A Comparison of Superscalar and Decoupled Access/Execute Architectures

Matthew K. Farrens, Pius Ng, and Phil Nico
Department of Computer Science
University of California - Davis
Davis, CA 95616

Abstract

This paper presents a comparison of superscalar and decoupled access/execute architectures. Both architectures attempt to exploit instruction-level parallelism by issuing multiple instructions per cycle, employing dynamic scheduling to maximize performance. Simulation results are presented for four different configurations, demonstrating that the architectural queues of the decoupled machines provide similar functionality to register renaming, dynamic loop unrolling, and out-of-order execution of the superscalar machines with significantly less complexity.

1. Introduction

The importance of being able to make use of the available parallelism in a task is becoming more and more clear. Research over the last 30 years has demonstrated the difficulty of writing parallel programs; however, there are a number of techniques for exploiting *instruction-level parallelism* that have been proposed.

The most commercially successful approach has been *superscalar* architectures [John91], which execute two or more independent instructions in parallel. Superscalar implementations provide dynamically scheduled multiple instruction issue and often out-of-order execution using techniques like register renaming [Kell75].

Decoupled Access/Execute Architectures (DAE) [Smit82, SmWP86] use an alternate approach to exploiting instruction level parallelism. In DAE machines the memory access and execution portions of an instruction stream are *decoupled.* In such a machine the two instruction streams are statically scheduled, but are allowed to ‘‘slip’’ dynamically with respect to each other in order to provide maximum parallelism.

This paper presents the results of an investigation into the relationship between superscalar and decoupled access/execute(DAE) architectures. Both approaches are dynamically scheduled (though they don’t share the same mechanism) and provide the potential of issuing multiple instructions per cycle.

2. Decoupled Access/Execute Architectures

Decoupled access/execute (DAE) architectures separate the memory access and execution components of a program into independent instruction streams, and execute them on seperate processors. These processors are loosely coupled through architectural queues, which provide an asynchronous message passing channel and allow the two instruction streams to execute asynchronously. Branch queues are used in order to allow the two processors to synchronize at branch points.

The access processor is responsible for all data memory accesses and performs loop-index calculations, address generation, and operand fetches. Typically the access processor runs ahead of the execute processor, prefetching memory operands and enqueuing them on the access-to-execute queue (AEQ) for the execute processor. It also writes store addresses onto the memory address queue (MAQ) as soon as it calculates them; it is not required to wait for any synchronization signal from the Execute processor. Data arriving from either the execution unit of the access processor or from the execute-to-access queue (EAQ) will be paired with the address at the head of the MAQ and the resulting write request will be issued to the memory system. This mechanism does not require any software control.

The execute processor receives its operands either from its own register file or from the access-to-execute queue (AEQ) and performs the heart of the calculations of the task. It places any results generated by the computation on the execute-to-access queue (EAQ). These will be paired up with store addresses previously generated by the access processor and written to memory.

DAE architectures rely on an intelligent compiler to separate, schedule, and coordinate the two independent instruction streams cooperating on a task. The compiler decouples a program into the access and execute instruction streams, using the queues to maintain the logical order of instructions. The ability of the two instruction streams to ‘‘slip’’ with respect to one another (i.e. execute at two different points with respect to the original semantics of the task), allows the access processor to lead the execute processor, effectively migrating data fetching to a earlier time and producing what amounts to *dynamic* loop unrolling. The development of a DAE compiler for the PIPE processor [Smit82] is an on-going research project at the University of California, Davis.

3. An Overview of Superscalars

Superscalar processors are defined as uniprocessors of any architecture capable of executing and completing multiple instructions per cycle from a single instruction stream [John91]. Since a number of studies have implied that the amount of instruction-level parallelism available is between two and three instructions, only a superscalar machine of degree 3 is considered in this study.

In most superscalar machines, instructions are prefetched into the *central window* where they are decoded and a directed acyclic graph (DAG) is constructed. As long as there are functional units available and dependency-free instructions at the leaves of the tree, these instructions are issued to the appropriate functional units. Issued instructions are then removed from the central window allowing any instructions dependent on them to become eligible for issue.

