Limits on ILP in Micronet-based Architectures

D. K. Arvind and C. Keepax
Institute for Computing Systems Architecture
School of Informatics, The University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ, Scotland.
Email: dka@inf.ed.ac.uk

Abstract

This paper describes the use of simulations to study the limits on Instruction Level Parallelism (ILP) in a micronet-based asynchronous processor. The impact of two features on the exploitable ILP were studied: the dependency lengths between instructions in the program, and the asynchronous synchronisation overhead in the architecture. The results demonstrate that the attenuation in the speedup due to ILP is moderate with increases in the synchronisation overheads (of upto 50% of ALU computation cost), thanks to the overlapping of communication and computation inherent in micronet architectures.

1 Introduction

The micronet [dka94] is a network of entities which compute concurrently and communicate asynchronously. This paper explores the impact of dependency lengths and synchronisation delays on the speedup due to ILP in multiple ALU micronet architectures. Section 2 gives an overview of the Micronet-based Asynchronous Processor (MAP) datapath, Section 3 describes the COMPASS design environment which was used to derive the results of the experiments described in Section 4.

2 The MAP Datapath

MAP is a family of processors based on the micronet model for the design of asynchronous architectures. The MAP datapath, as shown in Figure 8, is a *network* of execution units, such as the Instruction Buffer, the Instruction Issue Unit, Control Unit, Register Bank, and integer functional units such as the Memory Unit and the ALUs. The 32-bit datapath executes a subset of the MIPS instruction set. It contains sixty-four, 32-bit general-purpose integer registers, and a collection of dedicated registers for the program counter, link register (for the return address of a function), stack pointer (which points to the area in memory for the spilt variables), and HI and LO registers (used for storing the results of 32-bit multiplication). Each execution unit is composed of a *Functional Microagent (FM)* which executes a specific micro-operation, and communicates with other units via *Communicating Microagents (CM)* employing a four-phase handshaking protocol. The Instruction Buffer caches instructions for the Instruction Issue Unit, and the Control Unit mediates their operations. On the issue of an instruction, the appropriate control signals are asserted and the destination register is locked, and the instruction is considered issued once the signals have been acknowledged. The CMs between the register bank and the X and Y buses communicate to place the register values, as specified in the instruction, on the X and Y buses, respectively. The ALU will operate on the values and the results are placed on the Z-bus and written back to the destination register, which is then unlocked.

3 The COMPASS Design Environment

The COMPASS Design Environment integrates two major modules - a compiler based on SUIF which generates assembly code of ANSI C programs for the MAP target, and an object-oriented modelling and simulation environment for MAP architectures, which executes this code. COMPASS therefore supports the exploration of both the architectural and compiler optimisation spaces in a unified environment.

A C program is compiled into the SUIF intermediate representation using *C2suif*. Compilation passes are written to operate on this intermediate representation and program optimisation is achieved by running a series of different passes. Machine SUIF provides target-specific functionality and is built on top of SUIF and encapsulates MAP specific elements which are necessary for optimisations, such as instruction scheduling. Machine SUIF is built from a selection of libraries, each of which can be extended or mod-

ified to implement different architectural forms or compilation passes. The most important one is the Machine library, which specifies the Machine SUIF intermediate representation known as SUIF Virtual Machine (SUIFVM). The SUIF intermediate representation is lowered into a SUIFVM representation before executing MAP specific optimisations. SUIFVM represents entities at a lower level and is closer to the way the MAP architecture actually represent instructions. Rather than representing a *for* statement directly as SUIF does, SUIFVM would represent it as a block of instructions (the body of the loop) and a conditional branch (the terminating condition). As in SUIF, a series of passes is used in the Machine SUIF compilation process: two passes are used to lower the SUIF intermediate representation into SUIFVM, followed by a second lowering to the target MAP machine library. MAP specific optimisation passes are next run, followed by register allocation and code finalisation, such as stack layout. Finally, a printer object outputs the program in an assembly form for execution on the instruction level simulator of the MAP architecture.

