Computer Systems Group

Simplifying Hardware for Out Of Order Execution using the Decoupling Paradigm

by

Graham P. Jones and Nigel P. Topham

CSG Report Series **CSG-32-97**

Computer Systems Group **September 1997**

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

Simplifying Hardware for Out Of Order Execution using the Decoupling Paradigm

Graham P. Jones * and Nigel P.Topham †

Technical Report CSG-32-97

Department of Computer Science
University of Edinburgh

September 23, 1997

Abstract: Future hardware and software technology 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 issue 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 the behaviour of data prefetching on an access decoupled machine and a superscalar machine. Access decoupling is a latency hiding technique that partitions a program into two separate *instruction streams* to aggressively prefetch data. Superscalar architectures can support data prefetching through out-of-order execution, non-blocking loads and lock-up free caches. In this paper we investigate if there are benefits to using the decoupling paradigm given that an out-of-order (o-o-o) superscalar architecture could in principle prefetch to the same degree as an access decoupled machine.

We have found that for large *issue width* the access decoupled machine can hide memory latency more effectively than a single instruction window o-o-o superscalar architecture. For realistic window sizes, our results show that to achieve the same performance as an access decoupled machine our o-o-o superscalar machine requires an instruction window 2.5 to 5 times larger.

Given that window issue logic is critical to processor clock speeds and is dependent on window sizes, architectures that reduce window logic complexity will be of interest to future designers. Our findings demonstrate that an access decoupled machine offers the benefits of effectively hiding memory latency whilst reducing the complexity of window issue logic.

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

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

1 Introduction

The future of high performance microprocessor design is to provide improved performance by extracting higher degrees of instruction level parallelism. In superscalar architectures parallelism is exploited by reordering instructions within an instruction window and issuing multiple independent instructions per cycle. However as processor speeds increase and issue widths get larger the cost of a main memory access is becoming relatively more expensive. One solution is to hide memory latency by *data prefetching*.

Data prefetching is a technique that hides memory latency by overlapping access and data operations. Data prefetching can be implemented in either hardware [8] and software [4] or a hybrid [5] of both schemes. However as memory latencies become relatively more expensive the number of independent overlapped instructions required to hide the access times increases. Larger instruction windows are therefore required to detect independent instructions that can execute in parallel with memory access operations.

The pressure to increase window sizes is also driven by the goal of providing ever larger issue widths. However large window and issue width sizes introduces greater complexity in window issue logic. A recent paper has shown that delays in the issue logic vary quadratically with window and issue width size [12]. Since delays in issue logic will be critical to processor clock there is a need to consider architectures that simplify issue window logic.

To solve the window complexity problem some architectures use separate microclusters. Microclusters may share or have a dedicated instruction window, but each has its own register file and function units. This design simplifies window logic by flagging instructions for execution on particular microclusters. This reduces the size of the instruction window but can limit the number of instructions issued per cycle.

Access decoupling is a latency hiding technique that partitions a programs - statically or dynamically - into two separate instruction streams in order prefetch data aggressively. The instruction streams are loosely coupled. One stream, executed on an *address unit* (AU), prefetches data for the second stream, executed on a *data unit* (DU). Memory accesses can then be pipelined to tolerate large memory latencies provided the two streams can *decoupled* sufficiently.

In principle the same level of prefetching in an access decoupled machine could be achieved with an out-of-order (o-o-o) superscalar architecture. The question is then "why should designers consider using the decoupling paradigm?"

Memory latencies are typically 20-50 cycles whereas arithmetic function latencies are 2-5 cycles (excluding divide and intrinsics). A system could easily tolerate a small degree of o-o-o execution amongst arithmetic operations provided loads could *slip* by a large amount with respect to arithmetic operations. This slippage between arithmetic and load operation is exactly what occurs in a decoupled machine. In other terms, we can have small instruction windows for arithmetic and access operations provided the latter can slip by a large amount with respect to the former. To illustrate this idea in section 6 we introduce the concept of the *equivalent single window* (ESW). The ESW is the minimum size of window required by a processor with a single window to have the same instructions resident as the access decoupled machine.

