T-Star (T*): An x86-64 ISA Extension to support thread execution on many cores

Antoni Portero, Zhibin Yu and Roberto Giorgi
University of Siena Dipartimento di Ingegneria dell'Informazione Via Roma, 56 -- 53100 SIENA - Italy

ABSTRACT

The number of cores per chip keeps increasing in order to improve performance while controlling the power. According to semiconductor roadmaps, future computing systems will reach the scale of 1 Tera devices in a single package and therefore many-core (e.g. 1000 or more) will be the norm. Here, we describe an ISE (ISA Extension) that we are experimenting in the x86-64 ISA in order to provide an efficient, fast support for fine-grained threads. The new ISE enables a different execution model based on the availability of data and opens the doors for many architectural optimizations not possible in current cores. We also describe the architectural support related to the T* extension

KEYWORDS: Many-core; Multi-Threading; Dataflow; Parallelism;

1 Introduction

We assume that the applications can take advantage of either sequential or parallel programming models [EDP10]. At compile time the application is divided into a number of data-flow threads (DF-threads) and a producer-consumer graph that encodes the data dependencies among the threads [KGA01],[GPP07],[GPP09]. A DF-thread can be scheduled for execution after all its inputs become available. It enters the execution phase once the physical resources (e.g. a core) are available for it, based on scheduling criteria. Therefore, the basic execution model is a multithreaded execution model that exploits application parallelism at different levels. It derives from the data-flow principles [EDP10],[PZG11]. The platforms that we are considering contain 1000 or more cores.

2 Basic Execution Model

The execution model aims at supporting the execution of a very large number of fine-grained threads that are generated after the compilation process. We also want to target threads that have a "dataflow behavior" in the sense that they are repeatable with no-side effects and their inputs and outputs are clearly identified. We call these threads DF-Threads (dataflow threads).The execution model uses special instructions (the T* ISA Extension) to support the execution of data-flow threads (DF-threads). In this scenario we are exploring how different applications may exploit the benefits of the features of different programming models. Experimentation and tight collaboration with the compiler and simulation tools will assess the merits of these programming models. The results will

help in the development of optimizations that will be studied in the context of a more advanced architecture. We begin by providing background information on the data-flow execution model that we use.

3 T* Architectural Support

T-Star (T*) is designed to exploit Thread Level Parallelism (TLP) by using many simple off-the-shelf cores and it builds on previous models like DTA [GPP07],[GPP09] and SDF [KGA01].Similarly to DTA, T* addresses scalability by a hierarchical structure of the nodes and a distributed scheduler. The architecture is divided into nodes. Each thread that uses T* have a portion of local memory (known as frame) associated with it, where data that are needed for the execution are kept. Only when all data that are needed for execution are stored into the frame, a thread will execute. Since frames are located near the processor, accesses to frame memory should have very low latency. Hence, the pipeline should not stall because of frame memory accesses. The thread management is handled by a Distributed Thread Scheduler (DTS) which is composed by a hierarchy of node level scheduling units (D-TSUs or Distributed Thread Scheduling Units) and of core level scheduling units (L-TSUs ore Local Thread Scheduling Units). The D-TSU is responsible for allocating tasks for processors inside the node and for maintaining balanced workload on each of them. The L-TSU is located inside each core, and it is responsible for managing frames and execution of threads in the core.

4 A possible top-level design of a T* based architecture

We assumed that the compiler implicitly embeds the threads' data dependencies and consumers in the code of each thread. Moreover, at runtime new dependencies can be managed through the allocation of the threads' DF-frames. A DF-frame is allocated for every newly created DF-thread and each DF-thread writes its results in the DF-frames of its consumers. A thread is scheduled to run when all its input data are available in the corresponding frame and its "parent" completed.

