Parallel execution with Data-Driven Multithreading

Kyriakos Stavrou, Paraskevas Evripidou, and Pedro Trancoso

Department of Computer Science, University of Cyprus, 75 Kallipoleos Ave., P.O.Box 20537 - 1678 Nicosia, Cyprus, E-mail: {tsik,skevos,pedro}@cs.ucy.ac.cy

ABSTRACT

Chip Multiprocessors are currently dominating the market and the number cores is increasing rapidly. As such, models of execution able to efficiently exploit this parallelism are in high demand. Models based on the dataflow principles, such as the Data-Driven Multithreading (DDM), are known to be very good candidates. In this paper we briefly present the benefits of the DDM model of execution, its current implementation and the way it handles loop structures. In addition, we briefly present the future directions of this work.

KEYWORDS: Data-Driven Multithreading, Chip Multiprocessors, DDM-CMP, multi-core chips

1 Introduction

Chip Multiprocessors dominate the current CPU-market. Major chip manufacturers, like Intel, are already working on prototypes with 80 or more cores. As the trend of including an increasing number of cores in the same chip is expected to continue, models of execution that efficiently exploit this parallelism are in high demand. Such models should exploit the available parallelism with minimum overheads. At the same time, the programming model must hide the details of the underlying hardware. Finally, programming should be intuitive and close to sequential model.

The dataflow model of execution, proposed in the early 70’s by Dennis, is well known to exploit the maximum amount of parallelism. In dataflow instructions are partially ordered, only the true data dependencies exist. In contrast, in the Von Neuman model of execution, instructions have full ordering and special hardware techniques are required to dynamically identify parallelism. With dataflow execution however, parallelism is inherent in the model.

Despite its advantages, dataflow has certain limitations. In particular, it is not trivial to handle complex data structures, it requires significant hardware resources and special programming languages. Data-Driven Multithreading (DDM) [Kyri05] is a model of execution which is based on the dataflow principles but manages to overcome its limitations.
2 Data-Driven Multithreading (DDM)

A program in DDM is an arbitrary collection of DDM threads. In turn, a DDM thread is a collection of instructions. A thread is scheduled for execution in a data-driven manner i.e. only when all the threads it depends on have completed their execution. The dependencies between DDM threads are expressed as the Synchronization Graph of the program (nodes represent threads and arcs data dependencies). For thread synchronization based on the DDM model of execution, each processing node incorporates a simple hardware unit, the Thread Synchronization Unit (TSU). Based on the synchronization information the TSU schedules the threads for execution.

DDM does not require special programming languages, as such, it can handle data structures in a general way [Stav06]. In addition, DDM does not require any modifications to the CPU core [Kyri05]. As for the TSU, it is implemented as a memory-mapped and therefore the communication between CPU and TSU can be performed with simple read and write instructions. This allows DDM applications to execute on top of unmodified Operating Systems. Finally, as explained in [Stav06], the hardware requirements of the TSU are minimal.

2.1 Current state of the DDM-CMP architecture

The current implementation of DDM applies the DDM model of execution on a shared-memory Chip Multiprocessor. DDM-CMP [Stav06] is a single-ISA, homogeneous CMP able to support the Data-Driven Multithreading model of execution. DDM-CMP combines the benefits of the DDM model together with those of the CMP architecture without requiring any modification to the OS or the CPU cores. As such, it enables the execution of both DDM and non-DDM applications with minimal hardware overhead.

A full-system simulator has been developed for the DDM-CMP architecture based on the Simics platform. The simulated system was tested with both x86 and SPARC CPUs and multiple different versions of Linux. DDM programs are full compatible at the source-code level i.e. the same DDM program can be compiled for all the different configurations tested without any modification to the source code.

At the same time, a hardware prototype is being developed for the architecture using the Virtex II Pro platform. The two CPU cores will be used for the execution of the DDM threads whereas the FPGA logic for the implementation of the TSU.

Finally, to ease programmability we have developed pragma directives that allow expressing the data dependencies between threads. Using a specially developed tool, the DDM-C-Preprocessor [Tran07], these directives are translated into instructions of the C programming language and calls to the DDM runtime system.

Currently we are converting several benchmark applications to the DDM model of execution in order to perform a detailed design space analysis. These benchmarks will also be used to evaluate several potential enhancements.