In answer to our question, we believe that an access decoupled machine can be viewed as a variant of a microcluster architecture with two separate instruction windows. The asynchronously executing units, through code partition and dynamic slippage, combines the benefits of reducing window logic complexity with data prefetching.

In this paper we compare the relationship between window size and memory latency for an *access decoupled machine* (ADM) and a *single window* o-o-o *superscalar machine* (SWSM). We also evaluate the size of window required by the SWSM to achieve the same performance as the ADM.

The thesis of this paper is developed in the following way. In section 2 we discuss previous work on access decoupling. In section 3 and section 4 we outline the ADM and SWSM, respectively. In section 5 we outline the memory system used for both architectures. In section 6 we discuss the notion of the *effective single window size* (ESWS) to help explain some of our findings. In section 7 we describe our simulation technique and the experimental benchmark programs. In section 8 we present the results of our work. Finally in section 9 we draw together our findings and suggest avenues for future work in this area.

2 Background

Access decoupling is an asynchronous data prefetching technique that tries to hide memory latency by overlapping computation and memory access operations. Central to all decoupled machines [1, 6, 11, 17] 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 [6] and PIPE [11] 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 [1], MISC [15] and ACRI [3] 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 [15] architecture, derived from PIPE, has four asynchronous units each with their own instruction cache, but a 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 [18]. 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 (ADM)

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.

Each unit has is own separate instruction window, functional units and register files. The units can share results by moving results between register files. 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 as 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 [1, 11, 6].

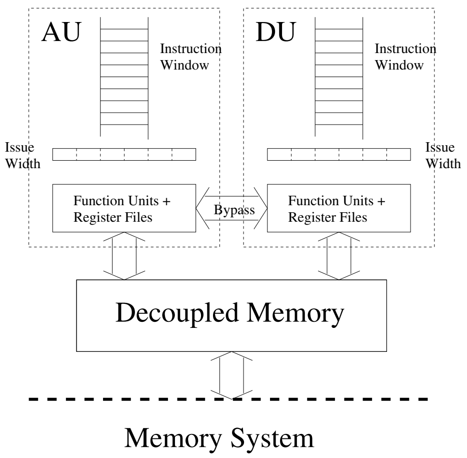

Figure 1: ADM

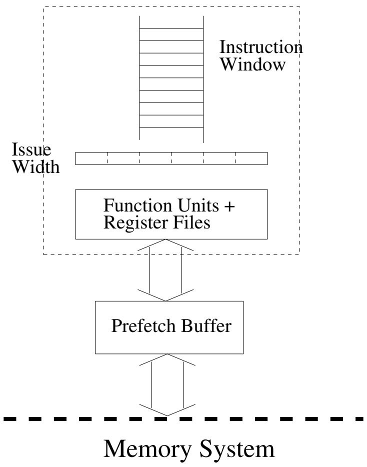

Figure 2: SWSM

4 The Single Window Superscalar Machine (SWSM)

The SWSM is shown in Figure 2. The architecture is an o-o-o machine with a single instruction window for reordering operations. In each cycle independent operations which are ready to execute are issued to the function units. Unlike the ADM the full issue width is available for issuing instructions every cycle. This means that if the SWSM is able to guarantee the full issue width is utilised it could outperform the ADM.

There are different types of hardware, software and hybrid schemes for data prefetching. For SWSM we use a hybrid scheme. Every memory operation takes two instructions, a prefetch and an access operation. The prefetch instruction pre-loads data into the prefetch buffer ahead of the access operation. Prefetch operations, unlike software schemes, are allowed to begin execution as soon as runtime resources allow. Using this scheme we gain the benefits of exact address computation with dynamic execution. The prefetch buffer is a fully associative buffer responsible for storing prefetched data. Requests from the prefetch buffer take a 1 cycle.

5 The Memory System

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 any perceived memory differential.