Although the conceptual basis of superscalar architectures is quite simple, many issues must be resolved before the anticipated speedup is realizable. Fundamental constraints such as data dependencies and resource conflicts must be effectively dealt with; the interrelationships between instructions in a single instruction stream provide a formidable barrier to achieving enough instruction-level parallelism to justify the cost and complexity of superscalar implementations. Sophisticated compiler techniques such as trace scheduling, procedural inlining, loop unrolling, and software pipelining must also be employed in order to enlarge the size of basic blocks to provide sufficient instruction-level parallelism to ease the code scheduling process [HePa90]

4. Register Renaming and Out-Of-Order Execution

The reuse of registers can cause interlocks between instructions even though the conflicting instructions are otherwise independent. These interlocks are not true data dependencies, but rather an artifact of the finite number of registers available. These hazards can be alleviated by either optimizing code scheduling and register allocation, or providing additional registers used to reestablish the correspondence between registers and values.

If additional registers are provided, they can be allocated by the hardware at run time by coupling the registers with the values needed by the program using a technique called *register renaming* [Kell75]. Register renaming works as follows: If the hardware detects a dependency between registers, it assigns a different logical identifier to the physical register, and replaces the original logical register identifier in the instruction with a reference to the new physical register containing correct value. Thus, an instruction will fetch the value from the newly allocated physical register rather than the outdated physical register. In different instructions the same logical register identifier may refer to several different physical registers, depending on the locations of register references relative to register assignment [John91]. By removing these constraints, available instruction-level parallelism is increased.

Architectural queues provide an alternate register renaming mechanism. In PIPE [GHLP85] for example, the heads of the queues are accessed as registers, while the empty slots in these queues serve as the extra registers. When an instruction references a queue, the queue is advanced, and the next element will be available for the next reference. The correct correspondence between the data fetched by the load instruction in the access processor and operands required by the execute processor is guaranteed by the order of accesses. The same logical identifier in several different instructions will access different slots in queues, providing correct operation without false data dependencies.

Supporting out-of-order execution is another way to increase the amount of available instruction-level parallelism available; however, this requires the use of hardware techniques such as Tomasulo's algorithm [Toma67] in addition to register renaming to eliminate dependencies among instructions [SoVa87] The adoption of out-of-order execution also raises the question of how to deal with interrupts, branches, and exceptions [SmPl88].

Architectural queues also support out-of-order execution between the two processors with respect to the original program sequence. The register renaming effect of removing storage conflicts allows data fetch instructions to complete before their corresponding positions in the original sequence of the program. This feature is one of the keys to the effectiveness of the DAE architecture because it allows the access processor to run ahead of the execute processor, continuously providing data to the execute processor and effectively hiding the memory latency.

5. Dynamic Loop Unrolling

Due to the limitation of instruction-level parallelism within basic blocks, loop structures are often unfolded and replicated multiple times [HePa90]. This technique, referred to as loop unrolling [John91], allows a code scheduler to arrange code across multiple iterations of a loop in order to exploit instruction-level parallelism across multiple basic blocks. Loop unrolling attempts to hide both memory and functional unit latency by migrating multicycle instructions to earlier locations in the instruction stream, reducing the effect of dependencies among instructions. Loop unrolling not only allows the exploitation of instruction-level parallelism across different iterations of a loop, but it also hides functional unit latencies by overlapping the executions of different iterations. It is a simple and effective method to extend the realm of instruction-level parallelism; however, because of dependent loop-indices, true data dependencies between loop iterations, and the decrease in code density, it is often only applicable to a limited extent.

As discussed above, in the DAE architecture the access processor runs ahead of the execute processor, migrating memory accesses and loop condition calculations to an earlier time relative to their uses. This *dynamic loop unrolling* makes DAE architectures relatively insensitive to memory latency. Studies have also shown that DAE can sustain higher throughput than other architectures like the Cray-1 in memory systems with high latency [SmWP86].

One advantage of dynamic loop unrolling over static loop unrolling is that, since the access processor leads the execute processor and determines the termination conditions for loops, it may be able to unroll a loop completely. Another advantage is that the DAE can unroll loops with characteristics which are difficult to unroll statically. By letting the access processor determine the branch outcome in most cases ahead of the execute processor, the potential for the branch decision latency to interfere with loop unrolling is hidden.

