The Use of Caching in Decoupled Multiprocessors with Shared Memory

Tim Harris and Nigel Topham
University of Edinburgh

Abstract

In the following we evaluate the costs and benefits of using a cache memory with a decoupled architecture supporting shared memory in both the uniprocessor and multiprocessor cases. Firstly we identify the performance bottleneck of such architectures, which we define as *Loss of Decoupling* costs. We show that in both uniprocessors and multiprocessor machines with high latency such costs can greatly effect performance. We then assess the ability of cache to reduce loss of decoupling costs in both uniprocessors and multiprocessors. Through use of graphical tools we provide an intuition as to the behaviour of such decoupled machines. In multiprocessors we define the target model of shared memory and introduce various coherency schemes to implement the model. Each coherency scheme is then evaluated experimentally. We show that hardware coherence schemes can improve the performance of such architectures, though the relationship between hit rate and performance is substantially different than in the non-decoupled case. Our results are based on discrete-event simulations which take as input address traces from various scientific applications.

1 Introduction

Though Shared Memory has proven an effective programming model for parallel machines, there are still open questions as to the best way to implement the model efficiently so that it can be scaled to a large number of processors. Such a machine will by necessity have memory distributed among the nodes, and we can expect the latency of memory access to be substantially larger than the amount of time required to perform a simple computation. To provide reasonable performance for such a model both latency reducing and latency tolerating techniques may be necessary.

In this paper we focus on the latency tolerating technique of decoupling. Substantial work has previously considered decoupling in various contexts. In [1] a VLSI decoupled architecture was compared to a traditional architecture while altering the speed of memory. In [2] a decoupled machine with interleaved memory was compared with the CRAY-1 architecture. In the related work of [3] decoupled architectures were shown to be insensitive to memory latency when performing optimally. In the recent work of [4] decoupled architectures are compared against traditional uniprocessor systems with caches. Although one of the stated goals of that paper was to consider use of cache in decoupled architecture, very few such results were presented and the authors conclude only that caching has some potential for such machines.

The goals of this paper are to consider indepth the value of caching in decoupled architectures, and we present the first results to consider multiprocessor decoupled architectures which support shared memory. We first begin by identifying the salient features of decoupled performance, namely the Loss of Decoupling events which occur when the latency tolerating ability of the architectures is temporarily curtailed. We use graphical results from our simulator to develop an intuition of decoupled execution and the role such events play. We then quantify the benefits that can be achieved through use of caching in a decoupled uniprocessor for a suite of applications; also explaining when those benefits can be expected to be accrued. We then describe various schemes for maintaining coherency in a multiprocessor decoupled system supporting a weakly coherent model of shared memory, and evaluate the performance of each scheme. Our results show that caching often provides substantial benefits in both the uniprocessor and multiprocessor cases, but that the relationship between hit ratio for a caching scheme and subsequent performance is more involved than in the case of traditional architectures. In the case of multiprocessors, it is only the more expensive coherency schemes, yielding high hit rates, that are able to achieve substantial performance gains.

2 Architectural Assumptions

The basic idea of a decoupled architecture is to divide the instruction stream produced by the compiler into two sub-streams; one that is entirely addressing and memory fetch instructions and the other that is entirely computations. These two streams are then executed in parallel by two separate processors, with the memory fetch operations being being pipelined as much as the memory system architecture permits.

A simple diagram our assumed architecture is shown in figure 1. The address stream is executed by the Address Unit, or AU, while the computation stream is executed by the Data Unit or DU. To execute a Load instruction from its stream the AU will calculate the load address and then put the request to memory into the Load Address Queue (LAQ). This request will then be serviced by main memory, with the resulting operand being placed in the Load Data Queue (LDQ). At the initiation of a program the DU will initially stall until sufficient operands arrive in the Load Data Queue such that the first computation may be initiated. In the meantime, the AU will continue to issue requests to memory and ideally operands will begin to arrive in the LDQ at a steady rate after this initial memory latency. To execute a store the DU places an operand in the Store Data Queue (SDQ), while the AU places the store address in the Store Address Queue, and when both items are in their respective queues the request is forwarded to memory.

