Cores as Functional Units: A Task-Based, Out-of-Order, Dataflow Pipeline

Yoav Etsion[1], Alex Ramirez,
Rosa M. Badia, Jesus Labarta

Barcelona Supercomputing Center (BSC), c/Jordi Girona 29, 08034 Barcelona, Spain

ABSTRACT

The shift towards on-chip parallelism brings forth an effort to design intuitive parallel programming models that can be used by common programmers. In that context, dataflow programming models show promise for their simplicity and potential performance gains. However, dataflow models require complex runtime support as it is infeasible to statically identify all data dependencies at compile time. In this paper we overview a hardware based out-of-order, pipeline that manages coarse-grain data dependencies at runtime. The design harnesses the inherent parallelism available in hardware to efficiently manage the runtime dependency graph and issue multiple independent tasks concurrently. While the design itself is fashioned after the StarSs programming model, it is general enough to support other coarse-grain dataflow models.

KEYWORDS: StarSs, dataflow, parallel programming

1 Introduction

For the past two decades parallel programming was largely based on notoriously hard-to-use programming models such as MPI and OpenMP, thus leaving large scale parallel programming in the hands of a small number of skilled professionals. The shift towards on-chip parallelism however, motivated researchers to design more intuitive programming models [HM93, NBGS08]. In this context, dataflow programming models such as the StarSs [BPBL06, PBBL07], Data-Driven Multithreading (DDM) [TESK06], and DTA [GPP07] have particularly shown promise in their ease-of-use and obtained performance.

This paper describes a hardware accelerator supporting execution of task-based dataflow programming models on heterogenous CMPs. The accelerator manages all runtime data dependencies, identifying and issuing data-independent tasks concurrently. While the accelerator is designed after the StarSs programming model, it is general enough to support most

[1]Yoav Etsion is supported by a *Juan de la Cierva* Research Fellowship from the Spanish Ministry of Science.
[2]This work is supported by the HiPEAC European Network of Excellence (IST-004408), the SARC European Project (EU contract 27648-FP6), and the Spanish Ministry of Education (CICYT TIN2004-07739-C02-01).
[3]E-mail: {yoav.etsion,alex.ramirez,rosa.m.badia,jesus.labarta}@bsc.es

coarse-grained data-flow models, as long as data dependencies are known at task creation.

The StarSs programming model — or *-SuperScalar in its full name — follows the out-of-order execution model of modern superscalar processors [SL04]. This task-based model requires that the programmer to explicitly annotate functions as computational tasks, and mark their input and output operands. At runtime, program execution consist of running the sequential main thread, which spawns the computational functions as tasks. Programmer annotations are this translated at runtime to a data dependency graph, enabling tasks created in program order to be executed in data order.

The hardware data-flow accelerator presented here is modeled after modern superscalar processors, but while such processors track fine-grain data dependencies to increase available instruction level parallelism, the accelerator targets coarse grained *task-level* parallelism. Issued tasks therefore wait in *task reservation stations* (TRSs) until their data dependencies are met, at which time they are sent to an available processing element (PE) for execution. In this manner, heterogenous processor cores are utilized in the same manner as functional units in a superscalar processor.

To achieve high performance and scalability, the accelerator utilizes the parallelism inherent in hardware design to reduce the overall task preparation and dispatch overhead. This is achieved by utilizing two orthogonal methodologies:

1. Pipelining the preparation of multiple tasks to increase throughput

2. Distributing the inter-task data-flow by using multiple asynchronous instances of the different operational elements to reduce the per-task overhead.

In addition, the need to scale the data-flow design in order to manage hundreds and even thousands of task simultaneously, precludes the use of any inherently non-scalable broadcast-based communications used in modern superscalar processor cores to dissipate operand data. For this purpose, the mechanism's only requirement from its internal interconnect is to provide scalable point-to-point communications, thus providing efficient support for a large number of in-flight tasks. Furthermore, the latency tolerance provided by managing coarse-grained task-level data dependencies enables the use of large embedded DRAM (eDRAM) blocks to store in-flight tasks, thus allowing the task window to scale even further — with a goal of supporting 100,000 in-flight tasks.

