An Improved PTD Scheduler for MAP Architectures

D. K. Arvind and S. Sotelo-Salazar
Department of Computer Science, University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ, Scotland.

Abstract

The Penalise True Dependences (PTD) scheduler considered the effects of true dependences between successive instructions and contention for resources to better utilise the functional units in micronet-based asynchronous processors. This paper presents an improved version which considers dependences beyond successive instructions, and identifies clearly through subgraphing instructions which could be moved to reduce the effects. Performance results are presented where the improved PTD scheduler compares favourably against two well-known list schedulers - the Gibbons and Muchnick [4] and the balanced scheduler [5].

1 Introduction

Micronet-based asynchronous processor (MAP) architectures [1] consist of a network of functional units which operate concurrently and communicate asynchronously. The issue and execution of instructions consist of a sequence of micro-operations involving the Issue Unit, Operand Fetch Unit, the Register File and the appropriate Functional Units (Figure 1). An instruction is issued when its operands are available. It runs to completion unless it is stalled due to resource contention at any of the following points in the trajectory of the instruction: the read ports of the register files, the functional units, the write-back ports. The latencies of the instructions are not fixed (in contrast to clocked processors), but depend on a number of factors: instruction type, the data on which they operate, and the contention for resources which in turn depends on the mix of instructions. Data dependences between successive instructions introduce other delays in asynchronous architectures. In synchronous datapaths, for instance, we can predict exactly when the result of a previous instruction will be available in order to issue the following instruction. In the case of the micronet the issue unit will have to stall for a period of time until the result of the previous instruction is written back to the register file.

In scheduling instructions for a micronet-based target, we seek to minimise the effects of resource contention and data dependences in an environment where the latencies of the instructions are themselves not fixed but vary over a range. In a previous paper [2], we had proposed a method for scheduling instructions for MAP datapaths. The Penalise True Dependence (PTD) scheduler calculates a penalty measure which reflects the degree of resource contention and stalls due to data dependences. The scheduler addresses the problem by moving instructions around which would result in a legal schedule with a lower penalty measure. (Tables 1 and 2 give the range of latencies for the functional units and the penalty measures).

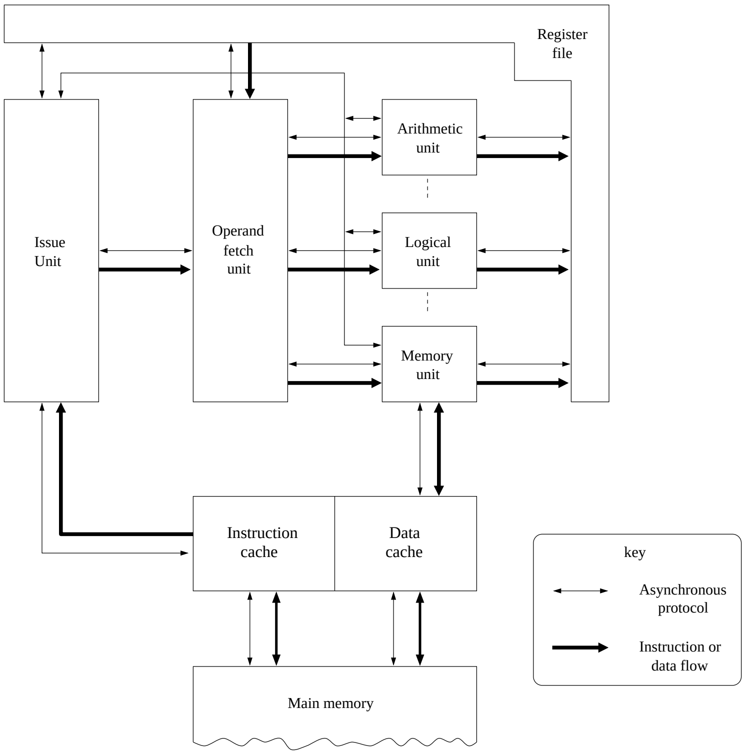

Figure 1: The MAP target architecture

This paper presents improvements to the PTD scheduler. Firstly, candidates for penalties are extended to include dependences beyond just successive instructions. Secondly, the basic blocks of instructions are subdivided into subgraphs to scope the candidates selected for moving instructions.

In the rest of this paper, the algorithm is described and its time complexity is derived, and performance results are presented where the improved PTD scheduler compares favourably against two well-known list schedulers - the Gibbons and Muchnick [4] and the balanced scheduler [5].

