The Performance Mapping of a Class of Fully Decoupled Architecture

Alan W.R. Crawford

M.Phil.

The University of Edinburgh

1998

Declaration

I, the undersigned, hereby declare that the contents of this thesis have been composed solely by myself, and that the work described herein is entirely my own.

Date: 27th May 1998.

Acknowledgements

I would like to acknowledge the vital contribution of my supervisor, Dr. Nigel Topham, without whose continuing guidance, advice, comments, suggestions, criticism and encouragement I would never have finished this thesis. I would also like to thank my second supervisor, Prof. Roland Ibbett, and Muriel Mewison, of the SHIPS project, who provided the LOD frequency information that greatly helped to reduce the amount of work required in Chapter 7. Many thanks also to Dr. Rob Pooley and Dr. Tony Fields, the examiners, for their feedback, suggestions and constructive criticism. Particular thanks to Rob for his tolerance and all his help during this past year.

Credit is also due to Trident Systems, my former employer, for allowing me the use of their computing facilities during the period 1995-1997.

Finally, many thanks to my ex-wife Jenni for all she put up with during the period when I was working on my research and thesis.

Abstract

Fully decoupled architectures present a possible solution to the problems posed by the growing disparity between memory and processor speeds. In this thesis the performance of this class of fully decoupled architectures is examined in detail. The causes and effects of both access/execute and control decoupling are investigated using simulation and modelling, and techniques for their more efficient exploitation are proposed. A similar investigation is made of losses of decoupling (LODs), which eliminate many of the advantages of decoupling, and particular attention is given to a number of methods for the removal of LODs or the reduction of their effects. Finally, a number of architectural techniques to enhance the performance of this class of architectures is presented.

Table Of Contents

List Of Figures

1. Overview

Fully decoupled processor architectures present one possible solution to the problems posed by the growing disparity between memory and processor speeds. Full decoupling is a fusion of two separate architectural techniques - access/execute decoupling and control decoupling.

The first of these separates program execution into two distinct sub-tasks - the access of data and the execution of arithmetic operations upon the data - and allows these sub-tasks to run asynchronously on dedicated processing units. This can in principle allow the access sub-task to run ahead of the execution sub-task, fetching data from memory sufficiently far in advance of the point in time at which it will be needed that memory latency can be hidden.

Control decoupling applies a similar asynchrony to the sub-tasks of control and calculation. In a control decoupled system, a dedicated processing unit traverses the program's control flow graph, attempting to determine the path that program execution will take sufficiently far in advance of the calculation sub-task that instructions can be fetched far enough in advance to hide their memory latency.

When both techniques are combined they can, in theory at least, produce a processor architecture with a high tolerance for long memory latencies, where the pre-fetching permitted by decoupling hides the latency of memory accesses for both instructions and data.

In this thesis I illustrate the capabilities of this class of architectures, examining which architectural parameters affect performance most strongly, and highlighting potential weaknesses in the architecture.

In Chapter 2 the problems associated with high memory latencies are examined, and a number of existing architectural techniques that aim to either hide or reduce memory latency are discussed. Chapter 3 presents the ideas behind both access-execute and control decoupling in more detail. Chapter 4 examines a number of possible techniques for modelling the behaviour of decoupled architectures, including analytical modelling, simulation, and a fusion of both techniques. The model derived in this chapter is put to use in Chapter 5, in which a series of experiments are performed to determine the architectural parameters which most strongly influence the degree to which a decoupled processor can exploit both access/execute and control decoupling. A number of techniques to enhance decoupling, both software- and hardware-based, are presented.

Chapter 6 addresses the problem of loss of decoupling, a phenomenon whereby the two decoupled sub-tasks must resynchronize and thus temporarily lose their ability to mask memory latency. A further series of detailed experiments is performed to determine the causes and effects of loss of decoupling, as well as to examine the potential of a number of software- and hardware-based techniques to remove or reduce the impact of losses of decoupling.

The decoupling experiments in Chapters 5 and 6 were based upon the examination of a number of program kernels, short segments of code that illustrate some fundamental behaviour characteristic of certain classes of program. In Chapter 7 these results are validated in the context of real programs, by conducting a series of experiments based upon full-sized programs. The decoupling behaviour of these is analyzed with respect to the fundamental behaviours examined in the previous two chapters.

Finally, in Chapter 8, conclusions are drawn as to the potential of fully-decoupled architectures as well as their drawbacks, and several possible avenues for future research are described.

2. Introduction

2.1 The Problem

As the 1990s have progressed the disparity between processor and main memory performance has increased inexorably. While the speed of main memory has risen for a number of reasons, this has been more than offset by the increased demands made of it by modern high-performance processors.

The improvements in memory performance are due to a number of factors. A fortuitous side effect of the drive for ever larger capacity memory ICs is a decrease in feature size which has in turn reduced memory cycle times. Page-mode dynamic RAM chips which allow sequential data to be accessed more efficiently have become commonplace, and DRAMs have also appeared which incorporate small internal caches of fast static memory.

At the same time, however, processor clock speeds have risen considerably. High-end microprocessors are at the time of writing typically clocked at somewhere between 300 and 400 MHz, and projections for the end of the 1990s show a steady climb towards clock speeds in the 500-700 MHz range. These high clock speeds place a heavy load on memory in terms of the instruction and data memory bandwidth required. Add to this the emergence of superscalar instruction issue, with several instructions being initiated per processor cycle and potentially issuing multiple requests to memory for data, and the gap between the low memory latencies that processors require and what a realistic memory system can provide becomes wider still.

A number of solutions have been proposed to deal with this problem. Most of these fall into two distinct categories: memory latency reduction and memory latency tolerance.

2.2 Latency Reduction

Up until now the main thrust of research into improving computer performance has been closing the processor-memory gap, largely through the use of memory hierarchies such as caches. These work on the principle that, due to spatial and temporal locality in memory reference patterns, the vast majority of memory requests will be met by a fast, local cache. A penalty will be paid when there is a cache miss, but these are sufficiently infrequent in most cases to have little effect on performance. The concept of caches and their behaviour are well understood and have been covered in detail in the literature, so the topic will not be covered in any depth here. Detailed coverage of the issues involved in cache design can be found in Chapter 8 of Hennessy and Patterson's *Computer Architecture: A Quantitative Approach* [HP90] and also in Hennessy, Horowitz and Przybylski's paper on cache design tradeoffs [Pr88] amongst others [Smth82].

The usefulness of caches is limited, however, since Amdahl's Law places a strict bound on the performance gains that can be obtained. Some proportion of memory accesses will always miss the cache and, as the gap between processor and memory speeds widens, the effect of these cache misses on overall performance increases. Even if the proportion of accesses that cause cache misses were to remain constant (which is by no means certain when dealing with shared memory multiprocessors and their requirements for cache coherency), processor performance will become increasingly limited by the disparity between processor and memory speeds due to the effects of these cache misses.

2.3 Latency Tolerance

Most current techniques to close the gap in speed are, alas, only partially successful in treating the symptoms of this disparity. Also, these techniques attempt only to reduce the symptoms, not to cure the problem itself. In fact, it appears that no complete cure is yet possible, unless an unexpected technological breakthrough leads to a drastic reduction in memory latencies. There is another approach to the problem though - the development of processor architectures which are "immune" to long memory latencies. Heavily-interleaved memory systems are, once up to speed, quite capable of providing sufficient memory bandwidth to keep a modern processor supplied. The problem lies in creating a situation in which this high potential bandwidth can be reliably exploited, and the high latency (which no amount of interleaving can remove) of memory accesses has little or no effect on processor performance.

One way of achieving this is to dispense with the idea that a memory access is atomic. In most conventional modern processors, a load is a single indivisible operation, from the placing of the address on the address bus to the arrival of the requested data on the data bus. If the data is not immediately available, the processor must stall until it arrives. Thus the latency of a load operation is directly determined by the latency of memory and it is through this method that the effects of slow memory are transmitted directly into the most immediate measure of processor performance, program execution time.

Similarly, when a store operation is initiated the processor waits for it to complete before proceeding. This is due to the need to prevent any subsequent store instruction from changing the contents of the address and/or data bus before the current store has completed.

The use of a cache mitigates these effects somewhat, since with a reasonably efficient (write-back rather than write-through) cache, most loads and stores complete within a cycle or two. However, on those loads that miss the cache and those stores that force a write-back (or all stores in a system with a write-through cache) the full latency of these operations becomes apparent and the processor is stalled, wasting many processor cycles that could conceivably have been used to perform other operations. By overlapping the load or store latency with other subsequent operations and thus hiding or masking the latency, its effect on the program's execution time could be reduced considerably.

Historically, a number of approaches have been tried that attempt to do this. They vary considerably, but all rely on the use of some form of non-blocking or *decoupled* load or store.

2.4 Non-Blocking Stores

The idea of non-blocking stores is far from new, and is frequently encountered in the form of write buffers. In a system with a write buffer, the store address and data are placed in a temporary buffer until the store has been committed to memory. The processor can continue unhindered while the store completes. It is only necessary for the processor to stall if it tries to execute a load from the same memory location as the pending store (which would cause a read-after-write hazard) or if a second store is attempted before the first has completed.

However, even these two situations can be exploited in certain circumstances. A load attempting to access a location to which a store is still pending can take its data directly from the buffer, eliminating the need to stall. A second store to the same memory location can overwrite the value in the buffer, since no reads of the old value have been performed in the intervening time. By substituting a queue for the simple buffer and thus allowing multiple stores to be pending at any one time, stalls on stores can be eliminated in the vast majority of cases, although the need to compare any subsequent loads with the addresses of all pending stores requires additional hardware and also makes it difficult for a load to take its data directly from the buffer should a pending store to the same location be present.

The benefits of write buffers are particularly apparent in systems with write-through caches. In these systems the processor can potentially be stalled for a considerable amount of time, since all stores go directly to main memory as well as the cache. However, even in systems with a write-back cache the presence of a write buffer can help to mask the high latency of a cache miss. As will be seen later, the techniques used to mask store latencies in fully decoupled architectures are a natural extension of the write buffer concept.

2.4.1 Non-Blocking Loads

The techniques that have been used to mask load latencies are more varied than those used to mask store latencies although, presumably due to the prevalence of caches, the area has been less widely explored. Historically, a number of approaches have been tried and these are briefly surveyed.

2.4.2 The CDC 6600 Address Unit

Perhaps the simplest form of latency masking load appeared as long ago as the late 1960s, in the Control Data Corporation's CDC 6600 supercomputer [Ibb82]. This architecture implicitly split each load into two decoupled operations by virtue of its unconventional approach to memory accesses. The CDC 6600 possessed three distinct sets of registers. The A (address) registers were used, as their name suggests, for specifying addresses for memory accesses, although the method by which this was accomplished was unusual. The X (operand) registers were used to store the operands for arithmetic operations, while the B (index) registers were used as offsets to memory accesses.

Loads were performed by storing an address (with an optional offset specified by a B register) into one of a certain subset of the A registers. This initiated the load operation, and the requested value appeared in the corresponding X register some number of cycles later (that is, write to A1 and the value at the address specified by A1 will eventually appear in X1). In the meantime, other operations could be performed, hiding the latency of the load. The register scoreboarding mechanism used in the CDC 6600 ensured that any instruction attempting to read the contents of a register in the process of being loaded from memory would stall until the access had completed.

A similar mechanism was used for stores, whereby storing an address to a certain subset of the A registers would cause the contents of the corresponding X register to be written to memory. Again, the scoreboard prevented either of the registers involved being overwritten before the operation had completed while allowing other operations that did not use those registers to proceed unhindered.

2.4.3 Cache Preloading

In more recent times similar techniques [Sk92] [Smth82] [Chen95] has been employed by a number of architectures to preload data into the data cache. While this is not quite the same as decoupling a load into two distinct parts (the generation of the address and the fetching of the data), it does give similar benefits.

Preload instructions are statically inserted in the instruction stream some time before the data they access will actually be needed. Hopefully this will lead to the requested data being present in the data cache by the time it is needed.

This approach falls midway between reducing the *actual* latency through the use of a cache and reducing the *effective* latency by hiding the true latency of a possible cache fill underneath other operations. As such it combines elements of both latency reduction and latency tolerance and attempts to use some techniques commonly associated with the latter to improve the performance of the former.

An architecture that employs cache preloading may still occasionally block. There is no guarantee that the requested data will be fetched in time or, indeed, that it will be fetched at all. However, if the preload operation can be started sufficiently far in advance of the point at which the data is needed, the increased latency that might result from a cache miss and subsequent cache line fill will be hidden underneath any instructions executed between the issuing of the preload operation and the execution of the operation that will use the preloaded data.

This approach has both advantages and disadvantages. The primary advantage is, of course, the decrease in the number of visible cache misses and the subsequent reduction of the average effective latency. The technique also has the rather less obvious benefit of having no effect on the semantics of the load operation. There is no requirement that a memory access be preceded by a preload operation, which is little more than a hint to the memory system that the specified data is likely to be accessed in the near future and should be fetched into the data cache if possible. Preloads may be safely ignored without any change in program semantics. The only effect of ignoring a preload is a change in the effective memory latency.

This makes it practical to build a family of processors with the same instruction set architecture [A92] where, depending on how much money the customer wishes to spend on their processor, the preload operation may result in nothing happening at all, in a preload to cache being attempted, or in the specified data being moved up a single level in a multi-level memory hierarchy.

Those architectures which currently implement cache hints of this sort (such as Digital's Alpha architecture and the PA-RISC architecture from Hewlett Packard) tend to use them to provide an approximation of vector registers using the data cache. The preload operation (FETCH in the Alpha) optionally loads an aligned block around a specified address into cache which can then be accessed by the processor at cache speeds.

However, if the data cache has limited associativity or limited size problems can arise when preloading large data structures, such as long vectors or even fairly modest matrices. If the preload operation is executed sufficiently far in advance of the time at which the data will be required, a later preload operation might cause data loaded by the first preload to be overwritten before it has been used. When the instructions that use the first preloaded data are reached not only will the full memory latency be visible, but the reloading of that data due to the cache miss may also result in the data that had been loaded by the second preload being overwritten before it is needed, necessitating that *that* data also be reloaded.

On the face of it, this may seem to be no worse than the well known situation in which two regularly accessed memory addresses cause a cache to "thrash" due to insufficient associativity. However, on closer inspection it can be seen that preloading means that each cache line would be loaded not once but twice per access, wasting considerable amounts of memory bandwidth and severely reducing performance. For this reason, the effectiveness of preloading is limited by the need to ensure that cache contention of this sort will not occur frequently, since in some cases the use of cache preloading can actually *reduce* performance.

The cache thrashing problem reveals another limitation of preloading. The need to ensure that cache contention or thrashing does not occur make it necessary that programmers or compilers take upon themselves the responsibility for explicitly managing the cache. This requires an awareness of the physical structure of the cache, such as the line size, number of lines, and associativity, and even then is likely to produce sub-optimal results due to the effects of non-preloaded cache accesses and those memory access characteristics that cannot be determined at compile time.

In addition, carefully tweaked code that goes to great pains to prevent cache contention may, if run on a machine with a larger or smaller cache (or even one with a differing degree of associativity), either fail to take advantage of the greater cache size or perform poorly. It could even be suggested that an architecture employing preloading would be better to dispense entirely with the notion of a separate cache and instead use an explicitly mapped and accessed area of fast scratchpad RAM, with data being loaded into the scratchpad by a decoupled load of the form described in Section 2.4.2.

A further limitation of cache preloading centres around the required distance between the preload instruction itself and the first use of preloaded data. It is frequently impossible to move the preload far enough from the first use for the latency to be completely masked. For example, the Alpha architecture manual recommends that there should be no fewer than 64 unrelated instructions between a FETCH instruction and the first operation that uses the prefetched data. While it is true that this high figure is at least partly due to the Alpha FETCH operation preloading data in blocks of 512 bytes, it still illustrates the general principle that the required distance between preload and use may be large.

The problems this can cause are particularly noticeable in the case of tight loops. A tight program loop may result in it being impossible to separate a preload from the first use of the preloaded data by more than a handful of instructions, greatly reducing the usefulness of the technique. Techniques such as loop unrolling can reduce the effects of this problem but cannot always eliminate them. There will always be some cases in which there just are not enough instructions available between initiating the preload and the data being required to keep the processor busy.

2.4.4 Dataflow

Finding plenty of instructions to execute is, however, the forte of dataflow architectures [Arv91]. These machines are, by their very nature, tolerant of long-memory latencies. Rather than instructions fetching their operands, the availability of operands results in the execution of instructions that act upon those operands. An instruction *cannot* wait for a value to return from memory in a dataflow architecture, since the instruction will not even be activated until its operands become available.

On the surface it would seem that dataflow, with its ability to exploit parallelism to the full and tolerate slow memory is the solution to all the problems of long memory latencies. However, this is not the case in practice. For one thing, the apparently non-existent memory access latency for instructions is due more to cleverly restating the problem than actually eliminating the latency. Also the complexity of dataflow architectures, both from an implementation and programming viewpoint, means that to date dataflow has failed to move beyond the realm of research and into the commercial sphere.

2.4.5 Context Flow And Multi-Threading

Another architectural technique which has yet to be fully commercially exploited, but which at the time of writing seems to be a topic of considerable research interest in both industry and academia, is variously known as context flow or multi-threading [Dub94].

Multi-threaded architectures aim to obtain high throughput with the processor consistently running at near peak performance. They do this through the use of multiple independent processor contexts (each corresponding to one hardware thread of execution) which can be rapidly switched should the currently-running thread run into a possible delay. By the use of multiple threads, the processor can be kept busy until the cause of the delay (the latency of a memory access, or an inter-instruction dependency) has passed. Analogies can be drawn between the decoupled load mechanism of the CDC 6600 (where the gap between load initiation and completion is filled with instructions from the same process) and loads on a multi-threaded architecture where the gap between load initiation and completion is filled with instructions from a number of other, independent, threads.

Exactly what constitutes a "possible delay" can vary from architecture to architecture. At the pessimistic end of the spectrum, the assumption is that every single instruction can potentially cause a delay and that an implicit context switch should occur between each instruction regardless of whether any delay actually exists. More optimistically, there are architectures where context switches happen only on memory accesses or branches and the delays due to inter-instruction dependencies are tolerated. Both approaches have advantages and disadvantages [Lau94].

2.4.5.1 Pessimistic Multi-Threading

The pessimistic approach can potentially achieve very high throughput if sufficient threads of execution are available. By ensuring that no two instructions in the processor pipeline are from the same context, delays due to inter-instruction dependencies are eliminated. Conditional branch penalties also vanish, since each instruction is completed before the next starts, which makes the architecture very suitable for superpipelining [Jou89]. Even memory latencies are hidden, as threads that are waiting for values to return from memory are suspended until such time as their data becomes available. From the point of view of a single thread, all memory accesses have zero latency.

As long as sufficient threads are available to keep the pipeline full, the processor can be kept busy continuously. This can be seen graphically in Figure 1, which shows the theoretical maximum throughput of a context-flow architecture for various numbers of available threads and hardware contexts. Note that when the number of threads exceeds the number of available contexts, throughput levels out onto a high plateau. The model used to produce this graph relies on the assumption that no thread will experience a delay longer than *c* cycles, where *c* is the number of hardware contexts available, but the general principle is clearly illustrated.

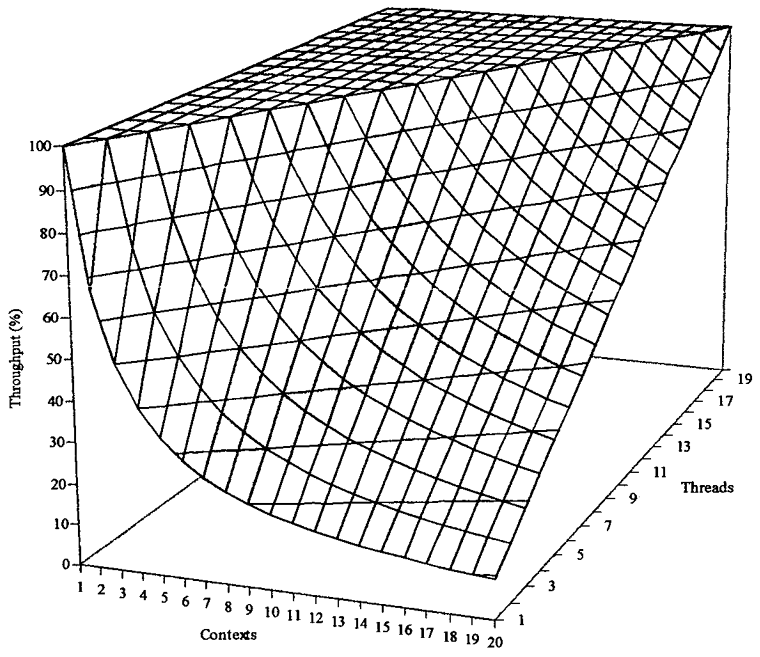


Figure 1 - Total throughput vs. contexts vs. threads

The pessimistic approach to multi-threading has two major shortcomings. The first is in the amount of silicon real-estate taken up by the large number of contexts. Since a context switch occurs between each and every instruction it is imperative that the switch be performed as quickly as possible [Gru96]. This rules out context switching techniques in which the thread's register set is completely or even partly written out to memory and reloaded when the context is next switched in. Generally the only practical way of implementing multiple contexts with no time penalty is through the use of a large partitioned register set. The implicit context switching would be implemented by associating a unique numeric tag with each hardware context. This tag would follow each instruction from that context down the pipeline and also follow each memory reference to memory. It would be prepended to any register identifiers specified in the instruction to specify the appropriate partition of the register file to access. When sent to memory it would be used to reassociate a returning value from memory with the appropriate context.

If there were 64 hardware contexts, each of which had access to 32 64-bit general purpose registers, the register file would contain no less than 2048 64-bit registers. While this not inconsiderable size is well within the reach of mid-90s technology it is open to question as to whether this is a good use to put such a large area of silicon, especially since the register file utilisation will remain constant regardless of how large it becomes.

The second shortcoming is not a technical one but a result of the architecture's semantics. By ensuring that no single process has more than one instruction in the processor pipeline at any time, we ensure that at best each thread can only use a small fraction of the processing power available. On a processor with n pipeline stages and t hardware contexts, each thread can at best use only $\frac{1}{n}$ th of the processor's total throughput, in the case where exactly n threads monopolise the processor and no delay is greater than n cycles. In the worst case, where there are more software threads than hardware contexts, each context can use only $\frac{1}{n}$ th of the total throughput available (assuming that $t > n$) with the available throughput per software thread dropping in inverse proportion to the increase in the number of software threads competing for the available contexts.

So, while the throughput of the architecture as a whole may be high, no individual thread can attain more than a fraction of this performance. Thus, unless the application that is being run is inherently highly parallel, peak performance will be difficult to achieve, since a reduction in the number of runnable software threads will not produce a corresponding increase in per-thread performance.

Although a typical workstation may be running many processes at a time (user shells and applications, a window manager, assorted daemons) these heavyweight processes would be unable to exploit multi-threading, other than internally, since their enlarged contexts would contain additional information such as cache TLB entries, which would have to be switched in the conventional manner. To obtain high performance from a multi-tasking system would also require that each of the processes was itself multi-threaded.

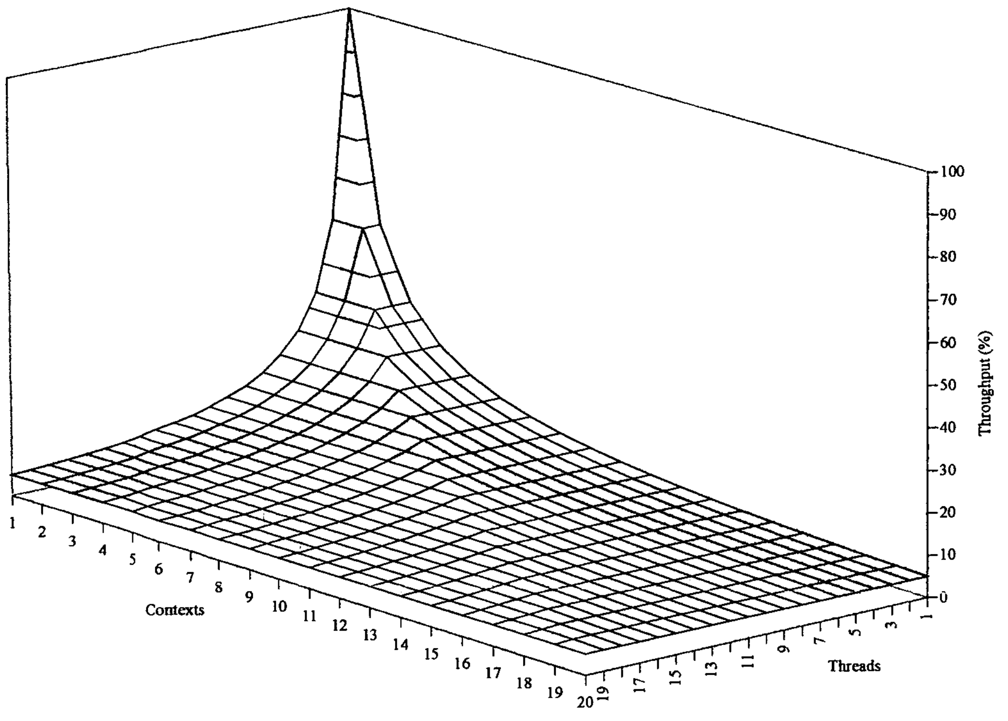


Figure 2 - Per thread throughput vs. contexts vs. threads.

If the number of active threads in a multi-threaded processor drops below the length of the processor pipeline, bubbles will form due to the one instruction per thread restriction. The effect of this on per-thread performance is illustrated in Figure 2. This shows the proportion of total throughput available to a single thread as the number of hardware contexts and runnable software threads are varied. No account has been taken here of the additional performance penalty that would be incurred by having to save and restore hardware contexts in the conventional sense when there are more threads than contexts, and it is also assumed that the processor has a CPI of 1, and is thus capable of 100% throughout. For this reason, Figure 2 actually gives an optimistic estimate of per-thread throughput.

2.4.5.2 Optimistic Multi-Threading

An optimistic approach to multi-threading overcomes many of the shortcomings of pessimistic multi-threading, at the expense of additional complexity. Exactly what constitutes "optimistic" can vary, but the general guideline is that a context switch should be performed only when necessary. It is in the precise definition of "necessary" and the measure of what constitutes an unacceptable delay in execution that variations arise.

In the most extreme form of optimistic multi-threading, contexts are switched whenever a real delay, of whatever kind, is encountered. This has the advantage over pessimistic multi-threading that contexts are switched only when real, rather than potential, delays are encountered. The restriction on the number of active instructions from a single context is relaxed, allowing several instructions from the same context to be in the processor pipeline at any one time, as long as none but the last will cause any pipeline stalls.

This means that, as long as delay-causing instructions do not occur too frequently, it should be possible to keep the pipeline full with a smaller number of active threads, and for each of those threads to potentially use a greater proportion of the total processor throughput. This in turn means that fewer hardware contexts may be required. In an extreme case, a single thread with no instructions that cause delays can utilise 100% of the processor throughput if all the other threads are stalled for some reason.

While the optimistic approach to multi-threading is undoubtedly more flexible than the pessimistic one, there are costs associated with this added flexibility. The pipeline requires some form of scoreboard to identify inter-instruction dependencies and switch contexts as necessary, as well as a mechanism for switching contexts on a cache miss and restarting the delaying instructions when the hazard or delay has passed. It may also be necessary to provide some form of pre-emptive switch between contexts (in addition to any higher level pre-emptive multi-tasking), otherwise a number of threads with long runs of non-delaying instructions could monopolise the processor, starving the other threads.

There are, of course, a number of variations in between the extremes of "switch context on every potential delay", "switch context on every real delay" and "do not switch context at all". Since delays can be caused by a number of events (an instruction dependency, a branch in control flow, a long latency memory access) and these delays all have differing costs, it may be desirable to choose from the three possibilities listed above.

An architecture may ignore the delays due to inter-instruction dependencies and conditional branches and allow pipeline bubbles to form, since the delays due to these are usually short, while switching context whenever a memory access instruction is encountered. Alternatively, it may ignore the delays due to inter-instruction dependencies, always switch contexts when a conditional branch is encountered, but only switch contexts if a memory access results in a cache miss. There are many possible combinations.

A detailed examination of multi-threading is, however, outside the context of this thesis. There is, however, much in the literature on this topic, ranging from discussion of multi-threading in multi-processors [Kur91] to more rigourous analytical examinations of the performance aspects of multi-threading [Dub94].

It should be stressed that multi-threading is being examined here only as an alternative latency tolerance technique. It is by no means considered inferior to the latency tolerance technique with which this thesis is primarily concerned, namely decoupling. The not-inconsiderable volume of research underway in the multi-threading field in the mid-90s is ample evidence of the technique's potential and, as we will see later, there is no reason that multi-threading cannot be combined with decoupling.

2.4.6 Decoupling

The concept of decoupling is far from new. As described in Section 2.3.3., the CDC 6600 architecture used a form of decoupled memory access to provide some tolerance of high memory latencies. Several contemporary architectures have also introduced some form of decoupling between the integer and floating point (FP) units of the processor, to "smooth out" differing rates of integer and FP computation and prevent integer operations being delayed by unrelated high latency floating point instructions.

The architectures with which this thesis is concerned use two forms of decoupling - control decoupling and access/execute decoupling. The combination of these two is known as full decoupling. These are described in detail in Chapter 3. Simply stated, the first of these separates the task of traversing a program's control flow graph from the actual computation that occurs at each node of that graph. One processor follows the flow graph, dispatching high-level descriptions of the code to be executed at that node in the graph to "work processors" which execute the code itself.

Access/execute decoupling [Ben91], on the other hand, separates the work allocated by the control processor into the task of generating the addresses for memory accesses and that of performing the actual computation. Both of these provide a natural method of extracting low-level parallelism and, as will be shown, are possible sources of latency tolerance for instruction and data memory accesses respectively [Kur94].

While fully decoupled architectures are a novelty (only one, the ACRI-1, has begun development to date [Top95] [Bird93], machines using access/execute decoupling are not a new development.

The origins of access/execute decoupled architectures can be traced to the early 80s with the PIPE architecture [Goo85] [You88] [Far91] and its successor MISC [Tys92], and the work of James E. Smith at the University Of Wisconsin, who wrote several seminal papers on the subject [Smit82] [Smit84]. This research eventually gave rise to the Astronautics ZS-1, an access/execute decoupled mini-supercomputer. Although this is amply described in the literature ([Smit87] [Smit89] [Mang91]) this architecture, being the only decoupled architecture to reach full commercial production to date, is worthy of closer examination and will be discussed in some detail in the next chapter. More recent access/execute architectures include the Rockwell/DARPA funded research into the HiDISC architecture at the University of Southern California [Cra96] and W.A. Wolf's WM architecture [Wol92].

3. Decoupled Architectures

3.1 Access/Execute Decoupled Architectures

Access/Execute decoupled architectures, as their name suggests, exploit the relatively independent nature of the memory access and data manipulation (or execution) tasks that make up a program. By separating, or decoupling, these tasks and running them concurrently, considerable performance gains can be obtained.

The first of these, the access task, is responsible for generating the memory address of data that is to be used by the execute task and either initiating the loading of that data from memory or providing the store address for a store operation. The generation of memory addresses may be straightforward, requiring only that the access task send the address of a scalar variable to memory, or it may be complex and require additional computation to convert the indices of a multi-dimensional array into a linear offset which, when added to the base address of the matrix, gives the actual address of the memory location to be accessed.

The second of these tasks, the data task, takes data arriving from memory, performs operations on it, then stores the result to memory. Since, for the most part, these two tasks are relatively independent and require very little in the way of inter-task communication, they provide an excellent source of low-level parallelism. In an access/execute architecture, each task runs on a separate specialised sub-processor, a processing unit that has much of the functionality of a conventional processor yet is unable to function without its companion sub-processor. These sub-processors may be further optimised for their role, with the mixture of available functional units and the design of their internal data path and instruction set reflecting their role. They are also likely to include certain specialised architectural structures which will be described shortly.

The splitting of a program into access and execute tasks is a relatively straightforward process, since the two tasks are naturally partitioned. This is not, however, immediately obvious from conventional architectures since these typically arbitrarily mix together instructions from the two tasks. This lack of separation of the two tasks can even inhibit performance, since it complicates the scheduling of instructions on today's superscalar architectures [Far93]. However, superscalar instruction issue and access/execute decoupling are by no means mutually exclusive. There is no reason why the two sub-processors of an A/E decoupled machine cannot themselves be superscalar, and it is possible that the separation of the two tasks might even allow a greater degree of low-level parallelism to be extracted from them.

The main benefits of access/execute decoupling are that it allows the easy exploitation of this natural low-level parallelism, and that since the two tasks are now running (almost) independently and asynchronously, the access sub-processor can run ahead of the data sub-processor and fetch operands from memory before they are actually needed. This is particularly advantageous when the system's memory latency is high, since it allows the address sub-processor to fetch data sufficiently far in advance that the required data is delivered to the data sub-processor before it actually needs it. The data sub-processor is thus able to use it immediately, with no noticeable delay due to memory latency. The memory latency is thus hidden and, as long as the address sub-processor can run sufficiently far ahead of its companion, the architecture exhibits latency tolerance.

This characteristic is valuable in high performance uniprocessors, where the processor clock cycle may be considerably smaller than the memory cycle time. For example, a mid-90s high performance processor may have a processor cycle time of 3ns or lower yet still use 60ns DRAM for its main memory. Latency tolerance is also of use in multiprocessor systems, where accesses to memory that is not local to a processing node may have a very high latency due to it having to traverse an interconnection network.

At the time of writing, access/execute architectures represent the state of the art with regard to the commercial use of decoupling. Although fully decoupled architectures, which will be described in the next section, have been designed, only access/execute decoupling has reached the marketplace so far, albeit with little commercial success.

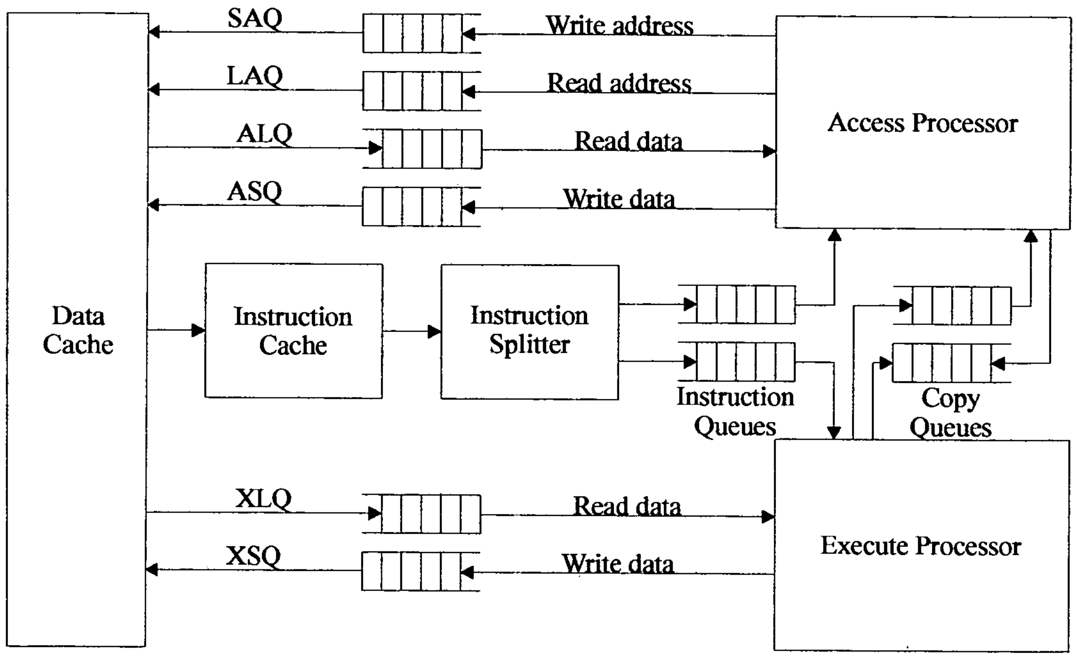


Figure 3 - The Astronautics ZS-1 CPU.

Two distinct families of access/execute decoupled architectures have emerged. These differ primarily in the degree to which they separate the address and data sub-processors.

3.1.1 Single instruction stream architectures

The first of the two families of access/execute decoupled architectures considers the two sub-processors to be little more than functional units of a conventional processor, or even just a particular partitioning of the functional units of a conventional processor. The resulting processor appears relatively conventional externally, with a single instruction data path. However, it splits the instruction stream internally and routes each instruction to either the address or the execute unit.

The Astronautics ZS-1 (discussed briefly in Chapter 2) is the most important example of this type of "instruction-splitting" decoupled A/E architecture. This machine, shown in Figure 3, has a single instruction stream. Instructions are fetched from a small 16K instruction cache (which in turn fetches them from a larger 64K unified cache) and fed into an instruction splitter. This decodes the instruction opcode and routes it to one or the other of the units, depending upon the operation to be performed. Compare instructions also set a branch bit in the splitter unit, allowing later conditional branch instructions to test the value before splitting occurs. The various queues allow decoupled loading and storing, loading and storing directly to/from the access unit and the transfer of data between the two units.

The primary advantage of this particular form of decoupled A/E architecture is that it can exploit the parallelism and latency tolerance inherent in access/execute decoupling, whilst appearing relatively conventional externally. The only oddity in the ZS-1 instruction set is the unconventional approach to load and store operations, whereby the load instruction implicitly stores a value in the appropriate queue on the access unit with loaded data being accessed explicitly via top-of-queue registers on the execute unit. There is, however, no reason that a processor of this type could not use the instruction set of an existing processor. Additional decoding of loads and stores in the splitter unit would result in appropriate instructions being placed in the instruction queues and copies between address and data registers could be achieved via the copy queues (again by decoding and splitting internally). Architectures with unified, general purpose register files would present a problem, although it is possible that some form of run-time dependency analysis along with a register renaming scheme could work around the possible implicit inter-unit dependencies that might result.

One problem with this approach, however, is that the use of a single instruction stream severely limits the degree to which the two units can decouple, with the inter-unit asynchrony (or, to use the term coined by Smith, dynamic slippage) being limited by the size of the internal instruction queue. Dual instruction stream architectures represent one way of increasing the degree to which decoupling can be exploited.

3.1.2 Dual instruction stream architectures

The second approach to access/execute decoupling separates the two tasks more completely and is illustrated in Figure 4. Rather than treating the two units as functional units of a single processor, they become more independent. Each has its own instruction stream and the asynchrony is bounded not by the size of internal queues but by dependencies within the program itself. These dependencies occur when the two units must communicate. This can happen if the address unit requires a data value for one of its calculations (e.g. indirection, list following) or if a control decision must be made. The control decision can either be made by collaboration between the two instruction fetch units (which handle control flow) or on the sub-processors themselves. In the majority of cases only one of the two units will make the control decision, however, and the result of this decision must be communicated to the other sub-processor.

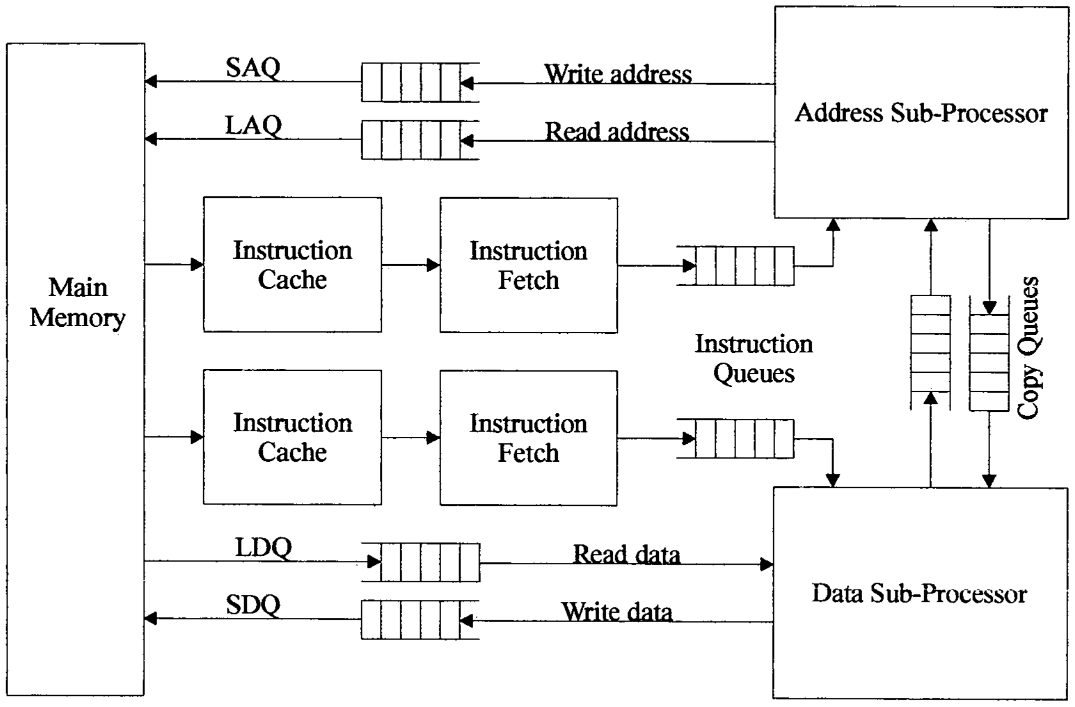


Figure 4 - A dual instruction stream access/execute decoupled architecture.

If the sub-processor making the decision is the address unit, this is not a problem. This unit will typically attempt to run ahead of the data unit providing latency tolerance. Should it be responsible for making a control decision such as evaluating the branch at the end of a loop, it will pass the result of this to the other unit. This incurs no performance penalty, although the degree of decoupling may be limited by the size of the queues via which this information is transferred.

However, if the control decision has to be made on the data unit, such as breaking out of a loop when a particular value is encountered, this information has to be communicated to the address unit. This will typically require the address unit to stop and wait until the data unit “catches up”. This is known as a loss of decoupling (LOD) and, as we will see later, is the major drawback of decoupled architectures. Loss of decoupling appears to be less of a problem in single stream architectures, since control decisions are made before the decoupling point and all instructions in the instruction queues can be safely executed without a LOD occurring. However, since there is less decoupling in a single stream architecture anyway, no performance improvement is obtained.

The other main disadvantage of dual instruction stream architectures concerns the instruction fetch mechanism. Surprisingly, instruction bandwidth is not one of them, since instructions (other than those involved with memory accesses) appear in one or other of the instruction streams, but rarely both. Some code bloat is inevitable, but the increase in object code size is not prohibitive. However, the instruction fetch hardware itself must be duplicated and the memory access patterns produced by the fetching of instructions are less predictable than those of, for example, a dual-issue superscalar architecture. This can be mitigated somewhat through the use of separate instruction caches for each unit but this once again increases the complexity of the architecture.

