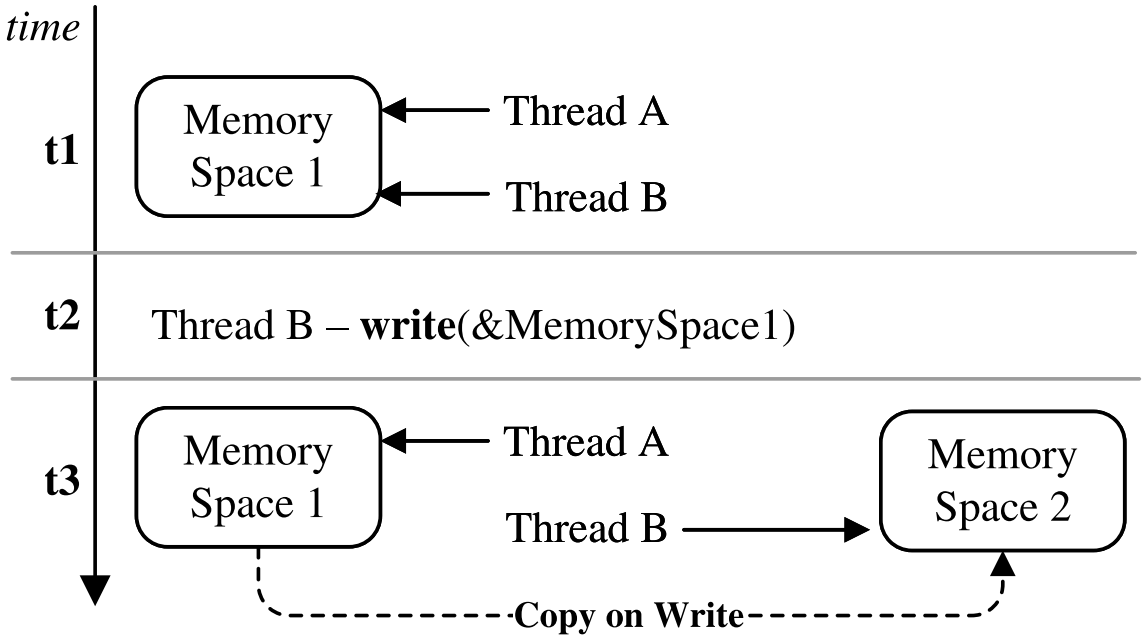


Figure 10 Principle of the Copy-on-Write (CoW) mechanism.

benefit of employing the NTM mechanism is that cached data from other applications are not trashed due to the copying required by any given application.

Since these approaches avoid the cache hierarchy, their operation is slower compared to `memcpy`. Intel shows that RAM-to-RAM achieves approximately half the speed of `memcpy` for large transfers (≥ 8 MiB) and many times slower for smaller transfers on x86 [13]. Table 4 depicts the results obtained by using NTM instructions.

The core code change to support NTM is shown on the right side of Table 4 as a for-loop structure. Initialization and termination has been omitted from this table. The procedure `_mm_stream_si32` is provided by Intel to call the appropriate assembly instruction for NTM operations. It copies 32 bits from a value (`src_buffer[i]`) to a given pointer (`dst_buffer[i]`). After the end of the for-loop, the data is copied to the area pointed by `dst_buffer`. Thus, the result is the same as calling `memcpy` but the related data will not be present in the caches if they were not already there before `_mm_stream_si32` is first called.

6 Results

This section presents the results about CoW and NTM using the three dataflow benchmarks and the 22 configurations.

6.1 Experimental Setup

All applications used in these experiments were generated using the PREESM framework (version 3.4), compiled using GCC v7.5.0 optimization -O2, and executed on Sniper simulator. The energy estimation is performed with McPAT [21], which is integrated into Sniper and provides reliable power and energy figures broadly used in state-of-the-art works [16, 17].

We develop an algorithm implemented in Python script language, which has as input the generated code of PREESM and has as output the new application code using the CoW or NTM technique. This algorithm detects in the code the memcpy patterns, which are candidates to be replaced by CoW or NTM. The algorithm is fully automatized, in the sense that

Table 3 Core change for supporting the CoW mechanism, assuming: (1) `src_buffer` is the source buffer, (2) `dst_buffer` is the destination buffer, (3) `copy_length` is the copy length, (4) `shm_open_fd` is a file descriptor created with the `shm_open` system call.

