Dataflow Support in x86_64 Multicore Architectures through Small Hardware Extensions

Andrea Mondelli†, Nam Ho*, Alberto Scionti§, Marco Solinas†, Antoni Portero‡ and Roberto Giorgi†
* University of Paderborn
Email:namh@mail.upb.de
† Università degli Studi di Siena
Email:{mondelli,solinas,giorgi}@dii.unisi.it
‡ IT4Innovations – National Supercomputing Center
Email:antonio.portero@vsb.cz
§ ISMB – Istituto Superiore Mario Boella
Email:scionti@ismb.it

***Abstract*—The path towards future high performance computers requires architectures able to efficiently run multi-threaded applications. In this context, dataflow-based execution models can improve the performance by limiting the synchronization overhead, thanks to a simple producer-consumer approach. This paper advocates the ISE of standard cores with a small hardware extension for efficiently scheduling the execution of threads on the basis of dataflow principles. A set of dedicated instructions allow the code to interact with the scheduler. Experimental results demonstrate that, the combination of dedicated scheduling units and a dataflow execution model improve the performance when compared with other techniques for code parallelization (e.g., OpenMP, Cilk).**

***Keywords*—manycore; multicore; dataflow;**

I. Introduction

In the near future, the number of cores is expected to be one or two orders of magnitude larger than current systems. In this context, it is well known that data movement is becoming more energy hungry than processing data [1]. This implies that data synchronization and communication overheads, experienced by multi-threaded real-world applications, may have an enormous impact on such systems. New execution models are needed to overcome the limitations of current technologies. Dataflow computing offers a simple way to achieve high-performance, and high degree of concurrency and speculation, by means of implicit synchronization [2], [3]. Architectural exploitation of dataflow principles have been investigated in several research works [4]–[11]. Generally, dataflow-inspired execution models split the applications into a large set of threads [14], [25]. Thus, a multicore architecture has the potential of executing many of these threads concurrently. Maxeler's dataflow engines [12] allow mapping FPGA's blocks with nodes of the dataflow execution graph. Other approaches, such as in Culler et al. [13], rely on the compiler capabilities, the programming languages features, and the availability of run-time libraries to provide efficient thread scheduling. The most similar approach to the ours is represented by Carbon [26]. It is designed to support the scheduling of fine-grained threads through an instruction set extension and a two-level hierarchy of hardware units. However it is limited to the pure scheduling and load-balancing of threads without considering special techniques to reduce the amount of data to be moved. On the contrary, in our approach the memory model is a key part of the design [21]–[23].

This paper advocates the enhancement of a x86_64 multicore architecture with a set of fine-grain thread scheduling units operating on the basis of dataflow principles. We synthetically refer to these units as the *Distributed Scheduler – DS*. The DS is organized as a two-level hierarchy with the aim of improving scalability of the architecture and eliminating a single point of failure [25], [28]. A small set of instructions is added to the x86_64 ISA limiting the complexity of the programming model [27].

II. System overview

We considered a System-on-Chip organized in the following manner. A scalable interconnection (e.g., Network-on-Chip) is used to connect a set of *Computing Clusters* (CCs) having the same internal organization. Specialized blocks can also be connected, in order to access external peripherals. Figure 1 shows an instantiation of the multi-node system, although this paper shows only the analysis of a single node.

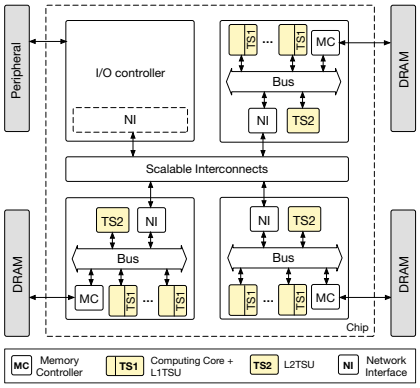

Figure 1: An example of the target system with 3 computing clusters and one I/O controller.

