Improving the Parallelism and Concurrency in Decoupled Architectures

L. K. John, A. Subramanian
Computer Science and Engineering
University of South Florida
Tampa, FL 33620

P. T. Hulina and L. D. Coraor
Computer Science and Engineering
Penn State University
University Park, PA 16802

Concurrency between access and execution has been exploited by queues in many decoupled access-execute architectures, but data dependent control dependencies often prohibit prefetching of data to queues. This paper investigates a technique to facilitate anticipatory loading to queues even in presence of data dependent control dependencies. The proposed method consists of fetching along one or both paths of a data dependent control dependency and inserting *consume* instructions in appropriate paths to consume the unnecessarily fetched data. The compiler hoists load instructions above control dependencies as in conventional load hoisting techniques. The technique is seen to be very effective in programs with data dependent if-then-else's. We also present an architecture with multiple access units, the mLSU architecture, which parallelizes the access process. Simulation experiments illustrate that multiple access units improve the performance if access processor instruction issue is a bottleneck.

Keywords: Access-execute decoupling, compiler optimization, load hoisting, code motion, memory access.

1 Introduction

The unending quest to faster computing in real-life applications is driving computer architects and designers to exploit as much parallelism as possible from the instruction stream. Exploitation of instruction level parallelism has led to the emergence of superscalar [13] [45], VLIW [8] [31], decoupled [38] [42] [12] [41] [33] [34] [23] [49] [9], multiscalar [10] [11] and other fine-grain parallel architectures. Decoupled access/execute (DAE) computer architectures [38] [42] [12] [41] [33] [34] [23] [49] attempt to separate access and execute processes in a job, provide concurrency between them, and yield high performance and increased flexibility. DAE architectures consist of an execute unit responsible for executing the core computation instructions in a program and an access unit responsible for generating operand addresses and accessing data from the memory. The two units are coupled through queues or FIFO buffers that serve to buffer data fetched by the access unit or hold the results to be stored back to memory. The independence of access and execute processes allows the access processor to fetch data elements ahead of demand by the execute processor and thus hide memory latency.

Compiler scheduling and rearrangement of instructions is required to minimize pipeline stalls in statically scheduled architectures. In dynamically scheduled processors, hardware (a scoreboard) could reorder instruction execution at runtime. Decoupled architectures incorporate principles of dynamic scheduling in easier and simpler manners than architectures with a scoreboard. The access and execute instruction streams are dynamically scheduled with respect to each other. The queues allow the memory access task to schedule itself dynamically ahead of the computations. This may also be viewed as a way of achieving dynamic register allocation. Load and store addresses are buffered in address queues. Load addresses may pass store instructions provided the addresses do not conflict. Decoupled architectures thus incorporate several techniques to achieve dynamic scheduling very easily. Using dynamic scheduling provides good performance on non-optimum code [40]. Software pipelining [26] [24] is essential in most pipelined architectures to reduce pipeline stalls. In decoupled architectures, the store address queue and other queues naturally yield the benefits of software pipelining and many other compiler optimizations.

Some decoupling between access and execution exists in all RISC architectures but decoupling has been explicitly implemented in a few commercial machines. The Astronautics ZS–1

[39] [43] implemented decoupling very explicitly while architectures such as i860, IBM RS/6000 and PowerPC implement fragments of the concept. The WM architecture [49] also accomplishes concurrent execution of accessing and computation operations and decoupling through queues, but the PowerPC architecture is perhaps the one commercially successful architecture that has imbibed many features of decoupled architectures. The POWER and PowerPC architectures [47] incorporate several queues as buffers in various stages of the pipeline and they overlap the execution of the fixed point and floating point unit.

1.1 Objectives

The objectives of this paper are to investigate techniques to improve the concurrency and parallelism in decoupled architectures. Traditionally DAE architectures work very well in most iterative computations, but in programs with data dependent control dependencies, the access unit does not 'slip' ahead of the execute unit. The access unit works in a lock-step fashion with the execute unit, with no concurrency. Due to this reason, in prior experiments [19] comparing software prefetching [35] [17] [36] with DAEs, in benchmarks with data dependent if-then-else's, software prefetching was observed to be superior. This paper presents a technique to facilitate anticipatory loading to queues even in presence of data dependent control dependencies. This technique improves the independence between the two processes while handling control dependencies. Thus the major contribution of the paper is a load hoisting scheme suitable for DAE architectures. A new instruction called *consume* that enables anticipatory prefetching in programs with data dependent control dependencies is introduced.

Another contribution is the design and performance evaluation of a multiple load/store unit decoupled architecture. Though some of our past research justified the need for multiple load/store units [19] [23], a detailed design was never presented. In this paper a detailed description of the architecture is presented. Architectures similar to the mLSU architecture have been briefly discussed in the past [46] [23] [4]; however, detailed performance evaluations have not been presented, especially those with nonnumeric programs.

1.2 Overview

In section 2, we describe the mLSU architecture with multiple access units. In section 3, the *consume* instruction that facilitates load hoisting above control dependencies is presented.

Performance evaluation of the proposed techniques is presented in section 4. Section 5 provides concluding remarks.

2 Architecture

We first present an architecture called the mLSU architecture (multiple Load/Store Unit architecture) that improves the parallelism in decoupled architectures. In an ideal DAE architecture, the access process will be completely hidden in the execute process and the execute process will always find the required data in the queues. But reality is significantly far from ideal. Past investigations [19] [23] have established that this is due to the imbalance and lack of concurrency between the access and execute sections of the workload. Our experiments [19] [23] demonstrated that often the execute process is waiting for the data to arrive in the queues. This is due to the heavy workload of the access unit combined with insufficient memory bandwidth.

One proposed technique to improve the parallelism and concurrency between access and execution involves executing multiple access instructions in each cycle. Issuing several access instructions will be effective only if the memory system supports multiple accesses and hence lock-up free caches and multiported or interleaved memories are required in these systems [22] [37]. Based on the nature of computations in frequently used algorithms, and our research on the MRU architecture [25], we reason that three load/store pipes would be sufficient. In the rest of this section, we present the details of the mLSU (multiple Load/Store Unit architecture) that parallelizes the access task. The architecture is illustrated in Fig. 1. It is comprised of one integer pipeline, one floating point pipeline, and three load/store pipes. Incorporating three load/store pipes helps to parallelize the access process which is essential for balancing accessing and execution. There are instruction queues in front of all the pipes, which allow a certain instruction stream to proceed even if another stream is blocked, essentially allowing the streams to be dynamically scheduled with respect to each other. There are separate load-queues for the integer execution unit and the floating point execution unit. ILQ1, ILQ2 are 32-bit integer load queues and FLQ1, FLQ2 are 64-bit floating point queues. ISQ is a 32-bit store queue and FSQ is a 64-bit store queue. This architecture has some similarities to the architectures in [46] and [4].

