Memory Latency Effects in Decoupled Architectures

Lizyamma Kurian
Computer Science and Engineering
University of South Florida
Tampa, FL 33620
Phone: (813)-974-2114
Fax: (813)-974-5456
kurian@sol.csee.usf.edu

Paul T. Hulina and Lee D. Coraor
Computer Science and Engineering
The Pennsylvania State University
University Park, PA. 16802
Phone: (814)-863-4253
Fax: (814)-865-0192
pth@ecl.psu.edu

Abstract

Decoupled computer architectures partition the memory access and execute functions in a computer program and achieve high performance by exploiting the fine–grain parallelism between the two. These architectures make use of an access processor to perform the data fetch ahead of demand by the execute process and hence are often less sensitive to memory access delays than conventional architectures. Past performance studies of decoupled computers used memory systems that are interleaved or pipelined, and in those studies latency effects were partially hidden due to interleaving.

A detailed simulation study of the latency effects in decoupled computers is undertaken in this paper. Decoupled architecture performance is compared to single processors with caches. The memory latency sensitivity of cache based uniprocessors and decoupled systems is studied. Simulations are performed to determine the significance of data caches in a decoupled architecture. It is observed that decoupled architectures can reduce the peak memory bandwidth requirement, but not the total bandwidth, whereas data caches can reduce the total bandwidth by capturing locality. It may be concluded that despite their capability to partially mask the effects of memory latency, decoupled architectures still need a data cache.

Index Terms: Access/Execute Overlap, Access Processor, Cache Memories, Memory Bandwidth, Memory Bottleneck, Memory Systems, Performance Analysis, Pipelined Computers, Simulation.

* *This work was supported in part by the National Science Foundation under grant number MIP-8912455.*

1. Introduction

The execution of a computer program involves two interrelated processes – accessing data elements from memory and the true computations. A large amount of parallelism exists between these two tasks. The concurrent execution of these tasks can result in high performance and this is the principle of decoupled access execute architectures. Many early high performance computers such as the IBM 360/370, CDC 6600, and CRAY–1 incorporated some techniques to exploit the parallelism between access and execute tasks, but several architectures in the past few years like the MAP-200 [4], DAE [19] [24], PIPE [8] [23], SMA [15], SDP [17], FOM [2], ZS–1 [20] [25] [26] and WM [28] partition access operations and computation functions in the program more distinctly.

Almost all of the aforementioned architectures consist of two processors, one to perform address calculations and load and store operations, and the other to operate on the data and produce results. The two processors are often referred to as the access processor and execute processor respectively. The essence of the job is the part performed by the execute processor, but the execute processor cannot perform its role without the information furnished by the access processor. In decoupled architectures FIFO buffers or queues are provided between the access and execute processors to maximize the overlap and independence of the two processors. The independence of the two processes allows the access processor to fetch data elements ahead of demand by the execute processor. This phenomenon has been called *slip* by previous researchers [9] [24] [27]. Slip actually refers to the extent of *decoupling* between the access and execute processes.

Memory latency effects in decoupled architectures have been studied in [24], [8], [26], [28] etc. Smith et al [24] compared the performance of a pipelined decoupled architecture to that of the scalar CRAY–1. This particular research effort also included studies on the effect of memory latency by varying the access time of main memory. Since a comparison is made with the CRAY–1, an interleaved memory configuration, (as in the CRAY–1), is assumed. Memory bank conflicts are also ignored. Here memory access time was varied from 5 cycles to 32 cycles, but since the memory system was 16 way interleaved, some of the memory latency effects may have been hidden. The first twelve Lawrence Livermore Loops were used as the simulation workload.

Goodman et al evaluated the performance of PIPE, a VLSI decoupled architecture using the Lawrence Livermore Loops [8] [9]. This work includes the effects of memory speed on performance by conducting studies on both a fast and a slow memory module. The fast memory had an access time of one clock cycle and the slow one had an access time of four clock cycles and was four way interleaved. The systems included a memory controller. With the overheads incurred in the controller, the effective delay seen in the case of the fast memory is 3 cycles and it is 6 cycles for the slow memory [9]. They also ignore memory module conflicts in their study.

