Memory Latency Effects in Decoupled Architectures with a Single Data Memory Module*

Lizyamma Kurian, Paul T. Hulina and Lee D. Coraor
Department of Electrical and Computer Engineering
The Pennsylvania State University
University Park, PA 16802

January 22, 1996

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. We undertake a simulation study of the latency effects in decoupled computers when connected to a single, conventional, noninterleaved data memory module so that the effect of decoupling is isolated from the improvement caused by interleaving. We compare decoupled computer performance to single processors with caches, study the memory latency sensitivity of the decoupled systems, and also perform simulations to determine the significance of data caches in a decoupled computer architecture. The Lawrence Livermore Loops and two signal processing algorithms are used as the simulation benchmark.

Keywords: Decoupled Architecture, Access Processor, Performance Evaluation, Memory Latency, Caches.

1 Introduction

Decoupled computer architectures exploit fine–grain parallelism existing between access and execute functions in a computer program. The memory access task and execution task required for accomplishing a job are actually two interrelated but independent processes. Decoupled computer implementations attempt to separate these two processes to provide concurrency between them and give high performance and increased flexibility.

Some separation of access and execute functions has been a feature of several early high performance computers such as the IBM 360/370, CDC 6600, CDC 7600 and CRAY–1. Several computer architectures have been proposed or built later, which partition access operations and computation functions in the program more distinctly. MAP-200 [4], DAE [18], PIPE [8] [22], SMA [14], SDP [16], FOM [2], ZS–1 [19] [24] [25] all belong to this category.

Performance of decoupled architectures has been studied in [23], [8], [25] etc. Smith et al [23] compared the performance of a pipelined decoupled architecture to that of 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 CRAY–1), is assumed. Memory bank conflicts are ignored too. Here memory access time was varied from 5 cycles to 32 cycles, but since the memory system is 16 way interleaved, the memory can be considered to be very fast. It is observed in this work that the performance improvement over the scalar CRAY–1 becomes more significant as the memory path length increases.

Goodman et al evaluated the performance of PIPE, a VLSI decoupled architecture [8]. The performance in decoupled mode is compared against the single processor mode. 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; the slow one had an access time of four clock cycles and was four way interleaved. They included a memory controller. With the overheads incurred in the controller, the effective delay seen in 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 [25] 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.

1.1 Objective

The performance study presented in this paper has three goals: first to compare the performance

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

of decoupled computers to uniprocessor systems with caches, second to study memory latency sensitivity of decoupled computers with a single non-interleaved non-pipelined memory module, 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 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. It is also reported that the speed up over 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.

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 mentioned 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 [23]. 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 detail the decoupled architecture that is used to conduct our performance studies. This architecture is similar to many of the other architectures and especially to PIPE. It should be noted that this specific architecture is not critical to this research. We would expect to obtain similar results with any decoupled system. But we describe this specific architecture since our results were observed in the course of some other studies on this architecture. The description in section 2 illustrates that this architecture is considerably 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.

Figure 1: A Decoupled Architecture

2 A Decoupled Architecture

In this section, we describe the decoupled architecture, which was used to conduct our simulation studies. An overall view of the architecture is presented in section 2.1 and section 2.2 gives a description of the specific access processor used in the system.

2.1 The Architecture Overview

The 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 [18], 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. 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. 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 architecture with AP instructions stored in the same memory unit as the data.

The two instruction streams run at their own speed, with the queues absorbing excess data. The AP fetches data elements into the Read Queue. The EP reads data from the Read Queue and stores its results into the Write Queue. 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 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.

Figure 2: Block Diagram of the Access Processor

We have not designed a specific EP, but intend to use our AP with any standard uniprocessor CPU with added queue accessing capability. We discuss the design of a 32–bit access processor in section 2.2.

2.2 The Access Processor

Efficient address generation for data structures is a primary goal in obtaining high performance in a computer system. The architecture of the AP and its instruction set is tailored to perform the address generation task efficiently.

The AP is comprised of a Read Queue, a Write Queue, an arithmetic unit, a register set and an instruction cache as in Fig. 2. The instruction set of the AP is classified into arithmetic, data transfer, control and queue manipulation groups. The arithmetic group consists of register add, immediate add, immediate multiply, register multiply, and register subtract instructions intended to aid in address calculations. The data transfer group consists of load and store instructions. There are byte, half–word and word loads and stores. In all the above instructions, the Read

Queue can be specified as a destination and the Write Queue as a source. Any invalid operation on a queue (reading from an empty queue, writing to a full queue) causes that instruction to be temporarily blocked.