The memory access instructions will have dependencies with previous address computation instructions and unless the address computation and load/stores are performed by the same

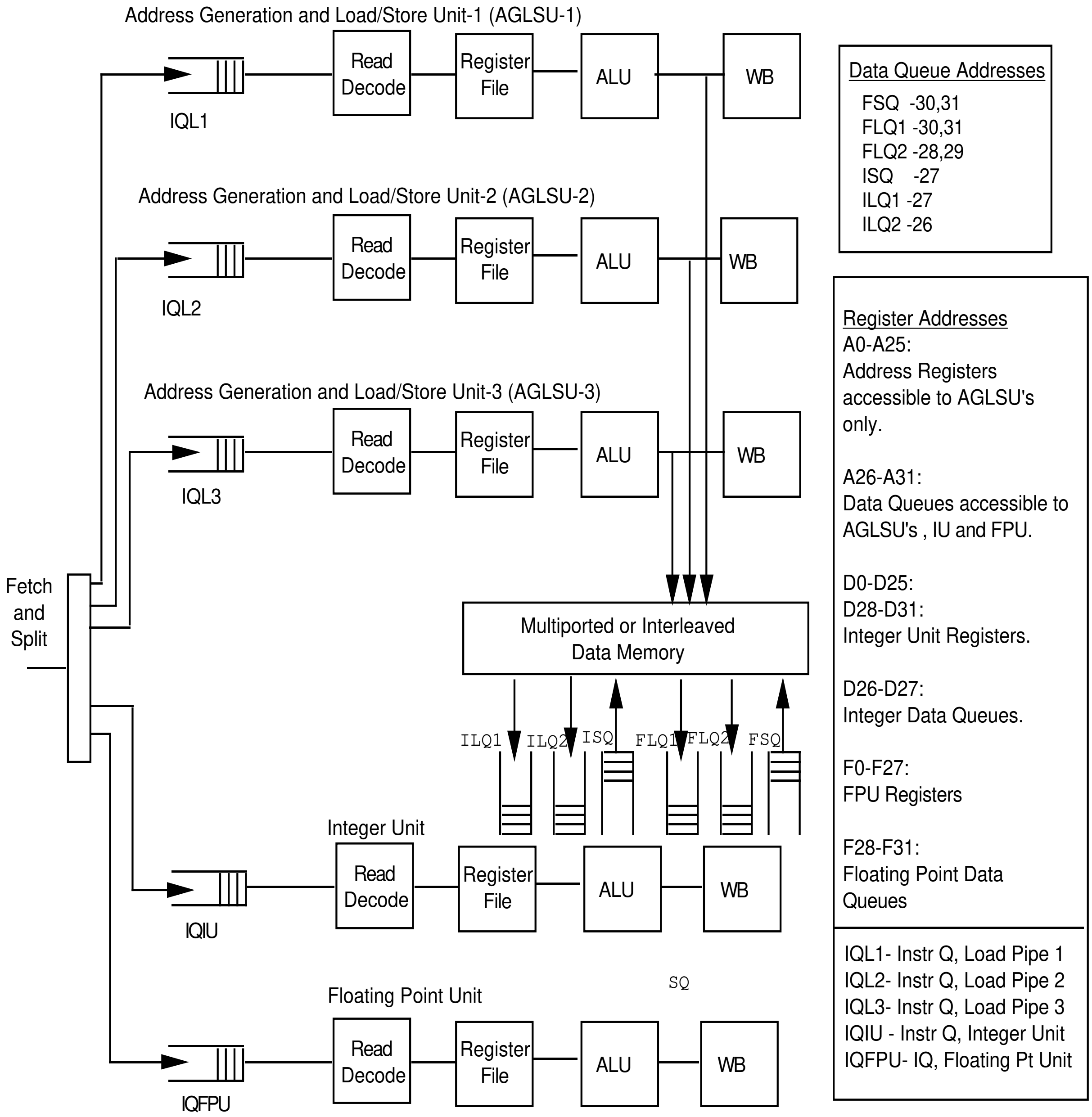


Figure 1: mLSU: A multiple load/store unit architecture that parallelizes the access task. Queue addresses for a possible implementation shown in side-bar.

AGLSU-1 operations	AGLSU-2 operations	AGLSU--3 operations	EP code
$32: load $30,0($1)	$32: lw $28,0($9)	$32: sw $30,0($15)	$32: mul $10,$30,a
addu $1,$1,incx*4	addu $9,$9,incy*4	addu $15,$15,incy*4	addu $30,$28,$10
bne $1,$2,$32	bne $9,$10,$32	bne $15,$16,$32	bneod
put eod to LQ1			

Figure 2: Code for saxpy for mLSU, the multiple load/store unit architecture

unit, there will be several dependencies between functional units. For this reason, and to provide concurrency between address generation and integer execution, we provide the load/store units with certain arithmetic and logic capabilities. Hence we call the load/store units as address generation and load/store units (AGLSU's). Addition and shift instructions are sufficient for most address computations because the multiplications involved in address computations are mostly multiplications by powers of two which can be accomplished by shifts. Branch capability may be added to each load/store unit or branches may be centrally resolved as in the PowerPC. Appropriate branch prediction schemes can also be incorporated.

To illustrate the operation of the mLSU architecture, a program example is provided in Fig. 2. The code segments for the AGLSUs are written in elementary operations in a MIPS style [16]. Two new instructions *puteod (put end of data)* and *bneod (branch-if-no-end-of-data)* have been added to the instruction set. We do not dwell on the details of these instructions since they have been explained elsewhere [19] [20].

In typical superscalar processors, there are two register files, the integer register file and floating point register file. Since the mLSU architecture contains 3 load/store pipes in addition to the integer execution unit, if a single register file is to be shared between the integer and load/store units, a large number of read/write ports would be required for the registers. The proposed architecture emphasizes on simplifying and minimizing the resource sharing between various functional units. A possibility is to use a *distributed register file* scheme in which each load/store unit can access registers of other load/store units, but parallel operations are permitted only if its own registers are accessed. Such a scheme can operate with 2 read ports and 1 write port for the registers. Since the access job is being parallelized into three threads, 8 registers would suffice for each pipe. Simultaneous accessing of each register file is thus possible without providing 3 read and 3 write ports. This scheme might necessitate one-time duplication of many address computation information like base addresses, indices, bounds etc, but would still be efficient in iterative computations.