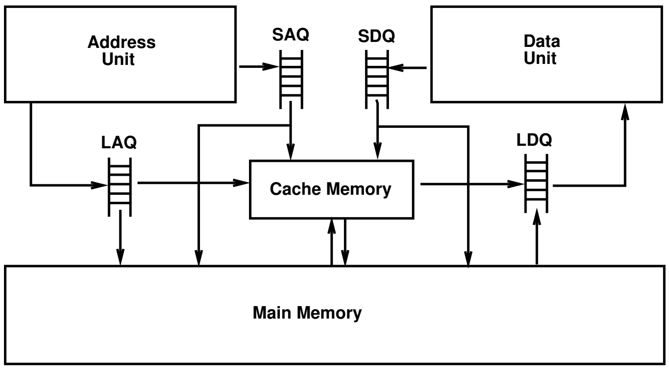


Figure 1: A Decoupled Architecture Model with Cache.

Ideal execution in such an architecture occurs when the DU is able to process data at its maximum rate, then the operands it requires will have been requested early enough by the AU that they will already be in the LDQ when required. In this sense the AU can be seen as a form of prefetch engine for the DU, and during this type of execution the latency of the memory system is fully tolerated [3]. The term *decoupled* refers to the fact that the time an AU fetches an operand is decoupled from the time that operand is used for a computation in the DU.

Most applications also require that the AU and DU periodically synchronize; an event called a Loss of Decoupling or LOD [5]. A loss of decoupling will take place at conditional jumps, for example, when the AU will need a result from the DU to determine the next instruction to be executed. Various coherency operations in multiprocessors will also cause an LOD. After an LOD the DU must again wait for the full latency of the memory system before executing its next computation and becoming decoupled again. For this reason we define any time that the DU is waiting for an operand to arrive in the LDQ as an LOD cost, and it is these costs that are the primary concern of our analysis. In figure 1 we also show the cache memory we assume in later sections of the paper. When a cache is present the AU will always attempt to fetch operands directly from the cache and into the LDQ. We assume a write-through and write-allocate scheme.

3 The Simulation

The experimental results described in this report have been generated by trace-driven discrete-event simulations. The input for the simulation consist of two traces of instructions, one of which contains AU instructions such as address calculations and memory fetches for given addresses, and the other of which contains DU instructions such as floating point operations and memory stores. These traces were generated by annotating programs such that the annotations compute the instruction streams which are then written to the trace file during program execution. The traces we have generated are parallel traces, where each instruction has a processor number to specify which processor is to execute the instruction. These processor IDs are generated by the annotation in a simple fine-grain manner, typically by using the induction variable of a do loop modulo the number of processors in the machine. The grain of work allocated to a processor is typically on the order of a BLAS 1 or BLAS 2 routine.

The applications we have used to generate traces are well-known scientific codes from established benchmarks, all written in Fortran, and all assuming shared memory in our formulation. The most well know is the Linpack benchmark, a linear algebra subroutine designed to factor a dense matrix into its lower and upper triangular components. This is a particularly floating point intensive application, though the size of the loops varies from the full width of the matrix down to very small inner loops as the target matrix becomes smaller.

The other two codes are both parallel versions of codes taken from the Perfect

Club benchmark suite [6]. The TFRD benchmark is a simulation of the behaviour of two electrons. The most computationally intensive routine, OLDA, performs integral transformations of four matrices and a transposition. Therefore there are a fairly large number of memory references per each floating point operation. The OCEAN benchmark is a fluid dynamics application which uses the spectral method, and is hence dominated by Fast Fourier Transformation (FFT) operations. This application also has a significant number of instructions which do nothing but copy data from one data structure to another.

In our simulation we assume an average memory access time of 100 cycles, and a average cache hit time of 5 cycles. We assume a lightly loaded system in the sense that we assume shared memory latency will have little variation, and we neglect contention in the interconnection network in the multiprocessor case.

4 Uniprocessor Performance

To understand the performance of decoupled multiprocessors one must first understand the uniprocessor case. Here we outline what characteristics are typical of such architectures, and in particular we show quantitative evidence that *loss of decoupling events* play a substantial role in performance.

4.1 The Saxe Diagram

The Saxe diagram, introduced in [7], allows one to visualize the behaviour of a decoupled architecture. Below we show the utility of these diagrams, and we augment the diagrams with more information which helps provide a concise explanation of cache performance for such architectures. We now use the diagram to explain normal modes of operation for decoupled architectures.