3.1.3 Decoupled loads and stores

Having stated that decoupled loads and stores are a good thing, they will now be described and details given of exactly why this is the case.

Before beginning, however, some terminology will be introduced. The two parts of a decoupled processor can be referred to by a number of names. Each part is known either as a *sub-processor* or a *unit*. The two terms are interchangeable although the former is preferable when discussing an architecture with multiple instruction streams and the latter is used only for brevity. The sub-processor concerned with generating memory addresses is known as the *address* or *access* sub-processor, while the other is known as the *data* or *execute* sub-processor. Generally the former term (which describes the type of data processed by the unit) is preferable, although the latter (which describes the task it performs) is also acceptable.

Memory accesses are known as *memory packets* and are split into two types. The address sub-processor generates *load packets* which contain the memory address to be accessed as well as other information, the purpose of which will be described later. When a load packet reaches memory the address is used to fetch the appropriate data and a *load response packet* is generated containing the data and other information (some of which may be copied from the first packet). A *store packet* contains data and an address and is "absorbed" by memory.

It should be noted that the use of the term "packet" in no way implies that the memory sub-system uses packet switching. In practice this may well be the case, especially if the processor is part of a multi-processor system, but the term has been chosen chiefly because in most cases the data or address is accompanied by additional information, the whole forming a "packet" of information.

The ideas behind non-blocking loads were discussed in Section 2.4.1. By splitting a load operation into two parts - one that generates the address and one that uses the fetched data - a processor need not necessarily stall while waiting for data to arrive from memory. If the first part of the operation can be issued sufficiently far ahead of the second, the memory latency can be hidden entirely from the "receiving" instruction. In a decoupled access/execute architecture, the two parts of the operation execute on different sub-processors, and in a "good" program the address generating part of the operation can be issued far in advance of the time at which the other unit will need the data. Since the two units run asynchronously, the address sub-processor can potentially initiate many such loads ahead of the time at which the data will be required. Since all of these could complete before the data sub-processor executed its part of even the first load, queues are necessary to buffer load packets that have arrived but not yet been used.

Two queues are used (see Figure 4). The first, the Load Address Queue or LAQ, is part of the address sub-processor and is used to store load packets that are waiting to be sent to the memory sub-system. This provides some degree of decoupling between the address sub-processor and the memory sub-system should, for one reason or another, the memory be incapable of accepting load packets as quickly as the address sub-processor can produce them. The second queue is known as the Load Data Queue or LDQ. This is part of the data sub-processor and stores load packets that have arrived from memory but which have yet to be used. This decouples the data sub-processor from memory, acting as a buffer when packets arrive too quickly for it to cope.

It can be seen above that the load path in fact consists of two decoupled processes, between address sub-processor and memory and memory and data sub-processor respectively. It is assumed that a well-specified memory sub-system will be capable of accepting load packets at least as fast as they can be produced, if the memory access pattern is reasonably uniform. On a machine with a single load path, this will be one load packet per cycle, or a multiple of this if there are multiple load paths. This is generally achieved through the use of highly banked, interleaved memory where the number of banks matches or exceeds the latency of the individual banks when measured in processor clock cycles. To use the example figures quoted earlier, if the processor has a cycle time of 3ns and memory has a cycle time of 60ns, each memory cycle takes 20 processor cycles to complete. If the address sub-processor generates a load packet on every cycle, it can initiate 20 loads during a single memory latency, which requires that memory be 20-way interleaved if full throughput is to be achieved even if memory accesses are "perfect" and access each bank in sequence. In practice some degree of over-banking will be required to handle the less-than-perfect access patterns typical of real programs.

The two decoupled processes that make up the load path can generally be thought of as a single process as long as the memory is sufficiently well-specified. If it has insufficient bandwidth it is *always* going to stall the address sub-processor eventually, and the rate of production and consumption of load packets will be bounded by the rate at which the memory subsystem can process them. A well-specified memory sub-system will present no bottleneck to the two sub-processors, with load packets arriving at the data sub-processor at the same rate that they were sent by the address sub-processor.

Two decoupled processes are also involved when data is stored to memory, although there is no "chain" between the two as with loading. When the address sub-processor executes a store instruction, it places the address to which the stored data should be sent in the Store Address Queue (SAQ). Similarly, when the data sub-processor encounters a store instruction, it places the appropriate data into the Store Data Queue (SDQ). The heads of both of these queues are paired and assembled into a store packet containing both address and data information, which is then dispatched to memory.

Both of the decoupling processes involved here are between a sub-processor and memory. The SAQ decouples the address sub-processor from memory and buffers addresses should they be generated faster than the memory sub-system can accept them, while the SDQ decouples the data sub-processor in a similar fashion. The rate at which store packets are actually dispatched to the memory sub-system is the minimum of the rates at which the two sub-processors generate their respective halves of the store packet. Since the address sub-processor generally runs ahead of the data sub-processor, it will usually be the case that the SAQ will fill until the data sub-processor reaches its first store. At this point stores will begin to be dispatched to memory. This seems to suggest that the SDQ might be redundant, since arriving data will almost always be immediately paired with an address and sent to memory. Whether this is in fact the case will be investigated in the next chapter.

Another design feature that will also be evaluated in a later chapter is one that appeared in the ZS-1 - a dedicated memory path for the address sub-processor. As can be seen in Figure 3, the address sub-processor is connected to memory by two "extra" queues, the ALQ and ASQ, which allow it to load and store data directly from/to memory without requiring the co-operation of the data sub-processor. The standard justification for this design feature is that it allows the address sub-processor to perform such tasks as spilling or reloading registers, performing indirection and list following without causing a loss of decoupling. The alternative would be to perform a decoupled load then pass the value returned to the address sub-processor via the transfer queues, which are described in the next section.

3.1.4 Inter-unit transfers

From time to time it is necessary for information to be passed from one of the sub-processors to the other. There are two ways in which this can be achieved. The most straightforward method, and that employed by both the ZS-1 and the sample dual stream machine shown in Figure 3, is to use architectural queues. These are mapped into the sub-processor's register file and may be accessed as registers in a similar manner to those queues concerned with loading and storing. This method is easy to use and fast but requires a considerable amount of additional hardware. Two extra queues are needed, and the internal data path of each sub-processor is complicated by the mapping of the queue heads onto register file.

A cheaper alternative, if the address sub-processor has a dedicated path to memory, is to perform inter-unit transfers via memory. For example, to transfer data from the address sub-processor to the data unit, the address sub-processor would first issue a self-store then initiate a decoupled load to the other unit. A transfer in the other direction would be achieved by performing a decoupled store then executing a self-load on the address sub-processor.

3.1.5 The problems of distributed control

The main weakness of access/execute decoupling is the need to communicate control information from one sub-processor to the other on each control decision. The single stream architecture only suffers from this problem to a minor extent, since the state information required for control flow decisions can be centralised in the instruction splitter, through which instructions for both units flow. This single locus of control means that control flow decisions are only complicated on those occasions where a decision is made based on information from *both* of the sub-processors.

In a dual stream architecture, however, the distributed nature of flow control is more of a problem. Each control decision must be communicated to the other unit, which can result in a loss of decoupling. One possible solution is to centralise the control. While this will not always eliminate the loss of decoupling, it does have many other advantages. Separating the execution of a program into three separate threads - access, execute and control - results in what is known as a Fully Decoupled Architecture.

3.2 Fully Decoupled Architectures

The ideas behind fully decoupled architectures are rather newer than those behind access/execute decoupling. The concept was first proposed by Peter Bird et al in 1991 [Bird91] who termed this a “semantically-driven architecture”. This name arose through the partitioning of the program onto specialised processing units closely following the various components of the computing formalism known as Action Semantics [Moss94]. The control task corresponded closely to the interpretation of the structure of the action semantics "rules", the access task to the binding of variables and the execution task to the semantic evaluation.

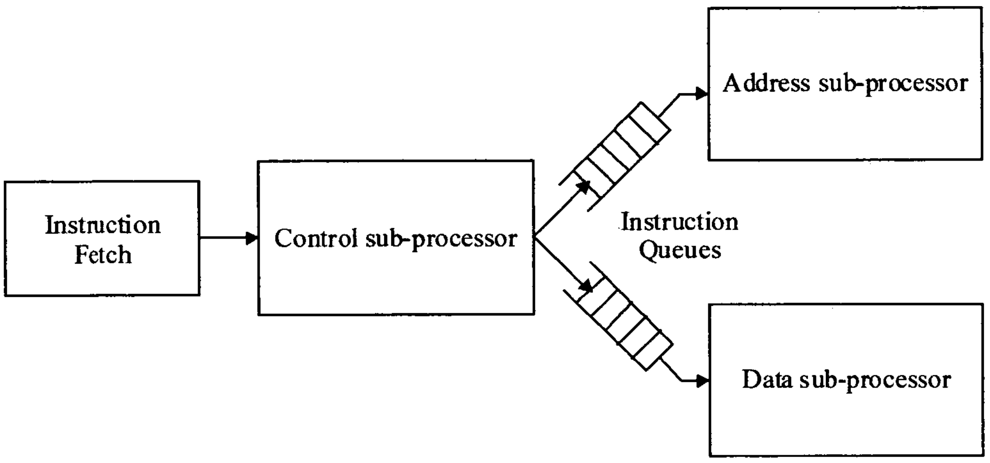


Figure 5 - A simple instruction dispatch mechanism.

A fully decoupled architecture has three sub-processors. Two of these are already familiar from access/execute decoupling, while the control sub-processor performs all control flow decision making and dispatches instructions to the other sub-processors. It can do this in one of two ways. The first, and simplest, is illustrated in Figure 5. All instructions pass through the control unit and are either executed on the control sub-processor directly (control flow instructions) or passed to one of the other two units. In this respect the architecture differs little from the ZS-1, except that the control sub-processor is more complex than a mere instruction splitter and that conditions are evaluated on the other sub-processors then passed back to the control SP to be acted upon.

Given the similarity between this approach to control decoupling and the ZS-1's treatment of access/execute decoupling, it should come as no surprise that this single instruction stream form of full decoupling suffers from the same problem - the available degree of decoupling between the control unit and the other units (and also between the sub-processors themselves) is limited by the size of the instruction queues.

The second approach to control decoupling rectifies this problem. It makes use of the fact that control flow instructions are only a small proportion of the total instructions that make up a program and each control instruction (a branch back to the beginning of a loop, a subroutine call) typically is followed by a sequence of uninterrupted instructions (a basic block). Since reaching a particular node in the control flow graph means that *all* of the instructions directly associated with that node are going to be executed, there is no need for the control processor to individually fetch and dispatch each instruction. Instead, as shown in Figure 6, it places "meta-instructions", high level basic-block descriptors that describe a sequence of sub-processor level instructions, in the meta-instruction queue, or MIQ. This queue does not feed directly into the sub-processor pipeline as with the simpler scheme but instead feeds into a specialised instruction fetch mechanism, or Fetch Engine. This is responsible for removing meta-instructions from the MIQ, decoding them and fetching the instructions which it then feeds to the sub-processor. Thus, a processor using this variety of full decoupling has three separate instruction streams.

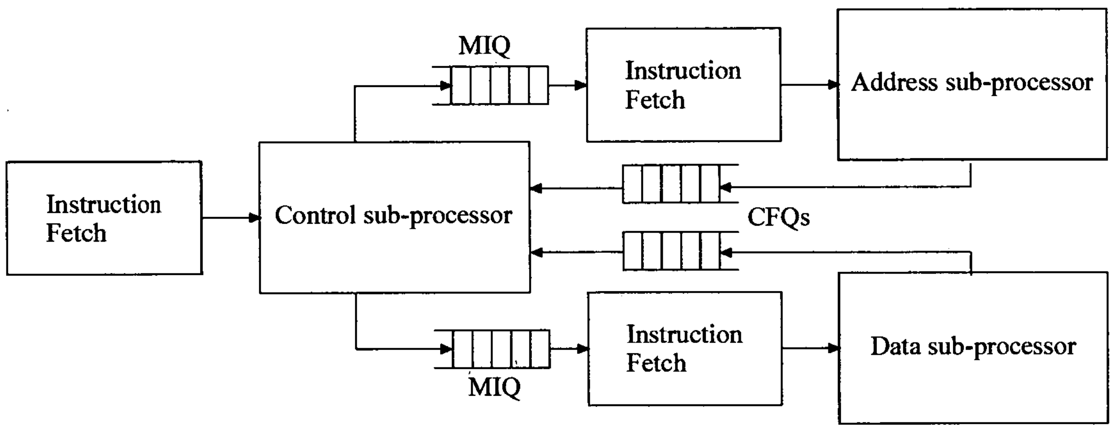


Figure 6 - A meta-instruction dispatch mechanism.

The access and execute sub-processors are known collectively as the work units because they do all the work, while the control sub-processor just tells them what to do. They are able to pass results back to the control SP which it then uses to make control flow decisions. This is done via yet another set of queues, the Condition Feedback Queues (or CFQs). These have the usual purpose of allowing some degree of decoupling to be maintained even if the rates at which results are produced and consumed differ.

3.2.1 MIQ and CFQ design issues

Unlike the other queues discussed so far, where the width has depended entirely upon the data or address size for the particular architecture, the width of the CFQ and MIQ is a design parameter.

Initially, a single bit wide CFQ was thought sufficient to pass the result of a simple test back to the control SP. However, it was noted that this rendered the efficient implementation of, for example, multi-way switch structures difficult. While control structures of this type could certainly be implemented, using a series of `IF-THEN` tests each of which would return a result to the control sub-processor, this would be hugely inefficient. Far better to pass back a full data word and use this as a jump table index for an indirect jump implemented on the control sub-processor.

The width of the MIQ is dictated by the meta-instruction format. The format of this depends upon whether the control sub-processor issues a single meta-instruction which goes to both work units, or issues one to each unit in turn. In the first case the meta-instruction must contain start addresses for the appropriate data and address sub-processor basic blocks and the lengths of each. In the second case, only a single start address and length are needed.

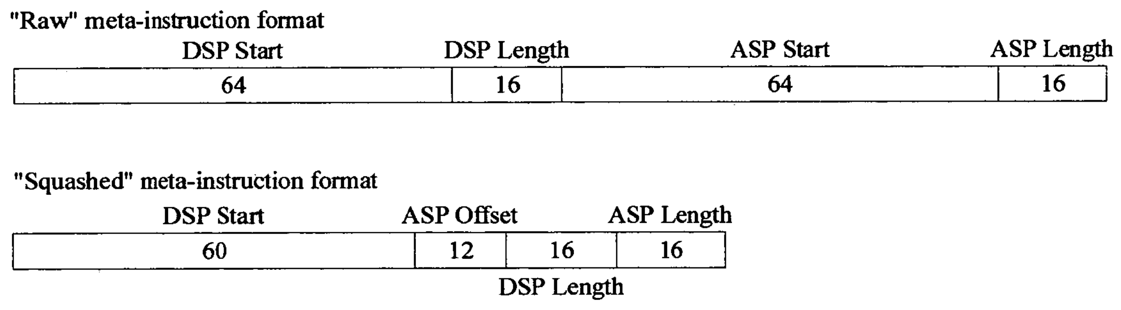


Figure 7 - Two possible meta-instruction formats.

In an architecture with a 64 bit address and which supports block lengths of up to 64K instructions (i.e. 16 bit block length) this requires a queue 160 bits wide. This "raw" format is shown in Figure 7. If this is considered excessive, however, some compression may be performed and the meta-instruction squashed. If basic blocks are likely to be over a certain length (say, 16 words), 4 bits can be stripped from each address if basic blocks are assumed to be aligned on a 16 word boundary. This will cause some fragmentation of memory, with dead space between blocks of code. If the sub-processor code for the data and address sub-processor basic blocks is stored consecutively in memory, the start address information for both sub-processors could be represented by giving a start address for the first of the two consecutive blocks plus an offset for the second sub-processors code. Using the example sizes given above and alignment to 16-word boundaries, this would require 60 bits for the start address plus a 12 bit offset, as well as the block lengths. This format is also shown Figure 7 and requires just 104 bits.

The large size of meta-instructions raises some design issues concerning the control sub-processor instruction set. In recent years architectural trends have tended to favour architectures with fixed width instruction sets. This simplifies instruction decoding and removes problems with instructions straddling page boundaries in virtual memory systems. Given a typical RISC architecture, 32 bits is usually sufficient to implement a reasonably complete RISC instruction set. One would imagine that the control sub-processor would use such an instruction set, with the usual retinue of control flow and integer arithmetic operations. This seems to be at odds with the wide instructions required for meta-instruction issue.

However, it should be remembered that from the point of view of the control sub-processor meta-instructions are just pieces of data describing basic blocks rather than an intrinsic part of the sub-processor. With this in mind, it may be more appropriate for the architecture of the control sub-processor to reflect this, providing specialised wide registers to hold meta-instructions, mapping the MIQ into this bank of registers, and providing operations to load, store and manipulate the contents of these registers.

At first glance, there may seem to be no advantage in providing so many operations that act upon meta-instructions. However, in the case of a tight loop it might be advantageous to issue meta-instructions in this way. With wide instructions that treat the meta-instruction data as literal information, each instruction issue may take 4-6 cycles to issue (since the instruction will be 4-6 words long) and the same literal data will be loaded on each iteration. With meta-instruction registers, the block descriptors could be loaded into registers before the loop was entered and each meta-instruction issue would take only a single cycle (since it would be achieved by a single register-

register copy). In fact, the control sub-processor instruction bandwidth could be reduced considerably for many subroutines which execute a small number of basic blocks repeatedly. Given the skeletal nature of control sub-processor programs, where issue operations constitute a large proportion of the instructions executed, the savings obtained by preloading the meta-instruction registers with the descriptors for all the blocks in a subroutine could be considerable.

The meta-instruction formats described above can also be thought of as instructions for the sub-processor fetch engines, with the operation (fetch *x* instructions starting at address *y*) being implicit. As later chapters will show, however, meta-instructions may contain additional information, such as an iteration count specifying how many times the basic block should be repeated, and condition or guard bits. Although these will be examined in detail later, they do have an effect on the meta-instruction format. They will require additional bits to store this information and will effectively add opcodes to the meta-instructions, changing the actions performed by the fetch engine accordingly. For this reason it may be advantageous to be able to modify meta-instructions dynamically (to set iteration counts, perhaps, or to specify conditions to be checked) in the control sub-processor. This is another advantage of the flexibility that could be provided by the use of meta-instruction registers.

More detailed investigation of meta-instruction formats and the relative strengths of meta-instructions as literals versus meta-instruction registers would form a major topic of study in itself. Their impact on performance is, however, of some interest and may be worthy of investigation at some later date.

4. The Modelling And Simulation Of Decoupled Architectures

The use of modelling and simulation is very important to today's computer architect. The time and cost involved in physically implementing a new processor design mean that much of the initial work, perhaps even as far as the development of the operating system, will be performed on a simulated processor, and much of the performance analysis done with a mathematical model. In the case of a whole new family of architectures, where general design concepts are being explored rather than the behaviour of a single design, modelling and simulation techniques are also useful because they allow various parameters of the system to be easily changed, and their effect on performance examined.

There are a wide spectrum of modelling and simulation techniques available to the computer architect. At one extreme there are completely analytical models where the behaviour of the architecture is reduced to a system of equations. These can be used, with the variables that represent certain system parameters set accordingly, to obtain numerical values for a number of features of the system being modelled, or to analyse the system's steady state behaviour.

The main advantage of the analytical approach is that once the appropriate systems of equations have been obtained, examining the effect of changes in various parameters is straightforward and quick, requiring only the solution of the system of equations that represents the system. In many analytical models it is also possible to see trends in the architecture's behaviour just from looking at the equations directly, which may illustrate clearly which factors dominate performance and give greater insight into the behaviour of the architecture.

However, the areas in which a fully analytical model can be used are limited. Not all systems are amenable to analytical modelling. It may be the case that the system being modelled is very complex. This will usually require that the model of the system be simplified to make it possible to obtain the necessary equations. This simplification can result in a loss of realism and even then is not always possible.

At the other extreme we have architectural simulations and processor emulators. These may vary greatly in the level of detail at which they simulate the architecture. At one end of the detail spectrum we have high-level simulations. These deal entirely in abstract concepts ("a processing node", "a memory module") and provide a quick but approximate picture of system performance. At the other end are transistor level emulators. These simulate a specific processor architecture in great detail - right down to the transistor level - and very precise measurements can be obtained.

The primary weakness of this type of modelling is that, although high-level simulations may provide a comparable degree of accuracy and speed to an analytical model, it is often far from clear how the various components that make up the system being modelled relate. In high-level simulations the processor, memory and other components may be little more than black boxes. You set the appropriate parameters, start the simulation running and at the end obtain some measurements. These measurements may accurately map out the performance envelope of the processor architecture being modelled but give little indication of the mechanisms by which the various components interact. Often intensive and exhaustive simulation will be required to obtain a reasonable feel for the system's behaviour and, if there are many variables involved, the resulting simulations may take a great deal of time to perform.

Additionally, to produce a reasonable high level model may well require that we already understand certain of the fundamental principles governing the architecture's behaviour, in which case we may well already have had to produce an analytical model.

More detailed simulations may provide more insight into the architecture's behaviour, since they "open" the black box components and allow their internal workings to be examined, but as the level of detail simulated increases, so does the run time of the simulation. Given that a full transistor level simulation of a high performance architecture may run at perhaps 1,000,000th of the speed of the processor it represents, it is often impossible to simulate the execution of realistic workloads since the simulation of even just a second of processor time may take weeks. Additionally, the use of a low-level simulation requires that many (perhaps irrelevant) features of the architecture be defined in detail. While this may be desirable towards the end of a processor's development, to test a design just before fabrication, it is probably overkill to use a simulator of this sort for initial investigations.

Between these two extremes lies a third option, the *simulated analytical model.* It may be the case that the basis for an analytical model, such as a queueing network, can be derived for a system, but that certain features of the model may make it impossible (or at least extremely difficult) to "solve" the model, make the model unacceptably inaccurate or make the equations very computationally intensive to solve.

In this eventuality, a very high-level simulation model can be built. Rather than providing only an approximate simulation of the real system, this approach results in an accurate simulation of an analytical model of that system. While this approach lacks the transparency of a true analytical model, it is quicker to run and simpler than a full simulation model. These two factors in combination make running exhaustive simulation studies more practical, while the simplicity of the model makes the interactions between components more obvious than in a complex and detailed model. They also allow the validity of the analytical model to be established by comparison with a more detailed architectural simulation, even if the equations describing the model cannot be obtained.

In the remainder of this chapter we will examine whether and how each of these approaches can be applied to the task of modelling a fully-decoupled architecture. Their relative merits and applicability will be discussed, and details of the methods used to produce each model given.

4.1 An Analytical Model

Given the architectural features of a fully-decoupled architecture, the most obvious method of obtaining an analytical solution for this class of architecture is to model the system as a queueing network [Tri82] [Wal88].

On the face of it, it would seem that the architecture is ideally suited for this. The processors and memory modules can be modelled as servers, with finite queues directly corresponding to the architectural queues in the system. However, as we will see, some features of the architecture render precise modelling impossible and make even approximate modelling impractical.

It was apparent from a fairly early stage that it would not be possible to build a single, unified analytical model for the architecture. Queueing networks excel when it comes to modelling the steady state of a system. However, the execution of a program on a fully decoupled architecture may have no single steady state. Loss of decoupling events, which we will discuss in detail in Chapter 6 represent discontinuities in program execution. Between two loss of decoupling events the system may well reach a steady state before the discontinuity forces a recoupling. In any non-trivial program it is likely that there will be at least one loss of decoupling and that the steady state before and after each LOD may not be the same.

Even if some way was discovered to remove the problem of the discontinuities caused by loss of decoupling events, analytical modelling of the system is still far from straightforward. To illustrate this, we will look at the problems associated with modelling only one small part of a fully decoupled architecture, the access/execute load path.

We would ideally like to model the load path of our architecture using the queuing network shown in Figure 8. This models the path as a server (the address subprocessor) feeding loads at some Poisson distributed rate R_a into a blocking finite queue (the LAQ) of length Q_a with unit service time. The LAQ feeds these loads, with a uniform random distribution, into a system of m servers (the memory banks) each of which is preceded by a blocking finite queue of length Q_m (the bank buffer). These servers process loads at some fixed rate R_m and all the servers feed into a finite re-ordering buffer (the LDQ) of length Q_d. This in turn feeds another server (the data subprocessor) which removes loads from the LDQ at some Poisson distributed rate R_d. The finite number of loads which may be active in the memory system at any time is set to q_d, and we keep these loads in the system by closing the network so that loads which have been removed from the system circulate back to the address subprocessor.

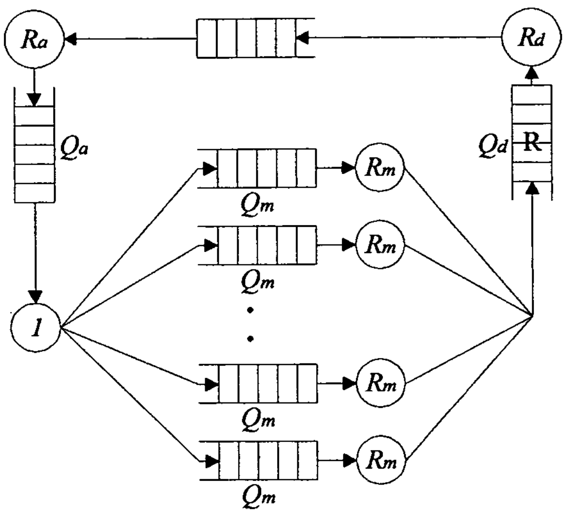


Figure 8 - Load path queueing model.

This network has two problems which make analytical modelling difficult. These are the numerous blocking queues and the presence of a re-ordering buffer. In this section it will be shown that the complexity of an analytical model of this system is such that a simulation model is more effective, both in terms of the ease with which results can be obtained and the "visibility" of the processes that govern the system's behaviour.

The use of blocking queues disrupts the system, as the arrival time of a customer in a queue is no longer independent of all previous arrivals. If the queue is full, the arrival of the next customer will be delayed until such time as space becomes available. The flow through a blocking queue corresponds to no known distribution. If we are willing to sacrifice accuracy and use a heuristic, the behavior of a blocking queue may be modelled in an approximate fashion.

We do this by assuming that all flows in the network are independent Poisson processes, and adjust the rate through the queue i downwards to $\lambda_i=\lambda(1-B_i(\lambda))$, where λ is the unadjusted rate of the arrival process, and B_i is the probability that the queue will block. As queue length increases, the probability of the queue being in a blocked state decreases, and the adjusted arrival rate tends to the ideal arrival rate as queue length tends to infinity. Conversely, as the probability of the queue being blocked increases, the arrival rate tends to 0. The probability B_i is the probability that the current queue length is q_i - that the queue is full - and this can be determined using standard techniques for calculating the steady state probability of the queue being this particular length.

However, even the approximation of the blocking queues by this technique does not remove all the problems. The second major difficulty surrounds the nature of the LDQ, which is a re-ordering buffer.

Just as it was possible to approximate the behaviour of a blocking queue by adjusting the queue arrival rate to take account of the likelihood that the queue is full, a similar approximation technique seems a likely candidate for modelling the effects of a re-ordering buffer. By adjusting the service rate of the DU server, we can compensate for the possibility that data may arrive in the LDQ out of order.

The combination of a re-ordering buffer and a memory system with multiple banks, each possessing a small local buffer, enforces an unusual queueing discipline. Memory accesses can arrive in the buffer in almost any order, and are removed from the buffer in the order in which they entered the load path as a whole, rather than the order in which they arrived at the buffer itself. While it is no longer necessary to model blocking effects (the length of the queue is fixed at the maximum number of loads that may be active in the system at any time) it is necessary to take into account the possibility of loads overtaking each other in the memory subsystem.

While an approximation can be obtained for the processing rate of a re-ordering queue, the end result is complex, requires an excessive number of simplifying assumptions, and is in all probability more opaque than a simulation would be. When this is coupled with the inaccuracies introduced by the approximation of the blocking queues used in the memory modules, the end result is an analytical solution that is both highly inaccurate and in which it is difficult to see the underlying behaviour of the system. It thus lacks both of the main advantages of an analytical model In addition, later experiments indicated that the model described here had further flaws, which would render a useful analytical solution still more unlikely.

4.2 A Simulation Model

Although we have shown that analytical modelling of the queueing network shown in Figure 8 is either impossible or approximate and impractical, there is no reason that the behaviour of that queueing network cannot be modelled by simulation.

With this in mind, a simulation model was built. The model was constructed using simplified implementations of the C++ objects used to build the full architectural simulator (which is described in Section 4.3). The model accurately reflected the queueing network, with the only major changes being the elimination of the feedback queue used to restrict the number of memory accesses in the network and a switch to discrete rather than continuous timing. Since each memory access had to be tagged anyway (to ensure correct re-ordering at the LDQ), a tag reservation scheme was used to restrict the number of active memory accesses. Each load access produced is assigned a numeric tag (actually its eventual LDQ position). If no tags were available (meaning that all LDQ positions were either occupied or reserved for accesses currently in other parts of the network) no load accesses were generated. This is equivalent to the closed network since if the entire population of loads is in the LDQ, LAQ and memory, there can be none in the feedback queue. If there are no loads in the feedback queue, the AU server cannot re-circulate them.

There is no reason that the simulation model could not directly simulate the feedback queue (with the queue initially containing Q_d accesses), but since the components from the architectural simulator already provided the necessary hooks to implement the tag reservation scheme, and the tags were required for re-ordering anyway, the method described was used. Since modelling the network by simulation was very straightforward compared to analytical techniques, the model was also extended to include the access/execute store path, something which would have been difficult to model analytically due to it requiring the pairing of access halves from the AU and DU.

Load and store accesses are both generated in a producer object that represents the AU. It is possible to specify that accesses be generated either according to a Poisson process (for which the mean can be set) or at fixed intervals. The latter allows more accurate modelling of certain situations. For example, it may be the case that in a process we wish to simulate the producer generates a load on every second cycle. To simulate this accurately we want loads generated at a fixed, deterministic interval, rather than distributed around this value, as would result if a Poisson distribution were used. In those situations where a Poisson distribution is used, the service times of the appropriate server (the gap between loads or stores) are rounded to the nearest time unit, due to the simulator using discrete rather than continuous time.

Loads and stores are only generated if space is available in the appropriate queue. If either queue is full, the corresponding server stalls until space becomes available, whereupon an access is immediately generated. This is more accurate than the blocking approximation proposed in the previous section, since it no longer assumes that arrival times are independent. There is an additional restriction on the generation of loads that they may only be produced if there are less than Q_d loads currently active in the memory subsystem.

The LAQ and SAQ feed memory accesses into a network of memory modules. As each load or store reaches the front of the queue it is assigned a memory module as its destination. This assignment is produced in one of two ways. It can be randomly generated, with memory accesses being uniformly distributed over the available modules, or a deterministic distribution with a user-specified stride can be used. The latter would be difficult to model analytically but, as with the fixed interval access generation described above, can allow the more realistic modelling of program behaviour. Random, uniformly-distributed accesses are useful when modelling the behaviour of a system with an address remapping scheme, such as Rau's hashing scheme [Rau91].

If the assigned module is busy and there is no space in its buffer, the load or store will be held and the queue stalled until such time as the module is idle or buffer space becomes available. Again, this models the stalling behaviour exactly, which is preferable to an approximation.

The movement of memory accesses from queue to memory module is one area where the simulation model introduces inaccuracies. Modelling the pairing up of the two halves of a store would not be easy in an analytical model. Additionally, no check is made for the dependencies between loads and stores which would occur in a real system. Modelling this analytically would be tricky, since it would involve stalling accesses if they caused a hazard, and implementing this in even a simulation model would require that the memory access patterns be modelled at a finer level of granularity than is desired. Finally, no check is made to see if the physical constraints that would be present in a queue-memory interconnect are violated, and it is quite possible in the simulation model for a load and a store to be transferred to the same bank at the same time. This would not happen in a real system.

The memory modules themselves are modelled simply as a finite queue feeding into a server with a fixed service time. Both the queue length and the service time (memory latency) can be varied. The use of a simulation modelling technique also allows separate access and cycle times to be specified for the memory module. This could no doubt be modelled analytically, but would complicate the analytical model still further.

Serviced loads are then transferred into the LDQ. The model makes a simplification here that, while accurate in terms of the queueing network that the simulation represents, does not accurately reflect the behaviour of a real system. No restriction is made on the number of loads that can be transferred from memory to the LDQ at a time. In principle, every module could finish processing a load and attempt to transfer a value to the LDQ during the same cycle. In practice, however, this is unlikely and tests with the module show double transfers being rare, triple transfers rarer still, and the transfer of more than three loads at a time is unknown. Thus the inaccuracies introduced are small.

At the end of the load path, loads are removed from the LDQ and sent to the consumer server, where they are absorbed and their assigned tags freed. As with the producer, the consumer server can have its service time set to be Poisson distributed or set to a fixed interval for greater accuracy when modelling the execution of certain deterministic access patterns.

The simulation model is heavily instrumented, which goes some way towards rectifying the lack of transparency compared to an analytical model. Data regarding various intermediate latencies (i.e. the time taken for a memory access to reach a specific point in the network) is collected, not only in the form of the usual statistical values such as means and standard deviations, but also in the form of detailed information from which accurate pictures of latency distributions can be formed.

4.3 An Architectural Simulator

While high-level models, both analytical and simulation-based, allow performance trends in an architecture to be examined, they provide only an approximation of the architecture's behaviour. It is desirable to obtain more exact measurements by "fleshing out" a skeletal model into a full-blown architectural simulation from which it is possible to examine the behaviour of a less idealised system.

With this in mind, an architectural simulator was developed to complement the simulation model. The simulator that was produced was known as `fdps`, an acronym for Fully Decoupled Processor Simulator, and was intended to allow the fast and flexible simulation of a reasonably clearly-defined processor architecture.

The simulator had its origins in `ades`, a simulator developed by S. Manoharan as part of the ACRI-1 project. Although `fdps` used the same language (C++), internal structure (direct mapping of hardware components to objects) and simulation method (time rather than event driven) as `ades`, by the time it was completed there was almost no code commonality.

Both simulators were written in C++, since the object-oriented design approach is well suited to architectural simulation. It is relatively simple to map hardware components onto C++ objects, resulting in code that makes the relationships between hardware components explicit and easy to understand.

The simulation methodology adopted was to simulate the architecture on a cycle by cycle basis, rather than using an event-driven approach. This decision had both advantages and disadvantages. The main argument against cycle by cycle simulation is that if a simulated component remains in the same state for a substantial period of time, it is unnecessary and wasteful to simulate that component's behaviour for every single cycle of that period. An event-driven approach would require only that the component be simulated at those times where its state changes.

An example of this is the simulation of a memory module. This accepts memory requests and, some fixed latency later, produces the requested data. When an event-driven approach is used, the arrival of a memory request at the memory module is an event which in turn schedules a second event (the production of the requested data) at some fixed time later. There is no need to simulate the memory module at any time between these two events. By contrast, the time-driven approach requires that each and every cycle of the memory access be simulated. The arrival of a memory access may set an internal counter which, when it reaches zero, causes the requested data to be output. In a time-driven simulation, every component is simulated on every cycle, even if the component is idle or it is known that its state will not change for some time.

On the face of it, the event-driven approach seems far superior. However, the case for event-driven simulation is not as strong as it at first seems, especially in architectural simulations. While the memory module simulation described above does seem to perform a lot of unnecessary work, in practice the overhead of performing a simple decrement-and-test operation is low. Also, at the opposite extreme we find simulated components such as processors where many events may take place per cycle. For example, in a simulation of a pipelined processor, each pipeline stage may produce several events on every cycle.

Additionally, there are hidden overheads incurred by the use of event-driven simulation. Some mechanism must be provided to ensure that events occur in the correct order and using event-driven simulation in conjunction with an object-oriented simulation also requires some method of ensuring that events are delivered to the appropriate component. Often it is also the case that a simulation that would be straightforward to simulate on a cycle by cycle basis requires considerable restructuring before it can be effectively simulated using an event-driven system.

These issues, combined with the important fact that `ades` had already been written with cycle by cycle simulation in mind, lead to `fdps` adopting the same simulation methodology as its predecessor. As will be seen later, the cycle-based simulation approach does lead to some performance difficulties when large simulations are run, but on the whole `fdps` was found to perform very satisfactorily.

Another issue in the design of architectural simulators concerns the methods by which the simulator is made to run real programs. In the simplistic case it is often possible to avoid this issue altogether, producing a simulator where loads and stores are produced on a probabilistic basis. However, if we wish to examine the behaviour of an architecture running real programs (or even just standard benchmark kernels) there has to be some way to make the simulator's behaviour reflect that of a real machine running the real program. The two most common approaches to this problem are instruction emulation and the use of instruction traces.

Instruction emulation involves producing a simulator that directly simulates the actual execution of a real program. Instructions are fetched from a simulated memory, decoded by a simulated instruction decode stage and passed down a simulated pipeline, setting simulated flags and operating on simulated registers as they do so.

The main advantage of instruction emulation is that it allows very accurate simulation of the program behaviour, since we are effectively emulating the architecture rather than just simulating it. If we wish to simulate an architecture that uses an existing instruction set and encoding, we can use the standard tools (compiler, assembler etc.) for that architecture to produce object files that, with a little work, can be loaded into the simulator and run.

However, these features of instruction emulation are also its main weakness. The level at which the architecture must usually be simulated may slow the simulation down considerably, while the problems associated with emulating a new instruction set are manifold. To produce usable object files will require at least the development of an assembler, the detailed definition of the instruction encoding and probably the development of a compiler too. The amount of work involved is large. If the architecture being simulated is likely to be built eventually, the costs associated with developing these tools may be reduced, since they would be required for the eventual machine too. However, in the common situation where a new architecture is in a state of almost continual flux with frequent design changes, it may be difficult for the assembler and compiler to keep up with instruction set changes. It is also not always desirable to tie an architecture down to a specific instruction set and encoding so early in its life cycle.

The other approach to running real programs is to use instruction traces. This involves in some way annotating a program (either by modifying the program source prior to compilation or by direct modification of the object code) so that when run it produces a log recording the behaviour of the program. This log may range in the level of details it records from a direct instruction-by-instruction list of those operations performed when the program was run to a record of the times at which certain events occurred, and all points in between.

The primary disadvantage of using instruction traces is the size of the log files produced. Unless only a very small subset of program events are being recorded these may be very large indeed. For example, a complete instruction trace for a program that runs for just 20 seconds on a 50 MIPS architecture will produce a trace containing 100,000,000 instructions. If the instructions recorded are 32 bits each, this will require some 400MB of disc space.

However, one of the good points of using instruction traces is that it is possible to record only that information which is needed. If a cache is being simulated the trace need only contain details of memory accesses. If a branch target buffer is being tested we only need details of branch instructions. It may also be the case that we are interested only in the program's behaviour in a very general sense, in which case a greatly simplified abstract instruction set may be used. The use of a simple abstract instruction set may also speed up the simulator being used which, when combined with the ability to use a single trace file for multiple simulations, can save a great deal of time.

Object annotation is invariably done automatically - the task of manually modifying an object file, adjusting branch destinations and so forth is one that dates back to the pre-history of computing and that is nowadays done only in extreme circumstances. Source annotation can, however, be done either manually or automatically, although the practicality of the former depends largely on the format and complexity of the annotation information being generated and the size of the source being modified.

In all but the simplest cases (single kernels from the Livermore loops, trivial test programs) it is best to use an automatic source annotator. Producing such a tool is not a minor task. At worst it may require the implementation of the same optimisations as would be required by a full-blown compiler. However, it is generally easier to write an annotator capable of annotating a program with the equivalent of reasonably good code than to produce a compiler capable of actually generating that code. Also, with a suitable abstract instruction set, annotation may be greatly simplified.

The relative merits of source and object annotation depend largely on the tools available to the programmer and to the desire for portability. Annotated sources are more portable than annotated objects. As long as a compiler for the language being annotated exists on the desired platform, portability can be achieved, although the end result may be slower to run than if object annotation had been used.

4.3.1 The Source Annotator

Before describing the structure of the simulator itself, it is useful to examine the tool that generates the instruction traces that it uses.

The first approach tried was to modify an existing simulator for the (paper) DLX [HP90] architecture so that, when programs were run on it, traces were produced. A C compiler already existed for DLX, so this was seen as potentially being a low cost route to trace generation.

However, there proved to be a number of weaknesses to this approach, which were mainly concerned with the supplied C compiler. While it proved easy to modify the simulator itself to generate trace information, the C compiler had a number of limitations. Foremost amongst these was a tendency to crash severely when asked to compile anything other than trivial code. Even when the compiler could be persuaded to compile code, the resulting object code (actually DLX assembly language) was far from optimal.

Additionally, the approach taken required that an additional tool be developed to process the resulting traces (one each for the access and execute processors) into a fully decoupled trace, generating the control processor trace and blocking the other traces into meta-instructions at the same time.

In the end it seemed that this "low cost route" was in fact unnecessarily complex, and work on this approach to trace generation was abandoned. It was decided instead to develop a source annotator that would directly modify program sources, which could then be compiled and run to produce traces.

The basis for this annotation tool was the Sigma toolkit [Gan92] developed by Professor Dennis Gannon and associates at the University Of Indiana. Sigma consists of two main components. The first of these, a compiler called `cfp`, reads in a source program (written in Fortran 90, and possibly consisting of several separate modules) and produces as output not an object file but a database that represents the parse tree of the program.

The second component of Sigma is a C library that can be used by a user application to analyse and/or modify the database. After any desired changes have been made to the database, it can be unparsed to generate a modified source program.

Although Sigma is primarily intended for analysing and modifying sources with a view to parallelising them, it proved very useful for developing a source annotator.

The tool that was developed used Sigma to traverse the parse tree of a Fortran program, annotating each procedure of the source in turn, to produce an annotated source which could be compiled, linked with a library and then run to produce a trace (see Figure 9). Each procedure is traversed in turn, statement by statement, and an internal list built up that describes the annotation of the source so far. When a basic block boundary is detected (generally a statement that affects control flow or is the destination of a jump from elsewhere) the corresponding annotation is placed in the database and a new internal list started.

The annotator makes a small number of simplifying assumptions when analysing code. It is assumed that the vast majority of scalar accesses will be met either from a register file or a data cache, so scalar variable references generate no memory accesses and are treated in the same manner as arithmetic instructions. Similarly, it is assumed that no attempt will be made to cache array accesses, and a decoupled load is always generated for these.