The control group consists of jump instructions and branches. The queue manipulation group consists of PUT EOD and Branch-on-no-end-of-data (BNEOD) instructions. The PUT EOD instruction is used to deposit EOD tokens into the queues. The BNEOD instruction is the counterpart of this instruction in the sense that whenever one processor inserts an EOD token, the other processor can check for the token using this instruction. Though only these two instructions are explicitly designated as queue manipulation instructions, it should be mentioned that other instructions can also operate on the queues by specifying the queue address in the register fields.

The *End-of-Data* token bit of the head of the queue is treated as an EOD flag. A data read from the queues will not occur if the EOD flag is set. The queue will be left unaffected in such cases. The BNEOD (Branch if no End-of-Data) instruction checks the EOD flag (or the token bit of the head of the queue) and if it is set, it clears the flag as well as the head of the queue. If the EOD flag is not set, the BNEOD instruction leaves the queue unaffected.

This 32-bit access processor could be used along with a standard RISC processor as the execute processor to form a usable decoupled architecture. The EP should also have queue manipulation instructions as in the AP instruction set. Even for the EP, one register bit pattern should indicate the queues and it should be possible to use the queue as the source or destination operand in any operation. It is possible to design instructions in such a way that both source operands come from the queue, but for sake of simplicity, we do not consider such instructions. It is quite possible to have the AP instruction set as a subset of the EP instruction set.

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 [23]. 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 amount of 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 memory access instructions, etc.

4 Simulation Methodology

To realize the architecture, we use a RISC processor, actually the MIPS R2000 as the execute processor in combination with the AP described in section 2.2. Performance simulators were developed for the MIPS R2000 system and the decoupled system with the MIPS R2000 as the execute processor. 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 UNIX operating system. The results from the MIPS R2000 system form the baseline for comparison. Our simulators are mostly table driven so that the effect of changes in system parameters can be easily studied.

Both the EP and the AP are assumed to have a pipeline control similar to the MIPS R2000. The instruction pipeline is five stages deep, IF (instruction fetch), RD (read and decode), ALU, MEM (memory access) and WB (write back). Each of these steps requires approximately one CPU cycle [11]. 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 that short queues are sufficient to achieve performance close to the maximum available with unbounded queues [23].

4.1 Benchmark Details

The simulation workload consists of some of the Lawrence Livermore Loops (LLLs) and some signal processing algorithms. The LLLs were chosen since they were used in research in the past [23] [8] and since they are important to a wide range of scientific algorithms and applications. In addition to the LLLs, we have synthesized a convolution and correlation algorithm as well. These signal processing algorithms are selected since they contain some addressing patterns other than purely sequential. They have been used in the past to benchmark special address generation units [10]. These algorithms exhibit good locality properties. Since our studies involve cache based systems, these algorithms are very relevant for our studies. We intended to include results from some large benchmarks like SPEC, but were unable to do so in the limited timeframe, due to lack of compiler support.

The LLLs can be classified as pure vector loops, linear recurrent loops and vector reduction loops [25]. We have picked up a few loops from each class so that the performance of the system for various types of code is obtained. We consider 'integerized' versions of the LLLs, since our execute processor is not a floating point processor. It would be possible to consider the EP as a system that has a fixed point unit and a floating point unit, say an R2000 and R2010. Our main purpose here is to study the performance enhancement contributed by the access processor, and hence considering data elements to be integer instead of floating point does not invalidate the results. At this point, it may be noted that integerized LLLs were used in the studies in [8] also.

Among the benchmarks we studied, LLL5 and LLL11 involve memory RAW (Read After Write) hazards. The loops can be optimized and RAW hazards avoided by storing intermediate results to a register and using the register content in the succeeding iteration, rather than again reading the data from memory. For loop 5, we consider only the optimized version. For loop 11, both versions are considered; the unoptimized version is labelled as loop 11a and the optimized version is denoted as loop 11b.

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.