In figure 2 we see a uniprocessor decoupled architecture with a "fast" Data Unit, i.e. a DU which can consume data as fast as it is produced by the AU. The diagram represents the performance on a small kernel of TFRD code with approximately 450 operands to be fetched and three Loss of Decoupling events. In the diagram the first (solid) line represents the rate of the AU requesting operands from the memory system, while the second (dashed) line shows the DU rate of consumption for these operands.

In this model of a lightly loaded system we expect the memory latency to be a constant 100 cycles. Therefore we can see that the DU must wait 100 cycles before beginning execution, as this is the time until the first operands arrive in the

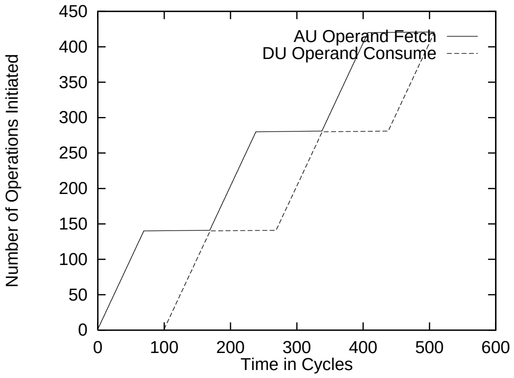


Figure 2: Example Saxe Diagram for Uniprocessor.

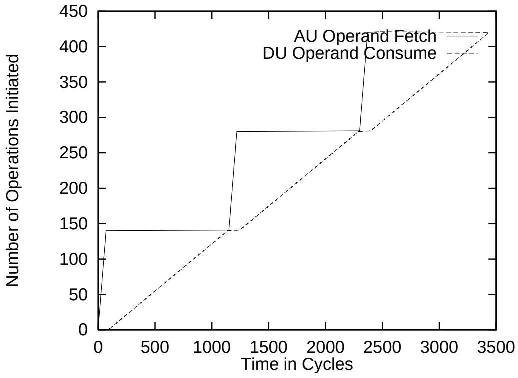


Figure 3: Same example with slower DU.

LDQ. After about 130 operands have been fetched by the AU we see the first LOD event. This corresponds to a conditional jump which is based on a result from the DU. The AU must wait until the result is produced by the DU, about 100 cycles later, and then it can determine the destination of the jump and begin processing again. However, the DU will need to wait for the full latency of the memory system again, as the LDQ will remain empty after an LOD until a request can make the complete circuit from the AU through the memory system. Therefore the LOD cost in this case is about 100 cycles. The time when the two lines meet is naturally the time during which the two processors are synchronized.

If the DU can not consume data as fast as the AU can fetch it, than the DU will naturally "decouple", in the sense that the data it uses will have been fetched by the AU long before it was needed. We refer to this as a "slow" DU, as seen in figure 3. The DU progress may also be slowed down by the nature of the application, ie. if each operand is reused many times than the consumption rate of operands by the DU will be slower. However, in current architectures memory systems are typically the bottleneck, rather than floating point performance. Given that fact we focus our study in the rest of the paper on the case of a fast DU, in the sense that we expect the DU to be typically waiting on operands to arrive in the LDQ rather than consuming operands at a rate slower than they arrive. The case where the DU consumes operands slower than they arrive may be considered the ideal case in the sense that such a machine will usually perform well, independent of other considerations.

4.2 LOD Costs

We now show how the influence of LOD costs varies from application to application. In fact, such costs can be used to characterize an applications suitability for decoupling. The suitability of applications for decoupling has previously been quantitatively assessed in [5], where various compiler techniques were outlined for reducing the frequency of LOD events. Here we have fixed the frequency of such events through extraction of the instruction traces as described earlier, and we consider the influence of increasing memory latency on LOD costs for an application. In figure 4 below we show the execution time and LOD costs of the TFRD application as we increase average memory latency from 20 to 200 cycles. At a high latency of 200 cycles per memory access more than half the execution time is attributable to LOD costs. On the other hand, the OCS example of figure 5 has very few LOD events and hence even with high latency memory the percentage of execution time attributable to LOD costs is small. We can see this as an indication that OCS is an inherently well-suited application for decoupling. TFRD, on the other hand, is an application which has substantial LOD costs and hence without caching is poorly suited for these architectures. Applications such as OCS will run at close to the throughput of the DU, and their performance will not be substantially altered by modification of the memory system. The Linpack application, which falls in between these two in terms of its frequency of LODs, is shown in figure 6.