The annotator performs a number of standard optimisations, such as constant and copy propagation, constant folding, common sub-expression elimination and a number of more specific peephole optimisations. The resulting traces correspond to those which might be generated by a compiler with an intermediate level of optimisation. Should more optimal tracing be required, individual annotated source routines can be modified by hand.

The internal list that is built is, in fact, a collection of three lists, one for each sub-processor. Operations are placed on a particular list depending both on their type and their context. Control flow operations are always placed on the CU list, memory accesses are generally placed on both the AU and DU while expressions are placed on the DU trace. Obviously there are exceptions to this. Expressions that are used to calculate an array index are placed on the AU and, depending upon the level of optimisation specified, loop conditions may be placed on the CU list.

Other placement decisions made by the annotator depend largely on which command line flags are specified. The most important of these allows the richness of the processor to memory interconnection to be specified, while others allow for processor architectures with conditional or guarded meta-instruction issue and for a number of methods of loop execution. These options

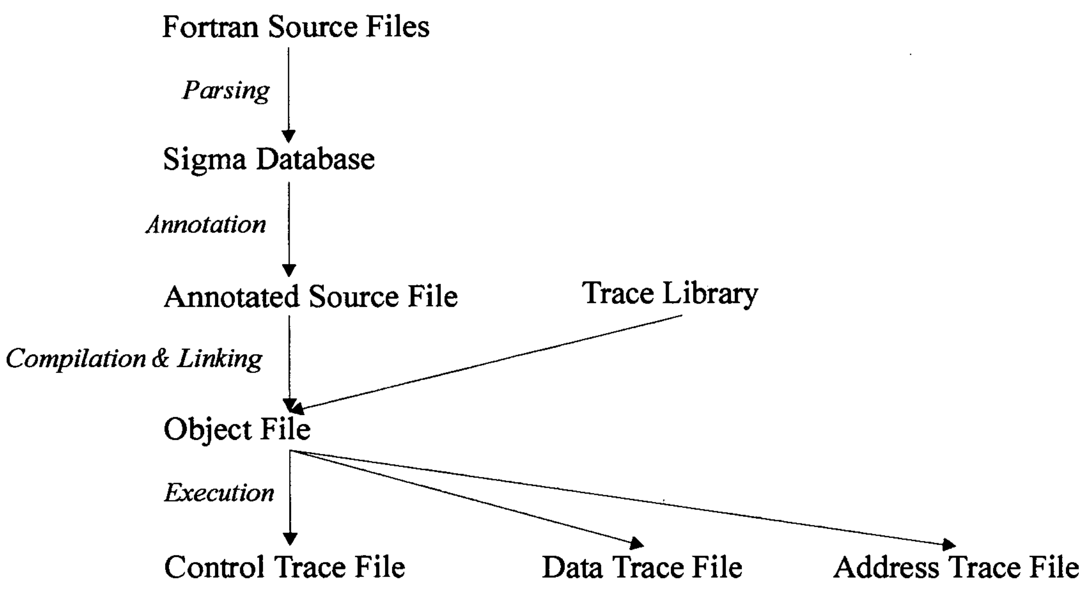


Figure 9 - The annotation lifecycle.

collectively allow code to be annotated for all three levels of fully decoupled architecture (which will be described later). Additional flags allow the user to specify whether generated traces should be automatically compressed, and whether the trace information generated should be human-readable, numeric ASCII or in a binary format. Only the last option is supported by the simulator, but the others allow generated traces to be more readily examined.

After annotation is complete, the annotator outputs a modified program source. Figure 10 shows the results of annotation for a short program (a SAXPY loop). The source is annotated by the addition of calls to externally defined procedures which correspond directly to the supported simulator primitives. The operation is specified by the name of the procedure while the parameters specify an operation count, a total cycle count (not present in all primitives) and, in the case of memory access operations, a memory address.

Original source

```
      program example
      real y(100), x(100), a
      integer i

      do 10 i = 1, 100
        y(i) = a * x(i) + y(i)
10    continue
      end
```

Annotated source

```
      program example
      real y(100), x(100), a
      integer i
      call Open_Trace ('prog1', 1, 0) ;          ] Open program trace
      call CU_Issue_AU ()                        ┐
      call CU_Op (1, 1)                          │ Loop prologue
      call AU_Wait_CU ()                         │
      call AU_Op (1, 1)                          ┘
      do 10 i = 1, 100, 1
        y(i) = a * x(i) + y(i)
        call CU_Issue ()                         ┐
        call CU_Op (2, 2)                        │ CU loop body
        call CU_LocalBranch (1, 1)               ┘
        call AU_Wait_CU ()                       ┐
        call AU_DU_Load (loc (x(i)), 0)          │
        call AU_DU_Load (loc (y(i)), 0)          │ AU loop body
        call AU_DU_Store (loc (y(i)), 0)         │
        call AU_Op (1, 1)                        ┘
        call DU_Wait_CU ()                       ┐
        call DU_Load (1)                         │
        call DU_Op (1, 1)                        │ DU loop body
        call DU_Load (1)                         │
        call DU_Op (1, 1)                        │
        call DU_Store (1)                        ┘
10    continue
      call CU_End ()                             ┐
      call AU_End ()                             │ Stop tracing
      call DU_End ()                             ┘
      call Close_Trace ()                        ] Close program trace
      end
```

Figure 10 - A sample source annotation.

The modified source is then compiled and linked with a specially written trace library. This library provides definitions for all of the externally defined procedures in the annotated source. The use of a separate library allows for increased flexibility, since it places all the responsibility for the translation of the abstract information specified by the annotated source into whatever physical representation is required on the trace library. This means that a single annotated source may be linked with different trace libraries to produce different trace formats.

The standard trace library is written in highly optimised C for speed. It generates three trace files (one for each sub-processor) although there is no reason why a trace library could not generate a single trace file and tag each trace element as being for a specific processor. As has already been mentioned, traces can be generated in human-readable format or in a considerably more compact format for simulator use. This, combined with compression, greatly reduces the size of traces. For example, the compressed trace for the first kernel in the Livermore loops manages to compress several thousand control sub-processor operations down to just 91 bytes.

4.3.2 The Simulator Structure

The object-oriented approach of the C++ language provides a very useful framework within which an architectural simulation can be built.

The top level object in the simulator is the system (class `Asim_System`) itself. Each system object contains a fully decoupled processor and memory subsystem. The main method for the system object cycles the entire system for a single cycle. This involves moving data between the processor and memory objects (enforcing the limitations of the interconnect as it does so), as well as the cycling of the processor and memory themselves.

The system object is highly parameterisable. Most of these parameters configure the internal components of the object and will be listed when those are described, but those of interest only to the system object itself largely concern the format of the report that is generated at the end of a simulation, specifically the level of detail in the representation of memory access statistics.

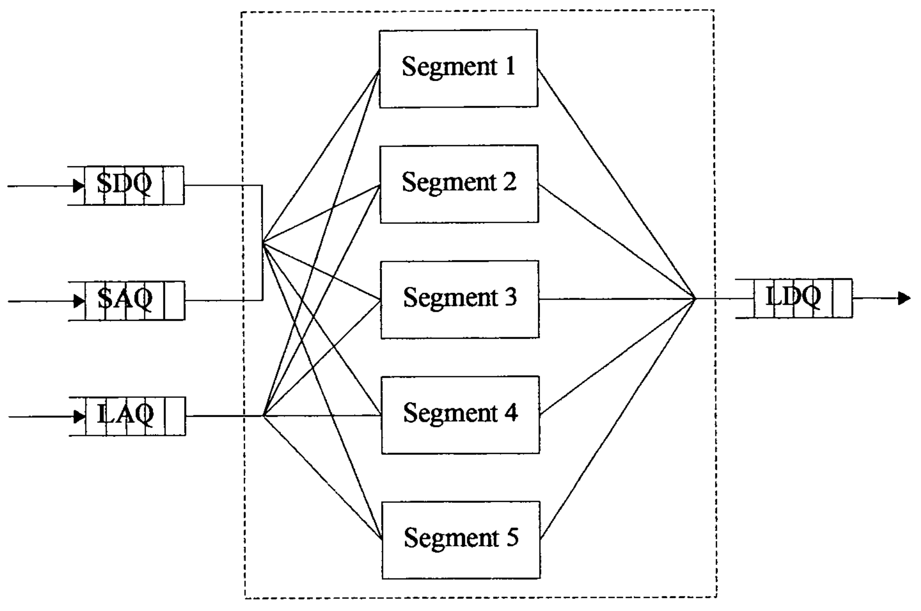


Figure 11 - The simulated memory subsystem structure (5 segments).

Within the system object, we find the memory subsystem (class `Asim_Memory`) and the fully decoupled processor itself (class `Asim_Processor`). The interface to the memory object is straightforward, and provides methods to check the memory subsystem's readiness to produce or accept data, and to add or remove memory accesses. The internal structure of the memory system is invisible during operation, with all internal routine being handled by the object itself. The internal structure can, however, be specified when the memory object is first created. This allows the simulation of a variety of memory subsystems, although for the experiments carried out here a segmented and banked system was used.

Internally, the memory is split into a number of segments, as shown in Figure 11. The number of segments (objects of class `Asim_Segment`) in the subsystem can be parameterised, and is set (arbitrarily) to five in the example shown. There are two inputs to the basic memory subsystem, and these are assumed to be connected to the segments by a $2 \rightarrow n$ crossbar switch. This is implemented directly by the object, and a round-robin discipline is enforced to ensure that should both inputs wish to access the same segment each will get its turn. This becomes more important in the more elaborate variations of the architecture described in Sections 4.3.3.2 and 4.3.3.3, where there may be more than two input sources to the network.

At the other "end" of the memory subsystem the segments are connected to a single output port by an $n \rightarrow 1$ crossbar switch (i.e. a demultiplexer). In the more elaborate variations of the architecture, the outdegree of the crossbar switch may be greater. A round-robin discipline is again enforced to ensure that if more than one segment wishes to output data, each gets its turn. Cycling the memory subsystem object results in memory accesses (if any are present) being moved from segments to output and from inputs to segments, as well as the individual segments themselves being cycled.

Each segment of the memory subsystem (see Figure 12 for the structure of a segment with four banks) consists of an input buffer, an output buffer and a number of memory banks (class `Asim_MemBank`). Segments have only one input and output regardless of the number of banks they contain. New memory accesses are first placed in the input buffer. If the bank to which the access is to be routed is busy the segment input becomes blocked, otherwise the access is routed directly to the bank. It is possible to specify a delay in transferring the value from buffer to bank to allow for the simulation of transmission delays or address/data line setup times. As data becomes available from a bank, it is transferred to the segment output buffer (assuming that this is not occupied). If the output buffer is occupied, the bank will stall until the transfer can be accomplished. It is also possible to specify a delay for the transfer from bank to output buffer.

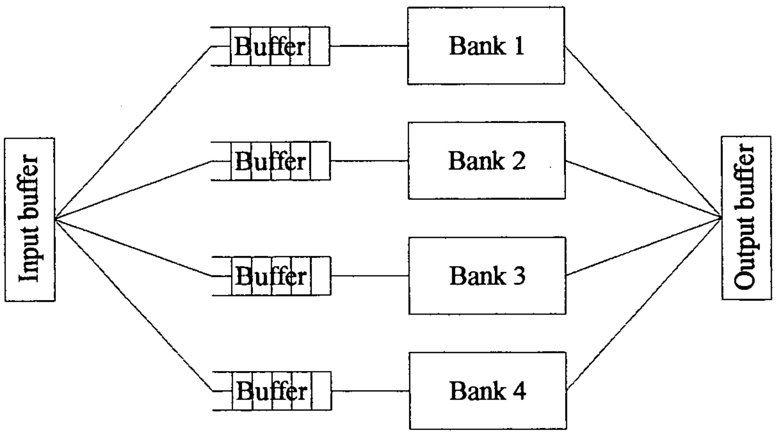


Figure 12 - The simulated memory segment structure (4 banks).

Methods provided for each segment include the ability to inquire whether a segment has output data waiting, whether it can accept a new access, the addition and removal of data from input and input and the ability to cycle the segment. Cycling a segment object causes various internal data transfers to take place, as well as cycling the banks within the segment.

The banks themselves are uncomplicated, consisting only of a small input queue and the memory bank itself. Each bank collects statistical information regarding the utilisation of its input queue and the latency of all store accesses that it receives. In the more elaborate architectural variations, where stores may come from any one of several sources, it is possible to record the latency of each type of store or gather statistical information regarding all types of store combined. Statistics regarding load latencies are gathered at the end of the load path rather than in the banks.

If it is not required that per-bank latency and bank queue utilisation figures be gathered, segments can be instructed to gather and assimilate data from all their constituent banks. If even this is unnecessarily detailed, average latency and queue utilisation figures for the memory subsystem as a whole can be produced.

Returning once again to the system level, the second major component of this object is the processor itself, which is of class `Asim_Processor`. This contains a number of components, namely the three sub-processors that make up a fully decoupled processor, plus the fetch engines used to break meta-instructions down into their constituent instructions. Methods are provided for examining whether a specific sub-processor wishes to access memory, removing a memory access from a specific sub-processor to deliver to memory, and delivering the result of a load access to the appropriate destination.

The processor differs from most other objects in the simulator by having two cycle methods. This is required to ensure correct timing and is brought about by the fact that some components of the processor must be cycled before the memory object and some afterwards.

The classes of the three sub-processor objects contained within the processor differ considerably, but all three are derived from the same parent class (class `Asim_SubProcessor`). This parent class provides a number of features common to all three sub-processors, such as a processor pipeline and an assortment of statistical variables used to keep track of cycles completed, the number of stalls and the frequency with which each type of operation has been executed.

The first derived class represents the control sub-processor. This inherits the objects described above from its parent class and adds objects to represent the MIQ and CFQ, as well as additional architectural structures required by the higher level architectures which will be described later. Copious statistics are collected to measure pipeline utilisation, MIQ and CFQ lengths, stalls due to CFQ waits, and so forth. The usual retinue of input/output methods are provided, allowing memory accesses to be transferred to/from the sub-processor in architectural variations which allow this. The transfer of meta-instructions between the MIQ and the fetch engine is *not* modelled in the usual manner, however, as the fetch engine object is allowed direct access to the MIQ and requires no externally accessible method to do this.

Most of the actions performed during the control sub-processor's cycle method are common to all three sub-processor designs. The pipeline is cycled and the operation "expelled" from the end is recorded in the table of instruction execution frequencies (used to determine the dynamic instruction mix). Then, if the pipeline is not stalled, a new operation is fetched and added to the pipeline. The control sub-processor is assumed to be directly connected to a perfect instruction cache.

Each pipeline stage, from the first unstalled stage to the last, is then examined and one of a number of actions taken depending upon the operation and the stage it appears in. Operations may stall the pipeline. Once all the unstalled pipeline stages have been "run", various internal statistics are gathered, such as pipeline utilisation and number of cycles spent stalled.

Operation	Description
OP	Generic CU arithmetic/logical operation.
GUARD OP	Guarded CU operation.
WAIT DU	Receive half of CU to DU transfer.
WAIT AU	Receive half of AU to CU transfer.
ISSUE	Issue meta-instruction.
ISSUE AU	Issue AU-only meta-instruction.
ISSUE DU	Issue DU-only meta-instruction.
JMP	Unconditional jump.
BSR	Branch to subroutine.
RTS	Return from subroutine.
BRA	Conditional branch.
BRA AU	Conditional branch on data from AU to CU transfer.
BRA DU	Conditional branch on data from DU to CU transfer.

Table 1 - Level 1 control sub-processor operations.

The operations which may be executed on the control sub-processor (at Architecture Level 1, at least) are shown in Table 1. As has already been explained, the simulator does not simulate the execution of a particular processor instruction set but rather runs a simplified, abstract instruction set that groups together instructions unimportant to the issues being investigated and distinguishes only between operations relevant to decoupling. All of the irrelevant operations are classified as one type of instruction, **OP**, and act as little more than spacers, to ensure that the correct amount of time elapses between "interesting" operations. This instruction type encompasses all arithmetic (integer and floating point) and logical operations. The control sub-processor assumes that one of these operations can be issued every cycle (pipeline stalls permitting) so inter-instruction dependencies are not explicitly modelled. However, as we will see when we examine the modelling of the fetch engine, the effect of stalls due to inter-instruction dependencies can be simulated in other ways.

The other operations supported are relatively self-explanatory. The **GUARD OP** operation is functionally identical to **OP** (that is, it does nothing) but allows instructions which are not executed due to guarding to be distinguished from others. A selection of deliberately vague operations are provided for the modelling of meta-instruction issue and the return of data from the work sub-processors. Standard control flow operations are also provided, although in the current simulator these are considered functionally equivalent to **OP** and do not actually cause any special action to be taken.

The address sub-processor differs in a number of ways. Methods are provided for handling memory accesses, and the internal structure of the object reflects its different role, with queue objects to simulate the LAQ, SAQ and TRQ. The rather minimalist instruction set supported by the `Asim_AddrSubProcessor` class is show in Table 2. Perhaps the most important difference in the address sub-processor's operation, however, is that it takes new instructions not from the implicit perfect instruction cache used by the control sub-processor, but from a dedicated fetch engine, which may cause it to stall if no instructions are available.

Operation	Description
OP	Generic AU arithmetic/logical operation.
GUARD OP	Guarded AU operation.
DC LD	Addressing half of AU/DU decoupled load.
DC ST	Addressing half of AU/DU decoupled store.
WAIT DU	Receive half of AU/DU transfer.
SIGNAL CU	Send half of AU/CU transfer.
SIGNAL DU	Send half of AU/DU transfer.

Table 2 - Level 1 address sub-processor operations.

The data sub-processor works in a similar manner, and the instruction set supported by the simulated DU is shown in Table 3.

Operation	Description
OP	Generic DU arithmetic/logical operation.
GUARD OP	Guarded DU operation.
LD	Data half of AU/DU decoupled load.
ST	Data half of AU/DU decoupled store.
WAIT AU	Receive half of AU/DU transfer.
SIGNAL CU	Send half of DU/CU transfer.
SIGNAL AU	Send half of DU/AU transfer.

Table 3 - Level 1 data sub-processor operations.

As has already been stated, each of the work sub-processor components contains a fetch engine object. This represents that part of the real hardware that would be responsible for translating meta-instructions from the MIQ into a stream of sub-processor instructions. In principle the simulation of these objects could include the simulation of some of the interesting instruction caching techniques that will be described in Section 8.2, but in the current simulation it is assumed that the fetch engines are connected to more of those magical perfect instruction caches that feed the control sub-processor.

The fetch engine object simulates this by reading an instruction trace for the sub-processor from a file. Details of the compression techniques used to compact the traces down to a manageable size were given in Section 4.3.1 but for the moment all that needs to be revealed is that the fetch engine reads tuples consisting of an operation, the number of times that operation is to be repeated, and a minimum cycle count. The last field of this tuple is generally set to be the same as the repetition count (implying that an instruction can be produced every cycle) but it is possible to manually alter the trace to increase this value and thus produce an approximation of the pipeline delays that might result from inter-instruction dependencies.

The stream of decompressed instructions that the fetch engine reads consists almost entirely of instructions (as specified in Table 2 and Table 3) for one of the work sub-processors which are passed to the appropriate object on request. However, the instruction stream can also contain two additional instructions, both of which are filtered from the instruction stream that reaches the work sub-processor.

The first of these is the **WAIT CU** operation, which is (implicitly) executed on the fetch engine itself. When the fetch engine encounters one of these in the instruction stream, it attempts to remove a meta-instruction from the MIQ. If none is available, the fetch engine stalls until one arrives.

This filter mechanism allows the implicit synchronisation between the AU or DU and the CU that occurs when a meta-instruction is removed from the MIQ to be modelled. It effectively inserts into the work sub-processors instruction stream the synchronisations with the CU that result from control decoupling, without requiring that they be explicitly modelled on the work sub-processors.

The second additional instruction is the **END** operation. The fetch engine, on receiving one of these, knows that the end of the instruction trace has been reached and “shuts down”. The operation is passed to the appropriate work sub-processor but is not considered a true instruction, since its only function is to halt the simulation when it reaches the end of the processor pipeline.

Methods provided for the fetch engine include checks to see if it is stalled or finished, to read the next available instruction and to cycle it.

4.3.3 The Simulated Architecture

To provide a consistent base on which all experimental work could be based, a family of upwardly-compatible abstract architectures was first defined.

Three levels of architecture were defined. The Level 1 architecture is the simple baseline architecture and later levels increase the richness and complexity of the processor design. All three levels of architecture are assumed to use the same basic processing units, with minimal instruction set enhancements.

4.3.3.1 The Level 1 Architecture

The simplest of the three simulated architectures is the Level 1 architecture, which is illustrated in Figure 13. This was originally intended to correspond as closely as possible to the semantics-driven architecture already described [Bird91] but certain weaknesses in that architecture's capabilities lead to Level 1 having a rather richer degree of interconnection than was originally intended.

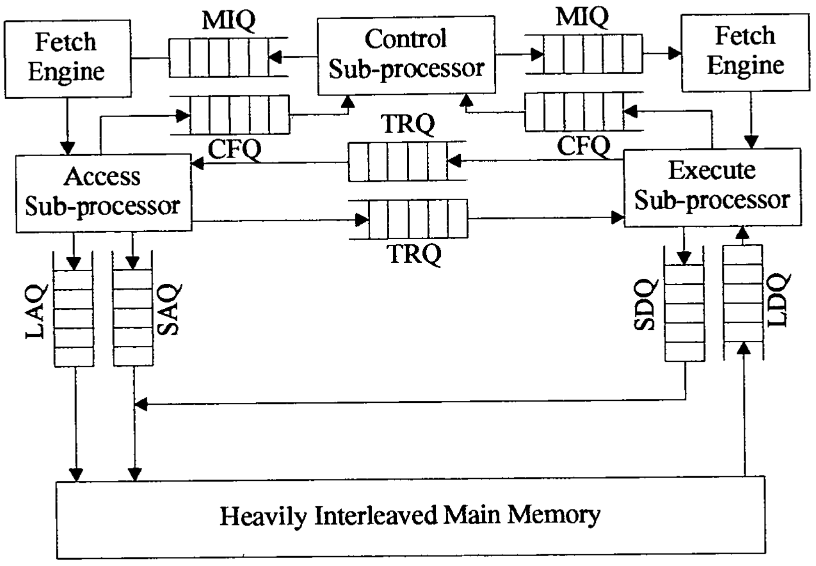


Figure 13 - Level 1 architecture.

In this architecture the various sub-processors are minimally connected. The control processor does no computation itself and restricts its activities to performing the traversal of the program control flow graph. It is able to pass meta-instructions to the work units and can receive the results of computations back from either work unit. It is incapable of accessing memory itself, and if it requires to do this it must do so indirectly via the work units. This adheres closely to the purist view of the control processor, moving from node to node in the flow graph and performing no explicit computations other than the examination of the results of branch condition evaluations.

In Bird's original semantics-driven architecture, there was no connectivity between the two work units. However, this made indirection difficult and required that such accesses be performed by the control processor, which moved away from the purist view of the control processor's role. In the Level 1 architecture, the control unit is unable to make such accesses, but a data path exists between the two work processors, allowing indirected values to be passed directly from the DU to the AU.

To give an example of why this is necessary, the Fortran expression `x = a(b(i))` would prove awkward to evaluate on Bird's architecture. The address processor could generate the address `b(i)` but we would then find the contents of this array element, which we wish to use as an index for our access of the array `a` stranded on the DU. This problem could be resolved by performing two non-decoupled memory accesses on the CU, or possibly by performing a single non-decoupled load on the CU and then somehow passing this value to the AU if the meta-instruction format allowed the passing of parameters. This loses any control decoupling that may have arisen, however, and leaves the work processors idle. If indirections of this sort were infrequent, performance would not be noticeably effected, but such indirections are typical of many sparse algorithms, such as the particle in cell benchmarks in the Livermore loops, as well as in such menial tasks as list following.

Indirection on the Level 1 architecture can be achieved by performing a standard decoupled load, then passing the loaded operand back through the transfer queue to the AU. This eliminates the loss of control decoupling but loses access/execute decoupling. Compiler optimisations could reduce the frequency of the loss of decoupling by splitting a series of indirect loads into a series of loads followed by a series of transfers followed by a second series of loads, but this would introduce the risk of deadlock unless the state (and length) of the queues used for the DU to AU transfer were known beforehand. A more robust approach to the problem of indirection appears in the definition of the Level 2 architecture.

The Level 1 architecture is not intended as a serious proposal for a real computer architecture, but rather as a baseline from which the parameters which dictate the performance of a fully decoupled machine can be deduced, their effects analysed and comparisons made with more complex architectures. The simplicity of the Level 1 machine also makes the development of a model of a fully decoupled architecture more straightforward.

4.3.3.2 The Level 2 Architecture

The second level architecture differs from Level 1 chiefly in having more paths to memory. As can be seen in Figure 14 the address and control sub-processors both now have direct non-decoupled connections to memory.

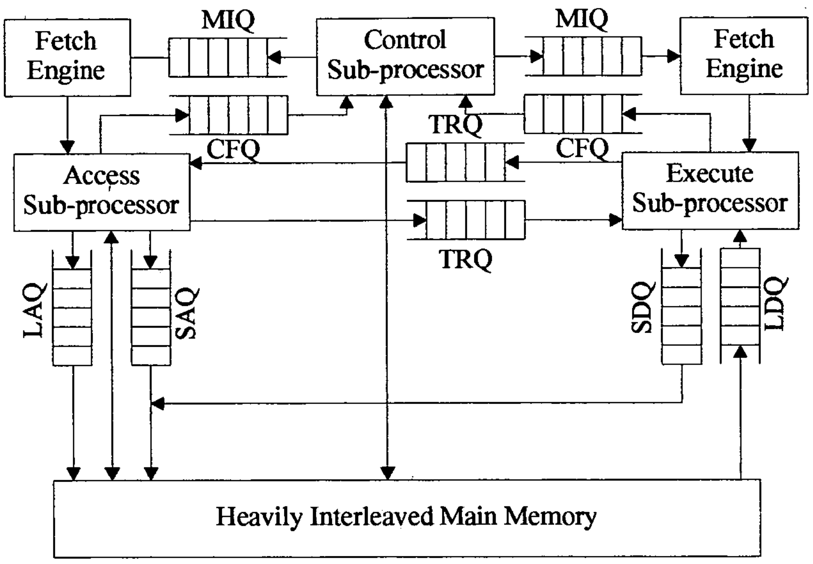


Figure 14 - Level 2 architecture.

This is intended to allow these sub-processors to access memory without forcing any losses of decoupling. The effective latency of these memory paths is high, since neither is decoupled, but if use of direct accesses of this sort is infrequent the performance penalty is considerably lower than the cost of repeated losses of decoupling. Additionally, the latency of these accesses might be reduced by conventional latency reduction techniques, such as the use of data caches, although this would introduce cache consistency difficulties.

Operation	Description
SF LD	CU/AU self load.
SF ST	CU/AU self store.

Table 4 - Level 2 address and control sub-processor operations.

The instruction set extensions in the Level 2 architecture are the same for both the address and control sub-processors and are shown in Table 4. Both can now perform conventional non-decoupled loads and stores. This effects some other aspects of the simulated architecture, however. In addition to the minor changes in the instruction set "decoding" the sub-processor components are also equipped with single element data and address buffers which are connected to the memory subsystem in a similar manner to the queues used in decoupled accesses. The system object is modified so that memory accesses can be generated by four different sources and loads can return to one of three different destinations.

All three types of load (CU self load, AU self load and AU/DU decoupled load) are assigned identical priority, and a round-robin mechanism is used where multiple input sources wish to address the same bank. No such bottleneck occurs with returning loads, since only one self load of each type can be active in memory at any one time, due to the stalling nature of the self load operation.

4.3.3.3 The Level 3 Architecture

The third and final level of the architecture can be seen in Figure 15. It enriches the processor-memory interconnection still further, by providing a means whereby the control sub-processor can initiate decoupled loads for the address sub-processor. How useful this will be in practice remains to be seen. This new memory path is supported by the addition of a **DC LD** operation on the CU and a **LD** operation on the AU (see Table 5). These behave identically to the equivalent operations for AU/DU decoupled loads.

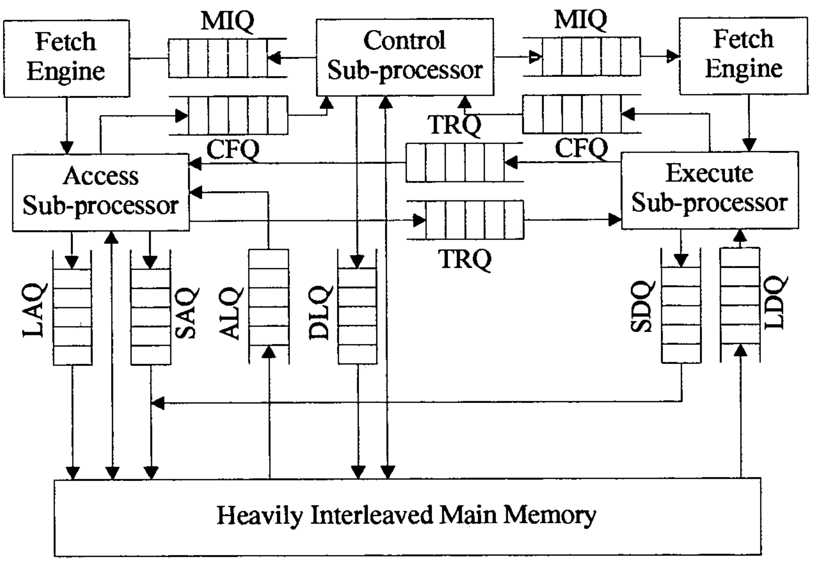


Figure 15 - Level 3 architecture.

The Level 3 architecture also considerably extends those parts of the CU instruction set concerned with meta-instruction issue. The new instructions are listed in Table 6 and provide two new means of meta-instruction issue, guarded and conditional. The semantics of these operations will be explained in Chapter 5.

Operation	Description
LD	Data half of CU/AU decoupled load.

Table 5 - Level 3 address sub-processor operations.

The additional features of the Level 3 architecture require a number of new sub-components for the address and control sub-processors. The control sub-processor requires a queue for the address half of CU/AU decoupled loads. This is known as the DLQ to distinguish it from the LDQ. The additional meta-instruction issues operations require no additional software components (although they would require considerable control logic in hardware) and are handled purely by changes in the instruction decoding.

Operation	Description
DC LD	Addressing half of CU/AU decoupled load.
COND ISSUE	Conditional meta-instruction issue.
COND ISSUE AU	Conditional AU-only meta-instruction issue.
COND ISSUE DU	Conditional DU-only meta-instruction issue.
GUARD ISSUE	Guarded meta-instruction issue.
GUARD ISSUE AU	Guarded AU-only meta-instruction issue.
GUARD ISSUE DU	Guarded DU-only meta-instruction issue.

Table 6 - Level 3 control sub-processor operations.

5. Decoupling

In Section 2.4.6 a brief introduction to the topic of decoupling was given. In this chapter decoupling will be discussed in greater detail, examining both of the forms of decoupling that occur in a fully decoupled architecture. The architectural and programming language features that affect the available degrees of decoupling will be examined in detail, showing both those features that cause decoupling in the first place and those that place limits on the extent to which it can be exploited.

For each type of decoupling, a series of experiments will be run on the simulated queueing network model described in the previous chapter. In parallel with this, a second set of experiments will be run using the full architectural simulator described in Section 4.3, and the results thus obtained compared with those from the queueing network simulation. From these results the performance envelope of the architecture can be mapped, and the validity of the analytical model established.

5.1 Access/Execute Decoupling

Access/execute decoupling is the more thoroughly understood of the two forms of decoupling. Some of the ideas behind it have been in use since the 1960s and a number of machines have been built which make use of it.

5.1.1 Experiments 1 & 2: Load streaming

For the purposes of the first two experiments, an artificial instruction trace was created by hand-annotation that represented a sequence of 100,000 decoupled loads. This figure was felt to be high enough to allow the system to stabilise in a steady state while also being low enough for simulation not to require an excessive amount of time.

The artificial trace was as minimal as possible, and designed only to test the characteristics of decoupled loads without involving any other features of the architecture which might skew the results obtained. The entire sequence of 100,000 loads was issued by a single meta-instruction on the control unit, and the other two units were configured so that, stalls within the system notwithstanding, one load packet would be generated on each cycle and one load packet consumed.

The memory subsystem was configured in the architectural simulator as sixteen segments of four banks each, and in the simulated model as 64 separate banks.

Two experiments were run. The first (which we will henceforth refer to as Experiment 1) varied two parameters, the LAQ/LDQ size and the memory latency. The former was set in turn to 1, 2, 4, 8 and 16 packets, then in increments of 16 up to 128 packets. The memory latency was varied between 8 and 128 processor clock cycles, in increments of 8 cycles. Memory access time was set to be 80% of memory latency in each case.

It seemed likely that, between them, these two parameters would allow the performance envelope of load streaming to be fully mapped. The range of memory latencies simulated was chosen so that it spanned a wide range of memory system characteristics, from having only half the required bandwidth for peak performance to having twice the required bandwidth. The selection of queue lengths chosen allowed the modelling of both "undertagged" systems, where the number of active load packets is less than the maximum the memory system can cope with, and systems where the number of active load packets can potentially fill the memory pipeline.

In each case, the processor was simulated both with and without address remapping. On the simulated model, this was modelled by the use of uniform random bank selection and unit stride bank assignment respectively.

The second experiment (Experiment 2) conducted was essentially the same as the first but held the length of the LAQ at one packet while varying the LDQ length, allowing the behaviour of the processor with and without an LAQ to be compared. The hypothesis of the experiment was that while the presence of an LAQ might have a minor effect on the performance of the address sub-processor, its absence would not adversely change the effective latency. Since the LAQ is a potentially expensive piece of hardware, it would seem desirable to eliminate it or even to reduce its size.

5.1.2 Unit stride accesses

The experiment was first run using memory accesses with unit stride. In the case of the simulated model this was achieved by using the provided command line option to force load packets to be routed to the memory banks in this manner. The trace used meant that the architectural simulator used unit stride by default, unless Rau's address hashing was used.

Running the experiment yielded some interesting results although in some cases they revealed more about the weaknesses of the model than the behaviour of the architecture.

Perhaps the most obvious measure of performance is the total runtime for the test sequence. For this it suffices to use the runtime for the data sub-processor since, in the example trace, this is guaranteed to be the last active sub-processor. It was found that the model and simulator produced similar results for runtime when the system was undertagged, but that in situations where the LAQ/LDQ length are greater than the memory latency, it tended to overestimate. This is shown diagrammatically in Figure 16, which illustrates the model error, relative to the runtime for the

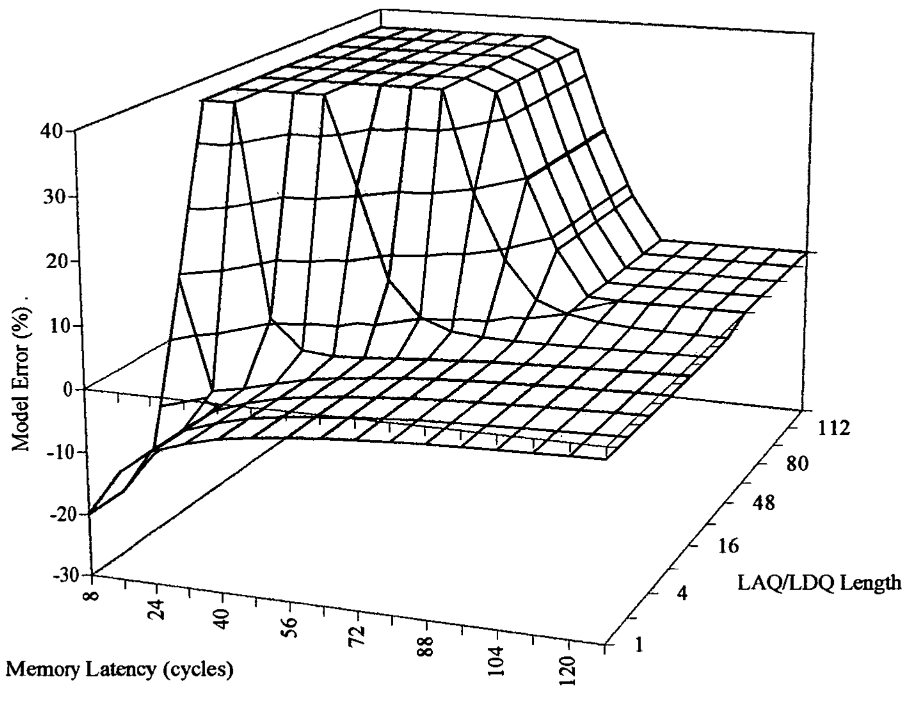


Figure 16 - Normalised model runtime error. (Ex. 1)

architectural simulator, for each combination of LAQ/LDQ length and memory latency. The error is due largely to small variations in the production/consumption rate on the AU and DU respectively which, combined with rounding errors, resulted in loads occasionally being inserted very two cycles rather than every cycle. Where the system was undertagged this had little effect since it was small compared to the effects of the undertagging, and when latency was high it was also too small to noticeably change the runtime. However, in those parts of the performance envelope where sufficient load tags were available, and memory latency was relatively short, it was found to introduce a considerable error.

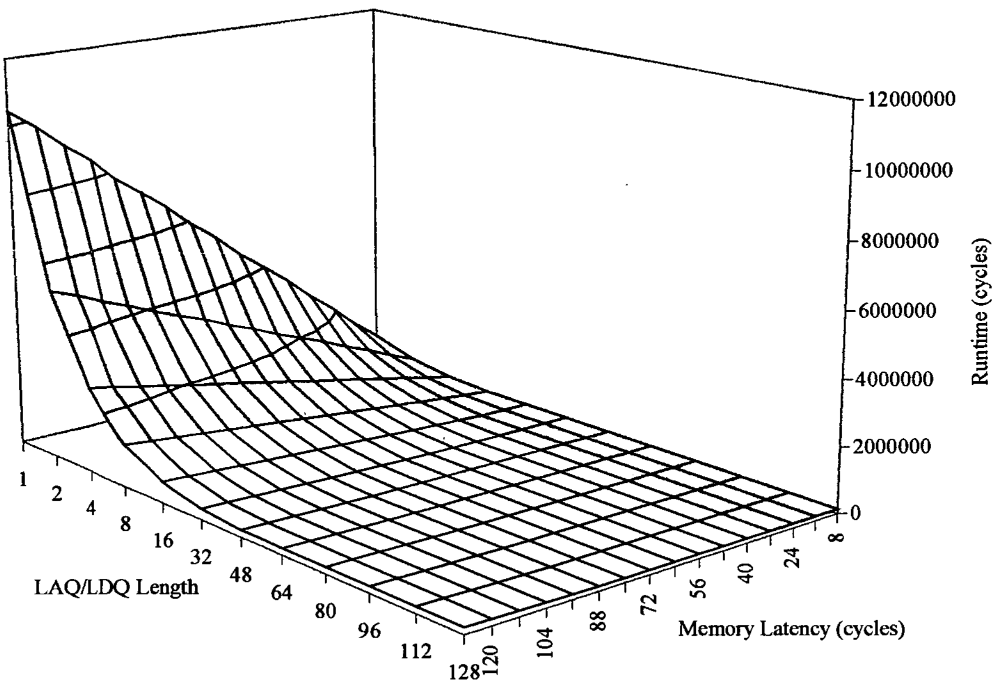


Figure 17 - Total runtime. (Ex. 1)

Figure 17 shows the simulated runtime of the test sequence for the range of memory latencies and queue lengths described above. The results indicate that, when the LAQ and LDQ are short, overall performance is strongly influenced by memory latency. This can be explained by the LDQ size limiting the number of active load packets that may be in the memory subsystem at any one time. If the LDQ is only a single element long, no more than one load packet may be active at any one time. This means that only a single bank will be busy at any time. Examining the "back" of the graph shows the runtime varying linearly with memory latency, as load packets pass through memory serially.

As the lengths of the queues increase, so does performance. More load packets may be active at any one time and this results in a rapid drop in runtime since more of the available memory bandwidth is being used. As can be seen in Figure 17 an increase in queue length reduces the rate at which runtime increases when the memory latency is increased. In other words, as the queues become longer the system becomes more latency tolerant.

These runtime estimates do not, in themselves, give a particularly good indication of the performance benefits of decoupling. For this reason, the runtime of the decoupled machine was compared with estimated runtimes for two conventional architectures which act as reasonable upper and lower performance bounds for conventional non-decoupled architectures. The first of these was a non-decoupled architecture with no cache. In this system loads are strictly sequential, regardless of how heavily banked the memory subsystem is. The performance for this architecture was found to be similar to that for the decoupled architecture (as shown in Figure 17) with an LDQ length of 1, which restricts the degree to which the available memory bandwidth can be used. It is reasonable to say that in this experiment the worst case performance of a decoupled architecture is comparable to that of a conventional processor without a cache.

The other architecture examine was a conventional architecture with a data cache. The cache was assumed to have a block size of 64 words. This would be unrealistic in practice - shorter block sizes are usually considered superior for the relatively non-linear access patterns of data accesses. However, on a system with 64 banks (i.e. the simulated system) which is running a program with strictly unit stride accesses, this block size performs particularly well, with a hit rate of 98.5%.

It was found that a processor with this near-optimal cache is clearly superior to a decoupled processor when the decoupled processor has a queue length of less than approximately 32 packets. This represents the point at which the penalties due to limitations imposed upon memory bandwidth utilisation by the LDQ size are comparable to those caused by cache misses on the conventional architecture. For longer queues, and memory cycle times up to 32 cycles, caching remains superior (although only marginally so). This is due to the cache miss penalty being less than the additional memory latency incurred by the traversal of the LAQ and LDQ.

However, outside those areas, decoupling is found to outperform even an unrealistically optimal cache (albeit only marginally). The small difference is due to the cache seeing a full memory access on each cache miss (i.e. every 64th access incurs a full memory latency) whereas the decoupled architecture sees the full latency on only the first access in the stream. The runtime for both architectures increases once the system became underbanked.