6 The Effective Single Window (ESW)

An advantage of the ADM is that the dynamic slippage between the window of instructions on the AU and DU means that the effective single window size can be greater than the sum of the individual units window sizes. Figure 2 illustrates the idea of the ESW. The diagram shows the streams for the AU, DU and a single instruction stream. In the single instruction stream the instructions are shown in program order (with later instructions appearing further down the page) and labelled with the units they execute on in the ADM. The diagram shows that due to the dynamic slippage between the units, the AU is executing instruction further into the instruction stream than the DU. In order for a processor with a single window to have the same instructions resident the minimum size of window must be as large as the ESW.

7 Benchmark and Simulation Technique

7.1 Simulation Technique

In our experiments we used a technique adapted from work by Petersen [13] and described in [9]. The technique works by annotating the source 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 ADM amd SWSM can then be performed by executing the annotated program.

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 data prefetching

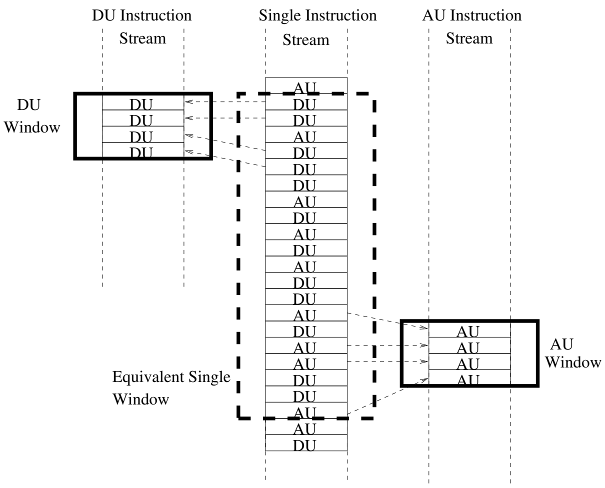

Figure 3: Effective Single Window

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 code compiler. These issues are not however dealt within the scope of this paper.

For the ADM partitioning of the code between the AU and DU is performed statically by the partitioning algorithm in the OCTAVE compiler [14]. 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. Code expansion can occur because some nodes are annotated to execute on both units. This means that at execution time some operations will be duplicated. To remove the effects of duplication we execute the same number of operations on the SWSM and ADM. The code expansion for the 7 programs simulated in our experiments is on average less than 5.5% (see Table 1). It is therefore believed that code expansion will have little effect on the results reported in this paper. In future work we intend to examine the effect when duplicate operations are removed for the SWSM.

Source level operations are translated at runtime into single instructions for the architecture simulator. On the ADM loads and Stores are executed as one instruction on each of the units. On the SWSM loads and stores generate a prefetch and an access operation. Integer and address computations have a 1 cycle cost. Floating point operations take 5 cycles to complete.

There is no speculative execution but we assume loop closing branches have been removed by optimisations like loop unrolling and branch prediction. Data dependency analysis is perfect and false dependencies are removed by renaming. The purpose of examining such an ideal case is to provide the best opportunity for prefetching data, to have high instruction level parallelism (ILP) and to place the greatest pressure on the latency hiding mechanism.

We limit the instruction issue width to projected future values. The technique we used to identify optimal configurations of AU and DU issue widths is however beyond the scope of this paper. The configurations we discovered to be most suitable are (1,1),(2,3) and (4,5). The first and second value in the brackets being respectively the AU and DU instruction issue widths. Here we only consider the (4,5) case for the ADM and an issue width of 9 for the SWSM.

7.2 Benchmark Programs

We chose a selection of 7 scientific Fortran programs from the PERFECT club suite [7] 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 its entirety.