The core of the bespoke simulator is the SPAM kernel which controls the order of events to be simulated and keeps track of the timing. The Memory and Sim components in the simulator are built on top of the SPAM kernel. The output of the Assembler fills the Memory with instructions, which is interpreted by the objects which model the processor behaviour in the Sim component. They inherit the functionality of the *Entity* class, and attached to the entities are ports which allow different entities to send and receive *events*. The events correspond to actions in a processor, such as the value of a register being read. The *entity* class inherits from the *context* class, which ensures that the entity keeps processing information, and the *entity* class handles timing and ensures that events get to the right place at the right time.

4 Experiments and Results

We investigated the influence of two factors, one software and the other hardware, which would affect the performance of programs executing on micronet archiectures. The first factor was the dependency lengths[1] of the instructions in the program. This is an important issue in the case of MAP architectures which have a number of ALUs to exploit instruction level parallelism. If the mix of instructions is dominated by ones with small dependency lengths, then there is a greater likelihood of the Instruction Issue Unit being stalled due to these dependencies. This would result in a low utilisation rates for these functional units, and a lower speedup[2]. The second factor was the delay of the Communicating Microagents (CM) and how this influenced the speedup. One of the features of the micronet model was to overlap the communication with the computation and thereby ameliorate the overhead of asynchronous communication. We wished to assess in a quantitative manner the influence of increasing the CM delay on the overall performance. We used synthetic benchmarks which were straight line code in which the dependency lengths between the instructions were uniform across the instructions, and ranged from 1 to 5 or was infinite, i.e. the instructions were independent. Figures 1, 3 and 5 illustrate the speedup for the benchmarks in the case of 2, 3 or 4 ALUs, where the ALU delay was set at 20 Time Units (TU), and the CM delay ranged from 1 to 19 TU. Figures 2, 4, and 6 illustrate the speedup for the same parameters listed previously, except that the ALU delay was set at 120 TU.

The best speedup corresponded to those cases where the dependency length is infinite and the CM delay is the lowest. As the CM delay is increased from 1 TU to 19 TU, an increase of 1800%, the speedup in the case of 4 ALUs - ALU delay of 120 TU (Figure 6) is attenuated by about 40%, compared to 62% in the case of ALU delay of 20 TU (Figure 5). A comparable reduction of 63% in the speedup is experienced in the case of 2 ALUs - ALU delay of 20 TU (Figure 5), compared to 22.5% reduction in the speedup in the case of the ALU with 120 TU delay. In the case of 1 ALU, the reduction in speedup is 22.5% (Figure 1) and 5% (Figure 2), respectively.

In summary, the performance is more tolerant to increases in the synchronisation costs, especially in cases where the CM delay is less than one-tenth of the ALU delay (Figures 2, 4 and 6), which reflects the relative implementation costs in practice.

Figure 7 illustrates the speedup for the inverse discrete cosine transform. The implementation of this algorithm is characterised by basic blocks of average size of 31 instructions, of which 81% are ALU instructions, and 19% memory instructions. 93% of the instructions have dependency lengths of either 1 or 2 instructions. There is a reduction of 21.5% in the speedup with 4 ALUs (delay 0f 120 TU) between the two extreme cases of CM delays.

References

[dka94] Arvind, D. K. and Rebello, V. 1994. *Instruction Level Parallelism in Asynchronous Processor Architectures* In *Proc. of the 3rd International Workshop on Algorithms and Parallel VLSI Architectures*, pp 80-91, Eds. M, Moonen and F. Cathoor, Leuven, Belgium, Aug. 1994, Elsevier.

[1] The dependency length of any two instructions within a basic block in a program is defined as the shortest distance (in terms of number of instructions) between the producer and the consumer instructions.