Smith, Abraham and Davidson [26] reported results on the effects of memory latency and fine grain parallelism on the Astronautics ZS–1, when connected to a pipelined memory system. They observed that once the slip limits were high, the computer system is almost insensitive to memory latency. This study is also based on the Lawrence Livermore Loops.

Wulf [28] presented preliminary results on the performance of the WM architecture. Though specific memory latency studies are not performed, it is mentioned that data FIFOs would partially mask the effects of memory latency. They also comment that the probable Achilles heel of the architecture would be to build a memory system capable of supplying the bandwidth that the processor can absorb.

1.1 Objective

The performance study presented in this paper has three goals: first to compare the performance of decoupled computers to uniprocessor systems with caches, second to study memory latency sensitivity of decoupled computers with a non-interleaved non-pipelined memory, and third to determine the significance of a data cache in a decoupled architecture.

The performance of decoupled computers is compared against single processors without caches in the previous studies. Conceptually, in decoupled systems, we are using a processor to do the access task and eliminate the delay for data operands. In cache based systems, we utilize a cache to capture locality and eliminate the long delays required to access main memory. It would be interesting to find out how the two schemes compare to one another. In this paper, we investigate whether an access processor hides the memory latency of a computer system better than a data cache. Hence we perform a comparison of the performance in decoupled mode against uniprocessors with caches.

Another objective of this paper is to study the sensitivity of a decoupled architecture to memory latency, when the memory is noninterleaved and nonpipelined. Previous studies report that decoupled computers have less sensitivity to memory path length than conventional systems [24] [8]. It is also reported that the speed up over a single processor configuration becomes greater as the memory becomes slower. But as mentioned before, these studies used interleaved or pipelined memory systems. We want to study whether system behavior will exhibit a similar pattern even when the memory is noninterleaved. We are not suggesting that decoupled architectures do not need interleaved memories. Our aim is only to isolate the memory latency insensitivity contributed by 'decoupling'.

A third objective of this paper is to study the significance of data caches in decoupled architectures when the memory system is not interleaved. Decoupled architectures generally do not have data caches. In the architectures described above, only the ZS–1 has a data cache. The reduced sensitivity to memory access time observed in the previous studies tend to suggest that the improvement possible with a data cache would be minor. Interleaving and memory pipelining obscure some of the memory latency and we suspect that, this is one reason for the insensitivity to longer memory cycle times in the studies in the past. We investigate whether a data cache would result in any serious performance advantage in decoupled architectures with noninterleaved memory units.

1.2 Overview

In section 2, we briefly describe the decoupled architecture that is used to conduct our performance studies. The description illustrates that this architecture is very similar to other decoupled architectures and hence results obtained from this system should apply at least qualitatively to other decoupled architectures as well. In section 3, we analyze the mechanism by which decoupled computers alleviate the memory latency problem. In section 4, we explain our simulation tools and describe the benchmarks used. In section 5, we detail the simulation results obtained. We compare cache based uniprocessors to decoupled systems, study the sensitivity of decoupled systems to memory latency, examine the limitations of caches and decoupled architectures in eliminating memory latency, and study the significance of a data cache in a decoupled architecture. We conclude the paper in section 6.

2. The DEAP Architecture

In this section, we describe the Decoupled Execute Access Processor (DEAP), which was used to conduct our simulation studies. The DEAP architecture uses two processors, an Execute Processor (EP) and an Access Processor (AP) as shown in Fig.1. As in standard DAE (decoupled access execute) architectures [19], the EP executes program instructions and performs all the required data computations. All accesses of the data memory are done by the AP. The two processors communicate through architectural queues. To avoid memory contentions between the EP and the AP, each processor is equipped with a separate memory unit. Access related instructions are fetched by the AP and computation instructions are fetched by the EP into their respective instruction caches. Data operands needed by the EP are fetched from the data memory unit by the AP and passed to the EP via architectural queues. Results of EP computations are deposited into the queues and transferred to the data memory by the AP. The EP has no access path to the data memory. AP instructions are stored in an instruction memory private to the AP to avoid bus contentions with the data fetch. Since pin limitations of the AP in a VLSI implementation might pose a problem, the AP instructions could also be stored in the global data memory unit. This will not degrade the performance in problems with loops since the AP has an instruction cache. Our performance results in this paper are based on the DEAP architecture with AP instructions stored in the same memory unit as the data. The architecture exists only in simulation form at this time.