6. Performance Comparison

In order to compare the effectiveness of superscalar and DAE machines, a set of benchmarks had to be selected and the machines to be compared had to be defined. The Livermore Loops were chosen for this initial study because they are amenable to hand-coding, and are fairly representative of a large class of scientific programs. The following section describes in detail how the benchmark programs were generated and the exact configurations of the target architectures.

6.1. The Models

The first 12 Livermore Loops were compiled using the MIPS C Compiler 2.10 with level O2 optimization on a DEC 5000/200 workstation. Directed Acyclic Graphs (DAGs) were then constructed from the resultant intermediate code. These DAGS were used to hand scheduled code for the four machine models below using the instruction latencies given in Table 1.

The **Basic** machine is a sequential machine with one functional unit and memory latencies given in Table 1. The total number of cycles needed to execute a single iteration of a loop is given in Table 2.

The **Ideal** model assumes perfect register renaming and an infinite number of functional units each with unit latency. Essentially, this model provides the upper bound of available instruction-level parallelism. The result is similar to [Wall91], and is essentially the longest path through the DAG.

The **DAE** model assumes both access and execute processors have pipelined integer and instruction fetch units, while only the execute processor has a pipelined floating point unit. All data accesses are handled by the access processor. The code for this machine was generated by hand from the DAGs for each loop.

Table 1: Latencies of Different Instruction Classes

Funct	Latency	Funct	Latency	Funct	Latency
load	6	Int. Add	1	Int. Mult	4
FP Add	3	FP Mult	4	FP Convert	2
branch	3	Others	1		

The **Superscalar** (or **SS**) model assumes the existence of four fully pipelined functional units; one floating point unit, one load/store unit, and two integer units (one which is dedicated to address generation). Both integer and floating point units allow addition, subtraction, multiplication, and division of the corresponding data type. In addition, floating-to-integer, single-to-double and double-to-single converters are included in the floating point unit. Branch operations are handled in the integer unit - however, no branch prediction is performed. The central window is assumed to be discrete and large enough to hold the entire basic block for each of the 12 Livermore Loops, and that the issue width of the machine is 3. Memory address aliasing and cache misses are ignored. Finally, out-of-order execution is permitted. Therefore, instruction execution is only inhibited by busy functional units and true data dependencies. (this model is quite optimistic compared with any realistic superscalar design; the hardware requirement for the support of register renaming, out-of-order execution, and other hazard detection is enormous). As with the DAE machine, the DAGs were used to calculate the number of cycles necessary to execute each loop on this machine according to the functional unit latencies given in Table 1.

The **Unconstrained Superscalar** model (or **USS**) is the basic superscalar model with the addition of enough functional units to remove all but control and true data dependencies. Only hardware has been added - no software techniques have been applied (no loop unrolling has been performed, for example).

7. Observations

Table 2 shows the execution times of all 12 Livermore Loops on all four models. Assuming uniform clock rates and instruction latencies, only the speedups relative to the basic model are given. It can be seen that for 10 of the 12 Livermore Loops, the DAE outperforms the superscalar model. This is particularly noteworthy because the superscalar can issue up to 3 instructions/cycle, while the DAE is limited to 2. Superscalar does outperforms DAE on loops 8 and 9, which is due to the relatively high instruction-level parallelism exhibited by these two loops. This parallelism permits the dynamic scheduling of out-of-order execution in the 3-issue superscalar model to hide the functional unit latency more effectively than the DAE model.

DAE even outperforms the unconstrained superscalar on several loops. This can be explained by the fact that DAE hides the memory unit latency naturally, which superscalar machines cannot achieve without loop unrolling or other compiler techniques. The fact that no loop unrolling is performed is probably the most important impediment to better performance by the superscalar model.

To investigate the size of this effect on superscalar

Table 2: Performance Comparison among Models

Loop #	Inst Count	Base (cycles)	Speedup over Base			
			Ideal	DAE	SS	USS
1	14	30	3.75	1.5	1.25	1.25
2	19	30	4.28	1.5	1.15	1.43
3	10	19	3.8	1.73	1.19	1.19
4	13	21	3.49	1.91	1.16	1.16
5	12	22	3.67	2	1.2	1.2
6	14	21	3.45	1.91	1.16	1.16
7	32	65	4.33	1.1	1.03	1.38
8	96	178	12.7	1.38	1.98	4.23
9	41	89	6.36	1.53	1.59	2.11
10	57	78	2.8	1.69	1.5	1.59
11	9	18	3.6	2.57	1.2	1.28
12	9	17	3.4	2.83	1.06	1.21

performance, loops 11 and 12 were unrolled 4 times and speedups of 2.48 and 2.72, respectively, were calculated. It is no surprise that unrolling the loops provides a significant speedup in the superscalar machine. However, the DAE model still outperforms the unrolled superscalar version, because DAE machines do loop unrolling naturally.