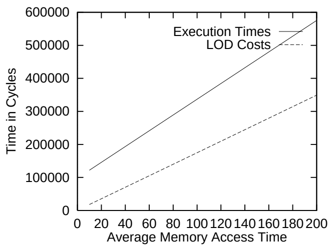


Figure 4: TFRD application.

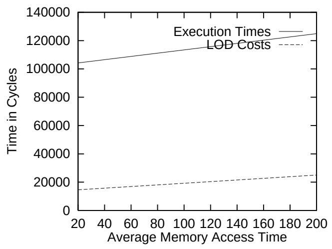


Figure 5: OCEAN application.

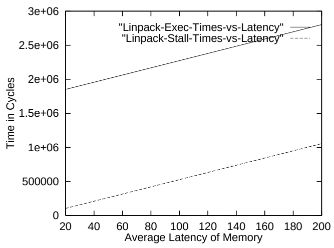


Figure 6: Linpack application.

5 Uniprocessor Caching

The goal of caching in decoupled architectures is to reduce the latency of memory accesses which are contributing to LOD costs. However, the relationship between hit rate and performance of a decoupled computer is profoundly different than that of a traditional architecture, as we now show.

Saxe diagrams typically have two lines, one for the AU progress and one for the DU progress. However, in the case of caching architectures we have found it useful to augment this with an additional line which represents the arrival time of an operand in the Load Data Queue. We can expect this line to be close to the AU line if a fetch results in a cache hit, and far from the AU line in the case of a cache miss. In the case of a slow DU the two processors will decouple naturally as the DU spends time computing, but in the time of a fast DU decoupling will only occur due to LOD costs. In this case the LOD costs experienced by the DU on a particular loop structure (without embedded LODs) will be equal to the maximum latency of any one memory fetch. To have LOD costs for a loop equal to the cache hit time rather than the cache miss time requires that the hit ratio for that loop is 100 percent. Therefore the overall LOD costs for an application will not be a simple linear function of hit rate as in traditional architectures, but instead be a threshold function whereby LOD costs are only reduced if some loop structures execute with all memory accesses being cache hits.

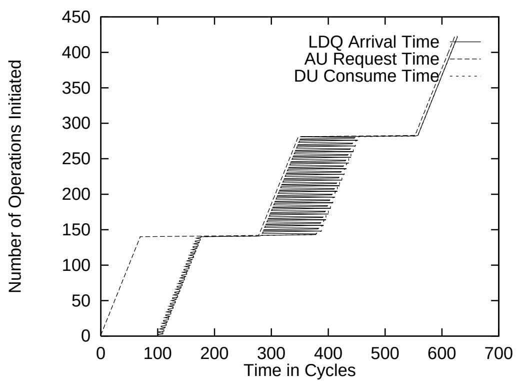


Figure 7: Saxe diagram for uniprocessor with cache.

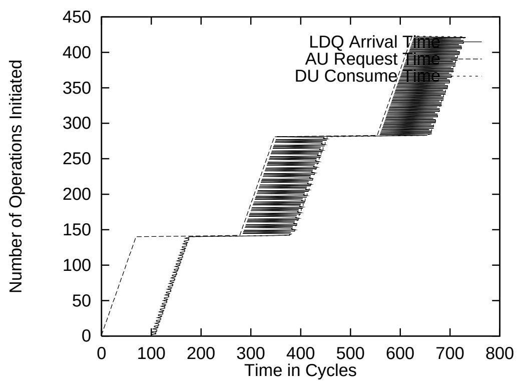


Figure 8: Same as previous figure but with lower hit ratio.