The AP and the EP see entirely different instruction streams. The code is split at compile time in such a way that all computation instructions are put in the EP section of the code and the address calculation and access instructions are put in the AP section. At execute time two different instruction streams enter the two processors from the two respective instruction memories. The AP makes address calculations and performs data memory accesses and furnishes the EP with the data it requires for the computations. The EP is thus free to perform its data computations while the AP is waiting for access requests to be satisfied by the memory.

The two instruction streams run at their own speed, with the queues absorbing excess data. Unlike PIPE, there are only two queues in DEAP, the Read Queue and the Write Queue (See Fig. 1). The EP reads data from the Read Queue and stores its results into the Write Queue. It may be noticed that the queues are named with reference to the EP and not the AP. Wherever coordination is required, the AP and EP use tokens to ensure correct operation. While accessing multiple-element data structures, the AP uses *End-of-Data* (EOD) tokens to separate batches of data such as an array, a column of a matrix, or the entire matrix. The AP can use the Read Queue to pass this EOD token to the EP. The EP uses these tokens to control its iterations. There is no potential problem due to sending such control information intermingled with data in the queues. Explicit instructions are used to deposit tokens and also to access them. Correctness of the program can be ensured as long as the EOD token is deposited only after the completion of any issued load instruction. In addition to keeping the system simple, a two queue implementation is as efficient as a system with separate data and control information queues. Since we use tokens to denote *End-of-Data*, it would necessitate using one more bit, called the EOD bit for each element of the queue.

In problems with static data bounds, the AP has prior knowledge about when to insert the EOD tokens. In problems with dynamic bounds such as the C library string copy (*strcpy*) and string compare (*strcmp*) where the end of the string is not known until it is actually encountered, the scenario is different. To exploit any advantage of a decoupled architecture in such an application, the AP can fetch the string elements one by one without checking for the delimiter. The EP can perform the comparison to find the end of the string. Due to slip, the AP would have fetched beyond the end of the string by the time the EP finds the end of string. In such a case, the EP sends an EOD token to the AP whereupon the AP stops its fetch operation and also flushes any unnecessary data it has fetched. The EP can meanwhile continue operations on its own, but it should not read any data until the AP has flushed its queue. This can be accomplished if the AP sends a token to the EP to mark that it has flushed the queue, and the EP waits for the token before reading data operands from the queue again. This creates a certain amount of busy waiting but it seems to be inevitable, if any parallelism is to be exploited in such a problem.

3. Memory Latency and the Access Processor

The Access Processor (AP) eliminates latency of main memory by performing the access process ahead of demand by the execute processor. When a program is looping, the access instruction stream often precedes the execution stream by at least one iteration. The Read Queue buffers the prefetched data. The execute processor does not have to wait to obtain data for its computations. The access processor would have already loaded the data into the Read Queue. Hence if sufficient slip is present, the EP obtains its operands with no delay. Similarly, in the case of memory writes, the EP can put the data into the Write Queue and proceed. The AP would store it back into the main memory later. It is assumed that the queue, like registers, can be accessed in a single cycle. Hence, if the AP can run ahead and load the queue with the data before the EP reaches the section of the code with a reference to the data, and if the Write Queue is long enough for the EP to dump its result and proceed, the EP never experiences a delay in accessing main memory. The AP thus hides memory latency from the EP.

The length of the queues is a critical factor in a decoupled architecture since often the distance the access process can run ahead is limited only by the queues. A slow memory access path can be compensated for by using longer queues [24]. With sufficiently long queues, a high average transfer rate can be achieved even with a memory of relatively low peak transfer rate capacity. The queues enable the system to utilize given memory bandwidth more efficiently. The memory latency insensitivity that can be achieved by a decoupled architecture depends on the slip that can be attained.

In the case of very fast memories, the address calculation instructions consume a significant fraction of the total execution time and there is a limit on the slip that can be attained. With slower memories, the environment permits more slip. But memory poses a bottleneck and slip is limited again as the memory speed decreases beyond a certain point, which can be considered as an optimum memory speed. This optimum is not a constant, but will depend on the characteristics of the program under execution, such as the fraction of load and store instructions, the relative AP and EP workload etc.