[2] The speedup is defined as the ratio of program execution times on one ALU to many ALUs

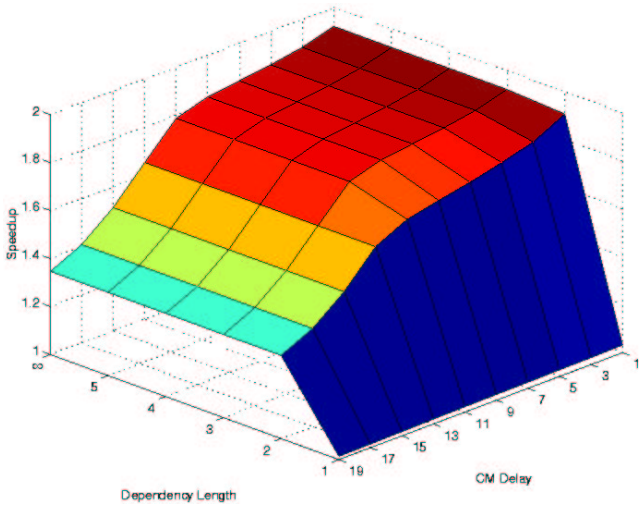

Figure 1. Speedup with 2 ALUs (delay 20 TU)

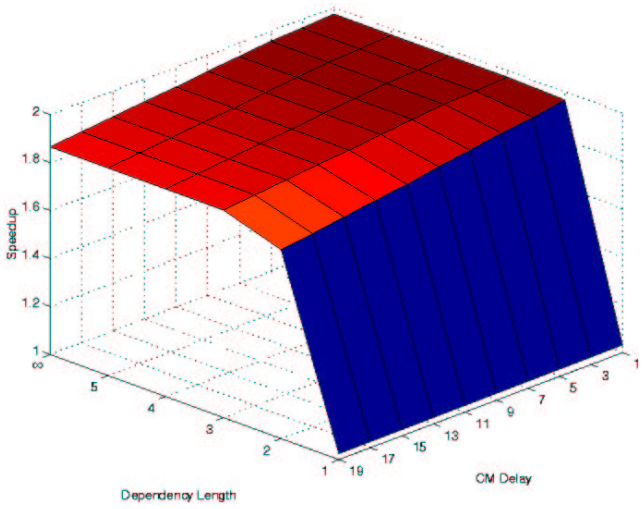

Figure 2. Speedup with 2 ALUs (delay 120 TU)

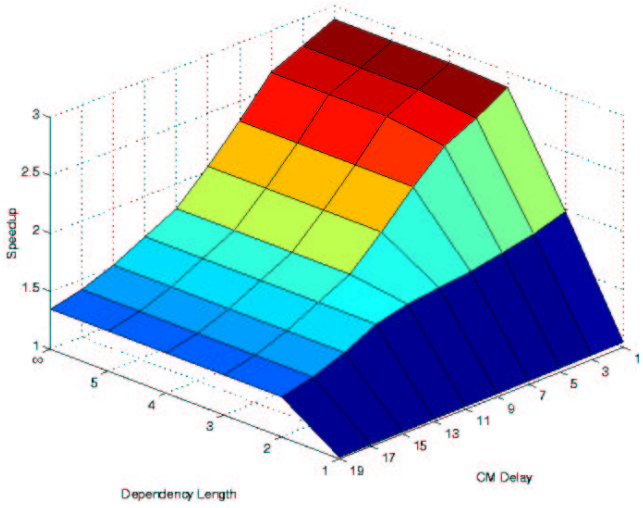

Figure 3. Speedup with 3 ALUs (delay 20 TU)

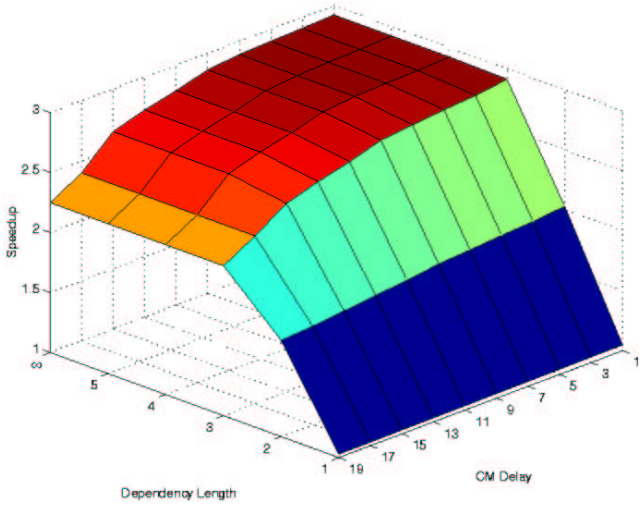

Figure 4. Speedup with 3 ALUs (delay 120 TU)