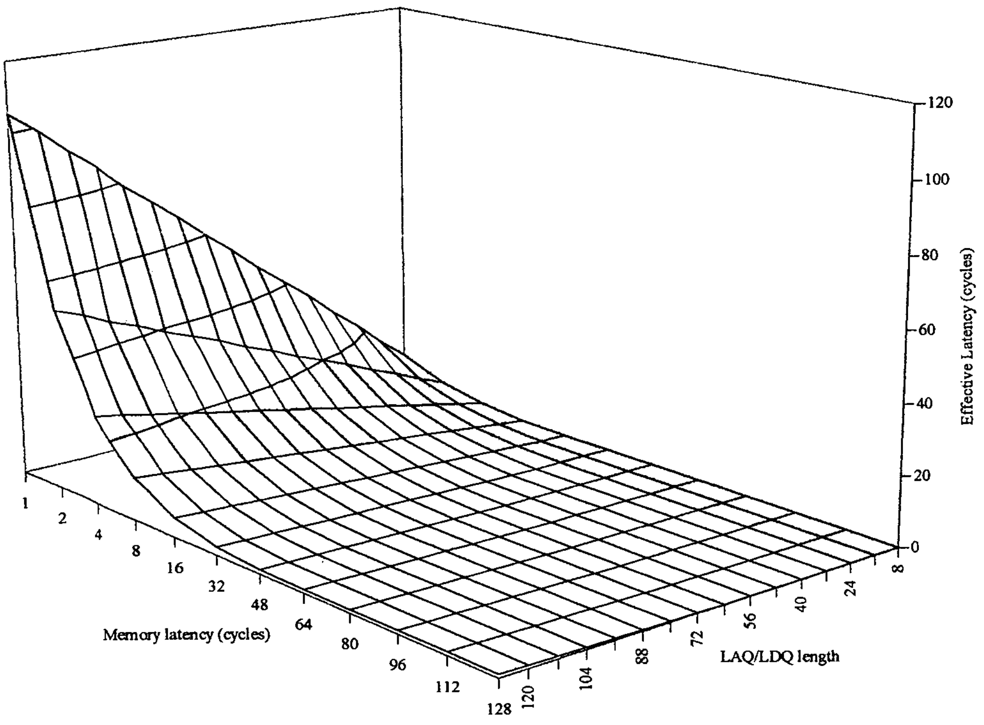


Figure 18 - Effective load latency with unit stride (Ex. 1)

Moving on to other aspects of system behaviour, the main influence on runtime in this experiment is the effective memory latency perceived by the data sub-processor. Regardless of the actual memory latency, it is the effective latency perceived by this unit that determines performance. Figure 18 shows the effective latency seen by the data unit in this experiment. The graph shows the behaviour of the architectural simulator, although the behaviour of the simulated model was affected in a similar manner to the runtime, and showed inaccuracies when actual latency was low and LAQ/LDQ length was high. A very strong correlation between effective latency and total runtime is apparent, with the graph being a replica of Figure 17 (albeit with a different scale).

Of less immediate interest but still worthy of mention is the round-trip memory latency. This is shown in Figure 19. Two features dominate - the relatively-flat plain on the left of the graph and the steep "hill" on the right.

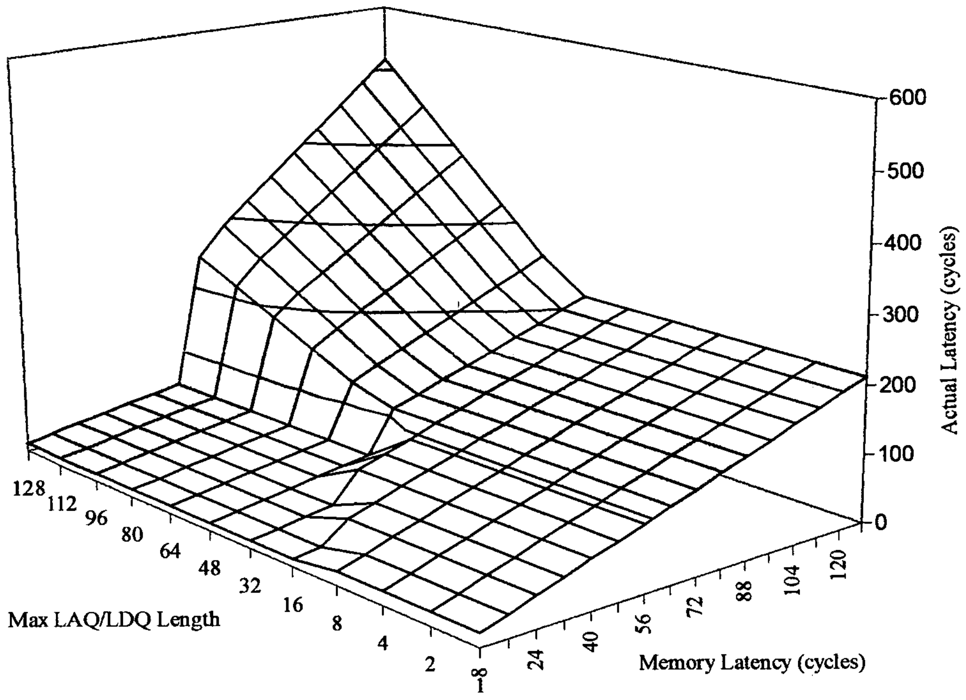


Figure 19 - Actual load latency with unit stride. (Ex. 1)

The former indicates those areas of the performance envelope where the system has become decoupled to some degree. The two characteristics which influence this are the memory latency (which must be sufficiently small that the memory subsystem is not underbanked) and the LDQ length. A short LDQ limits the number of active load packets and thus limits the use of the total memory bandwidth that is available. This is illustrated by the raised area at the front of the graph, which flattens out as LDQ length increases and bandwidth can be more fully utilised.

As memory latency increases, the LDQ length required for the system to become decoupled increases. Once memory latency is such that the system becomes underbanked and is incapable of delivering one load datum per cycle in even the best of circumstances, latency increases rapidly as the memory subsystem clogs up. One feature of the "hill" caused by this is that once the system is underbanked, increasing the size of the LAQ also increases the latency. This is due to the memory subsystem having become a bottleneck, which results in the LAQ filling up. This substantially increases the time a load spends waiting in the LAQ, which in turn increases the round-trip latency. Experiment 2 investigates this result further, in an effort to demonstrate the redundancy of the LAQ.

Looking at the behaviour of the queues themselves during the experiment produces some interesting results, as well as showing at least one flaw in the model. Figure 20 shows the average length of the LAQ during the simulation. The LAQ behaves much as expected and it should be noted that the shape of this graph show some resemblance to that describing the actual load latency (Figure 19).

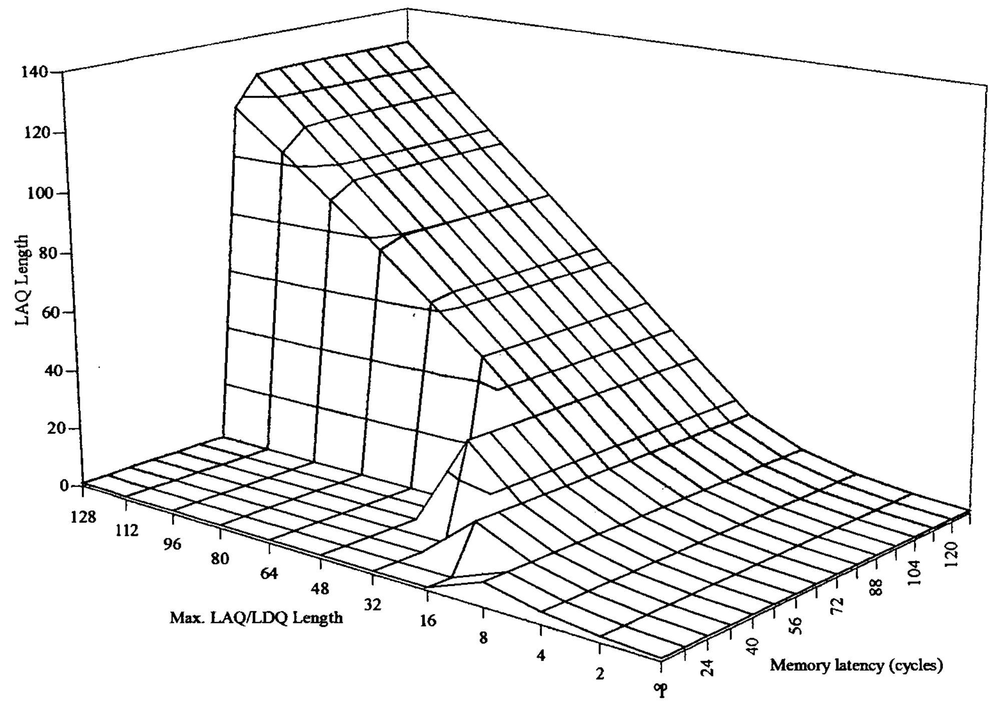


Figure 20 - Average LAQ length with unit stride for simulator. (Ex. 1)

The flat plain on the left represents an area where the memory system is overbanked and capable of accepting new loads as soon as they arrive in the LAQ. This means that the LAQ never fills, although the "foothills" at the front of graph are due (as described earlier) to the limitations imposed by the low LDQ size. The peak at the right occurs when the memory subsystem becomes underbanked, and its height corresponds to the maximum queue length as the queue is full in this area of the performance envelope.

Making a similar measurement of average length for the LDQ reveals some rather less predictable results. For example, the LDQ *never* fills in the simulator, since data always arrives in order and never arrives faster than the data sub-processor can remove it. However, on the simulated model, we can see in Figure 21 that the graph here is not flat and has a number of pronounced peaks, plus a clear ridge when the memory latency is around 70 cycles, with a raised plain beyond.

This unusual behaviour is due to one of the simplifications apparent in the simulated model, namely the restriction on the number of load packets which may be transferred from the memory subsystem

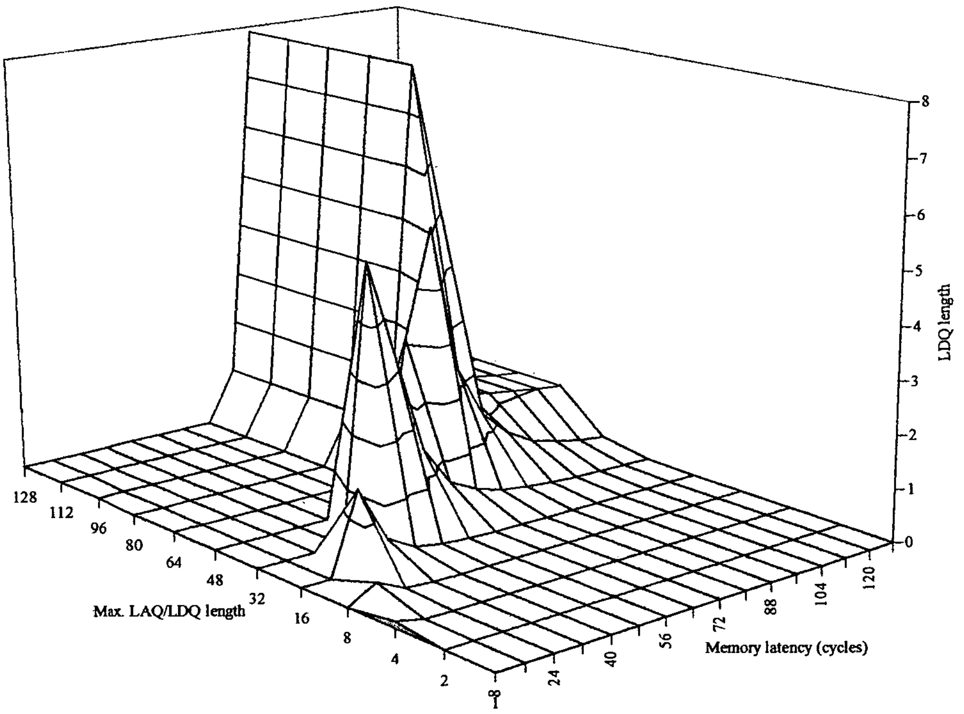


Figure 21 - Average LDQ length with unit stride for model. (Ex. 1)

into the LDQ at a time. The architectural simulator allows only a single load packet to be transferred per cycle, which ensures that loads cannot arrive faster than the data sub-processor can remove them (although this is not the reason for the restriction). This, when coupled with unit stride memory accesses in Experiment 1 and the nature of the round-robin bank selection scheme used to resolve conflicts when more than one memory segment wishes to transfer a load packet, ensures that load packets never arrive out of order and always arrive singly.

The simulated model allows as many load packets as are ready to be transferred in a single cycle. Typically this figure never exceeds two or three, but the cumulative effect of several packets being transferred every cycle for several cycles can result in the queue building up. The ridge on Figure 21 is the result of this behaviour, and the flat plain beyond is due to the same phenomenon manifesting itself to a lesser degree.

5.1.3 Random accesses

The same experiment, when repeated with random rather than unit stride accesses, also provides some interesting results. In some areas the measured performance is virtually identical to that obtained with unit stride accesses. For example, the total runtime matches that of the unit stride experiment very closely, so closely in fact that there is no discernible difference visible on a graph. Figure 22 shows the difference in runtime for the experiment when run with random and unit stride accesses. On an example where runtimes can exceed 1,000,000 clock cycles, the greatest difference is 20 cycles. Needless to say, the difference in effective latency is equally tiny, and was undetectable unless this metric was measured to four decimal places.

Small differences begin to emerge when we examine the actual latency when random accesses are used. When we compare the actual latency for the simulator when using random accesses with the same value when unit stride accesses are used, we find very little difference at all.

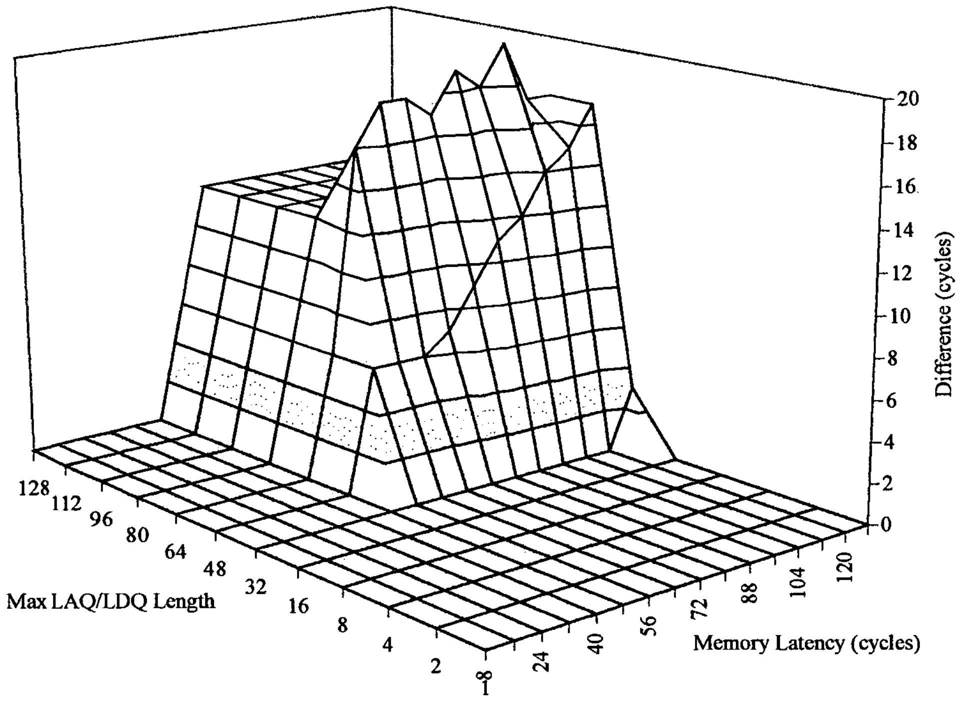


Figure 22 - Difference in runtime between random and unit stride accesses. (Ex. 1)

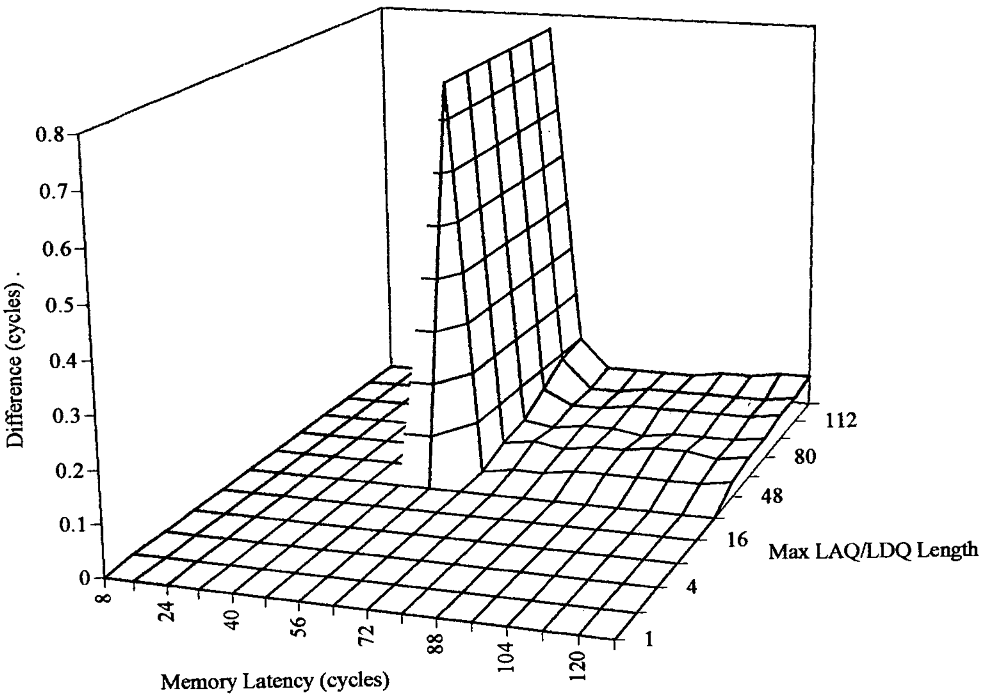


Figure 23 - Difference in actual latency for unit stride and random accesses. (Ex. 1)

The difference between the two is shown in Figure 23. However, this difference is on the order of a fraction of a cycle in the worst case (which is, interestingly, at the point where the system becomes underbanked). This indicates one of the good features of the address remapping scheme used - while the effect of bad strides is greatly reduced, there is little effect on good strides. In this case, the *distribution* of the bank utilisation remains uniform, only the order in which the banks are accessed is permuted. Thus there is still a flat area in the overbanked, overtagged area of the graph, since all that has changed is the order in which the banks are accessed.

This is not, however, true for the simulated model. Although load packets are routed to banks using a random uniform distribution, the access pattern is truly random rather than just a permutation and the distribution is guaranteed to be uniform only in the long term. Local irregularities in the access pattern can result in several load packets in a row going to the same bank. This, as can be clearly seen in Figure 24, causes the flat plain to disappear since it is now possible for the LAQ to fill due to localised non-uniformity in the access pattern even when the memory system is overbanked. Note also the general correspondence in shape between Figure 19 and Figure 24 for the area where the

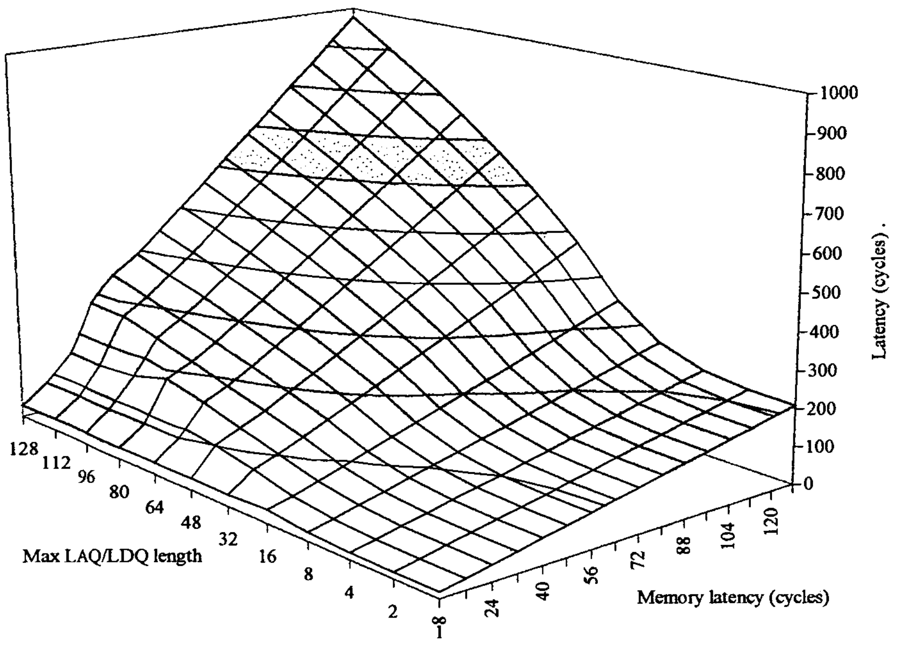


Figure 24 - Actual latency for model with random stride. (Ex. 1)

system is underbanked. In this part of the design space the two correspond quite closely, indicating that the effects of underbanking predominate over the irregularities caused by random bank accesses.

5.1.4 Conclusions

This experiment allows a number of useful conclusions to be drawn about both the experimental methodology used and decoupled architectures themselves.

From a simulation and modelling point of view, the experiment was of interest for two main reasons. It allowed a comparison of the behaviour of a detailed architectural simulator with that of a much simpler simulated model, and it allowed the comparison of unit stride access patterns with random access patterns. The latter may seem like an architectural issue, but if it can be shown that the random access patterns and unit stride access patterns give comparable performance figures, then the more general random access pattern can be used instead of a specific deterministic one.

As can be seen in the copious graphs which accompany this experiment, the simulated model appears to track the behaviour of the architectural simulator very closely in the large. The estimates produced by the model for runtime, effective and actual latency are extremely accurate. However, the correspondence between the two breaks down somewhat when the internal characteristics of the system are examined. While the modelled LAQ length accurately reflects the behaviour of the "real" system, the simplifications made in the model mean that the LDQ length is no longer accurately represented. This inaccuracy is acceptable if all that is required are measurements of performance for the system as a whole (for example, runtime) rather than specific internal measurements.

The comparison between random and deterministic unit stride accesses yields similar results. The use of random access patterns has little effect on measurements such as runtime and effective latency, but the behaviour of the queues differs considerably, since the randomness of the memory access pattern blurs the distinction between areas of the performance envelope that are distinct when deterministic access patterns are used. Again, it appears that the random model can be used, unless details of internal behaviour are required, when a more deterministic model must be used.

Even as simple an experiment as this allows us to observe some very important behaviours. It is clear that for the best performance the LDQ length must be sufficient that the available memory bandwidth can be fully exploited. If the LDQ is too short, the limit that this places on the number of load packets in the memory subsystem means that banks may become idle waiting for a packet to be removed from the LDQ and its associated tag freed. Many of the graphs shown here also exhibit a very clear boundary between the behaviour of the system when it is underbanked and when it is overbanked. As would be expected, the system performs better when there is excess memory bandwidth available. However, one notable quirk in the system's behaviour centres around the length of the LAQ. It appears to be the case that while a longer LAQ will generally result in no change (for better or worse) in system performance, it can substantially increase the actual latency of memory accesses. Before proceeding to investigate the behaviour of decoupled store operations, it is worth investigating this characteristic of the LAQ more closely.

5.1.5 Experiment 2: Is the LAQ really necessary?

This second experiment is constructed in a similar manner to Experiment 1. The simulator and model both perform a number of simulation runs, with each run containing 100,000 decoupled loads. The memory cycle and access times are varied as before, as is the LDQ length. However, instead of varying the LAQ length along with the LDQ length, two sets of simulations are run, one with the LAQ set to unit length (effectively no LAQ) and one with it set to a length of 128. In all four simulations deterministic unit strides are used.

Figure 25 is of particular interest, since this one graph shows the efficiency of both the address and data sub-processors for LAQ lengths of both 1 and 128. The significance of the fact that only one surface is shown is considerable - that the length of the LAQ makes very little difference to efficiency. Closer examination reveals that the data sub-processor efficiency is identical regardless of the queue size.

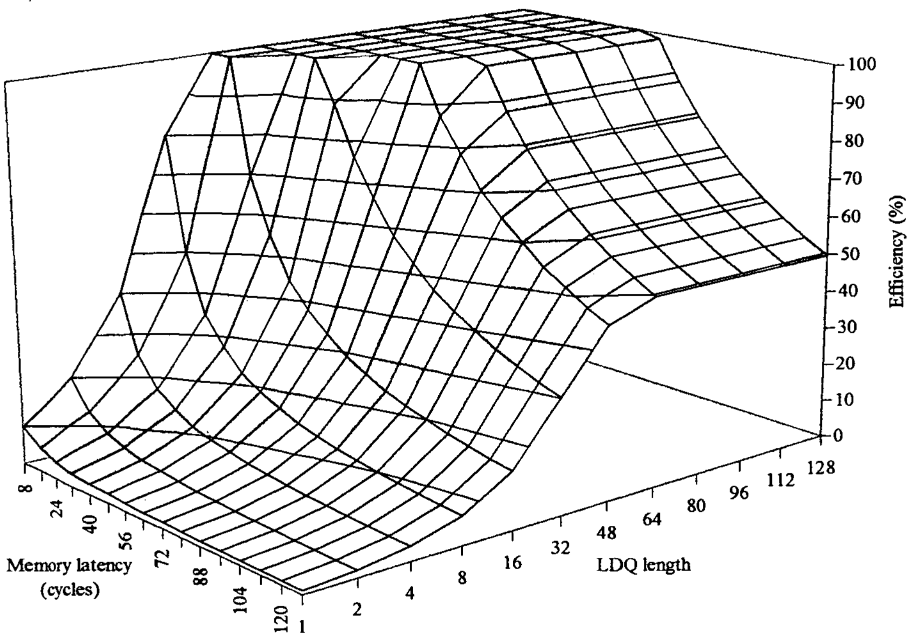


Figure 25 - Sub-processor efficiency. (Ex. 2)

Examining the difference in address sub-processor efficiency for the two LAQ lengths (Figure 26) reveals a very small difference between the two, on the order of $\frac{1}{10}^{\text{th}}$ of a percent. The shape of this surface indicates that in the overtagged, overbanked area of the performance envelope there is no difference at all. The biggest improvement in efficiency (a whole 0.12%!) occurs when the system has just become underbanked, although this increased efficiency drops away rapidly as memory latency increases. With regard to the other dimension of the performance space, increasing the LDQ size gives a slight performance advantage when the LAQ is long in the undertagged part of the graph, with the difference in efficiency elsewhere being dependent entirely on memory latency.

A similarly unvaried picture emerges when we examine the total runtime for the experiment. The architectural simulator shows no correlation whatsoever between LAQ length and runtime, with the experiment giving the same results whether the queue length is 1 or 128. The simulated model show a slightly less perfect correlation. Figure 27 shows that when the LDQ length is 64 and memory

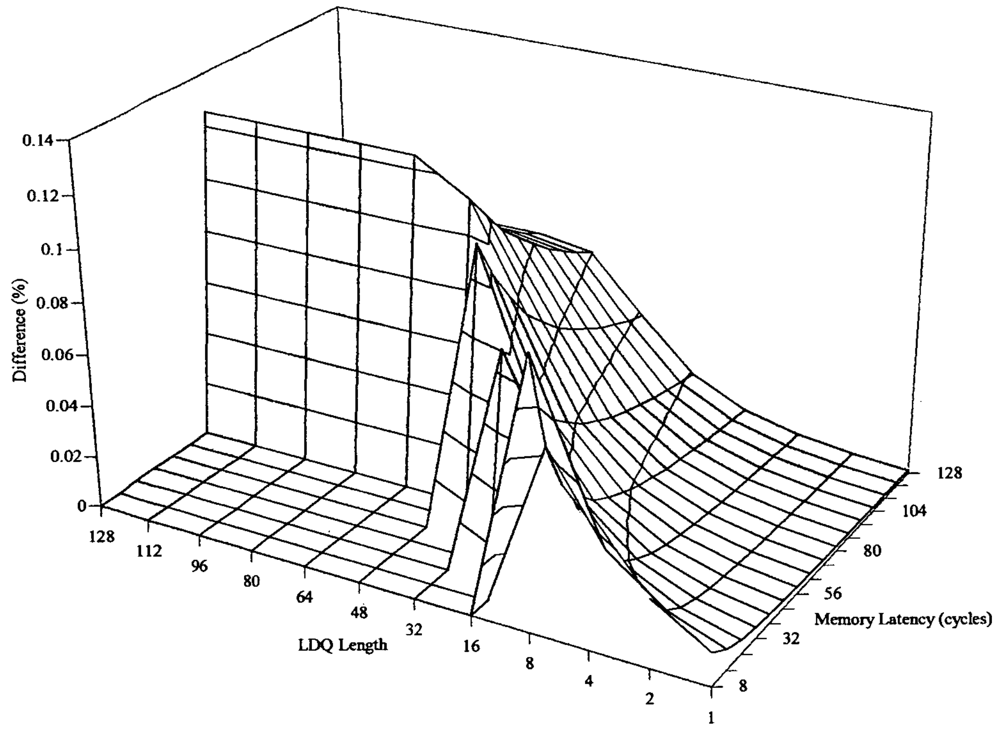


Figure 26 - Difference in simulated AU efficiency for LAQ lengths 1 and 128. (Ex. 2)

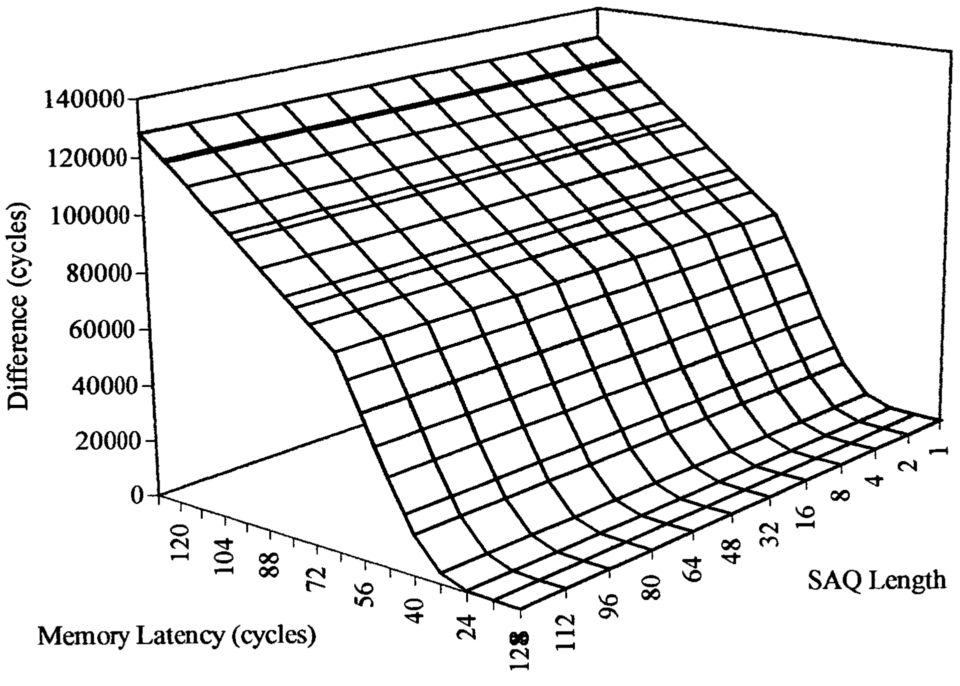


Figure 27 - Modelled runtime difference. (Ex. 2)

latency is 88 cycles, the system with LAQ length 1 outperforms the system with an LAQ of length 128, completing the test run some 500 processor cycles earlier. However, in terms of the total runtime for the experiment, this represents an error of some 0.003%. Comparison of the simulated model with the architectural simulator reveals negligible discrepancies, with the errors following the familiar distribution as shown in Figure 16 and others.

As would be expected given the near-identical runtimes, there is no substantial variation in the effective latency. However, when the actual latency is examined, the difference between the two systems becomes apparent. Figure 28 and Figure 29 show the actual latency experienced by load packets in systems with LAQ lengths of 1 and 128 respectively.

The shapes of the two graphs show the hypothesised behaviour. The actual latency for a queue length of 128 is very large indeed and is of the same order of magnitude as the figure obtained by multiplying the LAQ length by the memory latency. When the queue is a single element in size (that is, reduced to little more than an address buffer) actual latency is much lower. The latency does increase when the LDQ is short (due to undertagging) or when memory latency is high (due to the address buffer having to wait for the desired memory bank to become free) but even in these cases the total latency for the memory packet is some two orders of magnitude lower than with the longer LAQ.

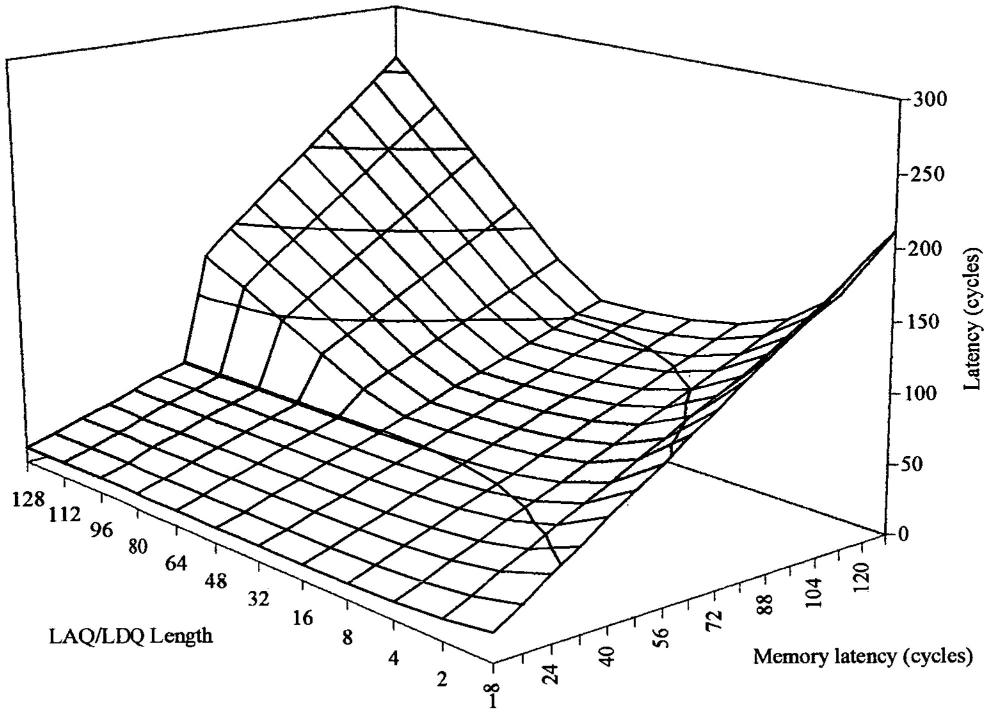


Figure 28 - Simulated actual latency for LAQ length 1. (Ex. 2)

5.1.6 Conclusions

This result is of considerable importance. When the system is overbanked and has sufficient load tags available, the LAQ is unlikely to be used since most load packets arriving in the queue will be dispatched to the appropriate memory bank almost immediately. In the case where the system is underbanked and/or undertagged, the LAQ queue fills rapidly. The speed with which it fills will be determined by the rate at which load packets are added and the size of the queue. A longer LAQ will postpone the point at which the address sub-processor stalls due to a full LAQ but this delay is likely to be negligible relative to the time the processor spends in this saturated state, adding new loads to the queue only when they are removed from the other end.

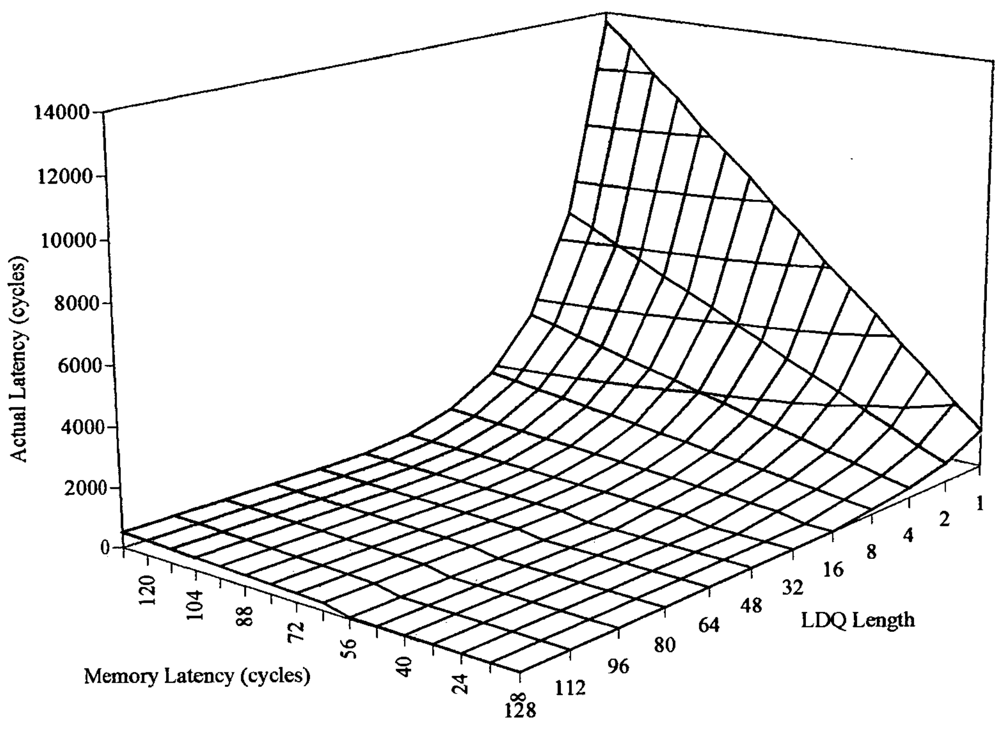


Figure 29 - Simulated actual latency for LAQ length 128. (Ex. 2)

A number of other factors also weigh against the use of an LAQ. It represents a considerable increase in process state which must be saved or restored every time a context switch is performed. For example, the time taken to save 128 64 bit values, would be considerable since this represents an additional 1kB of context. The presence of an LAQ also considerably complicates the detection of RAW and WAR memory hazards.

In a system with both an SAQ and an LAQ checks must be made to ensure that their possibly differing rates of progression do not lead to a store to some address overtaking a later load from that same address and vice versa. Some mechanisms for performing this were discussed in Section 2.4.6. If there is no LAQ, however, hazard avoidance becomes much more straightforward. Whenever a load is encountered, the address is checked against the (associative) SAQ. If a match is found, the load is stalled until such time as the matching store leaves the SAQ. If no match is found the load packet can be sent to memory.

Perhaps the only real advantage in having an LAQ is one which will only be touched on briefly. In the simulated architecture described here it is assumed that all accesses to memory are accesses to large aggregate structures such as vectors, matrices or lists. For the sake of simplicity, it has been assumed that all scalar variables are assigned to registers and that there are always enough registers. While this assumption makes little difference to the simulation results, in practice the LAQ and SAQ would contain the addresses of scalar accesses too.

In cases where a particular scalar variable is frequently stored to or loaded from memory, or in the case of certain classes of vector operation, it may still be desirable to cache frequently accessed variables in conventional fashion. If a cache is present, the LAQ can take advantage of the fact that load requests may arrive in the queue some time before the access actually reaches the queue head. In this situation it may be possible to exploit this to provide an implicit cache preload feature, so that when the load reaches the head of the queue the data is already in the cache.

Whether such a technique would be useful would depend on the frequency of access to scalar variables, the latency of memory and the average LAQ length. The last two of these would, in combination, determine just how much time was available to perform a preload.

Overall, it seems to be the case that the LAQ may be considered redundant. It gives no noticeable increase in performance, its absence produces no drop in performance, and the only effect of its presence seems to be to greatly increase the actual latency of load packets.

5.1.7 Experiment 3: Store streaming

Having examined the behaviour of a stream of decoupled loads, it is also worth investigating the behaviour of a similar stream of stores. This experiment was almost identical in form to Experiment 1, consisting of a series of simulations of a sequence of 100,000 decoupled stores while varying the memory latency and SAQ/SDQ length.

Building on the results of Experiment 1, all of the simulation runs used random (or pseudo-random) rather than unit stride accesses. Three separate runs were performed. The first varied the SAQ and SDQ length together, while the second and third held the SDQ length at 1 and 128 elements respectively. This was to test the hypothesis that just as the LAQ was redundant in the address sub-processor, the SDQ was redundant in the data sub-processor. This would seem a reasonable behaviour to expect. The address sub-processor will generally try to run ahead of the data sub-processor, meaning that the addresses for stores will already be available when the data sub-processor produces the matching data. Thus, stores will be sent to memory as soon as the data sub-processor completes the store packet unless the memory system is overloaded. If memory does

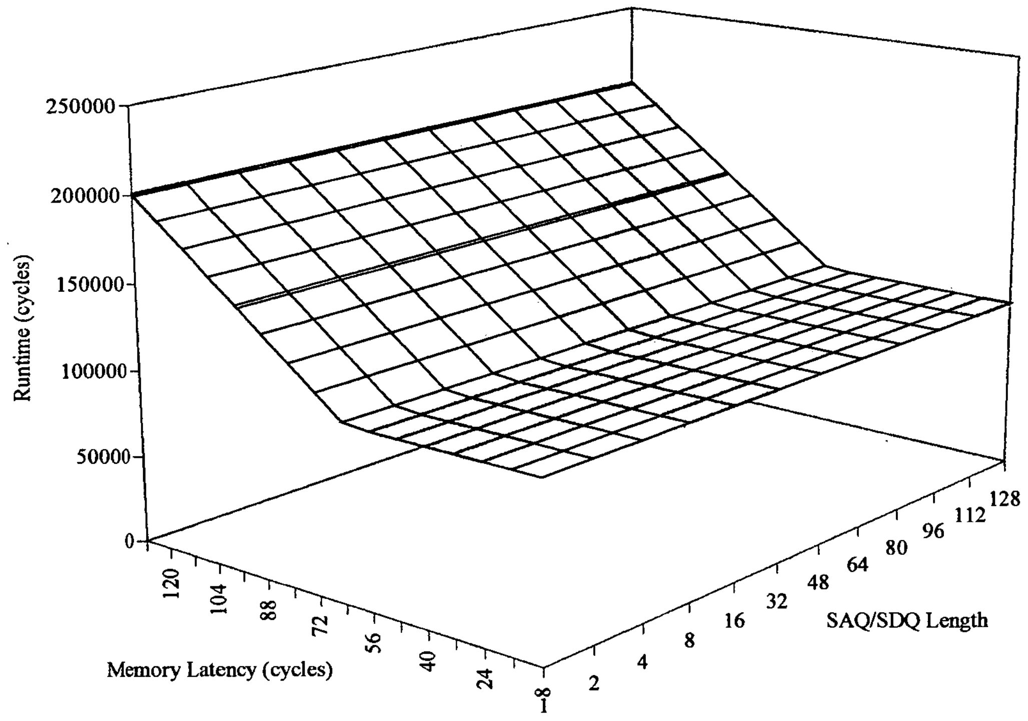


Figure 30 - Simulated runtime. (Ex. 3)

become saturated, longer queues will only postpone the inevitable and serve little purpose other than to delay the point at which the sub-processor stalls for a short while.