Original Code	CoW mechanism
`memcpy(dst_buffer, src_buffer, copy_length);`	`void *dst_buffer = mmap(NULL, copy_length, PROT_READ | PROT_WRITE, MAP_PRIVATE, shm_open_fd, 0);`

Table 4 Core change for supporting the NTM mechanism, assuming: (1) `src_buffer` is the source buffer, `dst_buffer` is the destination buffer, `copy_length` is the copy length.

Original Code	NTM mechanism
`memcpy(dst_buffer, src_buffer, copy_length);`	`for (i = 0; i < copy_length/4; i++) _mm_stream_si32(dst_buffer[i], *(src_buffer[i]));`

it detects the memcpy patterns by looking for join-broadcast, and broadcast dataflow patterns [14]. The algorithm can also be tuned with a parameter ψ, which allows defining a minimum threshold in the data size of a memcpy operation. By using ψ it is possible to eliminate small-size memcpy and only target the ones which transfer a large amount of data.

Algorithm 1 presents the method to detect the memcpy patterns which are candidates to be used in CoW and NTM. As input, the algorithm has three parameters: *src_o*: the application source code generated by PREESM; *t*: a flag that selects between CoW or NTM; and ψ: the parameter that indicates the minimum memcpy data size. Line 1 and 2 initialize two sets, called *join_broad_set* and *broad_set*, which will store the destination buffers' names of memcpys related to join-broadcast and broadcast-only, respectively. In line 3, the function *extract_memcpy* extracts from *src_o* all memcpy instance generated by PREESM, achieving the following information from each memcpy: data transfer *size*, *source* buffer, *destination* buffer, and the type (JOIN, BROADCAST, FORK, and ROUNDBUFFER [2]). The type is easily extracted due to a PREESM's characteristic in which it classifies the memcpy during its code generation, inserting its type as a comment in the line above each memcpy. All the memcpy instances are inserted in the list called *memcpy_list*.

Lines 5–14 identify join-broadcast patterns evaluating each element *mj* of *memcpy_list*. The condition of line 6 checks if the *mj* is a JOIN, if its destination buffer is not already in *join_broad_set*, and if the memcpy data size meets ψ. Once this check is true, the algorithm advances to the phase (lines 7 and 8) to confirm that the destination buffer of JOIN operation is also involved in BROADCAST. Once the buffer matches a join-broadcast pattern, it is inserted in the *join_broad_set* in line 9.

Lines 15-22 identify broadcast-only patterns evaluating each element *mb* of *memcpy_list*. Line 16 tests if *mb*'s type is BROADCAST. Another important verification is to check if the buffer is not in the *join_broad_set*, eliminating it if true. The same test of line 16 also seeks to eliminate the memcpys with a size lower than ψ. In case all conditions are meet, the destination buffer is added to the *broad_set* at line 18.

The last part of algorithm (lines 23–27) focused in verifying the value of *t*, calling the respective function which will applies CoW (line 24) or NTM (line 26). These functions evaluate all memcpys from *memcpy_list*, selecting those one containing the buffers name in *join_broad_set* and *broad_set*. The output of the algorithm is *src_t*, comprising the *src_o* with the matched memcpys replaced by the code presented in Table 3 and Table 4.

Algorithm 1: Patterning detection of CoW and NTM memcpy

```
   Input  : src_o (application source code generated
            by PREESM),
            t (techinique: CoW or NTM),
            ψ (minimum data size of a memcpy)
   Output: src_t (application source code adopting
            CoW or NTM)
 1 join_broad_set ← ∅;
 2 broad_set ← ∅;
 3 memcpy_list ← extract_memcpy(src_o);
 4 if memcpy_list ≠ ∅ then
       /* Identify join-broadcast patterns      */
 5     foreach mj ∈ memcpy_list do
 6         if mj.type = JOIN & mj.destination ∉
             join_broad_set & mj.size ≥ ψ then
 7             foreach mb ∈ memcpy_list do
 8                 if mb.type = BROADCAST &
                     mb.source = mj.destination &
                     mb.size ≥ ψ then
 9                     join_broad_set.insert(mj.destination);
10                     break foreach;
11                 end
12             end
13         end
14     end
       /* Identify broadcast-only patterns      */
15     foreach mb ∈ memcpy_list do
16         if mb.type = BROADCAST &
             mb.destination ∉ join_broad_set & mb.size
             ≥ ψ then
17             foreach mb ∈ memcpy_list do
18                 broad_set.insert(mb.destination);
19             end
20         end
21     end
22 end
23 if t = CoW then
24     src_t ← applies_CoW(src_o, join_broad_set,
         broad_set, memcpy_list);
25 else
26     src_t ← applies_NTM(src_o, join_broad_set,
         broad_set, memcpy_list);
27 end
```

