# The Home-Forwarding Mechanism to Reduce the Cache Coherence Overhead in Next-Generation CMPs

**Gabriele Mencagli***, **Marco Vanneschi, and Silvia Lametti**
Department of Computer Science, University of Pisa
Largo B. Pontecorvo 3, I-56127, Pisa, Italy
Email: {mencagli, vannesch, lametti}@di.unipi.it

**ABSTRACT**

On the road to computer systems able to support the requirements of exascale applications, *Chip Multi-Processors* (CMPs) are equipped with an ever increasing number of cores interconnected through fast on-chip networks. To exploit such new architectures, the parallel software must be able to scale almost linearly with the number of cores available. To this end, the overhead introduced by the run-time system of parallel programming frameworks and by the architecture itself must be small enough in order to enable high scalability also for very fine-grained parallel programs. An approach to reduce this overhead is to use non-conventional architectural mechanisms revealing useful when certain concurrency patterns in the running application are statically or dynamically recognized. Following this idea, this paper proposes a run-time support able to reduce the effective latency of inter-thread cooperation primitives by lowering the *contention* on individual caches. To achieve this goal, the new *home-forwarding* hardware mechanism is proposed and used by our runtime in order to reduce the amount of cache-to-cache interactions generated by the *cache coherence* protocol. Our ideas have been emulated on the Tilera TILEPro64 CMP, showing a significant speedup improvement in some first benchmarks.

***Keywords:*** Parallel Processing, Cache Coherence, Fine-grained Parallelism, Chip Multi-Processors

## 1 Introduction

Recent advances in microprocessor design have been reflected in high-performance computing architectures that rely on *Chip Multi-Processors* (CMPs) as basic building blocks. According to the new interpretation of Moore's law, the number of cores per chip will continue to double every two years, and prototypal architectures with thousands of cores per chip (like Adapteva Epiphany with up to $4,096$ cores) are now becoming reality [1]. Future CMPs must be equipped with high-speed on-chip interconnection networks (like optical networks [2]) and connected to very high-bandwidth 3D-stacked memory sub-systems. Along this line, hardware *cache coherence* (CC), which is still an expected feature of future CMPs for both technical and legacy reasons [3], still needs new advancements to support such architectures with the necessary scalability.

According to this future path, the gap between parallel architectures and parallel programming maturity tends to widen. To exploit at best the hardware potential, the performance of parallel software must scale almost linearly with the number of cores of next CMPs. This goal poses serious challenges in the design of the run-time support of parallel programming frameworks. In fact, from one side the exploitation of such large set of cores requires that a high number of concurrent activities (*tasks*) can be statically or dynamically identified. On the other side, good scalability can be achieved as long as the tasks computation time (*granularity*) is sufficiently large than the run-time overhead. The capability of executing small tasks with frequent synchronizations in an efficient way is prerogative of run-time supports targeting *fine-grained parallelism* [4].

Several papers have presented some solutions to enable fine-grained parallelism by reducing the run-time system overhead using lock-free data structures for low-latency thread cooperation [5], by improving load balancing using sophisticated work stealing techniques [6], or by supporting autonomic features [7–9]. We claim that the low-level sources of architectural overhead in CMPs must be formally analyzed and some countermeasures properly designed by introducing specific architectural mechanisms directly exploitable by the run-time system. As suggested in Ref. [3], the memory hierarchy and more specifically the cache subsystem is one of the best candidate for such study.

One of the approaches described in recent research papers consists in configuring the CC mechanisms in such a way as to exploit a specific sharing pattern of data and reduce the CC traffic. Examples are described in Refs. [10] and [11], where the authors have designed hardware components able to detect sharing patterns between cores by dynamically analyzing the sequence of memory accesses. The goal is to enforce hybrid configurations of invalidation-based

*Corresponding author. Phone: +390502213132, Fax: +390502212726

and update-based CC protocols in order to reduce the number of messages exchanged among caches.

Our approach has some analogies with such previous work. Our goal is to design a runtime for *structured parallel programs* [12], also known as Algorithmic Skeletons [13] and recently as parallel patterns [14]. They are based on the instantiation and composition of well-known parallelism forms (e.g., farm, map, pipeline, stencils, reduce, divide&conquer) with a precise cooperation semantics. Our fundamental observation is that the run-time support for such patterns can be designed with few base synchronization mechanisms. This allows us to orchestrate the CC protocol in such a way as to optimize the communication overhead, which is precondition to enable scalable fine-grained parallelism.

This work extends the paper published in Ref. [15] by providing the following specific contributions:

- the cost of CC protocols will be described and evaluated through benchmarks on CMPs;
- we will list the requirements for an efficient run-time support. Then, we will introduce the *home-forwarding mechanism*, which allows us to reduce communications among caches in the implementation of parallel patterns;
- we will describe a run-time support that matches our requirements;
- we will evaluate experimentally our approach through some benchmarks on the Tilera TILEPro64 CMP [16].

The organization of this paper is the following. In the next section we point out the motivation of our work. Sect. 3 will review some related works, and Sect. 4 will describe the nature of the CC overhead. Sect. 5 will give the basis for an efficient runtime design, and Sect. 6 will describe a runtime that meets our design principles. Sect. 7 will evaluate our runtime on some parallel programs. Finally, Sect. 8 will conclude this paper by outlining our future research directions.

## 2 Motivation

Any run-time support needs proper mechanisms for synchronizing *processing elements* (briefly, PE)[1]. We can distinguish between two basic synchronization problems: *symmetric synchronization* for mutual exclusion, and *asymmetric synchronization* for event notification.

The first problem is solved by adding *lock*/*unlock* primitives around the critical sections. In the second problem a *precedence relation* must be forced. As an example, let $PE_i$ and $PE_j$ be two PEs executing the sequences of operations $\{p;c_1;q\}$ and $\{r;c_2;s\}$, and suppose that the execution of $c_1$ must precede the execution of $c_2$. This can be expressed as follows: $\{p;c_1;notify(go);q\}$ and $\{r;wait(go);c_2;s\}$, where *notify* and *wait* generate and wait for an abstract *event*, i.e. *a pure one-to-one synchronization*. The wait primitive implies *busy-waiting* that can be implemented as a spin-loop on a shared boolean flag, or as an I/O inter-processor communication. In general, atomic instructions are not needed to perform the event notification/reception.

This distinction is important. Asymmetric synchronization is predominant in the design of run-time supports for structured parallel programs [12, 13, 17]. Such programs are characterized by the fact that logically the ownership of data structures is transferred among threads according to a *producer-consumer* scheme. As an example, in a *farm* pattern an emitter functionality is responsible to dispatch *tasks* (data items) to a set of workers by transferring the ownership of them. The availability of a new task can be notified using asymmetric synchronization, and the emitter, once transferred the ownership, no longer needs the data item forwarded. Analogously, in a *map* pattern a data structure (often a large array or a matrix) is scattered in partitions whose ownership is assigned to a set of independent workers.

The producer-consumer scheme can be optimized in terms of CC actions. In particular, we can note that:

- once the ownership has been transferred, some unnecessary CC interactions (e.g., read requests and invalidations) can be raised by operations performed by the owner, because some cache lines of the data might still be in the private caches of the PE that held the data;
- messages between caches not only increase the latency of read/write operations, but also generate *contention* among caches. In fact, *caches act as servers*, i.e. they receive requests from other caches and reply to them. High contention means long waiting times, and the real (under-load) latency of read/write operations experienced by a program can hamper scalability with high degrees of parallelism.

Optimizations aimed at reducing CC messages and contention can be applied to the run-time support of structured parallel programs, which can be based on event notification as the main synchronization mechanism used by threads. This will be the goal of our approach.

## 3 Related Works

The evaluation of the CC overhead on multiprocessors and more recently on CMPs has been studied in several research papers. In Refs. [18–21] an evaluation has been carried out on systems with invalidation-based CC. The results have been obtained empirically through benchmarks aimed at evaluating the base latency of read operations in different CC configurations. In Sect. 4.2, we performed similar benchmarks on the Intel Sandy Bridge and the TILEPro64 CMPs. In addition, we proposed an analysis of the w latency in the case of synchronous writes (with memory fences in the case of WMO machines), which are used in inter-process/inter-thread cooperation mechanisms. In other papers the CC evaluation has been performed by adopting several simplifications on the workload model, in order to predict analytically the overhead by using tools such as Markov chains [18,22], Generalized Timed Petri Nets [23] and Queuing Networks [24]. In this paper we are not interested in the

[1] In this paper we use the generic term *processing element* to denote the basic unit of parallelism at the architectural level (e.g., a core of a CMP).

exact quantification of the CC overhead, but rather in strategies to design the runtime system in order to reduce both base latency and contention.

To reduce the overhead of CC, a general approach consists in designing hardware components able to analyze the requests generated by the processors and to make speculative CC requests when a particular access pattern is detected. The rationale is to provide optimizations for computations that exhibit specific memory access patterns. An example is the work in Ref. [25], where the authors propose an approach to prefetch data and reduce CC interactions in the case of embedded processors. The idea of exploiting specific access patterns to optimize CC is also the motivation of our work.

There is a dichotomy between hardware-level approaches, in which *dynamic* optimizations are taken by the hardware interpreter [26], and *static* optimizations applied by the runtime developer or by a compiler through the proper use of non-conventional architectural mechanisms visible at the assembler level. The first approach does not need any assumptions on the running applications, because it is the hardware level, with a proper additional logic, which is in charge of detecting the cooperation patterns (e.g., cache interactions) and triggers some actions to optimize the architectural behavior. Despite the larger applicability, this approach contributes to increase the CPU chip complexity, thus to increase its area and, most important, its power consumption. Instead, in the second approach some knowledge of the parallel application structure must be owned by the developer of the run-time system or by the compiler, in order to use non-conventional and easy-to-implement low-level mechanisms provided by the hardware. In this way the hardware design is kept simple and efficient and compatible with the future trend of integrating ever more cores in the same chip. Leveraging the structured parallel programming methodology, in this paper we follow this second solution.

The problem of contention is critical in CMPs. In Ref. [27] an approach for mapping of real-time tasks onto PEs has been proposed to increase predictability of their running times. It consists in constraint programming techniques to find the optimal static task-to-PE mapping. This method has been applied with micro-benchmarks on the TILEPro64 CMP. It starts from a given set of tasks exchanging messages, and finds a way to map them in the architecture. As stated in Sect. 5, process/thread mapping is only one side of the problem to reduce contention. Even when the mapping is chosen accurately, high contention may still exist between PEs if the runtime does not exploit proper architectural mechanisms to deal with it. The task scheduling approach studied in Ref. [28] tries to reduce the base latency only.

The possibility of complementing CC with hardware mechanisms to reduce messages has been studied in the past. Similarly to our *home-forward* synchronous store, in Ref. [10] the authors propose the so-called remote store model, in which processes have private memory areas and a process can write directly into the memory of another process using remote stores. Although the similarities, this mechanism uses atomic instructions for the synchronization on remote writable areas while our runtime is based on asymmetric synchronization only, thus it is completely lock-free. Moreover, no consideration about the reduction of CC overhead and invalidation/C2C read requests has been made, while our home-forward technique takes into account CC explicitly.

Finally, a very recent (2015) research work with commonalities with this paper is the one in Ref. [11]. The authors introduce a hybrid CC approach that trades off the advantages of invalidation-based and update-based solutions. Their idea is to develop a hardware support able to dynamically detect sharing patterns in the running workload, and to identify the potential consumers for a cache line. The mechanism extends the existing CC protocol by performing speculative updates to the potential consumers in order to reduce cache misses. This work shares same of the ideas of our home-forwarding technique. However, in our approach the decision to home-forward a cache line is taken by the runtime developer and not dynamically by the hardware interpreter. This simplifies the hardware design because no special unit is needed to detect data sharing through a deep analysis of the access pattern. It is worth noting that our approach does not make the life more complicated to the high-level programmer of parallel applications, because the use of home-forwarding optimizations is bounded into the runtime support. Furthermore, home-forward write operations also imply self-invalidation, which further reduces the number of invalidation messages among PEs as already studied in Ref. [29]. Other papers like the one in Ref. [30] share analogous ideas with Ref. [11].

## 4 Cache Coherence on CMPs

In this section we recall the CC problem, which is a fundamental and well studied issue in multiprocessors and CMPs. The ensuing discussion will introduce our terminology that will be used to reason about CC and its impact on parallel programs. Then, we will show the cost of CC through synthetic benchmarks.

### 4.1 Preliminaries and abstract model

As it is known, the CC problem [31, 32] derives from the need to maintain shared data consistent in the presence of caching. In modern architectures this goal is achieved by means of *cache coherence protocols* [32] invisible to the programmer, which are in charge of automatically maintaining copies of the same data coherent.