2 Operational Flow

The design, depicted in Figure 1 can be viewed as a meta-pipeline operating at task granularity, and consisting of 4-stages: Fetch, Decode/Issue, Execute, and Commit.

The *Fetch* stage is nothing more than a DMA engine fetching task data from the memory resident task buffer. As tasks are written to the task buffer sequentially by the application's main thread, the fetch stage parses the data and forwards the tasks to the decode stage one at a time.

The *Decode/Issue* stage resolves data dependencies and issues the tasks to wait for their dependencies to be met. Issuing tasks must be performed in task order to assure correct resolution of dependencies. Since multiple tasks may require renaming of the same parameter, an out-of-order issue may incorrectly link consumers to the wrong producer.

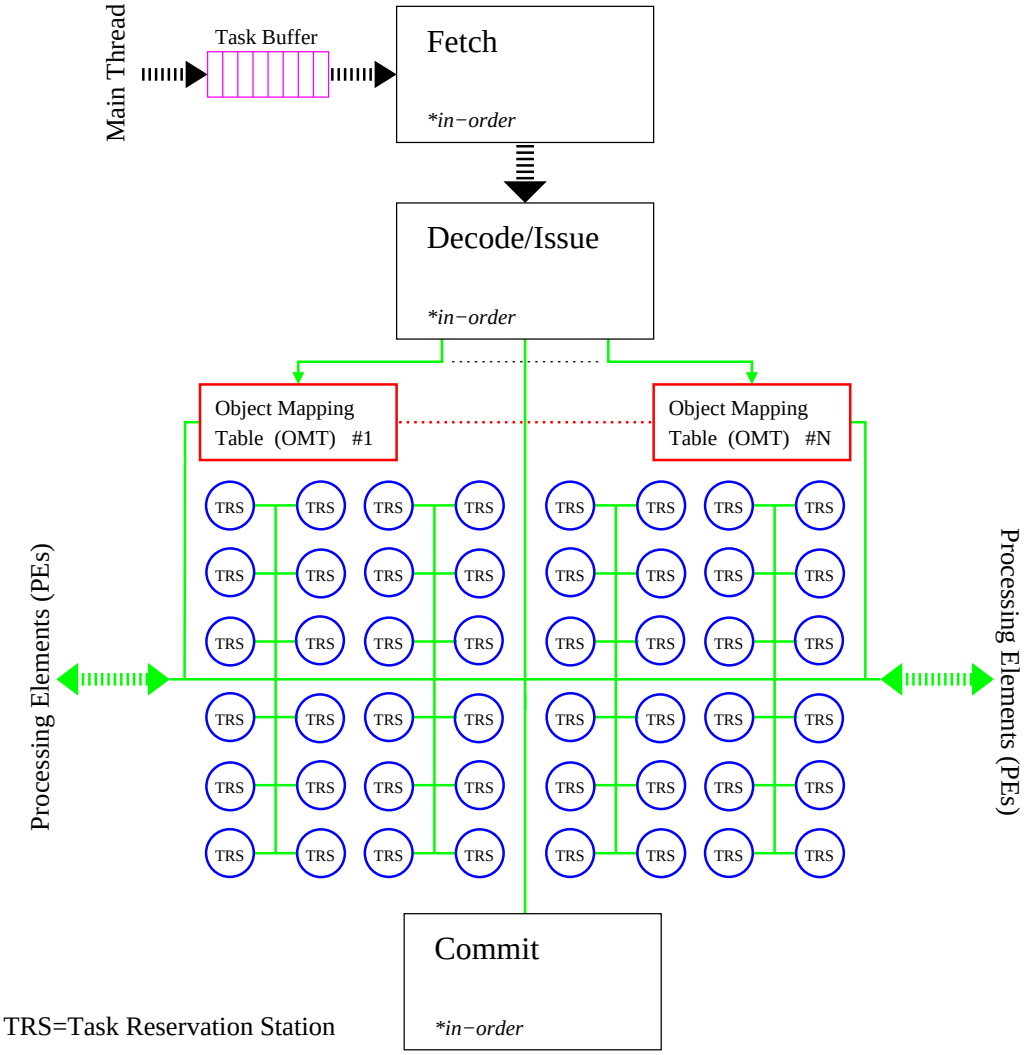

Figure 1: *Design of the task pipeline. Tasks are fetched from an in-memory buffer, and wait in the* task reservation stations *(TRSs) until all their dependencies are met. Effectively, the communications patterns between the TRSs represent an overlay mapping of the task dependence graph onto the TRS fabric.*