Table 5 details the values of ψ (2nd column) used for each application. The table also details the number of memcpy addressed in CoW and NTM (3rd column), and its respective total size (4th column).

Table 5 Memcpy profile addressed in CoW and NTM.

Application	ψ	# memcpy	Total memcpy size
Stabilization	200KB	1	0.21 MB
Stereo	400KB	5	2.4 MB
SIFT	400KB	8	24.4 MB

Table 5 shows that SIFT has more available memcpy to be optimized. The ψ was defined as 400KB for SIFT and Stereo. Stabilization does not have such a large memcpy, therefore we reduce the value of ψ to allow the algorithm to consider the bigger memcpy size of the application. These values of ψ were achieved after a design-time analysis and represent to the best execution times achieved for each application.

The next subsections present the results achieved by replacing the memcpy either using NTM or CoW for the three applications.

6.2 Non-Temporal Memory Copying (NTM)

Figure 11a presents the iteration execution time for all benchmarks using NTM. It is noticeable that execution time is slightly reduced in most of the cases, reaching up to -5.3% for Stabilization in configuration 21. The average execution time reduction was -1.9% for Stabilization, -1% for SIFT, and -0.3% for Stereo.

NTM also provided a small energy reduction, in average -0.84% (σ=0.7) for Stabilization, -0.2% (σ=0.5) for Stereo, and -1.03 (σ=1.1) for SIFT, reaching up to -2.7% for SIFT at configuration 16.

Figure 11b focuses on SIFT (high memory footprint application) and presents a perspective between the bars: L3 miss rate, execution time, and energy, with the lines that show the absolute number of DRAM accesses without NTM and with NTM. It is possible to observe that energy is reduced in most configurations, reaching up to -2,7% to configuration 16. The L3 cache miss was barely affected, presenting an average decrease of -0.13% with a slight DRAM increase compared to its respective version without NTM (+0.14%).

Figure 12 presents results for all applications on configuration number 18 (private L2 and L3 shared by all cores), which was the cache configuration that, in general, presented the best results considering speed-up and L2/L3 miss rate from previous cache analysis (see Section 4). NTM has presented improvements for all applications on this configuration, specifically for SIFT and Stabilization. Note that, despite Stabilization has a low memory footprint, the execution time reduction is higher than SIFT and Stereo, at cost of more L3 miss rate. On another side, SIFT presents a modest execution time reduction, but also