The existing solutions exploit *update-based* or *invalidation-based* protocols [31]. In the first, the store interpreter updates all the copies of the same cache line allocated in one or more caches. In the second, once a PE modifies a cache line in its local cache, the store interpreter is responsible to atomically invalidate all the other copies. In the sequel we will refer to the invalidation-based mechanism, which is mostly used in CMPs [32]. For the sake of simplicity, in this section we will suppose a hierarchy with main memory and L1/L2 caches private per PE.

CC needs a central locus of control, i.e. a *global knowledge* of the allocation state (e.g., modified, shared) of all the cache lines, which must be always updated and available to any PE. Fig. 1 shows a representation in which an abstract agent, called *Global Controller* (GC), is in charge of maintaining this *global state knowledge* (GSK). The figure shows that GC is able to logically communicate with all PEs, and that it is connected to main memory too. Also an abstract *cache-to-cache interconnect* (C2C) is shown.

In the case of a store instruction, the interpreter invokes GC, which atomically updates the GSK, determines the set of PEs that maintain a copy of the cache line, and sends an invalidation to them. Analogous is the case of a cache-miss load: GC updates the GSK to record that a new copy exists, and delegates a $PE_j$ (currently possessing a valid copy) to transfer it to the requesting PE though C2C, or (at least logically) it is transferred from the main memory under the GC control. This model can be optimized by maintaining a *local state knowledge* (LSK) in the cache units that act as *local controllers* (see Fig. 1). If LSK contains reliable information, it can be used to reduce the GC traffic and parallelize the CC actions through direct C2C communications.

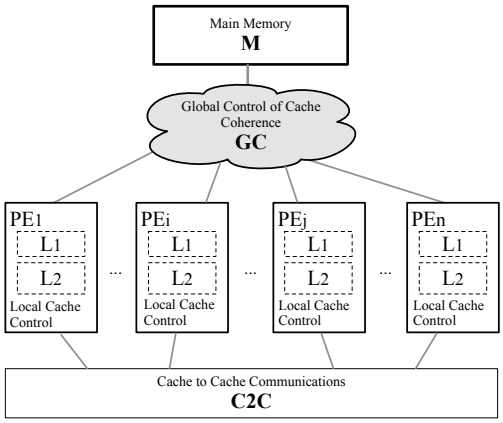



Fig. 1: Abstract model of cache coherence in a CMP.

Two different approaches are used to distribute the centralization point (GSK):

- GSK is atomically *replicated*, i.e. all LSKs are copies of GSK, as far as local cache lines are concerned. For example, every state change is atomically visible to all PEs (through broadcast/multicast communications);
- GSK is *partitioned*, notably different partitions correspond to distinct subsets of cache lines, and each partition has a distinct controller. In order to update the local LSK, a PE goes and asks the local controllers through proper point-to-point C2C communications.

The first solution characterizes *snoopy-based* systems [18, 32], which are suited for low parallelism degrees. The second characterizes *directory-based* architectures with high number of PEs.

In directory-based protocols the GSK is maintained explicitly in a data structure (*directory*), where each entry stores the state of a cache line. In particular, given a cache line $x$, we identify the following entities:

- ***home node***: it is the PE in whose main memory the cache line is allocated. If the architecture is not strictly NUMA, anyway it is the PE in charge of controlling a partition of cache lines that contains $x$. This PE is able to serve the requests for $x$ directly via C2C, when the line is present in its local cache, or by forwarding the request to the main memory if $x$ is not modified in another PE;
- ***requestor node***: the PE that issues an operation request for that line;
- ***owner node***: the PE that currently holds the valid copy of the cache line.

In the case of a cache-miss load, a request-reply interaction occurs between requestor and home. If the cache line $x$ is not modified in any other PE, the reply transfers $x$ from the home to the requestor (the home can transfer it from one of its local caches, if $x$ is present, or from the main memory). Otherwise, if $x$ is modified in another node the request is forwarded to the owner and $x$ is transferred to the requestor[2]. In parallel, the owner informs the home of a new copy.

Similarly, in the case of a store instruction a request-reply interaction occurs between requestor and home. In some schemes the request can convey the data word to be written. In that case the home node can update the main memory immediately or later on or; in some architectures [16, 33] the word is written in the local cache of the home. In all these cases, if copies of $x$ are present in other caches they must be invalidated. The home node sends invalidations to all such PEs, which reply with a sort of acknowledgment. The reply comes to the requestor once all the acknowledgments have been received by the home node. This reply is semantically needed for the sake of memory ordering in Total Store Ordering (TSO) machines [34]. In Weak Memory Ordering ones (WMO) this reply can be waited by a memory fence instruction (or by synchronous stores).

Fig. 2 shows an exemplification of this behavior in the Tilera TILEPro64 architecture, see Sect. 7 for a detailed description of this CMPs. Each core has a L2 cache that maintains directory information. A peculiarity of this architecture is the write-through C2C interaction between the requestor and the home node, i.e. at each store execution, the home node receives the word written by the requestor node and updates its copy in L2.

It is worth noting that today's shared-memory machines are often composed of multiple interconnected CMPs. A typical solution is to provide a hierarchical CC protocol:

- CMP internal caches are kept coherent by an *inner protocol*. In the case of low-parallelism CMPs ($6 \div 12$ cores), the inner protocol is usually snoopy-based;
- between CMPs the *outer CC protocol* is directory-based.

This hybrid approach is currently adopted in the composition of general-purpose low-parallelism CMPs. The abstract

[2]If more PEs have a valid copy, one is selected for the transferring (the *forward* state in the Intel `MESIF` protocol has this goal).

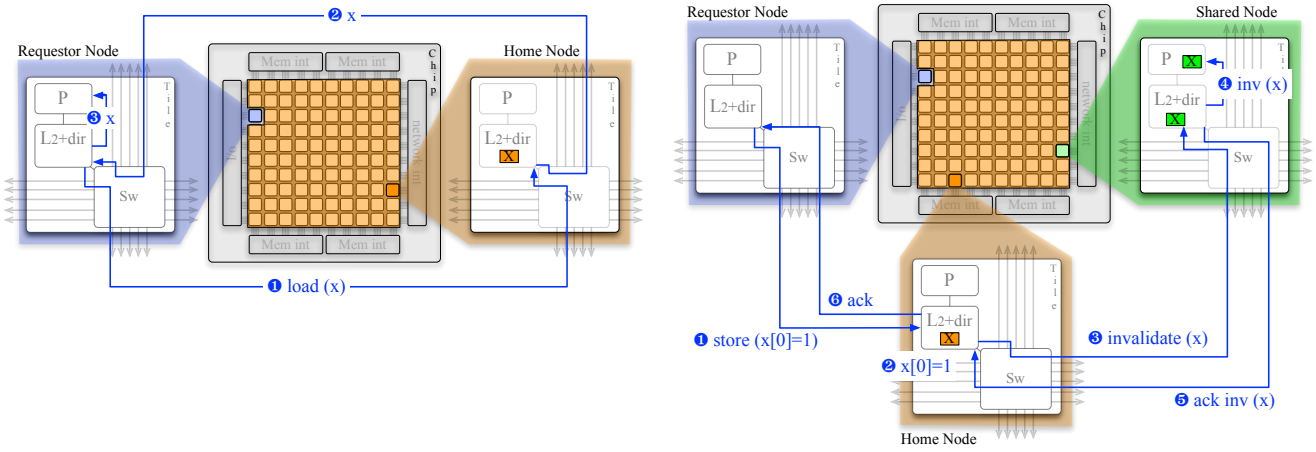



Fig. 2: Cache coherence interactions in the Tilera TILEPro64 CMP: (**left**) interactions between home and requestor nodes in the interpreter of a cache-miss load; (**right**) interactions between home, shared and requestor nodes in the interpreter of a store instruction.

model of this section can be easily applied to the directory-based outer protocol by using the term PE to identify a whole CMP (in the case of an intra-CMP shared L3, it maintains the directory of the cache lines present in the whole chip).

### 4.2 Benchmarking the CC overhead

Write and read operations have a different cost based on the actions taken by the interpreter, which in turn depend of the state of the cache line. This has been previously studied in Ref. [21] with specific benchmarks on the Intel Nehalem and AMD Shanghai CMPs. In this section we show the results of the same benchmarks on two different CMPs: the more recent Intel Sandy Bridge and the Tilera TILEPro64.

The Intel architecture is a machine equipped with two identical Intel Xeon Sandy Bridge E5-2650 CPUs for a total of 16 cores working at 2GHz. The two CPUs are interconnected through the QuickPath Interconnect (`QPI`). Each core is equipped with a private L1d cache of 32KB and a private L2 cache of 256KB. Each CPU has a L3 cache of 20MB. The CC protocol is based on a variant of the `MESI` protocol with the *forward* state [20]. The Tilera CMP, shown in Fig. 2, consists in 64 cores with private L1/L2 caches interconnected through a 2D mesh network. A detailed description of this CMP will be provided in Sect. 7.

***Read latency.*** This benchmark follows the same setting described in Ref. [21] and consists in a program written in the `C` language with parts written directly in assembler. The assembler parts are used mainly for fine-grained latency measurements by reading the *Time Stamp Counter* (TSC) register. In order to have reliable results, we turned off the power-saving facility by fixing the operating frequency to the maximum one of 2 GHz. Since the TSC registers of different PEs might be not aligned, we run each thread by pinning it exclusively on a dedicated PE (core).

A separate consideration must be made for the Hyper-threading feature provided by the Intel Sandy Bridge. In fact, though Hyperthreading is potentially able to reduce the read latency if the cache lines have been already loaded by another thread executed on a SMT context within the same core, in general such cache sharing can be detrimental if the working sets of the threads are large and their workload heavily exploits the processor's hardware resources (e.g., function units and other pipeline resources). Since the benefit of Hyperthreading depends on the cache utilization and computational properties of the executed workloads, in this analysis we decided to disable it.

Finally, due to the out-of-ordering execution model of the Intel PEs we used proper assembler instructions before the measurements in order to avoid executing the `RDTSC` instruction (to read the TSC register) earlier or later than the benchmark expects, and we disabled the hardware prefetches in order to have a full control of the loaded cache lines.

Prior to the measurements, data are loaded in the caches in a desired state from the CC perspective. To this end, our benchmark works as follows:

- a cache line is placed into the *modified* state by reading a cache line and then by modifying it (possibly invalidating the copies of the other threads);
- a cache line is placed into the *exclusive* state in the following way: a thread modifies a cache line which is invalidated in the other caches (if present); then the same thread invalidates its local copy (through a `clflush` instruction) and then it loads the same cache line again;
- a cache line is placed into the *shared* state as follows: a thread places a cache line in the exclusive state and then another thread loads it.

We configure the benchmark in order to place the data in certain cache levels. To reduce the effect of the TLB translation we use huge pages. Further details can be found in Ref. [21]. Tab. 1 shows the results where the first column identifies the source of the read operation. For example: the read latency from a core asking a line to a destination core in the same chip is about 108 clock cycles if the cache line is in state modified in the L1 of the destination core.

The latency to access local caches is independent from the CC state. Intra-/inter-chip requests depend instead on the

| Source | State | L1 | L2 | L3 | M |
|---|---|---|---|---|---|
| Local | M/E/S | 6 | 12 | 39 | 185 |
| Same Chip | Modified | 108 | 102 | | |
| | Exclusive | 90 | | | |
| | Shared | | | | |
| Other Chip | Modified | 304-312 | | | 280 |
| | Exclusive | 195-205 | | | |
| | Shared | 195 | | | |

Table 1: Memory read latency on the Intel machine (Intel Sandy Bridge) depending on the cache line state. Times in clock cycles.

state of the line. The access to a cache line in the shared state costs slightly more than reading from a local cache; modified and exclusive states significantly increase the read latency. The read latency through the `QPI` causes a latency increase between $2\times$ and $4\times$.

Tab. 2 shows the results of the TILEPro64 CMP, where we used a similar setting for the benchmark, i.e., a 64-bit free running cycle counter equivalent to the TSC of Intel is available to collect fine-grained time measurements [16].

| Source | L1 | L2 | M |
|---|---|---|---|
| Local | 2 | 8 | 120 |
| Incoherent | | | |
| Coherent | 40-70 | | 160-204 |
| HOME | | | |

Table 2: Memory read latency on the Tilera TILEPro64 architecture. Times in clock cycles.

In terms of cache coherence states, on the Tilera CMP we have a more limited set of possibilities. Owing to the fact that in this architecture the copy of a cache line in the home node is always updated, the state of a line can be either shared or invalid. Another interesting comparison is between having or not CC enabled. In fact, on TILEPro64 the CC can be disabled for certain memory pages, and this can be configured by the programmer through the use of special functions of the Tilera Multicore Library [16].

As we can see, disabling CC has a cost. With CC enabled, if a cache has the requested cache line, we pay from 40 to 70 clock cycles depending on the distance on the mesh between the source and the destination core of the request. Since this architecture is *distance-sensitive*, the benchmark has been repeated several times with different thread allocations, and the table reports the range of values or only the average value if the standard deviation of the measurements is small. Without CC, we must always go to the main memory, which costs 120 clock cycles on average.