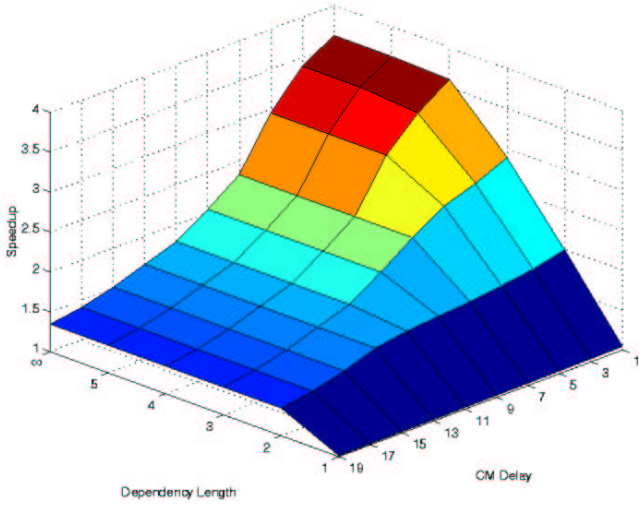

Figure 5. Speedup with 4 ALUs (delay 20 TU)

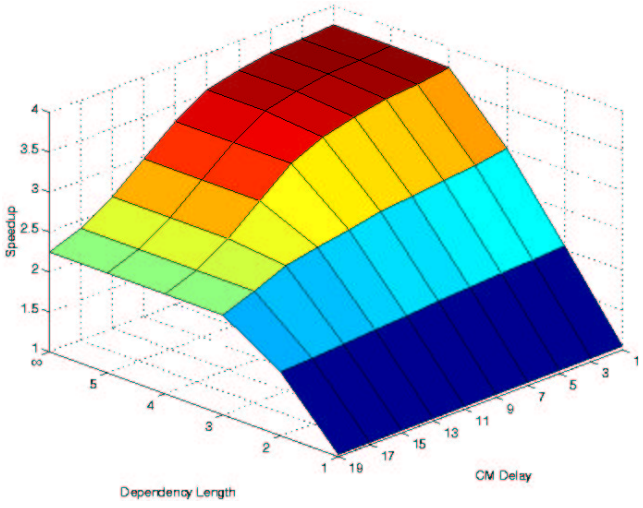

Figure 6. Speedup with 4 ALUs (delay 120 TU)