This explanation is clarified by figures 7 and 8. Figure 7 represents decoupled performance on a similar small kernel trace, but with a large cache resulting in a high hit rate. We see that in the first loop the cache is empty and no cache hits are observed. The decoupling between the AU and DU is the full latency of main memory, or 100 cycles. In the second loop there is a higher hit ratio, but still there are periodic misses which force the DU to wait a full latency, so the LOD costs remain as with no cache. Only in the third loop do we observe a 100 percent hit rate and hence benefits from using a cache. The degree of decoupling between the AU and DU in this final loop can be seen to be very small, corresponding to the 5 cycle cache hit time assumed. Figure 8 shows the same kernel and caching scheme, but with a smaller hit rate. We see that, though the hit rate of the third loop appears to be higher than either of the two previous loops, i.e. there are more hits shown on the LDQ-arrival-time curve, the LOD costs are still the same as if there was no caching as the DU will need to wait the full latency for whatever misses do occur.

Of course, a full application will have a large number of such looping structures and LOD events. The performance of an application will benefit anytime during execution a loop runs entirely in cache, and in the case of a slower DU frequently also if a loop begins with a substantial number of cache hits before encountering cache misses (allowing the slow DU to naturally decouple). The saxe diagram of a larger piece of an application will again show the characteristic stair step behaviour of loop structures and LODs, and the slope of this line will directly reflect the performance of the machine. In figures 9 and 10 we show such a diagram for trace of about 8000 operations and show how successful caching techniques result in steeper progress gradients.

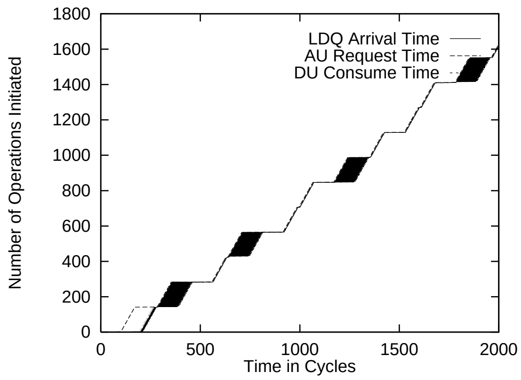


Figure 9: Caching characteristics with slightly larger trace.

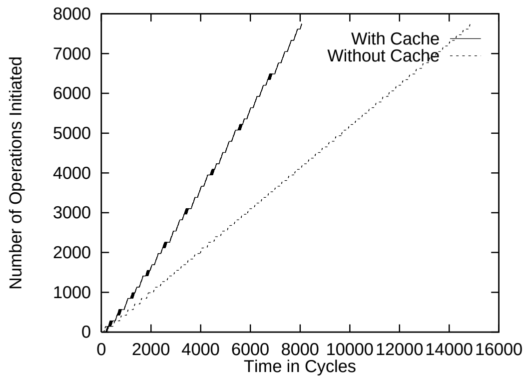


Figure 10: Slope comparison for cache and no cache.

In figure 9 we see that approximately two out of three loop structures end up running totally within cache and how performance improves as a result. Figure 10 shows clearly how reduced LOD costs result in steeper slope curves in the Saxe diagram and hence better execution time. In particular we observe that with cache this excerpt completes in roughly 8000 cycles, while without cache it requires over 14000 cycles.

6 Uniprocessor Results

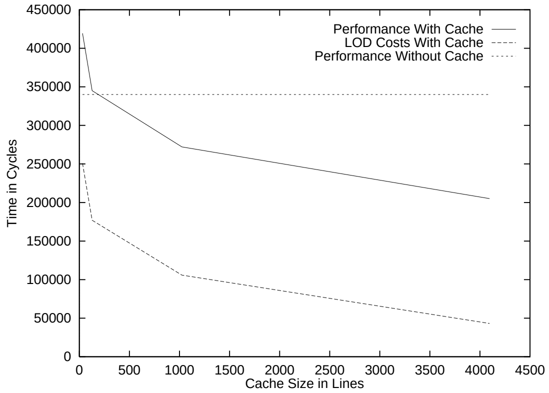


Figure 11: The Effect of Caching on a Uniprocessor Executing the TFRD Application.