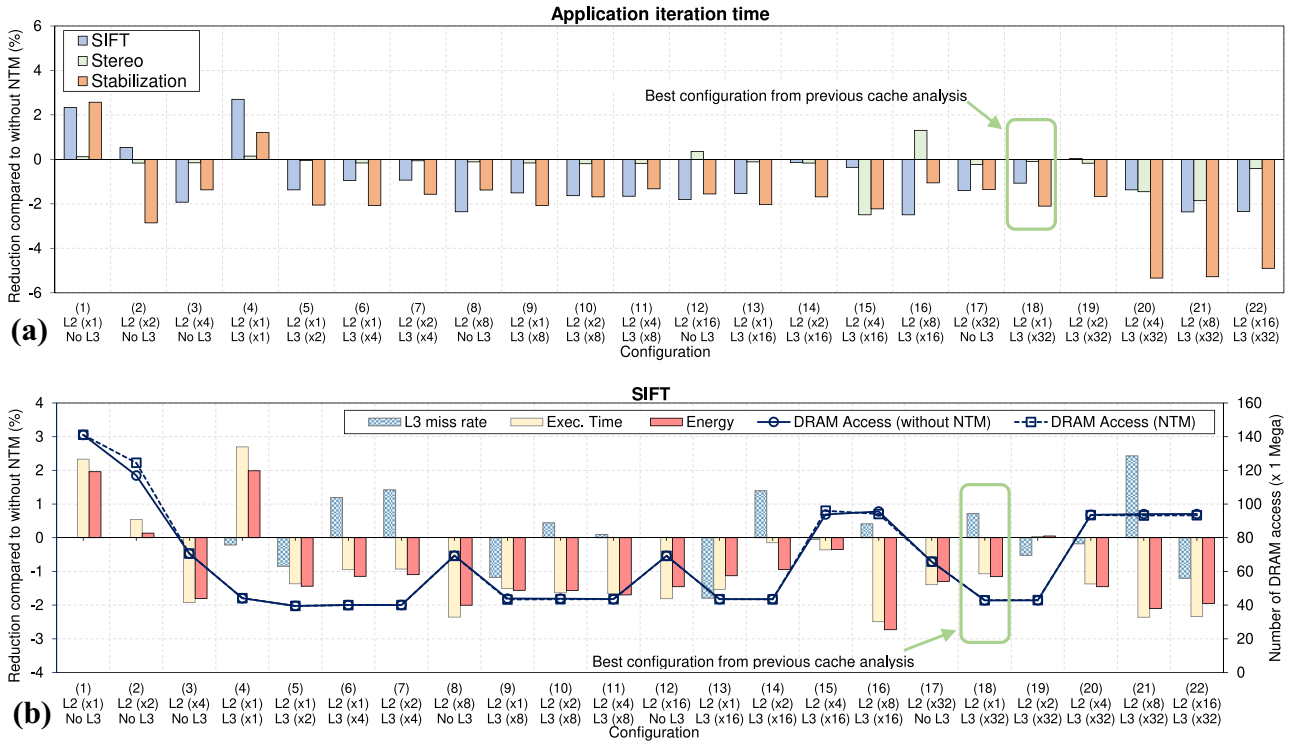


Figure 11 Results using NTM. (**a**) Evaluation of execution time. (**b**) Detailed evaluation for SIFT application.

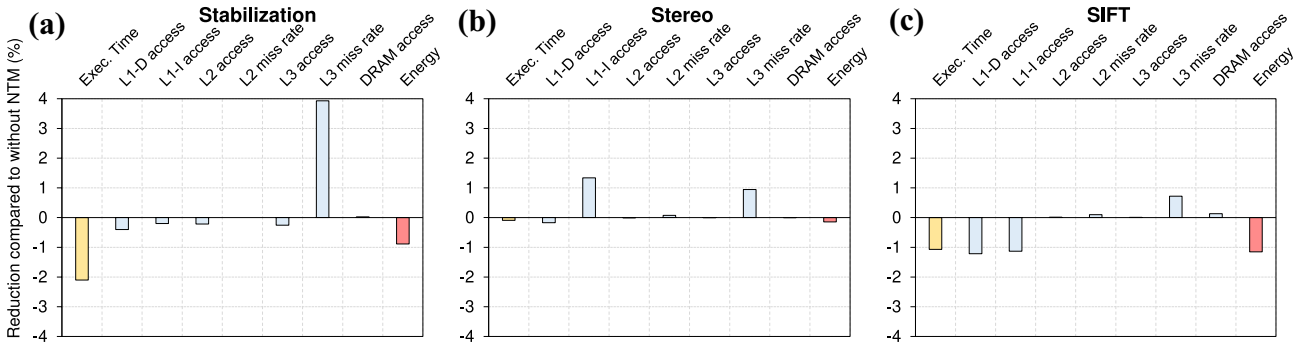


Figure 12 Results for configuration 18 using NTM. (**a**) Stabilization. (**b**) Stereo. (**c**) SIFT.

achieves reduction in all cache hierarchy, and specifically, in L1-D and L1-I access.

In summary of all results, it was possible to observe that NTM can improve the execution time and reduce energy, however, the gains were modest, not better than -1.9% in execution time and -2,7% in energy consumption considering all results.

