The Scalability of Decoupled Multiprocessors

Tim J. Harris

Department of Computer Science
University of Edinburgh
Edinburgh, Scotland

Nigel P. Topham

Department of Computer Science
University of Edinburgh
Edinburgh, Scotland

Abstract

In the following we consider the ability of the technique of decoupling to improve the scalability of multiprocessors which have physically distributed memory but which support a shared memory model of computation. We consider the performance of a variety of similar such architectures; those with and without caching and those with and without decoupling. As a metric of scalability we focus on the speedup of these architectures while executing a suite of parallel scientific applications. We show that decoupling can play a substantial role in improving the scalability of such an architecture. Furthermore the additional technique of caching with hardware coherence can further improve this scalability in the face of high latency memory access.

1 Introduction

Decoupling is a technique by which the addressing and memory fetch operations of a process are executed concurrently with the computations inherent in that process. A decoupled processor will typically have two sub-processors, one which is designed to do nothing but make requests to memory and the other which simply computes results from the data which has arrived in its queue, as we will describe in detail.

The shared memory model of parallel computing has received particular attention in recent years, due to its similarity with uniprocessor programming and its ease of use. However, there are still many difficult questions as to the best way to support this model, given that any truly scalable architecture will by necessity have its memory physically distributed across all the nodes of the system. To provide good performance on such a system both latency reducing techniques and latency tolerating techniques must be considered.

Latency tolerating techniques are those that allow the processor to continue doing useful work while waiting for a memory access to complete, instead of simply remaining idle for the full latency of memory after every access. Typically such techniques exploit on-chip parallelism, as in the case of multithreading or prefetching. Latency reducing techniques are those that reduce the delay associated with a memory access, but without mitigating the effect that delay will have on run time.

In this paper we evaluate the latency tolerating technique of decoupling in terms of its suitability for massively parallel architectures, especially those where the latency of a remote memory access grows as the size of the machine increases. We also consider the latency reducing technique of caching, and the way these two techniques interact on multiprocessors. Our goal is to evaluate the speedup achieved by a variety of such related architectures on a set of parallel scientific applications.

We now proceed to outline previous work in this area and how our contribution fits into this background. In section two we describe in more detail the architecture model we consider. Section three explains the techniques used in our simulation, and the applications from which we have generated our parallel address traces. Section four provides the bulk of our experimental results, and in section 5 we conclude.

1.1 Previous Work

Much work exists to address the subject of decoupled uniprocessors. In [4] a VLSI decoupled architecture was compared to a traditional architecture while the speed of memory is altered. In [8] a docoupled machine with interleaved memory was compared with the CRAY-1 architecture. In the related work of [9] decoupled architectures were shown to be insensitive to memory latency when performing optimally. Uniprocessors with caches are considered in comparison to decoupled machines in [6]. And in [3] various applications are considered in terms of their inherent suitability for decoupling.

Our main contribution is to consider the effects of decoupling as the number of processors in a machine is increased and the latency of requests to remote memory becomes subsequently larger. We also provide insight into the benefits of using decoupling in conjunction with caching, and the effects this combination has upon performance scaling.

2 The Architecture

The technique of decoupling consists of dividing a normal instruction stream into two sub-streams; one of which is entirely addressing and memory fetch operations, and the other of which is entirely arithmetic. The two streams are then executed in parallel by separate units, the Address Unit (AU) and Data Unit (DU), as shown in figure 1. The AU performs the necessary addressing operations and then places memory requests into the Load-Address-Queue (LAQ), hopefully in a highly pipelined manner. After the memory requests are serviced by the memory system, the resulting memory operands are placed in the Load-Data-Queue(LDQ). The two units store data by interacting via the Store-Data-Queue (SDQ) and the Store-Address-Queue (SAQ). Decoupled execution occurs when the DU is able to process data at its maximum speed with no idle time, due to the fact that the operands it needs will have been previously requested by the AU and will be already waiting in the LDQ. When this is the case the AU serves as a prefetch engine for the DU, and the latency of the memory system has no effect on the execution rate of the machine, ie. latency is "tolerated". The architecture shown in figure 1 also has a cache memory, as we assume in later parts of the paper.