We selected benchmarks from the PERFECT club to represent varying degrees of vectorisation 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 [16] and the decoupling efficiency (DE) obtained from [14]. 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

8 Experimental Results

In the section we present the major findings of the paper. All the programs shown in Table 1 were simulated in our experiments. For the purpose of this paper we have selected three representative programs that exhibit the range of observed behaviour. The three selected programs were FLO52Q, MDG and TRACK. Figure 4 shows the *latency hiding effectiveness* of all seven programs when the window size is unlimited and the memory differential is 60 cycles [1] The latency hiding effectiveness (LHE) is defined as $LHE = T_{perfect}/T_{actual}$ where T_{actual} is the execution time for the ADM and $T_{perfect}$ is the execution time for a machine with perfect latency hiding in which each memory access perceives a single cycle latency. It can be seen there are three bands in which the programs are highly (80-100%), moderately (40-60%) and poorly ($< 40\%$) effective at hiding latency. It can be seen that the three programs fall within each of the bands.

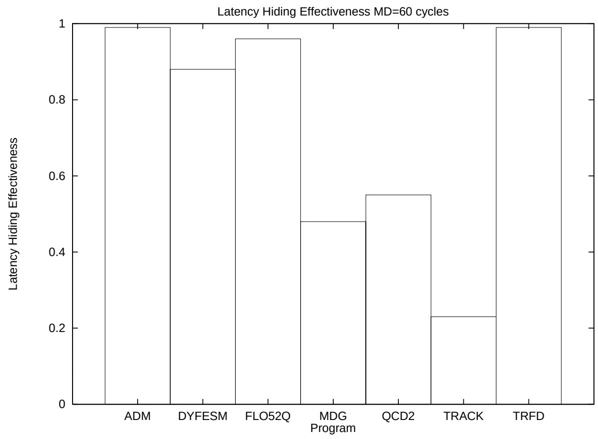

Figure 4: Latency Hiding effectiveness MD=60 cycles

Figures 5, 6 and 7 show the variation in speedup with window size for the access decoupled and superscalar architecture when the memory differential is 0 and 60 cycles. When MD is 0 we see that for small window sizes the ADM performs better than the SWSM with same window size. This is due to the ADM having two windows for reordering operations compared to one for the SWSM. This means there are fewer resource conflicts for window slots and greater scope for reordering operations. It will also be noticed that the graphs show the law of diminishing returns for increasing window size; once window sizes are above 10 instructions, doubling the size does not double the speedup. All the programs reach a cut-off point for window sizes between 40 and 80 instructions when the SWSM performs more effectively. This is due the benefit of the larger instruction issue

[1] An MD of 60 was chosen because it is comparable to the cost of a second level cache miss (the pentium Pro has 50 cycle L2 miss latency[2]) and it assumes a weak memory system 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.

width available to the SWSM. This benefit is only realised once the instruction window is large enough to utilise the available issue width.

In Figures 5, 6 and 7 we see that once MD reaches 60 cycles there is no cut-off point when the SWSM performs better than the ADM. This results applies even for very large windows of 100 instruction slots. The difference between the performance of the two machines must be solely due to the more effective data prefetching of the ADM. Operations on the SWSM which on ADM would have been executed on the DU, are causing address computations to execute later, reducing the pipelining of memory accesses and decreasing the effectiveness of the data prefetching. The difference in performance between the two machines is also dependent on the type of program. For FLO52Q which is highly parallel the gap between the ADM and SWSM is large. However, for TRACK which has little parallelism there is little difference between the two architectures.

We can state therefore that for all the programs we have simulated the ADM machine is more effective at hiding large memory latencies than the SWSM. The difference in performance is dependent on the parallelism and decoupling in the program. Programs that decouple well show the largest improvement in performance for the ADM.

In Figures 8, 9 and 10 we show, for a range of memory differentials, the *required increase* in the SWSM window size to yield the same performance as the ADM. The increase was derived by projecting from the ADM graph to SWSM graph in Figures 5,6 and 7. The graphs show the way in which the required increase varies as a function of the memory latency. It can be seen that as latencies approach 60 cycles the required increase gets larger. This is solely due to the more effective data prefetching of the access decoupled machine. As the memory latency increases, the DU waits longer for data to arrive and the slippage between the two units grows. This means conceptually that the effective single window size (see Figure 3) for ADM gets larger. In order for the SWSM to achieve equivalent performance it requires a correspondingly larger window.