8. Conclusions and Future Work

This paper has presented a comparison of Superscalar and Decoupled Access/Execute architectures. We have shown that DAE and its architectural queues support a limited version of both register renaming (through the FIFO queues) and out-of-order execution (the two threads of the DAE may be executed out of order with respect to the sequence of the original program, but not within the threads), and that with much less hardware the DAE model consistently outperforms a far more expensive superscalar model. It is true that very aggressive superscalar designs using loop unrolling, multiple copies of functional units, branch prediction, and register renaming can potentially come much closer to the ideal limit than can DAE, but it is at a much higher hardware cost.

DAE and Superscalar architectures take advantage of the same modes of instruction level parallelism through different mechanisms. With respect to implementation, however, the two are very different. Superscalar implementations require an enormous amount of hardware support and must consider clock-skew and other electrical problems caused by the extensive data and signal paths. DAE architectures, on the other hand, use two relatively simple processors coupled only through asynchronous architectural queues. Clock-skew and other electrical problems are much less intensive and the design should be significantly less complex and costly.

In addition, the DAE architectures scale more easily than the superscalar ones. Increasing the amount by which the access processor can run ahead of the execute processor requires only lengthening the queues while adding a larger central window to a superscalar can significantly increase the complexity of the processor.

In the future, similar experiments using larger benchmarks and compiler generated code will be required to more fully explore the relative capabilities of the two models. An analysis of the fault tolerance and reliability of the two approaches might be interesting as well.

9. References

[GHLP85] J. R. Goodman, J. T. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter and H. C. Young, ‘‘PIPE: a VLSI Decoupled Architecture’’, *Proceedings of the Twelveth Annual International Symposium on Computer Architecture*(June 1985), pp. 20-27.

[HePa90] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, San Mateo, California, (1990).

[John91] M. Johnson, *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, New Jersey, (1991).

[Kell75] R. M. Keller, ‘‘Look-ahead Processors’’, *ACM Computing Surveys*, vol. 7, no. 4 (December 1975), pp. 177-195.

[Smit82] J. E. Smith, ‘‘Decoupled Access/Execute Computer Architectures’’, *Proceedings of the Ninth Annual International Symposium on Computer Architecture*, Austin, Texas (April 26-29, 1982), pp. 112-119.

[SmWP86] J. E. Smith, S. Weiss and N. Y. Pang, ‘‘A Simulation Study of Decoupled Architecture Computers’’, *IEEE Transactions on Computers*, vol. C-35, no. 8 (August 1986), pp. 692-702.

[SDVK87] J. E. Smith, G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, C. M. Rozewski, D. L. Fowler, K. R. Scidmore and J. P. Laudon, ‘‘The ZS-1 Central Processor’’, *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California (October 1987), pp. 199-204.

[SmPl88] J. E. Smith and A. R. Pleszkun, ‘‘Implementation of Precise Interrupts in Pipelined Processors’’, *IEEE Transactions on Computers*, vol. 37, no. 5 (May 1988), pp. 562-573.

[SoVa87] G. S. Sohi and S. Vajapeyam, ‘‘Instruction Issue Logic for High-Performance, Interruptable Pipelined Processors’’, *Proceedings of the Fourteenth Annual International Symposium on Computer Architecture*, Pittsburgh, Pennsylvania (June 2-5, 1987).

[Toma67] R. M. Tomasulo, ‘‘An Efficient Algorithm for Exploiting Multiple Arithmetic Units’’, *IBM Journal*, vol. 11 (January 1967), pp. 25-33.

[Wall91] D. Wall, ‘‘Limits of Instruction-Level Parallelism’’, *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA (April 8-11, 1991), pp. 176-189.