We now briefly present the result of using caching as described in the three applications we are considering. Of course, the best benefits we can hope for are to reduce the LOD costs of an application to near zero, so applications with little LOD delays in the non-cache case will clearly not perform substantially better than the non-caching case. In figure 11 we see the effects of using caching on an application with large LOD costs as we vary the number of 4-word lines in the cache, and again assuming 100 cycle average memory access time. The constant line is the execution time of the TFRD application with no caching, and the bottom dotted line is the LOD costs of the application while using caching. We see that with very small cache sizes the extra overhead of loading cache lines and overwriting them frequently actually increases execution time. However, with cache sizes of 500 lines are larger we see substantial benefits from cache use. We see that with a 4K-line cache the LOD cost is reduced to less than 50,000 cycles, a substantial reduction from the near 200,000 costs shown in figure 4 above.

Our other two applications are naturally less bound by LOD costs, but still see performance improvements through the use of cache, as show in figures 12 and 13. However, in each case we see that the majority of the LOD costs which was show to exist previously without cache is eliminated.

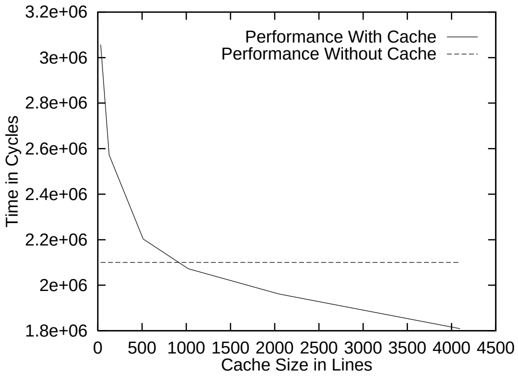


Figure 12: Linpack with Cache.

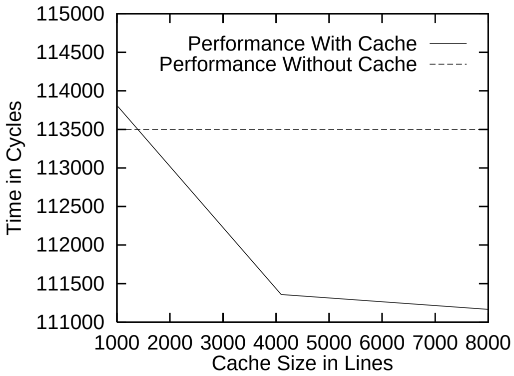


Figure 13: OCEAN with Cache.

7 Multiprocessor Caching

In caching for multiprocessors the primary concern becomes how to maintain coherency given the multiple copies of each line which can potentially exist in various caches. We now outline the three coherency schemes we evaluate, as well as describing the specific model of shared memory we assume. For a comparison of various coherency schemes in the context of non-decoupled multiprocessors see [8].

7.1 Memory Model

We assume a weakly coherent memory model, based on that supported by the DEC Alpha [9]. The Alpha Model allows for instruction reordering and memory buffering, which provides substantial potential for memory system performance improvements. To enforce strong ordering the scheme provides the Memory Barrier or MB. An MB insures that all instructions preceding the MB will be strongly ordered with regard to all those following the MB. It therefore becomes an important characteristic of our coherency schemes that we provide implementations for the Memory Barrier instruction, as well as ensuring that our implementations of cache coherency fulfil the assumptions of the model. Its important to note that in a decoupled architecture an MB also is a form of Loss of Decoupling, as it requires synchronization between the AU and DU. For a comparison of other weak consistency models see [10].

7.2 Hardware Coherency

In a typical hardware coherency scheme there will be state bits associated with each line in cache and also with each line in main memory. The bits are modified dynamically to ensure coherency during updates of cache lines. We assume a write-invalidate scheme, where if a processor chooses to write to a cache line it must first broadcast an invalidation request to all other processors holding a copy of that line, as well as changing the status of the line in main memory to note exclusive access. Within our weak model there is no need to wait for invalidation requests to be acknowledged before modifying a cache line. We assume a directory based and hence scalable scheme, and use a write-through policy, and allocate new cache lines on writes as well as reads.

The MB instruction is implemented in the following way. When a processor executes an MB it will broadcast that MB to all processors and stall until it has received acknowledgment from each processor. The processors will only acknowledge the outstanding MB once all the memory requests which have been routed to their memory have been serviced. In routing through our network we also need to ensure that such MB instructions will not pass any previously issued invalidation requests, thereby ensuring that all such requests will have arrived at their destination by the time processors attempt to acknowledge an MB. After the issuing processor has received acknowledgment from all processors it may proceed with the next instruction.