In the PowerPC, floating point loads have to pass through both the integer and floating point pipelines because the destination register is in the floating point pipeline whereas the load process is initiated from the integer pipe. To minimize the interdependence between the two pipes, the mLSU architecture has data load queues (typical of DAE architectures, but missing in PowerPC) which are addressable from the load/store and floating point pipes. However, the existence of several independent pipelines make the implementation of precise interrupts difficult, and impossible without limiting the 'slip' between the pipelines. Hence, as in PowerPC, precise interrupts will be available only in a special mode with reduced performance.

3 The *consume* instruction: Facilitating Load Hoisting Above Control Dependencies

Concurrency between access and execution has been exploited by queues in many decoupled access-execute architectures, but data dependent control dependencies often prevent the access process from 'slipping' ahead of the execute process or prefetching data. In prior experiments [19] comparing software prefetching [35] [17] [36] with DAEs, in benchmarks with data dependent if-then-else's, software prefetching was observed to be superior. DAEs generally do not allow anticipatory fetching of data that may not be used. In DAEs, the access and execution were happening in a lock-step fashion with each processor waiting for the other to finish. As a solution to this problem we propose to add a new instruction *consume* Q, which can dequeue an element from the queues. The proposed *consume* Q operation is equivalent to moving a queue element to a fictitious register. The MIPS R2000 processors provide a zero-value read-only register (R0). If such a register is available in the execute unit, moving the head of the queue to that register would accomplish the intended operation.

The compiler hoists load instructions above control dependencies, (as would be done in conventional load hoisting techniques), and adds *consume* Q instructions in appropriate paths to dequeue unnecessarily loaded data. The load hoisting can be performed using algorithms developed in the past for software pipelining and software prefetching. The only additional step would be to add a consume in the path taken if the condition is to fail. The process of load hoisting and consume insertion is illustrated in Fig.3. If a load between nodes C and F along the path of E is hoisted above C, a *consume* instruction should be added between C and F along path D. Similarly this or another load along A-C could be hoisted above A, and a *consume* inserted along B.

The use of the *consume* Q instruction is illustrated in Fig. 4. In the program excerpt illustrated in Fig. 4.a, after P(i) is computed, there is a conditional branch depending on its value. In a naive decoupled architecture, the fetching of $Q(i)$ or $R(i)$ does not occur until the branch is resolved and the execute processor informs the access processor as to which path to take. The *consume* Q instruction makes it possible to hoist loads above control dependencies.

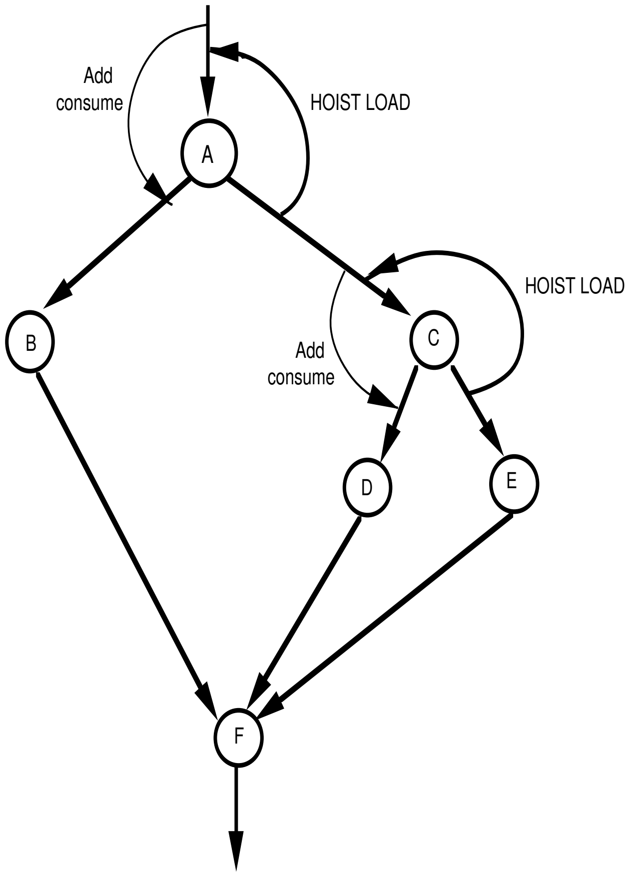


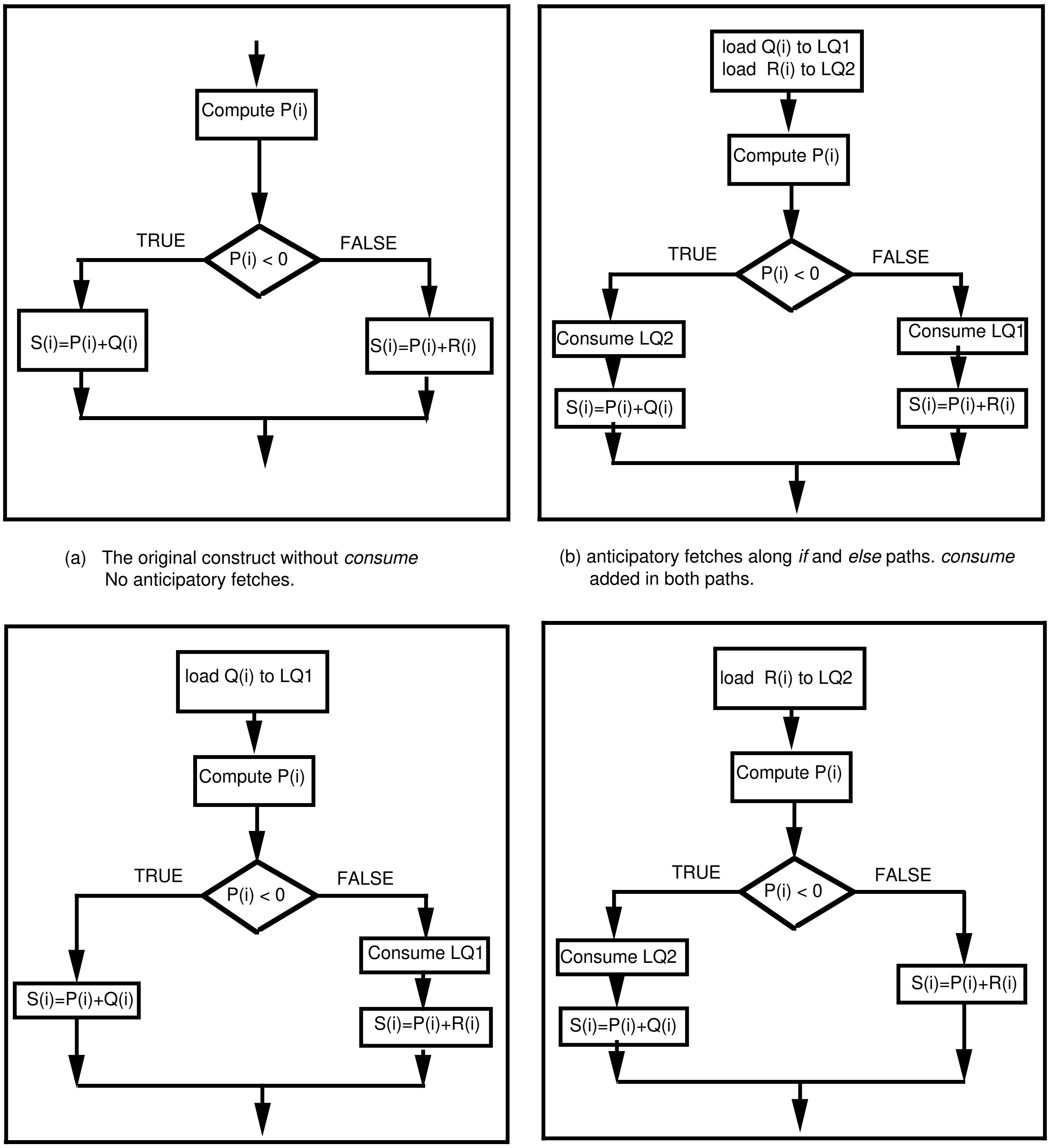
Figure 3: Load hoisting and insertion of consume instructions