Since x86_64 architecture is widely adopted in different contexts (e.g., HPC, general purpose desktop systems, etc.), we decided to select it as the basis for our design. However, different architectures (e.g., VLIW) can benefit from it. The structure of a CC is based on a shared bus with two separated channels: one channel is dedicated to exchange synchroniza-

978-1-4673-8035-5/15 $31.00 © 2015 IEEE
DOI 10.1109/DSD.2015.62

tion information among the DS units, while the other transmits conventional data/instructions. Albeit this type of interconnection presents many limitations, it is still largely used in commercial processors. Coherency at CC level is needed to run the operating system, but our dataflow execution model does not need coherency (shared accesses can be manged by software transactional memory mechanisms [15]). In Figure 1 we put in evidence other key components of a CC. The Network Interface (NI) allows talking with other clusters, while the Memory Controller (MC) allows accessing DRAM modules. Each core is enhanced with a *Level-1 Thread Scheduling Unit* (L1TSU) that is responsible for executing instructions devoted to manage dataflow threads. All these units exchange information with a *Level-2 Thread Scheduling Unit* (L2TSU) connected to the shared bus. Together, the L1TSUs and the L2TSU form the DS. Each CC, as well as each core within a CC, has an associated a unique identifier. Actually, they are used to select a specific L2TSU and a specific L1TSU in the chip. In this paper we focus our attention on the organization of the single computing cluster, and presenting its execution performance.

A. Dataflow execution model

The DS main purpose is to support the execution of applications in a dataflow fashion. Although we assume for simplicity to run only one dataflow thread at a time in each core, our execution model can take advantage from the implementation of a form of Simultaneous Multi-Threading. Unlike CUDA programming model, we support the execution of a large set of concurrent threads although we allow each thread executing its independent group of instructions.

Similarly to the DF-Thread model [16], in our model each thread is composed of few tens of instructions, and their body is structured in such a way that load operations of input data are performed at the beginning, while store operations of values for other threads are performed at the end of execution. Thus, the central part of the thread body contains only computations. We refer to this type of threads as *DataFlow Threads* (DFTs). Each DFT has an associated *Synchronization Count* (SC) and a special memory block called *Frame Memory Block* (FMB), which respectively store the number of required inputs and their values. The explicit implementation of a producer-consumer scheme ensures the correctness of the thread synchronization. The producer thread generates input data for consumer threads by writing into specific locations within their FMBs. At the same time the SC is atomically decremented, and when it is reduced to zero the DFT is marked as ready for the execution. FMBs are associated to DFTs at the time of their creation, while they are deallocated once the threads complete.

The interaction with the DS units and the DFT code takes place by means of four special instructions referred as T*64 [20]. They support the creation and deletion of DFTs, as well as write and read operations to/from the FMBs. T*64 instructions are decoded locally by computing cores in the decoding stage, and forwarded to the L1TSUs for the execution. This approach allows standard x86_64 instructions and T*64 instructions to maintain separate execution paths within each core, thus maximizing the exploitation of the instruction level parallelism.

III. Distributed Scheduler – DS

We enhanced computing clusters with a variant of the micro-architecture proposed in [21], [29] (see figure 2). L1TSUs are responsible for managing the T*64 instruction flow coming from the attached cores, allocating resources for the DFTs, and performing memory accesses to the frame memory blocks. The role of the L2TSU is that of distributing the workload among the available cores. To this end, the L2TSU monitors the activity of each L1TSU, and applies specific scheduling policies. In this work we limit our analysis on a simple scheduling strategy: L1TSUs that run out of resources can interact with the L2TSU to select another core where to move the DFT's requests.

A. Level-1 Thread Scheduling Unit – L1TSU

The Level-1 Thread Scheduling Unit (L1TSU) is composed of three functional blocks. The L1TSU Logic Block contains the finite state machine (FSM) that governs the operations of the unit. It consists of four sub-FSMs dedicated to handle the messages associated to T*64 instructions, which are activated by the decoding stage of the computing core. The L1TSU Logic Block also contains data structures holding the next message to be sent through the bus or the one to be decoded. Messages are used to transmit information regarding the operation to be performed (e.g., the write operation on the FMB): they are written in a write queue (see Figure 2), where they are extracted in the FIFO order by the bus interface. On the contrary, the L1TSU becomes a bus slave every time it has to receive a message. The local scheduling unit has a Ready-Thread Block for keeping information regarding ready for execution DFTs, which can access a private cache to store their FMBs. This cache is connected to a frame memory block prefetcher, which loads the frame memory blocks in the FMB cache, as well as the instruction pointer (IP) and the frame pointer (FP) associated to the corresponding threads. The prefetcher accesses to the interconnection bus using a separated communication channel (Ch-1 in figure 2). Moreover, since one DFT at a time can be executed by the core, an IP and a FP registers are integrated in the Ready-Thread Block to store the information of this thread. It is important to note that write operations involve a data transfer with an external unit, while read operations are always performed on the locally cached FMB of the executing thread. Thus, to hide the latency generated by the bus contentions and arbitration, write operations are actually performed to a write buffer. Asynchronously, the bus interface extracts data from this buffer and sends them to the appropriate unit.