The architecture is an "evolving architecture" where the Execution Model decouples the Programming Model from the Architecture. In the initial instance of the architecture the basic computational elements (Figure 1) are the Cores which contains a processing element (in our case an x86-64 based core with our T* ISE [RG11],[KGA01],[GPP07]) along with its L1 Cache and some core level hardware like the L-TSU. Each core may also include a partition of the L2 Cache. In order to support the data-flow execution of threads, each core also includes a hardware module that handles the scheduling of threads, the Local Thread Scheduling Unit (L-TSU). Cores are grouped together into nodes. In addition to the cores, the nodes also contain a hardware module to coordinate the scheduling of the data-flow threads among the cores, the Distributed Thread Scheduling Unit (D-TSU). We can assume that the cores within a node can communicate with low latency. Each core is identified with a unique id, the Core ID (CID), and each group of cores belongs to a node whose id is the Node ID (NID). Nodes are connected via an inter-node network, the Network on Chip (NoC). Cores within a node are also connected via the NoC. A NoC may not distinguish among inter-node or Inter-core NoC, i.e., it can be a single NoC. The Task Pipeline (TP), may decompose coarser grain threads into fine-grain threads to be scheduled by the TSU

units [ECR+10]. Also, we have the I/O devices at the top level. These devices are controlled by certain dedicated cores, the Service-Cores, which run the OS for that purpose (not specifically highlighted in the Figure 1).

5 Proposed Instruction Set Extension (T*)

Basically, the extension consists in two instructions for generating/stopping threads, two instructions for operating on input/output data of the thread, two instructions for allocating/freeing special purpose memory (the memory model is not in the scope of this document, but it is an essential part of our execution model). Besides the ISE, we need the architectural support that we described above. In the following, we assume that the size of the operands is by default 1 machine word (e.g. 64 bits for x86-64 platforms).

Table 1: T* proposed x86-64 ISA extension description

	T* INSTRUCTIONS	**IMPLIED COMPILER TARGET**
Synopsis	**TSCHEDULE *RS1, RS2, RD***	***<frame_pointer>* = TSCHEDULE(<*IP*>, <*SC*>)**
Description	This instruction allocates the resources (a DF-frame of size ***RS2*** words and a corresponding entry in the Distributed Thread Scheduler – or DTS) for a new DF-thread and returns its Frame Pointer (FP) in ***RD***. ***RS1*** specifies the Instruction Pointer (IP) of the first instruction of the code of this DF-thread and ***RS2*** specifies the Synchronization Count (SC).	
Notes	The allocated DF-thread is not executed until its SC reaches 0. The TSCHEDULE can be conditional or non-conditional based on the value stored in the zero flag. If the zero flag is set to 1 then the TSCHEDULE will take effect, otherwise it is ignored.	
Synopsis	**TDESTROY**	**TDESTROY**
Description	The thread that invokes TDESTROY finishes and its DF-frame is freed, (the corresponding entry in the Thread Scheduling Unit is also freed).	
Notes	-	
Synopsis	**TWRITE *RS, RD, offset***	***(<frame_pointer> + <offset>) = (<source_register>)**
Description	The data in ***RS*** is stored into the DF-frame pointed to by ***RD*** at the specified offset.	
Notes	***Side Effect***: The Distributed Thread Scheduler decrements the SC of the corresponding DF-thread entry (located through the FP): $SC_{FP} = SC_{FP}-1$	
Synopsis	**TREAD *offset, RD***	**(<destination_register>) = *(<self_frame_pointer> + <offset>)**
Description	Loads the data indexed by 'offset' from the self (current thread) DF-frame into ***RD.***	
Notes	***Assumption***: the DTS has to load into the register implicitly used by TREAD the value <self_frame_pointer>. In a x86-64 implementation, we can reserve RAX for this purpose.	
Synopsis	**TALLOC *RS1, RS2, RD***	**<pointer> = TALLOC (<*size*>, <*type*>)**
Description	Allocates a block of memory of ***RS1*** words. The pointer to it is stored in ***RD***. ***RS2*** specifies the special purpose memory type.	
Notes	The Distributed Thread Scheduler tracks the memory allocated. An implementation can code <type> in the 2 LSB of <size>	
Synopsis	**TFREE (*RS*)**	**TFREE(<pointer>)**
Description	Frees memory pointed to by ***RS***.	
Notes	The Thread Scheduling Unit tracks the memory deallocated.	

