Computer Systems Group

Design Issues for Latency Hiding on an Access Decoupled Machine

by

Graham P. Jones and Nigel P. Topham

CSG Report Series **CSG-33-97**

Computer Systems Group **November 1997**

Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

Design Issues for Latency Hiding on an Access Decoupled Machine

Graham P. Jones * and Nigel P.Topham †

Technical Report CSG-33-97

Department of Computer Science

University of Edinburgh

November 9, 1997

Abstract: Future software and hardware technologies will try to provide improved performance by extracting higher levels of parallelism. However the cost of a main memory access - in terms of missed instruction slots - increases with faster processors and greater issue widths. For this reason latency hiding technology remains one of the most important parts of high performance processor designs. In this paper we investigate a latency hiding technique known as Access Decoupling which partitions a program into two separate instruction streams in order to aggressively prefetch data.

We justify a renewed interest in Access Decoupling in two ways. Firstly as a latency hiding technique and secondly as a solution to the problem of hardware complexity in large issue width, out-of-order superscalar architectures. We show that in comparison to a single instruction stream architecture Access Decoupling is marginally more effective at hiding memory latency and capable of achieving higher performance through its simpler design.

After providing our justification for renewed interest in the decoupling paradigm we quantify the performance impact of different hardware/software design issues on Access Decoupled machines. We consider the effect of restrictions imposed by data dependency analysis, renaming, memory reordering, operation reordering, issue width and

synchronisation points on IPC and latency hiding effectiveness.

Keywords: Access Decoupling, Design Issues, ILP, Latency Hiding, Superscalar, Out-of-Order Execution

1 Introduction

Access Decoupling is a latency hiding technique which partitions a programs - statically or dynamically - into two separate instruction streams in order to aggressively prefetch data. The instruction streams are loosely coupled with one stream, executed on an Address Unit (AU), prefetching data for the second stream, executed on a Data Unit (DU). Provided the two streams can *slip* sufficiently memory accesses can be pipelined to tolerate large memory latencies.

In this paper we justify a renewed interest in the use of Access Decoupling by looking at the benefits afforded by using this form of asynchronous data prefetching. It is our belief that decoupling could be used to address two major problems in high performance out-of-order (o-o-o) superscalar design:

- *The need to hide relatively longer memory latencies*

  A consequence of larger issue widths and the increasing gap between memory and processing speeds is the higher performance hit of off-chip accesses. As processors aggregate dispatch rates increase the future trend will be to provide more aggressive prefetching techniques to hide memory latency. Access Decoupling, which prefetchs data as aggressively as the run-time state of the machine permits, is a natural choice.

- *The increasing hardware complexity of superscalar architectures*

*email: gxj@dcs.ed.ac.uk
†email: npt@dcs.ed.ac.uk

For superscalar architectures to provide greater performance the instruction widths and reordering windows will have to increase. This will require greater hardware complexity for instruction selection and wake-up logic in the instruction window. This complexity will increase delays in the timing for issuing instructions and affect processor clock speeds. A recent study on window logic complexity has shown that delays increase quadratically with instruction window size and issue width size [13].

One solution is to use separate "microclusters" as in the Alpha 21264 [9] each with it own register file and function units. Microclusters can share a single or have separate instruction windows. Access Decoupling, with its dual instruction streams, is a variant of this solution. It offers the benefit of smaller reordering window size and issue width coupled with the dynamic reordering of operations between the AU and DU.

In section 6 we demonstrate the performance payoffs between using a single instruction stream (SIS) and dual instruction stream (DIS). The results show that although the SIS achieves higher instructions per cycle (IPC) on average it requires larger instruction windows and issue widths.For these reasons we believe there is a need to reexamine the design issues for an Access Decoupled machine.

This paper is part of a larger study into the use of the decoupling paradigm for high performance systems. The experimental methodology behind this study has been to move from an ideal position, in which resources are unlimited and dependencies can be resolved perfectly, to more realistic case studies. This paper represents a selection of those case studies. In section 4 we identify those hardware/software *design issues* we believe are critical to the performance of the decoupled machine. In section 7 we explore the parameter space for each one of these issues.

We also discuss the *latency hiding effectiveness* and *scalability* of Access Decoupling. We define the latency hiding effectiveness as $E = T_{perfect}/T_{actual}$ where T_{actual} is the execution time for our Access Decoupled machine and $T_{perfect}$ is the execution time for a machine with perfect latency hiding in which each memory access perceives a single cycle latency. We define the scalability of access decoupling as the variation of latency hiding effectiveness with issue width.

In section 2 we discuss some of the previous work on access decoupling and the microclusters. In section 3 we outline our Access Decoupled architecture and in section 5 we describe the benchmarks and the simulation technique used in the experiments. In the final section we discuss the findings in the paper and the future work it suggests.

2 Background

Access decoupling is an asynchronous data prefetching technique which tries to hide memory latency by overlapping computation and memory access operations. Central to all decoupled machines [2, 5, 12, 19] is an architecturally visible address unit (AU) and data unit (DU); these units are responsible for performing, respectively, the memory accesses and data computations in a program. They each have their own program counter allowing the AU to run ahead of the DU. The degree to which the AU is ahead of the DU is called the *slippage*. The units communicate with each other and with memory via queues.

The early decoupled machines like the ZS-1 [5] and PIPE [12] differed in how they split the instruction stream. The ZS-1 had a single instruction stream with a splitter whereas PIPE had separate instruction caches for the access and execute unit. The ZS-1, unlike PIPE, also included a data cache. More recently decoupled machines like the DAE [2], MISC [17] and ACRI [4] have appeared. To increase slippage DAE includes specialised hardware for efficient address generation. This hardware is effective at reducing DU stall time and increasing cache utilisation. The MISC [17] architecture, derived from PIPE, has four asynchronous units each with their own instruction cache but common data cache. The ACRI machine included an additional control unit responsible for computing conditional branches and dispatching instructions to the AU and DU.