B. Level-2 Thread Scheduling Unit – L2TSU

The Level-2 Thread Scheduling Unit (L2TSU) distributes the workload among the computing cores, and synchronizes the information regarding DFTs to execute. Three global data structures serve the purpose of managing threads during their lifetime. The *Pending Thread Table* (PTT) consists of entries storing a $\langle$IP,SC$\rangle$ tuple. Only instructions that modifies the SC have access to it. The table is implemented as a small cache with a tag (TAG) and a validity (V) field for each entry. Since old data allocated by previous T*64 instructions have higher priority, this structure keeps them in the local lines and writes new data directly in the main memory. The *Preload Queue* (PLQ) is a scratchpad memory that keeps track of the threads ready for the execution. In order to run ready threads, each entry contains a $\langle$FP,IP$\rangle$ tuple. Finally, the *Free Frame Queue* (FFQ) is a scratchpad memory that keeps track of the free FMBs. Each entry contains the reference to a free FP. Due to limited storage space, these data structures are replicated in the

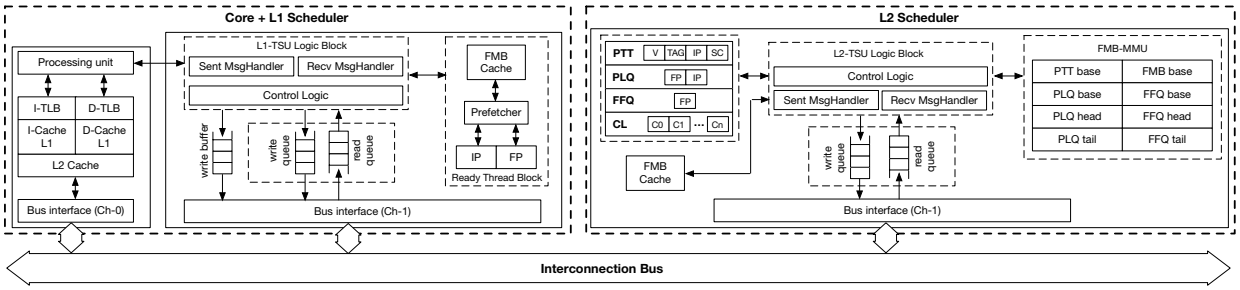

Figure 2: The internal architecture of the L1TSU integrated into each core, and the internal architecture of the L2TSU.

main memory. It is responsibility of the operating system to allocate the region where they are placed in. A set of special purpose registers (FMB-MMU) point to the base address of the data structures in the main memory. Similarly to the L1TSU, a FMB cache is used to make FMB accesses faster.

All the operations are governed by a dedicated finite state machine referred as L2TSU-FSM, which consists of four sub-FSMs (one for each specific T*64 instruction). These sub-FSMs are enabled by the interaction with the L1TSU: they update internal data structures whenever T*64 instructions are executed, and handle the messages sent/received by the L2TSU. Two registers (i.e., the Sent_MsgHandler and the Recv_MsgHandler registers) manage incoming and outgoing messages. Messages are stored in two distinct FIFO queues (respectively the Write Queue and the Read Queue). The *Core Load* (CL) data structure allows to distribute the workload among the cores in accordance to the policy implemented by the L2TSU-FSM. To this end, it holds the number of DFTs running on each core. Every time a core runs out of resources, the L2TSU identifies the core that is more suitable for the execution of a new DFT through a CL look-up.

C. Bus Transactions

In order to keep T*64 and standard traffic separated, we designed the interconnection system as a single bus organized with two channels. We assigned one channel (*Ch-0* in figure 2) to standard communications and data transfers (e.g., x86_64 instruction loading), and a second channel (*Ch-1* in figure 2) to T*64 data exchanges. L1TSU and L2TSU operations are internally translated into a sequence of transactions, while the communication protocol considers both request and reply messages. In addition, to support thread distribution among the L1TSUs, we defined a *Load Balancing* message. This message conveys the ⟨IP,FP⟩ tuple of a ready thread which is assigned to a L1TSU by the L2TSU.