Component type	Minimum latency	Maximum latency
Issue Unit (IU)	1.00 ns	2.00 ns
Input buses	2.00 ns	4.00 ns
Output buses	2.00 ns	4.00 ns
Arithmetic Unit (AU)	4.00 ns	8.50 ns
Logical Unit (LU)	2.00 ns	7.00 ns
Memory Unit (MU)	10.00 ns	20.00 ns

Table 1. Latencies values for the target architecture.

Cases of dependences	Consecutive instructions	Separated by one inst.
True dependency with a load inst.	3	1
True dependency with a branch inst.	2	0
Resource dependency within mem. inst.	1	0
Normal true dependences	1	0

Table 2. Table of penalties for true data dependences.

2 An improved PTD scheduler

The objective of the scheduler is to minimise, where possible, the penalty measure for a given schedule of instructions within a basic block. The approach is a greedy one, whereby candidates for movement are chosen such that no new penalties are introduced. This guarantees that the penalty measure is always reduced after each movement. Consecutive instructions are assigned higher penalties as they can potentially result in larger stalls. The value of the penalty falls with the distance between the producer and consumer of the result. The maximum distance that we would need to consider is equal to the number of functional units which can potentially operate in parallel.

The value of the penalty also depends on the types of functional units involved. For example, the cost of a true dependency between a memory load instruction is higher than between a register one. Tables 1 and 2 illustrate the latencies of the different units and the respective penalties. The same idea is extended to penalising resource conflicts.

2.1 Complexity

The time complexity of the improved PTD scheduler is now $\theta(pnec+pnc+pn)$, where n is the number of instructions in the basic block, e is the number of penalties, p is the number of functional units in the architecture, and c is a small constant ($c = 2, 3, 4$). If we analyse the above expression, the complexity of the scheduler can be reduced to $\theta(ne)$. However, in general conditions, as the algorithm progresses the number of penalties is reduced and therefore n becomes bigger than e, which means that the complexity can be reduced to $\theta(n)$.

The upper bound, which is represented by a pure sequential code, is $\theta(n^2)$, with $e = n-1$ and $c = 2$. Conversely, the lower bound is represented by a pure independent code and is the order of $\theta(n)$, with $e = 0$.

2.2 Subgraphs

A basic block is composed of a group of instructions that are related in an ordered way which perform computation over data and which may be divided into subcomponents (subgraphs) that perform part of the overall computation. For example, two separate subgraphs would be the computation of an address and the data that would be loaded or stored in that address. An example of a directed acyclic graph DAG with two subgraphs is shown in Figure 2. The node numbering reflects the order in the schedule and the highlighted arcs denote penalties.

A reordering of instructions by moving instruction 5 in between instructions 2 and 3, and instruction 6 in between 3 and 4, resulting in the sequence "2 5 3 6 4", would improve the penalty measure. Any further improvement is restricted by the overlapping chains of dependences between 2 and 3, and, 5 and 6. Ideally, an unrelated instruction between 5 and 3 would further reduce the penalty measure. Dividing the basic block DAG into subgraphs identifies potential source of independent instructions which can be moved to a smaller search area. In the example, the subgraph on the right is a better prospect for independent instructions to move between 2 and 3, and, 3 and 4, and would not result in overlapping chains. In practice, there is a greater probability of finding independent instructions from other subgraphs.

The selection and size of the subgraphs deserve attention. If the size is too small, then there is a greater chance of producing overlapping chains. If the size is too large, then the advantages of the subgraphs are diluted.

The granularity concern for the scheduler can be exemplified in the DAG example in Figure 2. If we choose a subgraph formed from nodes 1, 2, 3 and 4, and another subgraph from nodes 5, 6, 7, 8 and 9, the scheduler would try to intermix the penalties marked and this will end with overlapping chains.

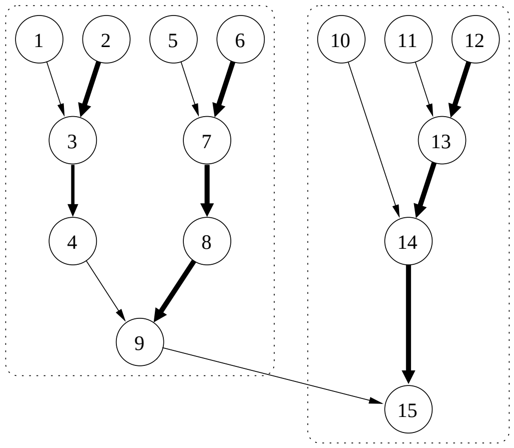

Figure 2: A basic block decomposed by two subgraphs.

