Enhancing an x86_64 Multi-Core Architecture with Data-Flow Execution Support

Nam Ho
University of Paderborn

Andrea Mondelli
University of Siena

Alberto Scionti
University of Siena

Marco Solinas
University of Siena

Antoni Portero
IT4Innovations, National Supercomputing Center

Roberto Giorgi
University of Siena

ABSTRACT

Future exascale machines will require multi–/ many-core architectures able to efficiently run multi-threaded applications. Data-flow execution models have demonstrated to be capable of improving execution performance by limiting the synchronization overhead. This paper proposes to augment cores with a minimalistic set of hardware units and dedicated instructions that allow efficiently scheduling the execution of threads on the basis of data-flow principles. Experimental results show performance improvements of the system when compared with other techniques (e.g., OpenMP, Cilk)[1].

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; C.1.3 [**Other Architecture Styles**]: Dataflow

Keywords

Many-core, multi-core, dataflow.

1. INTRODUCTION

Data-flow computing is a well known paradigm that is capable of taking advantage of the full parallelism offered by multi–/ many-core systems [2, 3]. In the data-flow execution model, a directed graph represents the flow of data among the computation activities (e.g., fine-grain threads). These activities are performed once all the required inputs become available. Research works proposed several ways to formalize this model of computation [1, 9]. Recently, the architectural exploitation has been investigated as well [2–4]. This paper proposes the enhancement of a x86_64 multi-core architecture with a small set of functional units and instructions (*Hardware Resource Manager – HRM*) that support the scheduling of fine-grain threads on the basis of data-flow principles.

[1]This article was elaborated within the framework of European Union funded project with reg. numbers CZ.1.07/2.3.00/30.0055, CZ.1.05/1.1.00/02.0070, and LM2011033, by the European FP7 projects HARPA id. 612069, ERA id. 249059, TERAFLUX id. 249013, and AXIOM id. 645496.

CF'15 , May 18-21, 2015, Ischia, Italy

http://dx.doi.org/10.1145/2742854.2742896

1.1 System overview

In our work we consider a computing system organized in the following manner: a scalable interconnection medium is used to connect a set of *Compute Clusters* (CCs), which exhibit an internal organization similar to [2] with a 2-channels shared bus interconnection. Each core is enhanced with a functional unit that executes instructions for synchronizing the data-flow threads. All these units exchange information with a centralized module connected to the shared bus (see figure 1). Our data-flow execution model is designed in such a way it can spawn a very high number of concurrent fine-grain threads (*Data-Flow Threads* – DFTs) [5, 6], each of them composed of few tens of instructions. Load operations of input data are performed at the beginning of the execution, while store operations of values for other threads are performed at the end. Each DFT owns a Synchronization Counter (SC) and a special memory region called Frame Memory Block (FMB) that stores the input values for the execution. A DFT becomes ready for the execution whenever all the required inputs become available (SC = 0). The interaction with HRM units and the DFT code is supported by four special instructions (T*64) [10].

2. HARDWARE RESOURCE MANAGER

A Local Scheduling (LS) unit is embedded in each core and directly communicates with the Central Scheduling (CS) unit. LS units are responsible for managing the T*64 instructions, allocating/de-allocating resources for the DFTs, and performing memory accesses to the FMBs. The role of the CS unit is that of distributing the workload among the cores. The LS unit is composed of three functional blocks. The LS Logic Block contains a set of finite state machines, each devoted to handle messages related to a specific T*64 instruction. Messages are sent/received through a couple of registers (*Sent MsgHandler* and *Recv MsgHandler* registers respectively). The interaction between the LS Logic Block and the bus interface is mediated by read/write FIFO queues. The local scheduling unit has a Ready Thread Block for keeping information regarding the DFTs marked as ready for the execution. This block contains a prefetcher connected to a private cache memory that ready DFTs use

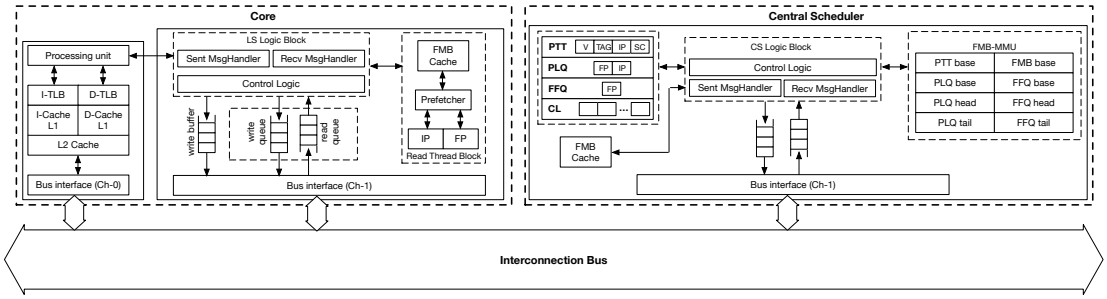

Figure 1: The internal organization of the Hardware Resource Manager (HRM).