Decoupling has gained currency in superscalar architectures like the MIPS R10000 [20]. The R10000 is able to support a decoupled mode of operation through o-o-o execution and a separate access instruction queue. The R10000 can decouple address and execute operations even though there is no architecturally visible AU and DU.

3 The Access Decoupled Machine

The Access Decoupled machine modelled in our experiments is shown in figure 1. The machine is based on previous decoupled architectures like the ZS-1 and PIPE. The machine consists of two separate out-of-order (o-o-o) superscalar processors, the Address Unit (AU) and the Data Unit (DU), responsible for executing the access and data operations. The extent to which access and data operations may be reordered is determined by the hardware and software configurations discussed in section 4.

Each unit has is own separate instruction window, functional units and register files. The units can pass results to each other via a bypass mechanism. The number of instructions issued per cycle is determined by the issue width. We refer to the sum of the AU and DU issue width and the *combined issue width* (CIW).

The *decoupled memory* lies between the AU and DU and the rest of the memory system. The decoupled memory receives addresses from the AU and sends them to the memory system. When a referenced value is returned the decoupled memory buffers the value until it is requested by the DU. Requests from the decoupled memory take a single cycle. AU self loads are executed in a similar way. Previously the decoupled memory has been implemented through the use of queues [2, 12, 5]. Part of the reason for our work is to consider if this is the most suitable structure for asynchronous data prefetching when memory operations may be executed o-o-o.

The basic memory system consists of the main memory but may also be composed of first or second level caches. We are not concerned with a detailed simulation of the memory system;

instead we model its execution by considering every access to have a fixed cost. The fixed cost we refer to as the *memory differential* (MD). The memory differential is the difference in time between a register and memory system access. The purpose of all latency hiding techniques is to eliminate the any perceived memory differential.

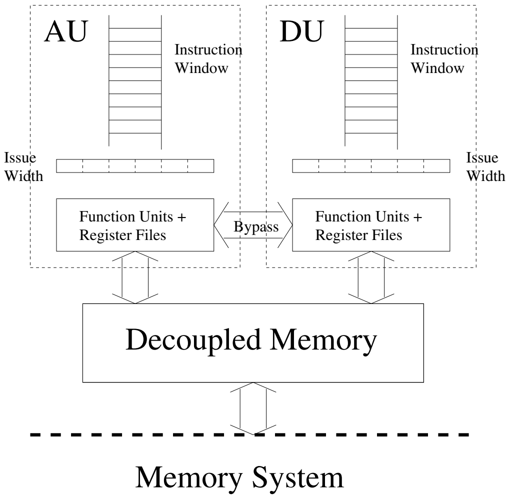

Figure 1: Access Decoupled Machine

4 Hardware and Software Design Issues

In this section we describe the hardware and software configurations considered in this paper. Our aim is to identify those features of hardware and compiler technology which enhance the the latency capability of an access decoupled machine. Previous work has already looked at the effect of queue sizes on a decoupled machine [5]. For this reason we chose to investigate hardware and software issues which effect the ordering of operations and hence the degree of slippage between the AU and DU. The modality of our architectural model are dependency analysis, renaming, memory reordering and instruction reordering scope.

- **Data dependency analysis**. There has been much research into the importance of precise inter and intra procedural data dependency analysis for program performance [10, 14, 15]. In this study we examine the lower and upper bound for any data dependency technique. We refer to these bounds, respectively, as *conservative* analysis (CTA) and *perfect* analysis (PFA). The difference between them lies in how they handle array references

  - In PFA dependency arcs exist between references to the same element of an array.
  - In CTA dependency arcs exist between any pair of references to the same array.

- **Renaming**. False dependencies occur because of the imperative and sequential nature of some programming languages. These languages allow the programmer to reuse the same memory location. All false dependencies can be removed by the introduction of new variables e.g. by array privatisation [7]. In our experiments we examine the behaviour of Access Decoupling without and without renaming.

- **Memory ordering**. The order in which memory accesses are sent to memory effects the amount of reordering that can take place between memory references. The three memory ordering schemes we consider are :-

  1. Strong Ordering (STO) : Loads and Stores occur in program order there is no reordering of memory operations. This is the simplest case requiring no additional hardware or software complexity.
  2. Semi Strong Ordering (SSO): Loads/stores preserve their program order with other loads/stores. Loads and Stores can reorder relatively to each other. In order to preserve flow dependencies this ordering scheme requires additional hardware for dynamical memory disambiguation [1]. This hardware is simplified by the SSO because the logic need only compare a load address against all stores issued since the last load.

[1] Preserving false dependencies is contingent on the use of memory renaming

3. Weak Ordering (WKO): Loads and stores can reorder with any loads and stores in the program, but loads and stores to the same location occur in program order. This is requires the most complex complex hardware. The logic must be capable of comparing all loads against all issued but not yet completed stores.

- **Reordering Scope**. The compiler may restrict the reordering that takes place between operations by inserting barriers into the instruction streams. All instructions occuring prior to a barrier complete before the barrier complete and all instructions after the barrier must be issued after completion of the barrier. We refer to the distance between two barriers as the reordering scope. We consider 7 cases

  RS0 : No out of order execution.

  RS1 : Operations can reorder between different iterations of leaf level loops but not across procedure call boundaries. Operations outside leaf level loops can reorder within basic blocks.

  RS2 : Operations can reorder between different iterations of leaf level loops. Operations outside leaf level loops can reorder within basic blocks.

  RS3 : Operations can reorder between different iterations of nested loops but not across procedure boundaries. Operations outside nested loops can reorder within basic blocks.

  RS4 : Operations can reorder between different iterations of nested loops. Operations outside nested loops can reorder within basic blocks.