The selection of subgraphs is based on the number of predecessors of each instruction and the ratio between the number predecessors and the height from its leaves. The number of predecessors is needed to indicate the size of the *DAG* at the point and the ratio between this parameter and the actual height is a rough measure of the potential parallelism in that part of the *DAG*.

3 Results

The improved PTD scheduler was compared against two other local schedulers, the *balanced* scheduler [5] and the original *list* scheduler from Gibbons and Muchnick (GM) [4]. In all the cases the scheduling was performed before register allocation and were tested over a set of benchmarks. An event-driven stochastic simulator was used to simulate them. The target architecture had one memory unit, one arithmetic unit, one logical unit and one branch unit.

The set of benchmarks chosen was the set of Livermore loops [3] (few basic blocks), and a set of control-intensive programs with a larger number of small basic blocks. Figure 3 depicts the comparison in performance for the three schedulers. The results presented are the average of five simulations for each benchmark.

The improved PTD scheduler consistently outperforms the other schedulers for the two set of benchmarks. In the case of `Loop7`, we see the detrimental effect of overlapping chains which are located in the critical path of the basic block in spite of subgraphing.

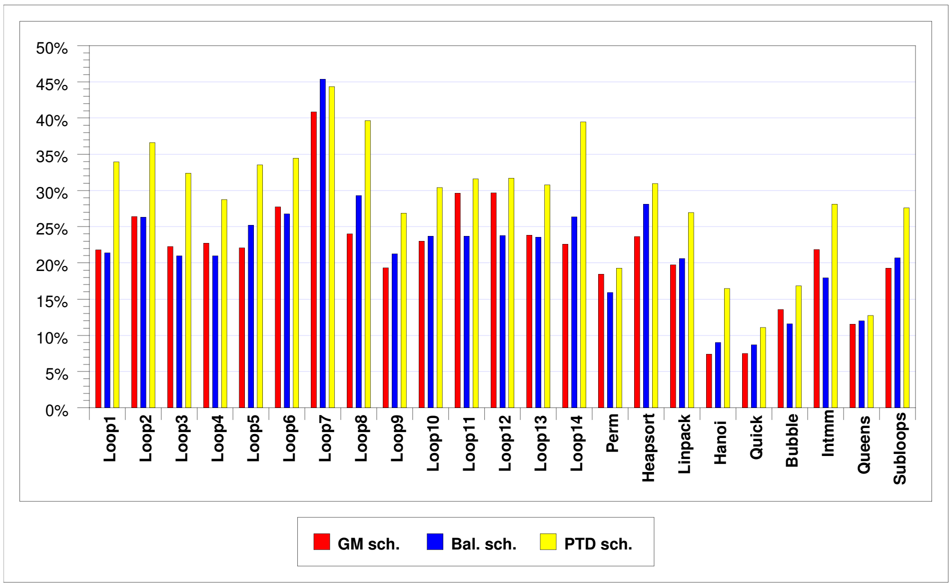

Figure 3: Performance comparison between the three schedulers.

4 Conclusions

The results from the simulations show that the PTD scheduler produces better quality schedules and has a lower time complexity than the list and balanced schedulers (The complexity of the list scheduler being $\theta(n^2)$, and $\theta(n^2\,\alpha(n))$ for the Balanced scheduler). The potential limitation is the introduction of overlapping chains, but in most cases this can be avoided by dividing the basic blocks into subgraphs.

Acknowledgements

We would like to thank the members of the MAP group for useful discussions. S. Sotelo-Salazar was supported by a postgraduate studentship from the Science and Technology National Counsel in Mexico (CONACYT).

References

[1] D. K. Arvind and V. E. F. Rebello. Instruction-level parallelism in asynchronous processor architectures. *Proc. 3rd. International Workshop on Algorithms and Parallel VLSI Architectures*, Leuven, Belgium, August 1994, pp. 203-215.

[2] D. K. Arvind and S. Sotelo-Salazar. Scheduling Instructions with Uncertain Latencies in Asynchronous Architectures. *Proc. 3rd. International Euro-Par Conference*. August 1997, pp. 771-778.

[3] J. Feo. An analysis of the computational and parallel complexity of the Livermore Loops. *Parallel Computing* Vol. 7, 1988, pp. 163-185.

[4] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *Proc. SIGPLAN 1986 Symposium on Compiler Construction*, SIGPLAN Notices, 21(7), July 1986, pp. 11-16.

[5] D. R. Kerns and S. J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. *In ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, 28(6), June 1993, pp. 278-289.