Figure 3: Performance of uniprocessor and decoupled systems

5.1 Comparison to Uniprocessors with Caches

The performance of the decoupled system in comparison with uniprocessors with and without caches is shown in Fig. 3. 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 look unrealistically small, but it should be remembered that the benchmarks are small too.) The cache is assumed to have a single cycle access time. 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.

5.2 Sensitivity of Performance to Memory Access Time

The effect of memory speed on performance is illustrated in Fig. 4 for both decoupled systems and cache based single processors. The total execution time is plotted as a function of the memory speed, for three representative traces, loop 3, loop 12 and the convolution algorithm. Loops 5 and 11b behave similar to loop 3; loops 1 and 11a are similar to loop 12 and correlation algorithm is similar to convolution. In Fig. 4a, the range of variation in execution time is the same for both types of systems. In Fig. 4b, uniprocessors with caches have a slightly lesser sensitivity to change in memory speed. The convolution and correlation benchmarks show the behavior in Fig. 4c. These algorithms have strong locality of reference and uniprocessors with caches are far more insensitive to memory speed than the decoupled architecture.

Fig. 5 illustrates the change in speed up of the decoupled system as the memory access time is varied. Results reported in [23] and [8] indicate that the speedup with a decoupled organization improves as the memory speed decreases. For most of the loops, we do observe the effect they reported, but we also notice that beyond a certain memory speed, the speedup declines. In [8], though it is mentioned that the performance advantage is more significant with a slower memory, 5 out of the 12 Lawrence Livermore loops they analyzed, show a smaller speedup for the slower memory module. Since they considered interleaved memories, this effect was not evident in the other loops in the ranges of memory speed they used.

5.3 Limitations Associated with Caches

Caches hide the latency of the main memory by capturing the temporal and spatial locality of the data references. 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

Figure 4: Execution Time vs Memory Speed

Figure 5: Speed up of decoupled system and cache based uniprocessor

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. Larger block sizes might increase the traffic ratio and actually increase the execution time. In several problems, precise tuning of the cache parameters is required to obtain performance advantage. For example, while running the convolution benchmark, it was noted that a system with a cache of size 1024 bytes is reasonably insensitive to memory speed when the block size is 4 bytes, but when the block size is 64 bytes, the 1024 byte size cache seems inadequate in eliminating latency.

Lack of temporal locality limits the capability of some problems to exploit the cache. In Fig. 3, one can notice that caches result in an increase of execution time in 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. Considering memory traffic issues, the best choice for block size would be 4 bytes. (Let us ignore the directory costs just to give caches the maximum potential advantage.) 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 loop 3, loop 5 and loop 11b in Fig. 3. The execution time for the cache based system is higher than that for the system without the cache. 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. For this reason, problems with only spatial locality sometimes do not benefit from caches. As previously observed in [1] [12] [20] unless carefully designed and implemented, caches result in increasing the total execution time of a program.

5.4 Limitations of Decoupled Architectures

With the presence of slip, the access processor in a decoupled architecture performs address computations and accomplishes data fetch ahead of EP demand. The access processor can thus hide the long

Figure 6: Fetch cycles as a fraction of total execution time

memory access times from being seen by the execute processor. But our simulations point to a limit to the performance enhancement that can be introduced by the AP. In conventional memories, if there are m memory accesses in the entire program and if t_{mm} is the main memory cycle time, a time of $(m * t_{mm})$ is required for the data access (assuming no cache and data size equal or greater than bus width). Irrespective of whether locality properties are exhibited by the data references or not, since the AP has to perform fetches for each individual reference, no reduction in the time $(m * t_{mm})$, is possible. Hence decoupled architectures can eliminate memory latency only as long as the total fetch time required can be accomodated 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 clearly evident in the total execution time. This is clear from Fig. 6. The solid section indicates fetch time as a component of the total execution time in the decoupled mode. Optimizations can result in better performance in the $t_{mm} = 5$ cycles case (and certainly in $t_{mm} < 5$ cycles) but in $t_{mm} = 10$ cycles and $t_{mm} = 15$ cycles, there is no room for further improvement. In cases with slower memory, data accesses constitute almost the entire 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.

Another limitation of decoupled computers relate to the overhead incurred in the process of exploiting the access execute parallelism in the computing process. Slight code expansion results while 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, one of the operands has to be moved from the queue to a register which also contributes to code expansion.