  RS5 : Operations can reorder within the same procedure.

  RS6 : No restrictions of reordering.

- **Synchronisation points**. The barriers placed in the AU and DU instruction streams can be either synchronisation or non-synchronisation barriers. Both barriers enforce the normal barrier semantics within an instruction stream. However the synchronising barrier imposes an additional constraint that the AU must wait on the DU before issuing any further introductions. This barrier introduces false loss of decouplings (LOD) events into the program. Non-synchronising barriers allow the AU to retain some slippage ahead of the DU.

5 Benchmark and Simulation Technique

5.1 Simulation Technique

In our experiments we used a technique adapted from work by Petersen [14] and described in [11]. The technique works by annotating the source level code with calls to routines within the architecture simulator. *Shadow variables* are inserted into the program to track the earliest time that program values become available. Each program variable has an associated shadow variable to track it throughout the execution. Shadow variables are passed as arguments to the simulator to enable operation start times to be computed. Simulation of the Access Decoupled machine can then be performed by executing the annotated program [2].

The benefit of using this approach is that the program can be simulated at the source level. This means we can concentrate on the high level semantics of access decoupling without bringing in issues of assembly code generation. It allows us to simulate the effect of data dependency techniques, renaming and reordering scope whilst remaining independent of any particular native compiler.

The splitting of the code between the AU and DU was performed statically by the partitioning algorithm in the OCTAVE compiler [16]. This compiler assigns each node in the data flow graph to one or both units. This information is then used to annotate the source program.

[2]Correctness of the annotated code was verified by comparing simulated execution times with results from Petersen's tool [14]

Source level operations are translated at runtime into single instructions for the architecture simulator and executed on the appropriate unit/s. Loads and Stores are executed as one instruction on each of the units. Integer and address computations have a I cycle cost. Floating point operations take 5 cycles to complete. There is no speculative execution but enforcing control dependencies for loop closing branches is dependent on the reordering scope used (see section 4).

In our simulations we assume unlimited resources for instruction window size, decoupled memory and register files. We limit the instruction issue width to realistic and projected future values. The configurations we have discovered to be most suitable for maintaining maximal throughput are (1,1),(2,3) and (4,5) [**?**]. The first and second value in the brackets are respectively the AU and DU instruction issue widths.

5.2 Benchmark Programs

We chose a selection of 7 scientific Fortran programs from the PERFECT club suite [6] as our benchmark applications. These were chosen because they represent real applications from the scientific community.

Rather than execute each program in full, which would have been prohibitively expensive, we adopted a *sampling* technique. We executed each program in full counting the number of AU, DU and decoupled load operations during fixed intervals throughout the program. This enabled us to build a run-time profile of the operations executed in each program. From this profile we were able to isolate repeating region from which a representative sample could be identified. It was then possible to arrange for the simulator to switch on and off at the beginning and end of these sampled regions. In this way we were able to simulate selectively without having to simulate the program in it's entirety.

We selected benchmarks from the PERFECT club to represent varying degrees of vectorization and also to span known degrees of decoupling. Table 1 shows the seven selected benchmarks. This table gives the reported proportion of vectorised operations (VO) obtained from [18] and the decoupling efficiency (DE) obtained from [16]. The other columns show the number of AU and DU operations, decoupled loads, do loops and while loops executed in the program. The table also shows the code expansion due to duplication of operations on the AU and DU.

Program	VO	DE	Operations (10^6)			Expansion	Loops (10^3)	
Name	(%)	(%)	AU (%)	DU (%)	Loads (%)	(%)	while	do
ADM	43	69	36.5 (51)	34.8 (49)	13.6 (19)	6	0.02	1.1
DYFESM	69	77	19.1 (53)	16.8 (47)	10.6 (29)	3	0	1.1
FLO52Q	92	82	28.0 (54)	24.0 (46)	13.9 (27)	3	0	1.0
MDG	88	92	52.5 (54)	44.1 (46)	20.1 (21)	3	0	5.9
QCD2	4	19	52.7 (55)	43.1 (45)	12.7 (13)	12	0	2.8
TRACK	14	14	9.6 (65)	5.2 (35)	3.2 (21.5)	9	8.5	0.7
TRFD	70	99	53.0 (51)	50.4 (49)	31.6 (31)	2	0	2.1

Table 1: Benchmark Programs from PERFECT club suite

6 Single versus Dual Instruction Streams

Some critics may argue that the level of prefetching of an Access Decoupled machine could be achieved with an o-o-o superscalar architecture. If this is the case why introduce the additional complexity of asynchronous instruction streams. The answer we believe lies in the major difference between the performance and hardware complexity of a DIS and SIS. We believe that smaller window size and issue widths offered by decoupled architectures will allow designers to simplify the instruction window logic, reducing the clock period. In this section we quantify the instructions per cycles, the latency hiding effectiveness and window size required for the cases of a single and dual instruction streams. The results show that the reduction in window size is considerable for a small reduction in the IPC.

In order to compare the DIS and SIS we executed the same number of operations on both systems. Although there is some duplication of operations in the case of DIS the increase is on average small (5.4%) and therefore could be ignored [3]. The programs were executed with perfect data dependency analysis, renaming, weak ordering and non-synchronising barriers.

Tables 2 and 3 show the average IPC measured for the 7 programs when the reordering scope was RS3 and RS6 respectively. The tables also show the reduction in IPC for the DIS and the latency hiding effectiveness of the two systems. The column labelled CIW is used to denote the combined issue width for DIS and the issue width for SIS. We did not consider any other cases than when the DIS combined issue width is equal to SIS issue width.