***Write latency.*** We adapt the benchmark to write operations, which have not been studied in Ref. [21]. We measure the latency of a store instruction with *synchronous* semantics, i.e. the one needed for synchronization mechanisms as it will be described in Sect. 6. On the Tilera (which uses the WMO model) this has been achieved using explicit memory fence instructions after the stores. Tabs. 3 and 4 show the results for the memory write latencies respectively on the Intel Sandy Bridge and on the Tilera TILEPro64 CMP.

| Source | State | L1 | L2 | L3 | M |
|---|---|---|---|---|---|
| Local | Modified | 9 | 15 | 39 | 185 |
| | Exclusive | | | | |
| | Shared | 80 | | 83 | |
| Same Chip | Modified | 102 | 97 | 44 | |
| | Exclusive | 86 | | | |
| | Shared | 83-95 | | | |
| Other Chip | Modified | 228 | 225 | 190 | 280 |
| | Exclusive | 213-215 | | | |
| | Shared | 233-285 | | | |

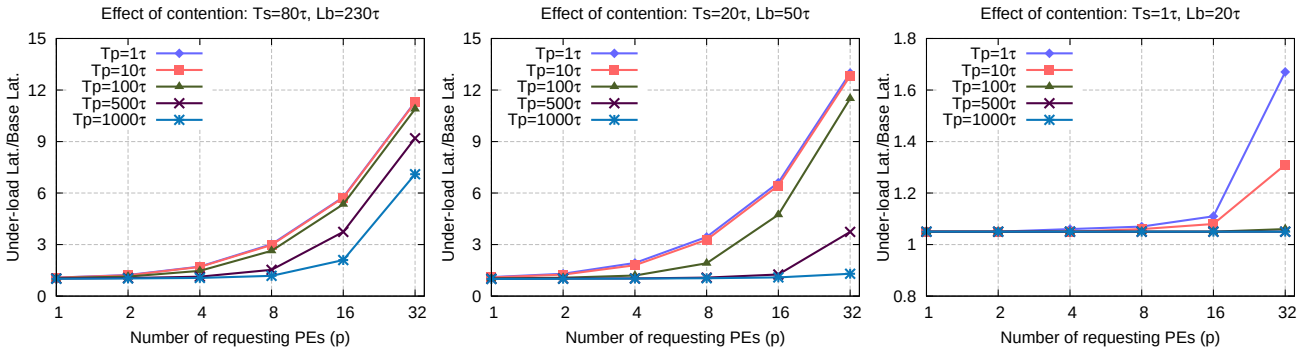
Table 3: Memory write latency on the Intel machine (Intel Sandy Bridge) depending on the cache line state. Times in clock cycles.

| Source | | L1 | L2 | M | |
|---|---|---|---|---|---|
| | | | | wRead | w/oRead |
| Local | Exclusive | 5-28 | | 174-219 | 57-86 |
| | Shared | 12-292 | | | |
| HOME | Exclusive | 45-73 | | | |
| | Shared | 52-332 | | | |
| Local = HOME | Exclusive | 2 | 7 | 226-274 | 121-147 |
| | Shared | 7-285 | | | |

Table 4: Memory write latency on the Tilera TILEPro64 architecture. Times in clock cycles.

For Intel Sandy Bridge we can make analogous considerations of that made for the read latencies. As we can observe, the write latency in the local private cache by a core (both L1 and L2) is higher if that cache line is in the shared state. In fact, in the shared case several invalidations must be sent and the synchronous semantics forces to wait for all the corresponding acknowledgment messages. In the case of a write operation from a core in the same chip, we can see that the latency is greater if the cache line is in state modified in the destination core because more actions must be executed, i.e. before invalidating the copy in that core the cache line must be transmitted to the source core and then modified (the standard store semantics is *fetch-and-write*).

In the TILEPro64 we have a similar behavior. As an example, a local write operation from a core on a cache line currently in its L1 cache in shared state costs from 12 clock cycles to 292. This difference is due to the number of invalidations that must be transmitted, which depends on the




(a) $T_S = 80\,\tau$, $L_b = 230\,\tau$. (b) $T_S = 20\,\tau$, $L_b = 50\,\tau$. (c) $T_S = 1\,\tau$, $L_b = 20\,\tau$.

Fig. 3: Effect of contention on a shared server receiving requests from $p$ *identical* clients. The system is modeled as a `M/D/1` queuing system. Cases with different server service time $T_S$ and base latency $L_b$ values expressed in clock cycles $\tau$.

number of cores that have a copy of the same line. Therefore, we repeated the experiments several times with different threads having a copy of the same cache line. The intervals in the TILEPro64 architecture are wider than in the Intel Sandy Bridge because the Tilera machine has more cores (64) interconnected in a distance-sensitive network (a bidimensional mesh).

In conclusion, these benchmarks confirm that the latency of read and synchronous write operations is dependent on the state of the cache line, and on the presence of other PEs having a copy of it in their private caches.

## 5 Goals and Requirements of an Efficient Runtime

From the previous section, we know that basic operations have different latencies according to the specific actions taken by the automatic CC mechanisms. Moreover, the effective latencies experienced by a parallel program may result higher than the ones measured by the benchmarks. This is due to the *contention* in using shared physical resources (e.g., memory and caches), which is one of the primary sources of performance degradation in shared-memory architectures [35]. The *base latency* of read and write operations is measured without congestion, and depends only on the architectural characteristics (e.g., which type of on-chip and off-chip interconnection networks are available). The *under-load latency* instead, includes the congestion delay in the network and in the shared memory modules/cache units, and depends both on architectural characteristics, and on the way in which the architecture is used by the parallel program.

In order to exemplify the concepts, suppose to have a memory module shared between the PEs of a CMP. Conceptually, the memory module is modeled as a *server*, which receives requests from the PEs and produces corresponding replies. Let $p$ be the *contention parameter*, i.e. average number of PEs making requests to the same server, and $T_p$ be the average time interval between two successive requests. The base latency comprises the time spent by a request to reach the server (including all the network links), the service latency to process the request, and the time spent to forward the reply. The under-load latency is the average *response time* of the server, denoted by $R_Q$. Approximatively, it is the base latency *plus* the average waiting time to serve the incoming requests, which in turn depends on the server's utilization. The figures show the ratio between the under-load latency and the base latency as a function of the number of PEs accessing the server. The results are obtained using the analytical model of a `M/D/1` queuing system [36], with exponentially distributed clients and a deterministic distribution of the server service time. This model describes a scenario in which several PEs independently generate requests to the same server which has a constant service time (this approximates the behavior of memories and caches). Distinct curves are depicted for different values of the $T_p$ parameter. Fig. 3 show the ratio between the under-load latency and the base latency for different combinations of the server service time $T_S$ and the base latency $L_b$.

For compute-intensive workload (i.e. with high $T_p$ values) the impact of contention tends to become negligible. In contrast, the impact is substantial for medium-grained and fine-grained computations because the utilization of the server increases. Our goal is to use the architecture in such a way that the under-load latency is very close to the base one (ratio near to 1), otherwise the efficiency and scalability of parallel programs become relatively low despite a large number of used PEs. Although the goal of this preliminary analysis is not to quantitatively model the performance of CMP, it gives us a reasonable insight into a first design principle of a scalable run-time support for parallel programs:

**Observation 1.** *Reducing the number of PEs making requests to the same server is a way to reduce contention.*

It is worth noting that contention can be alternatively reduced by increasing the $T_p$ parameter, i.e. by making less frequently requests to the same server. This aspect is the rationale followed by several optimization techniques of sequential programs that may also produce better scalability in parallel programming. These techniques are aimed at improving the exploitation of the cache hierarchy (e.g., tiling, blocking, polyhedral approaches to loop optimizations [37]). However, the idea to increase $T_p$ is strongly dependent on

the specific features of the algorithm, while the idea to reduce the contention parameter is of more general application provided that: *i)* parallel programs are expressed as instantiation of well-known parallel patterns; *ii)* the run-time support is designed with policies and mechanisms suited to handle contention by reducing the contention parameter. In this paper we follow exactly this idea.

Another observation is that the previous analysis can be generalized to any kind of server. In cache coherent CMP-based architectures, *local caches act as servers of CC operations*, e.g., cache-to-cache transfers and invalidation (or update) communications generated by the other cache units on-chip or off-chip. This consideration is important from the performance viewpoint: *automatic CC protocols may be a potential source of performance degradation since they contribute to increase contention in the cache units*. Just to exemplify, let us consider the example depicted in Fig. 4. The figure shows that case of a *farm* parallel pattern in which an *emitter* thread (`E`) distributes tasks (e.g., a data structure on which a computation must be applied) to a set of identical *worker* threads (`W`). Each task is assigned to an available worker.

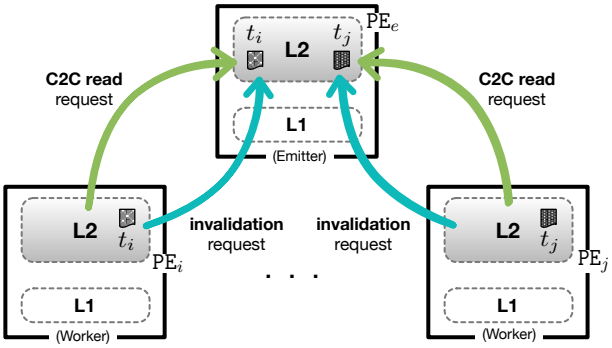



Fig. 4: Task scheduling in the *farm* parallel pattern. Each request has a corresponding reply not shown in the figure.

Suppose for simplicity a cache hierarchy in which each PE has private L1 and L2 caches (no shared caches), and that the home node of the cache lines of a task $t_i$ is the PE (PE$_i$) of the destination worker W$_i$. The figure shows the C2C requests in the case of a system with CC based on invalidation. Once the worker is notified of the presence of the task, e.g., through an asymmetric notification from the emitter, PE$_i$ generates several C2C read requests (once per cache line of the task) to the L2 cache of the emitter PE. In fact, as the emitter has prepared the task before the notification, the cache lines of the tasks are likely in its L2 cache. Moreover, if the task data structure is modified (e.g., during the worker computation), several invalidations are transmitted to the L2 cache of the emitter, and this happens in parallel for all the workers of the parallel program. The result is that the number of cache units making requests to the emitter L2 cache is proportional to the parallelism degree (number of workers) and this may be a source of congestion. For this reason, we can derive a second important observation:

**Observation 2.** *The shared data must be managed in such a way as to generate loosely-coupled interactions between caches, i.e. by making the contention parameter independent from the parallelism degree of the parallel program.*

This observation allows us to clarify an important and often misleading aspect of parallel programming:

- *process/thread mapping* consists in pinning them onto the PEs in such a way as to reduce the base latency. This affect the performance if the architecture is highly distance-sensitive. In general, the base latency is relatively low inside a low-medium parallelism CMP, while the network topology becomes important for highly parallel CMPs with distance-sensitive topologies;
- once a process/thread mapping has been chosen, *shared data mapping* has a relevant impact on under-load latency minimization.

In other words, at first glance it seems that the topological mapping of parallel programs is a critical issue, at least for distance-sensitive networks; however, shared data mapping is often much more relevant than process/thread mapping.

### 5.1 What do we need?

In this part we list the ingredients for a run-time support that minimizes the contention aspects discussed previously. A preliminary aspect is the placement of the home node. In general, depending on the architecture, it may or not be possible to choose the home node. Typically, to balance the workload among caches, systems adopt different policies:

- a *round-robin* home node placement, in which the home node of a cache line is chosen is a round-robin way, e.g., the selection is performed by a modulo operation on the low order bits of the physical address;
- a *hash-based* home node placement, in which the home node identifier is the result of a hash function applied on the starting physical address of the line. This approach is the default option on the TILEPro64 CMP [16].

According to the access pattern to a shared data structure by concurrent threads, it could be convenient to establish precisely the home node for all its cache lines. This feature is important for our approach, because allows us to configure the directory-based CC protocol in such a way as to be tailored for a specific and recurrent use case. Therefore, we identify a first requirement:

**Requirement 1.** *The architecture should allow the runtime system developer to manually configure the home node placement for the runtime data structures.*

The home node selection can be provided with different granularities, e.g., based on single cache lines or at the page granularity. As an example, the TILEPro64 allows the programmer to disable the default hash-based placement by allowing to statically fix the home node at the granularity of memory pages. The Tilera Multicore Library offers several malloc-like primitives to dynamically allocate heap memory,

and the user can choose the home node for all the cache lines of the allocated area.