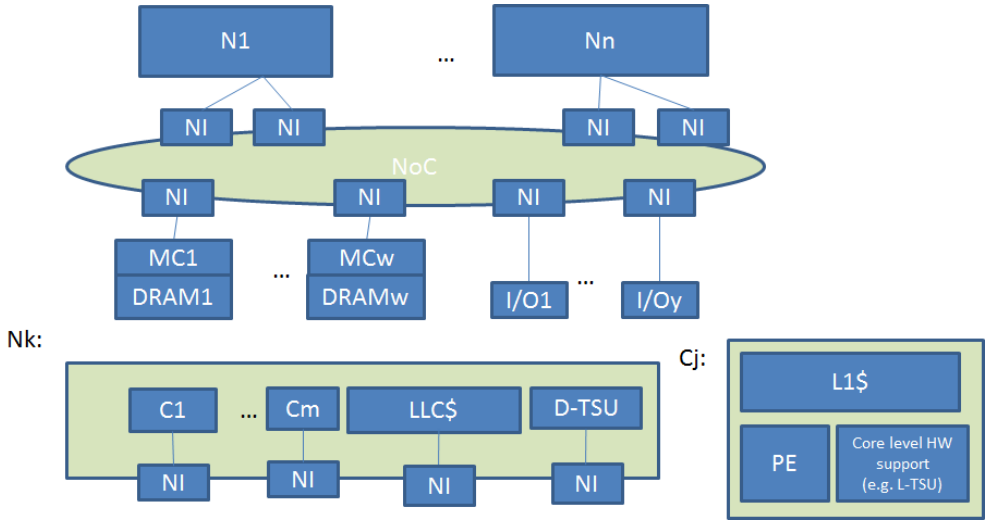

Figure 1: TERAFLUX High Level Architecture (NoC: Network on Chip, LLC: Last Local Memory, MC: Memory Controller, NI: Network Interface, N: Node, C: Core, PE: Process Element, L-TSU: Local Thread Scheduler Unit, D-TSU: Distributed Thread Scheduler Unit)

5 Conclusions

In this document we introduce a new x86-64 ISA Extension to support a fine-grained data-driven execution model. This execution model is based on multithreaded execution that exploits application parallelism at different levels. It derives from the data-flow. This model has already shown its benefits in the past but it has not been proved for the new many-cores. We are working in demonstrating the concepts in systems with 1000 cores or more.

Acknowledgements

This work was partly funded by the European FP7 project TERAFLUX id. 249013, http://www.teraflux.eu, HiPEAC IST-217068, and IT PRIN 2008 (200855LRP2).

References

[EDP10] Exploiting Dataflow Parallelism in Teradevice Computing. http://www.teraflux.eu, 2010-2014.

[PZG11] Antoni Portero, Zhibin Yu, Roberto Giorgi, "TERAFLUX: Exploiting Tera-device computing Challenges", fet11 The European Future Technologies Conference and Exhibition, 4-6 May, 2011, Budapest.

[RG11] Roberto Giorgi et al, "PR, D7.2– Def. of ISA extensions, custom devices and External COTSon API extensions", FET proactive 1: Concurrent Tera-Device Computing (ICT-2009.8.1) PRJ. NUM.: 249013

[KGA01] Krishna M. Kavi, Roberto Giorgi, Joseph Arul, "Scheduled Dataflow: Execution Paradigm, Architecture, and Performance Evaluation", IEEE Trans. Computers,, Los Alamitos, CA, USA, vol. 50, no. 8, Aug. 2001, pp. 834-846

[GPP07] R. Giorgi, Z. Popovic, N. Puzovic, "DTA-C: A Decoupled multi-Threaded Architecture for CMP Systems", Proc. IEEE SBAC-PAD, Gramado, Brasil, Oct. 2007, pp. 263-270

[GPP09] R. Giorgi, Z. Popovic, N. Puzovic, "Exploiting DMA to enable non-blocking execution in Decoupled Threaded Architecture", Proc. IEEE Int.l Symp. on Parallel and Distributed Processing – MTAAP Multi-Threading Architectures and Applications, Rome, Italy, May 2009, pp. 1-8.

[ECR+10] Yoav Etsion, et al, "Task Superscalar: An Out-of-Order Task Pipeline.", MICRO 2010, pp. 89-100