It can be seen that in all cases the DIS IPC is lower than the SIS. The reduction in performance is due to inflexible scheduling during *synchronising* and *startup* phases. When in the DIS system the AU is waiting on a result from the DU it's issues slots are being wasted. However, for the SIS the full issue width is available at all times. During the startup phase after a synchronisation point the SIS is able to initiate more accesses whilst the DU waits for the first value to be returned from the memory system. We would expect therefore that as the number of synchronising points decreases the reduction in IPC would be smaller. This can be seen in the difference between tables 2 and 3 where for a CIW of 9 and memory differential of 0 the reduction in IPC has dropped from 9% to 6%. This is due to RS3 reducing the slippage between the units and hence increasing the number of synchronising points in the program.

The difference in IPC also varies with larger CIW. This is due to the fact that when the CIW is 2 the AU and DU have an issue width of 1, effectively serialising the code. Some reordering will be possible but it will be small. When the CIW is increased to 9 the improvement in performance due to the extra issue slots available during synchronisation and startup phases is relatively smaller.

The tables also show that the latency hiding effectiveness is always marginally smaller for the SIS. We believed this is due to the AU having to contend with the DU for issue slots when both units are busy.

[3] In future work we plan to consider the impact of code expansion on our results

A large memory differential also reduces the performance difference between DIS and SIS. Large memory differentials reduce the parallelism in the program and limit the advantage gained by the SIS system during synchronising and startup phases on the DIS.

In our simulations we allowed the instruction window to have unlimited resources. We can however compute the average window size needed to achieve the IPC levels and latency hiding shown in tables 2 and 3. The average window size is estimated using the equation $WS = (MD + 1) * (IDPC - IPC) + IDPC$ (see figure 2). For the DIS the window sizes are shown for both the AU and DU window size. The final column shows the difference between the window sizes for the SIS and DIS. The difference is computed by subtracting SIS window size from the maximum of the AU and DU window sizes. Positive figures denote a reduction in window size. It will be noticed that in only one case, when the CIW=2 and RS=6 is the average DIS instruction window size greater than the SIS window size. The results clearly demonstrate that for the large CIWs (5 and 9) the reduction in window size is between 16% and 44%. Interestingly, the reduction in window size increases with larger issue width. For a CIW of 9 the reduction is approximately 44% for both RS3 and RS6.

The execution time of a program is given by the product of the number of operations, the IPC and the clock speed. Research has shown that delays due to window logic increase quadratically with instruction window size and issue width [13]. We would anticipate that the 9% to 5% reduction in IPC would be more than compensated for by the simpler issue logic in the DIS, especially in the case of large issue widths. These findings are consistent with a recent study into an microcluster architecture. The architecture had a single instruction window steering instructions to seperate microclusters. Each microcluster had its own mulitple FIFO instruction queues, register files and functional units [13].

This analysis has been based on average window sizes and does not take into account window resource constraints. However these are positive results and we postpone a discussion of the detailed effect of resource constraints on window size for later work.

The analytical model we have developed views the execution of the instruction window as a system with input and output rates. The input rate is the number of instructions decoded per cycle (IDPC) and the output rate is the number of instructions issued per cycle. If we assume that all instructions can issue once there operands become available (e.g. there is no conlfict for issue slots) then the maximum length of time any instruction will wait in the window is MD+1 cycles. We can therefore estimate the average window size $\overline{WS} = (MD + 1) * (\overline{IDPC} - \overline{IPC}) + \overline{IDPC}$. Since most machines tend to have a maximum decode rate equal to the issue width (IW), we can state that the upperbound for the average window size is given by $\overline{WS} \leq (MD+1) * (IW - \overline{IPC}) + IW$. In our experiments we assume perfect branch prediction so that the inequality becomes an equality condition in the above equation.

Figure 2: Instruction Window Analytical Model

Instruction Stream	CIW	Average IPC		Latency Hiding Effectiveness
		md=0	md=60	
Single	2	1.86	1.44	77
Dual	2	1.56	1.24	79
IPC reduction (%)		0.3 (16)	0.2 (14)	
Single	5	3.93	2.59	66
Dual	5	3.56	2.45	69
IPC reduction (%)		0.37 (9)	0.14 (5)	
Single	9	5.97	3.7	62
Dual	9	5.44	3.46	64
IPC reduction (%)		0.53 (9)	0.24 (6)	

Table 2: Comparison of Single and Dual Instruction Streams when RS=3

Instruction Stream	CIW	Average IPC		Latency Hiding Effectiveness
		md=0	md=60	
Single	2	1.92	1.76	92
Dual	2	1.68	1.55	92
IPC reduction (%)		0.24 (13)	0.21 (11)	
Single	5	4.54	4.00	88
Dual	5	4.18	3.74	89
IPC reduction (%)		0.36 (8)	0.26 (7)	
Single	9	7.94	6.16	78
Dual	9	7.43	5.87	79
IPC reduction (%)		0.51 (6)	0.29 (5)	

Table 3: Comparison of Single and Dual Instruction Streams when RS=6

RS	CIW	SIS		DIS			Difference in WS
		IPC	WS	AU IPC	DU IPC	AU WS/DU WS	
3	2	1.44	36	0.54	0.70	29/19	7 (19)
	5	2.59	152	1.04	1.41	61/100	52 (34)
	9	3.70	332	1.45	2.01	160/187	145 (44)
6	2	1.76	17	0.68	0.87	21/9	-4 (-24)
	5	4.00	65	1.61	2.13	26/56	9 (14)
	9	6.16	182	2.5	3.37	95/104	78 (43)

Table 4: Instruction window size (WS) for Dual and Single Instruction Streams

7 The Effect of Different Hardware and Software Configurations

Having provided justification for renewed interest in Access Decoupling as way to simplify o-o-o issue logic, we now quantify the performance impact of the different hardware and software configurations described in section 4. We consider the case when CIW is 9 (AU and DU issue width of 4 and 5, respectively) because it offers the best results for reducing window size. In section 7.1 we consider different data dependency and renaming configurations. In section 7.2 we consider different memory ordering and barrier configurations. In each section we quantify the performance of the configurations when the memory differential is 0 and 60 cycles. We choose an MD of 0 cycles in order to isolate the effect of the different configurations from memory latency. An MD of 60 was chosen because it is comparable to the cost of a second level cache miss [4] and it assumes a weak memory system (see figure 1) capable of capturing no locality. In practice for a high performance architecture the memory system will be able to reduce the average access time by using first and second level caches. Each section also discusses the latency effectiveness and scalability of the configuration.