A more general problem is the global synchronization implied by invalidation. In general, write operations are asynchronous, i.e. they are just launched by the processor without waiting the termination of their interpreter. However, synchronous write operations are needed in order to correctly perform thread synchronization. In Sect. 2 we hinted the case of *asymmetric synchronization* between threads, where a thread $A$ modifies a shared data structure $s$ and notifies a thread $B$. After receiving the notification, $B$ can use the data structure $s$ by possibly modifying it. For the sake of correctness, all the write operations performed by $A$ on $s$ must be visible to $B$ after the notification, i.e. they must be synchronous (in TSO) or alternatively a memory fence must be used (in WMO). From Sect. 4.2, we know that the latency of a synchronous store grows proportionally with respect to the number of invalidations, i.e. the number of copies. From this we derive a second requirement:

**Requirement 2.** *The runtime should be designed in such a way as to minimize the number of current copies of the same cache line.*

This requirement helps but does not solve the problem of contention raised in Observation 1. Fig. 4 shows that C2C read and invalidation requests from workers PEs to the emitter PE introduce contention even if each task is shared between the emitter thread and the assigned worker only. The number of entities making C2C requests to the emitter L2 cache is proportional to the number of workers, which is an undesirable property as stated in Observation 2.

To solve this problem a proper architectural mechanism must be provided at the hardware level. The idea is to introduce the possibility to use a different semantics for so-called *non-local* store instructions, i.e. a store that accesses a cache line $x$ whose home node is distinct from the PE executing the instruction. This is described by the following definition:

**Definition 5.1** (Home-forward semantics). The interpreter of a non-local store with the *home-forward* semantics executes the following sequence of actions:

1. as in any store semantics, the home node is informed of the store in order to update its LSK. In addition, the whole cache line modified by the store is transmitted (*forwarded*) to the home node. The line is allocated (or updated if already present) in the L2 cache of the home node and, possibly, in L1 too;
2. the referred cache line is *self invalidated* (de-allocated) from the requestor node caches (both L1 and L2). This action is performed locally by the requestor node;
3. (optionally) the home node may also be responsible to update the copy in main memory asynchronously.

This hardware mechanism must be visible at the assembler level through special synchronous store instructions or rather by a dedicated instruction `hfwd addr`, which performs the home-forwarding of the cache line of the logical address *addr* (e.g., passed through a register).

This technique has several advantages that contribute to reduce contention on caches. The first is that it allows to eliminate some invalidation communications between caches by reducing contention on non-home nodes. Going back to the example of Fig. 4, let us suppose that the store instructions performed by the emitter thread on the task $t_i$ have the home-forward semantics. If, once received, task $t_i$ is modified by the worker PE (which is the home node), no invalidation is performed because the cache lines of $t_i$ are no more valid in the local caches of the emitter PE (they have already been invalidated). It is worth noting that this advantage would be achieved also with only a self-invalidation mechanism performed by the emitter thread. However, in that case the successive load instructions performed by the worker thread would require to transfer the cache lines of $t_i$ from the main memory. This is the reason for point *1)* in Definition 5.1: the home-forward semantics generates a C2C transfer of the cache line from the requestor cache to the L2 home cache, and hopefully into home L1 too depending on the size of $t_i$. In this way the worker PE already finds the needed cache lines of the task at least in its L2. Fig. 5 shows the C2C requests in the case of home-forward write operations performed by the emitter thread.

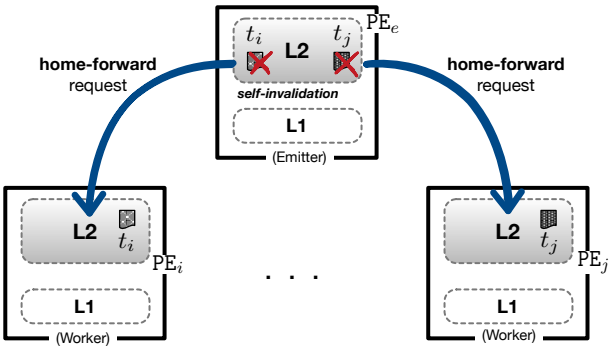



Fig. 5: Task scheduling in the *farm* parallel pattern: C2C requests in the case of invalidation-based CC with *home-forwarding*. Each request has a corresponding reply not shown in the figure.

The home-forward optimization minimizes contention (on both the home and requestor node): as we can see the only servers are the local caches of the workers PEs that receive C2C write requests. Therefore, the number of entities making requests to the same cache is minimized according to Observation 1 and this value is constant and does not depend on the parallelism degree of the computation (Observation 2). Furthermore, another important advantage is that when data are "*of interest of the thread executed on the home node itself*", this optimization reduces the base latency of read operations too. In fact, the thread executed on the home node PE can directly read the cache lines of the task directly from its L2 (or better L1) cache because it is also the owner of them. In this way the base read latency of those lines includes only local interactions between the processor and its L1/L2. As studied in Sect. 4.2, this is the ideal situation, with a latency of few clock cycles, see Tabs. 1 and 2.

Although simple, the scenario depicted in Figs. 4 and 5 is recurrent in parallel patterns as hinted in Section 2. This

result can be summarized in a last requirement:

**Requirement 3.** *In order to avoid making several cache-to-cache requests (e.g, read requests, invalidations) to the same cache, the architecture must provide mechanisms for single cache line forwarding and self-invalidation.*

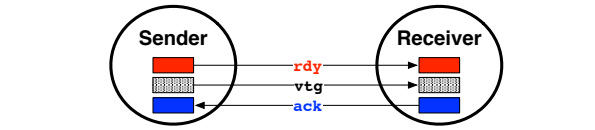
These principles will be extensively applied in the next section to the design of an efficient runtime.

## 6 Run-time Support Implementation

In this section we describe the design of a run-time support that matches the requirements discussed in the previous section. Our approach is inherently *lock-free*, and entirely based on asymmetric synchronization mechanisms. Furthermore, we will made the assumption that the underlying physical system is *dedicated* to host the execution of a single parallel program, i.e. each process/thread is mapped *exclusively* on a single PE. This assumption, often precondition to efficiently use lock-free mechanisms based on busy waiting, has justification in mission-critical applications with very strict high-throughput and low-latency requirements. In the future, we plan to study the applicability of the approach to more general multi-programmed/multi-tasked environments.

### 6.1 The base mechanism

The base mechanism of the run-time support is the so-called *rdy-ack communication interface*, described for the first time in Ref. [15] and used to synchronize and exchange messages between threads like in Ref. [38]. The mechanism provides a point-to-point communication between two partners, *sender* (`S`) and *receiver* (`R`), with a buffer of one position. The idea is depicted in Fig. 6.




```
send (msg) {
    wait until ack is true
    copy msg into vtg
    reset ack
    signal rdy to the receiver
}
```

```
receive (data) {
    wait until rdy is true
    copy vtg into data
    reset rdy
    signal ack to the sender
}
```

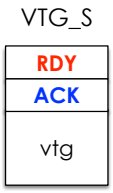
Fig. 6: Basic mechanism of the run-time support: rdy-ack communication interface between two partners.

The pseudocode of the *send* and *receive* primitives uses two boolean events:

- the *ready* event (`rdy`) is true when a new message is present in the *vtg* variable. It is false otherwise;
- the *ack* event (`ack`) is true when the last transmitted message has been received and copied into a private variable *data* by the receiver.

With the *signal* operation, the corresponding event is set to true, while the *reset* one sets the event to false. For the sake of correctness, the rdy and the ack events are initialized to false and true respectively.

Any possible implementation of this mechanism implements the *vtg* data structure as a shared variable between the sender and the receiver threads. Different possibilities can be adopted to implement the two events *rdy* and *ack*. A first solution consists in implementing them as shared *boolean flags*. We define a data structure `VTG_S` as composed of three fields: two boolean flags for the two events, and the *vtg* variable as shown in Fig. 7. The figure reports the `C`-like pseudocode, where the waiting of an event is implemented by a while-loop statement on the corresponding flag. The `VTG_S`, as well as all the other data structures of a communication interface that will be described in the sequel, are properly padded and aligned in order to avoid false sharing.




```
1  send (vtg_s, msg) {
2   while(vtg_s->ack == 0);
3   <copy msg into vtg_s->vtg>
4   vtg_s->ack = 0;
5   vtg_s->rdy = 1;
6  }
```

```
1  receive (vtg_s, data) {
2   while(vtg_s->rdy == 0);
3   <copy vtg_s->vtg into data>
4   vtg_s->rdy = 0;
5   vtg_s->ack = 1;
6  }
```

Fig. 7: Implementation of the *rdy-ack communication interface* through shared boolean flags for the synchronization events.

To prove the correctness, let us consider an abstract multi-processor architecture $\mathcal{M}$ respecting the Sequential Consistency memory model [34]. The following proposition holds:

**Proposition 6.1** (Rdy-ack correctness). *The send and receive algorithms executed on $\mathcal{M}$ implement a lock-free single-producer single-consumer buffer of one position.*

*Proof.* Initially $(rdy, ack) = (0,1)$, i.e. the sender can proceed by executing the *send* while the receiver is eventually waiting on line 2 of the *receive*. The sender copies *msg* into the *vtg* variable and sets the flags such that $(0,1) \rightarrow (1,0)$. Now the receiver can execute the *receive* primitive. It reads *vtg* and copies it into *data*, and sets the flags such that $(1,0) \rightarrow (0,1)$ going back to the initial condition. It is worth noting that row 3 in the *send* must be executed after *ack* is set to 1 (otherwise a new message can overwrite a previous and possibly unreceived message), and row 5 after row 3 (the *rdy* must be set to 1 after the store of the message in *msg* is visible to the receiver). Similarly, row 3 in the *receive* must be executed if and only if *rdy* is equal to 1, and row 5 after row 3 (saving the message in the private variable before it can be overwritten by the sender). □

In the case of an architecture adopting the WMO memory model, we need to enforce memory ordering of the operations in the send and receive primitives by adding proper memory fence instructions, as explained in Ref. [15].

Furthermore, to avoid the additional copy from *vtg* into *data*, an alternative code for the receive primitive can be provided. The only difference with the one in Fig. 7 is that in the receive algorithm we do not have the copy statement at row 3 and the last statement at row 5. A special *set_ack* primitive is provided to set the ack flag to true, and it is used when the destination thread has terminated to utilize *vtg*, and it content can be safely overwritten with a new message.

### 6.2 Generalizations and optimizations

The base mechanism can be extended to communications with more than one buffer position. Let $k \geq 1$ be the maximum number of messages that a sender can transmit without waiting for the first sent message being received. This communication can be achieved with $k$ instances of `VTG_S` used in a *round-robin* fashion by the sender and the receiver. To this end, each partner has a *private* array `VTG_V` of pointers to the $k$ `VTG_S` instances and a corresponding *private* integer *index* initialized to zero. Fig. 8 shows a representation of this implementation scheme.

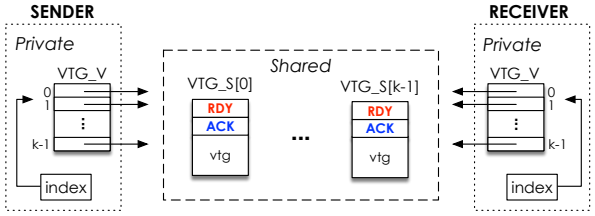



Fig. 8: Generalization of the *rdy-ack communication interface* with $k \geq 1$ buffer positions.

Each time the sender executes a *send* primitive, the `VTG_S` instance at address `VTG_V[index]` is used and the index is incremented by 1 modulo $k$. Symmetric actions are performed on the receiver's side.

An important alternative implementation consists in using *inter-processor communications* (e.g., I/O interrupts) for the notification of rdy/ack events. The goal is to avoid having shared boolean flags, i.e. the only shared variables between the sender and the receiver are the *vtg* variables in the `VTG_S` instances. Therefore, in this solution the `VTG_S` structure collapses in the *vtg* since shared flags do not exist anymore. In this implementation we must be able to distinguish the synchronization events, i.e. to identify whether inter-processor messages correspond to *rdy* or *ack* events and which is the corresponding communication interface and the target `VTG_S` instance. To this end, each communication interface in the parallel program has a unique identifier. For each thread of the application, a *private* table `TB` is used to obtain from the interface identifier the pointers to *all* the interesting data structures (`VTG_V` and the `VTG_S` instances). Furthermore, two additional data structures are associated with each communication interface:

- an array `EVENT_ACK` that contains the boolean *ack* flags of each `VTG_S` instance. All the flags of this data structure are now *private* of the sender;
- an array `EVENT_RDY` that contains the boolean *rdy* flags of each `VTG_S` instance. Symmetrically, this data structure is *private* of the receiver.

These two arrays have exactly $k$ flags. Each event (e.g., implemented as an I/O interrupt) is a pure synchronization message that conveys the tuple $(\texttt{event}, \texttt{ra\_id}, \texttt{vtg\_id})$, where the first element is the type of the event (*rdy* or *ack*), the second is the unique identifier of the communication interface, and the last is the identifier of the `VTG_S` instance, see Fig. 9. As we will see in Sect. 6.3, in this solution synchronization can be performed very efficiently.