Prefetching may be performed along one or both paths of the conditional branch. Fig. 4.b illustrates the case when loads along both the TRUE and FALSE paths are hoisted. Under this condition, the access processor does not have to wait for the outcome of the branch testing the value of $P(i)$. Each path after the branch contains a *consume* instruction to remove the data the other path would have needed. Fig. 4.c illustrates the case when loads along the TRUE-path are hoisted and Fig. 4.d illustrates the case when loads along the FALSE-path are hoisted.

The consume instruction can be successfully employed in routines such as the C language string copy *strcpy*. To copy a string, the access unit can continuously load the data into the queues, while the execute unit checks for the null delimiter and deposits the string elements to the store queue. When the delimiter is encountered, the execute unit sends an *end-of-data (eod)* token to the access unit. The access processor probably has fetched beyond the end of the string. Upon receiving the token, the access unit stops loading the string, sends an *eod-acknowledge* to the execute unit and starts prefetching future data. The execute unit *consumes* the queue elements until this *eod-acknowledge* token is received. These synchronization primitives improve the independence between the access and execute units.

As in software prefetching, the proposed scheme results in slight code expansion. Just as each PREFETCH instruction in software prefetching needs one cycle to issue, one cycle will be needed to issue each *consume* instruction. However, the consume instructions are in the E-stream (execute stream) and not A-stream (access stream). One could imagine building a DAE architecture with an access processor that performs software prefetching to cache. However, the proposed approach with *consume* carries more merit because the extra instructions are in the execute stream. Previous research has shown that access task is more heavy compared to execute task and hence the addition of the consume instruction in the E-stream is a good choice.

The proposed scheme can lead to a problem if the program is bandwidth limited. When loads along both paths of the branch are hoisted, the system is definitely going to fetch more data than it actually needs and if bandwidth is already a bottleneck, the problem will be aggravated. But this is a problem common to all aggressive prefetching techniques including software prefetching and not in particular to the proposed scheme. Anticipatory fetches increase bandwidth requirement and memory traffic, and are undesirable if memory bandwidth is already a bottleneck; but the capability to prefetch is essential to hide latency if some extra bandwidth



(a) The original construct without *consume* No anticipatory fetches.

(b) anticipatory fetches along *if* and *else* paths. *consume* added in both paths.

(c) Prefetching along the if-path. *consume* is added in the else-path

(d) Prefetching along the else path. *consume* is added in the if path

Figure 4: The consume Q instruction makes it possible to hoist loads above control dependencies. Prefetches may be performed along one or both paths of the conditional branch.

is available.

4 Performance Evaluation

We simulate a baseline uniprocessor architecture, a conventional DAE architecture, and a multiple load/store unit DAE architecture. The MIPS instruction set [16] is used as the core instruction set and appropriate enhancements are done depending on the case being experimented. For instance, for software controlled prefetching, a *prefetch* instruction as in [17] is added to the instruction set. For DAEs, special instructions such as *put-eod*, *bneod*, *consume*, etc are added. Some of the unused opcodes of the MIPS processor are used to represent these instructions (as in [36]). The baseline processor supports nonblocking loads and a lockup-free cache [18]. Compiler optimizations such as loop unrolling, software pipelining and software prefetching are incorporated in the baseline performance so that the baseline performance is the best uniprocessor performance.

A four-way sequentially interleaved memory system is assumed for the simulations. Each memory bank is 32 bits wide. The system bus is also 32 bits wide. All memory bank conflicts during accesses are considered. A 1 Kbyte instruction cache and a 1 KByte data cache are assumed to be present. Our benchmarks are small programs and hence the instruction cache hit-ratio is very high for most cache configurations. The cache sizes are unrealistic for modern microprocessors, but they have been intentionally kept small because our benchmarks are small. The cache block size is 8 bytes because lower block size keeps spurious effects from memory traffic to a minimum. The caches are 4-way set associative, and LRU replacement policy is employed. We did perform experiments with different block sizes, different associativity, etc, and the observations concur to results presented in previous cache studies, and hence we are presenting only the results from one typical cache configuration. The memory access time is varied from 5 processor cycles to 15 processor cycles, so that the impact of latency on the techniques could be studied. The data cache is nonblocking with four Miss Status Holding Registers (MSHRs).