6.3 Copy-On-Write (CoW)

Figure 13(a) presents the iteration execution time for all benchmarks using CoW. An average execution time reduction can be observed for Stabilization (-2%) and, most importantly, to SIFT (-10%), which reaches up to -15.8% for configuration 10. It is expected that SIFT benefits more from CoW since it has a large number of buffers used in memcpy compared to the other applications. On the other side, Stereo presents an average execution time increase of 1.3%. Stereo is known to be computation-intensive, and, therefore, the access to buffer mapped as CoW is less frequent than in Stabilization and SIFT, which makes the overheads of CoW (create shared memory and call of mmap()) overcome its benefits.

An energy reduction was achieved for all applications (-7.6% on average), with an average reduction of -2% for Stabilization, and -1.3% for Stereo. Again, SIFT is the application that benefits the most from CoW regarding energy. Figure 13b shows an overview of energy consumption for SIFT (bar graph) to the 22 configurations. On average, the energy reduction was -16.8%, reaching the best result of -21.8% to configuration 8. Again, SIFT benefits greatly from the CoW which allows data to be used without having to wait for the memcpy to complete. This behavior significantly affects the use of the CPU, which saves instructions in memcpy. This result can be observed following the dotted line of Figure 13b, which shows a significant reduction of L1-I (instruction cache) accesses of, on average, -62.3% (σ=3.4).

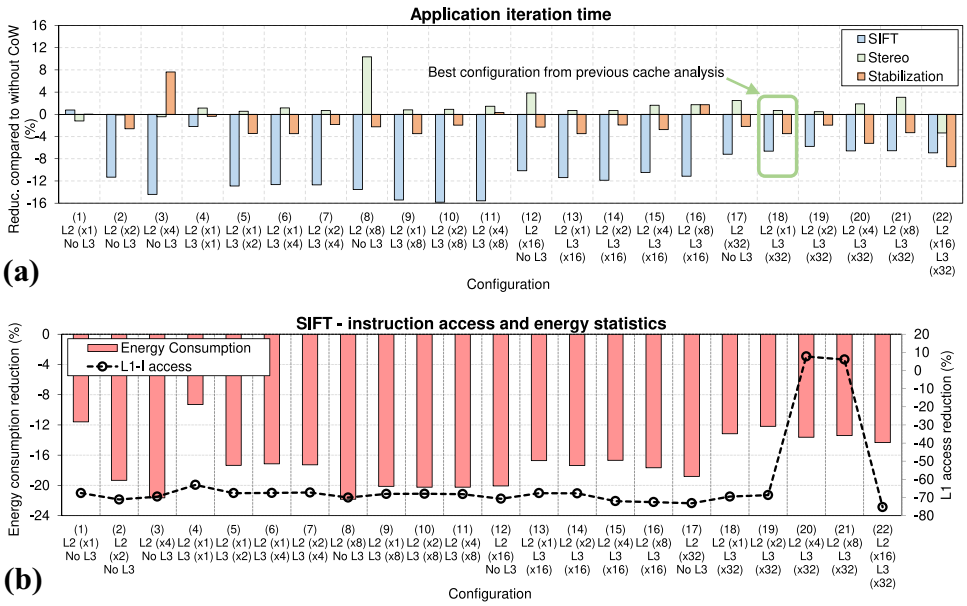


Figure 13 Results using CoW. (**a**) Execution time evaluation. (**b**) Energy evaluation.