4. Simulation Methodology

Performance simulators are developed for a uniprocessor, actually the MIPS R2000 [11] and a DEAP system with access and execute processors having the MIPS instruction set. For DEAP, modifications necessary in the R2000 for queue operations and the EP-AP interface are assumed. The simulators are written in C and run on a DEC 3100 station under the UNIX operating system. The results from the MIPS R2000 system form the baseline for comparison.

The AP and EP instructions are pipelined through fetch-decode-ALU-writeback stages in a fashion similar to that in MIPS R2000 [11], but with hardware interlocks. The R2000 tries to achieve single cycle execution for its instructions by delayed load and delayed branch techniques. The uniprocessor that forms the baseline system for comparison as well as the EP in the decoupled implementation incorporate these techniques in identical manners so that the effect of decoupling could be easily identified. For the decoupled mode, the length of the queues was kept at 20 in our simulations. It has been reported in [24],

[29] and [8] that short queues are sufficient to achieve performance close to the maximum available with unbounded queues. We also performed some experiments on queue lengths. Our observations confirm past results, except that loop unrolling slightly increases the queue length requirements.

We performed simulations with some of the Lawrence Livermore Loops (LLLs), two signal processing algorithms *convolution* and *correlation*, the *saxpy* routine from the LINPACK benchmark and the C library string copy *strcpy*. The LLLs were chosen since they were used in research in the past [24] [8] [9] and since they are important to a wide range of scientific algorithms and applications. The signal processing algorithms used contain addressing patterns other than sequential [10]. They exhibit good locality properties also. Since our studies involve cache based systems, these algorithms are very relevant for our studies. The *saxpy* routine from the linpack benchmark is run with both loop increments equal to one and also with unequal increments. The *strcpy* routine operates on character data which is 8 bits wide, while the other benchmarks use data that is 32 bits wide.

The benchmarks are compiled with the DEC 3100 workstation compilers with the highest level of optimization. The assembly output from the compiler is machine coded in the required trace format for the uniprocessor cases. For the decoupled version, the assembly code is manually split into the access and execute streams and coded into the required trace format. We could not include results from large benchmarks such as the SPEC due to the difficulty in generating the two streams of traces without a compiler for the decoupled system. The DEC compiler performs loop unrolling for most of the loops we used. If the uniprocessor trace is unrolled, the same degree of unrolling is retained in the traces for the decoupled system also.

5. Discussion of Simulation Results

In this section we present our simulation results. We first compare the performance of decoupled systems with single processors with caches. In order to relate our work to previous research, we also simulate uniprocessors without caches. During simulation runs, we vary the main memory access time to find out the sensitivity of system performance to memory path length. The simulation results are analyzed to identify limitations of decoupled architectures and cache based systems. Finally, the relevance of a data cache in a decoupled architecture is studied by simulating a system with a cache in the AP. At this stage a comprehensive comparison of uniprocessors with and without caches and decoupled systems with and without caches is presented. Similar studies with handcoded traces were presented in [13].

5.1 Comparison with Uniprocessors with Caches

The performance of the decoupled system in comparison with uniprocessors with and without caches is shown in Fig. 2. Memory cycle time is denoted by t_{mm} and is expressed in terms of processor cycles. Since block sizes and cache sizes can have a critical effect on the performance of cache based systems, we performed simulations with a few different cache parameters. But in order not to clutter the figures with too much data, we plot only one typical organization of the cache with a size of 1024 bytes. This cache size might seem to be unrealistically small, but it should be remembered that the benchmarks used are small too. The cache is assumed to have a single cycle access time. In Fig. 2, it can be seen that in the 5 cycle case, the decoupled architecture executes the code faster than the uniprocessor with and without cache, while at 15 cycles the uniprocessor with cache performs better than the decoupled system for several of the traces. The behavior is similar for all traces except for *strcpy*. For *strcpy*, even at $t = 5$ cycles, the uniprocessor with cache is superior to the decoupled system. The *strcpy* trace is unique in that the data element size is smaller than the bus-width and that multiple data elements can be fetched in one access.

