On Cache Limits for Dataflow Applications and Related Efficient Memory Management Strategies

Alemeh Ghasemi, Rodrigo Cataldo, Jean-Philippe Diguet, Kevin Martin

On Cache Limits for Dataflow Applications and Related Efficient Memory Management Strategies

Alemeh Ghasemi
Univ. Bretagne-Sud, UMR 6285, Lab-STICC, F-56100
Lorient, France
alemeh.ghasemi@univ-ubs.fr

Rodrigo Cataldo
Univ. Bretagne-Sud, UMR 6285, Lab-STICC, F-56100
Lorient, France
rodrigo-cadore.cataldo@univ-ubs.fr

Jean-Philippe Diguet
CNRS, UMR 6285, Lab-STICC, F-56100 Lorient, France
jean-philippe.diguet@univ-ubs.fr

Kevin J. M. Martin
Univ. Bretagne-Sud, UMR 6285, Lab-STICC, F-56100
Lorient, France
kevin.martin@univ-ubs.fr
[firstname].[lastname]@univ-ubs.fr

ABSTRACT

The dataflow paradigm frees the designer to focus on the functionality of an application, independently from the underlying architecture executing it. While mapping the dataflow computational part to the cores seems obvious, the memory aspects do not match accordingly. Dataflow compilers usually do not consider the presence of caches when generating code. A generally accepted idea is that bigger and multi-level caches improve the performance of applications. Unfortunately, state-of-the-art dataflow compilers may prove the exception to this rule. This paper presents two efficient memory management strategies for dataflow applications through a study on the impact of sharing, size, and the number of levels of caches on them. 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. We propose two strategies, that can be used concurrently, to address the memory aspects of the dataflow model: copy-on-write and non-temporal memory transfers. Experimental results show that we speed up a computer stereo vision application by 2.1× and reduce the number of L1 data cache misses by 45% while maintaining the actors' source code and design intact.

CCS CONCEPTS

• **Software and its engineering** → **Source code generation**; **Memory management**.

KEYWORDS

Dataflow programming, Cache, Compilers, Dataflow framework

1 INTRODUCTION

The last decade has witnessed the rise of multi- and many-core architectures coming with new challenges and opportunities. On the hardware side, multiplying the number of cores has led to higher pressure on the memory subsystem, mitigated by a cache hierarchy. On the software side, new demands on parallel languages and tools appeared to help applications to fully exploit all the capabilities provided by the hardware. Existing for 40+ years, the dataflow programming model may eventually stand as the ideal approach to bridge the gap between application and architecture resources. The left side of Fig. 1 shows that a dataflow application is modeled by a graph in which (i) vertices are processing elements, called actors, interconnected via (ii) edges, which are FIFO buffers, and exchanging (iii) data, called tokens, in an asynchronous way. This model allows the designer to specify explicitly both temporal and spatial parallelism of the application. An actor can start execution only if the data is present in the input, and enough space is present in the output. So memory access plays a key role in dataflow performance.

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 improve 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. Although our experiments are performed on static dataflow applications, any dataflow compiler that does not consider the cache parameters may present similar behavior.

Static dataflow models allow an optimal schedule of the application according to the hardware features, and the target code can be automatically generated [4]. Fig. 1 depicts the generation of two sets of code: (1) for the actors based on the specification provided by the designer; and (2) for the framework responsibilities (e.g., scheduling, FIFO handling, mapping). This paper demonstrates that even optimally scheduled applications do not scale as desired with the increasing number of cores, size of caches, 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 paper is to explore the behavior of dataflow applications with caches and provide experimental results demonstrating the extent to which they impact the application's overall performance. For this, we consider several configurations, including non-available yet platforms or non-realistic cache configurations, and use the Sniper simulation tool [1] to foresee the scalability of the dataflow applications considered.

A total of 22 architecture configurations demanding more than 800 hours of simulation time are presented in this work. From its

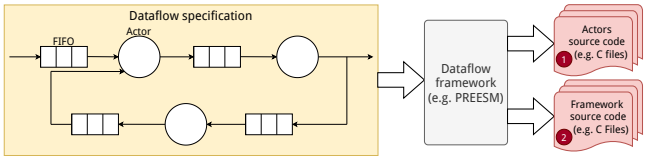


Figure 1: Dataflow programming model.

analysis, the second contribution of this paper is to propose two memory management solutions for dataflow code generation: copy-on-write (CoW) and non-temporal memory (NTM) copying. Both can reduce the number of cache misses and accelerate dataflow applications while modifying only the framework code. Therefore, no changes are required for the application design and code.

2 RELATED WORK

This section highlights studies that target the behavior of dataflow applications running on systems with a memory hierarchy. In [5], 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 [13], 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 [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 in [17]. 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.

Stoutchinin et al. [18] 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.

3 EXPERIMENTS

3.1 Experimental Setup

This work uses Sniper [1] to simulate multi- and many-core platform architectures. As the models of cores and caches are fully detailed in Sniper, it is a suitable tool for cache behavior evaluation. We employ the Xeon X5500 as the reference core processor. Detailed settings and configuration of the simulated hardware platforms are shown in Table 1a.