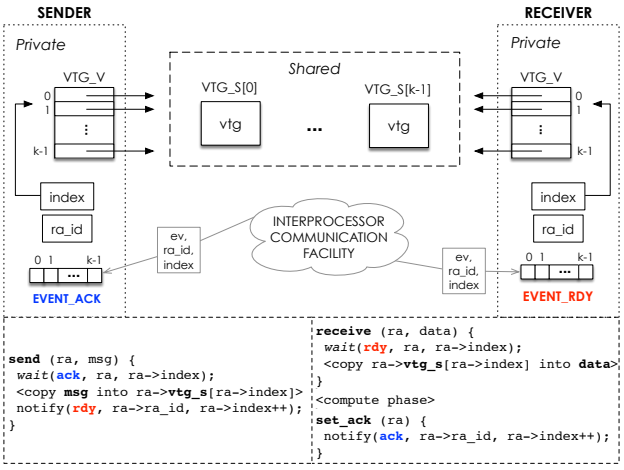



Fig. 9: Implementation of the *rdy-ack communication interface* with inter-processor communications.

The pseudocode uses two primitives *notify* and *wait*. The first performs an inter-processor communication of the tuple $(\texttt{event}, \texttt{ra\_id}, \texttt{vtg\_id})$. This is a wrapper to a proper set of assembler instructions that perform this notification. Different possibilities can be recognized:

- if the inter-processor communication is performed through I/O interrupts, the primitive contains proper instructions to prepare the interrupt message that will be transmitted to a local I/O unit in charge of executing the communication. This is performed by special assembler instructions or through standard store operations according to the memory mapped I/O model;
- alternatively, if some special architectural facilities are available, the notification can use a mechanism completely different from the I/O subsystem. As we will see in Sect. 7, in the Tilera TILEPro64 CMP this can be achieved by exploiting proper user-accessible on-chip networks, which provide a very efficient way to exchange small messages between cores.

Once received, the inter-processor communication must be properly processed by a run-time support interrupt *handler*. The handler inspects the message to determine the identifiers of the corresponding communication interface and of the `VTG_S` instance. The result is that the proper event flag is set to true in either `EVENT_RDY` or `EVENT_ACK` structures according to the event type.

Slightly more complex is the primitive *wait*, whose behavior is depicted in Fig. 10. This primitive checks the corresponding event flag, private of the calling thread. If it is already true, it resets it and returns. Otherwise, it waits for an inter-processor communication using a special assembler instruction (a generic `rcv_ip_comm` in the figure) that moves the calling processor into a sort of "wait for interrupt" mode until an inter-processor communication is received. During this waiting phase, other event notifications can be received, and they must be properly buffered by writing the corresponding flag in the correct data structure associated to the target communication interface.

```
1 wait(evt, ra, idx) {
2  let evt={RDY,ACK} be EV
3   if(ra->EVENT_EV[idx] == 1)
4    ra->EVENT_EV[idx] = 0;
5   else {
6    while(rcv_ip_comm(ev_rcv, ra_id_rcv, idx_rcv)) do
7      if !(evt==ev_rcv & ra->ra_id==ra_id_rcv & idx==idx_rcv)
8        if(ev_rcv==RDY)
9         TB[ra_id_rcv]->EVENT_RDY[idx_rcv] = 1;
10       else TB[ra_id_rcv]->EVENT_ACK[idx_rcv] = 1;
11   }
12 }
```

Fig. 10: Pseudocode of the *wait* synchronization primitive using inter-processor communication.

In the next part we will analyze the impact of this mechanism in terms of CC interactions.

### 6.3 Implications on CC

The first design choice is to properly choose the home node of all the data structures of a rdy-ack communication interface. An interface with home node sender has always a non-home node receiver. The vice-versa is not mandatory, although having the home node in one of the two communication partners allows the design principles exposed in Sect. 5.1 to be applied in order to reduce contention between caches.

***Non-home node runtime.*** Let us start with the case of *rdy* and *ack* events implemented as shared boolean flags (Fig. 8). Consider the phase in which a non-home node sender (receiver) reads (via C2C read requests the first time) the first cache line of the current `VTG_S`, and suppose it finds *ack* (*rdy*) equal to 0. If the sender (receiver) maintains this line in its local caches, the in-cache retry test pattern (while loop in Fig. 7) is performed without generating additional traffic between caches. Although useful for this reason, this scheme has a fundamental drawback: *it increases contention* in non-home local caches due to subsequent invalidations transmitted by the home node receiver (sender) when the event will be set to true. In the case in which the PE is the non-home node sender (receiver) of other communication interfaces, this increases contention because it receives many invalidations (e.g., one per worker PE in the farm case).

A solution to this problem is the following: the sender (receiver) self invalidates from its L1/L2 caches the cache line of the shared flag and the request is repeated (with C2C reading) until the event is true. However, this technique increases the frequency of the requests to the home node cache. This can be alleviated by introducing a *periodic retry* in the while loop, although its configuration requires proper tuning.

This issue can be solved efficiently by relying on inter-processor communications. The presence of an event is notified asynchronously by the partner using an inter-processor communication. The most likely situation is that the event has already been registered in the private data structures `EVENT_RDY` and `EVENT_ACK`. Otherwise, the sender (receiver) PE waits for the inter-processor communication without generating further cache-to-cache requests.

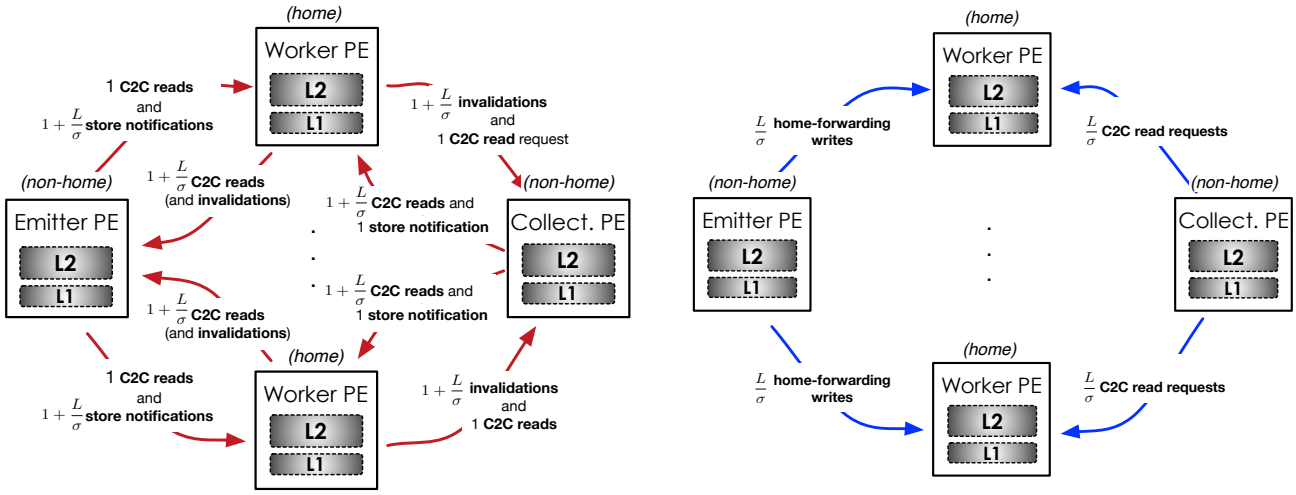
The other actions taken by the run-time support are the following:

- *(1)* in the *send* performed by a non-home node PE, the message is copied into the *vtg*. Each cache line must be *home-forwarded* to the L2 (potentially also L1) of the home node receiver PE. According to Definition 5.1, this causes the self-invalidation from the sender caches;
- *(2)* the write operations on the *vtg* performed by the non-home node sender PE must avoid loading those lines before modifying them. In fact, such lines are always written and never read by the sender. Therefore, a proper store instruction with *no-allocate-on-write* semantics must be provided by the architecture and used;
- *(3)* in the version with shared flags the cache line of the rdy (ack) once modified (rows 4 and 5 in Fig. 7) must be *home-forwarded* to the home node receiver (sender) PE;
- *(4)* once the non-home node receiver PE has terminated to use the *vtg* cache lines, those lines must be self invalidated from its L1/L2 caches in order to avoid further invalidations by the home node sender PE.

***Home node runtime.*** In the version with shared flags, the read operations on the *ack* (*rdy*) shared flag by the home node sender (receiver) are performed locally on the L1/L2 cache. When the flag is modified by the receiver (sender) PE, that cache line is home-forwarded to the sender (receiver) private caches. Therefore, the write operations to reset the flag do not cause invalidations to the non-home node PE receiver (sender) owing to home-forwarding. Furthermore, in the case of the home node sender PE, the store instructions on the message are performed locally without generating invalidations. Similarly, the home node receiver PE can directly use *vtg* whose cache lines have been directly forwarded into its L2 and, hopefully, also in L1.

***Example.*** Let us exemplify the behavior of the run-time support in the case of a *farm* parallel program with an *emitter* (`E`), a set of workers (`Ws`) and a collector (`C`). Let $L$ be the number of cache lines of a message transmitted from `E` to the workers and from the workers to `C`. Let $\sigma$ be the size of the L2 cache line, and suppose each worker PE the home node of the data structures associated with its input/output rdy-ack interfaces from `E` and to `C`.

Fig. 11a depicts the requests between L2 caches. As a first case we suppose the *basic invalidation semantics*, i.e. the run-time support implementation does not exploit home-forward stores and self-invalidations. Events *rdy* and *ack* are implemented as shared boolean flags stored in a single cache




(a) Without home-forwarding and self-invalidation.

(b) With home-forwarding and self-invalidation.

Fig. 11: Example of interactions between caches in the case of a *farm* pattern with an emitter (E), a set of workers (Ws) and a collector (C) thread.

line. The figure shows the number of requests produced per service time (i.e. per task). Each request has a corresponding reply not shown in the figure.

Intuitively, the L2 caches of E and C are heavily stressed. They receives invalidation and C2C read requests from the workers L2 caches. The number of clients that make requests with those L2 caches is equal to the parallelism degree of the program, and this can be a source of high contention. In this scenario the under-load latency can result much more greater than the base latency of read/write operations, and the scalability of the parallel program may be critically hampered.

Fig. 11b shows the same program with the run-time support exploiting the home-forward and self-invalidation technique with inter-processor communications. Here, *the L2 caches of E and C are no longer servers*, i.e. they produce home-forward write and C2C read requests to the workers L2 caches which, in turn, do not generate invalidations nor C2C read requests to them. Local caches are more loosely-coupled and the number of clients making requests with the same cache is limited ($p = 3$ in this example) and independent from the parallelism degree.

The counterpart is that the notification of the *rdy* and *ack* events occurs through inter-processor communications (not shown in the figure). It should be noted that such interactions are *asynchronous*, thus not according to a request-reply behavior as for the synchronous cache-to-cache interactions of CC. This time the critical parameter is not the number of clients accessing the same server, but the *server service time*, i.e. the interrupt handler service time (or the equivalent computation in the wait operation). Provided that it is smaller than the inter-arrival time of notifications, this mechanism does not introduce significant performance degradations. In conclusion, asynchronous notifications do not affect performance on condition that the inter-processor communication facility has low overhead, which is technologically feasible as it will be shown in the next section.

## 7 Experimental Evaluation

Our run-time support can be implemented on architectures on which the home-forwarding mechanism (see Def. 5.1) is available or it can be emulated. On Intel multicores this mechanism cannot be emulated, because a self-invalidation instruction is not directly provided and it is not possible for a core to allocate and write some cache lines into the remote caches of another core. Instead such features can be accurately emulated on the Tilera TilePro64 which represents a concrete architecture where the benefits of our runtime can be verified and assessed for the first time.

The TILEPro64 is a CMP specialized in the fast execution of network workload. Fig. 12 gives a general overview of the CMP. It is equipped with 64 identical processing cores (called tiles) interconnected by an on-chip network named iMesh. Each tile is composed of: *a)* a 3-way VLIW in-order processor running at 866MHz with a single thread context, *b)* a private cache subsystem composed of 16KB L1i, 8KB L1d and 64KB L2 (inclusive), L2 cache line size of 64 bytes, and *c)* a switch for the interconnection with the iMesh network. Four DDR2 memory controllers are placed at the edges of the chip. The TILEPro64 is mounted on a PCI express card of a host machine and it is equipped with on-chip PCIe and network controllers. The architecture supports a Weak Memory Ordering consistency model.