5.2 Sensitivity of Performance to Memory Access Time

The variation in execution time with increase in memory cycle time is illustrated in Fig. 3 and Table I. Three types of behavior can be observed Fig. 3 for memory latency sensitivity. The first graph corresponds to *strcpy* trace which is able to make use of the cache due to its spatial locality, and the uniprocessor with cache exhibits less sensitivity to memory access time, than the decoupled system. The second graph illustrates the typical behavior of benchmarks which do not make use of any locality and both uniprocessors with caches and decoupled architectures exhibit the same range of variation in execution time. The third graph illustrates convolution and correlation traces that contain true temporal locality and cached uniprocessors exhibit significant insensitivity to memory access time, whereas decoupled architectures are sensitive to the access time. Table I illustrates that for convolution, correlation and *strcpy*, the increase in execution time for decoupled architecture is significantly higher than that in cached uniprocessors. For the other five traces, both cached uniprocessors and decoupled architectures exhibit the same range of variation.

Benchmark	Uniprocessor with cache	Decoupled architecture
lll1	12022	10422
lll3	20000	20000
lll11	19524	20065
convolution	1938	31560
correlation	1070	30580
saxpy-eq	29987	29915
saxpy-un	24497	25435
strcpy	5030	40018

Table I. Increase in execution time (cycles) with tripling of memory cycle time

Fig. 4 illustrates the change in speed up of the decoupled system as the memory access time is varied. Results reported in [24] and [8] indicate that the speedup with a decoupled organization improves as the memory speed decreases. For some of the loops, we do observe the effect they reported, but we also notice that beyond a certain memory speed, the speedup declines. (More loops showed the effect reported in [24] and [8], when we used handcoded traces in [13].) In [8], though it is mentioned that the performance advantage is more significant with a slower memory, actually 5 out of the 12 Lawrence Livermore loops they used show a smaller speedup for the slower memory module. Since they considered interleaved memories, this decrease in speed-up after certain latency was not evident in the other loops in the ranges of memory speed they used.

5.3 Limitations of Decoupled Architectures

Decoupling can smooth out burst bandwidth requirements. But if the total bandwidth requirement is higher than the time the execute processor would take to complete its section of the code, memory becomes a bottleneck unless some technique to alleviate the bottleneck is incorporated. We quantify the memory bandwidth problem in decoupled architectures by comparing the total access time requirement to the pure computation time.

The time the EP takes to complete its section of the code assuming that it always finds the requested data in the load queue without waiting and that it is able to deposit the result into the store queue without waiting is the EP execution time with a perfect memory and perfect AP. Let us denote this time as the *EP stand-alone execution time.*

The total memory access time (which is analogous to bandwidth requirement) should be less than EP stand-alone execution time if sufficient memory bandwidth is available, but Fig. 5 shows that the memory access takes 1.8, 3.6 or 5.4 times the time the EP takes to complete its computations, at memory speeds 5, 10 and 15 cycles respectively. If the memory system was capable of furnishing the required bandwidth to complete the entire access in a time period less than the EP stand-alone execution time, the decoupled system would have performed better.

Decoupled architectures can eliminate memory latency only as long as the total time required for data fetch can be accommodated within the time the execute processor would consume to complete its section of the code without having to wait for any operands. Beyond that point, the effect of memory latency will be evident in the total execution time. This explains the sensitivity that decoupled systems exhibited to memory access speed in our simulations. It might be noted that increasing the queue length cannot hide the latency, once memory has become a bottleneck like this. Decoupled architectures can reduce the peak bandwidth requirement, but not the total bandwidth. Caches can reduce the total bandwidth requirement by capturing locality, but have other limitations which are addressed in section 5.5.

Load unbalance between the access and execute processors might also limit the speed up that can be achieved by the decoupled configuration. Typical general purpose instruction streams contain more access related instructions than true computations. The access processor often has to execute more instructions than the execute processor, as illustrated by the instruction counts in Table II. Severe load unbalance exists in *saxpy.unequal*. All the traces used for the reported results are generated with the highest level of compiler optimization (-O4) and no address calculation instructions appear within the loop in them except in *saxpy.unequal*. Code with less optimization exhibits more AP-EP unbalance due to the presence of address calculation instructions. The ratio of AP load to EP load is less than 2:1 in *saxpy.equal* where the optimizer was successful in removing the address calculation. In *saxpy.unequal*, which has detailed address calculation in each iteration, the corresponding ratio is greater than 5:1. The average ratio of AP instruction count to EP instruction count is 2.15:1. We have performed extensive studies on memory bandwidth, AP-EP unbalance and other bottlenecks in decoupled architectures. Due to space constraints, we cannot present all our results here, but interested readers may refer to [14] where we present all our results.

