An Evaluation of Medium-Grain Dataflow Code

Walid A. Najjar, Lucas Roh, and A. P. Wim Böhm

Received June 8, 1993

In this paper, we study several issues related to the medium grain dataflow model of execution. We present bottom-up compilation of medium grain clusters from a fine grain dataflow graph. We compare the basic block and the dependence sets algorithms that partition dataflow graphs into clusters. For an extensive set of benchmarks we assess the average number of instructions in a cluster and the reduction in matching operations compared with fine grain dataflow execution. We study the performance of medium grain dataflow when several architectural parameters, such as the number of processors, matching cost, and network latency, are varied.

The results indicate that medium grain execution offers a good speedup over the fine grain model, that it is scalable, and tolerates network latency and high matching costs well. Medium grain execution can benefit from a higher output bandwidth of a processor and finally, a simple superscalar processor with an issue rate of two is sufficient to exploit the internal parallelism of a cluster.

KEY WORDS: Medium grain dataflow; basic blocks; dependence sets; performance evaluation; architectural parameters.

1. INTRODUCTION

The fine grain dataflow model has demonstrated its ability to expose maximal parallelism at function, loop, and instruction levels. This is accomplished through very general run-time instruction scheduling based on token matching. However, the overhead of token matching, which is a form of hardware synchronization, is incurred for every instruction. This overhead is especially visible in the execution of inherently sequential threads of

1 This work is supported in part by NSF Grants CCR-9010240 and MIP-9113268.
2 Department of Computer Science, Colorado State University, Fort Collins, CO 80523. (e-mail: najjar@cs.colostate.edu).
code. The dynamic instruction scheduling also tends to mask the inherent instruction level locality.\(^{(1,2)}\) Furthermore, the main benefit of the fine grain model, exposing the maximal parallelism to mask latencies, can become a double-edged sword: when the degree of parallelism is very large, it forces the machine to handle a large peak demand for resources (e.g., matching store) which exacerbates the resource management problem.\(^{(3)}\)

Various multithreaded and hybrid von Neumann/dataflow execution models have been proposed that can alleviate these problems.\(^{(1,4–10)}\) These medium/coarse grain execution models recognize the advantages of scheduling code at compile time rather than at run time. This is most often accomplished by increasing the task granularity from a single instruction to many instructions and statically scheduling instructions within the task. This simple paradigm shift coupled with explicit allocation of resources (e.g., use of explicit token storage\(^{(11,12)}\)) has many advantages: reduced overheads, improved performance through shorter critical path in sequential code, and exploitation of intra-thread locality.

Several methods that generate medium/coarse grain code for these execution models have been developed by various researchers.\(^{(1,13–17)}\) These methods generally fall into two categories. In the top-down strategy, the code is generated directly from a program data-dependence graph. In the bottom-up strategy, fine grain dataflow code is first generated, and is then partitioned into a set of tasks or threads.

In this paper, several issues related to medium grain dataflow execution are explored. The basis for our study of the medium grain dataflow execution model is the concept of clusters of instructions that become the schedulable units of execution. In this model, clusters can be seen as the dataflow equivalent of basic blocks in sequential programs. We describe two bottom-up algorithms that generate these clusters from fine grain dataflow code as well as a set of optimizations of the medium grain code. The two algorithms are the basic block algorithm\(^{(18)}\) and the dependence sets algorithm.\(^{(1,15)}\) We also look at certain characteristics of the medium grain code that we generate. We run our code on a variety of realistic machine models using a simulator to derive performance figures. The performance figures are used to identify bottlenecks and gauge the effectiveness of various architectural features in terms of scalability, tolerance to network latency, and cost/performance tradeoffs.

Our results indicate that, even with simple bottom-up code generation strategies producing relatively small grains (on average between two and four instructions per grain), the medium grain model outperforms the fine grain model under a wide range of machine models. Further, the results demonstrate that the medium grain model, by exploiting instruction level locality and hence reducing communication and synchronization over-