7.1 Data Dependency and Renaming Configurations

Figures 3 and 4 show the measured IPC levels for different data dependency analysis and renaming configurations when the MD is 0 and 60 cycles, respectively. The postfix '+rename' and '-rename' indicates where the configuration included or excluded memory renaming. The measurements were made with weak memory ordering and non-synchronising barriers.

The major finding is that access decoupling can still achieve high levels of IPC even with large memory differentials. When MD is 0 and 60 cycles the IPC upper limits are 7.5 and 5.9, respectively. This means an issue efficiency of 83% and 66%, respectively. However this

[4] The pentium pro has 50 cycle L2 miss latency[3]

result requires the use of perfect dependency analysis, renaming and the ability to reorder operations anywhere in the program (RS6).

The 'pfa+rename' configuration in figure 3 shows that the large increase in IPC, from RS0 to RS1, occurs as operation reordering within leaf level loops and basic blocks is enabled. After RS1 the increasing scope for reordering operations provides smaller gains in IPC. The other significant increase occurs, from RS5 to RS6, when operations are allowed to reorder over procedure call boundaries.

The difference in behaviour of the configuration with 'pfa+rename' in figures 3 and 4 is due to the need for wider reordering scope to hide the MD. This can be seen clearly in the change for RS1. In figure 3 RS1 has an IPC at 70% of the upper limit of 7.5 IPC. Compare this with figure 3 where RS1 is only 37% of the upper limit. The configuration with 'pfa-rename' exhibits a similar type of behaviour. Wider reordering scope is therefore essentially for producing higher IPC for both configurations with perfect dependency analysis.

We also notice for the configuration with perfect analysis and renaming reordering operations across procedure boundaries is important to achieving high IPC when MD=60. This result can be seen in the large difference of 2.4 IPC (40% of the upper limit of 5.9 IPC) between RS5 and RS6 in figure 4. Although in figure 3 the increase is 2 IPC between RS5 and RS6 its significance is less important (27% of the upper limit of 7.5).

The lower bound for any data dependency analysis technique is shown by the configuration with 'cta-rename' in figure 3 and 4. We can see that with conservative analysis and no renaming the scope for reordering operations is not the performance bottleneck. There is therefore clearly no benefit to increasing the scope for reordering operations beyond RS1. The consequence is that for large memory differentials the latency can not be hidden and the IPC drops to 0.5 (see figure 4).

Renaming for both perfect and conservative dependency analysis only provides additional IPC for RS3 (barriers at nested loop and procedure boundaries) and above. The reason for this we can conjecture is due the majority of false dependency arcs crossing leaf level loop boundaries. For RS1 and RS2 barriers are placed at leaf level loop boundaries. Renaming provides little benefit because most of the false arcs cross these barriers. The ordering of memory operations is therefore enforced by the barrier rather than the false dependency arc.

Renaming compensates for conservative analysis when MD=0 (see figure 3). However with large latencies there is little difference between the 'cta+rename' and 'cta-rename' configurations. The same is not true for perfect analysis where the gap for RS6 between 'pfa+rename' and 'pfa-rename' widens when MD is 60. This indicates that for perfect dependency analysis renaming is more important when MD is 60. It also shows that renaming is more important to perfect than conservative dependency analysis when MD is 60 (see figure 4).

Figures 5 and 6 show the latency hiding effectiveness for different CIW for RS1 and RS6, respectively. To varying degrees all configurations show a deterioration in effectiveness as the CIW is increased. We can see that in both figures the gap between the conservative and perfect analysis is large, indicating the importance of high quality dependency analysis to latency hiding.

It will also be noticed that for both perfect analysis configurations the wider scope for reordering increases the latency hiding effectiveness. This can be seen clearly by the difference in the figures 5 and 6 where when CIW=2 the effectiveness increases by about 20%.

For the configuration with perfect analysis renaming improves the effectiveness of the latency hiding only when the reordering scope is large and dependency analysis is perfect. This can be seen from the difference between figure 5 and 6.

For RS9 and perfect dependency analysis renaming is important for improving the scalability of Access Decoupling. This can be seen from figure 6 where for a CIW of 2 the difference in effectiveness is only 1%. However when CIW is 9 the difference from renaming is 16%. This indicates that the effectiveness of the decoupling becomes more dependent on renaming as the CIW increases. The reduction in effectiveness for the configurations 'pfa+rename' and 'pfa-rename' is 14% and 28%, respectively. The scalability of access decoupling is therefore dependent on the use of renaming when dependency analysis is perfect.

The latency hiding effectiveness for RS9 and the configuration with perfect dependency analysis and renaming exhibits behaviour favourable to the scalability of access decoupling. Firstly when CIW ≤ 5 the effectiveness is relatively stable around 90%. Secondly when CIW is 9 the effectiveness, as would be expected for a large memory differential, decreases, but only by 16%. To improve the effectiveness we will need to adopt other latency hiding techniques for high CIW. We discuss one option in section 8.