3.1.1 Dataflow framework and applications. We employ PREESM as the dataflow framework [15] and focus on two applications: (1) stereo matching, and (2) stabilization, taken from PREESM repository [16]. Stereo matching algorithms are used in many computer vision applications to process a pair of images, taken by two separated cameras in a small distance, and produce a disparity map that corresponds to the 3rd dimension (the depth) of the captured scene. Stereo matching algorithms and their implementations are still heavily studied as they raise important research problems [7]. The large memory requirements and challenging features of this 3D application in memory usage (cache locality) make it an appropriate candidate for the evaluation of cache impact. (2) The principle of video stabilization filtering 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.

These two applications are specified through the PREESM framework, which is responsible for the compilation of the dataflow applications and code generation (C code is this case) as shown in Fig. 1, including FIFO copies when required (for instance, for broadcast actors [2]). Giving these responsibilities to the framework allows to implement new memory strategies without modifying the actors code (identified as (1) in Fig. 1). The C code can then be compiled with any C compiler.

3.1.2 Mapping. The choice of mapping has a significant impact on the performance and memory behavior of an application. This work uses a workload-based mapping; in other words, we divide as fairly as possible the work available by mapping actors into the set of cores. This task is eased using the PREESM framework, as the latter provides a Gantt chart of operation cycles for all cores.

3.1.3 Design Space Exploration. We run multiple versions of the Stereo and Stabilization applications. Table 1b presents the characteristics of the applications, and details their FIFO usage: PREESM FIFOs are the input and the output FIFOs for connecting the actors. Additionally, the table details the framework's (i.e., PREESM) and

Table 1: (a) Hardware and (b) software settings.

(a) Baseline hardware settings.

Core Model	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

cyc = cycles; lat = latency; LRU = Least Recently Used.

(b) Software simulation settings.

Application	# actors	PREESM # FIFOs	PREESM FIFOs size	Memory copying[a] PREESM	Memory copying[a] Actors
Stabilization	34	604	0.8 MB	0.5 MB	1.3 MB
Stereo	36	811	500 MB	364 MB	13 MB

(a) sum of all copied memory using the `memcpy` procedure.

Table 2: Four cache architecture examples.

Architecture	Cores	Cache configuration	Cache size
(5) Private L2 & Private L3	8	(8 × 256KB) + (8 × 8MB)	66 MB
	32	(32 × 256KB) + (32 × 8MB)	264 MB
(10) Shared L2 (2×) & Shared L3 (8×)	8	(4 × 512KB) + (1 × 64MB)	66 MB
	16	(8 × 512KB) + (2 × 64MB)	132 MB

actors' memory copying behavior. In order to fully benefit from parallelism provided by dataflow modeling, we run Stereo and Stabilization with 4, 8, 16, and 32 threads, mapping one thread per core. In addition, multiple cache configurations are explored to evaluate their impact on the behavior of these applications, considering the following parameters: cache size, number of levels, and sharing degree. The memory controllers are attached to the LLC and are multiplied according to the configuration.

This work evaluates 22 architecture configurations: 7 for sharing cache up to 4 cores, 4 for up to 8 cores, 5 for up to 16 cores, and 6 for up to 32 cores. Priority was given to sharing up to 4 cores as they are commonly found in desktop computers and high-end mobile; however, we also explored other available and non-available yet platforms. They represent a diverse set of cache configurations and, for fairness, share the same total memory hierarchy size for a given combination of threads and cache levels. Table 1a depicts the baseline memory per cache level. Thus, for example, a 32-core architecture with private L2 provides 256KB of L2 cache individually and overall L2 size of 8MB; the same architecture pairing L2 with two cores provide 512KB of L2 cache individually and the same overall L2 size of the previous example. Table 2 depicts the calculation for two architecture settings. Unfortunately, we are unable to exhaust all cache architectures possibilities as, for instance, just the 32-core platform can have 22 cache hierarchy configurations given our restrictions. Additionally, we evaluate four controller setups: per 4, 8, 16, and 32 cores.

3.2 Experimental Results

3.2.1 Speedup with the number of threads. The experimental results indicate that despite the available parallelism, dataflow applications do not always improve as expected with the number of threads and bigger caches. The execution time of Stereo and Stabilization with various cache hierarchy configurations are depicted in Fig. 2a and 2b, respectively. Additionally, detailed memory hierarchy results are depicted in Table 3. Due to restricted space, we depict only results for 32 threads; yet, it should be noted that the other configurations present a similar trend.

Stereo does not scale up to 32 threads for the analyzed cache architectures: the highest speedup is achieved using 16 threads on the (4) architecture setup: Private L2 & Private L3. The following results were collected (normalized to the lowest speedup found): 4 threads speeds up from 1.20× (2) to 1.42× {(5), (6)}; 8 threads speeds up from 1.50× (8) to 2.11× (7); 16 threads speeds up from 1.23× (16) to 2.41× (4); 32 threads speeds up from 1.00× (22) to 1.45× (7); and average speedup of {1.31, 1.87, 1.88, 1.27}× respectively.

