CARE: Overview
of an Adaptive Multithreaded Architecture

Andrés Márquez and Guang R. Gao

University of Delaware, Newark DE 19716, USA,
{marquez,ggao}@capsl.udel.edu

Abstract. This paper presents the CARE (Compiler Aided Reorder Engine (CARE)) execution and architecture model. CARE is based on a decentralized approach for high-performance microprocessor architecture design – a departure from the mainly centralized control paradigm that dominated the traditional microprocessor architecture evolution. Under CARE, a processor is powered by a large number of fine-grain threads (called strands), each enabled by individual events – such as those due to control dependency or data dependencies with unpredictable and/or long latencies. As a result, the CARE architecture consists of a large grid of small processing cells and their local memories. We outline the CARE architecture design as well as the related issues in strand chaining and synchronization support. Some experimental results are also presented.

In this paper, we study alternative approaches for microprocessor architectures as a departure from the mainly centralized control paradigm that dominated the traditional microprocessor architecture evolution. In other words, we can employ a fully decentralized control paradigm - that is the processor chip is controlled by a large number of fine-grain threads, each enabled by individual events due to program flow. As a result, the hardware architecture consists of a large grid of small processing cells and their local memories.

An important open question is: what execution and architecture model will be best suited to harness such a decentralized architecture paradigm? The CARE Execution and Architecture model introduced in this paper provides an entry point in developing an effective multithreaded execution model for such future massively parallel decentralized architectures. An important feature of CARE is the support for a multilevel fine-grain multithreading hierarchy. For example, the CARE model has two levels of threads: threaded procedures and strands. A threaded procedure is invoked asynchronously - forking a parallel thread of execution. A threaded procedure is statically divided into strands. A main feature of CARE is the effective synchronization between strands using only relevant dependencies, rather than global barriers. It also enables an effective overlapping of communication and computation, allowing a processor to execute any strand whose data and control is ready.

This paper provides an overview of the CARE execution and architecture model, explains how CARE supports explicit encoding of a partial order of strands – the scheduling quantum processed by the dispatch/issue section, and
introduces the concept of strand chaining used to take advantage of a sequential fetching/decode mechanism by fetching instructions in successive strands in order. We describe the CARE scheduling window and how it is designed to support the strand enabling mechanism for register-based dependencies – data, control and memory dependencies – through the CARE architecture mechanism.

In Sec. 1 we make the case for a decentralized control paradigm with the Quicksort Kernel. Section 2 introduces the Execution Model of threads, strand chains and strands. In Sec. 3 a brief overview of the compiler phases is presented. Section 4 explains the major architectural components with an emphasis on the register based synchronization mechanism. Next, Sec. 5 gives a cross section of compiler derived experimental data. Related work and conclusions wrap the paper up.

1 Motivation

Consider the control flow graph (CFG) for the modified Quicksort Kernel in three-address code as described in its original form in the “dragon book” [1] in Fig. 1. The code has two inner parallel loops LR2 and LR3 that traverse a list. The list has guard nodes on each end to check for list overflows – not shown. A node in the list has as members: (1) an element value, (2) an unique identification (id) that is totally ordered, (3) a pointer to the next node and (4) a pointer to the previous node. LR2 starts from the first element in the list and searches for an element that is smaller than a predetermined pivot element \( v \). \( v \) is the last element in the list. LR3 starts from the last element in the list and searches for an element that is larger than \( v \). These two elements are swapped in B5 and the search continues from where LR2 and LR3 left off by starting a new iteration of the outer loop LR1. Once the search spaces of LR2 and LR3 overlap each other, the outer loop LR1 exits. B6 performs a final swap between the pivot and the last fetched element in LR2. The kernel itself is invoked in a divide-and-conquer fashion – not shown – on the elements that were not traversed in the current call by their respective loops LR2 and LR3.

The main characteristics of this code are its small, independent inner loops LR2 and LR3 with unknown loop bounds. Each inner loop has two memory load instructions to fetch the address of a node and its corresponding value(4, 5, 7, 8). Memory is written in the swap instructions 13, 14, 16 and 17.