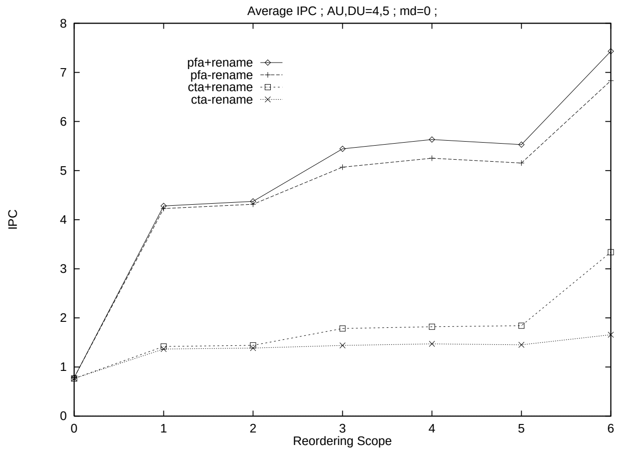

Figure 3: Average IPC when MD=0

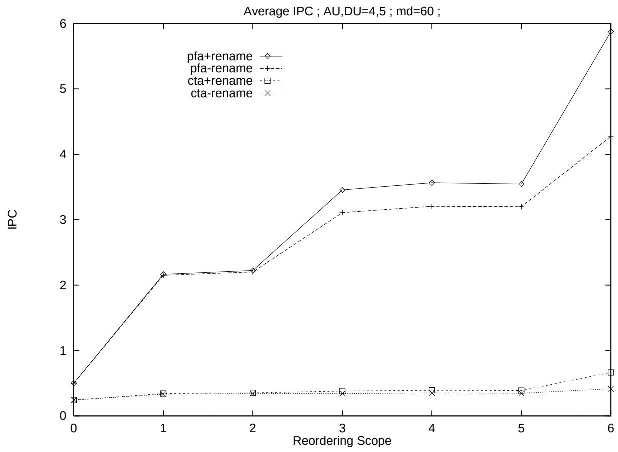

Figure 4: Average IPC when MD=60

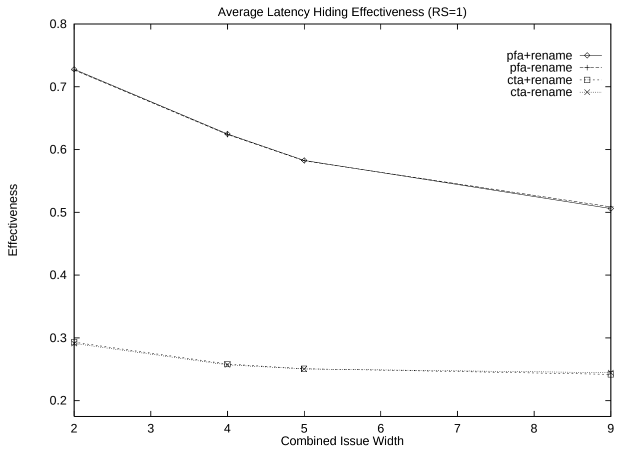

Figure 5: Average Latency Hiding Effectiveness (RS=1)

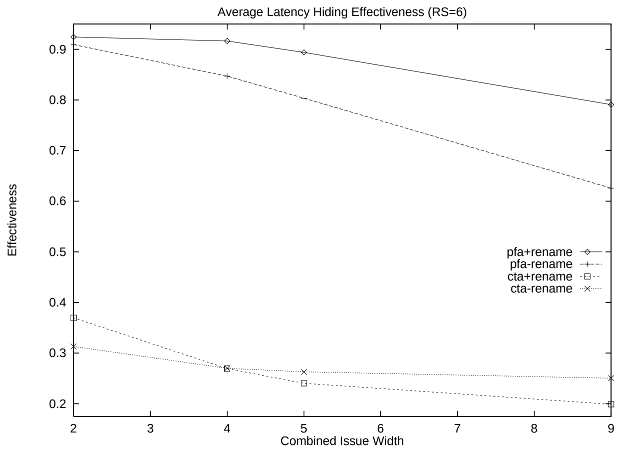

Figure 6: Average Latency Hiding Effectiveness (RS=6)

7.2 Memory Reordering and Barrier Configurations

Figures 7 and 8 show the IPC for different memory ordering and barrier configurations when the MD is 0 and 60 cycles respectively. The measurements were made with perfect dependency analysis and renaming. The different barriers configurations where measured using weak memory ordering.

Th most significant result is that any future Access Decoupled machines must support reordering of memory operations. This can be clearly seen in both figures 7 and 8 by the large gap between the WKO and, the SSO and STO configurations. The SSO configuration has only a marginally higher IPC than the STO configuration.

Like the configuration with conservative analysis discussed in section 7.1, the SSO and STO configurations show no change in performance for reordering scopes beyond RS1. In section 7.1 we commented on the necessity of wide reordering scope in order to hide the MD of 60 cycles. We can therefore infer that the STO and SSO configurations will have poor at latency hiding capabilities because it restricts the operation reordering for wide scopes. This can be seen clearly in the graph for latency hiding effectiveness in figure 10.

Figure 8 shows for RS1 and RS2 the importance of the AU being able to continue prefetching data across leaf level loop boundaries to hide large latencies. This result follows from the large IPC difference between the configurations with synchronising and non-synchronising barriers [5].

We also notice that our Access decoupled machines synchronises between nested loops and across procedure call boundaries due to data dependencies from the DU to AU. We can infer this from the converging lines for RS3 and RS5 when the configurations have synchronising and non-synchronising barriers (see figure 8).

Figure 9 and 10 show the latency hiding effectiveness for different CIW for RS1 and RS6, respectively. Again as in the previous section we observe, the varying degrees of reduction in

[5] The line for non-synchronising barriers is the same as the weakly ordered configuration line.

effectiveness as CIW increases.

The major differences between figures 9 and 10 is the increasing effectiveness of the configuration with synchronising barriers and the lack of variation in the behaviour of SSO and STO. For RS1 the effectiveness of latency hiding is poor when the configuration has synchronising barriers. As was noted above above this results from the need to allow the AU to prefetch data across leaf level loop boundaries. Access decoupling does not scale well with this configuration. When CIW is 9 the effectiveness is only 38% which is less than SSO and STO.