7.3 Software Coherency

Instead of altering state bits at run-time, a software scheme depends on static compile-time analysis to enforce coherency. Firstly it is necessary to identify computational units, or *epochs*, within an application, which typically consist of nested do-loops. A decision is then made for each epoch what data is cachable and what data can be read and written to only in main memory.

In our software scheme we attempt to allow all data to be cached, but then invalidate data at the end of each epoch when necessary. We consider two variants, the first of which is a simple scheme where all cached data is invalidated at the end of an epoch. Secondly we consider a scheme were compile time analysis lets us determine whether data is shared or private for an epoch, and we only invalidate shared data. We again use a write-through write-allocate scheme. The MB implementation outlined above is also used. However, an important attribute of the assumed memory model in the context of software schemes is that an epoch is naturally defined by Memory Barrier operations, so we also use the advent of an MB to initiate the cache invalidations required in the scheme. Though false sharing is also a hazard in software coherency, we assume small page sizes and therefore don't focus on this problem in our study.

8 Multiprocessor Results

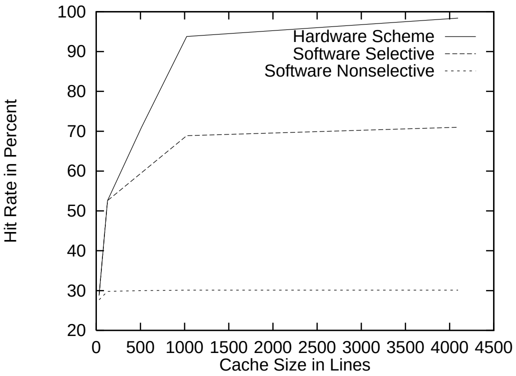


Figure 14: TFRD Cache Hit Rates.

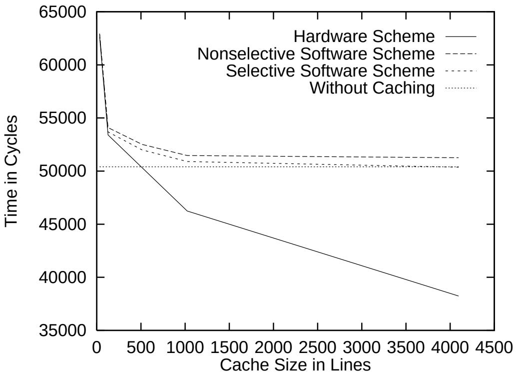
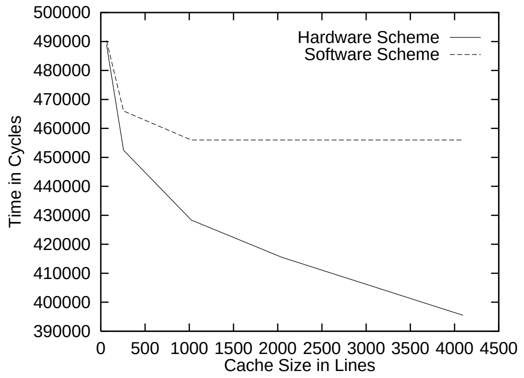


Figure 15: TFRD on a Multiprocessor.

In our multiprocessor simulations we assume an 8 processor system, where each AU-DU pair has its own cache, and where all processors share a common main memory subsystem. Some of the final results of the study are shown in figures 14 and 15. In figure 14 we show the cache hit rates for the three schemes we have described: hardware, software with global invalidation and software with selective invalidation. Global invalidation clearly is too simplistic a scheme, as the reuse of cached items can only occur within an epoch; no cross-epoch sharing can take place. The selective invalidation scheme does much better, resulting in hit rates of over 70 percent for a cache size of 4K lines. However, it is only the hardware scheme that allows hit rates to reach 90 percent and above for these cache sizes.

Perhaps more importantly, we observe in figure 15 that hit rates have a non-linear relationship with execution time, unlike traditional architectures. As shown earlier with Saxe diagrams for uniprocessor decoupled machines, the only way to reduce LOD costs is to have high enough hit rates such that at least some loop structures will observe localized 100 percent hit rates. This will only be possible in applications with significant data reuse, and even then we observe that only the best performing (and hence most expensive) coherency schemes will succeed.