With most programs the AU and DU will periodically need to synchronize, and event called a *Loss of Decoupling* or LOD. A Loss of Decoupling will take place at conditional jumps, where the AU will need a result to be computed by the DU before it can continue fetching operands. Various coherency operations in multiprocessors will also cause LODs. After an LOD the DU must wait the full latency of the memory system before its first operands arrive in the LDQ. The main technique of our analysis is to evaluate the cumulative LOD costs of applications under various conditions and a variety of architectures.

In the case of a multiprocessor decoupled machine, we assume each processor has an AU and DU, as well

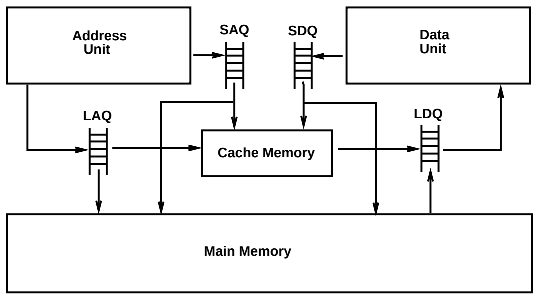

Figure 1: A Decoupled Architecture Model with Cache.

as a block of local memory. Shared memory is distributed amongst these memory modules such that the cost of a memory access increases in proportion to the distance a request must travel through the network. We assume a two-dimensional mesh interconnection network, so memory access time increases in proportion to the square root of the number of processors plus some initial cost which corresponds to accessing the memory module once the request has arrived at the appropriate node. The general idea is to assume a high bandwidth but reasonably high latency memory system, as this may be seen as typical of distributed memory supercomputers of the future and hence provides an interesting benchmark for analysis.

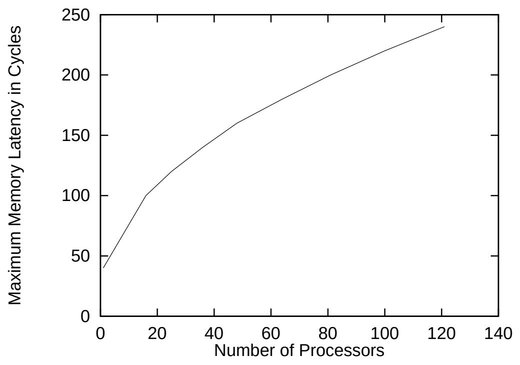

Figure 2: Memory Latency as a function of number of processors.

We assume a lightly loaded system, in the sense that memory latency is not a function of network contention in the system. We assume that each memory module has an access time of 40 cycles and that it requires an additional 20 cycles for a message to travel across one link of our interconnection network. The interconnection network we assume is a simple two-dimensional mesh, so the diameter of our network is always proportional to the square-root of the number of processors, and we assume that every memory request will pay the full latency of the network (unless it is a cache hit). With this architecture the latency of a memory access will increase substantially as the size of our machine grows, so in this sense we are considering an architecture that will naturally have difficulty scaling to a high processor count (see figure 2). This worst case model allows us to focus on the benefits of our latency reducing and hiding techniques in the presence of large memory access times.

2.1 Multiprocessor Caching

In much of the following we consider the performance of both decoupled and non-decoupled architectures with the inclusion of caching. In the decoupled case we assume each processor has its own small cache which is readable or writable by either the DU or AU, as shown in figure 3. The primary difference between the caches and local memory modules is that each data item held in local memory will exist only once, whereas a value in a cache will be a copy of a value which exists somewhere in a local memory of the system, and other such copies of the same value may exist in multiple caches.