The configurations for SSO and STO behave independently of instruction reordering for two reasons. Firstly the addition of extra issue slots provides little increase in IPC once CIW=4. Secondly as commented earlier in this section the IPC does not increase with wider reordering scope.

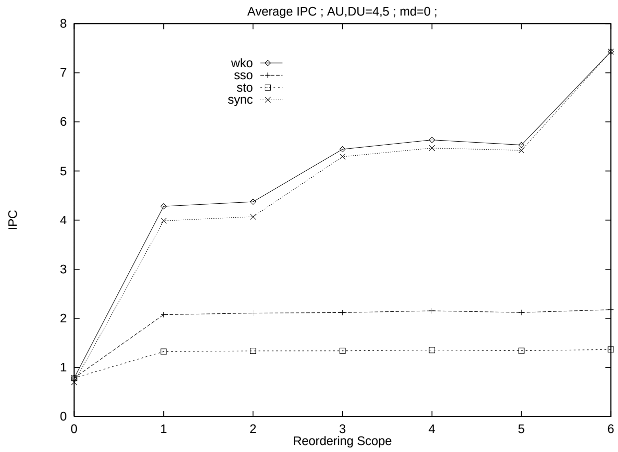

Figure 7: Average IPC when MD=0

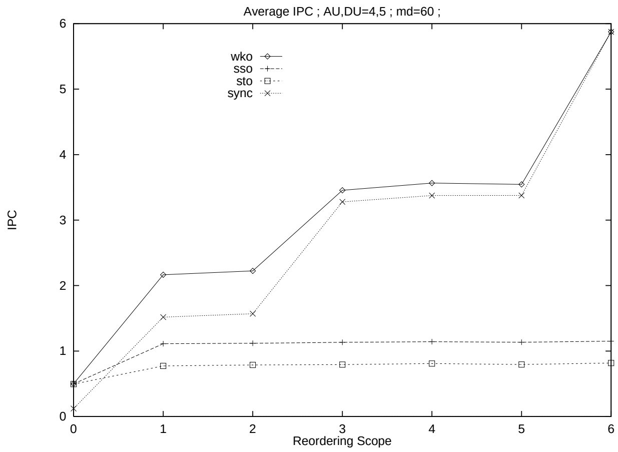

Figure 8: Average IPC when MD=60

8 Conclusion and Future Work

In this paper we have presented an argument for renewed interest in Access Decoupling as a way to reduce the complexity of hardware issue logic in an o-o-o machine. We have also examined the important hardware and software design issues which affect the IPC, latency

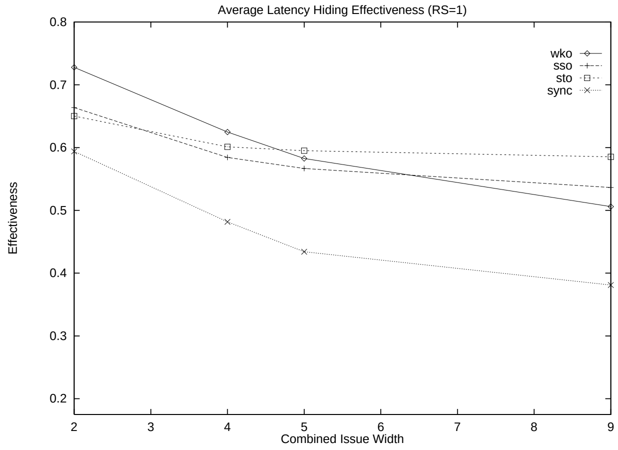

Figure 9: Average Latency Hiding Effectiveness (RS=1)

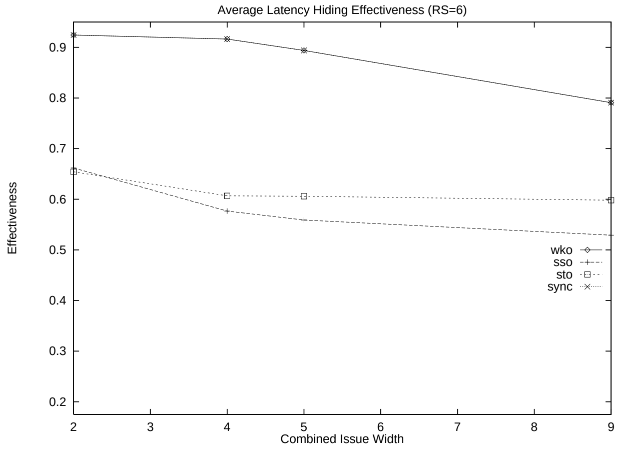

Figure 10: Average Latency Hiding Effectiveness (RS=6)

hiding effectiveness and scalability of our Access Decoupled machine.

Using results from analytical and simulation studies we have compared the performance and average window size of a SIS and DIS o-o-o machine. The results have shown that although IPC levels are marginally smaller for DIS (between 9% and 5% when CIW is 9) the reduction in average window size could be as much as 44% when CIW is 9. We have also observed that the reduction in average window size increases with larger CIW. We know that window logic delays impact upon clock period and vary quadratically with window size and issue width [13]. We would therefore anticipate that when CIW is large the payoff between lower IPC and faster clock speed will result in higher performance for the DIS machine.

We have also shown that when CIW is 9 we can achieve high IPC even when memory differentials are as large as 60 cycles. The results show that our Access Decoupled machine can achieve IPC levels of 7.5 and 5.9 for a MD of 0 and 60 cycles, respectively. These results are dependent on an architecture configuration with perfect dependency analysis, renaming, weak memory ordering and wide scope for reodering operations. For the same configuration we have also found behaviour favourable to the scalability of access decoupling. The latency hiding effectiveness has been shown to be stable around 90% for CIW ≤ 5 and only decrease by 16% when CIW is 9. We conclude therefore that at high CIW there is a need consider additional latency hiding techniques. One solution is to use a technique described in [1] which captures some of the temporal locality exposed by decoupling in the decoupled memory. When a load and store address match in the decoupled memory, the result for the store could be bypassed to the DU, removing the need to access the memory system.