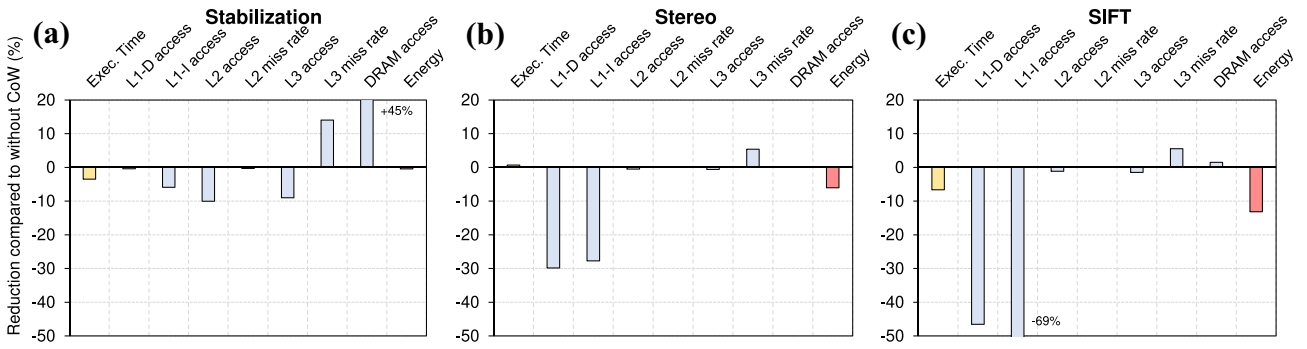


Figure 14 Results for configuration 18 using CoW. (**a**) Stabilization. (**b**) Stereo. (**c**) SIFT.

Figure 14 presents results for all applications in configuration 18. As expected, all applications have a reduction in the number of instruction access from the L1-I cache, which is justified by the saved memcpy instructions by using CoW. The instruction access gains progresses accordingly with the size of application's memcpy (as depicted in Table 5 (4th column)), with Stabilization presenting -0.41% less L1-I access, Stereo -29.8%, and SIFT -46.5%. This effect impacts the energy consumption and execution time, especially for SIFT, which benefits more from CoW due to its larger memory transfer profile.

7 Conclusion

This work presents a broad analysis of the impact of cache hierarchy configuration over static dataflow applications. In total, 37 different cache configurations (resulting in 213 simulations with 3 real applications) were adopted to evaluate variations in core count, L2/L3 sharing, and L2/L3 sizes. From this analysis, it is possible to conclude that bigger is not always better in terms of core count, L2 sharing, and L2/L3 size, since other aspects as efficient parallel workload division and computation/communication profile can prevent the application to benefit from more cache memory resources. This analysis shows that private L2 and L3 shared among all cores provide the best results in terms of application speed-up and L2/L3 cache miss for the adopted dataflow applications. As the second contribution, this work investigates the benefits of using copy-on-write (CoW) and non-temporal memory transfer copies (NTM) in dataflow applications. Results have shown that both techniques can contribute to improve execution time and save energy. NTM presents a modest reduction in execution time (up to -5.3%) and energy (up to -2.7%). CoW – specifically when used in applications with bigger memcpy transfers ($\geq$ 400KB) – shows important reductions, achieving up to -15.8% in execution time and -21.8% in energy consumption. These techniques are complementary to static state-of-the-art memory optimization approaches like [14], acting at runtime to reduce cache thrashing (NTM) and unnecessary data movements (CoW) among dataflow actors.

Future works include applying such evaluation in a different cache memory architecture, like distributed shared-memory systems. On the software side, some research can be conducted about source code generation from dataflow specification optimized to the cache hierarchy of the target SMP.

Funding Information This work is supported by the Agence Nationale de la Recherche under Grant No.: ANR-17-CE24-0018 We would like to give special thanks to the PREESM and Sniper communities for actively participating in the development of the tools which offer solid basements to this work.

Availability of Data and Material Not applicable.

Code Availability Not applicable.

Declarations

Conflicts of Interest The authors declare that they have no conflict of interest.

References

1. Furtunato, A. F. A., Georgiou, K., Eder, K., & Xavier-De-Souza, S. (2020). When parallel speedups hit the memory wall. *IEEE Access, 8,* 79225–79238. https://doi.org/10.1109/ACCESS.2020.2990418
2. Pelcat, M., Desnos, K., Heulot, J., Guy, C., Nezan, J., Aridhi, S. (2014). Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming. *In: European Embedded Design in Education and Research Conference (EDERC)*, pp. 36–40. https://doi.org/10.1109/EDERC.2014.6924354
3. Carlson, T. E., Heirman, W., & Eeckhout, L. (2011). Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. *In: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–12. https://doi.org/10.1145/2063384.2063454