The graphs in Figures 8, 9 and 10 also show that as the ADM window size is increased the required increase reduces. This is due to the SWSM architecture being able to reorder operations to a similar degree as the ADM, and also the benefits of the larger issue width.

Significantly it can be observed that for a realistic ADM window size of 30 instructions and a memory latency of 60 cycles, the required increase in window size for equivalent SWSM performance is dependent on the program but lies between 2.5 to 5 times. Experiments with the other benchmark programs shown in Table 1 have also been found to fall within this range. Larger windows introduce extra hardware complexity and longer window logic delays [2]. We can state therefore that the ADM requires smaller instruction windows and hence simpler window logic.

Having shown that the ADM performs consistently better than the SWSM we now compare the latency hiding effectiveness of the ADM against a perfect latency hiding technique (one in which all the memory differential is hidden). Table 2 shows the measured LHE for different window sizes when the memory differential is 60 cycles.

The results show that when window sizes are small increasing the window size causes

[2] In [12] is was shown that delays vary quadratically with window size.

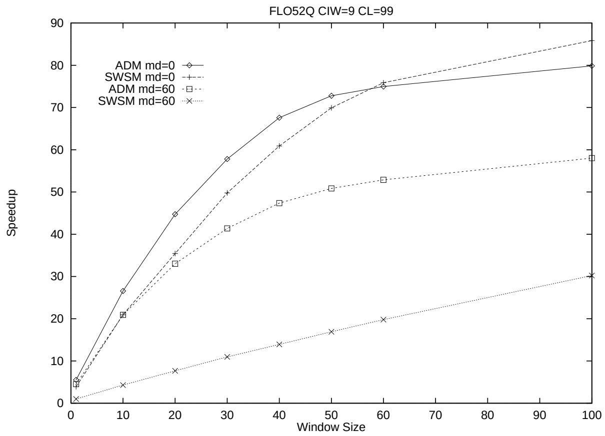

Figure 5: FLO52Q

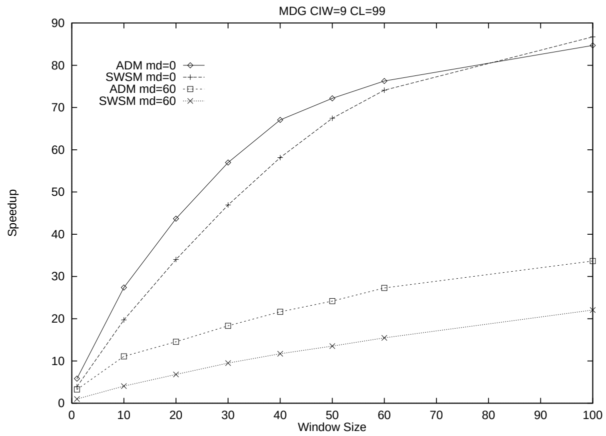

Figure 6: MDG

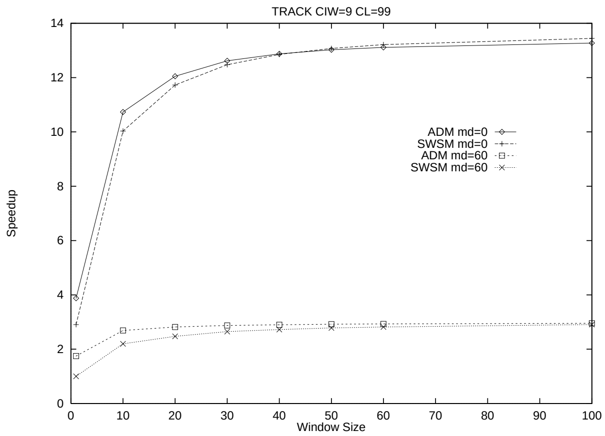

Figure 7: TRACK

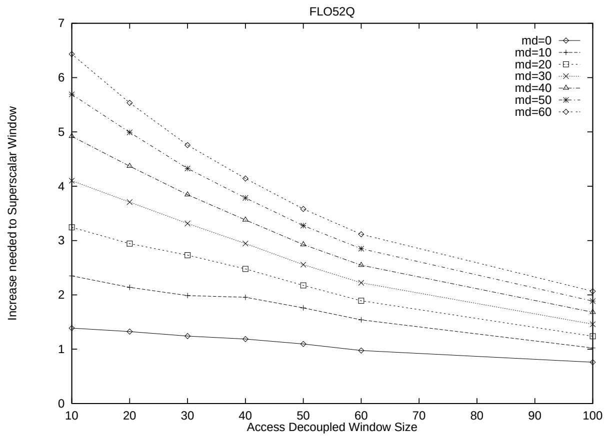

Figure 8: FLO52Q

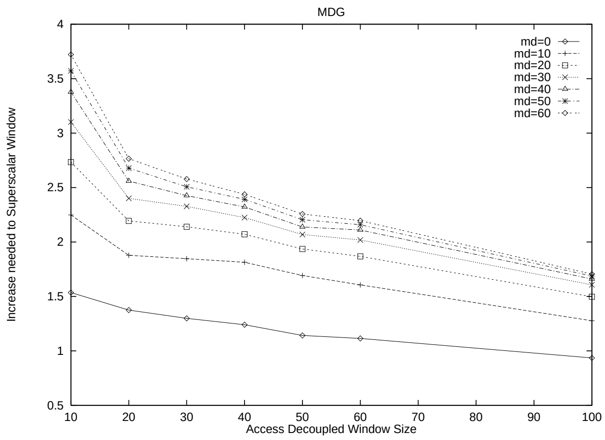