Our study into the hardware and software design issues which effect performance have shown the importance of sophisticated compiler technology. To hide large memory latencies the compiler must be capable of accurate inter and intra procedural dependency analysis. Without accurate analysis the latent benefits of wider reordering scope can not be realised and high latencies can not be hidden.

Another major finding is that the ability to reorder memory operations is critical to achieving high IPC and hiding memory latency. We have also found that reordering operations across procedure call boundaries is important to achieving high IPC when the MD is large; it has less significance when the memory differential is 0 cycles.

Wide reordering scopes can offer sufficient operations to hide latency but are dependent on the memory ordering scheme and the data dependency analysis. For configurations with STO, SSO and conservative analysis the benefits of wide reordering scope can not be realised, due to the memory ordering scheme and dependency analysis being the bottleneck.

For configuration with narrow reordering scopes, where barriers exist at leaf level loops, we have observed that it is important to allow the AU to continue prefetching data after the barrier. Synchronising barriers are observed to reduce performance especially when the MD is large.

In future work we plan to examine the effect of window size and code expansion on the DIS and SIS machines. We also plan to investigate the use of techniques like the address reorder buffer [8] to reorder memory operations in a decoupled machine. Finally we will look at ways to optimise the decoupled memory to utilise the locality exposed by decoupling when the CIW is large.

References

[1] Local Author. A Limitation Study into Access Decoupling. In *To be published in Euro-Par 97*, University of Passau, Germany, Aug. 1997.

[2] A. Berrached, L.D. Coraor, and P.T. Hulina. A Decoupled Access/Execute Architecture for Efficient Access of Structured Data. In *Proc. of the 26th Hawai Int. Conf. on System Sciences*, volume 1, pages 438–47, Los Alamitos, CA, USA, Jan 1993. IEEE.

[3] D. Bhandarkar and J. Ding. Performance Characterisation of the Pentium Pro Processor. In *Proceedings of the 3rd Int. Symp. on High Performance Computer Architecture*, San Antonio, Texas, USA., Feb. 1997. IEEE.

[4] P. Bird, A. Rawsthorne, and N.P. Topham. The Effectiveness of Decoupling. In *Proc. Int. Conf. on Supercomputing*, Tokyo, Japan, May 1993.

[5] J.E. Smith et al. The ZS-1 Central Processor. In *Proc. of the 2nd Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, October 1987.

[6] M. Berry et al. The Perfect Club Benchmarks, Effective Performance Evaluation of Supercomputers. Techreport 827, CSRD, University of Illinois, Urbana-Chmpaign, Urbana, Illinois., May 1989.

[7] W. Blume *et al.* Automatic Detection of Parallelism: A Grand Challenge for High Performance Computing. CSRD 1348, Center for SuperComputing Research and Development., University of Illinois at Urbana-Champaign, 1308 W.Main St, Urbana, Illinois, July 1994.

[8] M. Franklin and G.S. Sohi. ARB: A Hardware Mechanism for Dynamic Rordering of Memory References. In *IEEE Transactions on Computers*, volume 45, May 1996.

[9] L. Gwennap. Digital 21264 sets new standard. *Microprocesor Report*, 10(14), Oct. 1996.

[10] M. Kumar. Measuring Parallelism in Computation Intensive Scientific/Engineering Applications. *IEEE Transactions on Computers*, 37(9):1088–1098, Sept. 1988.

[11] local author. Evaluating the Limits of Access Decoupling using the Latency Hiding Model. Technical report, home university, 1997.

[12] M.K.Farrens and A.R.Pleszkun. Implementation of the PIPE Processor. *IEEE Computer*, pages 65–70, Jan 1991.

[13] S. Palacharla, N.P. Jouppi, and J.E. Smith. Complexity-Effective Superscalar Processors. In *24th Annual International Symposium on Computer Architecture*, 1997.

[14] P.M. Petersen. and D.A. Padua. Evaluation of Parallelsing Compilers. CSRD 1279, Center for Supercomputing Research and Development., University of Illinois at Champaign-Urbana, Urbana, Illinois, 61801, 1992.

[15] Z. Shen, Z. Li, and P-C. Yew. An Empirical Study on Array Subscripts and Data Dependences. In *1989 International Conference on Parallel Processing*, pages II–145–II–152, 1989.

[16] N.P. Topham, A. Rawsthorne, C.E. McLean, M.J.R.G. Mewissen, and P.Bird. Compiling and Optimising for Decoupled Architectures. In *Proc. of Supercomputing '95*, San Diego, Dec. 1995. ACM press.

[17] G. Tyson, M. Farrens, and A.R. Pleszkun. MISC : A Mutiple Instruction Stream Computer. In *Proc. of the 25th Ann. Sym. on Microarchitecture*, Portland, Oregon, Dec 1-4 1992.

[18] S. Vajapeyam, G.S. Sohi, and W-C Hsu. An Empirical Study of the CRAY YMP Processor using the PERFECT Club Benchmarks. In *Proceedings of the 1991 ACM Int. Conf. on Supercomputing*, pages 170–179, New York, 1991. ACM, ACM press.

[19] Wm A. Wulf. An Evaluation of the WM Architecture. In *Proc. Int. Symp. on Computer Architecture*, Gold Coast, Australia, May 1992.

[20] K.C. Yeager. The Mips R10000 Superscalar Microprocessor. *IEEE micro*, 16(2):28–41, April 1996.