Stabilization presents a performance increase from 4 to 32 threads: the highest speedup is achieved using 32 threads on the (12) architecture setup: Shared L2 (×16). The following results were collected (normalized to the lowest speedup found): 4 threads speeds up from 1.00× (1) to 1.27× {(3), (5), (6)}; 8 threads speeds up from 1.67× (4) to 2.32× (8); 16 threads speeds up from 1.62× {(15), (16)} to 3.39× (14); 32 threads speeds up from 1.70× (15) to 4.68× (12); and average speedup of {1.19, 2.10, 2.13, 2.86}× respectively.

3.2.2 Trend analysis. The applications are divided into three trends. They share the first trend: *performance upward trend, transitioning from 4 to 8 threads.* Stereo speeds up from 1.31× to 1.87×, while Stabilization speeds up from 1.19× to 2.10×, on average. Their behavior can be interpreted by results of the number of DRAM accesses, which are illustrated in Fig. 3. In fact, distributing the actors over more cores decreases the average number of actors per core. With fewer actors per core, actors have enough space to fetch their FIFOs in L1-D. The performance of Stabilization improved in this range by sharing its LLC {(3), (5), (6), (8)} while slowing down in the same range with private LLCs {(1), (4)}; Stabilization can fit all FIFOs in the cache, thus, sharing the LLC results in decreasing the cost of copying shared FIFOs, and the latency overall. Stereo cannot fit all FIFOs in the cache; thus, the smaller caches penalize the performance {(2) and (8)}, while the bigger caches improve it {(5), (6), and (7)}.

The second trend is also shared by both applications: *plateau trend from 8 to 16 threads.* Stereo and Stabilization only speed up by 0.01× and 0.03×, on average. In this case, we have an even fewer number of actors per core and more space in D-cache size. However, we also have a penalty of less data locality due to the distribution of more FIFOs over the cache, resulting in more cache miss rates and DRAM accesses. Stereo is penalized when an intermediate cache is present (7) compared to a two-level cache system (8) due to excessive cache misses resulting from FIFO copying. Additionally, the difference of the (16) and (4) architectures, for 16 threads, is that the latter had a significantly higher average DRAM queuing delay than the former. The delay is caused as (16) and (4) have 4 and 1 memory controllers available. Stabilization is not as affected

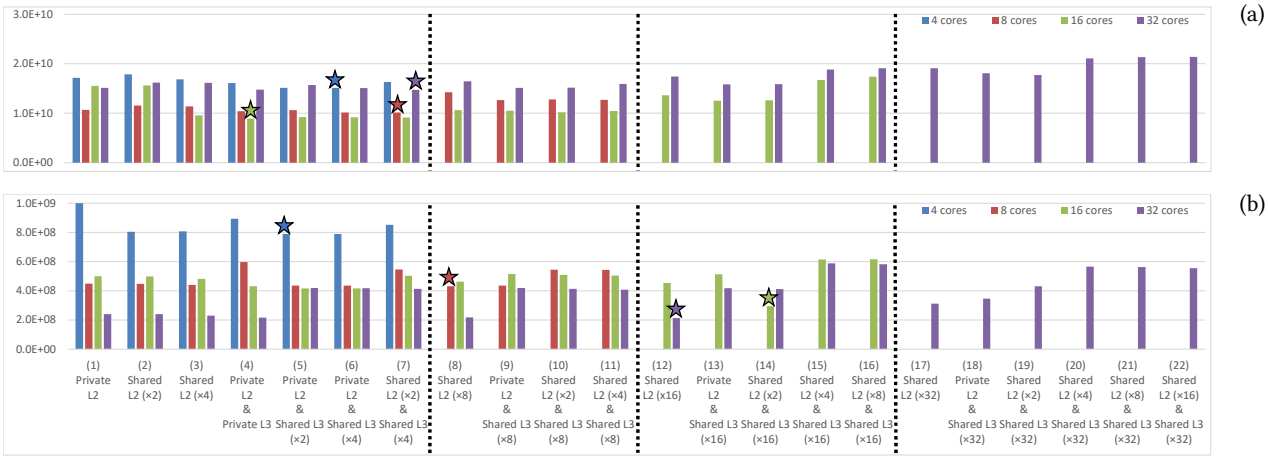


Figure 2: (a) Stereo and (b) Stabilization execution time for 4 combinations of threads on 22 architecture configurations. The star symbol identifies the highest speedup for a given combination of threads. The dotted black line separates the memory controller setup per 4, 8, 16, and 32 cores, respectively.

Table 3: Detailed memory hierarchy settings and results for Stereo and Stabilization employing 32 threads.