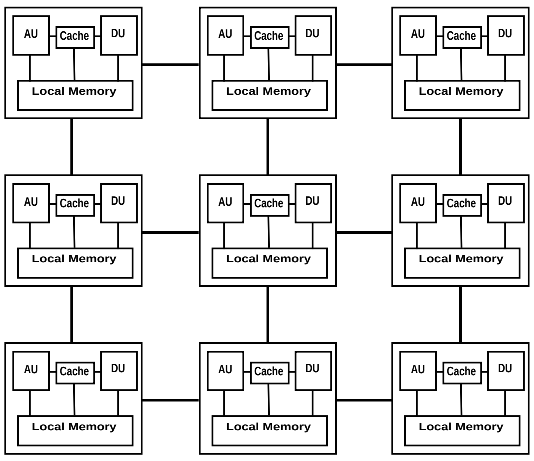

Figure 3: A Decoupled Mesh with Cache.

The primary difficulty in using cache in a multiprocessor is coherency; ensuring that processors see a consistent view of memory despite the various copies potentially distributed throughout the system. We assume a hardware coherency scheme with one centralized directory of cache information. Our scheme is write-though with invalidate and allocate-on-write, so to write a cached value a processor will first need to achieve exclusive access to a cache line by sending invalidation messages to all other processors that currently hold a copy. Upon each subsequent write the updated values will also be sent to the appropriate location in the shared distributed memory to maintain coherency between the cached values and main memory. We assume a 5 cycle cache hit time, and our cache is direct mapped. We assume a weak coherency scheme in the sense that coherency operations are not guaranteed to finish until a barrier operation is encountered. Execution of a barrier operations requires waiting until all pending invalidations have taken place and the operation has been globally acknowledged. The memory model we implement corresponds loosely to that supported by the DEC Alpha Chip set [2]. More details on this caching scheme and memory model may be seen in [5].

3 The Simulation

We simulate these architectures with an event-driven simulator written in C. The input for the simulator is a set of address traces, generated by executing a suite of Fortran applications, each of which was annotated such that every memory access is followed by writing a line to the trace file which specifies the address of the access which has taken place. Additionally, each memory operation is allocated to a particular processor, so processor IDs are another component of the trace files. Unlike our previous work [5], in generating these traces we have used a fine grain approach to maximize the parallelism of the applications, often using the index of the inner most do-loop as the processor ID indicator. In addition to memory fetch operations, we also include memory barrier operations in the annotations to the applications, as required to maintain consistency within the assumptions of our weakly coherent model [2].

In our multiprocessor results we focus on machines with between one and thirty-two processors. Due to the large computational costs of detailed simulations of highly parallel processors it becomes impractical to model much larger machines, but we expect our results will naturally extrapolate to higher numbers of processors. Our limits on computational power have also reduced our ability to generate large numbers of data points for each curve, as should be clear to the reader. However, despite these limitations we believe we have identified trends in execution time and speedup that will be maintained for a large variety of machine sizes.

3.1 The Application Suite

We have considered the performance of our various architectures on three scientific programs written in Fortran. One is the well known Linpack benchmark; a linear algebra subroutine which factors a dense matrix into its upper and lower triangular components. This is a particularly floating point intensive application, though as the factoring progresses the loop sizes become small enough that some efficiency is lost.

The other two programs are parallel versions of benchmark codes taken from the Perfect Club suite [7]. 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. The trace files generated by these applications typically contain hundreds of thousands of events.

4 Results

We begin by considering the execution times of an application as a function of the number of processors for both the decoupling case and the non-decoupling case. Given that the our non-decoupled model has few optimizations and is quite sensitive to latency, it is little surprise that the decoupled architecture provides substantial performance gains, as shown in figure 4 for the case of the Linpack benchmark. Such a result is also a logical extension of the work shown in [4, 8, 6]. The on-chip parallelism of each node allows the latency of memory access to be hidden, as the AU and DU are able to stay busy once the initial startup latency has past.