We certainly appreciate the value of being able to incorporate the different techniques in a compiler and use large benchmarks such as the SPEC. But we faced the same problem as Rogers and Li [36] did, and without a compiler that would apply these optimizations, we were forced to intervene in the code compilation process. Lack of a compiler prevented us from using large benchmarks, and we perform simulations with a set of eleven programs which include four programs with data dependent control dependencies. One of these four programs is the IFLOOP which has a data dependent IF-THEN-ELSE construct inside every iteration. The other programs considered are string copy (*strcpy)*, string compare (*strcmp*), and a binary tree search algorithm. The IFLOOP program is illustrated below.

```
for (k=0; k≤ max; k++) {
    di = y[k] - (g[k] / (xx[k] + dk));
    if (di≤ 0) x[k] = z[k] /di;
        else x[k] = w[k];
    }
```

Double precision data and arithmetic are assumed for the IFLOOP. This is a benchmark in which there is significant arithmetic. The two double precision division operations consume several cycles and hence the access task associated with this program is not much larger than the execute task. The other three benchmarks do not have any real computations. In string copy and string compare, inherently there is very little execute task. The execute processor has no task except checking for the delimiter. The binary tree search algorithm *binsrch* also used in [3] assumes that the tree is stored in the data memory with each of its nodes consisting of four fields stored in consecutive locations. The four fields are the key of the node, pointer to the left child, pointer to the right child, and the data respectively. Access tasks dominate the instruction stream for the *binsrch* program. The different characteristics of these different benchmarks reveal the different features of the architectures and techniques being studied.

Total program execution time is the ultimate measure of performance and we use execution time as the performance metrics. Traces are generated from the DEC station 5000 which uses MIPS R3000/3010 processors. The benchmark sources are compiled with the highest level of optimization (-O4). Some trace preprocessing was done at this stage to suit the trace to the different architectures being studied.

Fig. 5 illustrates a comparison of execution times for conventional DAE (naive DAE), naive DAE plus consume, mLSU architecture, mLSU architecture plus consume, and the baseline uniprocessor with software prefetching.

The various factors that affect performance include memory latency, memory bandwidth and

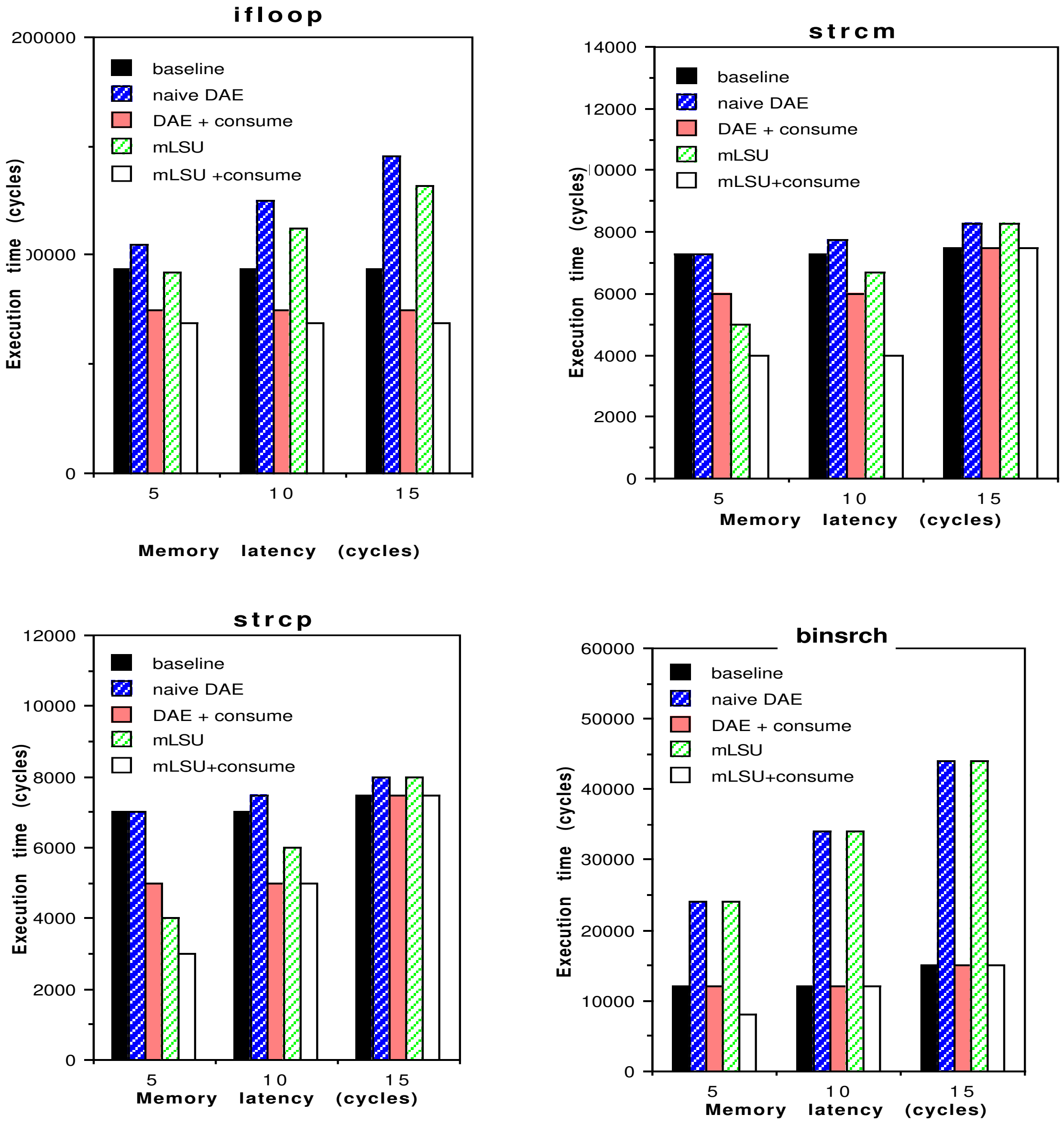


Figure 5: Comparison of execution times for conventional DAE (naive DAE), naive DAE plus consume, mLSU architecture, mLSU architecture plus consume, and the baseline uniprocessor with software prefetching

access processor instruction issue bottleneck. If bandwidth is not a bottleneck, latency hiding techniques such as the proposed scheme will be very effective. Consider the graphs for *ifloop* in Fig. 5. The naive DAE and mLSU with no *consume* exhibit poor performance compared to the baseline, DAE with *consume* and mLSU with *consume* architectures. The worst performance is from the naive DAE architecture. The access processor waits on the execute processor to finish its computations and resolve the data dependent branches and hence it cannot prefetch data efficiently. In IFLOOP, the memory bandwidth does not become a bottleneck at the memory latencies experimented. This is due to the large cycles taken by the floating point division compared to the memory latencies we were experimenting. With four-way interleaved memory, there was sufficient bandwidth for the latencies we experimented. Once sufficient memory bandwidth is available, prefetching will be very effective in hiding latency as illustrated by the good performance of DAE with *consume* and mLSU with *consume*.