Architecture	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)	(19)	(20)	(21)	(22)
Cache Size (MB)	8	8	8	264	264	264	264	8	264	264	264	8	264	264	264	264	8	264	264	264	264	264
MemCtrl (#)	8							4				2					1					
Stereo																						
Avg DRAM delay (ns)	12	9	9	83	15	42	45	49	232	235	136	167	2748	2644	225	228	2114	104	250	491	549	506
Avg L1-D Miss Rate	11%	6%	6%	10%	6%	6%	6%	6%	6%	6%	6%	7%	6%	6%	6%	6%	6%	6%	6%	7%	7%	7%
Avg L2 Miss Rate	95%	85%	79%	94%	95%	95%	85%	70%	95%	85%	79%	66%	95%	86%	81%	78%	56%	96%	85%	79%	75%	73%
Avg L3 Miss Rate	–	–	–	76%	75%	67%	74%	–	58%	64%	67%	–	56%	61%	87%	91%	–	44%	49%	79%	85%	88%
Speedup[a]	**1.4×**	**1.3×**	**1.3×**	**1.4×**	**1.4×**	**1.4×**	**1.5×**	**1.3×**	**1.4×**	**1.4×**	**1.3×**	**1.2×**	**1.4×**	**1.4×**	**1.1×**	**1.1×**	**1.1×**	**1.2×**	**1.2×**	**1.0×**	**1.0×**	**1.0×**
Stabilization																						
Avg DRAM delay (ns)	4	2	2	1	1	0	0	17	1	1	1	91	3	7	18	60	235	152	172	89	173	117
Avg L1-D Miss Rate	< 1%																					
Avg L2 Miss Rate	95%	71%	50%	95%	95%	95%	72%	30%	95%	73%	51%	15%	95%	73%	57%	37%	3%	95%	72%	54%	36%	32%
Avg L3 Miss Rate	–	–	–	94%	69%	49%	66%	–	29%	39%	55%	–	15%	20%	48%	66%	–	1%	1%	16%	26%	57%
Speedup[a]	**4.2×**	**4.2×**	**4.4×**	**4.6×**	**2.4×**	**2.4×**	**2.4×**	**4.6×**	**2.4×**	**2.4×**	**2.5×**	**4.7×**	**2.4×**	**2.4×**	**1.7×**	**1.7×**	**3.2×**	**2.9×**	**2.3×**	**1.8×**	**1.8×**	**1.8×**

MemCtrl (#) = Number of memory controllers; Avg = Average; [a] Normalized to the slowest speedup found.

by DRAM as is Stereo, and, following its previous trend, tends to perform better with shared LLCs. The architecture (14) presents a near-perfect 1.95% miss rate, which resuls in the best speedup of this set of results.

Finally, the applications did not trend the same way from 16 to 32 threads. Stabilization has an *upward trend from 16 to 32 threads*, achieving an average speedup from 2.13× to 2.86×. Conversely, Stereo has a *downward trend for the same set of threads*, degrading the speedup from 1.88× to 1.27×. On the one hand, Stabilization can fit the entire FIFO set in the cache hierarchy; therefore, the hierarchy can provide fast access even with 32 threads. Table 3 shows that sharing a single LLC for all cores results in better data locality than the other settings (i.e., private or multiple LLCs). The same trend happens with the other threads sets. On the other hand, Stereo requires a challenging FIFO set for the cache hierarchy as it does not fit in any of our architecture configurations. Adding this FIFO set with more pressure on the memory hierarchy caused by the higher core count result in Stereo not scaling to 32 threads. Table 3 depicts prohibitive LLC miss rates for this application.

3.2.3 Cache size and performance. The results show that just increasing the cache size does not imply better performance. For instance, Table 3 shows that Stabilization is almost twice as fast in architecture (12) than (11), while the cache size is 8 MB and 264 MB, respectively. Memory operations by actors are private, as an actor can only be present in a single core at a time in PREESM. Meanwhile, memory operations by PREESM (i.e., input and output FIFOs) can be executed among cores and, thus, are shared according to the mapping used.

Stabilization tends to perform better with shared LLCs: {(5), (8), (14), (12)} achieved the highest speedup for 4, 8, 16, and 32 threads, respectively. When an intermediate cache is employed (i.e., L2 in a three-level cache), its benefit is dubious; in the plurality of cases, sharing it results in high miss rates that only decrease the performance. Table 1b is essential to understand this phenomenon. Multiple memory copying operations are used to share memory space among cores and FIFOs by PREESM; for Stabilization, these operations copy a total of 417 KB per application loop. As the copying procedure requires reading from one memory position and writing to another, in total, 834 KB are accessed during these

Table 4: 16 threads Stereo execution time, in seconds, for five replacement policies on two cache hierarchies.

Configuration	Replacement Policies				
	LRU	MRU	NMRU	NRU	Random
(4) Private L2 & Private L3	8.88	10.2	9.15	9.15	9.26
(7) Shared L2 (2×) & Shared L3 (4×)	9.11	10.4	9.2	9	9.12

LRU = Least Recently Used; MRU = Most Recently Used;
NMRU = Non-MRU; NRU = Not Recently Used;

operations; therefore, they can fit in any single L3, but only in some of the L2s (Table 1a and Table 2). The L2 must be shared at least 4× to have 834 KB for these operations. For these reasons, Stabilization improved its performance for shared LLCs.