The Tilera CMP is often used as a co-processor for accelerating time-critical kernels, thus the dedicated environment assumption made at the beginning of Sect. 6 is justified. Furthermore, though not able to reach the peak performance of off-the-shelf multicores produced by leading vendors, the TILEPro64 has very peculiar features that make it a valuable candidate for prototyping our methodology. More specifically:

- TILEPro64 allows the programmer to dynamically allocate memory (e.g., using malloc-equivalent calls like

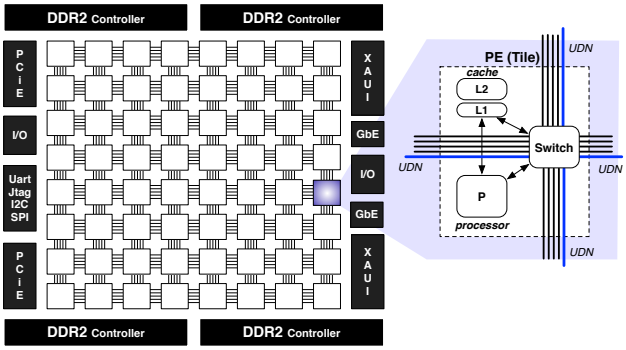



Fig. 12: Tilera TILEPro64 CMP: interconnection networks and structure of a generic PE (*also called tile*).

`tmc_alloc_map`, `tmc_alloc_set_home`) by explicitly choosing the home node PE (Requirement 1) at the granularity of virtual memory pages;

- the interaction between a requestor PE and the home node has the *write-through* semantics, i.e. the data word written by a store instruction is forwarded to the home node PE which, if not present, allocates the corresponding cache line in its L2;
- one of the interesting facility of this CMP is the *User Dynamic Network* (UDN), an on-chip network for the transfer of small messages (up to 128 32-bit words). Each PE has five UDN hardware queues accessed as registers through special assembler instructions. PEs can transmit messages composed of one header word, a tag word and the payload.

We exploit such architectural features in order to provide an implementation of the rdy-ack run-time support with an emulation of the home-forward mechanism (Requirement 3):

- as said before, each word written by a non-home node PE is forwarded to the home node that maintains an updated copy in its L2. This, in conjunction with a subsequent self-invalidation (`inv` instruction) and a memory *fence* allows us to emulate synchronous writes with home-forward semantics. It should be noted that this mechanism is less flexible, because every cache line homed at a remote PE is forwarded, while in general we could desire a finer control;
- store instructions on write-only data structures (e.g., the *vtg* by the sender PE) can be performed using the `wh64` instruction (*write hint*), which hints that the program intends to write every byte of the specified cache line before reading it. Using it, the processor avoids fetching the line from main memory or another cache;
- for the sake of memory ordering, because of the WMO consistency model adopted by Tilera, some *fence* instructions are added in proper points of the rdy-ack runtime. See Ref. [15] for further details.

Finally, the architecture allows us to implement interprocessor communications for notifying the *rdy* and the *ack* events. For that, we exploit the UDN network facility. The *wait* and *notify* primitives used in Sect. 6.2 are implemented as wrapper to *pop*-like/*push*-like assembler instructions on UDN hardware queues of each PEs. UDN messages have a tag field. Each time a new message is received, the tag is inspected and the message is de-multiplexed into the corresponding queue. In the case of a tag-miss, the message is enqueued into the *catch-all* queue and a configurable interrupt is raised [33]. We use this mechanism to implement the optimized rdy-ack communication interface.

### 7.1 Evaluation of the base mechanism

In this section we will evaluate the rdy-ack mechanism on the TILEPro64 CMP. We measure the *communication latency*, i.e. the time from the execution of the *send* primitive to when the destination thread receives the message. To acquire such measurements, we use the same benchmark setting described in Sect. 4.2.

The scheme of the benchmark is a basic ping-pong program with two threads `A` and `B` mapped onto distinct PEs. `A` transmits a minimal (one-word) message to `B` and waits for a reply from it. `B` receives the message and sends back a reply to `A`. The benchmark consists of $10^5$ iterations. We measure the completion time and we divide it by $2 \cdot 10^5$ in order to approximate the communication latency. The experiments have been compiled with the `tile-gcc` cross compiler with the `-O3` optimization flag enabled. The test has been repeated several times by changing the mapping of threads onto the PEs, in order to study the latency with different distances between the two communication partners.

The results are shown in Fig. 13 for three implementations: *i)* `RA` is the implementation with the *rdy/ack* events implemented as shared boolean flags, basic invalidation semantics with default home node selection (hash-based); *ii)* `RA_HF` is the implementation with the emulation of home forwarding (the home node of every rdy-ack interface is the destination PE); *iii)* `RA_UDN` is the implementation with inter-processor communications and home forwarding.

Fig. 13: Communication latency of the *rdy-ack communication interface* with different distances between communication partners.

The experiments show that `RA-HF` outperforms the basic version `RA`. The average improvement is of $50 \div 65\%$. With longer distance between the two threads, we measure

a higher latency in the RA−HF version, while this is not true for RA. The reason is that in the RA version the home nodes of the rdy-ack interface cache lines are chosen by the OS. Therefore, the home PEs can be any third PE with respect to the ones of A and B. The version with UDN outperforms both the variants based on shared flags. On average, the communication latency achieved with RA−UDN is one third of the latency with RA−HF and it slightly increases with the distance. This confirms that the UDN network facility is highly optimized for the transmission of short messages (few data words), as needed by the rdy-ack mechanism. Tab. 5 provides the numerical results and further information about the standard deviation and peak measurements.

| *Implementation* | *Hops* | Comm. Latency | | | |
|---|---|---|---|---|---|
| | | *Avg* | | *Std dev* | *Max* ($\tau$) |
| | | $\tau$ | $\mu$sec | | |
| RA_UDN | 1 | 42.13 | 0.0486 | 0.091 | 42.39 |
| | 8 | 49.63 | 0.0573 | 0.087 | 49.88 |
| | 14 | 55.64 | 0.0642 | 0.080 | 55.84 |
| RA_HF | 1 | 363.55 | 0.4198 | 0.2039 | 363.83 |
| | 8 | 337.01 | 0.3892 | 0.1621 | 337.14 |
| | 14 | 331.06 | 0.3823 | 0.1769 | 331.22 |
| RA | 1 | 124.27 | 0.1435 | 0.0848 | 124.38 |
| | 8 | 150.31 | 0.1736 | 0.1121 | 150.42 |
| | 14 | 168.33 | 0.1934 | 0.1081 | 168.42 |

Table 5: Results of the communication latency of the *rdy-ack communication interface*.

### 7.2 Evaluation on parallel programs

The goal of this final section is to provide a first experimental evaluation of our methodology on parallel programs. In order to emphasize the possible outcome of our optimizations, we target *fine-grained* computations.

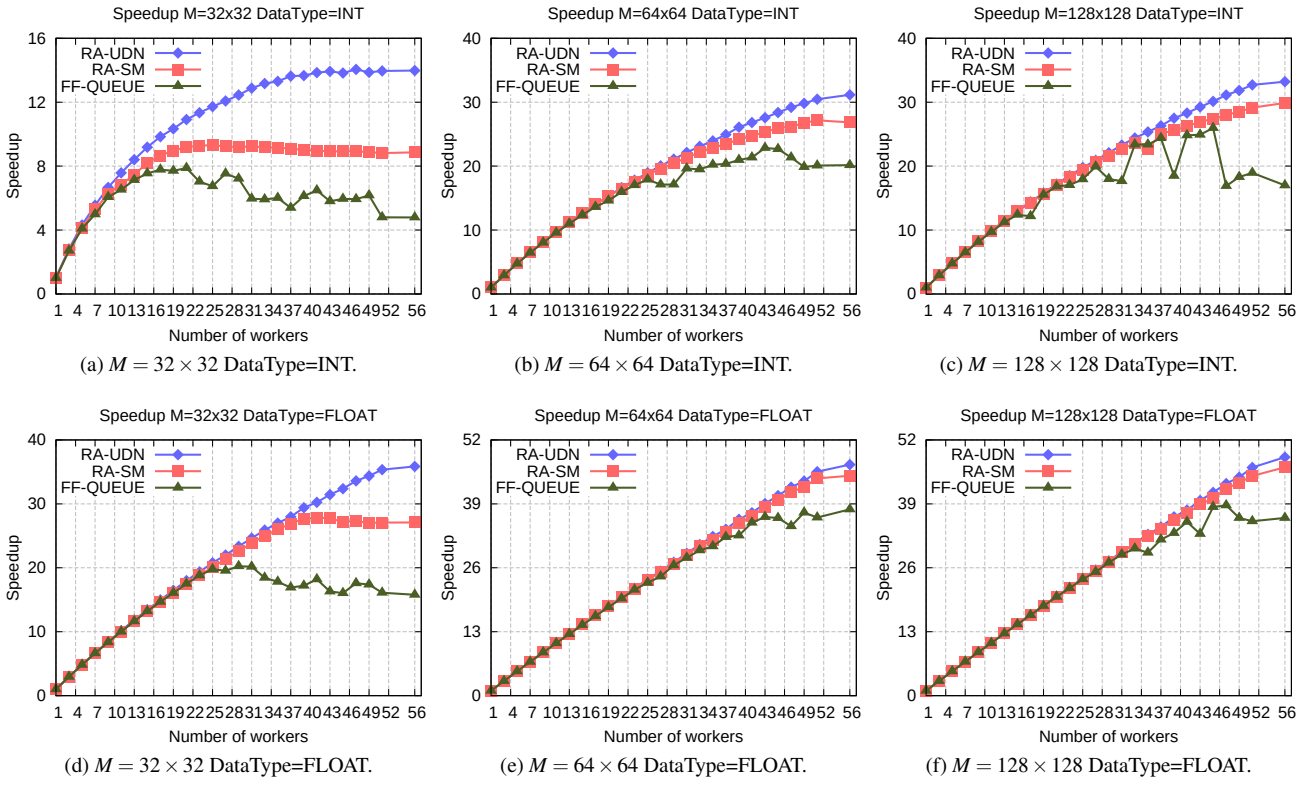
Fine-grained parallelism is characterized by a low computation-to-communication ratio, i.e. cooperation/synchronization primitives between the execution of different tasks are very frequent, owing to the low running times of tasks. A parallel program that executes in parallel tasks with a small sequential computation time is considered fine-grained in our terminology. As a general principle, the more fine-grained the parallel program the more efficient the runtime mechanisms in order to achieve good speedup.

We show two parallel programs operating on large input streams of tasks received from the network. In order to take advantage of our approach, both the benchmarks are characterized by a small computational grain per task whose computation operates on a relatively small working set. We will show the potential of our approach in reducing the service time per tasks (also in cases of very fine-grained functions lasting from tens of microseconds to few milliseconds) which reflects in a substantially lower completion time to process all the tasks belonging to the stream (reduction of several minutes/seconds). In term of parallelism, the first benchmark is based on the *farm* parallel pattern while the second example is parallelized as a *map*. Further studies of other patterns (e.g., stencils, reduce, parallel prefix, data-flow parallelism) will be analyzed in our future works.

***Farm benchmark.*** The program consists in a farm pattern composed of an emitter, a set of workers and a collector as described in the previous sections. The emitter produces a stream of $10^5$ matrices of size $M$. The basic elements of the matrices are integers each one represented by 4 bytes. Each matrix corresponds to a task, transmitted by the emitter (E) to a worker (W) selected according to an on-demand scheduling policy. Each worker operates on each input tasks independently and produces a result data structure which is transmitted and gathered by the collector (C). On each received task, a worker executes a *matrix-convolution* algorithm, typical of difference methods and image filters. The new value of each matrix element is calculated by applying a mathematical transformation on the surrounding elements. In the farm approach each matrix represents an independent task to be computed in parallel, i.e. workers compute different matrices in parallel. Alternatively, the same problem could be parallelized by exploiting parallelism within the same matrix, i.e. all the workers compute the same matrix working in parallel on a subset of its rows. In this case a task is a partition of the matrix. In this part we analyze the case of the farm parallelization, while at the end of this section a data parallel example will be described and studied in detail.

Since we operate on a stream of tasks, we are interested in the *service time* measurement, i.e. the average time interval between the beginning of the executions on two consecutive input tasks (matrices). We measure the relative speedup as the ratio of the average computation time of the sequential algorithm on each input task over the service time of the parallel implementation. The results of this first experiment show that while for large matrices (more than $1,000$ rows) this program achieves near-ideal speedup up to the maximum number of PEs of the architecture, the speedup is relatively low with very small matrices of few tens of rows. This is the scenario that we try to optimize with our run-time support and optimizations. Fig. 14 shows the results of the experiments in which we report the speedup achieved with a different number of workers. The maximum is 56 because two PEs are used by the emitter and the collector threads and the others are reserved to the operating system. In the experiments we pin each thread onto a PE in an exclusive fashion and we repeated each experiment 50 times. The standard deviation of the measurements is low (less than 10% of the average). Therefore, we omit to report the error bars for the sake of representation clarity. Since standard parallel programming frameworks like OpenMP and MPI are not available on Tilera, we compare the execution of the application with the following runtimes:

- the first one is our runtime based on *rdy-ack* interfaces with inter-processor communications over the UDN network. For each worker, the data structures of the E-W and W-C interfaces are homed in the PE executing the worker thread. As studied in Sect. 6.3, this home selection strategy in combination with home-forwarding and




(a) $M = 32 \times 32$ DataType=INT. (b) $M = 64 \times 64$ DataType=INT. (c) $M = 128 \times 128$ DataType=INT.