In *strcp* and *strcm*, the majority of the instructions are access related. Hence, at lower latencies, access processor instruction count is the major bottleneck. The mLSU architecture can parallelize the access stream and hence it performs significantly better compared to both uniprocessor and naive DAE. One may observe this in Fig. 5 at latency of 5 processor cycles. Memory bandwidth becomes a bottleneck at higher latencies. At latency of 15 cycles, the naive DAE and mLSU architectures take more time because they are incapable of prefetching and hiding latency. In fact the data cache has contributed significantly to the performance of every case including the naive DAE and mLSU. The data size is only a byte but each cache block fetches 8 bytes and thus the cache implicitly performs a significant amount of prefetching.

In *binsrch*, the performance of naive DAE and mLSU are worse than other architectures even at lower latencies. This is because the access processor gets information to go to the left child or right child only after the execute processor checks the current key. Since all the four items of each node of the tree is stored in consecutive locations, increasing the cache block size to fetch the left and right children's addresses along with the key would improve this situation. Currently, only one child's address gets fetched with the 8 byte block cache, but this child may or may not be the one required.

In general, the performance gains from the mLSU architecture are more significant at lower latencies when instruction issue is a bottleneck. At higher latencies, memory latency is likely to be a bottleneck rather than instruction issue and under that condition, if extra bandwidth is available, the *consume* instruction is very effective in hiding memory latency.

Fig. 6 illustrates the impact of latency on conventional DAE (naive DAE), naive DAE plus consume, mLSU architecture, mLSU architecture plus consume, and the baseline uniprocessor with software prefetching.

The two architectures that exhibit high sensitivity to latency in all the benchmarks are naive DAE and mLSU. This is due to the inability of these architectures to prefetch. The access processor waits for the execute processor to resolve the data dependent branches before it fetches data for computations after the branch.

In programs with sufficient bandwidth, DAE with consume, mLSU with consume and the baseline uniprocessor yield very good performance and perfect insensitivity to latency by prefetching. One example program with this behavior is IFLOOP. In *strcp* and *strcm*, mLSU has very good performance at lower latency, however, memory bandwidth is a bottleneck at higher latencies and all techniques are sensitive to latency once bandwidth becomes a bottleneck.

Another technique that may be of merit in certain cases is loop splitting. Consider the IFLOOP with 3 unconditional loads which provide data for a computation, the result of which is tested and one of two possible paths taken depending on the outcome. The loads in the conditional path prevents the naive access processor from fetching the data for the next iteration. The performance of such a scenario can be improved by loop splitting. The loop is split into two smaller loops, the first one with the unconditional loads and the second one with the conditional loads. In this case, naive decoupling could improve the performance of the first loop at least. However, this scheme may necessitate additional intermediate storage. DAE architectures need this technique only if *consume* is not present.

The performance potential of the mLSU architecture did not become very evident from the four benchmarks chosen above. Fig. 7 illustrates the performance of the mLSU architecture for a collection of vector programs at a memory latency of 5 processor cycles. The benchmarks used for these include saxpy with equal increments (*s.eq*), saxpy with unequal increments (*s.uneq*), convolution, correlation, and three Lawrence Livermoore Loops. These results illustrate that mLSU architecture can significantly improve the performance when multiple vectors are being accessed simultaneously.

Simulations on large benchmarks will be performed after modifying the SUIF compiler [48] to generate code for the mLSU processor with the proposed consume instruction.