Stereo uses the same memory copying mechanism as Stabilization; however, its case requires 728 MB (364 × 2) accesses overall. No cache hierarchy in our study can handle these accesses; therefore, Stereo performance is directly affected by the number of memory controllers, as DRAM accesses were up to 2 orders of magnitude higher than Stabilization, as shown in Fig 3. All the highest speedups of this application are found on the first 7 architecture designs that provide the most memory controllers (per 4 cores). No tendency to private or shared was shown since Stereo's miss rate is too high to matter. Using a memory footprint-based mapping can increase the locality of these application FIFOs. However, the performance will be limited by the unbalanced long chain of processing actors. Thus, we decided to employ a workload-based mapping.

3.2.4 Number of memory controllers. Fig. 3 depicts the number of DRAM accesses for all architectures defined in this work. As discussed previously, Stereo is a demanding application for DRAM. In every case there is a decrease of available memory controllers, from (7) to (8), (11) to (12), and (16) to (17), the number of DRAM accesses spikes upward. The same behavior can be discerned in Table 3 for average DRAM queue delay. Stabilization suffers from the same phenomenon; however, it speeds up regardless of the number of memory controllers because it can fit its FIFOs in the cache: from (16) to (17) Stabilization speeds up from 1.7× up to 3.2× even though the number of controllers were halved. The reason can be noticed by the average cache miss rate and the number of caches on Table 3 – the LLC miss rate dropped from 66% to 3%, and the system went from 3 to 2 cache levels.

3.2.5 Replacement policy. The replacement policy of caches directly affects applications that cannot fit its data in the cache; for our study, Stereo shows this behavior. Table 4 depicts the execution time for two architecture designs using five cache replacement policies. Even with significant miss rates present in Stereo (76% and 67% for LLC), there is still some temporal locality in the cache: the MRU replacement policy had the worst results for both architectures.

3.3 Findings

We summarize our most important findings in this section, presented as follows from a top- to bottom-level approach.

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

Both applications explored in this work present prohibitive data miss rates after the private L1-D cache. Only for some architectures and for Stabilization, the miss rate of LLC was acceptable {(17), (18), (19)}. The miss rates of L2 and L3 in these applications defeat the purpose of using caches; in fact, it decreases the application performance (for instance, from (12) to (13)). Besides, these miss rates are far from the expected cache behavior (i.e., $\leq 10\%$) [14].

Table 1b 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 employed to properly handle existing and future cache-based systems in regard to memory copying.

4 PROPOSED SOLUTIONS

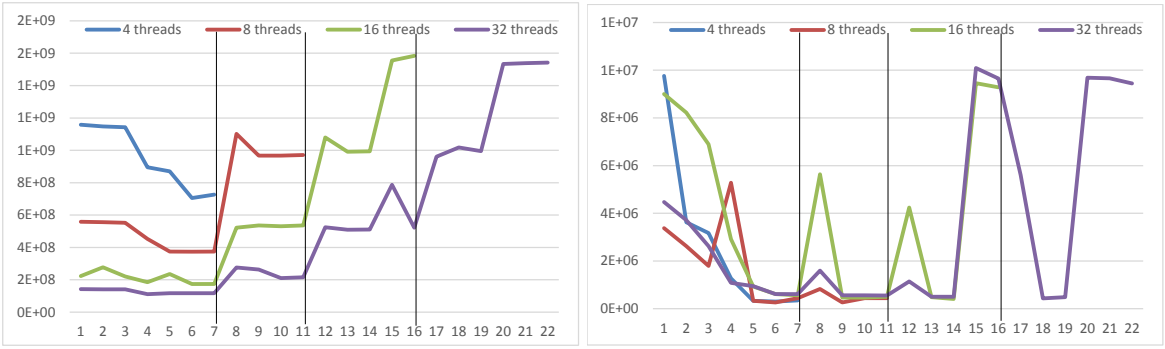
We propose two memory management strategies for a dataflow framework; as such, the modifications will enable the framework to be legacy-code compatible with existing dataflow applications (i.e., it only affects the code identified as (2) in Fig. 1). Nevertheless, the same strategies may also be used internally by the application. We posit the use of CoW and NTM mechanisms.

The mechanisms minimize the impact of cache trashing caused by the CPU replacing existing cached data with data requested by the `memcpy` procedure. In addition, as the CPU does not process the `memcpy` data – it just copies from one place to the other – the framework should either share the data by default (CoW) or make it independently of the cache hierarchy (NTM instructions or RAM-to-RAM DMA). Note that both solutions can be used concurrently; FIFOs that are mostly read can be shared with CoW, while FIFOs that must be private can employ NTM copying.

These FIFOs can be identified by the designer or anyone familiar with the application actors. PREESM is capable of parsing the BeanShell scripting language for information regarding the actors memory usage [3]. Currently, the parser allows the identification of read- or write-only FIFOs; we extended it for allowing the identification of read-mostly and private FIFOs.