Figure 9: MDG

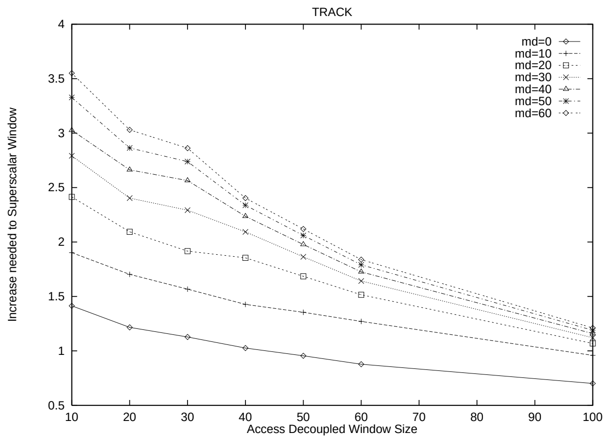

Figure 10: TRACK

a reduction in the LHE. This is due to the extra parallelism on the DU placing greater pressure on the memory system. The AU window is not yet large enough allow the AU to pipeline accesses sufficiently to hide the latency. However there eventually comes a point when the larger window size allows more operations to execute in parallel and the LHE starts to improve. For MDG and FLO52Q that point is 40 and 50 instructions respectively. This result suggests that for realistic window sizes (1 to 30 instructions), increasing the window size will result in the latency hiding mechanism of the ADM performing less effectively. Table 2 also shows even with large window sizes we do not approach the LHE of an ADM with unlimited resources.

Our findings show that for realistic window sizes the ADM can hide latencies better than SWSM but that as the window size increases its effectiveness at hiding latency deteriorates. This illustrates the tensions that exist between having greater parallelism and the access decoupling mechanism. As the window size get larger, the instruction level parallelism increases and the execution times fall. However the extra parallelism places greater pressure on the decoupling mechanism resulting in a decrease in LHE. The result is that more of the critical path time is now composed of the memory differential. There comes a point however, when the AU window is large enough to compensate for the extra parallelism on the DU, and more address operations can be pipelined to hide the latency.