IV. Experimental Results

The simulation platform we chose for the experimental evaluation consists of the integration of COTSon and SimNow tools [30]. We integrated a timing model of the L1TSU and L2TSU units. To this end, we estimated the latency for the operations performed by L1TSU and L2TSU as the number of cycles equals to the number of states of each sub-FSM. Similarly, we estimated the latency of the bus module for different configurations of the system. Cores in a single computing cluster share the same memory hierarchy: a 2-way 32KiB L1-I$ and 32KiB L1-D$, both with 3 cycles latency, a 4-way 512KiB L2$ with 5 cycles hit latency, 1GB of main memory (150 cycles latency). The cores run at 1GHz and have the following configuration for the DFT data structures: 1-way 16KiB L1TSU FMB$ with 1 cycle latency, 1-way 128KiB L2TSU FMB$ with 5 cycles latency, 32 entries of 16B line size with 2 cycles latency for PTT, PLQ, and FFQ, 32 entries of 1B for CL. The performance have been evaluated using a computing cluster configured with 1, 2, 4, 8, 16, and 32 cores, and resorting to two widely deployed examples: Recursive Fibonacci Sequence and Block Matrix Multiplication applications. The speedup has been normalized to that of the sequential execution, and compared with OpenMP, Cilk[1]. The results show that the choice of deploying a hardware scheduler is most effective when the number of running threads increases.

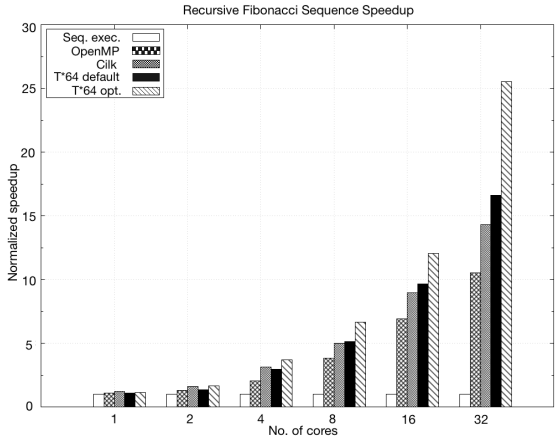

Figure 3: Recursive Fibonacci Sequence: evaluation of T*64, T*64 optimistic, OpenMP, and Cilk executions.

Figure 3 shows the normalized speedup (it is the ratio between the execution time of a parallelized execution and the sequential one) for the Recursive Fibonacci Sequence (input $n = 35$) when the T*64, OpenMP and Cilk implementations are used. For the T*64 implementation we considered two cases: (*i*) the latency are set as described at the beginning of this section (default execution), (*ii*) CPU executes one instruction per cycle, i.e., CPI = 1 (optimistic execution). The T*64 optimistic execution presents a speedup of +18% when compared to the Cilk execution on 4 cores, +34%

[1]OpenMP and Cilk versions of the test applications have been compiled without any specific optimization.

when compared to the Cilk execution on 16 cores, and +78% when executing on 32 cores. Also the T*64 default execution generally performs better than OpenMP and Cilk. This shows the capability of our hardware scheduling support to manage a high number of concurrent threads. Figure 4 shows the

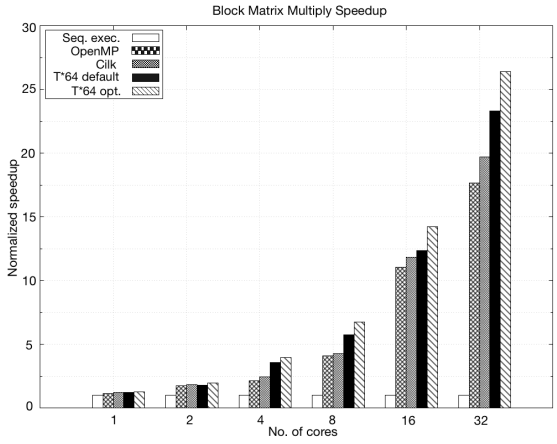

Figure 4: Block Matrix Multiplication: evaluation of T*64, T*64 optimistic, OpenMP, and Cilk executions.

normalized speedup for the Block Matrix Multiplication (input 256×256 matrices). Similarly to the previous case, the T*64 execution on 16 cores presents a +20% of speedup w.r.t. the Cilk execution, while it is smaller moving from 16 to 32 cores. The speedup is also much greater than the OpenMP. Finally, we estimated the area overhead measuring the Register Bit Equivalent (RBE)[2] for the implementation of the L1TSU and L2TSU unit on a 32 cores computing cluster. Implementing one L2TSU and 32 L1TSUs require 654.4KiB, leading to a 3.5% of the overall cache size (18MB for the 32 cores). This demonstrates the benefits of adopting our design: higher performance and scalability w.r.t. pure software scheduling approaches, and low area overhead.