Despite the large difference in hit rate between the two software schemes, neither of them can improve performance beyond that of the no-cache case.



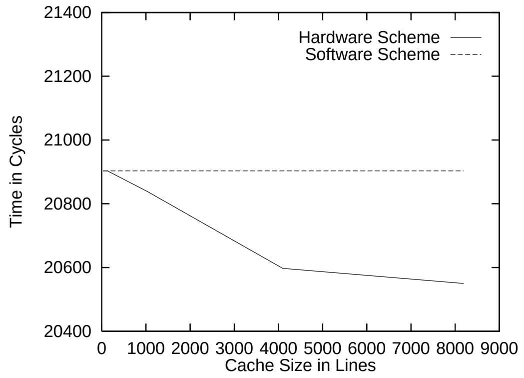


Figure 16: Linpack on Multiprocessor. Figure 17: OCEAN on Multiprocessor.

The other two applications show similar behaviour. We now show only the selective-invalidation case of software coherency, as the two software schemes always result in near identical curves for the reasons we have explained. In both OCEAN and Linpack the hardware scheme results in some speedups, while the software schemes only manage to approach the performance of the non-caching case.

9 Conclusions

We have shown how decoupled performance depends on the frequency of Loss of Decoupling events, and have described the relationship between the latency tolerating nature of the architecture and this frequency. Through the use of Saxe Diagrams we have shown that in decoupled architectures with fast Data Units cache only succeeds in improving performance if a very high hit rate is achieved. Applications that result in cache misses being unevenly distrubuted can be expected to gain more from the use of cache, as they will have a higher likelihood of epochs having no cache misses than will applications where the misses are evenly spread throughout the code. Multiprocessor caching will naturally result in lower hit rates due to coherency operations, and hence gains will be more limited than in the uniprocessor case. However, the hardware coherency scheme we have describe still achieves at least small gains for all three applications we considered, whereas the software schemes do not.

10 Acknowledgements

We would like to thank Alasdair Rawsthorne, Peter Bird, and Bob Fredieu for our many discussions on caches in decoupled architectures. Many of the motivating principles of this work evolved from those discussions.

This work has been funded by the European Community ESPRIT project SHIPS, contract number P6253.

References

[1] J. Goodman, J. Hsieh, K. Liou, A. Plezkun, P. Schectuer, and H. Young. PIPE: A VLSI Decoupled Architecture". *Proc. 12th International Symp. on Computer Architecture*, 1985.

[2] J. Smith, S. Weiss, and N. Pang. A Simulation Study of Decoupled Architecture Computers. *IEEE Trans. on Computers*, Vol. C-35, No. 8, August 1986.

[3] J. Smith, et. al. The ZS-1 Central Processor. *Proc. 2nd Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, October, 1987.

[4] L. Kurian, P. Hulina, and L. Coraor. Memory Latency Effects in Decoupled Architectures with a Single Data Memory Module. *Proc. 19th Int. Symp. on Computer Architecture*, May, 1992.

[5] P. Bird, A. Rawsthorne, and N. Topham. The Effectiveness of Decoupling. *Proc. 7th Int. Conf. on Supercomputing*, July, 1993.

[6] G. Cybenko, L. Kipp, L. Pointer, D. Kuck, Supercomputer Performance Evaluation and the Perfect Benchmarks", *International Conference on Supercomputing*, 1990.

[7] A. Rawsthorne and N. Topham. Saxe Diagrams: A Notation for Visualizing Performance in Decoupled Architectures. *In Preparation.*

[8] S. Adve, V. Adve, M. Hill, and M. Vernon. Comparison of Hardware and Software Cache Coherency Schemes. *Computer Sciences Technical Report No. 1012*, University of Wisconsin-Madison, March 1991.

[9] Alpha Architecture Handbook. Digital Equipment Corporation, 1992.

[10] K. Gharachorloo, A. Gupta, J. Hennessy. Performance of memory consistency models for shared-memory multiprocessors. *Proc. 4th Conf. on Architectural Support for Programming Languages and Operating Systems*, 1991.