A more important measure for the purposes of determining scalability is the speedup as a function of the number of processors, which we show in figure 5. We define the speedup with n processors for a given architecture as the execution time for 1 processor on that architecture divided by that on n processors. If

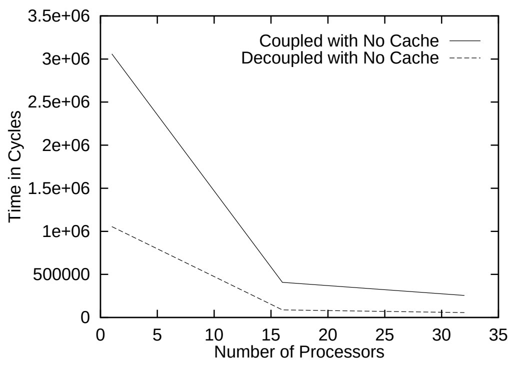

Figure 4: The execution time of Linpack.

we defined speedup in terms of the best case serial algorithm as often suggested, then it would fail to compensate for the substantial differences in uniprocessor performance seen in our various architecture models. Our goal is instead to about scalability independent of the range of uniprocessor execution rates. With our definition the speedup curves provide a measure of the scalability of a model in the presence of increasing latency of memory access, and it serves to balance out some of the absolute differences in execution times for our models. The speedup curve of figure 5 shows that our basic decoupled processor has good potential as a latency hiding and hence scalable architecture. However, a more interesting comparison is with decoupling and other techniques for achieving fast memory access.

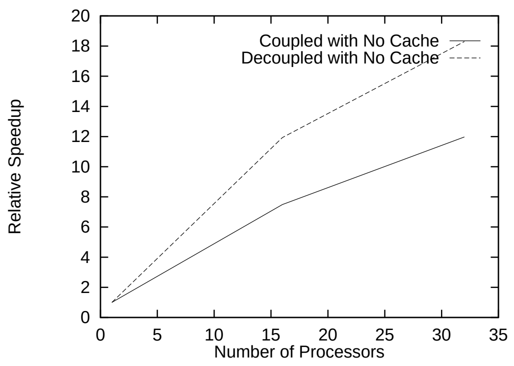

Figure 5: The speedup of Linpack.

We therefore now consider the performance of our non-decoupled model with cache to a decoupled architecture without cache for a variety of sizes. Intuitively one might guess that our model of decoupling will provide better execution times than simple caching, for a variety of reasons. First, to read a cache line will cost additional time when compared to a read in a non-caching architecture, and given the coherency operations necessary for a parallel cache, such as possibly frequent invalidations, it is possible the newly read cache line will never be accessed again. Secondly, when a decoupled machine is behaving optimally, the latency of memory access is entirely hidden, whereas the best a cache architecture can hope to achieve is a reduction in that latency. These ideas are born out in figure 6, where we compare the speedup achieved from the use of caching in a non-decoupled machine with a decoupled architecture which does not use cache. We see that the benefits of caching are substantial in the non-decoupled architecture, but that the decoupled machine still provides more scalability than either the non-cached or the cached traditional architecture.

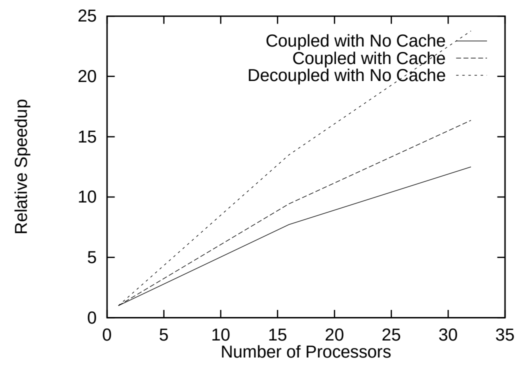

Figure 6: Speedup of OCEAN.

4.1 Caching in Decoupled Architectures

The benefits of caching in conjunction with decoupling are varied for a multiprocessor. For the TFRD application, shown in figure 7, we see substantial improvements in speedup due to the inclusion of caching in our decoupled model. However, if we consider instead Linpack or OCEAN, there are few improvements. We now outline the reasons for these differences.