to store their FMBs. The prefetcher accesses to the interconnection bus through the bus interface, which uses a separated communication channel (Ch-1 in figure 1) w.r.t. the standard core. Since one DFT at a time can be executed by the core, pointers to the instructions and the FMB (IP and FP registers) are integrated in the Ready Thread Block. Finally, a write buffer is added to each LS unit aiming at improving performance. Three functional blocks compose the CS unit. A memory management unit (FMB-MMU) devoted to the FMBs allows to correctly address DFT's data structures in the main memory. This unit contains a set of registers that point to the base address of three global data structures: the *Pending Thread Table* – PTT (a table which entries store ⟨IP,SC⟩ tuples, where IP is the instruction pointer of the thread), the *Preload Queue* – PLQ (a queue that keeps track of the threads ready for the execution), and the *Free Frame Queue* – FFQ (a queue that keeps track of the free frame memory blocks). PLQ and FFQ also have tail and head dedicated management registers. The CS unit has a FMB cache memory to make FMB accesses faster. All the operations performed by the CS unit are governed by the CS Logic Block which contains a finite state machine and two registers for managing incoming (*Recv MsgHandler*) and outgoing (*Sent MsgHandler*) messages on the shared bus. This finite state machine has four subsets of states, each devoted to handle the messages that are related to a specific T*64 instruction. Finally, a *Core Load* (CL) data structure, holding the number of DFTs associated to each compute core, allows to distribute the workload. Due to the limited storage space available in the CS unit, all the data structures are mapped on the main memory [2].

3. EVALUATION

We evaluated the performance of the proposed architecture using the COTSon simulation framework [7, 8]. In order to simulate LS and CS units, we extended the set of components available in simulation framework. For the experiments we considered a system running at near 1GHz equipped with a L1 (32KB + 32KB for instruction and data caches) and L2 (512KB) cache memories for each core. In order to correctly simulate the system we also estimated the latency of the T*64 instructions[3]. The performance have been evaluated resorting to the *Recursive Fibonacci Sequence* (RFS) and a *Block Matrix Multiplication* (BMM) applications. We compared T*64 execution with OpenMP and Cilk execution. Our results show that the choice of deploying a hardware scheduler is most effective when the number of running threads increases. Considering the BMM (figure 2), the speedup is improved up to 16 times the single core execution, and it is twice the OpenMP and Cilk results when executing with 32 cores. We also estimated the area overhead: the HRM represents the 3.4% of the overall cache memory area (L1 + L2 caches). This demonstrates the benefits of adopting our design: higher scalability, and low area overhead w.r.t. software scheduling approaches.

[2] PTT, FFQ, PLQ, and the frame memory blocks.

[3] With the default execution we used specific latencies for each T*64 instruction, with the optimistic execution all the latencies are set equal to 1 cycle.

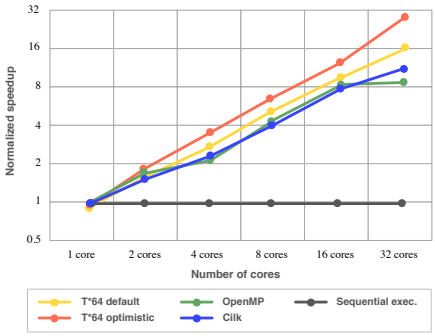

Figure 2: BMM – normalized execution speedup (input 256).

4. REFERENCES

[1] K. M. Kavi, et al., *A formal definition of dataflow graph models*, IEEE Tr. on Comp., Nov 1986.

[2] R. Giorgi, A. Scionti, *A scalable thread scheduling co-processor based on data-flow principles*, FGCS, January 2015.

[3] S. Zuckerman, et al., *Using a "codelet" program execution model for exascale machines: position paper*, ACM EXADAPT'11, USA, 2011.

[4] A. Portero, et al., *TERAFLUX: Exploiting tera-device computing challenges*. Procedia Computer Science, 2011.

[5] R. Giorgi, P. Faraboschi, *An introduction to DF-Threads and their execution model*, IEEE Proc. MPP-2014.

[6] R. Giorgi, et al., *TERAFLUX: harnessing dataflow in next generation teradevices*, MICPRO, vol. 38, no. 8, 2014.

[7] A. Portero, et al., *Simulating the future kilo-x86-64 core processors and their infrastructure*, ANSS-12, 2012.

[8] N. Ho, et al., *Simulating a multi-core x86_64 architecture with hardware ISA extension supporting a data-flow execution model*, IEEE Proc. AIMS-2014.

[9] L. Verdoscia, et al., *A clockless computing system based on the static dataflow paradigm*, IEEE DFM-2014.

[10] R. Giorgi, *TERAFLUX: exploiting dataflow parallelism in teradevices*, ACM Computing Frontiers, 2012.