A number of statistics were gathered for each run of the experiment. These included the total runtime for the data sub-processor, the average lengths of the SAQ and SDQ, and the actual store latency. It was found that neither queue particularly effected the total runtime. Figure 30 shows the runtime for all three runs. The runtime remains constant and low (a near flat plain) while the system is overbanked then starts to rise as underbanking sets in and the memory system begins to saturate. However, this behaviour seems to be completely independent of the length of the SAQ or SDQ.

Closer examination of the data shows that there are differences, but that they are very small. Figure 31 shows the difference between the runtime when the SDQ length is fixed at 128 elements and at 1 element. Notice that the two are identical when the system is overbanked, with the system with a 128 element SDQ gaining a tiny performance advantage once memory becomes underbanked. This

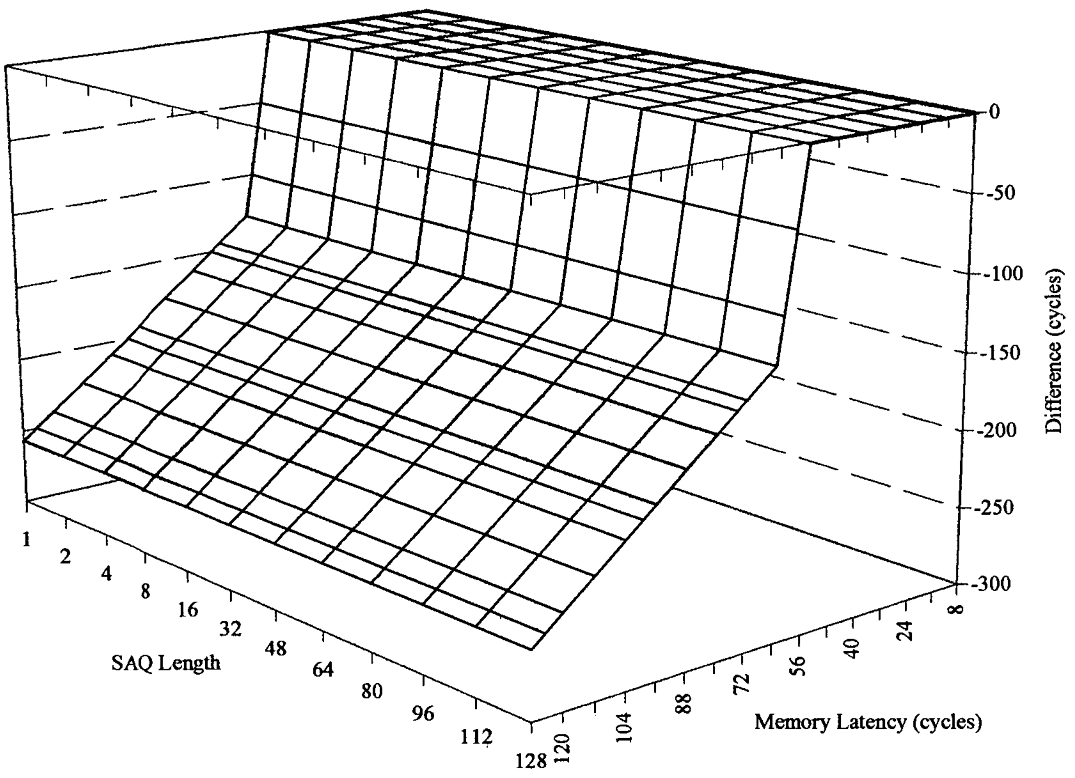


Figure 31 - Runtime difference for SDQ lengths of 1 and 128. (Ex.3)

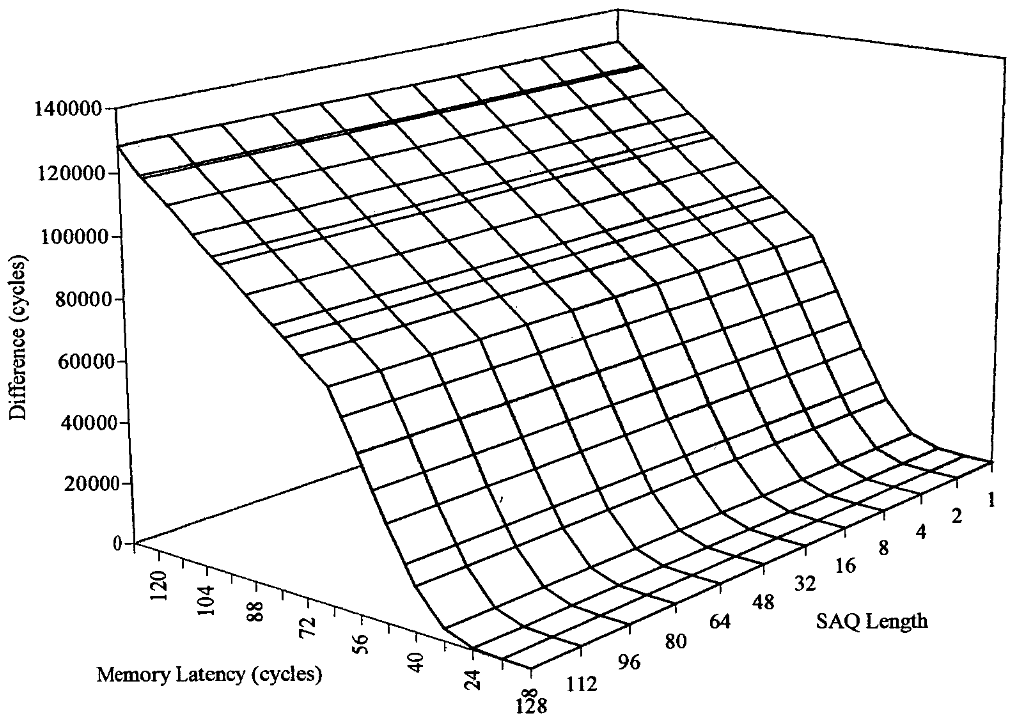


Figure 32 - Difference between model and simulator runtime. (Ex. 3)

small difference is due only to the larger queue postponing the point at which the data sub-processor begins to stall for a few cycles. At the point at which the data sub-processor executes its last store the SDQ will still be full and it will take some time for all of the pending stores to complete. This means that the actual runtime for the system as a whole is almost identical, regardless of the queue length although the data sub-processor runtime does not always reflect this very accurately in this experiment.

Interestingly, the behaviour of the simulated model does not map very closely to that of the architectural simulator. The level of accuracy obtained is considerably lower than that in the previous two experiments. Initially the two models track well, but as memory latency increases the two drift apart, until the runtime when the memory latency is set to 128 processor cycles is a full 50% greater for the simulated model than for the architectural simulator. This behaviour is illustrated in Figure 32, which shows the difference in runtime between the architectural simulator and the model.

The reason for this difference is not immediately obvious. The store path is one part of the simulator model which relies more heavily on an architectural model than a queueing network, so one would expect it to behave in a similar manner. The most likely explanation is that the difference is due to the architectural simulator using a pseudo-random address remapping scheme which is designed to have little effect on performance when used with a wide range of strides. The simulated model, on the other hand, uses a completely random bank assignment method, which can potentially lead to stores being distributed unevenly over the banks. This was verified by running the simulated model again, this time with a fixed unit stride access pattern.

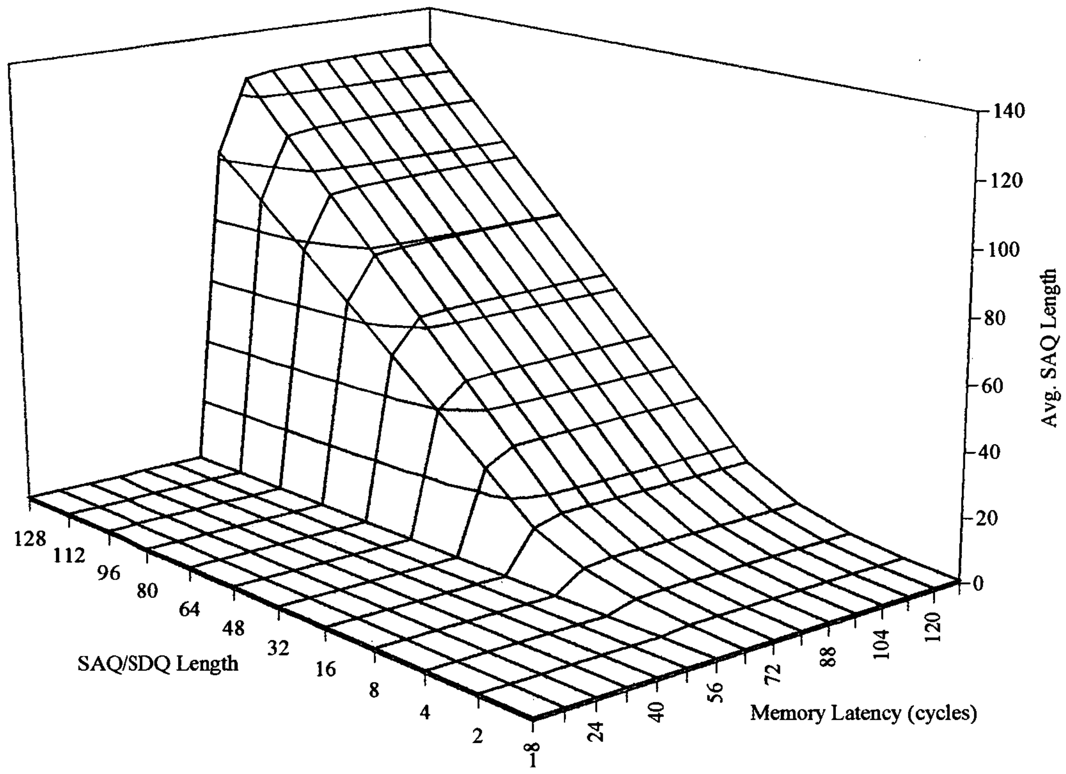


Figure 33 - Simulator average SAQ length. (Ex. 3)

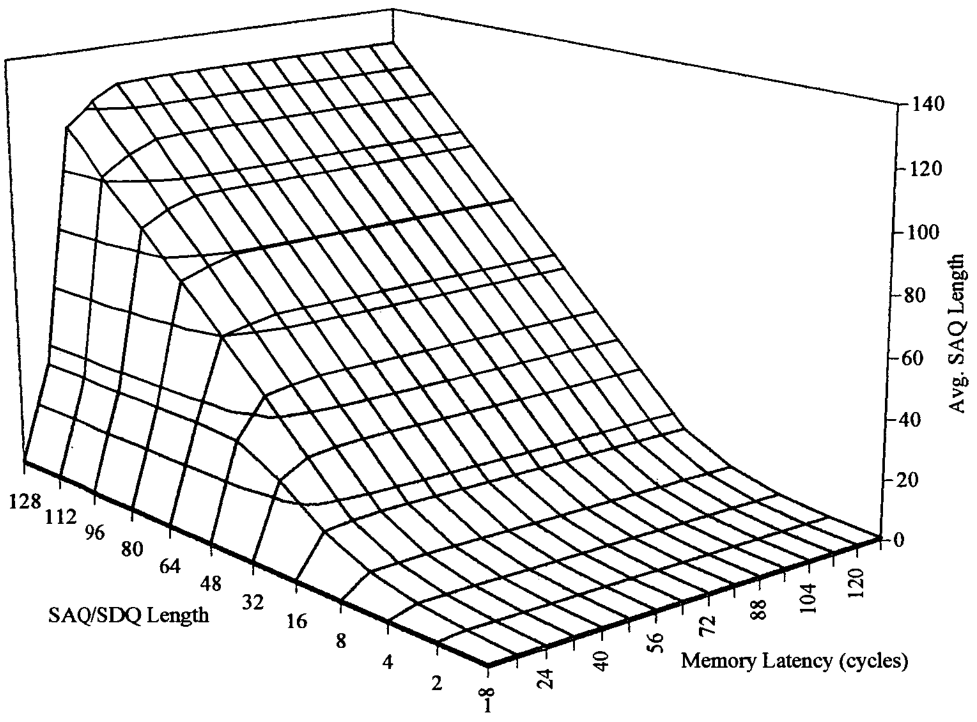


Figure 34 - Model average SAQ length. (Ex. 3)

Despite these minor differences, the simulated model appears to verify the fact that the runtime is completely independent of the SDQ length, since all three runs produced identical runtimes on the simulated model.

Other measurements taken included the average lengths of the SAQ and SDQ during the simulation run. Figure 33 and Figure 34 show the average SAQ length for the architectural simulator and simulated model respectively, while Figure 35 and Figure 36 show the average SDQ length when the maximum SDQ length varies with the SAQ length. Figure 37 and Figure 38 show the same measurement when the maximum SDQ length is held at 128 elements.

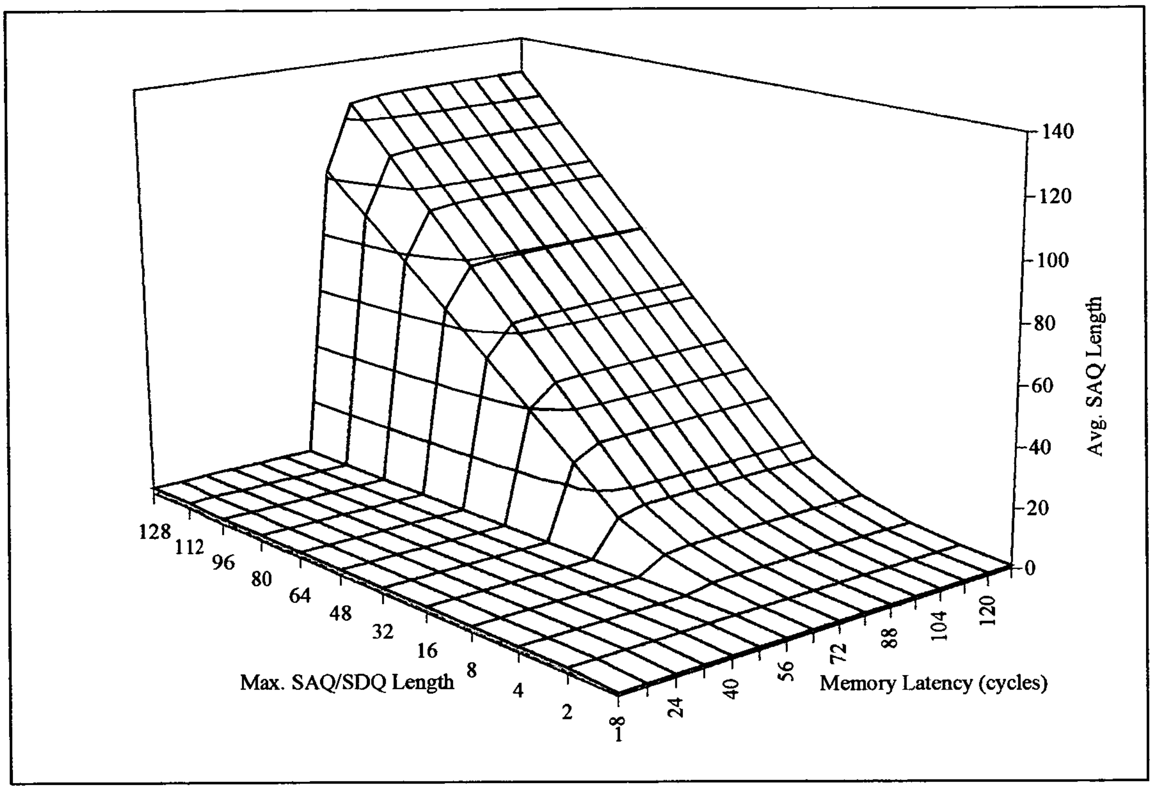


Figure 35 - Simulated SDQ length when SAQ and SDQ lengths are varied. (Ex. 3)

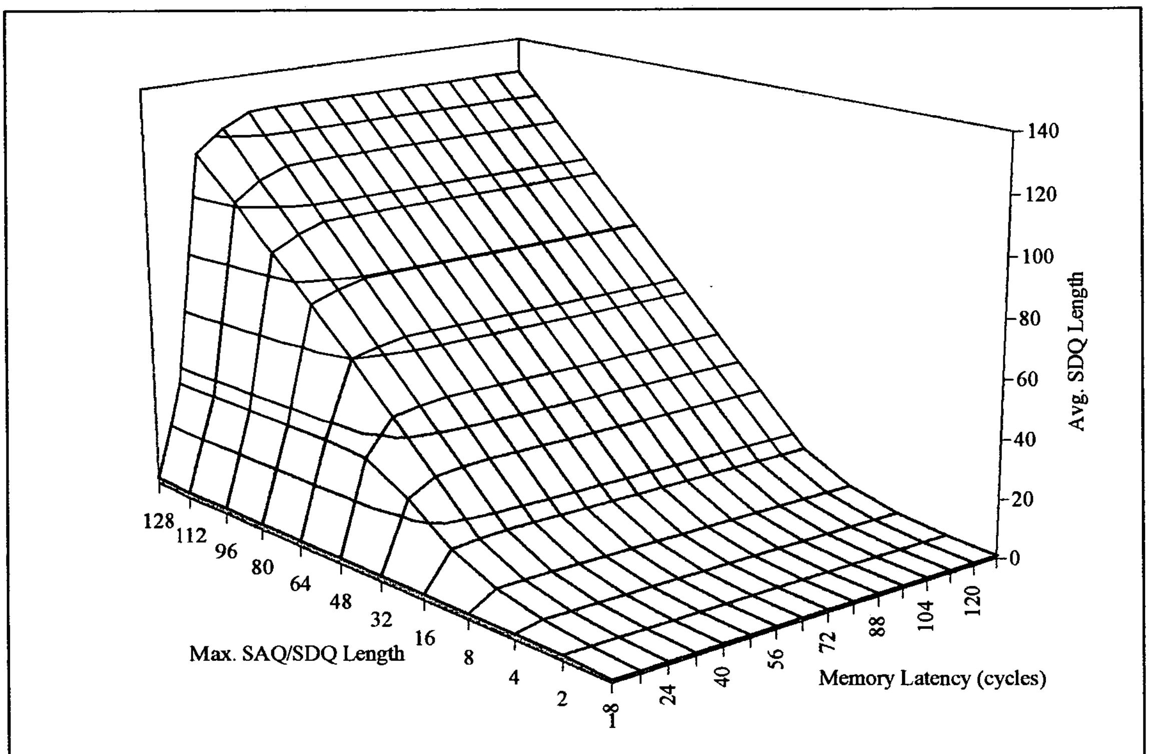


Figure 36 - Modelled SDQ length when SAQ and SDQ lengths are varied. (Ex. 3)

In the case of the SAQ, the average length is much as would be expected, although the random access pattern used by the simulated model does produce some interesting differences. The architectural simulator has two distinct behaviours, depending on whether or not the memory subsystem is over or underbanked. When overbanked, the SAQ remains empty, as each arriving store address is paired with a store datum from the data sub-processor and sent immediately to memory. The overbanking in the memory subsystem combined with the remapping algorithm used means that the bank is always free, so no stalls of the SAQ occur. This is represented by the flat area on the left hand side of the graph.

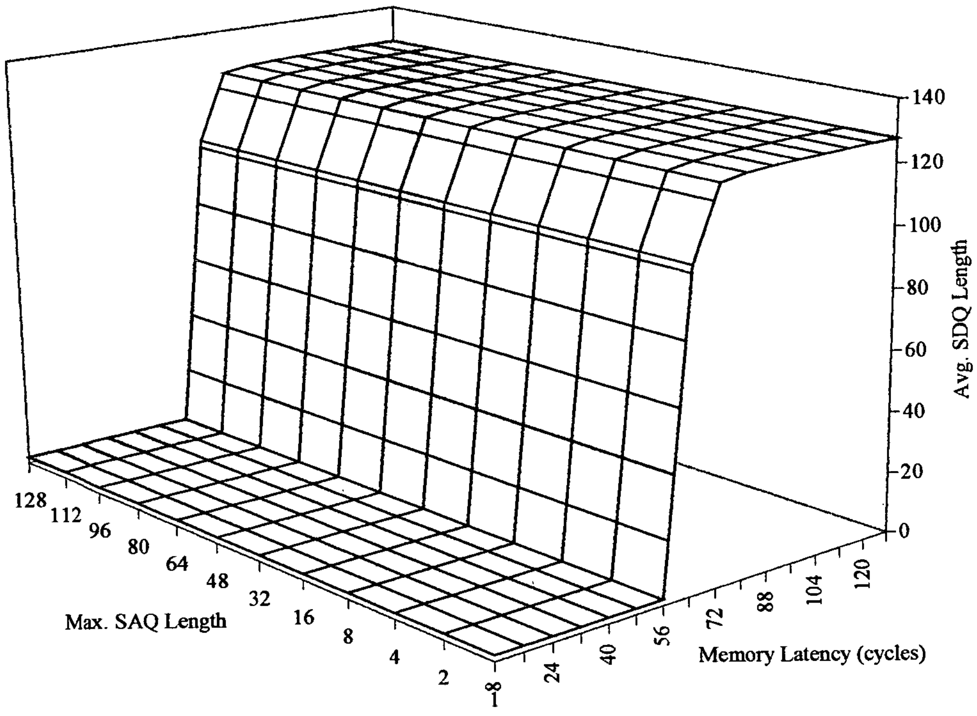


Figure 37 - Simulated SDQ length when SDQ size is 128. (Ex. 3)

If the memory latency is increased to the point where the system becomes underbanked, the behaviour changes. Stalls begin to occur due to store packets being produced faster than the memory subsystem can accept them, and the SAQ begins to fill. Figure 33 confirms this, and shows that once the system becomes underbanked the average SAQ length grows until the queue is full. Around the balance point where the system's behaviour switches from the overbanked to underbanked mode, there is an area of instability in which the SAQ does begin to fill but fails to reach its full potential length.

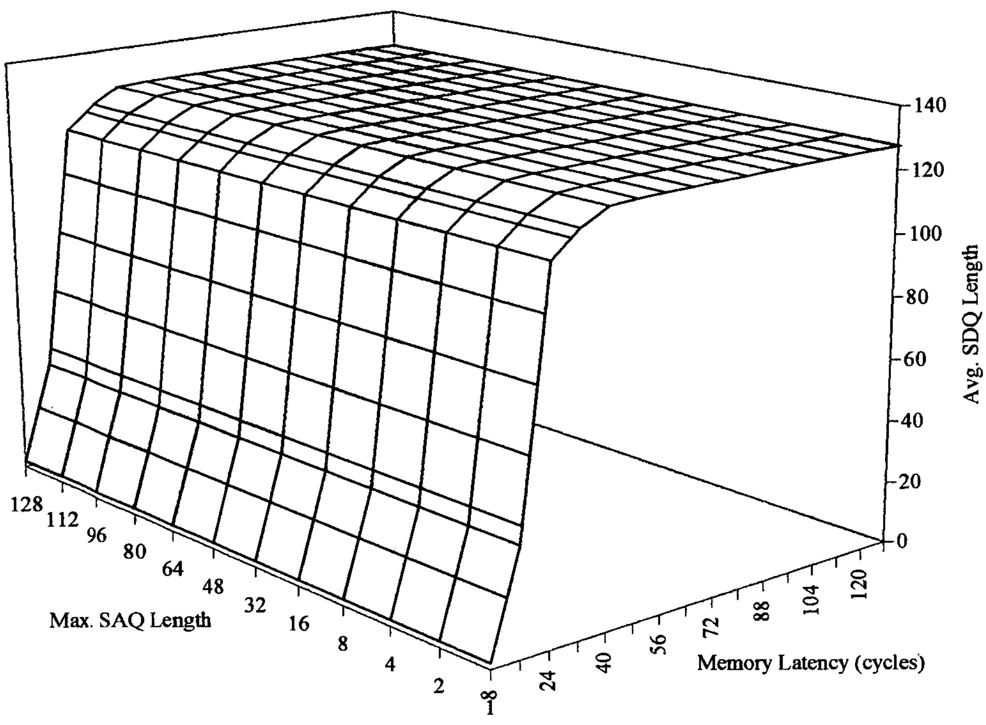


Figure 38 - Modelled SDQ length when SDQ size is 128. (Ex. 3)

The behaviour of the simulated model (see Figure 34) is slightly different, due largely to the random access pattern used by the model. This pattern means that it is possible for consecutive stores to access the same bank, and results in the SAQ filling even when the system is overbanked. Once again there is an area of instability where the SAQ fills but not completely. However, when the memory access pattern is random this zone is clearly in the overbanked region of the design space.

The SDQ length exhibits a similar behaviour to the SAQ and this is shown in Figure 35 and Figure 36. The queue remains empty when the system is overbanked and simulated on the architectural simulator, but fills when the system becomes underbanked. When simulated using the simulated model, the queue fills before the system becomes underbanked, due to the random access patterns causing hot banks. Figure 37 and Figure 38 show the average length when the SDQ length is fixed at 128 for both the simulator and model. Note that in both cases, once the point is reached at which stalls can occur, the queue fills.

5.1.8 Conclusions

This experiment shows some interesting features of store behaviour. It appears to be the case that the lengths of the SAQ and SDQ have absolutely no bearing on performance. If the memory subsystem is overbanked, neither queue fills and both are thus redundant. If the memory system is underbanked, both fill rapidly and will eventually saturate, stalling the appropriate sub-processor. Since saturation and stalling is inevitable in an underbanked system, regardless of queue length, it would seem sensible to eliminate the SAQ and SDQ entirely, thus reducing hardware costs.

However, this experiment has looked only at streams of decoupled stores in isolation. In this case both sub-processors operate independently and the limiting factor is the speed at which the memory system can accept new store packets rather than the speed of either sub-processor. When decoupled loads are factored into the equation, this behaviour changes. It becomes advantageous for the address sub-processor to be able to run ahead of the data sub-processor. If this happens, it is undesirable for the address sub-processor to stall on the next store since this will block subsequent load operations. So while the presence or absence of the SAQ has no effect on the performance of store operations, its absence could be very damaging when a program is generating a more realistic mixture of loads and stores. This behaviour will be investigated further in the next experiment.

The experiment also revealed a limitation in the simulated model, namely the inaccuracies caused when using random access patterns to approximate the pseudo-random permutation behaviour of the memory access hashing algorithm. While this had little effect in previous experiments, it produced noticeable differences in the behaviour of the model and simulator.

This is not, however, felt to be a flaw in the model itself but rather an inaccuracy brought about by comparing random and pseudo-random addressing. Since the inaccuracies skewed the model results in a conservative rather than an optimistic direction, it was felt that this inaccuracy was acceptable, as long as additional tests were performed to verify that the inaccuracy was due to the access pattern rather than some more serious flaw in the model. In more general examples where the model was used in isolation without the support of an accompanying simulation, it is felt that the fully random access pattern would still give a reasonable approximation of system behaviour and would, at least, not give misleadingly optimistic results.

5.1.9 Experiment 4: The SAXPY loop

The SAXPY loop is one of the kernels that form the core of the standard (if rather simplistic) Linpack benchmark. It evaluates the equation $z = a \times x + y$ for vectors x, y and z and scalar a.

While the limited capabilities of the Level 1 architecture prevent a truly high-performance implementation of SAXPY (for example, the ACRI-1 is capable of initiating one iteration per clock cycle) it provides a more realistic test loading of the memory subsystem than any of the earlier

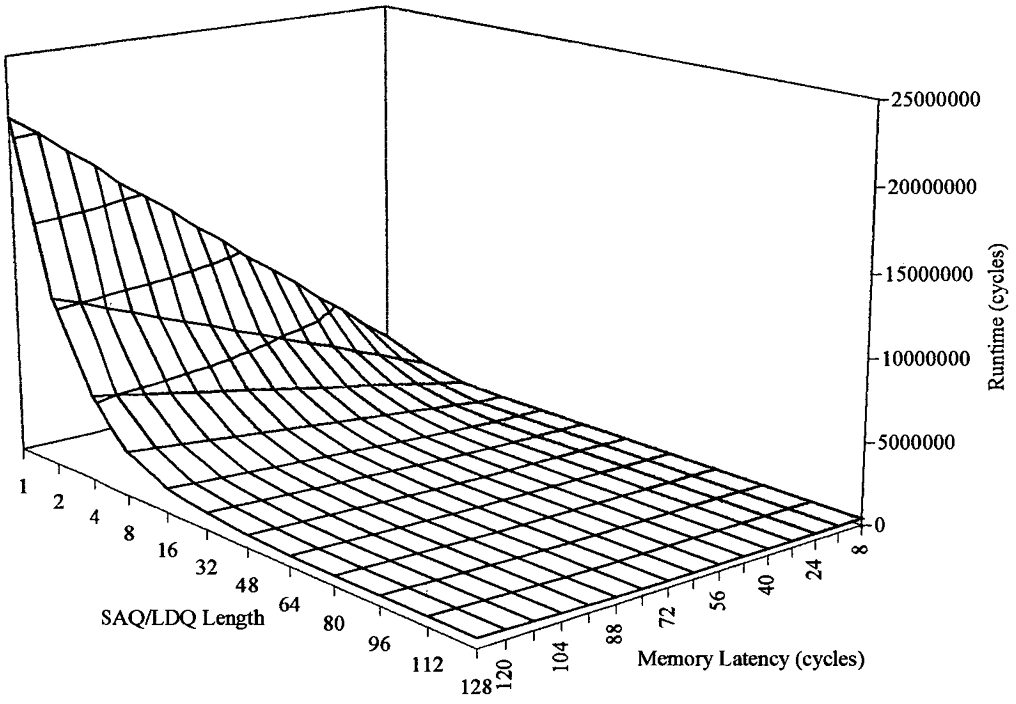


Figure 39 - Data sub-processor runtime for first and second. (Ex. 4)

experiments and gives a reasonable idea of the system's behaviour when running a typical vector-based operation.

For the purposes of this experiment, which is intended solely to show access/execute decoupling behaviour, the SAXPY loop has been fully unrolled. In practice this would be highly inefficient, since each of the vectors x, y and z have been set to have a length of 100,000 elements. However, these large vector lengths help to even out any localised irregularities in system behaviour, causing the results to approximate the steady state in the loop more accurately.

Three separate runs of the experiment were performed, with each run encompassing a range of memory latencies and SAQ/LDQ lengths. On the first run the LAQ and SDQ lengths were held at 1 packet, thus effectively eliminating them, while on the second both were set at 128 packets. The purpose of these two runs was to establish whether the combination of load and store operations might produce measurable effects on performance that had not appeared in the previous three experiments. These showed that the LAQ and SDQ were redundant, but it was felt necessary to test this further in case the interaction of the two produced changes. The third test run held the LAQ, SDQ *and* the SAQ at unit length. Experiment 3 appeared to indicate that the SAQ was also redundant, however it was felt that this was unlikely. By comparing the third run with the first, it was possible to compare the behaviour of the system with/without an SAQ.

Figure 39 and Figure 40 show the total data sub-processor runtime for all three runs. They appear to confirm the two principal hypotheses of the experiment: that the LAQ and SDQ are redundant, while the SAQ, despite the indications of Experiment 3, *is* necessary. For clarity, the runtimes for the first two runs have been combined into a single graph, since they differ by a very small amount. The

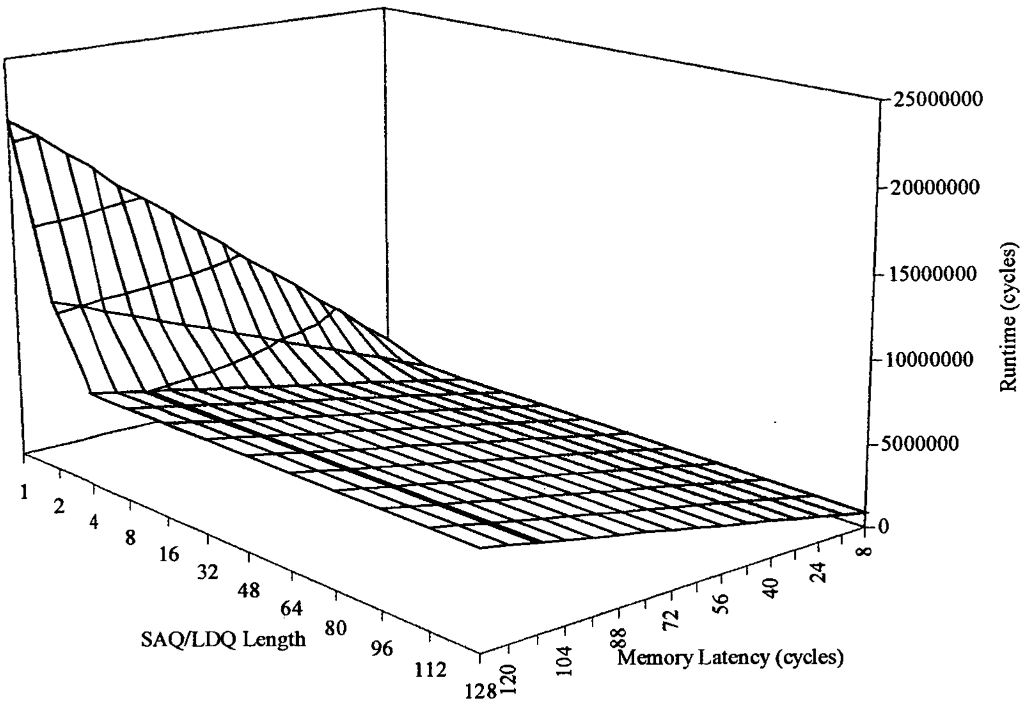


Figure 40 - Data sub-processor runtime for third run. (Ex. 4)

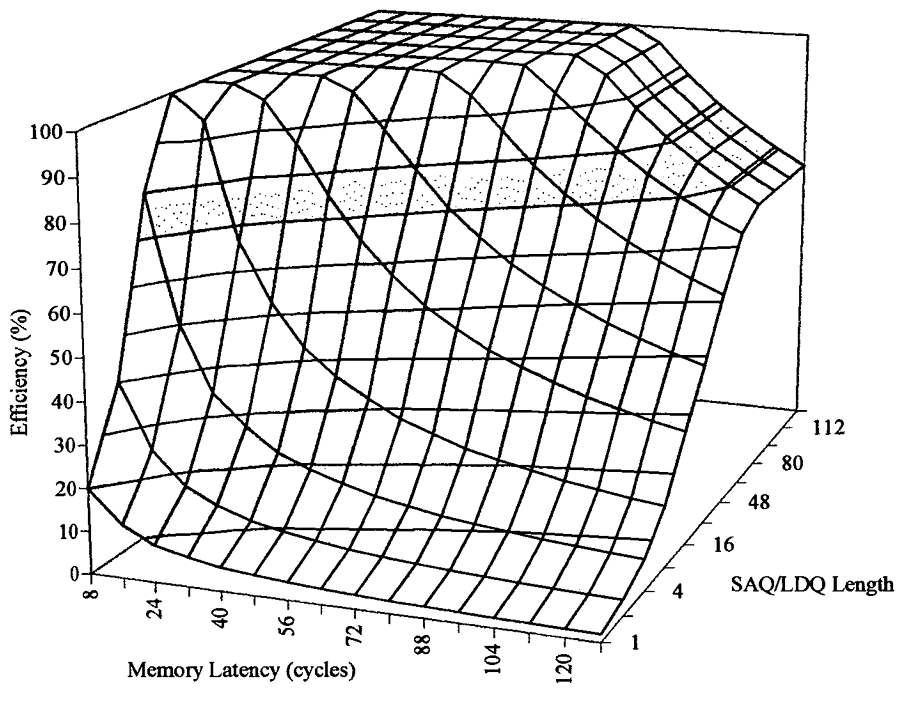


Figure 41 - Data sub-processor efficiency for first and second runs (Ex. 4).

runtime for the third run is shown separately, to more clearly illustrate the differences. Note that in both graphs, the axis labelled "Queue Length" refers only to the lengths of those queues that are varied - the LDQ in all three experiments, and the SAQ in the first two runs.

All three runs have similar performance for short queues. This is due to undertagging being the dominant characteristic of the system, with the processors unable to fully utilise the available bandwidth. However, as the queue length increases the runtime stabilises, with the first two runs demonstrating the characteristic performance of a decoupled system. For undertagged systems the performance is at least partly dependent on memory latency, but as the LDQ and SAQ lengths increase the performance improves, rapidly reaching a flat performance "plain" where increases in memory latency have no effect.

The surface described by the runtime of the third run is very noticeably different from the first two. Past a certain point (LDQ length somewhere between 8 and 16) increases in queue length seem to have no effect, although the resulting plain has a marked slope, indicating that the system is not as

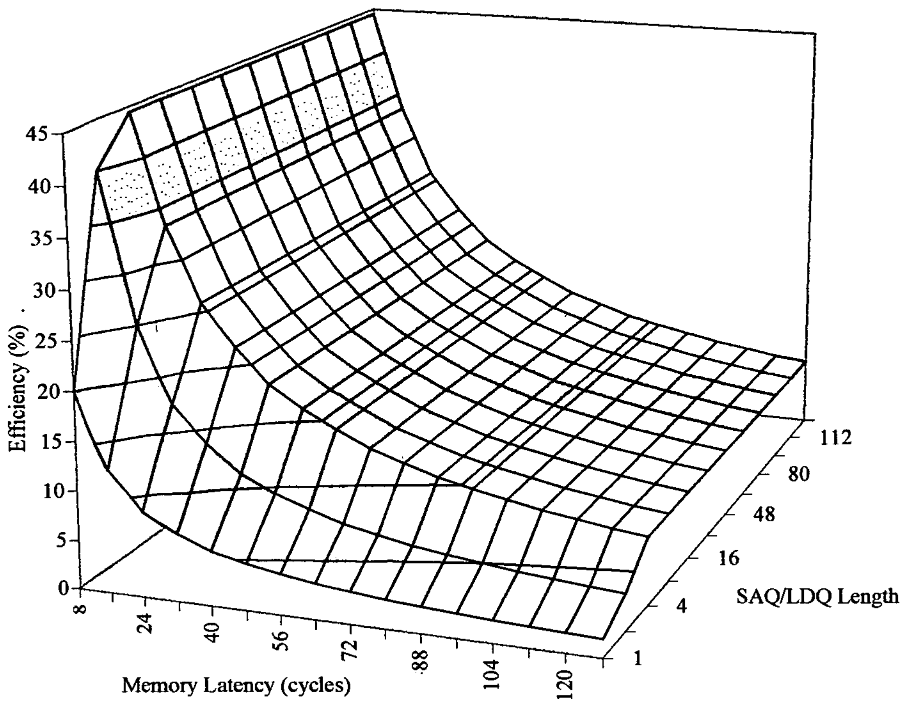


Figure 42 - Data sub-processor efficiency for third run (Ex. 4).

effectively decoupled as the first two runs and that increases in memory latency produce a corresponding increase in runtime.

Another measure of overall system performance in shown in Figure 41 and Figure 42. The first of these shows the data sub-processor efficiency (i.e. the percentage of time it spent actually doing something rather than sitting around stalled) for the first two runs, while the third shows the efficiency for the third run.

As these figures show, the efficiency of the data sub-processor in the first two runs is largely independent of the presence or absence of an LAQ or SDQ. In both cases efficiency is high (close to 100%) unless the system is underbanked or undertagged. Loss of efficiency due to undertagging sets in earlier for low memory latencies (the disparity between actual bandwidth and peak bandwidth being greater) with full efficiency being roughly limited to those systems with a queue length of greater than 64 (i.e. overtagged) and a memory latency of less than 64 cycles (i.e. overbanked). Outside this area of the performance envelope, efficiency suffers. In overtagged, underbanked systems efficiency drops away steadily (and near linearly) as memory latency increases, while the performance dropoff in undertagged systems is steeper.

The efficiency profile of a system where the SAQ is also held to a length of 1 element differs greatly. With the exception of the case where the LDQ length is held to 1 (which adversely affects performance), the processor efficiency is independent of the length of the LDQ. This is due to the limited SAQ length effectively synchronising the two sub-processors and preventing decoupling from occurring. In the absence of any influence from the LDQ length, efficiency is related directly to the memory latency, with a value for a memory latency of 8 cycles of about 40% dropping rapidly to under 10% for a system with a latency of 128 cycles.

The efficiency of the address sub-processor (Figure 43 and Figure 44) follows the same trends as the data sub-processor for all three runs but is correspondingly lower, with a maximum efficiency of around 70%. This is due to the loop being executed being limited (in an overtagged, overbanked system) by the rate at which the data sub-processor can accept data. Since the address sub-processor

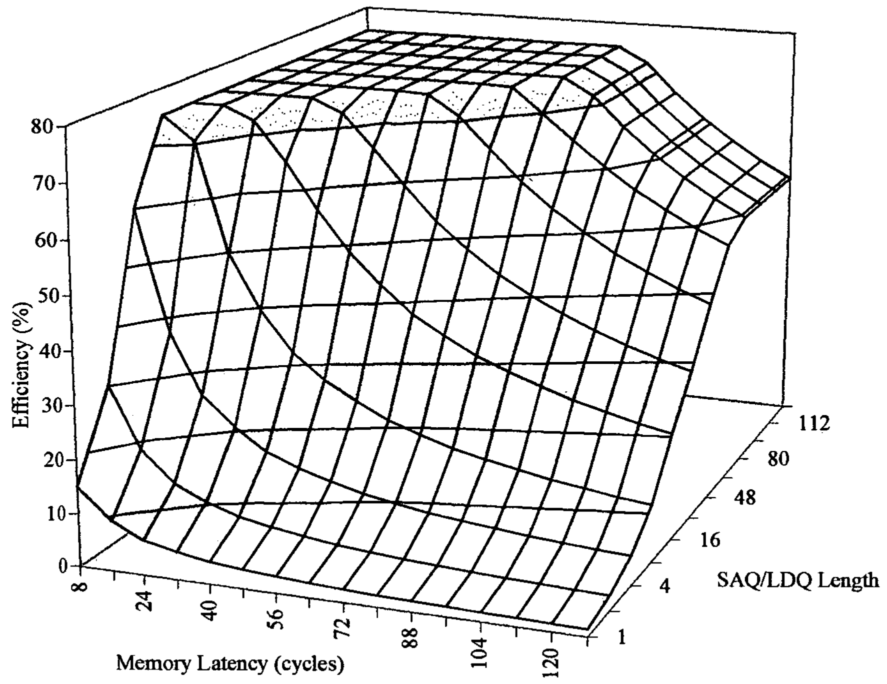


Figure 43 - Address sub-processor efficiency for first and second runs. (Ex. 4)

loop takes at least three clock cycles and the data sub-processor loop at least four, the expected efficiency would be in the region of 75%.