(d) $M = 32 \times 32$ DataType=FLOAT. (e) $M = 64 \times 64$ DataType=FLOAT. (f) $M = 128 \times 128$ DataType=FLOAT.

Fig. 14: Speedup of the *farm* benchmark with different run-time supports and optimizations on the Tilera TILEPro64 CMP.

self-invalidation is capable of reducing the number of clients making requests to the same caches;

- the second runtime is the one at the previous point except that we use shared flags for inter-thread synchronization (*rdy* and *ack* events);
- the third is a generic runtime that exploits the lock-free shared queues [39] used by the low-level mechanisms of the *FastFlow* parallel programming framework [5][3]. In this model threads cooperate by exchanging memory pointers to shared data structures through push-like and pop-like operations on shared queues. In this runtime the homing of the data structures is left to the OS. We choose this runtime because, as stated by the authors [5], it targets fine-grained parallel computations.

Fig. 14 indicates these implementations with the names `RA-UDN`, `RA-HF` and `FF-QUEUE` respectively. We perform experiments with different matrix sizes ($M = 32 \times 32, 64 \times 64, 128 \times 128$) and we consider two cases in which the elements are integers and single-precision floating-point numbers. The TILEPro64 CMP does not have floating-point units, and decimal operations are emulated by software. Thus, by using floats we can study the effect of coarser-grained computation with the same number of cache misses (floats require the same number of bytes of integers).

In general, the finer the computation the greater the benefit of our approach. In the case of $M = 32 \times 32$ we achieve an improvement of 65% in speedup with `RA-UDN` compared with `FF-QUEUE`, although the speedup is far from being ideal due to the very fine-grained nature of the computation with very small matrix sizes. The use of inter-processor communications through the UDN network makes it possible to greatly improve performance. The gain with respect to using shared flags is of 36%. In fact, the solution with shared flags for the *rdy* and *ack* events requires additional interactions among caches. As stated in Sect. 6.2, the non-home node PE must repeatedly read and self-invalidate the cache line of the flag by generating additional network traffic on the chip. This synchronization is instead efficiently implemented by the UDN, and requires just one message when the notification must be transmitted from the home node.

With larger matrices the difference between `RA-UDN` and `RA-SM` becomes smaller (13% with $M = 64 \times 64$ and 11% with $M = 128 \times 128$). In fact, with larger matrices the impact of the cache-to-cache interactions for the shared flags becomes less important while the advantage in the *vtg* transmission is dominant with respect to `FF-QUEUE`. In our runtime a matrix is directly home-forwarded into the L2 cache of the corresponding worker PE, which can start using it without paying additional cache misses (except for L1-L2 transfers that are local to the PE). This optimization cannot be exploited in the `FF-QUEUE` runtime, which provides a maximum speedup 35% lower than `RA-UDN` for $M = 64 \times 64$ (24% with $M = 128 \times 128$). The problem here is that the destination worker of a task is defined late at runtime (on-demand scheduling), and the homing of the cache lines of the tasks is left to the operating system.

[3] `http://mc-fastflow.sourceforge.net/`

In the case of floats instead of integers, we observe that the difference between runtimes becomes small as soon as matrices of size $M = 64 \times 64$ are used. Therefore, we can conclude that our runtime is mainly effective with *very* fine-grained computations, which is the final goal of our work. Tab. 6 summarizes the best service time and speedup results achieved in the different cases by the sequential version and by the parallel implementations with the various runtimes.

| | $32 \times 32$ | | $64 \times 64$ | | $128 \times 128$ | |
|---|---|---|---|---|---|---|
| | $T_S(\mu sec)$ | $\mathcal{S}$ | $T_S(\mu sec)$ | $\mathcal{S}$ | $T_S(\mu sec)$ | $\mathcal{S}$ |
| Seq. (int) | 52.6 | 1 | 218 | 1 | 848 | 1 |
| RA − UDN (int) | 3.76 | 13.98 | 6.99 | 31.16 | 25.53 | 33.21 |
| RA − SM (int) | 5.65 | 9.31 | 8.12 | 26.85 | 28.16 | 29.9 |
| FF − QUEUE (int) | 6.68 | 7.88 | 9.42 | 23.15 | 34.06 | 24.9 |
| Seq. (float) | 187 | 1 | 748 | 1 | 3035 | 1 |
| RA − UDN (float) | 5.21 | 35.86 | 16.69 | 46.98 | 62.58 | 48.50 |
| RA − SM (float) | 6.70 | 27.89 | 17.53 | 44.72 | 65.34 | 46.45 |
| FF − QUEUE (float) | 9.26 | 20.17 | 20.69 | 37.89 | 78.42 | 38.7 |

Table 6: Summary of results of the farm benchmark: best service time ($T_S$) and speedup ($\mathcal{S}$) results for each configuration.

Finally, Fig. 15 shows the estimated number of CC requests exchanged between L2 caches during the whole execution of the benchmark. The results show that the RA−UDN version significantly reduces the number of requests. On average the reduction is of 57%. No invalidation message is transmitted in the RA−UDN version owing to home-forwarding synchronous writes to the home node that also self-invalidate the cache lines from the non-home node caches. In addition, the contention parameter is smaller: in FF−QUEUE, which is a generic run-time support, all the workers PEs make CC requests to the L2 caches of the emitter and collector PEs (see Fig. 4), while in the RA−UDN/RA−SM versions workers PEs receive C2C requests from the emitter and collector PEs only.

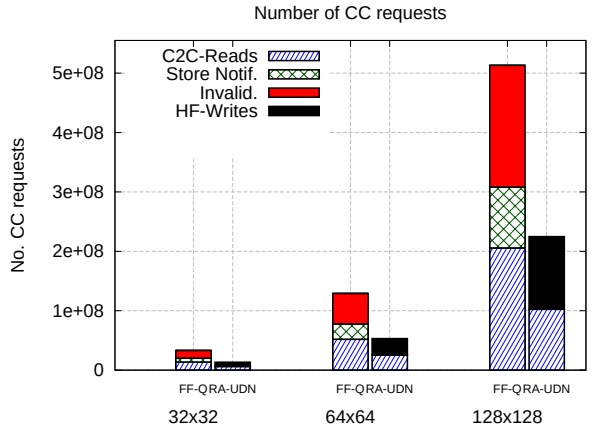



Fig. 15: CC requests exchanged between L2 caches: comparison between FF−QUEUE and RA−UDN with various matrix sizes.

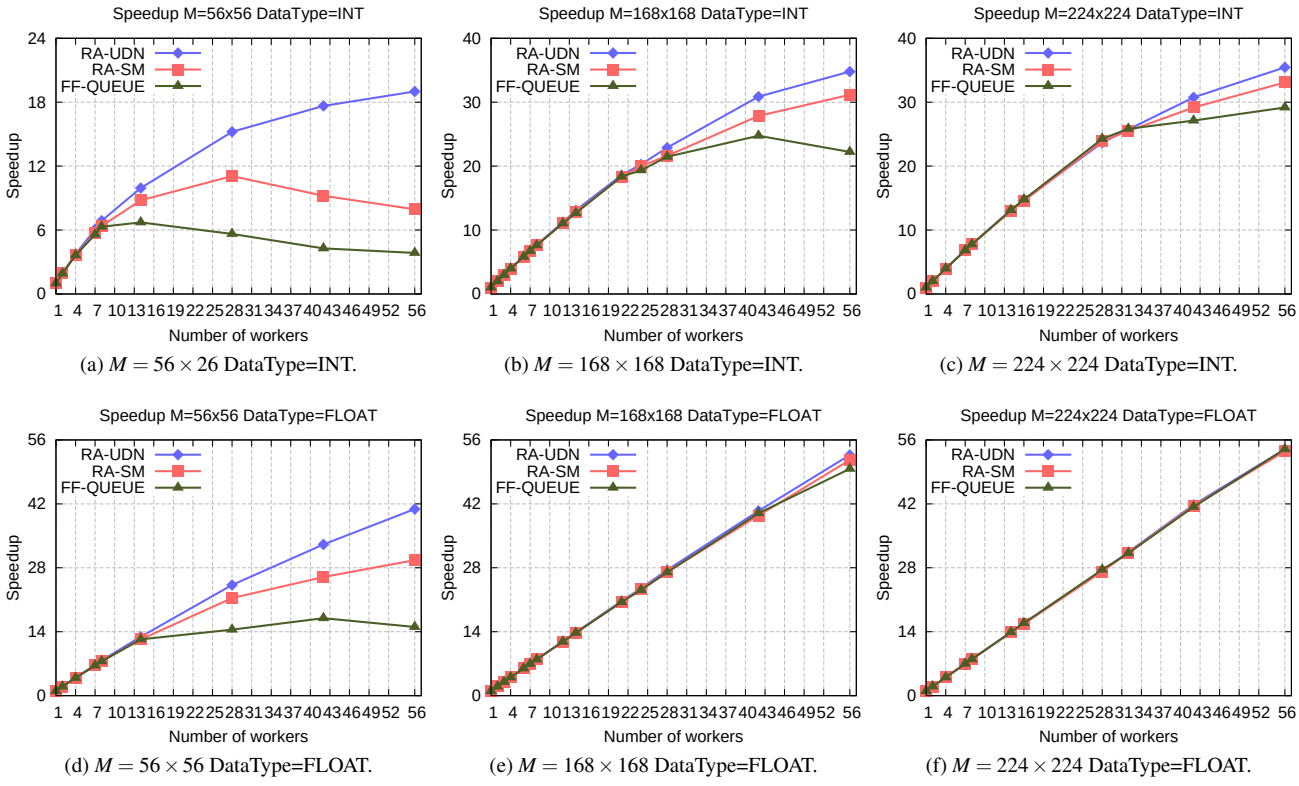
***Map benchmark.*** This benchmark consists in a linear algebra computation (*generalized matrix-vector product*) applied to a stream of $10^6$ matrices of size $M$ (one order of magnitude longer than the previous experiment). For each matrix $A$, the computation produces a vector $\mathbf{c}$ by applying the operation $\mathbf{c} = \alpha A\mathbf{x} + \beta\mathbf{y}$, where $\mathbf{x}$ and $\mathbf{y}$ are constant vectors of size $\sqrt{M}$ and $\alpha$ and $\beta$ are constants. The program is a map parallel pattern where each worker (W) operates on a partition of the current input matrix (i.e. a task is a set of contiguous rows) and produces a set of entries of the array $\mathbf{c}$. To work independently, the workers must share the vectors $\mathbf{x}$ and $\mathbf{y}$.

In order to avoid being a bottleneck, the *scatter* distribution of the input matrices is performed according to a *binary tree* topology mapped onto the workers according to a depth-first strategy. A S thread (root of the three) divides each matrix in two partitions and transmits them to two workers. Each non-leaf worker divides the received partition in two parts and transmits them to other two workers and so forth. Each worker applies the computation on its partition of A independently, and produces a corresponding number of entries of the vector $\mathbf{c}$, which are transmitted to a *gather* thread G that collects them and builds the final result vector. As for the farm, the rdy-ack communication interfaces S-W and W-G are homed at the worker PE. Furthermore, workers communicate to perform the tree-based scatter distribution. Each communication interface between $W_i$-$W_j$ is homed at the destination worker PE. Since each node of the tree has a unique parent, this strategy minimizes the number of non-home PEs accessing to the same home node PE cache.

Also in this case we study the speedup by varying the size of the matrix $M = 56 \times 56, 168 \times 168, 224 \times 224$ and the data type. We choose these values in order to have the same number of rows per worker with the maximum parallelism degree (1, 2 and 3 rows with 56 workers). The results are shown in Fig 16. The sequential time of the program is 90 $\mu$sec, 840 $\mu$sec and 1500 $\mu$sec with the three matrix sizes respectively and integer data type. For floats the grain increases of 3.60 times with respect to using integers.

Our approach achieves the greatest benefits with *very fine-grained* computations and *large* parallelism degrees. With $M = 56 \times 56$ RA−UDN is able to increase speedup of 41% and 64% than RA−SM and FF−QUEUE respectively. This case is remarkable: although the partition of each worker consists in one row of the matrix $A$ with 56 workers, the computation is still able to achieve a decent speedup of 19.21 with RA−UDN. The distance is still important but lower if we use floats instead of integers. With integers the difference between runtimes remains also with the other sizes, though smaller. A near-ideal speedup is achieved in the case of floats and $M = 168 \times 168, 224 \times 224$, and the difference is negligible with $M = 224 \times 224$. Finally, in terms of CC requests exchanged, we measured a 48% reduction with the RA−UDN runtime with respect to FF−QUEUE. The reduction is lower than in the farm benchmark, because here the communication pattern is more complex as the worker caches communicate with each other in order to perform the tree-based scatter distribution.

Tab.7 summarizes the best service time and speedup results obtained in this benchmark.




