ARTICLE IN PRESS

Future Generation Computer Systems ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

The home-forwarding mechanism to reduce the cache coherence overhead in next-generation CMPs

Gabriele Mencagli *, Marco Vanneschi, Silvia Lametti

Department of Computer Science, University of Pisa, Largo B. Pontecorvo 3, I-56127, Pisa, Italy

HIGHLIGHTS

- The overhead of cache coherence protocols has been evaluated.
- Specific requirements for an efficient runtime support have been identified.
- The home-forwarding mechanism has been described.
- A runtime support for fine-grained parallelism has been implemented.
- Our approach has been evaluated through benchmarks on the TILEPro64 CMP.

ARTICLE INFO

Article history:
Received 28 February 2016
Received in revised form
27 December 2016
Accepted 7 January 2017
Available online xxxx

Keywords:
Parallel processing
Cache coherence
Fine-grained parallelism
Chip Multi-Processors

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.

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

* Corresponding author. Fax: +39 0502212726.
E-mail addresses: mencagli@di.unipi.it (G. Mencagli), vannesch@di.unipi.it (M. Vanneschi), lametti@di.unipi.it (S. Lametti).

http://dx.doi.org/10.1016/j.future.2017.01.009
0167-739X/

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 sub-system 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. [10,11], 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 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. Section 3 will review some related works, and Section 4 will describe the nature of the CC overhead. Section 5 will give the basis for an efficient runtime design, and Section 6 will describe a runtime that meets our design principles. Section 7 will evaluate our runtime on some parallel programs. Finally, Section 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 Section 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 exact quantification of the CC overhead, but rather in strategies to design the runtime system in order to reduce both base latency and contention.

[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).

Please cite this article in press as: G. Mencagli, et al., The home-forwarding mechanism to reduce the cache coherence overhead in next-generation CMPs, Future Generation Computer Systems (2017), http://dx.doi.org/10.1016/j.future.2017.01.009