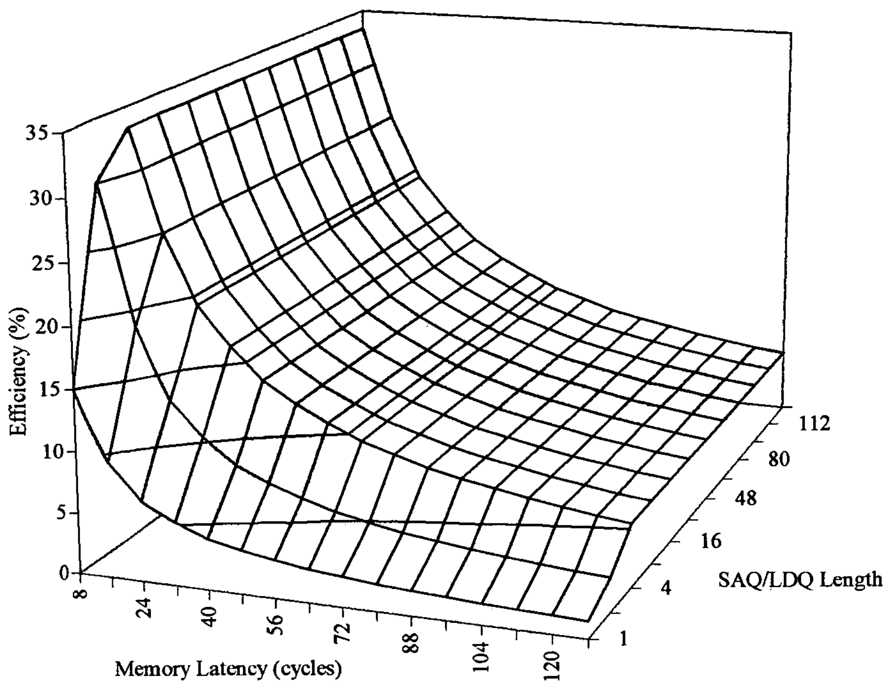


Figure 44 - Address sub-processor efficiency for third run. (Ex. 4)

Examining the actual and effective load latencies for all three runs reveals few surprises. The actual latencies are shown Figure 45, Figure 46 and Figure 47. The most distinctive of these is the actual latency for the third run. As would be expected given the absence of an LAQ and the forced synchronisation problems caused by the absence of either an SAQ or an SDQ, the system runs in a near-sequential mode, with little memory congestion and load packets being removed by the data sub-processor as soon as they arrive. Thus the surface shown is for the most part flat, with a slope that (unsurprisingly) indicates that the actual latency of loads corresponds closely to the memory access time. For an undertagged system with a short LDQ the actual latency increases due to each load having to wait for its predecessor to exit the system.

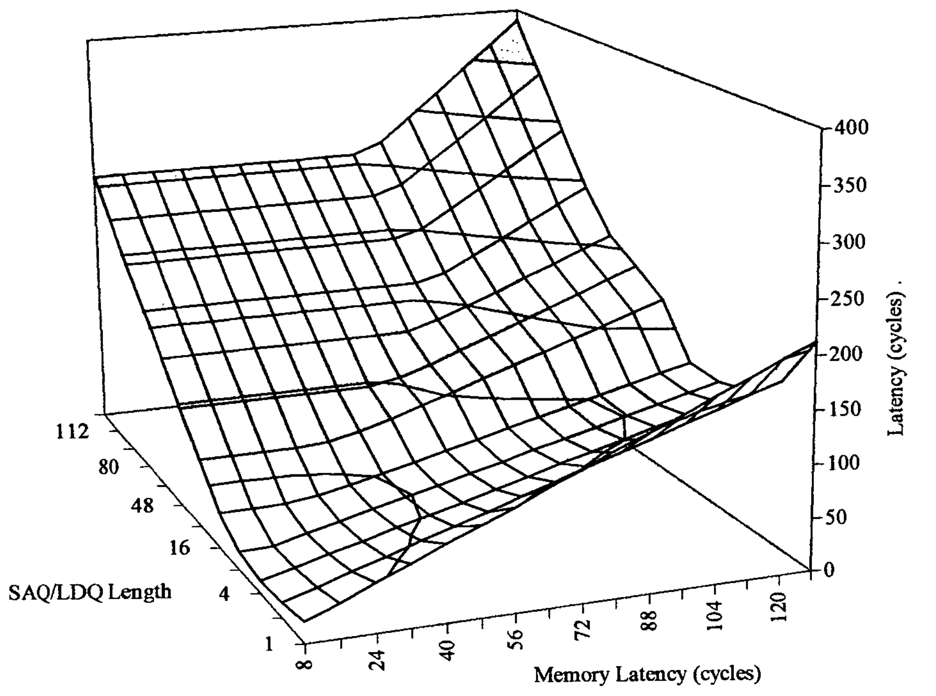


Figure 45 - Actual memory latency for first run. (Ex. 4)

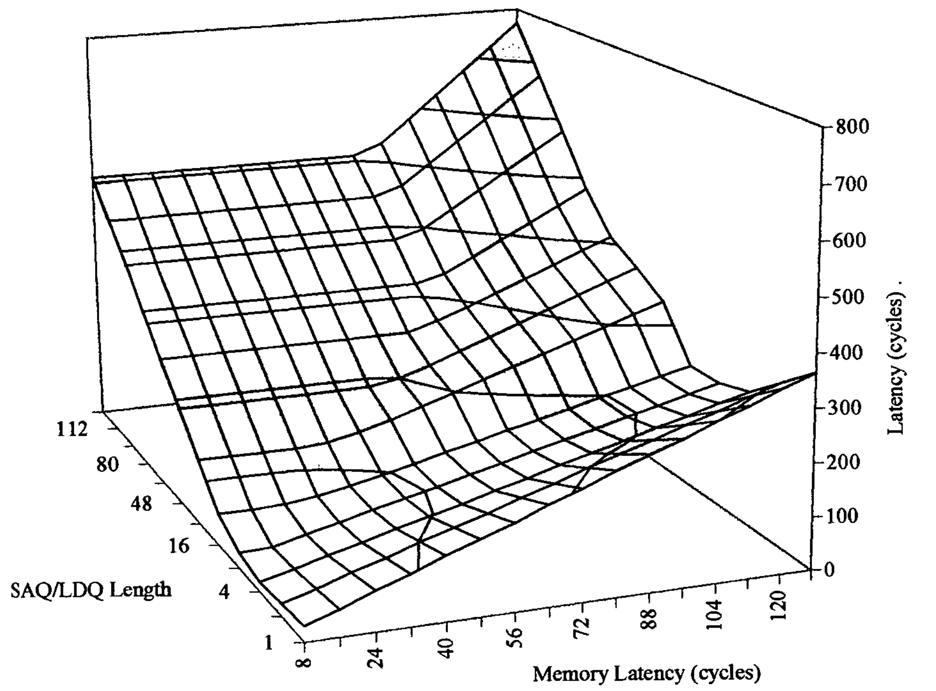


Figure 46 - Actual memory latency for second run. (Ex. 4)

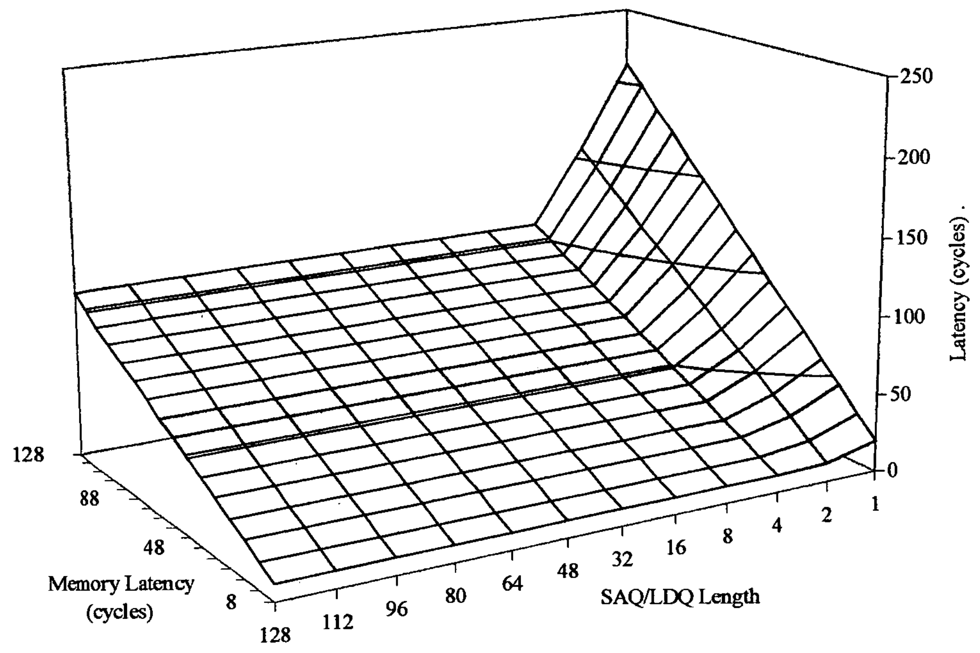


Figure 47 - Actual memory latency for third run. (Ex. 4)

The first run, with an SAQ but no LAQ or SDQ, shows a higher latency due to the greater utilisation of the memory sub-system. This causes the LDQ to fill which in turn increases the latency of the load. Finally the second run represents a system with LAQ, LDQ, SAQ and SDQ. The SAQ and SDQ have no effect on load latency but the presence of the LAQ increases the latency of loads still further since they may potentially be queued there for some time before being dispatched to memory.

The effective latency is much as expected, with the surfaces describing the latencies of the three runs being similar to those of the data sub-processor runtime. The first and second runs behave badly when the system is undertagged but the effective latency drops rapidly as the size of the LDQ increases, levelling out. In the system simulated in the third run, increasing the queue length causes the system to enter a similarly stable behaviour, where effective latency is dependent on memory access time. While some of the actual latency is hidden (contrast this Figure 47) it is not nearly as effective as in the other two runs.

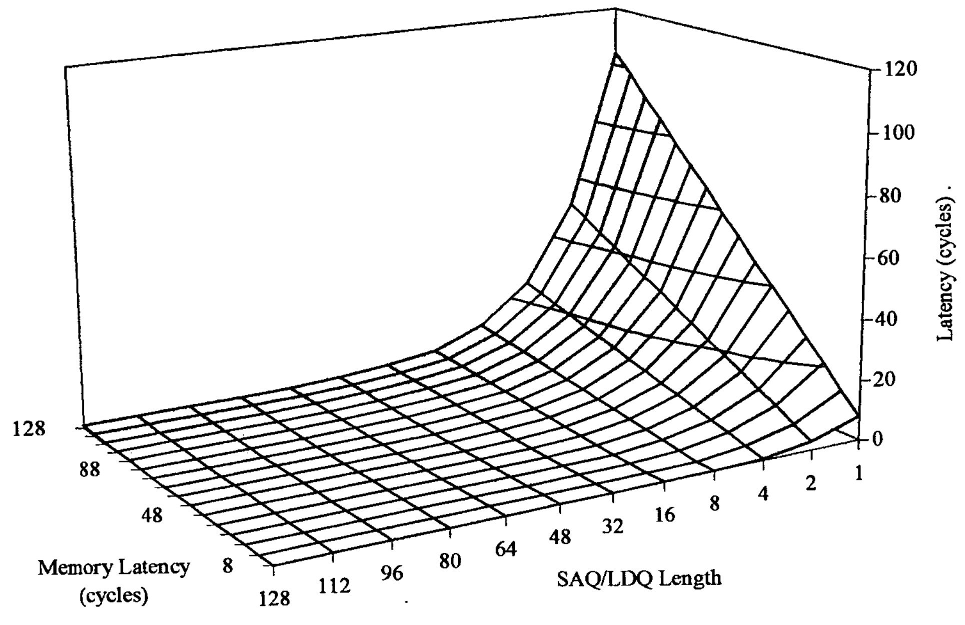


Figure 48 - Effective memory latency for first and second runs. (Ex. 4)

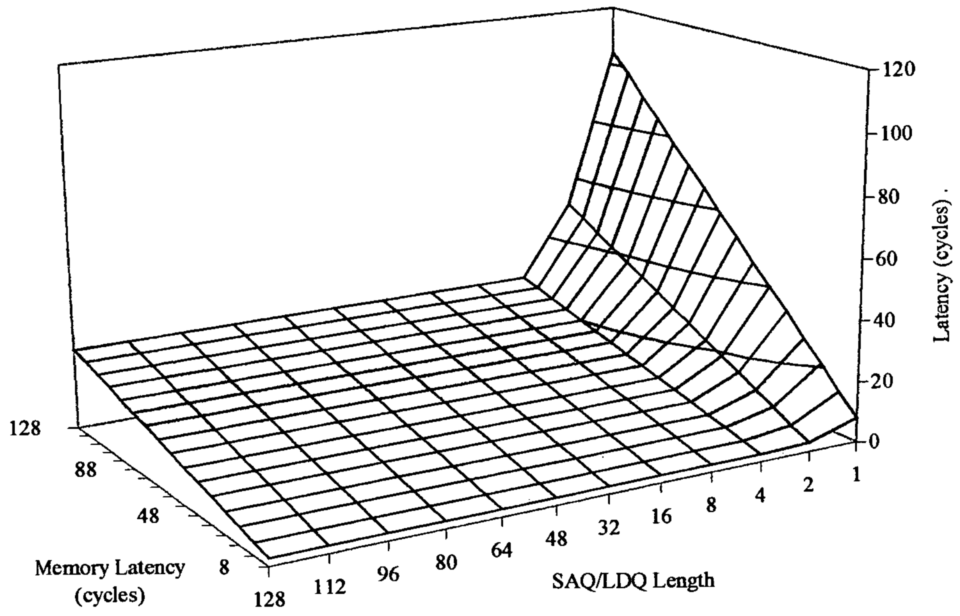


Figure 49 - Effective memory latency for third run. (Ex. 4)

The LDQ itself behaves much as would be expected, with the queue during the first two runs (shown in Figure 50) being at its fullest with short memory latencies due to load packets arriving from memory faster than they can be used. As the memory latency increases the average LDQ length drops slowly until the system becomes underbanked, at which point the length drops off more rapidly. It seems to make no difference whether the system is overtagged or undertagged, as the average LDQ length varies almost linearly with the maximum LDQ size.

When the system is running with no SAQ and no SDQ in the third run, the LDQ fails to fill regardless of memory latency or the LDQ length. This is due to the stores in the program acting as a restraint on performance, synchronising the two units with every store, and thus severely limiting the utilisation of the available memory bandwidth. The graph for this data is not shown.

5.1.10 Comparisons with the simulated model

While it was found that the behaviour of the simulated model for the first and second runs tracked that of the architectural simulator quite closely, this was not the case with the third run. Figure 51

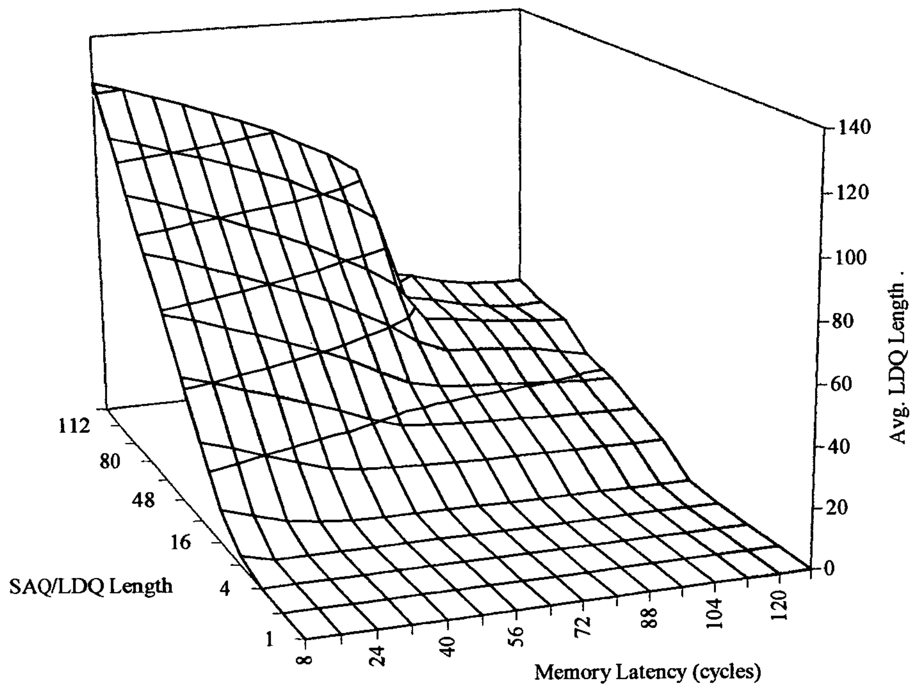


Figure 50 - LDQ length for first and second runs. (Ex. 4)

shows the modelled data sub-processor runtime for all three runs. The shape of this graph closely corresponds to that for the simulated architecture, but behaves differently for the case where the system has no LAQ, SDQ or SAQ. The architectural simulator showed that performance in this case was inferior due to the enforced synchronisations caused by the absence of an SAQ.

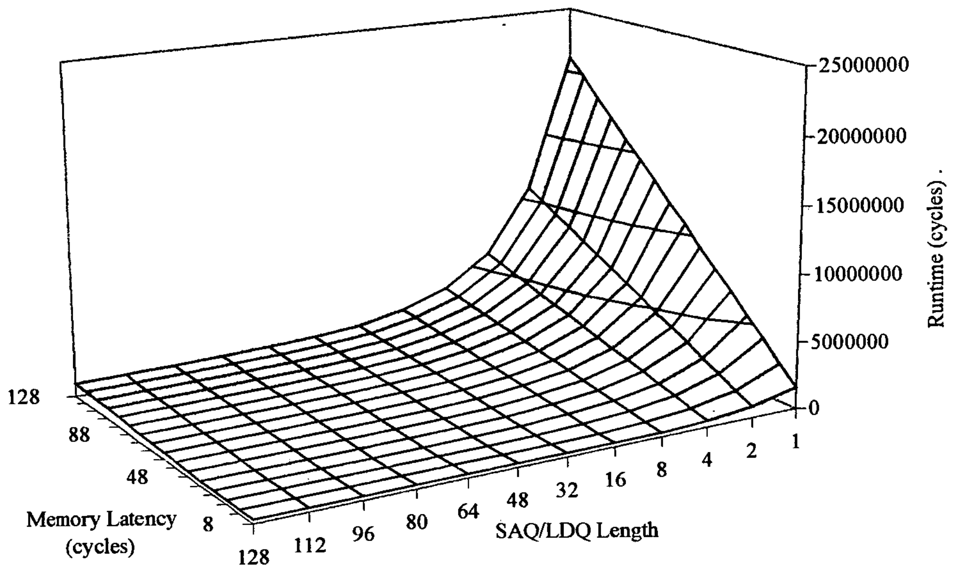


Figure 51 - Modelled data sub-processor runtime. (Ex. 4)

This does not seem to be the case for the simulated model. The difference is due to the way in which the system is modelled. The actual program executes two loads followed by a store, on successive clock cycles, for each "loop iteration". The model knows only that it should initiate a load every 1.5 cycles and a store every 3 cycles. If the store is unable to take place due to the last store not having completed this will delay subsequent stores but *will not* delay subsequent loads. This is not possible on the architectural simulator since the ordering of instructions ensures that the store has to complete before the loads can be issued. In the model the Poisson or deterministic sources for loads and stores are completely separate, and there is nothing to stall a load should the previous store operation have not yet completed. In short, the model fails to take account of the interaction between loads and stores that occurs in the real system.

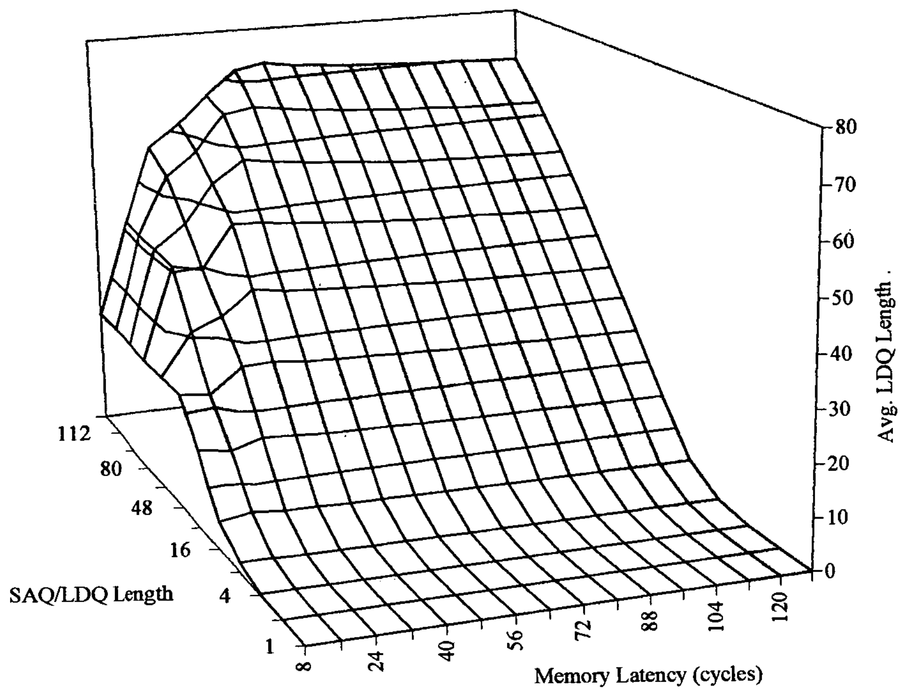


Figure 52 - Modelled LDQ length for all three runs (Ex. 4).

It is difficult to see how this interaction between the two streams of memory accesses can be modelled simply. It requires that individual accesses be ordered in some way (i.e. "generate two loads then a store, then another two loads, then another store") and, while this is easily done with a simulator, producing a queueing network to do so seems a non-trivial task.

This result severely restricts the area in which the model can be effectively used. If there is a strong dependency between the ordering of the load and store streams, the model becomes inaccurate. There will always be some degree of dependency due to instruction ordering in the system being modelled, but when the system parameters are such that this ordering can be violated frequently (i.e. regular store stalls), the discrepancy will increase. Thus, while the model remains useful for predicting the behaviour of well-behaved systems, it is liable to become inaccurate when faced with a poorly configured system.

The examination of a number of other measures of system behaviour, such as LDQ length, strengthens this hypothesis. Figure 52 shows the average LDQ length for all three runs. Once again, the third run tracks the first two closely (it differed in the simulator) due to the ordering problem. The differences between this graph and those produced by the simulator may also be due to this problem, with the removal of loads from the LDQ happening more rapidly due to their not being blocked by stores. However, it is difficult to be sure if this is in fact the case, since LDQ behaviour in the model differs from that in the architectural simulator as has already been mentioned.

5.1.11 Conclusions

This experiment has two important results. The first is that, contrary to the results of the previous experiment, the presence of the SAQ is critical. This was not apparent in Experiment 3 because the SAQ only becomes useful when both loads and stores are present in a program's memory traffic. The SAQ's function is to prevent loads from being stalled due to an uncompleted store. In other words, the SAQ acts as a write buffer.

Without an SAQ the system performs poorly. No address sub-processor store instruction can complete until the data sub-processor provides its half of the store operation. This causes the two units to become synchronised and severely restricts the degree of decoupling.

The experiment also showed that by failing to take account of the ordering of loads and stores explicitly, the model which had been developed was unable to predict the behaviour of a system with no SAQ. Fortunately the presence of an SAQ, in permitting loads to overtake stores and vice versa negates this weakness in the model and the model's estimates of program runtime were reasonably accurate for systems with an SAQ.

5.1.12 Access/execute decoupling - a summary

Through these experiments a number of important conclusions about access/execute decoupling can be drawn. Perhaps the most important of these is that, when given a simple stream of loads, even a poorly configured decoupled architecture performs at least as well as a conventional architecture with no data cache. When the decoupled architecture is configured in such a way that full decoupling can occur, and when the system is well configured, the performance of the (cacheless) architecture is at least as good as that of a conventional uniprocessor with a cache.

In the above paragraph the phrases "poorly configured" and "well configured" were used. The exact meanings of these, at least in the context of access/execute decoupling, were determined by the above experiments. To perform efficiently, the decoupled processor must not be underbanked or undertagged. What constitutes overtagged or undertagged, overbanked or underbanked, is determined by a number of factors - the rates at which the address and data sub-processors are capable of generating and removing memory access packets from the memory sub-system and the total bandwidth and capacity (that is, the number of simultaneously active memory requests) in the memory sub-system.

An interleaved memory is considered to be underbanked if there are insufficient memory banks to cope with the maximum load under which the memory as a whole is likely to be placed. Typically, in a system with a maximum memory load of l (l memory packets are sent to memory on every processor clock cycle) and a bank cycle time of n processor clock cycles, we require at least $l \times n$ memory banks in order to avoid being underbanked. The actual value of l for a particular program will vary depending upon the architecture it is running on and the efficiency of the compiler used to compile it, but in the artificial example used above it is held at 1. That is, on every cycle either a load or a store is added to the memory subsystem. Although a load or a store can potentially be dispatched to memory in the same cycle (one from the LAQ, one from the SAQ), only one can actually be added to the memory queues in a single cycle, so although the load on memory may reach two on occasion, over the entire duration of the program's execution it will have an average value of 1.

Due to the way in which decoupled loads are re-ordered on arrival at the LDQ, the total number of loads in transit between the address sub-processor and the data sub-processor at any one time is limited to s_{LDQ}, the maximum size of the LDQ, since each in-transit load must be assigned a tag that records the position in the LDQ that it is to be stored at. If the number of available load tags is sufficiently small that the full available memory bandwidth cannot be used, performance will suffer. The best performance is obtained when s_{LDQ} is greater than n and it is possible for all memory banks to be simultaneously active. In practice the point at which a system ceases to be undertagged may be slightly lower than n due to the difference between memory cycle times and memory access times, and also due to the presence of store operations, which keep banks busy without requiring tags. Some aspects of system performance will continue to improve as the "memory pipeline" fills. In the sub-system used in these experiments, the capacity of the memory pipeline is determined by the number of banks n and the size b of the buffer attached to each bank. In this case, the memory pipeline can contain up to $n \times b$ active memory requests at any one time.

Many of the performance graphs show a distinctive shape, with a noticeable shift in behaviour as the system crosses the boundary from being undertagged ($s_{LDQ} < n$) to overtagged ($s_{LDQ} > n$). Performance may continue to improve as the degree of overtagging increases, until the total number of load tags equals the capacity of the memory pipeline ($s_{LDQ} = n \times b$) at which point increasing the LDQ size has no effect. This is due to the saturation of the memory pipeline, which means that no new memory accesses can be added until older ones are removed. No matter how many tags are available, the number of active loads cannot exceed $n \times b$.

It was also shown that neither the Load Address Queue or Store Data Queue were necessary.

The Store Data Queue, on the other hand, is redundant because, should the system be properly decoupled, any store data will immediately be paired with the address half of the store packet and dispatched to memory. The only circumstance where this will not happen is if the memory pipeline is saturated. If this happens frequently, then the memory system is underspecified and delays of this nature have to be expected. The presence of an SDQ would do little more than provide a short reprieve against an inevitable stall, in much the same way that the LAQ does on the address sub-processor. In any event, the brief stalls that may occasionally occur due to short-term stalls of the memory pipeline are, in the overall scheme of things, unimportant and have little effect on performance.

Experiment 3 gave some indications that the SAQ was also redundant. Further investigation of this led to the interesting discovery that the SAQ was important, but only if the memory workload consisted of a mixture of loads and stores. The SAQ was of no benefit whatsoever to store latency and had no direct effect on decoupling or the performance of the data sub-processor, but instead affected these indirectly by preventing subsequent loads being held up. So, somewhat surprisingly, the SAQ's presence is important to decoupling through its effects on decoupled loads rather than on stores.

5.2 Control Decoupling

5.2.1 Experiment 5: The SAXPY loop again

In Experiment 4 the behaviour of a fully unrolled SAXPY loop was examined. In practice, fully unrolling a 100,000 iteration loop would be somewhat memory inefficient and, on a control decoupled architecture, the loop would be implemented in a manner similar to that shown in Figure 53.

```
Control Sub-Processor      Address Sub-Processor           Data Sub-Processor

    ISSUE A0, E0           A0  LD a1, ADDR(x)              E0  LD r1, #a
    LD r1, #1                  LD a2, ADDR(y)              E1  LOAD r2
L1  ISSUE A1, E1               LD a3, ADDR(z)                  MUL r4, r1, r2
    ADD r1, r1, #1             LD r1, #0                       LOAD r3
    CMP r1, #bound         A1  LOAD r1 + a1                    ADD r4, r3, r4
    BLE L1                     LOAD r1 + a2                    STORE r4
    END                        STORE r1 + a3
                               ADD r1, r1, #wordsize
```

Figure 53 - Fully decoupled pseudocode for SAXPY loop.

This uses a method of control decoupling that is slightly impure when compared to Bird's original action semantics model, whereby the incrementing of the loop index variable and its comparison with the loop bound are performed on the control sub-processor. Although this marginally increases the workload for this unit, it removes the need for the control sub-processor to synchronise with the data unit on each iteration to transfer the result of the comparison from the DSP to the CSP.

In this experiment the execution of a program similar to that shown in Figure 53 is simulated on a variety of related architectures. Three simulation runs were performed. In each run certain features were kept constant. The LDQ length was held at a constant value of 512 packets while the LAQ was set to length 1, effectively removing it. The memory sub-system was configured as 32 segments, each containing four banks. Hence, the system becomes undertagged if the number of potential active loads exceeds 512 and is underbanked if the memory latency exceeds 128 processor clock cycles.

For each simulation run, two parameters were varied. These were the size of the meta-instruction queue, which ran from 8 to 64 meta-instructions in increments of 8, and the memory bank latency,

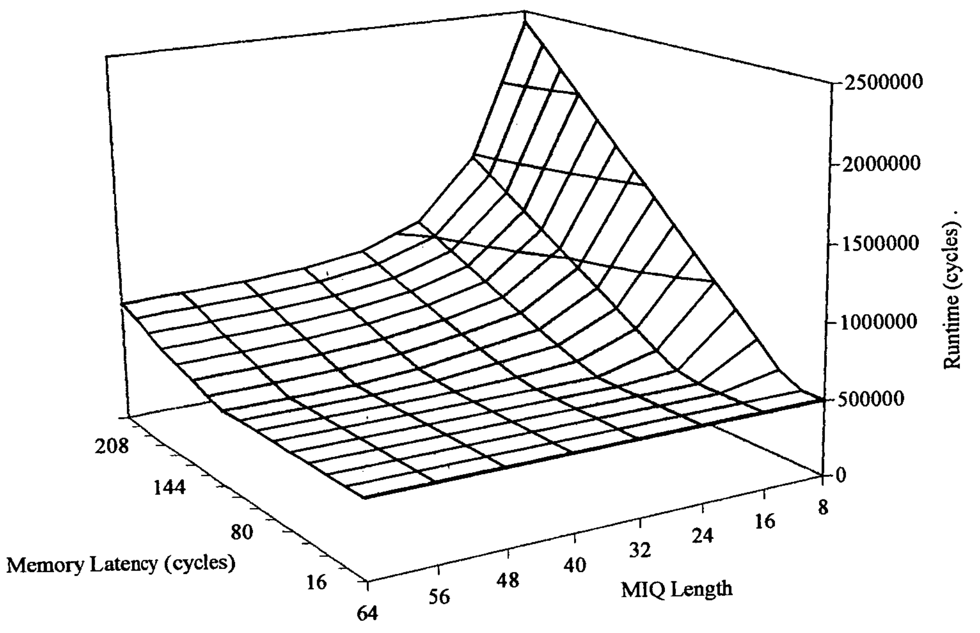


Figure 54 - Data sub-processor runtime for first and second runs. (Ex. 5)

which was varied from 16 processor clock cycles up to 256 processor clock cycles in increments of 16.

The three runs differed in the lengths of the SAQ and SDQ. This was designed to test whether the known effects of varying the lengths of these queues (covered in earlier experiments) had an effect on control decoupling. During the first run, the SAQ and SDQ were both set to length 512. If the system behaved as expected, this should not have differed appreciably from the second run, which removed the SDQ while holding the SAQ at the same length. The third run, which removed the SAQ as well, was expected to have an effect on performance but it remained to be seen what effect this would have on control decoupling.

The fully decoupled program shown in Figure 53 has potentially unbounded control decoupling, since no information has to be passed back from the work processors to the control sub-processor. However, as with access/execute decoupling there is a practical bound on the degree of decoupling available. For access/execute decoupling this was found to be the smaller of the capacity of the

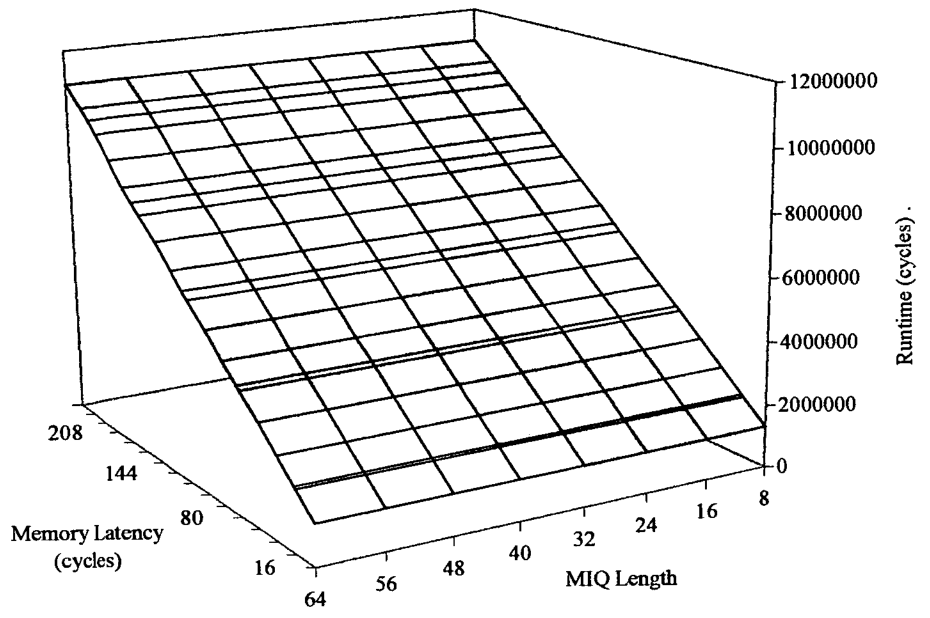


Figure 55 - Data sub-processor runtime for third run. (Ex. 5)

memory pipeline or the size of the LDQ. It would be expected that MIQ size limitations would have a similar effect on the amount of control decoupling available.

As usual, the most obvious measure of the effects of varying the MIQ length on behaviour is to look at the program runtime. The runtime for the data sub-processor is used since, unless the data sub-processor returns a value to the control sub-processor as its last action, this will always be the last sub-processor to finish.

Figure 54 and Figure 55 show the simulated data sub-processor runtime for the first and second runs and the third run respectively.

Looking first at Figure 54, the two runs are shown together as there is no difference in their runtimes. This appears to confirm the results of Experiment 4, that the SDQ is redundant, while extending it to show that the presence or absence of the SDQ has no secondary effects on control decoupling.

Performance seems to be strongly affected by both memory latency and the length of the MIQ. The reasons for the former are already well-known, and the fact that runtime seems to increase once the memory sub-system becomes underbanked (for reasonable MIQ lengths, at least) should come as no surprise. However, the reasons that cause the MIQ length to affect the total runtime are rather more complex.

For a process to become decoupled, the "producer" must go faster than the "consumer". In this case, the producer is the control sub-processor and there are two consumers, the work units. Speed is measured, for the control sub-processor, by the rate at which new meta-instructions are placed in the MIQ and for the work units by the rate at which new iterations of the loop begin or the rate at which meta-instructions are removed from the MIQ.

As can be seen from the code in Figure 53, the control sub-processor main loop takes four processor clock cycles to initiate (issue, increment, compare, branch) and thus a new meta-instruction is generated on every fourth cycle. The address sub-processor loop body also takes four clock cycles to initiate, while the data sub-processor takes five cycles. Thus, the address sub-processor and control sub-processor fail to decouple while the control sub-processor and the data sub-processor can potentially decouple by one instruction every iteration.

In practice, however, the degree to which the control and data sub-processors can become decoupled is limited by the size of the MIQ. Once the MIQ has filled, the control sub-processor is only able to add a new meta-instruction to the queue when the data sub-processor fetch engine removes one from the other end of the queue. Once this happens, the control sub-processor loop iteration time increases to match that of the data sub-processor. In this case, it means that the control SP now takes five cycles per iteration, with each attempted write to the MIQ resulting in a single cycle stall.

However, the limited size of the MIQ does not just effect the amount of control decoupling that is available. When the control sub-processor executes an issue instruction, it attempts to place meta-instructions in the queues for both of the work sub-processors. If one or both of the MIQs is full, the operation will stall until there is space available in both queues.

In the SAXPY example used in this experiment, the address and control sub-processors fail to decouple due to their producing and consuming meta-instructions at the same rate. This means that the address sub-processor MIQ remains empty, with each meta-instruction being immediately expanded by the fetch engine as it arrives. However, when the data sub-processor MIQ fills, this causes the control sub-processor to stall, starving *both* of the work units even though the address SP has an empty MIQ.

Once this state has been reached, a new meta-instruction will only arrive for the address sub-processor when one is removed from the data SP's queue, thus allowing the control SP to proceed. The address SP iteration time increases to five cycles, matching that of the data and control units. Up until this point access/execute decoupling has been possible, since the address sub-processor has been initiating loads more quickly than the data sub-processor can accept the arriving data (two loads initiated every four cycles versus two loads removed every five cycles). When the two units become locked together like this, the degree of access/execute decoupling can increase no further and will at best remain at the same level it had reached prior to the data sub-processor's MIQ filling up.

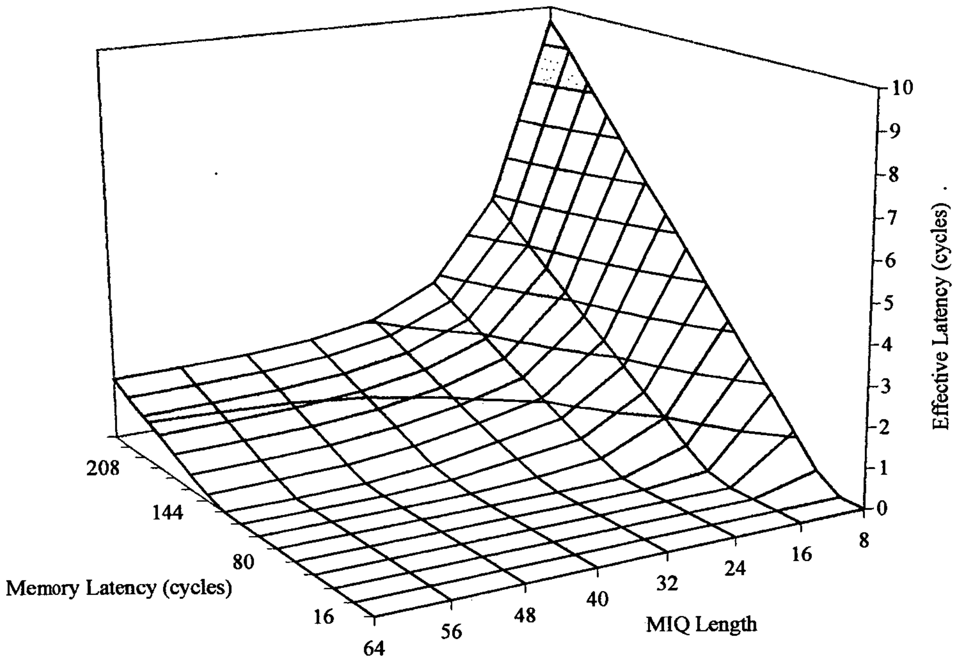


Figure 56 - Data sub-processor effective latency for first and second runs. (Ex. 5)

The results of this can be seen in Figure 56, which shows the effective latency as perceived by the data sub-processor for the first and second runs of the experiment. When the MIQ length is short, the various sub-processors become locked together fairly rapidly, limiting the available decoupling. This means that the full memory latency cannot always be hidden and the resulting drop in latency tolerance causes the effective latency (and thus the overall runtime) to increase. As the MIQ length increases so does the memory tolerance, and the point at which the latency tolerance is lost increases with the MIQ length. However, in all cases here the effective latency becomes non-zero for memory latencies greater than around 150 processor clock cycles, this being the point at which the system becomes undertagged.

As can be seen in Figure 55, the third run behaves very differently. In this run there is no SAQ or SDQ. As was shown in the previous experiment, this prevents the data and address sub-processors from decoupling fully. The graph shows clearly that in the absence of an SAQ, the limitations on access/execute decoupling dominate and the length of the MIQ has no effect on performance, since the system as a whole is in the "locked" state from the very beginning. Thus the data and address sub-processor runtimes are unvarying for any given latency, regardless of the MIQ length. The control sub-processor runtime varies slightly depending on the MIQ length, but the variation is accounted for by the time taken for the MIQ to fill and the control sub-processor to become locked to the other two sub-processors. When the MIQ is longer, it takes slightly longer for this to occur, but given that this difference amounts to at most a few thousand cycles over a total runtime of over 10,000,000 cycles, it is negligible.

5.2.1.1 Increasing The Available Decoupling

In the above experiment it was shown that the length of the MIQ has a direct effect on the degree to which the available access/execute decoupling can be exploited. In the case of the SAXPY loop, the available decoupling is potentially infinite (or, at least, bounded by the total number of loop iterations) but in practice is bounded by the number of meta-instructions that can be held in the MIQ. Since each meta-instruction corresponds to a single iteration of the SAXPY loop, the address sub-processor cannot run more than *n* iterations ahead of the data sub-processor, where *n* is the length of the MIQ.

This limitation can be largely removed by modifying the meta-instruction format and providing a more powerful instruction fetch engine. If each meta-instruction has an additional field, specifying an iteration count for the block being issued, the bottleneck imposed by the meta-instruction queue is removed. Even if the iteration count was restricted to a 16 bit value, the entire 100,000 iterations of the SAXPY loop specified above could be issued with just two meta-instructions. A 32 bit iteration count would allow almost any loop that might feasibly be encountered to be issued with just a single meta-instruction.

In addition to requiring an extra meta-instruction field for the iteration count, providing a loop mode would also entail modifying the instruction fetch engine on each work unit to repeatedly fetch the same block until all the loop iterations had been issued. This would be relatively straightforward, but may cause complications when used with the instruction pre-fetch mechanisms described in Chapter 8.

By way of an illustration of the performance gains that could be achieved through the use of a loop-mode issue, the experiment was re-run, this time simulating the behaviour of an architecture with a loop mode. The MIQ length and memory latency were varied as before. Figure 57 shows the runtime that is achieved through the use of a loop mode in a system where LDQ and SAQ are 512 elements in size, and there is no LAQ or SDQ.