(a) $M = 56 \times 26$ DataType=INT. (b) $M = 168 \times 168$ DataType=INT. (c) $M = 224 \times 224$ DataType=INT.

(d) $M = 56 \times 56$ DataType=FLOAT. (e) $M = 168 \times 168$ DataType=FLOAT. (f) $M = 224 \times 224$ DataType=FLOAT.

Fig. 16: Speedup of the *map* benchmark with different run-time supports and optimizations on the Tilera TILEPro64 CMP.

| | $56 \times 56$ | | $168 \times 168$ | | $224 \times 224$ | |
|---|---|---|---|---|---|---|
| | $T_S(\mu sec)$ | $\mathcal{S}$ | $T_S(\mu sec)$ | $\mathcal{S}$ | $T_S(\mu sec)$ | $\mathcal{S}$ |
| Seq. (int) | 94.1 | 1 | 843 | 1 | 1502 | 1 |
| RA − UDN (int) | 4.95 | 19.01 | 24.23 | 34.79 | 42.31 | 35.45 |
| RA − SM (int) | 8.50 | 11.06 | 27.06 | 31.15 | 45.33 | 33.09 |
| FF − QUEUE (int) | 14.00 | 6.72 | 34.05 | 24.76 | 51.4 | 29.18 |
| Seq. (float) | 343 | 1 | 3093 | 1 | 5502 | 1 |
| RA − UDN (float) | 8.39 | 40.86 | 58.70 | 52.69 | 101 | 53.98 |
| RA − SM (float) | 11.56 | 29.67 | 59.90 | 51.63 | 102.8 | 53.51 |
| FF − QUEUE (float) | 20.21 | 16.97 | 62.27 | 49.97 | 101.9 | 53.98 |

Table 7: Summary of results of the map benchmark: best service time ($T_S$) and speedup ($\mathcal{S}$) results for each configuration.

## 8 Concluding Remarks and Future Work

This paper proposes a novel approach for enabling scalable fine-grained parallelism on next-generation CMPs. The tendency of recent architectures is to have ever more cores integrated into the same chip, properly interconnected by complex on-chip network infrastructures and cache hierarchies. To achieve good scalability even with *very* fine-grained parallel computations, all the possible sources of overhead in the architecture must be taken into account in the design of a run-time support for parallel programs. Notably, CC is known to be a critical part of modern CMPs, whose overhead may be a limiting factor to achieve linear scalability [18].

Our approach targets structured parallel programs [12, 13]. These patterns are characterized by a precise interaction scheme between processes/threads following a producer-consumer paradigm that can be implemented according to asymmetric synchronization mechanisms. We analyzed the impact of CC in modern CMPs: our benchmarks confirmed that the base latency of read/write operations is significantly affected by CC. Then, we identified the goals that our methodology. The first is to properly designing the run-time support in such a way as to minimize the base latency of co-operation mechanisms. Furthermore, we analyzed the problem of contention which, especially in fine-grained computations, affects caches more than memories. Caches are heavily involved in request/reply interactions and they may became highly congested in highly parallel programs that frequently exchange/invalidate cache lines. To this end, we defined a low-level architectural mechanism to deal with contention and latency reduction in producer-consumer schemes: synchronous stores with *home-forward* semantics.

This architectural mechanism allows us to design a very efficient runtime that minimizes contention among caches and the base latency in the case of structured parallel programs. We exemplified our methodology on the TILEPro64 CMP, which is equipped with non-conventional hardware facilities that make it possible an emulation of our ideas. The results confirmed our expectations: our run-time support and optimizations make it possible to achieve good speedup even in the case of very fine-grained parallel computations.

In the future our work could be extended in several research directions. First of all, other parallel patterns can be evaluated on the TILEPro64, like stencils and reduce. Sec-

ond, the approach is designed to be general and not only tailored to the Tilera CMPs. Unfortunately, some of the mechanisms that we need are not available on commodity CMPs produced by the major vendors. Therefore, we plan to provide a validation of our approach using proper architecture simulation tools by carefully understanding the potential pitfalls of using such tools to handle mechanisms not available on commodity CMPs. Third, a future interesting point is to evaluate the applicability of our approach to multi-programmed environments, not studied in this work. Finally, the idea of home-forwarding writes can be adapted to the case of multi-CMPs architectures too, featuring a hierarchical CC subsystem and shared levels of caches. This case, introduced at the end of Sect. 4.1, has not been studied in this work and will be analyzed in the future.

**Acknowledgements**

We would like to thank Dr. Daniele Buono (PhD) and Dr. Tiziano De Matteis (PhD) for their collaboration with useful and thoughtful discussions during the preparation of this work.

**References**

[1] Kurian, G., Miller, J. E., Psota, J., Eastep, J., Liu, J., Michel, J., Kimerling, L. C., and Agarwal, A., 2010. "Atac: A 1000-core cache-coherent processor with on-chip optical network". In Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT '10, ACM, pp. 477–488.

[2] Shacham, A., Bergman, K., and Carloni, L., 2008. "Photonic networks-on-chip for future generations of chip multiprocessors". *Computers, IEEE Transactions on,* **57**(9), Sept, pp. 1246–1260.

[3] Martin, M. M. K., Hill, M. D., and Sorin, D. J., 2012. "Why on-chip cache coherence is here to stay". *Commun. ACM,* **55**(7), July, pp. 78–89.

[4] Kumar, S., Hughes, C. J., and Nguyen, A., 2007. "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors". *SIGARCH Comput. Archit. News,* **35**(2), June, pp. 162–173.

[5] Danelutto, M., and Torquati, M., 2015. "Structured parallel programming with "core" fastflow". In *Central European Functional Programming School*, V. Zsók, Z. Horváth, and L. Csató, eds., Vol. 8606 of *LNCS*. Springer.

[6] Faxen, K. F., 2010. "Efficient work stealing for fine grained parallelism". In 2010 39th International Conference on Parallel Processing, pp. 313–322.

[7] Mencagli, G., and Vanneschi, M., 2011. "Qos-control of structured parallel computations: A predictive control approach". In 2011 IEEE Third International Conference on Cloud Computing Technology and Science, pp. 296–303.

[8] Bertolli, C., Mencagli, G., and Vanneschi, M., 2010. "A cost model for autonomic reconfigurations in high-performance pervasive applications". In Proceedings of the 4th ACM International Workshop on Context-Awareness for Self-Managing Systems, CASEMANS '10, ACM, pp. 3:20–3:29.

[9] Mencagli, G., Vanneschi, M., and Vespa, E., 2014. "A cooperative predictive control approach to improve the reconfiguration stability of adaptive distributed parallel applications". *ACM Trans. Auton. Adapt. Syst.,* **9**(1), Mar., pp. 2:1–2:27.

[10] Hoffmann, H., Wentzlaff, D., and Agarwal, A., 2010. "Remote store programming: A memory model for embedded multicore". In Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers, HiPEAC'10, Springer-Verlag, pp. 3–17.

[11] Kayi, A., Serres, O., and El-Ghazawi, T., 2015. "Adaptive cache coherence mechanisms with producer-consumer sharing optimization for chip multiprocessors". *Computers, IEEE Transactions on,* **64**(2), Feb, pp. 316–328.

[12] McCool, M., Reinders, J., and Robison, A., 2012. *Structured Parallel Programming: Patterns for Efficient Computation*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[13] González-Vélez, H., and Leyton, M., 2010. "A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers". *Softw. Pract. Exper.,* **40**(12), Nov., pp. 1135–1160.

[14] Mattson, T., Sanders, B., and Massingill, B., 2004. *Patterns for Parallel Programming*, first ed. Addison-Wesley Professional.

[15] Buono, D., and Mencagli, G., 2014. "Run-time mechanisms for fine-grained parallelism on network processors: The tilepro64 experience". In High Performance Computing Simulation (HPCS), 2014 International Conference on, pp. 55–64.

[16] Tilera Corporation, 2011. UG101 - Tile Processor User Architecture Manual. http://http://www.tilera.com/scm/docs/UG101-User-Architecture-Reference.pdf.

[17] De Matteis, T., Luporini, F., Mencagli, G., and Vanneschi, M., 2013. "Evaluation of architectural supports for fine-grained synchronization mechanisms". In Proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Networks.

[18] Dubois, M., and Briggs, F. A., 1982. "Effects of cache coherency in multiprocessors". *Computers, IEEE Transactions on,* **C-31**(11), Nov, pp. 1083–1099.

[19] Eggers, S. J., and Katz, R. H., 1988. "A characterization of sharing in parallel programs and its application to coherency protocol evaluation". *SIGARCH Comput. Archit. News,* **16**(2), May, pp. 373–382.

[20] Molka, D., Hackenberg, D., and Schöne, R., 2014. "Main memory and cache performance of intel sandy bridge and amd bulldozer". In Proceedings of the Workshop on Memory Systems Performance and Correctness, MSPC '14, ACM, pp. 4:1–4:10.

[21] Hackenberg, D., Molka, D., and Nagel, W. E., 2009. "Comparing cache architectures and coherency proto-

cols on x86-64 multicore smp systems". In Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, ACM, pp. 413–422.

[22] Bennett, A. J., Field, T., and Harrison, P., 1996. "Modelling and validation of shared memory coherency protocols". *Performance Evaluation,* **2728**, pp. 541 – 563.

[23] Vernon, M. K., and Holliday, M. A., 1986. "Performance analysis of multiprocessor cache consistency protocols using generalized timed petri nets". *SIGMETRICS Perform. Eval. Rev.,* **14**(1), May, pp. 9–17.

[24] Yang, Q., Bhuyan, L. N., and Liu, B.-C., 1989. "Analysis and comparison of cache coherence protocols for a packet-switched multiprocessor". *IEEE Trans. Comput.,* **38**(8), Aug., pp. 1143–1153.

[25] Marandola, J., Louise, S., Cudennec, L., Acquaviva, J., and Bader, D., 2012. "Enhancing cache coherent architectures with access patterns for embedded manycore systems". In System on Chip (SoC), 2012 International Symposium on, pp. 1–7.

[26] Vanderwiel, S. P., and Lilja, D. J., 2000. "Data prefetch mechanisms". *ACM Comput. Surv.,* **32**(2), June, pp. 174–199.

[27] Zimmer, C., and Mueller, F., 2012. "Low contention mapping of real-time tasks onto tilepro 64 core processors". In Proceedings of the 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium, RTAS '12, IEEE Computer Society, pp. 131–140.

[28] Muddukrishna, A., Podobas, A., Brorsson, M., and Vlassov, V., 2013. "Task scheduling on manycore processors with home caches". In Proceedings of the 18th International Conference on Parallel Processing Workshops, Euro-Par'12, Springer-Verlag, pp. 357–367.

[29] Lebeck, A. R., and Wood, D. A., 1995. "Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors". *SIGARCH Comput. Archit. News,* **23**(2), May, pp. 48–59.

[30] Kayi, A., Serres, O., and El-Ghazawi, T., 2014. "Bandwidth adaptive cache coherence optimizations for chip multiprocessors". *Int. J. Parallel Program.,* **42**(3), June, pp. 435–455.

[31] Patterson, D. A., and Hennessy, J. L., 2013. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[32] Sorin, D. J., Hill, M. D., and Wood, D. A., 2011. *A Primer on Memory Consistency and Cache Coherence*, 1st ed. Morgan & Claypool Publishers.

[33] Wentzlaff, D., Griffin, P., Hoffmann, H., Bao, L., Edwards, B., Ramey, C., Mattina, M., Miao, C.-C., Brown III, J. F., and Agarwal, A., 2007. "On-chip interconnection architecture of the tile processor". *IEEE Micro,* **27**(5), Sept., pp. 15–31.

[34] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J., 1990. "Memory consistency and event ordering in scalable shared-memory multiprocessors". *SIGARCH Comput. Archit. News,* **18**(2SI), May, pp. 15–26.

[35] Sorin, D. J., Pai, V. S., Adve, S. V., Vernon, M. K., and Wood, D. A., 1998. "Analytic evaluation of shared-memory systems with ilp processors". *SIGARCH Comput. Archit. News,* **26**(3), Apr., pp. 380–391.

[36] Gchnefeld, R., and Jacobi, H., 1985. "Stochastic model of a multiprocessor system in the presence of memory contention". *Annual Review in Automatic Programming,* **12**, pp. 449 – 452.

[37] Kowarschik, M., and Weiß, C., 2003. *Algorithms for Memory Hierarchies: Advanced Lectures*. Springer Berlin Heidelberg, Berlin, Heidelberg, ch. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms, pp. 213–232.

[38] Buono, D., Matteis, T. D., Mencagli, G., and Vanneschi, M., 2014. "Optimizing message-passing on multicore architectures using hardware multi-threading". In 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 262–270.

[39] Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., and Torquati, M., 2012. "An efficient unbounded lock-free queue for multi-core systems". In Proc. of 18th Intl. Euro-Par 2012 Parallel Processing, Vol. 7484 of *LNCS*, Springer, pp. 662–673.