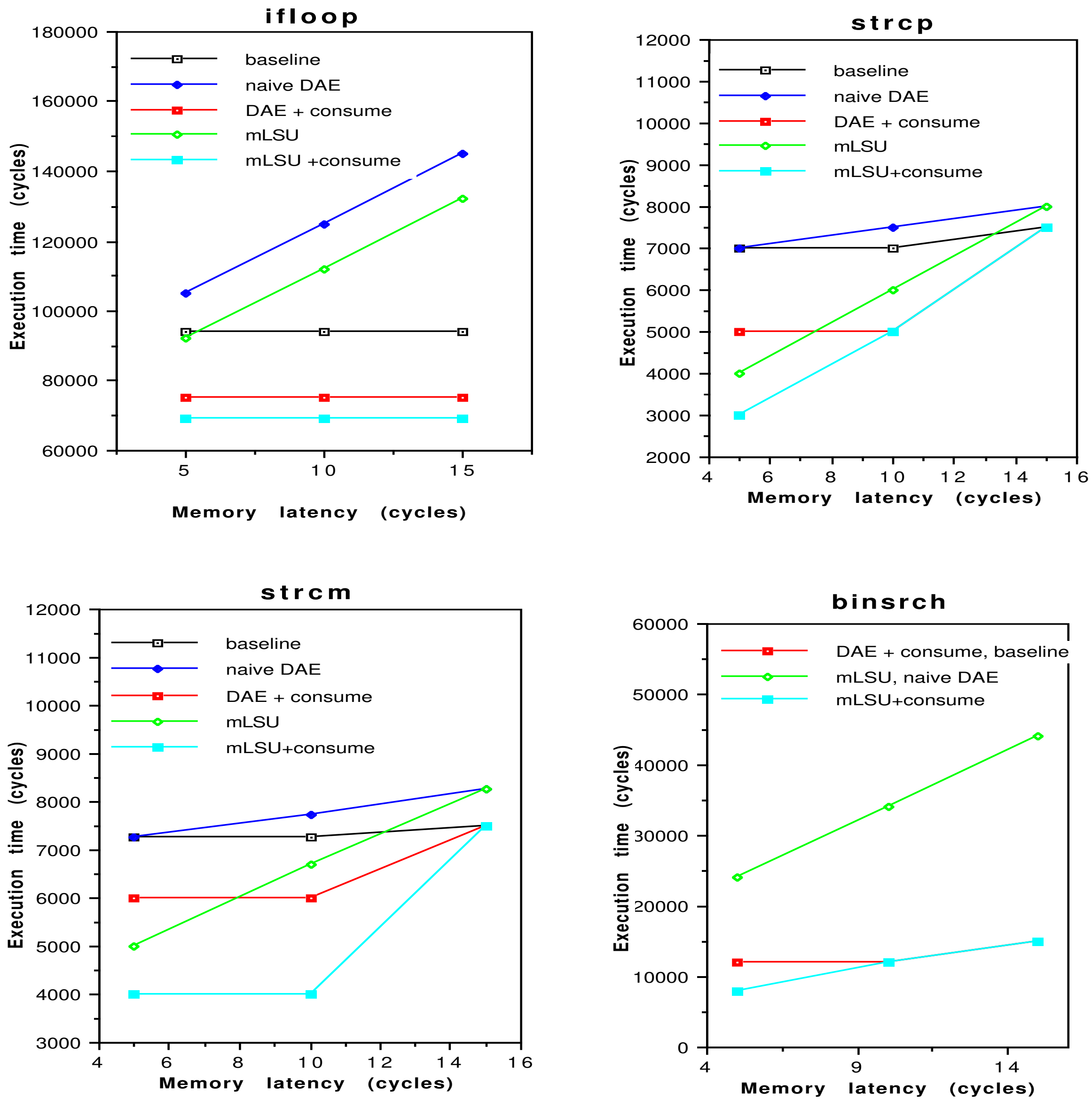


Figure 6: Latency sensitivity of the five schemes under comparison

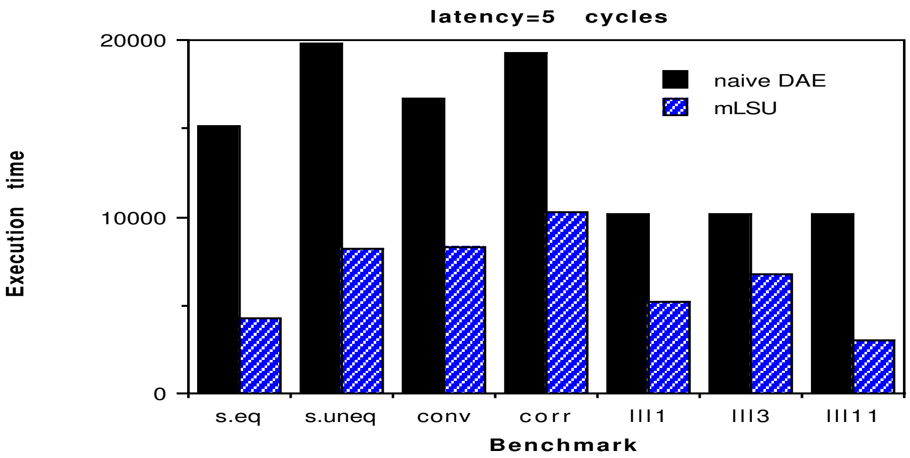


Figure 7: mLSU performance for vector programs

5 Summary and Conclusion

In this paper, we investigated a new technique to improve the concurrency in decoupled architectures. A new instruction called *consume* that will dequeue unnecessarily fetched elements from the queues is introduced. This instruction removes the complications associated with anticipatory fetching into data queues. Decoupled architectures with a *consume* instruction are very effective in hiding memory latency in programs with data dependent control dependencies, provided memory bandwidth is not a bottleneck. It is often advantageous to prefetch beyond control dependencies. If the memory bandwidth requirement of the program is high, aggressive prefetching is not effective, but the capability to prefetch is essential to hide latency if some extra bandwidth is available.

We also presented a multiple load/store unit decoupled architecture called mLSU architecture that parallelizes the access task. The performance gains from the mLSU architecture are more significant at lower latencies when instruction issue is a bottleneck. At higher latencies, memory bandwidth is likely to be a bottleneck rather than instruction issue. The mLSU architecture is particularly effective for vector programs that use several vectors simultaneously.

References

[1] R. D. Acosta, J. Kjelstrup, and H. C. Torng, "An instruction issuing approach to enhancing performance in multiple functional unit processors", IEEE Transactions on Computers, C-35, No. 9, pp. 815-828, September 1986.

[2] D. W. Anderson, F. J. Sparacio, R. M. Tomasulo, "Machine Philosophy and Instruction Handling", IBM Journal of Research and Development, 11, No.1, 8-24, January 1967.

[3] A. Berrached, P. T. Hulina and L. D. Coraor, "A Decoupled Access/Execute Architecture for Efficient Access of Structured Data", 26th Hawaii International Conference on System Sciences, January 1993.

[4] A. Berrached, P. T. Hulina and L. D. Coraor, "Structured Data Access Mechanisms for a Decoupled Computer Architecture", Proceedings of the 1994 International Conference on Parallel Processing, Vol. II, pp. 285-289, August 1994.

[5] M. E. Benitez and J. W. Davidson, "Code Generation for Streaming: an Access/Execute Mechanism", Proceedings of ASPLOS-IV, pp. 132-141, 1991.

[6] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu., "IMPACT: An Architectural Framework for Multiple Instruction Issue Processors", Proc. of the 18th Annual International Symposium on Computer Architecture, Toronto, Canada, May 1991, pp. 266-275

[7] Cohn R., Gross T., Lam M, and Tseng P.S., "Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor", Proc. 3rd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, 1989, pp. 2–14.

[8] Colwell R.P, Nix R.P., O'Donnell J.J., Papworth D.B., and Rodman P.K., "A VLIW Architecture for a Trace Scheduling Compiler", IEEE Transactions on Computers, Vol.37, No.8, August 1988, pp. 967-979.

[9] R. Espasa and M. Valero, "Decoupled Vector Architectures", HPCA-2.

[10] M. Franklin and G. S. Sohi, "The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism", Proceedings of the 19th International Symposium on Computer Architecture, Australia, May 1992, pp. 58-68.

[11] M. Franklin, "The Multiscalar Architecture", Ph. D. thesis, Computer Science Department, Univ of Wisconsin- Madison, 1993.

[12] J.R. Goodman, J.T. Hsieh, K. Liou, A.R. Pleszkun, P.B. Schechter, and H.C. Young, "PIPE: A VLSI Decoupled Architecture", 12th Annual International Symposium on Computer Architecture, (June 17-19, 1985), Boston, Massachusetts, pp.20-27.

[13] Grohoski G.F., "Machine Organization of the IBM RISC System/6000 processor", IBM Journal of Research and Development, vol.34, pp. 37–58, January 1990.

[14] Hulina P.T., and Coraor L.D., " Coprocessor architectures for efficient address computation and memory accessing", Computer Systems Science and Engineering, Vol.5, No.3, July 1990.

[15] P.T. Hulina, L.D. Coraor, and S.W. Sun, "Performance Analysis of an Address Generation Coprocessor", IEEE International Conference on Parallel Processing, 1991.