In certain problems, like 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 overheads that may increase the execution time more than the uniprocessor mode. If the compiler recognizes such problems, the DAE system should not be slower than a uniprocessor.

5.5 Significance of a Data Cache in a Decoupled Architecture

We observed that uniprocessors with data caches performed considerably better than decoupled architectures, for slow memories. (See t=15 cycle case in Fig. 3). 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. 7. 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 perform better than the other systems. This can be attributed to the strong temporal locality present in the data references in these benchmarks. The effect of caches is not so pronounced in benchmarks like Lawrence Livermore Loops 1, 11a and 12. Loop 3, loop 5 and loop 11b (loop 11 after optimization) do not benefit from caches. For these benchmarks, the cache has a similar effect in uniprocessors too.

$t{=}15cycles$

Benchmark	Uniproc with cache	Decoupled without cache	Decoupled with cache
LLL1	1.22	1.20	1.53
LLL3	0.95	1.53	1.44
LLL5	0.96	1.18	1.13
LLL11a	1.36	1.08	1.29
LLL11b	0.97	1.10	1.06
LLL12	1.39	1.08	1.58
Mean	1.14	1.20	1.34
Correlation	2.12	1.46	2.77
Convolution	2.04	1.37	2.71
Mean	2.08	1.42	2.74

$t{=}5cycles$

Benchmark	Uniproc with cache	Decoupled without cache	Decoupled with cache
LLL1	1.06	1.22	1.43
LLL3	0.92	2.35	2.16
LLL5	0.92	1.55	1.37
LLL11a	1.18	1.25	1.25
LLL11b	0.93	1.27	1.17
LLL12	1.18	1.25	1.53
Mean	1.03	1.48	1.49
Correlation	1.23	1.51	1.61
Convolution	1.13	1.60	1.61
Mean	1.18	1.55	1.61

Table I. Speedup comparison

The speedup figures for memory access time equal to 5 cycles and 15 cycles is presented in Table I. Speedup is calculated with reference to uniprocessors without caches. The mean values for Lawrence Livermore Loops and convolution and correlation benchmarks are shown separately. The contribution of caches is significant in the t=15 cycles case, but is minor in the t=5 cycles case. Optimized code often cannot exploit caches to the same degree as their unoptimized versions, and that is illustrated by the difference in performance for loops 11a and 11b. All the limitations associated with caches in uniprocessors seem to also apply to decoupled systems with caches. Nevertheless, the improvement resulting from caches in programs with temporal locality cannot be neglected.

6 Conclusion

We have presented simulation results for a decoupled architecture when connected to a noninterleaved memory. 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 it 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.

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 so that the total data access time required can be accomodated within the runtime of the execute processor if it does not have to wait for any operands. 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 non–cached uniprocessor is significant.

We also performed simulations to determine whether decoupled architectures can obtain 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.

7 Acknowledgement

We would like to thank Ali S. Berrached for his help during the development of the simulator.

References

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

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

[3] Coraor L.D., Hulina P.T. and Mannai D.N.,"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] Cohler E.U., Storer J.E., "Functionally parallel architectures for array processors", IEEE Computer, vol.14, pp.28–36, Sept.1981.

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

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

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

[8] Goodman J.R., Hsieh J.T., Liou K., Pleszkun A.R., Schechter P.B., Young H.C., "PIPE: A

VLSI Decoupled Architecture", 12th Annual International Symposium on Computer Architecture, June 17-19, 1985, Boston, Massachusetts, pp.20-27.

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

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

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

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

[13] Pleszkun A.R., "A Structured Memory Access Architecture", Computer Systems Group report CSG-10, Coordinated Science Lab, University of Illinois, Urbana, Oct 1982.

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

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

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

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

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

[19] Smith J.E., 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, pp 199–204, October 1987.

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

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

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

[23] Smith J.E., Weiss S. and Pang N.Y, "A Simulation Study of Decoupled Architecture Computers", IEEE Transactions on Computers, Vol.C-35, No.8, August 1986.

[24] Smith W.M, Abraham S.G. and Davidson E.S, "A performance comparison of the IBM RS/6000 and the Astronautics ZS-1", IEEE Computer, January 1991.

[25] Smith W.M., Abraham S.G., Davidson E.S., "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.

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