4. Slingerland, N., & Smith, A. (2001). Cache Performance for Multimedia Applications. In: International Conference on Supercomputing (ICS), ICS '01, pp. 204–217. ACM, New York. https://doi.org/10.1145/377792.377833
5. Alves, M. A. Z., Freitas, H. C., & Navaux, P. O. A. (2009). Investigation of shared l2 cache on many-core processors. *In: International Conference on Architecture of Computing Systems*, pp. 1–10
6. Garcia, V., Gomez-Luna, J., Grass, T., Rico, A., Ayguade, E., & Pena, A. (2016). Evaluating the effect of last-level cache sharing on integrated GPU-CPU systems with heterogeneous applications. *In: IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–10. IEEE, New York (2016). https://doi.org/10.1109/IISWC.2016.7581277
7. Domagala, L., van Amstel, D., & Rastello, F. (2016). Generalized Cache Tiling for Dataflow Programs. *In: SIGPLAN/SIGBED, LCTES*, pp. 52–61. ACM, New York. https://doi.org/10.1145/2907950.2907960
8. Maghazeh, A., Chattopadhyay, S., Eles, P., & Peng, Z. (2019). Cache-Aware Kernel Tiling: An Approach for System-Level Performance Optimization of GPU-Based Applications. *In: Design, Automation, and Test in Europe (DATE)*, pp. 570–575. IEEE, Florence. https://doi.org/10.23919/DATE.2019.8714861
9. Stoutchinin, A., & Benini, L. (2019). Streamdrive: A dynamic dataflow framework for clustered embedded architectures. *Journal of Signal Processing System, 91*(3–4), 275–301. https://doi.org/10.1007/s11265-018-1351-1
10. Basilio, B. (2021). Fraguela and Diego Andrade: A software cache autotuning strategy for dataflow computing with upc++ depspawn. *Computational and Mathematical Methods 1*(1), 1–14. https://doi.org/10.1002/cmm4.1148
11. Bovet, D. P., & Cesati, M. (2006). *Understanding the Linux kernel*, 3rd edn., chap. 10, p. 295. O'Reilly
12. Intel Corporation. (2020). *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes*. Intel Corporation
13. Le, Q. T., Stern, J., & Brenner, S. (2020). *Fast memcpy with SPDK and Intel® I/OAT DMA Engine*. Retrieved March 15, 2021. https://software.intel.com/content/www/us/en/develop/articles/fast-memcpy-using-spdk-and-ioat-dma-engine.html
14. Desnos, K., Pelcat, M., Nezan, J. F., & Aridhi, S. (2016). On memory reuse between inputs and outputs of dataflow actors. *ACM Transactions on Embedded Computing Systems 15*(2). https://doi.org/10.1145/2871744
15. Kurd, N., Mosalikanti, P., Neidengard, M., Douglas, J., & Kumar, R. (2009). Next generation intel core micro-architecture (nehalem) clocking. *IEEE Journal of Solid-State Circuits, 44*(4), 1121–1129. https://doi.org/10.1109/JSSC.2009.2014023
16. Kim, T., Sun, Z., Chen, H., Wang, H., & Tan, S. X. (2017). Energy and lifetime optimizations for dark silicon manycore microprocessor considering both hard and soft errors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 25*(9), 2561–2574. https://doi.org/10.1109/TVLSI.2017.2707401
17. Rathore, V., Chaturvedi, V., Singh, A., Srikanthan, T., & Shafique, M. (2020). Longevity framework: Leveraging online integrated aging-aware hierarchical mapping and vf-selection for lifetime reliability optimization in manycore processors. *IEEE Transactions on Computers* pp. 1–1. https://doi.org/10.1109/TC.2020.3006571
18. PREESM. (2021). *PREESM Applications Repository* (https://github.com/preesm/preesm-apps).
19. Hamzah, R., & Ibrahim, H. (2015). Literature Survey on Stereo Vision Disparity Map Algorithms. *Journal of Sensors, 16*(1), 1–23. https://doi.org/10.1155/2016/8742920
20. Lowe, D. G. (1999). Object recognition from local scale-invariant features. *In: IEEE International Conference on Computer Vision (ICCV)*, vol. 2, pp. 1150–1157 vol.2. https://doi.org/10.1109/ICCV.1999.790410
21. Li, S., Ahn, J. H., Strong, R. D., Brockman, J. B., Tullsen, D. M., & Jouppi, N. P. (2009). Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. *In: International Symposium on Microarchitecture (MICRO)*, pp. 469–480. IEEE, New York, NY, USA.
22. IEEE. (2017). *IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications*, Issue 7. IEEE Std 1003.1-2017 1(1), 1–3951. https://doi.org/10.1109/IEEESTD.2018.8277153