In the short to medium term high performance architectures will have window sizes in the range that shows a reduction in the LHE. In future work we will investigate mechanisms to improve the latency hiding of the ADM. One possibility is a bypass mechanism which captures the temporal locality exposed by decoupling [10].

Program	ADM Window Size								
	∞	1	10	20	30	40	50	60	100
ADM	1.00	0.72	0.51	0.43	0.41	0.39	0.38	0.38	0.39
DYFESM	0.87	0.81	0.59	0.53	0.49	0.48	0.50	0.49	0.49
FLO52Q	0.97	0.82	0.79	0.74	0.72	0.70	0.70	0.71	0.73
MDG	0.48	0.56	0.41	0.33	0.32	0.32	0.34	0.36	0.40
QCD2	0.55	0.76	0.54	0.46	0.44	0.43	0.43	0.43	0.44
TRACK	0.22	0.45	0.25	0.23	0.23	0.23	0.22	0.22	0.22
TRFD	1.00	0.90	0.72	0.53	0.43	0.41	0.42	0.43	0.45

Table 2: Latency Hiding Effectiveness for MD=60 cycles

9 Conclusion and Future Work

This paper has focused on two objectives in the design space of future microprocessors; the need to hide large memory latencies and the need to reduce the complexity of window issue logic. We have investigated the use of data prefetching on an access decoupled machine and a single window o-o-o superscalar architecture.

In this paper we have examined the relationship between memory latency, window size and speedup for the two architectures. In order to remove the impact of other architectural issues we have assumed an idealistic environment. This environment provides good conditions for data prefetching, high levels of ILP and places the greatest pressure on the latency hiding mechanism.

We have found that the ADM is more effective at hiding memory latency than the SWSM. For large memory differentials (60 cycles) we have found that even for large window sizes of 100 instructions, the ADM consistently performs better than the SWSM. Our results have also shown that to achieve the same speedup as an ADM the SWSM needs a window size between 2.5 to 5 larger. The increase in window size required to achieve equivalent performance on the SWSM was also found to increase with larger latencies.

To explain some of our findings we have introduced the concept of the effective single window. The ESW conceptually illustrates how the ADM is able to perform better than an architecture with twice the size of instruction window.

Our results have also shown how the latency hiding effectiveness of the ADM decreases as the window size increases to 50 instructions. Though the speedup did increase with larger window size the ADM was not found to be as effective at hiding latency. However when windows were greater than 50 instructions the LHE was found to improve. This behaviour illustrates the tensions that exist between higher ILP and the access decoupling mechanism.

This paper has shown that access decoupling can combine the benefits of latency hiding with simplifying the window logic complexity. We would conclude therefore that there is a need for further work in the use of access decoupling. In future work we will examine the effects of code expansion on the ADM and SWSM. We will also compare the difference in performance between a static and dynamic partition of the code on the ADM. Finally, inorder to improve the LHE of the ADM we will investigate a mechanism for capturing the temporal locality exposed by decoupling.

References

[1] 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.

[2] 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.

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

[4] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. In *4th Annual Symposium on Parallel Languages and Operating Systems*, pages 40–52, Santa Clara, California, April 1991.

[5] Tzi cker Chiueh. Sunder : A Programmable Hardware Prefetch Architecture for Numerical Loops. In *Proc. Supercomputing '94*, pages 488–497, Los Alamitos, CA, USA, Nov 1994. IEEE Comput. Soc., ACM , SIAM, IEEE Comput. Soc. Press.

[6] 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.

[7] 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.

[8] J.W.C. FU and J.H. Patel. Data prefetching strategies for vector cache memories. In *Proceedings. The fifth International Parallel Processing Symposium*, pages 555–560, Los Alamitos, CA, USA, April-May 1991. IEEE Computer Society Press.

[9] G. Jones. Evaluating the Limits of Access Decoupling using the Latency Hiding Model. Technical report, Edinburgh University, 1997.

[10] G.P. Jones and N.P. Topham. A Limitation Study into Access Decoupling. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing*, pages 1102–1111. University of Passau, Germany, Springer, Aug. 1997.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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