Benchmark	AP instr count	EP instr count	AP count/EP count
convolution	5000	3975	1.26
correlation	11083	5926	1.87
saxpy.unequal	12253	2251	5.44
saxpy.equal	3502	2251	1.56
lll11	2765	2501	1.11
strcpy	10012	6001	1.67

Table II. AP and EP instruction counts and their ratio

Another limitation of decoupled processors relates to the overhead incurred in the process of exploiting the access execute parallelism in the computing process. Slight code expansion results when AP-EP code is generated. Branch instructions must appear in both processors. If the execute processor does not support operations with both operands from the same queue (Read Queue), one of the operands has to be moved from the queue to a register which also contributes to code expansion. This problem can be alleviated by having two Read Queues, with the two data elements being loaded to alternate queues. In such a case, two register addresses have to be used for the queues.

In certain problems, such as the transposition of a matrix, array copy, etc. the whole problem is of an access nature and there is no role that the execute processor can play. Effort to parallelize these problems can result in serious overhead that may increase the execution time to more than the uniprocessor mode. If the compiler recognizes such problems, the DAE system should not be slower than a uniprocessor.

5.4 Significance of a Data Cache in a Decoupled Architecture

In section 5.1 we observed that uniprocessors with data caches performed considerably better than decoupled architectures, for slow memories. (See t=15 cycle case in Fig. 2.) This observation naturally leads to the question of whether a decoupled architecture could also benefit from caches. We performed simulations to investigate this and the results appear in Fig. 6. The total execution time of each benchmark is plotted for decoupled systems with caches, uniprocessors with and without caches and ordinary decoupled architectures. The results are characteristic of the memory referencing behavior of each benchmark. For the convolution and correlation algorithms, the decoupled system with caches performs better than the other systems. This can be attributed to the strong temporal locality present in the data references in these benchmarks. The *strcpy* program benefits from spatial locality, and for this benchmark also, decoupled system with cache exhibits superior speedup than other systems. The LLLs and the saxpy do not benefit from caches. For these benchmarks, the cache has a similar effect in both uniprocessors and decoupled architectures.

$t{=}15 cycles$

Benchmark	Uniproc with cache	Decoupled	Decoupled with cache
LLL1	0.96	1.10	1.05
LLL3	0.95	1.22	1.14
LLL11	0.97	1.13	1.29
saxpy-eq	0.96	1.16	1.11
saxpy	0.96	1.15	1.10
Mean	0.96	1.15	1.14
Correlation	3.35	2.22	6.29
Convolution	3.95	2.19	5.15
strcpy	2.97	1.06	3.63
Mean	3.42	1.82	5.02

$t{=}5 cycles$

Benchmark	Uniproc with cache	Decoupled	Decoupled with cache
LLL1	0.93	1.04	1.00
LLL3	0.89	1.65	1.37
LLL11	0.94	1.44	1.32
saxpy-eq	0.92	1.46	1.29
saxpy	0.96	1.43	1.35
Mean	0.93	1.41	1.27
Correlation	1.59	2.39	3.08
Convolution	1.89	2.47	2.47
strcpy	1.45	1.19	1.91
Mean	1.64	2.02	2.49

Table III. Speedup comparison

The speedup figures for memory access times equal to 5 cycles and 15 cycles are presented in Table III. Speedup is calculated with reference to uniprocessors without caches.

Mean values of the speedup figures are shown separately for programs with different locality characteristics. The Lawrence Loops and saxpy do not benefit from caches and form one group. Convolution, correlation and *strcpy* which exhibit locality characteristics form another group.

5.5 Limitations Associated with Caches

Caches hide the latency of the main memory by capturing the temporal and spatial locality of the data references. Locality of reference enables caches to reduce the bandwidth requirements of programs. It was illustrated in the previous section that data caches can improve the performance of decoupled architectures also. But there are several limitations associated with caches.