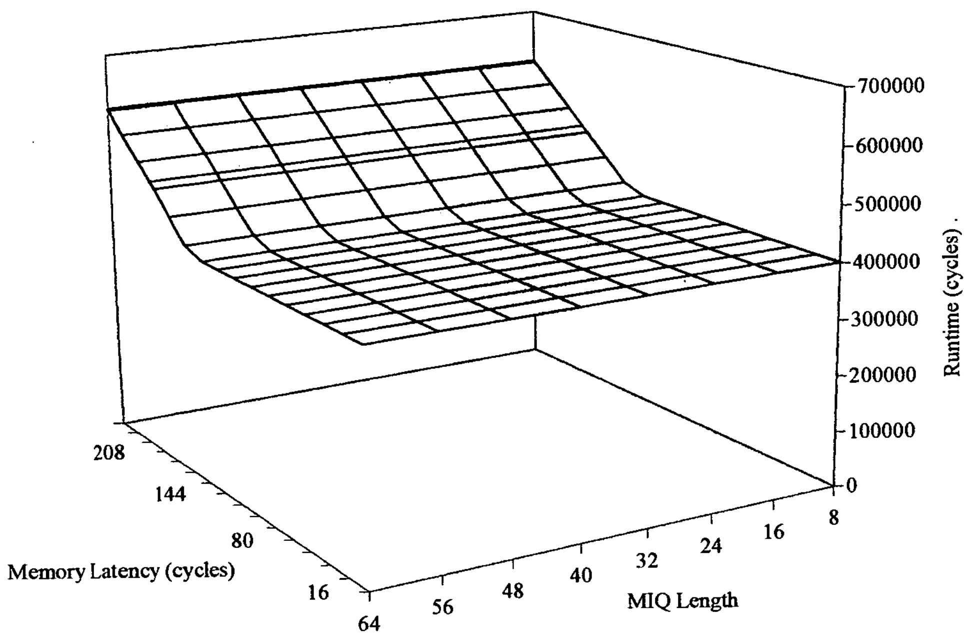


Figure 57 - Data sub-processor runtime with hardware loop mode. (Ex. 5)

As expected, the performance of the loop mode machine is unaffected by the MIQ length, since the entire loop is issued by a single meta-instruction. The total runtime remains constant while the machine is overbanked and begins to increase slowly once underbanking sets in. However, even when the system is overbanked performance is superior, due to the system's ability to fully exploit the available access/execute decoupling.

6. Losses Of Decoupling

6.1 Losses Of A/E Decoupling

6.1.1 Causes

Access/execute decoupling relies on the natural "flow" of data from the address sub-processor to the data sub-processor. The vast majority of all communications between these two are memory accesses. Loads begin on the address sub-processor and end on the data sub-processor, while there is no obvious ordering between the address and data halves of a store operation. Since the addressing half of a load must always occur before the data half, access/execute decoupling endeavours to move the addressing phase of a load as far ahead of the data use phase as is possible. Most of the time this works well, with the distance between the two being such that the memory latency is invisible to the data sub-processor.

However, from time to time it is necessary to pass information against the flow of data, from the data sub-processor back to the address sub-processor. This can happen for a number of reasons. The majority of variables in a program can be split into one of two categories - data variables and address variables. Most fall into the first category with only a few types (such as array indices and pointers) falling into the latter. When a program is compiled for a decoupled architecture, address variables will be placed on the address sub-processor where possible, and data variables will be placed on the data sub-processor. There will be some cases, however, where a variable is *both* a data variable and an address variable. For example, the array reference $b[i+v1 \times v2]$ accesses variables $v1$ and $v2$ which are used in (or are the result of) calculations on the data sub-processor. When the time comes to obtain their product, add it to the address variable i and use this as an array offset to access the data variable $b[i+v1 \times v2]$ the values of $v1$ and $v2$ must be transferred from the data sub-processor to the address sub-processor, against the flow of data. A similar situation can be encountered when a program is attempting to follow a linked list, or if an array is being used to provide indices into a second array. In the first of these cases, large numbers of loss of decoupling events may be caused as the processor attempts to follow the list.

A third type of A/E decoupling event results only if the architecture does not provide the address sub-processor with a direct path to memory. In this case the address sub-processor can only load data by moving the data (via decoupled load) to the data sub-processor and then transferring it (via the TRQ). Stores are handled in a similar manner.

All of the above situations where data must be passed against the flow result in loss of decoupling events, the effects of which can be serious.

6.1.2 Effects

A loss of decoupling event temporarily (for the time taken to transfer a single datum) reverses the direction of data flow within the system. It switches from a state where the address sub-processor is running ahead of the data sub-processor to one where the reverse is true. If the address sub-processor has become heavily decoupled, the cost of this will be high. It may take several hundred, or even several thousand, processor clock cycles for the data sub-processor to catch up to and overtake the address sub-processor. During this time the address sub-processor is idle, waiting for data to be transferred from the other unit. In addition to this "catch-up" delay, all decoupling will be lost, and the memory sub-system and queues will be drained. This means that even once the LOD has passed it will take some time for the system to reach peak efficiency again. If loss of decoupling occurs frequently (for example, in a loop) then the memory sub-system and queues may not even manage to reach a reasonable level of efficiency before if is forced to recouple. This may limit the performance of the memory sub-system to a fraction of its efficiency.

By optimizing for a particular direction of data flow, access/execute decoupled architectures pay a heavy price on those rare occasions that data must travel in the other direction and performance will compare poorly with even a conventional architecture. The key to retaining a reasonable level of performance on an A/E decoupled architecture is to minimize the frequency with which these events occur.

6.1.3 *Solutions*

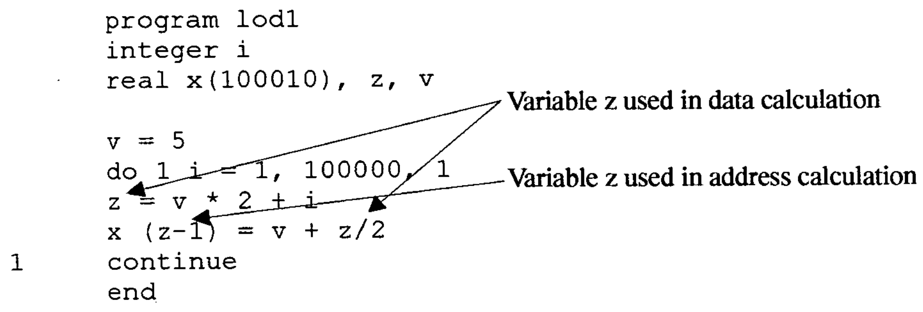
There are two main avenues that can be explored in the elimination of loss of decoupling events. Software based solutions rely on code restructuring and optimization to either eliminate LODs entirely or move them to where they will have less effect. Hardware solutions enhance the architecture itself to bypass situations in which LODs may occur.

6.1.3.1 Code replication

In an ideal world, the placement of code on a fully decoupled architecture would follow Bird's original action semantics based model exactly, with only control flow calculations on the control sub-processor, only binding of variables to addresses on the address sub-processor, and so forth. Unfortunately, the strict implementation of such a model leads to frequent loss of decoupling events, with a resulting loss of performance. The most theoretically pure model for code placement on a fully decoupled architecture requires a very strict partitioning of operations over the three sub-processors into the three aspects of action semantics - control, binding, and computation. The only instructions on the control sub-processor should be operations such as branches and jumps, tests on values returned from the work units, and the issuing of basic blocks to them. Similarly, the address sub-processor should perform arithmetic only in connection with the calculation of memory addresses. However, while this mapping of operations to processors is very *pure* with regard to the semantically-driven model, it neglects the potentially high cost of LODs.

At the other extreme, we can imagine an architecture with a highly capable and largely conventional control processor. The compiler initially assumes that all code should be executed on the control SP and then migrates suitable "safe" code segments to the work units. A code segment is considered safe if it does not cause any losses of control decoupling. The most frequent example of a safe code segment is an innermost loop with no internal control flow or decision making, such as the SAXPY kernel. This code placement algorithm abandons the semantically-driven model and instead treats the work units as a powerful programmable vector co-processor, to which selected tasks can be dispatched for execution. It is much closer to a conventional programming model than the semantically-driven approach. A third, intermediate approach is to try to distribute work across the three processors in a semantically-driven style but to selectively replicate calculations on multiple sub-processors where necessary so as to remove losses of decoupling.

To give an artificial example, take a statement like $x(y+z)=a+2z$. In this statement, the variable z is required by the data sub-processor to determine the value that must be stored, and by the address sub-processor to determine the memory location that the value should be stored at.



```
          program lodl
          integer i
          real x(100010), z, v

          v = 5
          do 1 i = 1, 100000, 1
          z = v * 2 + i
          x (z-1) = v + z/2
1         continue
          end
```

Variable z used in data calculation
Variable z used in address calculation

Figure 58 - Kernel code. (Ex. 6)

The pure semantically-driven approach would calculate z on the data sub-processor then pass it to the address sub-processor (causing a loss of decoupling) for use in the address calculation before calculating the rest of the expression and storing the result to memory. The conventional "co-processor" approach would consider the loop safe (the only loss of decoupling is between the data and address units) and migrate the loop body to the work units. The iteration control (maintaining the loop index variable and testing it against the loop bounds) would remain on the control sub-processor, which would issue the loop body once on each iteration of its own "local" loop.

In the intermediate approach, both the data and address sub-processors would perform the calculation of z resulting in two copies of the variable. While this would increase the amount of work being performed on the address sub-processor, it is highly unlikely that the additional redundant calculation would take longer than the loss of decoupling that would otherwise result.

All three of these approaches are examined in Experiment 6. For this experiment, the artificial kernel shown in Figure 58 was used. This kernel was designed to require the sharing of a variable between the data and address sub-processors in such a way that all three of the code placement strategies mentioned above could be tested. The actual operations performed on the data were unimportant, what is relevant here was their placement.

A total of five runs of the experiment were made. The first and third of these used a very simple control decoupling model where the control unit recouples with the work units after each iteration, while the second used no explicit model of control decoupling, since the loop control flow is automatically placed on the control processor under the co-processor approach.

However, the lack of control decoupling in the first and third runs that resulted from the control decoupling model used was likely to severely inhibit the degree of access/execute decoupling available and thus adversely affect performance. The three experiments were thus re-run using a more flexible control decoupling model, where loop bound tests were made on the control sub-processor rather than returned from one of the work units. In the case of the co-processor model, where this occurred anyway, a second test was made to examine how the model would have performed had, for example, the loop been considered unsafe and thus unsuitable for migration to the work units. Only five runs needed to be made because the placement that resulted from the co-processor model and that which resulted from implementing the pure semantically-driven approach in a model with loop iteration handled by the control sub-processor were exactly the same.

The five runs were thus:

1. Semantically-driven model, array bounds tests on data sub-processor.

2. Co-processor model, loop body considered safe to migrate.

Semantically-driven model, array bounds tests on control sub-processor.

3. Intermediate model, array bounds tests on data sub-processor.

4. Co-processor model, loop body considered unsafe to migrate.

5. Intermediate model, array bounds tests on control sub-processor.

The memory latency on all five runs was varied from 16 to 256 processor clock cycles, in increments of 16 cycles. On those runs where the TRQ was used (the first two), the length of the TRQ was varied from 8 to 128 elements, in increments of 8.

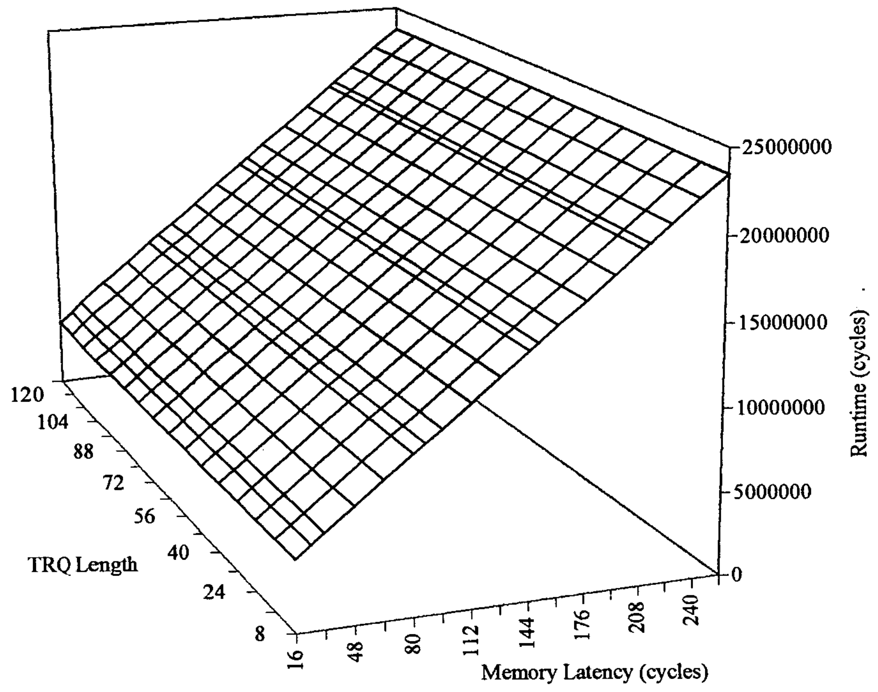


Figure 59 - Total runtime for first run. (Ex. 6)

Figure 59 shows the total runtime for the first run. That for the second run is similar in form, but is lower by a constant value.. The most noticeable features of this graph are the high runtime and the apparent independence of performance from TRQ length. Both of these are due to the repeated losses of decoupling between the address and data sub-processors. This makes the full memory latency visible on the data sub-processor on every iteration, as well as limiting the utilization of the TRQ to a single datum at a time. The difference in runtimes between the two runs is due to the first run causing both control and access/execute decoupling to be lost, while the second run features only the latter variety of LOD. One might expect that two losses of decoupling on each iteration would result in significantly inferior performance when compared with a single LOD, but the access/execute LOD masks most of the effects of the control LOD, resulting in only a small increase in total runtime.

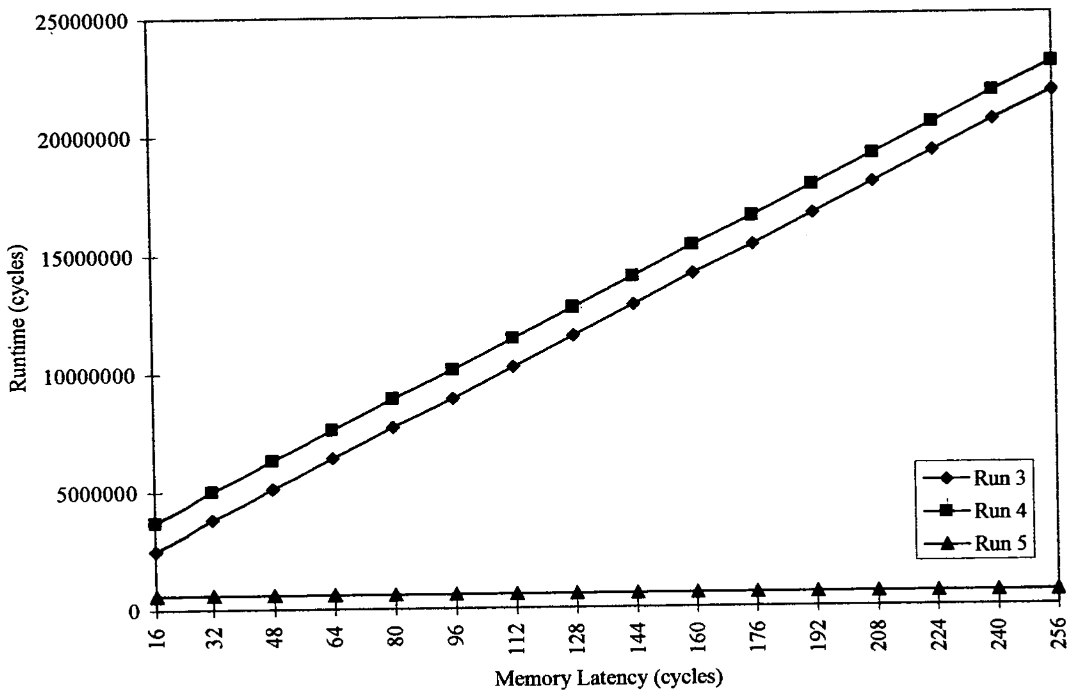


Figure 60 - Total runtime for runs 3-5. (Ex. 6)

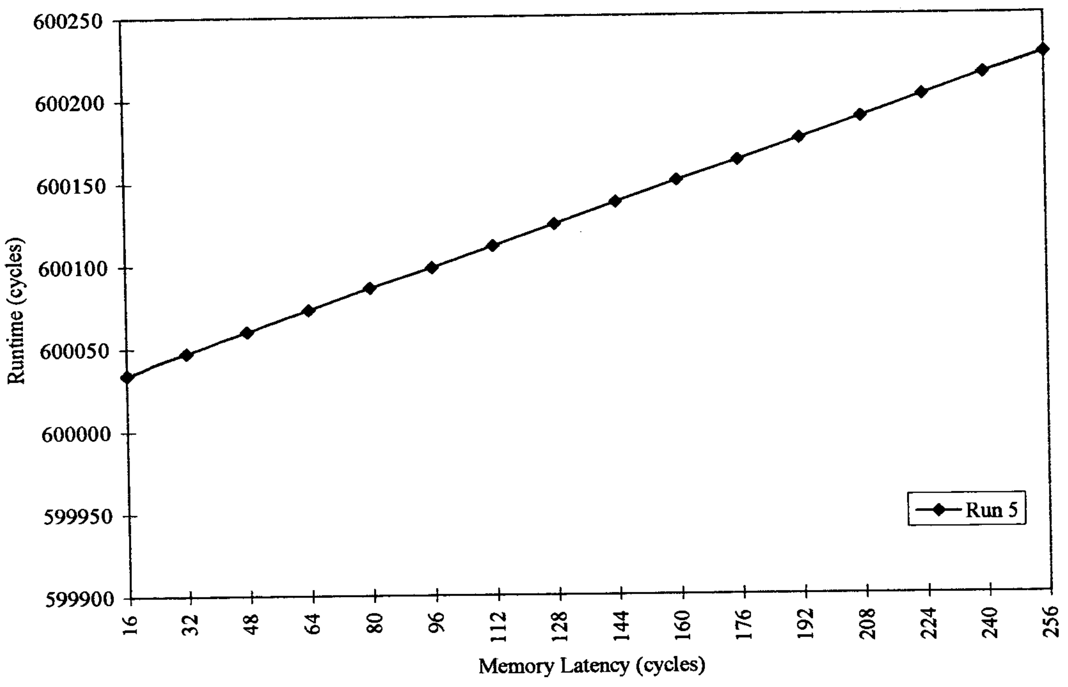


Figure 61 - Total runtime for run 5. (Ex. 6)

The remaining three experimental runs were made over a one-dimensional data space, varying the memory latency as before. There was no point in varying the TRQ length as was done in the first two runs, since none of these runs actually made use of the TRQ.

The relative runtimes of runs 3-5 are shown in Figure 60, and the runtime for run 5 is also shown separately in Figure 61, since the scale of Figure 60 makes the runtime for that run appear constant.

What is immediately apparent from Figure 60 is that, while the runtimes for runs 3 and 4 are of comparable magnitude, the runtime for the fifth run is considerably lower. Dealing first with the runtimes for runs 3 and 4. While both demonstrate a similar gradient (which in this case indicates that both runtimes are determined largely by the memory latency), the fourth run is appreciably faster, requiring approximately 1,000,000 less processor cycles to execute. This is due to the loss of control decoupling that occurs on every iteration in the third run. While use of the intermediate code placement model has eliminated the access/execute LOD that occurred in run 1, there is still a control decoupling on every iteration when the data sub-processor returns the result of the array bounds test to the control sub-processor. This control LOD in turn inhibits access/execute decoupling, which makes the full memory latency visible.

In the fourth run the entire program kernel is executed on the control sub-processor, with memory accesses using the control sub-processor self-load operation. There are no losses of decoupling since no decoupling occurs, and both work units remain idle throughout. The lack of a data cache on the control sub-processor means that once again, the full memory latency is visible on each access. The difference in runtime between runs 3 and 4 can be accounted for by the cost of the control LOD on the third run and also by any differences in the iteration time between the code that is run on the address/data sub-processors during run 3 and on the control sub-processor during run 4.

The fifth run shows demonstrably different behaviour. Indeed, in Figure 60, the runtime for run 5 appears to be independent of memory latency. Figure 61, which shows the run 5 runtime on a more useful scale, shows that there is indeed a small variation of runtime with memory latency. However, this is negligible. The combination of loop bound evaluation on the control sub-processor and the eplication of code over the data and address sub-processor eliminates all loss of decoupling events. Performance here is limited only by the available control decoupling and memory latency. The small variation in runtime with memory latency shown in Figure 61 is due only to the startup time of the memory pipeline - the delay before the first load packet arrives at the data sub-processor from memory. These runtime figures are supported by Figure 62 which shows the effective load latency as perceived by the data sub-processor (or in the case of run 2, the control sub-processor). The effective latency for the second and third runs shows clearly that the entire memory latency is visible, with the small difference between the effective latency in the two runs being due to the slightly longer memory path for a decoupled load. By way of a contrast, the effective latency for run 5 is a constant two clock cycles. The effective latency is non-zero because of the relative loop iteration times on the data and address sub-processors, but remains constant regardless of the actual memory latency.

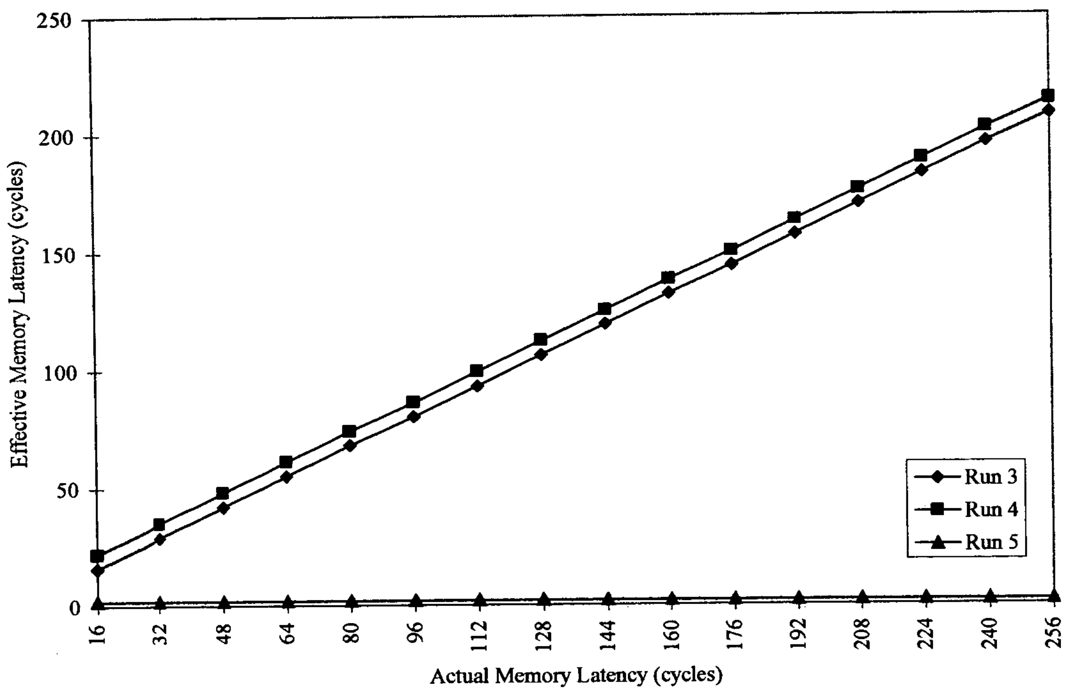


Figure 62 - Effective latency for runs 3 to 5. (Ex. 6)

The results of this experiment show clearly the effects that loss of access/execute decoupling can cause. In this case, the entire memory latency is made visible, since the two units are unable to decouple by any noticeable amount before decoupling is lost. It is also shown that for any of the ifferent code placement base solutions tested, the absence of both address/execute and LOD events vas required. For example, the code replication technique used in the intermediate placement model nd tested in runs 3 and 5 was only effective in the fifth run, when combined with the migration of he loop bound test to the control sub-processor, thus eliminating the control LOD. A comparison of he various experimental runs also makes obvious the great performance improvements that can be btained when LODs are eliminated (or, from a more pessimistic point of view, the huge penalties mposed by LOD events).

This experiment illustrates the decoupling problems that can occur when data has to be shared by the two work units because a "data" variable is used in an "address" calculation. In this case, the dependencies between the two units were reduced somewhat by the assumption that the code being placed was sufficiently optimized that all scalar variables could be held in registers and were only loaded from memory once, before the loop was entered. However, this is not always possible. An array structure in memory might hold not data, but the addresses of data. This approach is common in the implementation of sparse data structures, where it would be impractical to store an entire matrix in memory, or in particle-in-cell problems where an array of "particles" records their position in a much larger matrix representing the space in which the particles are moving . Access to these structures requires that an entry in the "index" is examined, yielding either an index (or set of indices) into the second structure, or a pointer to another data structure in memory. Figure 63 shows a simple Fortran loop that uses indirection of this type.

```
          tot = 0                     Indirection
          do 10 i = 1, 100000 ▲
             tot = tot + z(x(i))
10        continue
          end
```

Figure 63 - Indirection in a Fortran loop.

Loss of decoupling problems arise because the data stored in this index is in fact address information, or at least is going to be used in the calculation of an address. Each element of the index array must in some way be moved to the address sub-processor and then used to access the actual data on which e data sub-processor is to operate. In this experiment, two ways of doing just this are examined. he first is a hardware technique, the second is software-based.

.1.3.2 Address SP self-load and self-store operations

Many LOD events occur because the address processor requires data that it cannot access directly tself and must instead fetch via the data sub-processor, causing a loss of decoupling in the process. f, however, the address sub-processor has some means of accessing memory directly without the co-operation of the address sub-processor, this can be avoided. This is one of the features of the Level 2 architecture described in Section 4.3.3.2. The instruction set of the address sub-processor is extended to include operations that allow the address sub-processor to load and store from memory directly. In this architecture the main loop (on the address sub-processor) of the example program described above would consist of a self-load to access the index array, followed by a decoupled load for the data access itself. No loss of decoupling would occur. This approach is not without its disadvantages however. It requires that the address sub-processor must have, in some form or other, two paths to memory, one for decoupled accesses and one for self-loads and self-stores. These may be implemented as a single physical path, with one form of load taking priority over the other, or as two separate paths, giving higher bandwidth to memory at the expense of a considerable extra cost in hardware.

The use of self-loads re-introduces many of the memory access problems that decoupled access/execute architectures were designed to remove. Long memory latencies can once again adversely affect performance, and as will be shown in this example, the absence of a cache on the address sub-processor's memory path can cause severe performance problems, with the full latency being visible on every self-load.

6.1.3.3 Strip Decoupling

Strip decoupling is a restructuring technique that, in certain circumstances, can be used to rewrite code that uses limited degrees of indirection in such a way that the effects of the resulting losses of decoupling are hidden.

The most basic requirement of strip decoupling is that the code to which it is to be applied has no loop carried dependencies. This effectively rules out the use of the technique for list following. However, it may be useful in cases where, for example, an index array is used to access a larger array. The technique attempts to apply some conventional latency hiding techniques to the address sub-processor's access pattern, by separating the initial access to the pointer (or index) from the actual use of that data by as much time as possible. Hopefully, by the time the address sub-processor actually needs the pointer or index it will have been placed in the transfer queue by the data sub-processor. In effect, strip decoupling is an attempt to perform execute/access decoupling, where the data sub-processor is temporarily running ahead of the address sub-processor. Just to make matters particularly tricky, the address sub-processor must be able to run ahead of the data sub-processor *at the same time*. It may seem to the casual observer that some minor problems, such as causality, might make this technique difficult to apply in practice. Strip decoupling avoids this problem by splitting the indirection into two separate phases. In the first phase, all of the initial accesses (fetching the pointers) to the array of indices (or analogous structure) are made. During this phase, the data sub-processor is largely idle and the only work it performs is to transfer each pointer arriving in the LDQ into the TRQ. During the second phase, the address sub-processor removes this data from the TRQ and uses it to perform the indirect access while the data sub-processor performs the "real" execute task. To obtain reasonable performance using this technique, a number of criteria must be met. In addition to there being no loop-carried dependencies, there must also be sufficiently many loads required that by the time the address sub-processor has finished executing the first phase, the data sub-processor has begun to copy the loaded pointers from the LDQ into the TRQ. Additionally, the second phase should not attempt to fetch data from the TRQ more quickly than the data sub-processor can place it there. Given the minimal nature of the data sub-processor's task during the first phase, this seems likely to occur only if the two units are unable to decouple properly on the first phase.

The strip decoupling technique lends itself most readily to loops where there are a large number of single indirections and the indirections are independent of each other. If this is the case, a loop of *n* erations can be split into an outer loop of $\frac{n}{i}$ iterations and two inner loops of i iterations each. These two loops implement the corresponding phases of the strip decoupling technique and it is the form of this nested loop, where a single dimensional loop is processed as a series of strips rather than n one go, that gives the technique its name. There is no reason why the technique could not be extended to multi-dimensional arrays, as long as there are no loop-carried dependencies. It should also be possible to extend strip decoupling to cope with multiple indirections (using multiple phases) as long as the chains of indirections are finite. However, as the number of indirections increases, the amount of time wasted copying values from one queue to another by the various sub-processors will increase, reducing efficiency. Figure 64 illustrates the behaviour of strip decoupling for part of the execution of a one-dimensional loop with a single indirection. The two phases of the inner loop for the address sub-processor take the same time to execute in ideal circumstances (when there are no stalls), while the data sub-processor's phases are of different lengths. The first is shorter, reflecting the simple copy task that the processor is performing, while the second is longer, reflecting the computation being performed on the indirected data. The arrows on the diagram illustrate the flow of data between the two units.

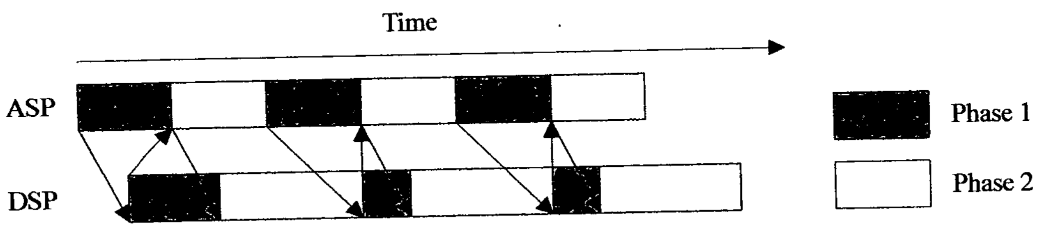


Figure 64 - Strip decoupling.

The address sub-processor begins its first strip and, after a period of time determined by the memory latency, data starts to arrive at the data sub-processor. This is immediately re-routed into the TRQ. Since the address sub-processor is still busy when the data sub-processor enters its first phase, the TRQ will begin to fill. Once the address sub-processor has finished the first phase, it immediately begins to remove values from the TRQ and initiate the second series of loads. In this example, these begin to arrive at the data sub-processor just as it finishes executing the first phase. This will not lways be the case - depending upon the characteristics of the loop being transformed, the data might tart arriving before it is needed or the data sub-processor may stall waiting for data to arrive. Once he address sub-processor has finished this second phase it executes the first phase of the second strip, and the data thus fetched begins to arrive at the data sub-processor just as it finishes the second phase of the first strip.

The example used in Figure 64 is rather fortunate in that it manages to keep both work units constantly busy. However, only the first iteration illustrates the way in which it is possible to achieve both access/execute and execute/access decoupling simultaneously. From the point of view of the data sub-processor during its first phase, it is indeed access/execute decoupled from the address sub-processor's first phase. However, from the perspective of the address sub-processor's second phase, it is also execute/access decoupled, since its source of data (the first phase on the data sub-processor) is running ahead of it. On later iterations the degree of execute/access decoupling is 0, but the low latency of the TRQ means this does not noticeably affect performance. Strip decoupling is not without its problems. In addition to the already-mentioned restrictions on the type of indirection it can be used to simplify, there is a potential for deadlock. There must always be space in the TRQ for all of the data in a strip, otherwise a deadlock may occur. Since the state of the TRQ may be difficult to predict, strip decoupling can only be safely used if it follows a loss of decoupling (which should result in the TRQ being flushed) and if the strip size is smaller than the TRQ length. This suggests that if strip decoupling is to be used effectively, the TRQ should be as long as is practically possible.

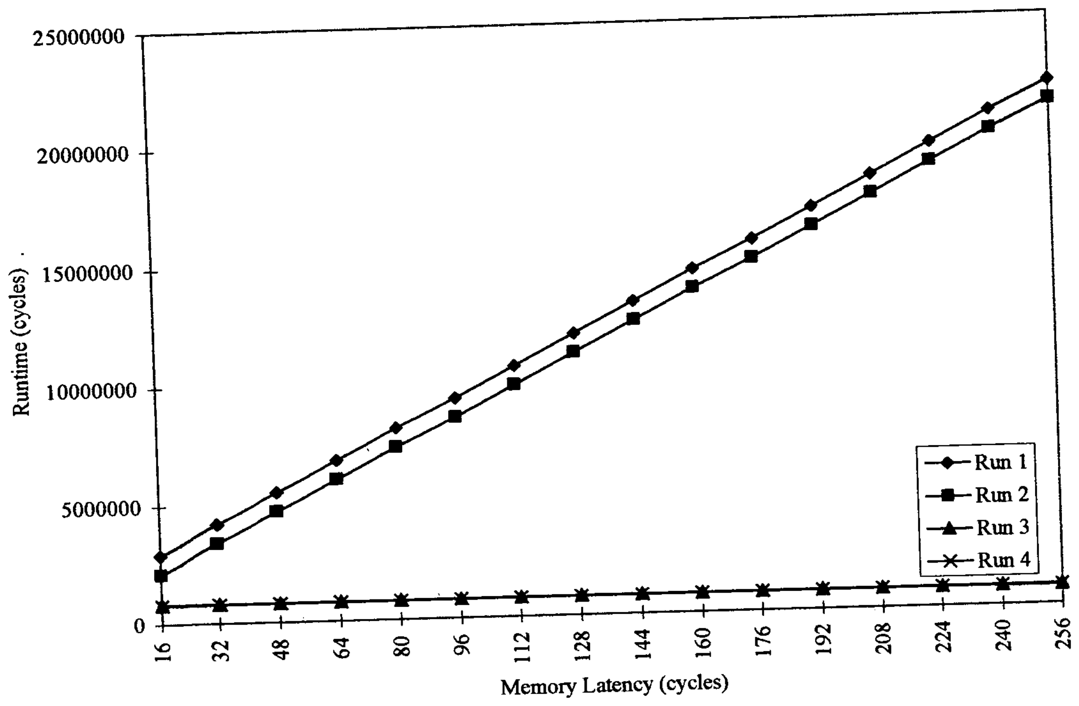


Figure 65 - Total runtime. (Ex. 7)

A hardware-based variation on strip decoupling would involve equipping the address sub-processor with a more sophisticated, non-blocking self-load operation. Load packets arriving as a result of these loads would be stored in a dedicated queue on the address sub-processor rather than in the TRQ. This avoids the need to involve the data sub-processor in what is essentially an address sub-processor operation. However, during the first phase the data sub-processor would still be wasted (doing nothing at all rather than executing a simple copy loop) and adding an extra queue while leaving the TRQ empty also seems wasteful of resources. Four runs were made in the course of the experiment, using the program loop shown in Figure 63. The first tested a naive implementation of the loop, with access/execute decoupling being lost on every single iteration. The second run added the address sub-processor self-load capability of the Level 2 architecture. The third and fourth runs used the basic Level 1 architecture in conjunction with code restructured to allow the use of strip decoupling. The third run split the 100,000 iterations of the loop into 1,000 strips of 100 accesses, while the fourth run split them into 200 strips of 500 accesses.

In all four runs a 128-way interleaved (32 segments of 4 banks) memory sub-system was used. Memory latency was varied from 16 processor cycles up to 256 processor cycles in increments of 16 cycles. The LAQ was eliminated and the LDQ set to 512 packets in size. The MIQ was set to 512 meta-instructions in size to reduce the effects of limited control decoupling. In all but the last experiment the TRQ size was set to 512 elements. In the fourth run it was felt that this might be insufficient, and the TRQ length was set to 1,024 elements. As usual, the most straightforward comparison between the varying runs is to examine their total runtimes. These are shown in Figure 65. From the graph it can be seen that the first run, where an access/execute LOD occurs on every cycle, performs most poorly. The constant gradient of the line representing this experimental run indicates that the runtime here is dictated largely by the memory latency. This in turn implies that decoupling is not occurring to any noticeable degree (as would be expected with a loss of decoupling on every iteration). This is supported by an examination of the effective memory latency, as shown in Figure 66. This reveals that while the effective memory latency is high, perhaps 50% of it is hidden. This seems reasonable, since there can be two loads active in the memory sub-system at any one time - the second load of iteration i accessing the data itself, and the first load of iteration $i+1$, performing the indirection. With two loads active simultaneously, the throughput of memory is doubled and the effective latency is (approximately) halved.

The second run of the experiment, which uses the address sub-processor self-load instruction, performs only marginally better. The address sub-processor sees the full memory latency for both loads in each iteration. This is due to both the blocking nature of the self-load operation (which sequentialises the loads) and the lack of support for a cache on the address sub-processor. However, despite this apparent handicap, the runtime is still lower. This can be put down to two primary factors. The most important of these is that although the naive implementation used in the first run has a lower effective memory latency, it must also recouple on each and every iteration, considerably increasing runtime. In addition, the memory path for the self-load operation is slightly shorter than that for a full decoupled load, trimming a few cycles from the memory latency.

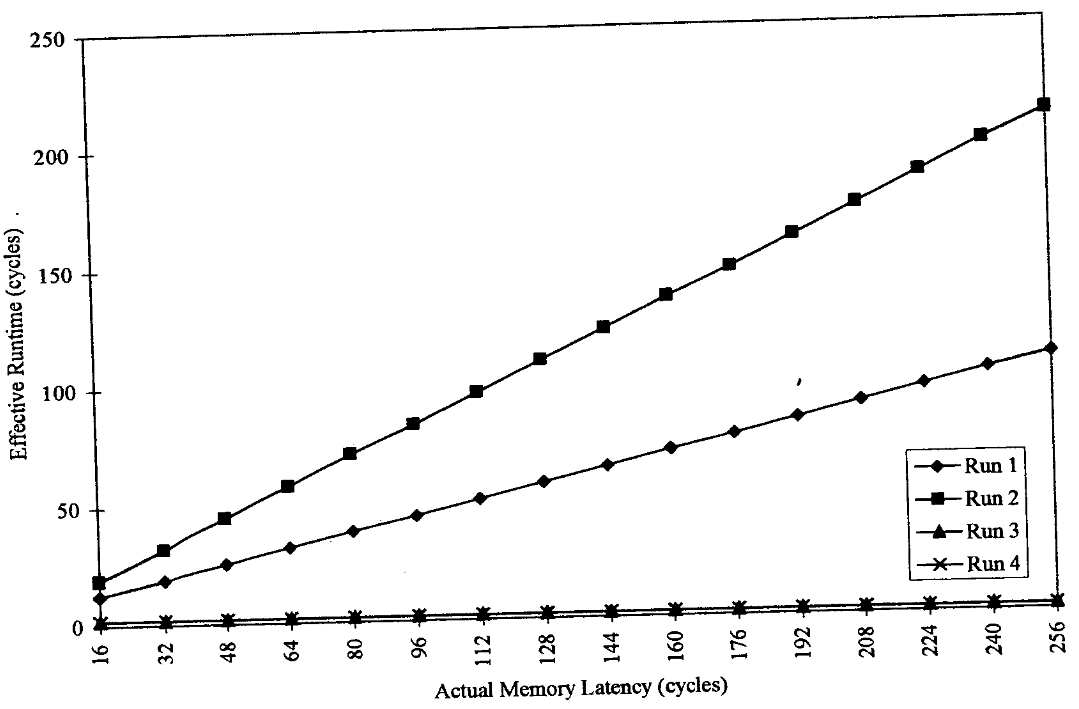


Figure 66 - Effective memory latency. (Ex. 7)

The third and fourth runs of the experiment appear at the bottom of both Figure 65 and Figure 66. Due to the scale forced by the first two runs, the third and fourth runs appear overlaid in both cases. This is not in fact the case, and the difference in runtime between the two runs can be seen more clearly in Figure 67, which shows the runtime of the final two runs in isolation. As this graph clearly illustrates, the fourth run demonstrates a runtime that is consistently lower than that of the third. In both cases the runtime appears to be largely independent of the memory latency, indicating that full decoupling has been achieved. This is supported by an examination of the effective latency of the third and fourth runs on Figure 66 which shows that both have constant effective latencies of around 2 processor clock cycles. The slight slope in the runtime graphs that are noticeable in each case are due to the memory pipeline startup time, as discussed in the previous experiment.

It is readily apparent from the above results that, in this case at least, the provision of a self-load operation on the address sub-processor, even in the absence of a cache, provides a small (and apparently constant) performance advantage over a naive implementation of redirection. This performance improvement pales into insignificance however, when compared with that obtained through the use of strip decoupling. Figure 67 also shows that increasing the "degree" of strip decoupling using longer strips improves performance slightly. This is due to the manner in which strip decoupling, while eliminating the effects of loss of decoupling events, does not actually remove the LODs. There may still be some delay due to LODs between strips. However, if there are fewer, longer strips there will also be fewer inter-strip delays, and the runtime is reduced.