The issue logic first allocates a *task reservation station* (TRS) slot that will hold the pending task. Each TRS is designed to hold a large number of pending tasks whose data dependencies have not yet been met, and to issue the task to preallocated PE once it is ready to execute. Once a TRS slot has been allocated, the *Decode/Issue* logic sends each of the task's parameters to the allocated task TRS through the object mapping tables (OMTs). These tables hold information about the latest version of the parameter object and its producing task. The specific table that should be accessed is determined by hashing the parameter's address using a simple hash function that distributes the memory objects across the tables. All parameters must therefore be looked up in the OMTs in order to identify the inter-task data dependencies. If the parameter is not found in the OMTs, it can be safely read from its original in-memory location — which is sent to the task's TRS. Otherwise, the issued task must wait until the producer task finishes for the data dependency to be met.

When a TRS receives a parameter along with the information about the data producer

— as looked up in the OMTs — the TRS registers itself as a consumer with the TRS managing the producer task. The registration with the producer enables the latter to notify data consumers when the data is ready, without resulting to broadcasting.

The *Execute* stages manages the pending tasks until their dependencies are met, at which point tasks are dispatched to PEs. As described above, once a TRS is assigned a task it registers itself with all the task's data producers. When a data producer notifies that an operand is ready, the TRS marks the operand as ready in the task's slot. As soon as all of task's operands become ready, the task can be dispatched, and is sent to a preallocated PE for execution. Dispatching a task to a PE and involves transferring all the task's parameters to the PE's local store. Since all the task's data dependencies are known beforehand, this is in fact done while the PE computes a previous task, thus hiding all communications latency.

TRSs are therefore passive components in the distributed mechanism, and only react to task-issue and data-ready messages. They thus consist mainly of an eDRAM block to store task information, and a simple logic to manage the communications protocol. Furthermore, the early dependence resolution performed by the decoder enables each TRS to operate independently, limited only by the inter-task data-flow. This design provides the scalability required in order to handle a large number of pending tasks — simply by replicating TRSs.

Finally, the *Commit* stage acts as an sequencer finalizing the tasks: each TRS signals the commit unit when the task is finished, and the commit unit orders finalizations by signaling finished TRSs to finalize in-order. In-order commits guarantee that there are no more consumers trying to read a memory locations that is about to be written to by a new producer. This eliminates the need to maintain a version chain for each data object.

References

[BPBL06] P. Bellens, J.M. Perez, R.M. Badia, and J. Labarta. CellSs: a programming model for the Cell BE architecture. *Supercomputing*, Nov. 2006.

[GPP07] R. Giorgi, Z. Popovic, and N. Puzovic. DTA-C: A decoupled multi-threaded architecture for CMP systems. In *Intl. Symp. on Computer Architecture and High Performance Computing*, pages 263–270, Oct. 2007.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Intl. Symp. on Computer Architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[PBBL07] Josep M. Perez, Pieter Bellens, Rosa M. Badia, and Jesus Labarta. CellSs: Making it easier to program the Cell Broadband Engine processor. *IBM Journal of Research and Development*, 51(5):593–604, Aug 2007.

[SL04] John Paul Shen and Mikko H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. Mcgraw-Hill, Jul 2004.

[TESK06] Pedro Trancoso, Paraskevas Evripidou, Kyriakos Stavrou, and Costas Kyriacou. A case for chip multiprocessors based on the data-driven multithreading model. *Intl. Journal of Parallel Programming*, 34(3):213–235, 2006.