4.1 Copy-on-Write (CoW) mechanism

CoW is achieved by employing the `mmap` system call. Dataflow is well-suited for this scenario as the application actors are recommended to use FIFOs as read-only data [2]. For example, FIFOs for Broadcast or Roundbuffer actors require the allocation of $n + 1$ FIFOs, where n is the number of FIFOs connected to the output ports, and the number of input ports of the actors, respectively [2]. These allocations can be exchanged by (i) creating a single shared memory area with `shm_open`, and (ii) creating multiple copies pointing to the same area with the `MAP_PRIVATE` flag as described in POSIX [9]. In UNIX-based system (e.g., Linux, BSD), this mechanism is extensively used for shared libraries. CoW will copy data only when one of its users requests a write operation. Even then, the copy is made



(a) Stereo **(b) Stabilization**

Figure 3: Number of DRAM accesses made by (a) Stereo and (b) Stabilization. The black lines separate the number of memory controllers for 4, 8, 16, and 32 cores, respectively. X axis is the architecture configuration as in Fig. 2

Table 5: Core change for supporting the CoW mechanism, assuming: (1) `src_fifo` is the source FIFO, (2) `dst_fifo` is the destination FIFO, (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
`1. memcpy(dst_fifo, src_fifo, copy_length);`	`1. void *dst_fifo = mmap(NULL, copy_length, PROT_READ | PROT_WRITE, MAP_PRIVATE, shm_open_fd, 0);`

on a per-memory page basis (typically, 4 KiB); therefore, mostly read structures will avoid unnecessary data copying.

The core code change to support CoW is shown on the right side of Table 5 as a single line. Initialization and termination has been omitted for this example. The CoW mechanism is achieved by mapping the destination FIFO (named `dst_fifo`) to the same physical address located in the 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 permission (`PROT_READ | PROT_WRITE`). The combination of these two flags will create a new copy of the physical address when a written has been made to the memory area (in other words, a copy-on-write). Finally, we allow the operating system to decide the virtual address of this new FIFO by passing nil (`NULL`) as the first parameter to `mmap`.

The CoW procedure is typically handled by the operating system kernel. Unfortunately, the Sniper simulator has limited operating system modeling capabilities to evaluate kernel-based strategies [1]. Thus, for our experiments, we used a combination of user- and kernelspace interaction so that Sniper can account execution time accurately. Specifically, the framework only activates CoW when data is written to a read-only FIFO. This means that writes to the data triggers the `SIGSEGV` signal and interrupts the causing thread[1]. Then, the framework changes the offending memory page to use CoW. In regard to the code presented in Table 5, we change the capability of the memory regions to read-only (`PROT_READ`) and install a signal handler to re-enable the write capability for this mapping.

Table 6 depicts the experimental results for our applications set. We have converted their most demanding FIFOs to use CoW: for Stereo, two FIFOs concerning the `weight` variables have each 19 copies (9 MB each) that now share a single 9 MB area; for Stabilization, three YUV video copies are mapped to the same area sizing 71, 54, and 219 KB, respectively.

We used two architecture designs for evaluation: (2) and (4). Both of them support all core configurations used in this work, while the former and latter have two and three cache levels, respectively. Besides, architecture (2) and (4) have up to 8MB and 264MB cache size available (Table 3); thus, we can evaluate the impact of CoW on disparate scenarios. Stereo is expected to be more impacted than Stabilization by CoW given the conversions done for this experiment; i.e., Stereo `weight` FIFOs do not fit either the L1 or L2 cache and occupy 65% of the L3. Since a `memcpy` operation needs a source and destination positions, they will not fit even in the L3. However, Stabilization YUV copies can fit in the L2 cache even when `memcpy` is employed.

The results show that Stereo speeds up to 1.60×, on average, and 2.1×, for 16 threads, by employing CoW. Table 6 also shows that cache accesses and misses of Stereo have decreased significantly because copying is made only when necessary. Therefore, valuable cache space is kept and the application can run faster in 7 out of 8 scenarios. For 32 threads on architecture (4), the application kept the same performance while reducing cache accesses and misses. As CoW initially requires handling multiple FIFO memory area via `mmap` calls and latter interrupting the execution to do memory copying, it presents an overhead higher than a simple call to the `memcpy` procedure. Even so, this overhead was negated by the gains in the cache system. DRAM accesses and delay have also been reduced in all cases. However, the impact for DRAM delay on architecture (4) is more significant given that it already had the highest delay on the set of architectures supporting all core configurations (83 ns for 32 threads – Table 3). Finally, the architectures presented a disparity in the number of instructions by using the same binary code. The

[1] `SIGSEGV` is a synchronously-generated signal and is guaranteed to be delivered to the causing POSIX thread [9].

Table 6: Architecture characteristics using CoW. Besides speedup, values are reduction (-) and increase (+) in percentage of the CoW mechanism against the original code.