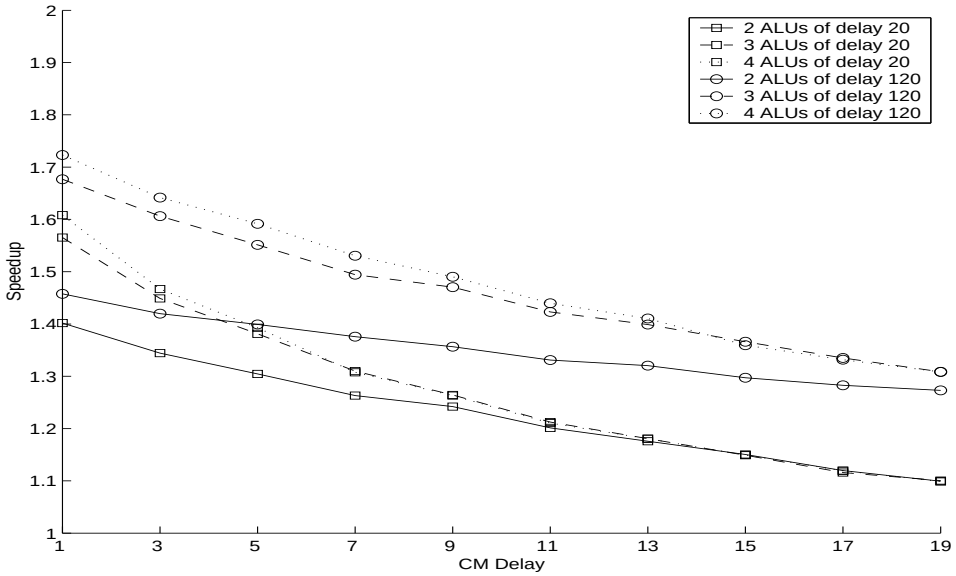

Figure 7. Attenuation in Speedup for the Inverse Discrete Cosine Transform

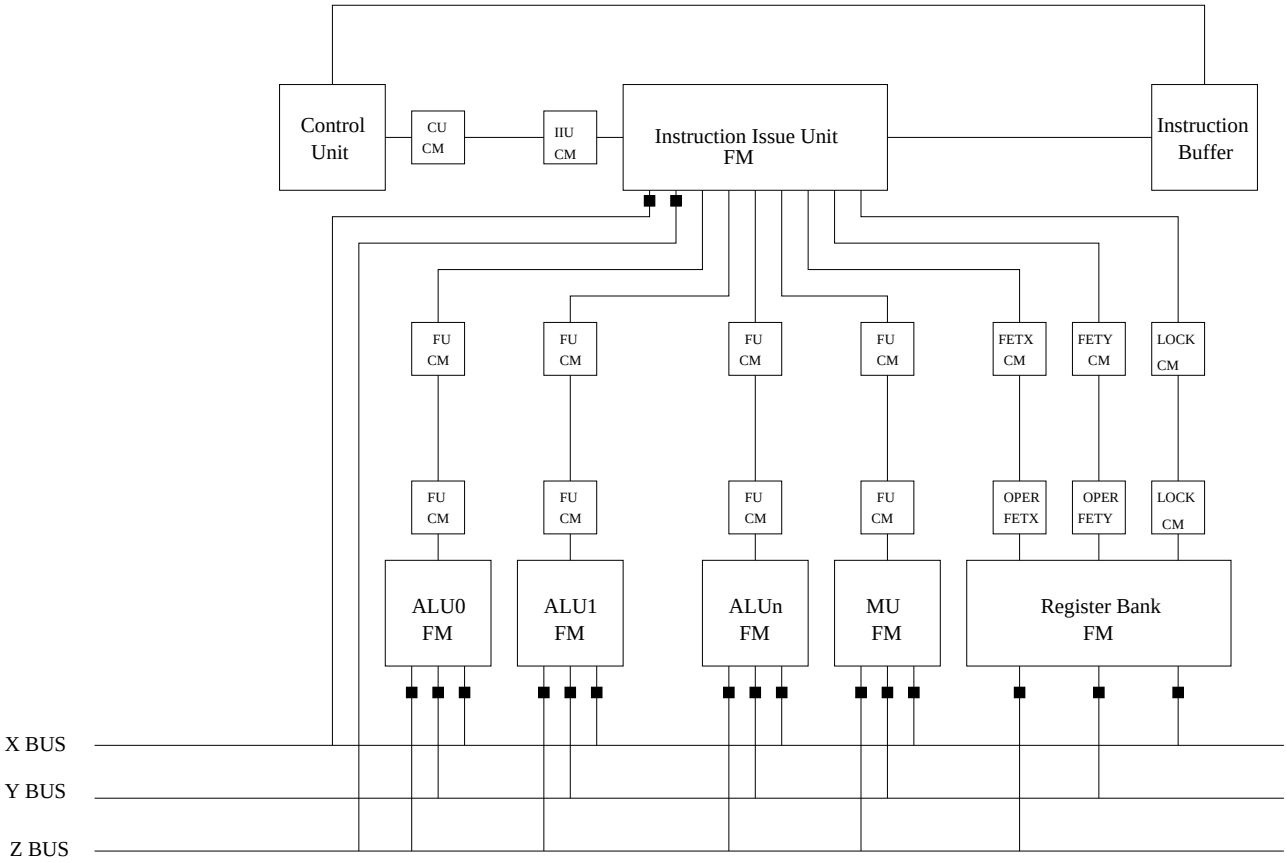

Figure 8. The MAP Datapath