Lack of temporal locality limits the capability of some problems to exploit the cache. In Fig. 2, one can note that caches cause an increase in execution time for some of the benchmarks. Consider Lawrence Livermore Loop 3.

```
      q = 0.0
      do 3 k = 1,1000
3      q = q + z(k)*x(k)
```

Here the program steps through the arrays and hence there is spatial locality. In our study, the elements of the array have a 32 bit representation and the data bus width is 32 bits as well. The main memory is organized with a word size of 32 bits and each access furnishes one word or 32 bits. If a block size of 4 bytes is used, only the array element in demand is loaded each time there is a miss. Since no array element is used again (or in other words there is no temporal locality), a cache of 4 byte blocks does not improve the performance. Having a larger block size would exploit the spatial locality, but since fetching a larger block requires a proportionately larger number of cycles, the cache does not decrease the total execution time. The cache can yield an advantage if some of these fetch cycles could run in parallel with computation cycles when the CPU is busy, but does not use the bus. Otherwise, the cache just slows down the system by adding the cache access time to each reference. This phenomenon can be observed in all the LLLs in Fig. 2. The execution time for the cache based system is higher than that for the system without the cache. This could have been avoided by looking into the cache and the main memory at the same time, in which case, there would have been no deterioration, but still there would be no improvement. We thus notice that problems with only spatial locality sometimes do not benefit from caches. If the data size is smaller than a word, or in other words, if more than one data element could be fetched at the expense of one fetch, performance improvement could be obtained. Among the benchmarks we used, *strcpy* is the only trace which exhibits this characteristic. This benchmark does achieve strong insensitivity to memory latency.

Success of a cache organization depends on minimizing the miss ratio, the delay due to a miss, and the penalty for updating main memory. The number of fetches required to load a block of a given size, depends on the bus width. Larger block sizes capture spatial locality and may decrease the miss ratio, but increases both the delay due to a miss and also the updating penalty. The increase in hit ratio obtained with larger block sizes may mislead designers and architects. It should be cautioned that in several cases, the bus traffic and the memory bandwidth requirements increase dramatically with increase in block size and the overall effect is a reduction in performance despite the increase in hit ratio. A typical example is shown in Fig. 7 for *saxpy* with unequal increments. The execution time increases threefold when the block size is increased from 4 bytes to 16 bytes. Similar behavior was observed in several other traces with only spatial locality. Another fact that can be observed in this figure is that the limitations associated with caches affect both uniprocessors and decoupled systems identically. The sensitivity of performance to block size is also illustrated for the correlation algorithm in Fig. 7, and it can be observed that this benchmark which has strong temporal locality does not exhibit such sensitivity to cache block size. Also, the execution time in the system with the cache is always less than that of the system without a cache. A general observation is that temporal locality is often stronger than spatial locality.

It may be concluded that precise tuning of cache parameters is essential for the success of cache organizations. As also observed in [12] [1] and [21], unless carefully designed and implemented, caches may result in minimal performance improvement, or may even constitute a burden.

6. Conclusion

We have presented simulation results on the memory latency effects in decoupled access execute architectures. Since caches are a time-tested mechanism to solve the memory access problem, we also compared the decoupled architecture performance to uniprocessors with caches. We see that caches and decoupled systems achieve their best performance in different domains, since their mechanisms to alleviate memory bottleneck depend on different characteristics of a computer program. There are cases in which both might result in a performance advantage and there are problems where one scheme might not contribute significantly to the performance. Caches have the potential of efficiently hiding the main memory latency in programs that exhibit strong locality properties but they may also slow down the system if not carefully designed. If one fixed organization of the cache is going to be used for all applications, there is heavy risk of the cache affecting the system adversely under some conditions. Another observation is that temporal locality often produces stronger effects than spatial locality.

In the case of decoupled systems, we note that there is more scope for improvement from 'decoupling' and 'slip', once the main memory is fast enough to provide the bandwidth that the processor demands. With very slow memory, the effect of memory latency will be clearly evident in the total execution time. Memory poses a bottleneck here and decoupled computers cannot lower the execution time below the total memory access time. But even in that region, programs might benefit from caches if strong temporal locality is present. In spite of the memory posing a bottleneck in some cases, the speedup of the decoupled system relative to a noncached uniprocessor is significant.