(a) Stereo								
Architecture	(2) Shared L2 (×2)				(4) Private L2 & Private L3			
Threads	4	8	16	32	4	8	16	32
Speedup	**1.92×**	**2.07×**	**2.10×**	**1.58×**	**1.60×**	**1.32×**	**1.19×**	**1.00×**
Instructions	-19.18%				-3.09%			
L1-D Accesses	-59.90%	-59.89%	-59.95%		-9.18%			-9.31%
L1-D Misses	-45.80%	-45.79%	-45.83%	-47.02%	-45.79%		-45.82%	-45.81%
L2 Accesses	-45.43%			-45.42%	-45.77%			-45.75%
L2 Misses	-48.66%	-48.62%	-48.73%	-48.80%	-48.01%	-48.02%	-48.04%	
L3 Accesses	—				-47.99%	-48.00%	-47.98%	47.99%
L3 Misses	—				-60.97%	-60.05%	-59.62%	-62.92%
DRAM Accesses	-53.48%	-53.26%	-53.40%	-53.58%	-69.44%	-65.47%	-62.92%	-72.45%
DRAM Delay	-1.54%	-2.99%	-0.73%	-0.45%	-69.70%	-21.19%	-26.30%	-21.06%
(b) **Stabilization**								
Architecture	(2) Shared L2 (×2)				(4) Private L2 & Private L3			
Threads	4	8	16	32	4	8	16	32
Speedup	**1.03×**	**1.02×**	**1.03×**	**1.05×**	**1.00×**	**1.00×**	**1.00×**	**1.00×**
Instructions	< -0.01%							
L1-D Accesses	-0.66%				-0.04%			
L1-D Misses	-18.26%	-17.25%	-16.60%	-16.37%	-18.05%	-16.97%	-16.48%	-18.39%
L2 Accesses	-17.53%	-17.11%	-16.33%	-16.11%	-17.75%	-16.86%	-16.25%	-15.93%
L2 Misses	-6.73%	-2.39%	-6.84%	-5.83%	-2.68%	-9.14%	-10.74%	-11.02%
L3 Accesses	—				-10.12%	-13.49%	-13.11%	-12.66%
L3 Misses	—				+5.79%	-14.41%	-13.39%	-13.60%
DRAM Accesses	-19.52%	-3.26%	-14.08%	-12.96%	+1.62%	-1.09%	-3.95%	-6.98%
DRAM Delay	+6.58%	+2.91%	+0.70%	+0.14%	-13.63%	-1.80%	-0.13%	+0.12%

reason for this is the speculative nature of the cores combined with their branch prediction unit: even a small number of branches (15) can execute 25% more instructions by predicting them wrong [8].

Stabilization speeds up to 1.02×, on average, by employing the same strategy. Therefore, CoW had minimal impact in this application. This shows that CoW should be preferably used for large memory areas, as its overhead is higher than `memcpy`. In addition, the number of cache misses increased in a case for Stabilization (4 threads on architecture (4)). As copies are avoided with CoW, and Stabilization could fit them in the L2, some of the cache's temporal locality is lost. DRAM access and delay do not affect this application significantly (Section 3.2.4), thus the observed variation did not affect the application execution time negatively. For example, Architecture (2) with 32 threads increased the DRAM delay in 0.14% (from 0.06 to 0.08 ns) while speeding up 1.05× the total execution time.

For both applications, architecture (2) was more affected by CoW than architecture (4). Such condition was expected as (2) has less cache space and levels available than (4). Consequently, memory copying will rely more on slower caches or, even, DRAM accesses. When these accesses are reduced by employing CoW, it directly impacts the application performance.

4.2 Non-Temporal Memory (NTM) Copying

NTM copying is ideal for memory locations 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 [10], and (ii) through the I/OAT DMA engine available in some processor designs [11]. In any case, the `memcpy` procedure is replaced for another procedure that uses either technique. Another 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 [11]. Ergo, we have excluded Stabilization from this experiment as it does not transfer large memory areas. Table 8 depicts the results obtained by using NTM instructions.

The core code change to support NTM is shown on the right side of Table 7 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_fifo[i]`) to a given pointer (`dst_fifo[i]`). After the end of the for-loop, the data

Table 7: Core change for supporting the NTM mechanism, assuming: (1) `src_fifo` is the source FIFO, `dst_fifo` is the destination FIFO, `copy_length` is the copy length.

Original code	NTM mechanism
`1. memcpy(dst_fifo, src_fifo, copy_length);`	`1. for (i = 0; i < copy_length/4; i++) 2. _mm_stream_si32(dst_fifo[i], *(src_fifo[i]));`

Table 8: Architecture characteristics using NTM copying.

	Stereo			
	4 th.	8 th.	16 th.	32 th.
Speedup	**1.00×**	**1.01×**	**1.03×**	**1.03×**
Instructions	+23.6%	+23.1%	+22.9%	+22.6%
Cache-misses	-45.9%	-53.8%	-19.9%	-36.0%
dTLB-misses	-90.6%	-55.7%	-51.0%	-64.4%

th = threads; TLB = Translation Lookaside Buffer.

is copied to the area pointed by `dst_fifo`. Thus, the result is the same as calling `memcpy` but the related data will not be present in the caches if they were no already there before `_mm_stream_si32` is first called.

Sniper does not support NTM instructions and extending its capabilities is outside of the paper scope; therefore, the evaluation of this mechanism was conducted on a 2× Intel Xeon Silver with 32 cores, private L1 and L2 caches (64 KB and 1 MB, respectively) and a shared L3 cache of 13.75 MB per processor. Besides, each processor has two memory controllers. The cache design matches the architecture (13) explored in Section 3.1.3. Experimental results were conducted up to 10 times with hardware and software counters (perf [12]), and their average is presented.