Firstly, its important to realize that caching can only provide additional improvements in latency reduction when the job of latency hiding attempted through decoupling has not been entirely successful. This is typically the case in applications where many synchronisation events are inherent in the algorithm,

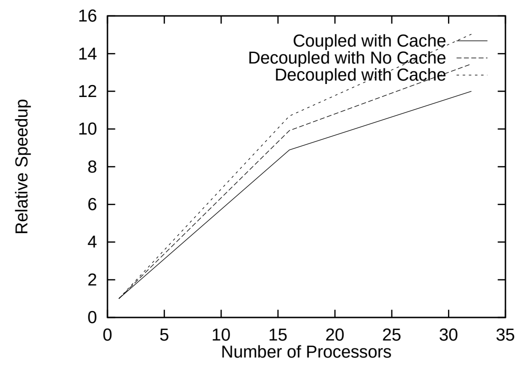

Figure 7: Speedup of TFRD.

such as one with many conditional branches or frequent barrier operations for ensuring cache coherency. We refer to such synchronisation costs as loss of decoupling (LOD) delays, and we have displayed the accumulated delays of the three applications when using decoupling without caching in figure 8. It shows that the OCEAN benchmark has substantially fewer LOD delays as the processor count increases, and hence that we can expect little gains from the use of caching to reduce these costs.

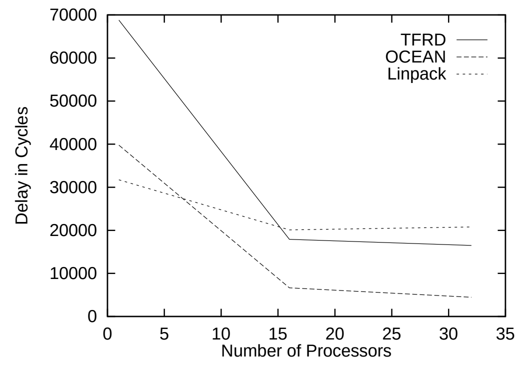

Figure 8: Accumulated LOD delays.

Secondly, an application does have substantial LOD costs such as TFRD and Linpack, the hit rate of the cache will still need to be high to achieve any performance gains from use of a cache. In figure 9 we compare the hit rates of the two applications and see that, as the processor count increases, the hit rate of Linpack rapidly falls off. TFRD, on the other hand, maintains a good hit rate and hence can reduce the LOD costs we have shown above and finally achieve benefits from the additional use of cache with decoupling. Keeping in mind these two factors, it is clear that TFRD is the only application of the three that achieves the two necessary conditions for achieving the speedup benefits.

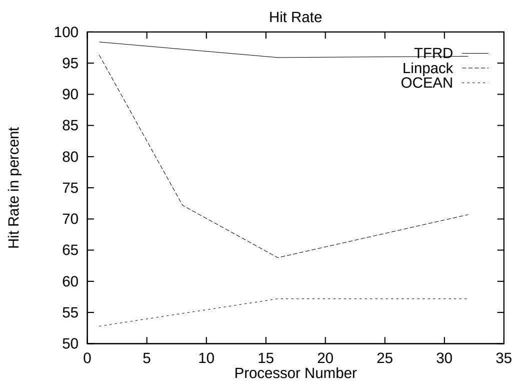

Figure 9: Hit rates for decoupled model.

5 Summary and Conclusions

In most cases multiprocessors with distributed memory will have memory access latencies that increase rapidly as the number of processors in the machine increase. We have shown that decoupling serves a useful purpose in such machines in hiding that latency and hence allowing good speedups on such machines though the general technique of on-chip parallelism. We also have shown that the additional technique of caching for latency reduction can further enhance the benefits of decoupling, and believe that the as a general result multiprocessors of the future will often benefit from the such combinations of reduction and tolerating techniques.

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] 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.

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

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

[4] 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.

[5] T. Harris and N. Topham. Performance of Weak Consistency Schemes on the DEC Alpha. *Proceedings of International Conference on Parallel Computing '93*, Grenoble France, North-Holland Publishing, September 1993.

[6] 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.

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

[8] 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.

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

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