V. Conclusions

We discussed a x86_64 multicore system with a small set of hardware units supporting the scheduling of fine-grain threads. The paper also advocates an execution model based on the dataflow principles, which allows to exploit a more efficient thread communication and synchronization mechanism. Experimental results demonstrate the benefits of adopting the proposed scheduling system in terms of performance and scalability w.r.t. pure software approaches (OpenMP, Cilk). The area cost of the proposed solution is also negligible.

Acknowledgment

This article was elaborated within the framework of European Union funded projects 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/H2020 projects HARPA id. 612069, ERA id. 249059, TERAFLUX id. 249013, and AXIOM id. 645496.

[2]It measures the area in terms of the number of high-performance 6T-SRAM cells required to implement the storage structures of the specified circuit.

References

[1] J. Dongarra et al., *The international exascale software project roadmap*, Int. J. High Perform. Comput. Appl., 2011.

[2] K. M. Kavi, B. P. Buckles, U. N. Bhat, *A formal definition of dataflow graph models*, IEEE Tr. on Comp., Nov 1986.

[3] G. R. Gao, *Exploiting Fine-grain Parallelism on Dataflow Architectures*, Parallel Computing, Vol. 13, No. 3, March 1990.

[4] J. B. Dennis, G. R. Gao, *An Efficient Pipelined Dataflow Processor Architecture*, IEEE, 1988.

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

[6] Yazdanpanah F., et al., *Hybrid Dataflow/von-Neumann Architectures*, IEEE Trans. Parallel Distrib. Syst., June 2014.

[7] Matheou G. and Evripidou P., *Architectural Support for Data-Driven Execution*, ACM Trans. Archit. Code Optim., January 2015.

[8] L. Verdoscia, et al., *A matrix multiplier case study for an evaluation of a configurable Dataflow-Machine*, ACM CF'15 - LP-EMS, 2015.

[9] V. Milutinovic, et al., *Guide to DataFlow Supercomputing: Basic Concepts, Case Studies, and a Detailed Example*, Springer, 2015.

[10] L. Santiago, et al., *Stack-Tagged Dataflow*, IEEE SBAC-PAD, 2014.

[11] Y. Etsion, et al., *Task Superscalar: An Out-of-Order Task Pipeline*, IEEE/ACM Microarchitecture, 2010.

[12] O. Pell, V. Averbukh, *Maximum performance computing with dataflow engines*, Computing in Science Engineering, 2012.

[13] D. Culler et al., *TAM: A Compiler Controlled Threaded Abstract Machine*, J. Parallel and Distributed Computing, 1993.

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

[15] S. Weis, et al., *Architectural Support for Fault Tolerance in a Teradevice Dataflow System*, Int'l Journal of Parallel Programming, 2014.

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

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

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

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

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

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

[22] R. Giorgi, *Transactional Memory on a Dataflow Architecture for Accelerating Haskell, WSEAS Trans. On Computers*, 2015.

[23] R. Giorgi, *Accelerating Haskell on a Dataflow Architecture: a case study including Transactional Memory*, CEA, 2015.

[24] Feng Li, et al., *Automatic Extraction of Coarse-Grained Data-Flow Threads from Imperative Programs*, Micro, IEEE, 2012.

[25] R. Giorgi, et al., *DTA-C: A Decoupled multi-Threaded Architecture for CMP Systems*, IEEE SBAC-PAD, 2007.

[26] K. Sanjeev et al., *Carbon: Architectural Support for Fine-grained Parallelism on Chip Multiprocessors*, Comput. Archit. News, 2007.

[27] M. Solinas, et al., *The TERAFLUX project: Exploiting the dataflow paradigm in next generation teradevices*, IEEE DSD, 2013.

[28] S. Weis, et al., *A Fault Detection and Recovery Architecture for a Teradevice Dataflow System*, IEEE DFM, 2011.

[29] N. Ho, et al., *Enhancing an x86_64 Multi-Core Architecture with Data-Flow Execution Support*, ACM Computing Frontiers, 2015.

[30] E. Argollo, et al., *COTSon: infrastructure for full system simulation*, ACM SIGOPS Operating Systems, 2009.