Stereo speeds up by 1.01× on average by employing NTM copying. As discussed previously, these operations are not faster than the `memcpy` procedure, but they provide other benefits: Table 8 shows that cache and data TLB misses are reduced by 38.9% and 65.4% on average, respectively. These results demonstrate that the cache and TLB have been used more effectively, leading to faster execution time. However, the use of NTM increased the number of instructions by 23% on average. We employed the `_mm_stream_si32` procedure for copying 32 bits at a time; newer architectures (Skylake and after) can use the 128-bit version for reducing the number of calls. In this case, the increase of instructions is reduced to 5%, and results are similar to those presented here.

5 CONCLUSION

We present the behavior and performance of two dataflow applications on 22 architecture configurations comprised of various cache levels, sizes, and number of memory controllers. Both presented poor cache locality resulting from the generated dataflow code. Thus, we proposed two solutions (copy-on-write and non-temporal memory transfer) to address this phenomenon so that the dataflow methodology can fully enjoy the benefits of the cache hierarchy. Experimental results showed that we were able to reduce dramatically the cache miss rate for Stereo application (60% less in the best case) and speed up the applications up to 2.10× without changing the actors' source code.

ACKNOWLEDGMENTS

This work is supported by the Agence Nationale de la Recherche under Grant No.: ANR-17-CE24-0018 (https://anr.fr/Projet-ANR-17-CE24-0018)

REFERENCES

[1] T. Carlson, W. Heirman, and L. Eeckhout. 2011. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *SC* (2011). ACM Press, Seattle, 1–12. https://doi.org/10.1145/2063384.2063454

[2] K. Desnos. 2014. *Memory Study and Dataflow Representations for Rapid Prototyping of Signal Processing Applications on MPSoCs.* Ph.D. Dissertation. UEB.

[3] Karol Desnos. 2020. Advanced Memory Footprint Reduction. https://preesm.github.io/tutos/advancedmemory/.

[4] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi. 2015. Memory Analysis and Optimized Allocation of Dataflow Applications on Shared-Memory MPSoCs. *J Sigl Process Syst* 80, 1 (2015), 19–37. https://doi.org/10.1007/s11265-014-0952-6

[5] L. Domagala, D. van Amstel, and F. Rastello. 2016. Generalized Cache Tiling for Dataflow Programs. In *SIGPLAN/SIGBED* (2016) *(LCTES 2016)*. ACM, New York, 52–61. https://doi.org/10.1145/2907950.2907960

[6] V. García, J. Gomez-Luna, T. Grass, A. Rico, E. Ayguade, and A. Pena. 2016. Evaluating the effect of last-level cache sharing on integrated GPU-CPU systems with heterogeneous applications. In *IISWC* (2016-09). IEEE, New York, 1–10. https://doi.org/10.1109/IISWC.2016.7581277

[7] R. Hamzah and H. Ibrahim. 2015. Literature Survey on Stereo Vision Disparity Map Algorithms. *Journal of Sensors* 16, 1 (Dec 2015), 1–23. https://doi.org/10.1155/2016/8742920

[8] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (Jan. 2019), 48–60. https://doi.org/10.1145/3282307

[9] IEEE. 2020. IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7. *IEEE Std 1003.1-2017* 1, 1 (Jan 2020), 1–3951. https://doi.org/10.1109/IEEESTD.2018.8277153

[10] Intel Corporation. 2020. *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes.* Intel Corporation.

[11] Q.-T. Le, J. Stern, and S. Brenner. 2020. Fast memcpy with SPDK and Intel® I/OAT DMA Engine. https://software.intel.com/content/www/us/en/develop/articles/fast-memcpy-using-spdk-and-ioat-dma-engine.html.

[12] Linux Kernel. 2020. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[13] A. Maghazeh, S. Chattopadhyay, P. Eles, and Z. Peng. 2019. Cache-Aware Kernel Tiling: An Approach for System-Level Performance Optimization of GPU-Based Applications. In *DATE* (2019-03). IEEE, Florence, 570–575. https://doi.org/10.23919/DATE.2019.8714861

[14] D. Patterson and J. Hennessy. 2013. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface* (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco. 1–800 pages.

[15] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J. Nezan, and S. Aridhi. 2014. PREESM: A Dataflow-Based Rapid Prototyping Framework for Simplifying Multicore DSP Programming. In *EDERC*. IEEE, Milano, 36–40. https://doi.org/10.1109/EDERC.2014.6924354

[16] preesm-apps. 2020. Repository for PREESM applications. https://github.com/preesm/preesm-apps.

[17] N. Slingerland and A. Smith. 2001. Cache Performance for Multimedia Applications. In *ICS* (2001) *(ICS '01)*. ACM, New York, 204–217. https://doi.org/10.1145/377792.377833

[18] A. Stoutchinin. 2019. *A Dataflow Framework For Developing Flexible Embedded Accelerators.* Ph.D. Dissertation. Unibo.