[16] G. Kane, "MIPS RISC ARCHITECTURE", Prentice Hall, NJ, 1989.

[17] A. C. Klaiber and H. M. Levy, "An architecture for software-controlled data prefetching", 18th Intl. Symp. on Computer Architecture, May 1991, pp. 43–53.

[18] D. Kroft, "Lockup-free Instruction Fetch/Prefetch Cache Organization", Proc. of the 8th Annual Intl. Symp. on Computer Architecture, pp. 81-87, June 1981.SA Ames Research Center, Moffett Field, CA, July 1992.

[19] L. Kurian, "Issues in the Design of a Decoupled Architecture for a RISC environment", Ph.D. thesis, Computer Engineering, The Pennsylvania State University, August 1993.

[20] L. Kurian, P. Hulina and L. Coraor, "Memory Latency Effects in Decoupled Architectures with a Single Memory Module", Proceedings of the 19th International Symposium on Computer Architecture, May 1992, Queensland, Australia, pp. 236-245

[21] L. Kurian, P. Hulina, and L. Coraor, "Memory Latency Effects in Decoupled Architectures", IEEE Transactions on Computers, Vol. 43, No. 10, pp. 1129-1139, October 1994.

[22] L. Kurian, B. Choi, P. Hulina and L. Coraor, "Module Partitioning and Interlaced Data Placement Schemes to Reduce Conflicts in Interleaved Memories" *Proceedings of the 23rd International Conference on Parallel Processing*, Vol. I., pp. 212-219, August 1994.

[23] L. Kurian, V. Reddy, P. Hulina, L. Coraor, "Program Balance and its Impact on High Performance Architectures", Proceedings of the International Symposium on High Performance Computer Architecture, Jan 1995, pp. 37-379.

[24] L. Kurian-John, V. Reddy, P. Hulina, L. Coraor, "A Comparative Evaluation of Software Techniques to hide Memory Latency", Proceedings of the 28th Hawaii International Conference on System Sciences, pp. 229-238, Jan 1995.

[25] L. Kurian-John, P. T. Hulina, L. Coraor, and E. John, " Design and VLSI Implementation of an Address Generation Coprocessor", IEE Proceedings on Computers and Digital Techniques, Vol. 142, No. 2, March 1995.

[26] M. S. Lam, "Software pipelining: An effective scheduling technique for VLIW machines", Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, pp. 318–328, 1988.

[27] M. S. Lam and R. P. Wilson, "Limits of Control Flow on Parallelism", Proceedings of the 19th International Symposium on Computer Architecture, Australia, May 1992, pp. 46-57.

[28] S. A. Mckee and W. Wulf, "Access Ordering and Memory Conscious Cache Utilization", Proceedings of the High Performance Computer Architecture Symposium, Jan 1995, pp. 253- 262.

[29] Melvin S. and Patt Y., "Exploiting Fine–Grained Parallelism Through a Combination of Hardware and Software Techniques", Proc. of the 18th Annual International Symposium on Computer Architecture, Toronto, Canada, May 1991, pp. 287-296

[30] Mowry T.C., Lam M.S. and Gupta A. "Design and Evaluation of a Compiler Algorithm for Prefetching", Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems", October 1992, pp. 62 – 73.

[31] Nicolau, A and Fisher, J.A., "Measuring the Parallelism Available for Very Long Instruction Word Architectures", IEEE Transactions on Computers C-33, No.11, November 1984, pp. 968–976.

[32] S. Palacharla, J.E. Smith, "Decoupling Integer Computation in Superscalar Processors", To appear in the Proceedings of MICRO-28.

[33] A. R. Pleszkun, and E.S. Davidson, "Structured Memory Access Architecture", IEEE International Conference on Parallel Processing 1983, pp 461–471.

[34] A. R. Pleszkun, G.S. Sohi, B.Z. Kahalleh, and E.S. Davidson, "Features of the Structured Memory Access (SMA) Architecture", Third IEEE Computer Society International Conference, San Francisco, CA, March 1986.

[35] A. Porterfield, "Software Methods for Improvement of Cache Performance on Supercomputer Applications", Ph. D. dissertation, RICE COMP TR 89-93, May 1989.

[36] A. Rogers and K. Li, "Software Support for Speculative Loads", *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 38-50, October 1992.

[37] G. S. Sohi and M. Franklin, "High Bandwidth Data Memory Systems for Superscalar Processors", *Proceedings of ASPLOS-IV*, pp. 53-62, 1991.

[38] J.E. Smith, "Decoupled Access/Execute Computer Architecture", ACM Transactions on Computer Systems, Vol.2, No.4, (November 1984), pp 289-308.

[39] J.E. Smith, et .al. "The ZS–1 Central Processor", Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, Palo Alto, CA, (October 1987), pp 199–204.

[40] J.E. Smith, " Dynamic Instruction Scheduling and the Astronautics ZS–1", IEEE Computer, (July 1989), pp.21–35.

[41] J.E. Smith, A.R. Pleszkun, R.H. Katz, and J.R. Goodman, "PIPE: A High Performance VLSI Architecture", IEEE Workshop on Computer System Organization, New Orleans, LA, (March 1983), pp.131–138.

[42] 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.

[43] W.M. Smith, S.G. Abraham, and E.S. Davidson, "A performance comparison of the IBM RS/6000 and the Astronautics ZS-1", IEEE Computer, (January 1991), pp.39–46.

[44] W.M. Smith, S.G. Abraham, and E.S. Davidson, "The Effects of Memory Latency and Fine–Grain Parallelism on Astronautics ZS–1 Performance", Proceedings of the 23rd Annual Hawaii Intl. Conf. on System Sciences, CS Press, Los Alamitos, Calif, Order No.2008, (1990), pp.288-296.

[45] Wall D.W., "Limits of Instruction-Level Parallelism", Architectural Support for Programming Languages and Operating Systems, 1991, pp. 176–188.

[46] S. Weiss, "Multiport Memory Access in Decoupled Architecture Processors", *Proceedings of the International Conference on Parallel Processing*, I-373-I-376, 1991.

[47] S. Weiss and J. E. Smith, POWER and PowerPC: Principles, Architecture, Implementation, Morgan Kaufman Publications, 1994.

[48] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S-W Liao, C-W Tseng, M. Hall, M. Lam, and J. Hennessy, "An Overview of the SUIF Compiler System", SUIF Release I Notes, Stanford University.

[49] W.A. Wulf, "Evaluation of the WM Architecture", Proc. 19th Intl. Symp. on Computer Architecture, Australia, May 1992, pp.382–390