Efficient Implementation of Sequential Loops in Dataflow Computation

Boon Seong Ang

NE43-205, Laboratory for Computer Science,
Massachusetts Institute of Technology, Cambridge, MA 02139.
hahaha@abp.lcs.mit.edu

Abstract

The implementation of sequential loops in dataflow computation had traditionally not received very much attention as it was assumed that most loops would be executed in parallel. This assumption was valid for earlier dataflow machines such as the MIT Tagged Token Dataflow Architecture (TTDA)[2], Sigma-1[9] but not for the newest generation of dataflow machines including Monsoon[6], EM-4[11] and Epsilon-2[7]. On the latter machines, sequential loops use less memory, and can execute in fewer instructions, albeit with lower parallelism than the parallel versions. This characterisation of sequential and parallel loops suggests that programs should have parallel outer loops and sequential inner loops. The run time of sequential loops therefore become significant in the overall run time. We also found that previous implementations of sequential loops can incur fairly high overheads. In this paper, we present two new ways of implementing sequential loops that have lower overhead then previous methods. We studied this problem in the context of compiling Id[14, 15] for Monsoon.

1 Introduction

Loops executing on dataflow machines can be classified into three categories according to the amount of inter-iteration parallelism allowed. At one extreme is an *unbounded* loop, where there is no artificial bound on the number of concurrent iterations. Such a loop is compiled into a recursive procedure which together with non-strict procedure call convention lead to unrestrained unraveling of the loop. Next, we have what is often referred to as *k-bounded* loops[5], where the number of iterations executing concurrently is bounded to a value k which can be computed at run time. Finally, a *sequential* loop is one where one iteration of the loop has to complete before the next one executes.

The implementation of sequential loops in dataflow computation had traditionally not received much attention as it was assumed that most loops would execute in parallel, predominantly as k-bounded loops. This assumption was valid for earlier dataflow machines such as the MIT Tagged Token Dataflow Architecture (TTDA)[2] and Sigma-1[9] where the per-iteration cycle cost of executing a loop is the same whether the loop is k-bounded or sequential. The consumption of resources, in the form of token "tags" is also the same for both cases. For these machines, it is natural to execute loops in parallel to reap full benefit of parallel processing.

With the newest generation of dataflow machines (e.g. Monsoon[17, 6], EM-4[11] and the Epsilon-2[7]) the situation is different. These machines use *frame memory* on each processor to implement token-matching(see Section 2.1). This has two consequences for loops: (i) every *concurrent* iteration needs its own frame; (ii) sending data between two iterations that do not share (in a time multiplexed way) the same frame requires communication through the interconnection network. The former means that memory usage is lower for sequential loops while the latter suggests that they can execute in fewer cycles. Additional saving comes from fewer frame allocations and deallocations. The benefits come, however, at the expense of reduced parallelism. Thus there is a place for both k-bounded and sequential loops. Outer loops should be k-bounded for parallelism, while inner loops should be made sequential for efficiency reasons. Such a strategy places great important on the efficiency of sequential loops as the run time of inner loops often dominates the overall run time of programs.

Much as sequential loops can potentially be cheap, past implementations[8, 5] have not achieved the goal of keeping the costs down. In this work, we present two new ways of implementing sequential loops that incur less overhead then previous methods. We studied this problem in the context of compiling Id[14, 15], a non-strict, inherently parallel language, for Monsoon[6, 17]. The two implementations use slightly different execution models which are discussed in the paper.

We start by providing some background in Section 2 and proceed to describe some of the past approaches to implementing sequential loops in Section 3. In Section 4, we present a new implementation, the *self-gating* sequential loop schema, which implements a sequential loop in the pure dataflow model. We improve upon this schema by adopting a mixed dataflow-von Neumann execution model to produce the *frame-based-variable* sequential loop schema. This is presented in Section 5. Run time statistics of Id code compiled using the different implementation strategies are presented in Section 6. Conclusion follows in Section 7.

2 Background

This section provides a brief introduction to dataflow computation with emphasis on explaining loops. Dataflow com-