We also performed simulations to determine whether decoupled architectures can obtain a performance advantage from data caches. The contribution of caches is minor when the main memory is fast. But in cases with strong temporal locality, decoupled architectures with caches achieve a level of memory path insensitivity superior to all other configurations. It can be concluded that caches are as relevant to decoupled computers as they are to uniprocessors.

Though we used noninterleaved memory for the studies in this paper, the major conclusions in this paper will hold for systems with interleaved memories also, once memory latency is very high. Interleaving can hide latency effects up to a certain latency, but bandwidth is likely to become a bottleneck, beyond a certain latency. Caches would be significant in decoupled architectures with interleaved memory also, once memory bandwidth is a bottleneck. Using non-interleaved memory simply enabled us to see the effects with low latency itself.

References

[1] D.B. Alpert and M.J. Flynn, "Performance Trade-offs for Microprocessor Cache Memories", IEEE Micro, (August 1988), pp. 44-53.

[2] W.C. Brantley and J. Weiss, "Organization and architecture tradeoffs in FOM", presented at IEEE Int. Workshop Comp. Syst. Organization, New Orleans, LA, (March 1983).

[3] L.D. Coraor, P.T. Hulina, and D.N. Mannai, "A Queue– based Instruction Cache Memory", Proc. of the International Symposium on Computer Architecture and Digital Signal Processing, (October 1989), Hong Kong, pp.281-286.

[4] E.U. Cohler and J.E. Storer, "Functionally parallel architectures for array processors", IEEE Computer, vol.14, (Sept. 1981), pp.28–36.

[5] R.J. Eickemeyer and J.H. Patel, "Performance Evaluation of On-chip Register and Cache Organizations", Proc. 15th Int. Symp. on Computer Architecture, (1988), pp. 64-72.

[6] M.K. Farrens and A.R. Pleszkun, "Improving Performance of Small on–chip Instruction Caches", Proc. 16th Int. Symp. on Computer Architecture, (1989), pp. 234-241.

[7] M.K. Farrens and A.R. Pleszkun, "Implementation of the PIPE processor", IEEE Computer, (Jan. 1991), pp. 65–70.

[8] 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.

[9] J.T. Hsieh, A.R. Pleszkun, and J.R. Goodman, "Performance Evaluation of the PIPE Computer Architecture", Technical Report #566, Computer Sciences Department, University of Wisconsin-Madison, (Nov. 1984).

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

[11] G. Kane, MIPS RISC Architecture, Prentice–Hall, Englewood Cliffs, N.J., (1988).

[12] L. Kurian, P.T. Hulina, L.D. Coraor, and D.N. Mannai, "Classification and Performance Evaluation of Instruction Buffering Techniques", 18th Intl. Symposium on Computer Architecture, Toronto, Canada, (May 1991), pp.150–159.

[13] L. Kurian, P.T. Hulina, and L.D. Coraor, "Memory Latency Effects in Decoupled Architectures with a Single Data Memory Module", 19th Intl. Symposium on Computer Architecture, Australia, (May 1992), pp.237–245.

[14] L. Kurian, P.T. Hulina, and L. D. Coraor, "Bottlenecks in Decoupled Architecture Performance", Technical Report TR-92-115, Computer Engineering Program, The Pennsylvania State University, November 1992.

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

[16] 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).

[17] R.R. Shivley, "Architecture of a Programmable Digital Signal Processor", IEEE Trans. on Computers, vol.C–31, (Jan. 1982).

[18] A.J. Smith, "Cache Memories", ACM Computing Surveys, Vol.14, No. 3, (September 1982), pp. 473-530.

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

[20] 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.

[21] A.J. Smith, "Line (Block) Size Choice for CPU Cache Memories", IEEE Transactions on Computers, Vol. C-36, No. 9, (September 1987), pp. 1063-1075.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] G.S.Sohi and E.S. Davidson, "Performance of the Structured Memory Access Architecture", Proc. of the International Conference on Parallel Processing, (August 1984), pp.506–513.

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

[29] H.C. Young, and J.R. Goodman, "A Simulation Study of Architectural Data Queues and Prepare-to-Branch Instruction", IEEE Intl. Conf. on Computer Design, (Oct. 1984), pp. 544–549.