3 Case Study: The trapezoidal rule for integration (Trapez)

In this section we present the execution of the DO-ALL loop found in the Trapez scientific kernel under the DDM model. Note however, that DDM is not limited to DO-ALL loops, but is able to handle complex types of dependencies between loop iterations.
The core of the Trapez application is presented in Figure 1-(a). To port the program to the DDM model of execution, the user needs to add the directives shown shaded Figure 1-(a). These pragma directives denote that all iterations of the loop can be executed in parallel with the reduction variable area accumulating the result.

Given this information the preprocessor will generate the Synchronization Graph depicted in Figure 1-(b). Thread 1 executes the initialization phase, one instance of Thread 2 per execution processor executes the loop, Thread 3 for CPU1 and Thread 4 for CPU 2 add the partial results, and finally, Thread 5 prints the output.

![Figure 1: (a) The TRAPEZ scientific kernel and its DDM representation (b) Synchronization Graph for a system with 2 execution nodes (c) The execution of the main loop (d) The TSU support for loops](image)

What deserves better explanation is the operation of Thread 2 which executes the iterations of the loop. Notice that a thread is not identified by its number only but also by the iteration it executes. As such, multiple instances of the same thread can be active at any time point, each referring to a different iteration of the loop. In our example, Thread 2 has 32 active instances for each execution node at any time point.

In DDM no ordering between iterations is enforced, i.e. iteration \( n \) may be executed before or after iteration \( n - 1 \). What is important is that this applies for the iterations executed by the same processor as well (Figure 1-(c)). This characteristic allows the TSU to perform efficient data prefetching [Kyri04]. For the iterations that are executable, the TSU issues the corresponding prefetch requests. When the CPU requests the next iteration to execute the TSU will select among the ready ones, the one for which the data it requires are in the cache.

The TSU offers special support for loops. This allows it to handle loops with complex dependencies among iterations as well as loops with very large number of iterations. This second characteristic is explained using Figure 1-(d). Loading onto into the TSU information about all iterations would result in very large hardware requirements. The TSU is able to provide the same functionality using only minimal hardware resources. In particular, initially only a subset of the loop iterations are loaded onto the TSU (32 in our example). When the CPU completes the execution of an iteration, the TSU automatically loads the next iteration (Phase 2). Therefore, at any time point 32 iterations of the loop will exist in the TSU. This number can be increased by initially loading more threads. The number of active iterations introduces a tradeoff between the potential for better performance and TSU space utilization. Finally, for the last iterations (Phase 3), the TSU just decreases the Ready Count value, a value set statically to identify when a thread is ready for execution, of the next thread (Thread 3).

The performance results, which are not presented here due to space limitations, are very promising especially for loop with larger bodies.
4 Future Directions

In this section we describe the future directions of our work.

**Operating System:** The DDM-CMP machine uses an unmodified Operating System. We are currently studying the interaction between the DDM applications and the OS. Such topics include the OS-related overheads, statically loading the TSU and avoiding Context Switches through combining DDM applications.

**Prefetching:** The fact that in DDM it is the TSU that defines which thread is to be executed next, allows the TSU to prefetch the data the thread requires [Kyri04]. Such prefetching techniques could require small modifications to the caches for direct communication with the TSU and/or special functionality for efficient TSU-triggered prefetching.

**Multi-chip DDM:** Shared-memory Chip Multiprocessors are known to have scaling problems. For efficient scaling to a very large number of processors, a hierarchical DDM-CMP configuration that combines multiple DDM-CMP chips can be used. Execution nodes inside the CMP will operate using shared-memory whereas for the communication between these CMPs we will study special TSU-TSU interconnection schemes.

**Software emulated DDM:** The traditional implementations of DDM use a special hardware unit, the TSU, for thread synchronization. For existing shared memory multiprocessor systems to benefit from the DDM model of execution a different approach must be used. Instead of having to add the TSU as a hardware unit to the system, its functionality may be emulated by a software entity that could execute on a core of a commodity multi-core architecture.

5 Conclusions

In this paper we have presented the current status of the DDM-CMP architecture which applies dataflow principles on modern CMPs with no need for modifications to the OS or the CPU core. We explained how DDM manages to execute loops with very large number iterations. Finally, we presented future directions of this work.

References