This is once again analogous to strip-mining on a vector processor. Both the vector unit of a vector processor and the "soft pipeline" constructed by strip decoupling have a certain startup time. Longer vectors or strips reduce the impact of this startup time. Although this experiment clearly illustrates the potential benefit of strip decoupling, it remains to be soon how widely applicable this technique

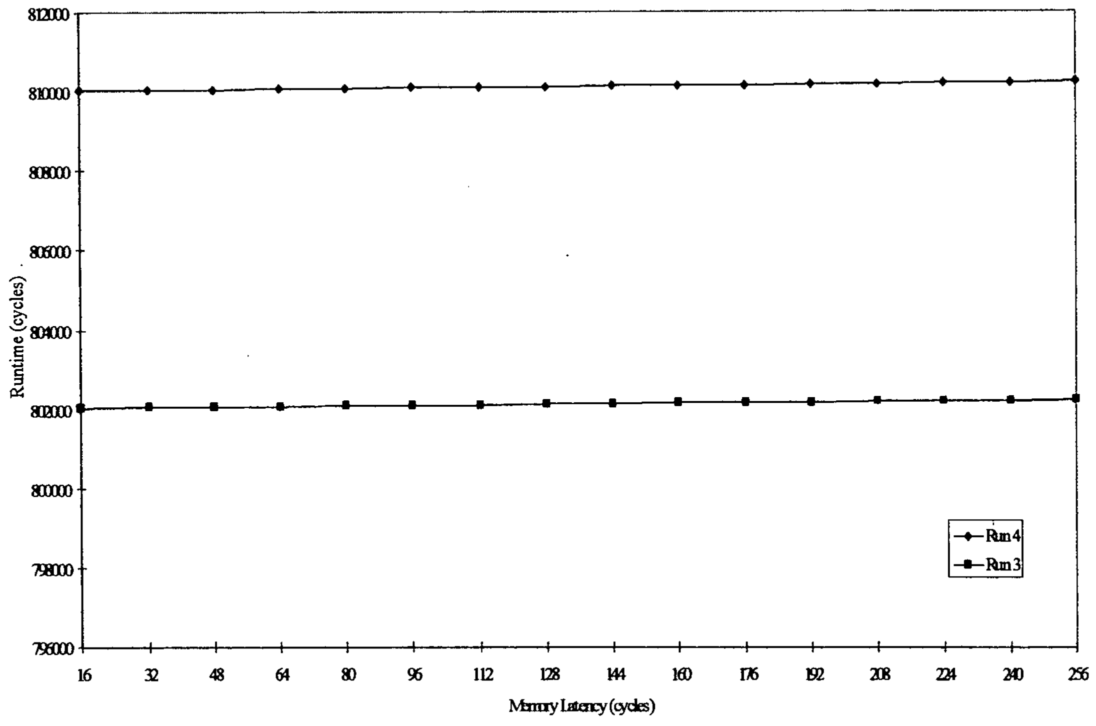


Figure 67 - Total runtime for runs 3 and 4. (Ex. 7)

vill be in practice.

.1.3.4 Code motion

While code replication can be used to eliminate loss of decoupling events, it is also sometimes ossible to reduce the number of LODs using conventional code motion techniques. Often a compiler will perform this automatically, and code motion for other purposes may often eliminate LODs implicitly. For example, a variable might be calculated before entering a loop and placed on he data sub-processor. If it is then used on each loop iteration, and by the address sub-processor, here will be an LOD on each iteration. However, only the most naïve compiler would generate code that was this inefficient. An optimizing compiler would copy the valuable of the variable from the address unit to the data unit before the loop was entered, eliminating all but one of the LODs. The use of code motion to eliminate LODs is relatively straightforward, so no experiments were performed to illustrate this technique. The basics of code motion are well covered in the literature [Aho86] and easily adapted to remove LODs once information concerning code placement has been generated by the compiler.

6.1.4 Conclusions

In this section it has been shown that the effects of a loss of access/execute decoupling can be serious. In addition to making the memory latency more visible, the repeated synchronisations caused by loss of decoupling events in a loop can also severely hamper performance by limiting the utilization of the memory sub-system. For example, if a loop contains four decoupled loads followed by a loss of decoupling, then no more than four loads can be active in the memory sub-system at any one time. In a heavily interleaved system, which may potentially have hundreds of banks, this can restrict memory bandwidth to a fraction of its peak capacity.

It has also been shown, however, that a number of techniques exist which can be used to eliminate, or reduce the effects of loss of decoupling. These include compiler optimizations such as code replication and code motion, code restructuring techniques such as strip decoupling, and the use of

dditional hardware such as the provision of a separate dedicated path to memory for the address
ub-processor.

6.2 Losses Of Control Decoupling

6.2.1 Causes

Losses of control decoupling occur for much the same reason as losses of access/execute decoupling. The fully decoupled architecture, optimized as it is for a particular "flow" of information from control sub-processor to address sub-processor to data sub-processor, performs badly when this flow must be temporarily reversed, and data passed from either the address or the data sub-processor back to the control sub-processor.

The primary cause of control LODs are conditional branches. This encompasses a number of language structures such as IF-THEN-ELSE statements, switches and numerous forms of loop construct - in short, any construct where the control sub-processor's traversal of the control flow graph depends on the evaluation of a condition. Unlike access/execute decoupling, where variables can usually be easily classified as either address variables or data variables, there is no easily identifiable class of control variables. However, most condition evaluations can be thought of as a use of a single-use control variable, evaluated on the data and/or address sub-processors and where the result is passed to the control sub-processor for use. Any address or data variable involved in a condition evaluation may be thought of as a control variable, shared between its "correct" unit and the control sub-processor.

6.2.2 Effects

Control decoupling events are of two distinct types. Address/control LODs occur when the address sub-processor and control unit must synchronize so that data can be transferred. This forces the control sub-processor to stall until the address sub-processor has caught up with it and transferred the required information via its CFQ. During this stall time, the decoupling distance between both work units and the control sub-processor will reduce - to zero in the case of the address sub-processor. Whether a loss of decoupling of this type causes any decoupling that exists between the data sub-processor and the control unit to be lost depends largely on the degree of decoupling present between the data and address units. The second form of control LOD is data/control decoupling, when the data and control sub-processors synchronize. This variety of LOD is particularly harmful, since it blocking nature of the queues could cause further difficulties. Attempting to read from an empty queue would cause the pipeline to stall. However, if the instruction attempting to read from the queue was to be suppressed anyway, there should be no need to stall. Unfortunately, there would be no way to tell until the guard condition had been set. Additionally, it may be necessary to propagate guard information from one unit to another. This will require a loss of decoupling. Even if a guard condition can be independently calculated on each of the work units (for example), decoupling itself could lead to problems, as a not-yet-suppressed instruction might read queue data that was intended to be read by an instruction in the other fork of the structure.

It appears to be the case that for guards to be used on a decoupled architecture where instructions may attempt to read from queues, their use should be limited to situations where no attempt is made to access the queue. Alternatively, guard instructions should stall the pipeline until they have committed their results. In a conventional architecture this arrangement would be senseless - it would effectively restore the branch penalty that guarding had been introduced to remove - but on a decoupled architecture, where the penalty incurred may well be far higher, this approach may still have some merit.

Guarding should thus be used on a case-by-case basis, examining (if possible) the likely result of the condition, the relative sizes of the blocks, the branch penalty and whether or not either of the forks accesses any of the queues. However, in the case of an `IF-THEN-ELSE` construct that may cause a loss of decoupling event, the use of guarding seems preferable, since it appears to give easily superior performance when the branch penalty is large.

This is examined in more detail in Experiment 8. For this experiment, three implementations of Livermore kernel 24 were compared. This kernel is designed to find the location of the minimum value in an array and consists of a loop over the array, testing the contents of each array element in turn against the smallest value previously encountered. The data in the array was fixed so that all but one elements of the array were set to zero, while an element with a negative value was placed at also causes access/execute decoupling to be lost. When the LOD occurs, the control sub-processor stalls until the data sub-processor reaches the synchronization point. During this time the control sub-processor is unable to issue meta-instructions to the address sub-processor, with the result that both the control and address sub-processors will stall until the data sub-processor reaches the synchronization point, causing both forms of decoupling to be lost.

Unfortunately, the vast majority of control LOD events are likely to cause a loss of decoupling of the second variety. Since this variety of loss of decoupling is even more costly than an access/execute LOD, eliminating or reducing the cost of losses of control decoupling is a high priority.

6.2.3 Solutions

As with losses of access/execute decoupling, both hardware and software techniques may be used in the elimination of control LODs. Many of these are analogous to the solutions used to deal with access/execute LODs, but the rather different nature of control decoupling, and the situations in which control LODs are likely to occur also mean that some rather different techniques may be used, while others such as strip decoupling have no obvious analogue in control decoupling.

6.2.3.1 Code Replication

The use of code replication to eliminate losses of control decoupling has already been touched upon, and was used in several of the examples in the previous section to make the processing of loops more efficient and thus prevent control LODs from masking the effects of the various access/execute LOD elimination techniques that were being tested. Deterministic loops are particularly well-suited to this optimization technique. The comparison of the loop index variable and the loop bounds can be migrated in its entirety to the control sub-processor, eliminating the control LOD that would otherwise have occurred after each loop iteration. In most cases, the only variable that must be shared is the loop index variable itself, which in many cases is required only on the address sub-processor. The code that must be replicated is often simple and usually requires only that both sub-processors increment the variable after each iteration. Once the replication has been performed, it is safe to perform any further required optimizations on expressions using the variable on the address sub-processor (i.e. strength reduction, scaling, the use of auto-incrementing instructions) as long as ny uses of the variable on the address sub-processor after the loop obtain a fresh copy of the variable.

More general code replication, where an address or data variable is used in a condition, can also be performed in a manner analogous to that for access/execute decoupling. As with that technique, the applicability of the technique is limited by the cascade effect, whereby replicating one variable may cause others to require replication, which may cause others to require replication, and so on. This can eventually lead to large chunks of code requiring replication, which limits the usefulness of the techniques, especially if the replicated code accesses memory. If the cost of the control LODs in such situations is prohibitively high, it may even make more sense to migrate the code concerned to the control sub-processor in its entirety. The migrated code is unlikely to be fast, but will perform at least as well as it would if replicated.

6.2.3.2 Guarding

Guarding is a well-established hardware technique designed to allow the elimination of conditional branches in conventional architectures. It is also of use in a fully decoupled architecture, where it can be used to eliminate certain types of control LOD. The principle of guarding is straightforward - every guarded instruction becomes conditional and is executed only if the guard condition is fulfilled. The guard condition varies from implementation to implementation but could be one of the standard arithmetic flags, the contents of a guard or condition register (or one of a number of guard registers) or even a combination of several guard bits. Depending upon the condition of the specified flag(s), the guarded instruction will either execute as normal or be suppressed, effectively becoming a null operation. While the nullifying of the instruction results in a pipeline bubble, the cost of the guarded instruction is generally lower than that of the branch delay caused by a conditional branch. In a fully decoupled architecture, where loss of decoupling can potentially make the cost of a conditional branch astronomically high, guarding is potentially a very useful technique. Guarding is best suited to conditions where as few guarded operations will be suppressed as possible. If t_{COND} is the time (in processor clock cycles) taken to evaluate the condition of a typical `IF-THEN-ELSE` construct, t_{BRANCH} the branch delay, t_{THEN} the execution time of the first branch and t_{ELSE} the execution ime of the ELSE branch, we can approximately compare the performance of guarding with a conditional branch. If a conditional branch is used, the total execution time t of the construct is given by $t = t_{COND} + t_{BRANCH} + \left(\Pr(THEN) \times t_{THEN}\right) + \left(\Pr(ELSE) \times t_{ELSE}\right)$. That is, we always evaluate the branch condition and wait for the branch delay, then follow one of the branches.

With guarding, on the other hand, the execution time is always $t = t_{COND} + t_{THEN} + t_{ELSE}$. Of this time, on average $\Pr(THEN) \times t_{ELSE} + \Pr(ELSE) \times t_{THEN}$ cycles are spent executing suppressed instructions versus the t_{BRANCH} cycles wasted when a conventional conditional branch is used. A comparison between conditional branching and guarding can be seen in Figure 68, which shows the execution time of a conditional branch $t_{COND} = 10$, $t_{THEN} = 50$ and $t_{ELSE} = 5$, as might occur where, following a test, a variable is either set to a constant value or its value calculated by some more complex and time-consuming method. The branch penalty t_{BRANCH} is varied from 0 to 256 clock cycles, covering a wide range from the delays of a few cycles typical of branch delays in a conventional processor, to the large delays caused by a loss of decoupling. Finally, the probability of taking the THEN fork of the branch is varied from 0 to 1, allowing the comparison of performance when the more complex of the two possible forks is taken more or less frequently.

The performance obtained with guarding is not shown, since this is a constant value determined by $t = t_{COND} + t_{THEN} + t_{ELSE}$, which evaluates to 65 clock cycles for this example. It is apparent that guarding produces superior performance in most cases, while the use of a conditional branch is most advantageous when the branch penalty is low and it is more likely that the shorter of the two forks is taken. This is expected - both implementations of the IF-THEN-ELSE structure require the evaluation of the condition. The conditional branch then suffers the branch delay, before following one or other fork. As the branch delay tends to zero, the execution time is given by the sum of the condition evaluation and the fork taken. If the shorter fork is taken more frequently, the execution time will drop. The execution when guarding is used, however, is independent of both the branch penalty (since no branch occurs) and the probability of one particular fork being taken (since, in effect, both branches are taken every time).

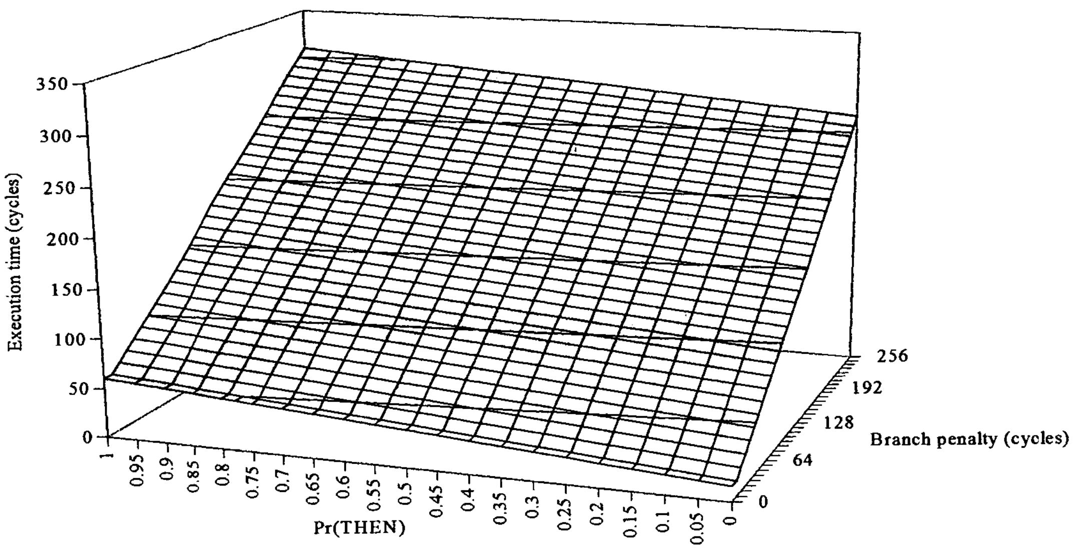


Figure 68 - Execution time for conditional branch.

Guarding faces several other limitations on a decoupled architecture. The most common method of suppressing the execution of a guarded instruction is to prevent it writing a result to its destination register or altering any condition flags. Performing either of these actions would change processor state. Reads from the register need not be suppressed, since they have no effect other than on local pipeline state, and throwing this information away when the instruction is ignored should have no effect. However, in a decoupled architecture it may well be the case that several of the registers in the register file are in fact the tops of queues. This poses a number of problems. When the queue is read this modifies global state - it removes a value from the queue - and this should not be done if the instruction is to be suppressed. Unfortunately, if the instruction setting the guard is the previous instruction, it may not be discovered that the instruction should be suppressed (and thus should not have read the queue) until after the read has been performed. There is no obvious solution to this problem. If the update of queue pointers could be delayed until such time as the status of the reading instruction was known, any subsequent instructions that wished to access the queue would be stalled since it would not be possible to determine the correct value for them to read until the status of the reading instruction itself was known. Even if it was possible to get around this somehow, the he array's midpoint. This meant that in the vast majority of cases, the ELSE branch was taken and hat $\Pr(THEN)$ was very low.

The basic loop, over an array size of 1000 elements, was repeated 100 times. The first of the three implementations of the kernel tested was the most straightforward. The main loops were migrated to the control sub-processor but the test itself was implemented in a naive manner, with the comparison of each element with the current minimum being performed on the data sub-processor and passed back to the control sub-processor on every iteration. The control sub-processor then examined the result and, if a new minimum had been found, issued the basic block to update the value of the current minimum. Once this had been done, the control sub-processor then (locally) incremented and tested the loop index variable and either started the next iteration or exited the loop.

The second implementation of the loop made use of guarding to avoid a loss of decoupling on each iteration. The basic nested loop structure was the same as before, with loop bound tests being performed on the control sub-processor. However, rather than passing the result of the test between the value currently being examined and the current minimum value back to the control sub-processor, a flag was set locally and the update of the variable performed under a guard. Since the guarded operations do not access any queues, it is safe to use a guard.

The third implementation was essentially similar to the second. No new control LOD eliminating technique was used, but it was assumed that the architecture supported an iteration count on issued meta-instructions and that the entire inner loop could thus be issued with a single meta-instruction (something that would not have been possible had the control LOD not been eliminated). This third run of the experiment was designed to highlight the performance improvements that could be achieved through not only the elimination of the control LOD but also the use of the other techniques that can only be used when LODs have been eliminated.

In all three cases, memory latency was varied between 16 and 256 clock cycles (in increments of 16 cycles), while the MIQ length was varied between 32 and 512 elements, in increments of 32 elements. While the primary interest was in the effect of variation in memory latency, the variation in the MIQ also allowed any performance variation due to limitations in the maximum possible degree of control decoupling to be measured. Those queues related to access/execute decoupling were held at a fixed size of 512 elements. The memory subsystem used had 32 segments, each with 4 banks, for a total of 128 banks.

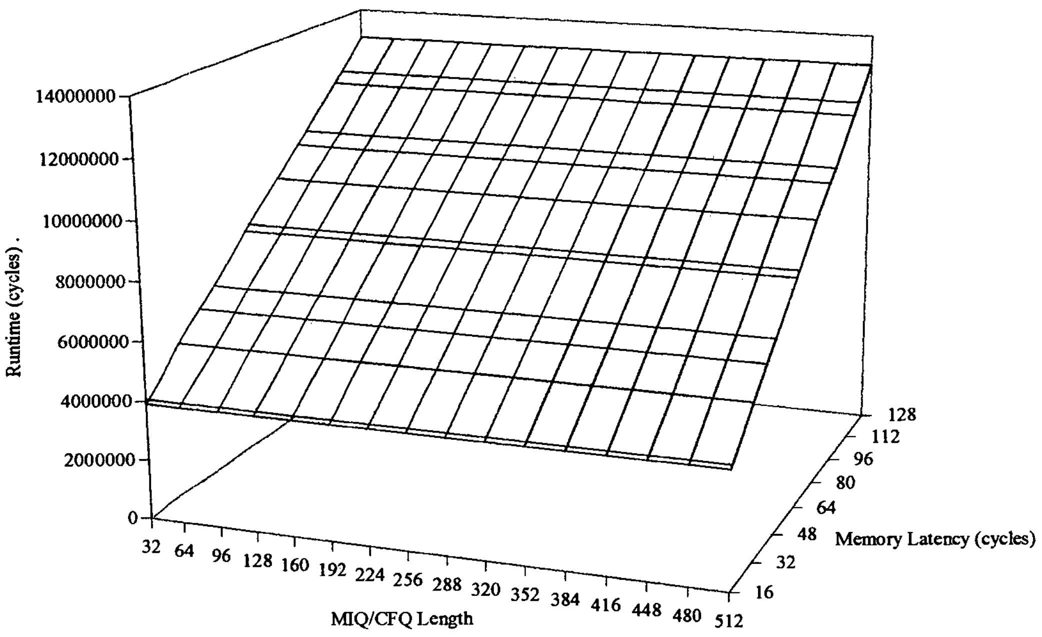


Figure 69 - Data sub-processor runtime for first run (Ex. 8).

The results of this experiment were largely as would be expected. As can be seen by contrasting Figure 69 and Figure 70, guarding produces a performance improvement when the memory subsystem is well-balanced. As the memory bandwidth becomes saturated, the benefits of guarding are reduced until, at the saturation point, memory latency dominates and there is no advantage in using guarding. The difference in runtimes between the second and third run (shown in Figure 71) appears negligible. This is what would be expected, since the control decoupling characteristics of the kernel are such that using a loop-mode block issue has little advantage. Note that while Figure

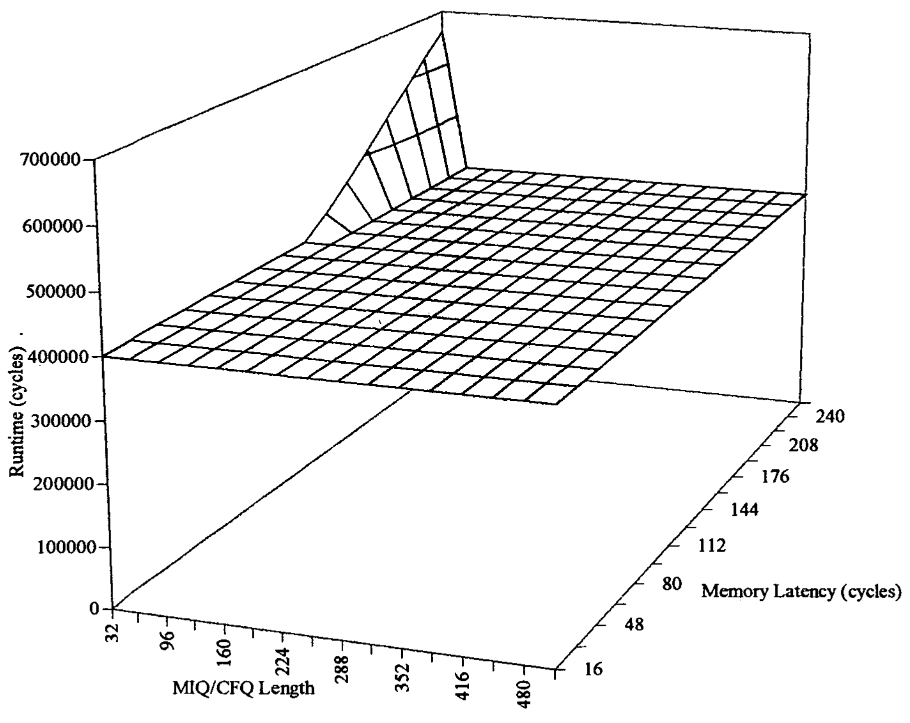


Figure 70 - Data sub-processor runtime for second run (Ex. 8).

70 and Figure 71 appear to show very different behaviours, this is an illusion due to their differing scales. Other than in the area where memory is saturated and latency dominates, the two behave identically.

6.2.3.3 Control SP self-load and self-store operations

In Section 6.1.3.2 the provision of a dedicated path to memory for the address sub-processor was examined. It was shown that while allowing the unit to directly access memory through the use of self-load and self-store instructions could eliminate losses of decoupling, it also made the full memory latency visible once more. Obtaining reasonable performance required the addition of a cache to reduce the effective latency.

This also applies to the addition of a self-load and self-store path on the control sub-processor. Losses of control decoupling due to data sharing between the control sub-processor and another unit can be eliminated, but in the absence of a data cache on the control sub-processor, the latency of self-loads may be considerable.

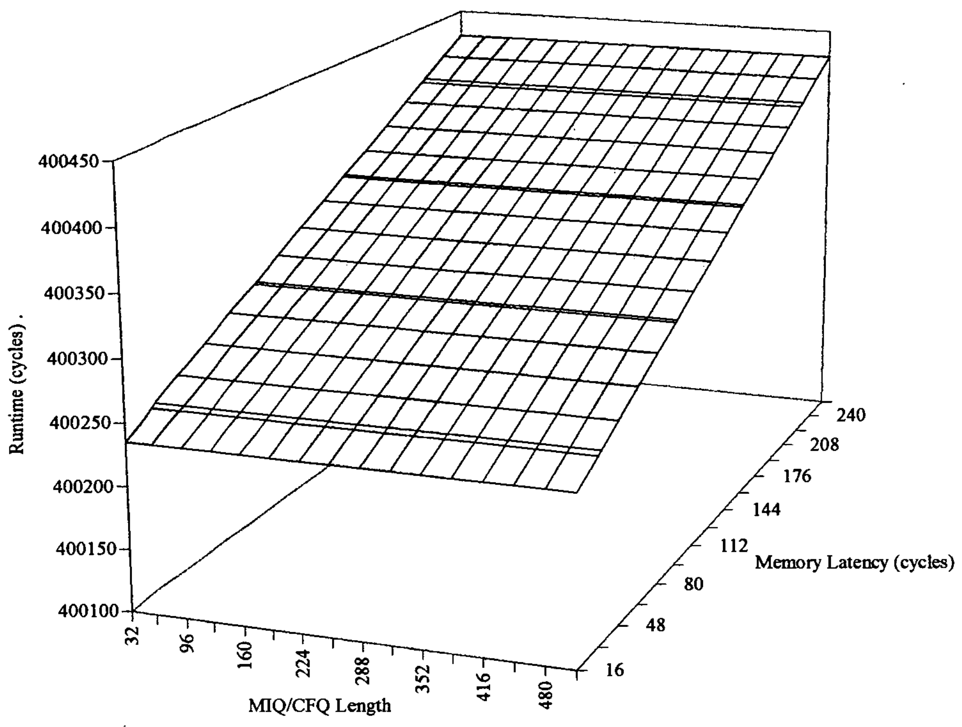


Figure 71 - Data sub-processor runtime for third run (Ex. 8).

However, the penalty thus imposed may be of considerably less impact relative to a full loss of decoupling. This is due to a number of factors. Although all loss of decoupling events are potentially expensive, control LODs are particularly so, since they cause both control and access/execute decoupling to be lost. While the memory latency of control and address self-loads may be the same, the former may represent a much greater saving in time over a loss of decoupling than the latter. Additionally, control LODs may occur in different places, and with a lower frequency. The example used for access/execute decoupling was that of an indirected array access, performed on every iteration of a loop. This, potentially causing an LOD on every iteration, made reducing the impact of any LOD elimination technique of great importance. However, although Livermore kernel 24, as used above, also causes on LOD on every iteration, many control LODs occur with rather less frequency. An LOD-causing control decision may be followed by a considerable amount of work, none of which causes an LOD. With a lower LOD frequency, it becomes more acceptable to use an LOD elimination technique that still has a high latency, or even to leave the LOD alone.

6.2.3.4 Guarded meta-instructions

The primary disadvantage of guarding is that, for a typical `IF-THEN-ELSE` construct, both the `THEN` and `ELSE` must be executed in their entirety, with any state-modifying actions of one or other of the forks being suppressed. If the two forks in the construct are of disparate size, the usefulness of guarding becomes very strongly dependent on the frequency with which the longer fork is taken. If one fork of the construct contains 250 instructions and the other 5, a situation where we execute the shorter fork 95% of the time and thereby suppress 250 instructions every time we execute the construct is clearly undesirable.

However, it is worth noting that in constructs such as an `IF-THEN-ELSE` the minimum unit of computation is the basic block. The very semantics of the construct break each fork into one or more distinct basic blocks. Since each meta-instruction represents a basic block, this suggests that by guarding the meta-instructions, rather than the individual instructions that they are expanded into, less effort will be wasted suppressing instructions. All that will be suppressed will be a single meta-instruction, rather than the hundreds of instructions it may represent. This is particularly important if the architecture supports iteration counts in meta-instructions, where one meta-instruction may be expanded into hundreds of thousands of sub-processor level instructions. Guarded meta-instructions would solve this problem, although they would also introduce difficulties of their own. Each meta-instruction would contain a guard mask of some form, determining either the guard register or combination of guard bits to be checked. This would be compared against the appropriate register or flags in the sub-processor by the fetch engine associated with that processor before the meta-instruction was expanded. If the meta-instruction was to be suppressed, it would be discarded and the fetch engine would begin processing the next meta-instruction in the queue. Using this approach, an `IF-THEN-ELSE` construct could be implemented by first issuing the meta-instruction(s) to evaluate the `IF` condition, then issuing two guarded meta-instructions, one for the `THEN` block and one for the `ELSE` block. The first of these would have its guard mask set so that it would be expanded only if the condition evaluated to true, the second so that it would be expanded only if the condition was false. Having issued these three meta-instructions, the control sub-processor could then continue unhindered by the control LOD that would have been required had guarding not been used. Guarded neta-instructions would, however, have a number of limitations. The fetch engine would be unable o determine the status of a guarded meta-instruction until the appropriate flag or register on the sub-processor had been set. In addition to requiring a close coupling of the fetch engine and the sub-processor to allow the fetch engine direct access to the flag or register, this may also introduce pipeline bubbles. The condition will normally only be written into the flag or register in one of the later stages of the pipeline. This will result either in the pipeline having emptied before the guarded meta-instruction can be expanded or something analogous to branch delay slots being required, so that some useful work can be performed while waiting for the condition to commit and the fetch engine to expand the next meta-instruction.

Although there may be a delay of several cycles between the condition being evaluated and the status of the next meta-instruction being determined, this delay will generally be no worse than would be caused by the evaluation of a conditional branch in a conventional processor. Additionally, many of the tricks and optimizations that have been used in the past to limit the delays caused by conditional branches, such as making the test that sets the flag as early in the pipeline as possible, can be used. The resulting delay would also be considerably less than that caused by a loss of decoupling and is also superior to at least one of the two forms of instruction-level guarding already discussed. The penalty incurred when a guarded instruction has to wait for the previous (flag setting) instruction to store its result is of similar magnitude to that which would be incurred by a guarded meta-instruction. Additionally, the use of a guarded meta-instruction would avoid the penalty of executing then discarding the results of large numbers of guarded instruction, producing a considerable saving.

The other primary limitation of guarded meta-instructions is a result of the manner in which the control decision is only partially migrated to the fetch engine. If each branch of an `IF-THEN-ELSE` construct consists of several separate basic blocks a guarded meta-instruction must be issued for each block. Once again the situation is encountered where both paths of the construct are executed, and a number of instructions suppressed. While the number of meta-instructions is far smaller than the number of sub-processor level instructions that would be suppressed were instruction-level guarding used, it is still wasteful.

Finally, it must be stressed that the use of guarding, whether at the instruction or the meta-instruction level, is not practical in all situations. While multi-way branches could be implemented in this manner, using multiple guards, the ratio of executed/suppressed instructions would be worse than in a simple two-way branch. There are also many situations in which the result of a condition evaluation must be passed back to the control sub-processor. For example, in a non-deterministic loop the condition must be passed back so that the loop can be terminated. Each individual iteration could be implemented using guarding (with the loop body and the loop footer having complemented guards) but there would be no way to place the meta-instructions for the next iteration into the MIQ (or prevent the control sub-processor from issuing any further iterations) without passing the result of each iteration back. Such losses of decoupling seem unavoidable.

6.2.3.5 Conditional meta-instructions

Since the majority of situations in which guarded meta-instructions would seem to be useful are simple `IF-THEN-ELSE` constructs, it may be the case that optimizing for this particular case would be beneficial. Such a conditional meta-instruction would be generally similar to a pair of guarded meta-instructions with the minor difference that the two meta-instructions would be implicitly paired and would require only one check of the guard flags since only one of the two would ever be executed. Whether such meta-instructions would be issued as a single operation on the control sub-processor or separately is an implementation dependent issue.

6.2.3.6 Performance of meta-instruction level guarding

Having outlined the concept of meta-instruction level guarding in the previous section, an experiment was carried out to evaluate the relative performance of conventional instruction-level guarding, guarded meta-instructions and conditional meta-instructions. Experiment 9 consisted of three simulations of a kernel based on Livermore kernel 24, which was previously used in Experiment 8. However, to accentuate the effects of guarding, the relative weights of the two branches in the `IF-THEN-ELSE` that lies at the core of this kernel were altered to skew the cost towards the less frequently taken branch. This better illustrates the inefficiency of instruction-level guarding, since on the majority of iterations the loop suppresses a large number of guarded instructions.

The first of the three simulation runs used conventional instruction-level guarding. The second used guarded meta-instructions, with the control processor issuing two meta-instructions (with complementary guard values) on each iteration. The third and final run was similar to the second, but used conditional meta-instructions, where the two guarded meta-instructions are combined into a single instruction at the control sub-processor. How this instruction would be implemented in practice would depend upon the specific architecture in question. Two possibilities would be the use of a wide meta-instruction queue (although this would seem wasteful), or the implicit queuing of complementary guarded meta-instructions. The latter approach is the one taken by the architecture simulated by `fdps`.

It was expected that the performance of the second and third runs would be very similar. From the perspective of the work sub-processors there is in fact no difference between the use of guarded and conditional meta-instructions. However, the use of conditional meta-instructions should reduce the instruction count on the control sub-processor.

Each simulation run was performed 64 times, varying the memory latency between 32 and 256 clock cycles (in increments of 32 cycles). Since the previous experiment had shown little variation in performance due to meta-instruction queue length, this was held constant at 512 elements. Other aspects of the simulated processor were held to the baseline used in earlier experiments. The first measure of performance examined was the total runtime of the data sub-processor. As can be seen from Figure 72, the performance of the simulations using guarded and conditional meta-instructions is almost identical. This was as expected, and the runtimes of the second and third batch of simulations appears to depend linearly on memory latency. The only part of the performance envelope where the use of guarded meta-instructions was noticeably superior to that of conventional guarding is where the memory latency is low. With higher latencies, the effects of the memory bank conflicts on performance mask any improvements due to the use of guarded meta-instructions.

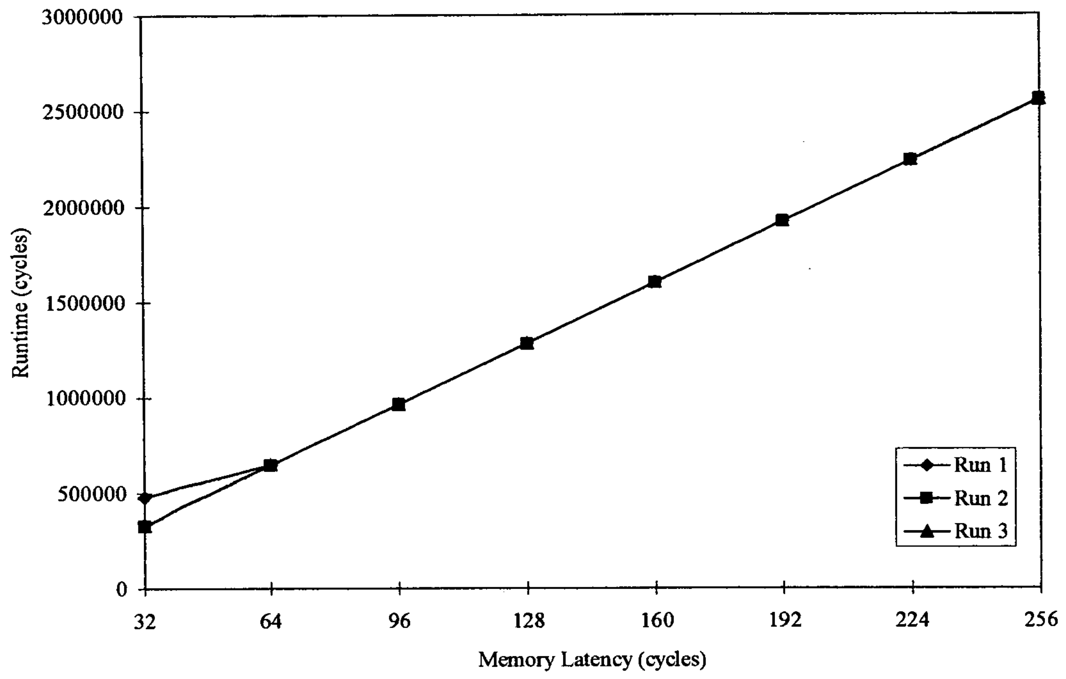


Figure 72 - Data sub-processor runtime. (Ex. 9)

Further insight into the behaviour of the various forms of guarding was gained by examining the control processor efficiency. This showed greater efficiency when using guarded meta-instructions than conditional meta-instructions. While this might at first seem to be counter-intuitive, it is due primarily to the time it takes to fill the MIQ. With guarded meta-instructions each iteration requires the issue of two meta-instructions, which in turn requires the execution of two instructions on the control sub-processor. The issue of a conditional meta-instruction, however, still stores two values to the MIQ but requires only a single instruction to be executed on the control sub-processor. Thus, the use of conditional meta-instructions fills the MIQ at twice the rate of a program using guarded meta-instructions. Once the queue is full both approaches proceed at a similar rate, only adding meta-instructions when space is available. Since the third simulation run spent more time in an idle state (because it filled the queue earlier), efficiency was reduced.

5.2.3.7 Decoupling-specific guarding techniques

The combination of guarding and decoupling has both drawbacks and advantages. Perhaps the biggest drawback to its use occurs when code on both the address and data sub-processors must be guarded. This might happen if the code that is to be guarded attempts to access memory, or even if it has to update some register variables. Generally this will require that the result of the guard condition be shared between both sub-processors, causing a loss of access/execute decoupling. Alternatively, replicating the code that evaluates the condition over both units may be possible. This limitation may in many cases restrict the efficient use of guarding to basic blocks that are executed entirely on one processor, such as the manipulation of register variables on the data sub-processor. However, decoupling can also help as well as hinder the efficient guarded execution of certain structures, by virtue of it making it possible to guard *parts* of decoupled operations.

This is best illustrated by a number of variations on the C tertiary operator `a = x ? y : z`. This is a simplified and specialized instance of an `IF-THEN-ELSE` construct where, depending upon the result of the evaluation of expression `x`, either `y` or `z` is assigned to `a` - it is equivalent to `IF x THEN a = y ELSE a = z`. In a conventional architecture with guarding, this would be implemented as a condition evaluation plus two guarded loads and two guarded store operations. This in itself is fairly efficient, but if calculating the address of `a` is more complex (for example if `a` is a reference to a multi-dimensional array rather than a scalar variable) efficiency will drop, since the expensive evaluation of `a` will be performed twice, once for the fork being executed, and once (although suppressed) for the guarded fork.

This problem can be considerably reduced by "hoisting" the evaluation of the address of `a` out of the condition. However, it can be eliminated entirely on a decoupled architecture, by evaluating the guard only on the data sub-processor. The address sub-processor executes the code to evaluate the address of `a` regardless of the value of `x` and the guarded code on the data sub-processor generates the data value (either `y` or `z`) to be stored at that address.

```
Original Code              ASP Pseudo-code               DSP Pseudo-code

IF x THEN                        g1 = EVAL (x)                 g2 = EVAL (y)
   IF y THEN                 g1: LAQ = ADDR (a)            g2: LDQ = m
      a = m                 !g1: LAQ = ADDR (b)           !g2: LDQ = n
   ELSE
      a = m
   END IF
ELSE
   IF y THEN
      b = m
   ELSE
      b = n
   END IF
END IF
```

Figure 73 - Guarded pseudo-code for nested IF-THEN-ELSE.

This principle can also work in reverse, storing a value at one of two addresses (`IF x THEN a = y ELSE b = y`). This would guard the address sub-processor code, but generate a single, unguarded store of `y` on the data sub-processor. In fact, both can be used together, so that the nested `IF-THEN-ELSE` shown in Figure 73 can use a separate guard on each sub-processor to implement the construct extremely efficiently.

6.2.4 Conclusions

In this section it has been shown that losses of control decoupling are every bit as harmful to performance as losses of access/execute decoupling. In many cases control LODs also force an access/execute LOD, increasing the impact of these events.

Many of the software techniques that can be used to eliminate access/execute LOD events are also applicable to control decoupling. Software techniques such as code motion, replication and hoisting can eliminate potential LODs.

Hardware techniques such as the provision of a separate memory path for the control processor were also shown to be of use. Additionally, a number of hardware techniques specific to control decoupling were examined, such as the use of instruction guarding at both the instruction and meta-instruction levels. While the use of both types of guarding was shown to have a positive effect on performance, other characteristics of the benchmark used to test the techniques predominated, making it difficult to quantify exactly how effective the various techniques were. Instruction-level guarding is relatively easily implemented, and is a far from uncommon feature in high-performance processors of the 90s. By contrast, meta-instruction level guarding is more involved and would require a tighter coupling between the processor's work units and their respective instruction fetch units than currently exists. This, coupled with what information could be gleaned from the limited results of Experiments 8 and 9, suggests that the benefits of meta-instruction level guarding (particularly the more involved forms, such as the use of conditional meta-instructions) are small and are outweighed by the additional hardware complexity that they require. Instruction-level guarding, by contrast, offers a straightforward means of eliminating some frequently occurring classes of LOD event.

7. The Decoupling Behaviour of Real Programs

In the previous two chapters, the characteristics that permit uncoupling and the causes of losses of decoupling were examined for a number of typical scientific kernels. Simple code kernels such as those used are useful when examining the fundamental behaviour of an architecture, since they permit the isolation of the behavioural characteristic or architectural feature under examination.

However, such kernels tell only part of the story of processor performance. This is particularly so in the case of fully decoupled architectures. The use of simple kernels provides an accurate estimate of processor behaviour in certain highly controlled situations. However, such situations are often unrealistically simple and fail to capture the behaviour of the processor when running real programs.

Real programs, by contrast, may have a number of modes of behaviour that they switch between frequently. Loops may only be executed tens of times rather than the thousands of times that are typical of kernels, or they may be executed different numbers of times on different iterations of some outer loop. In the case of fully decoupled architectures, switching between different modes of behaviour may result in poor decoupling or loss of decoupling. Even a small change in the degree of available decoupling has the potential to reduce efficiency if the program switches rapidly between two modes with differing degrees of decoupling. A loop that might have the potential for full decoupling may be alternated with a section of code with poor decoupling characteristics, or even a loss of decoupling. Alternatively, the loop might only be executed twice, and thus fail to become fully decoupled. To determine the effects of such varied and complex behaviour, it is necessary to examine the decoupling characteristics of real programs.

7.1 The problems of simulating the execution of real programs

In this chapter, the behaviour of fully decoupled architectures when running realistic programs rather than simple kernels is examined. In the case of a decoupled architecture, "realistic programs" will typically be used refer to scientific or numerically intensive code, rather than interactive I/O driven applications such as editors or graphical user interfaces. A good example of the class of programs that a decoupled architecture might typically run is given by the Perfect Club suite [Ber89] [Cyb90], a selection of real programs that are commonly used for benchmarking the behaviour of processor architectures.

However, time constraints and the sheer size of some of the Perfect Club benchmarks make a wide-ranging investigation of the Perfect Club suite impractical. Preliminary calculations showed that to simulate just one benchmark in the suite - the TRFD program - in the same manner as the experiments performed earlier in this research (typically 512 simulation runs for each of several possible architectural variations) would take approximately 30 years of continuous simulation on the available processors.

It was decided instead to choose a single example from the Perfect Club benchmarks to examine processor behaviour at the level of a full-sized program. The choice of which benchmarks to use was based on the results presented in a paper by N.P. Topham and K. McDougall [Top95], an analytical survey of access/execute decoupling in the Perfect Club benchmarks. By using the results in this paper it was possible to select a single benchmark which had a relatively low runtime and which demonstrated both periods of decoupling and the presence of loss of decoupling events.

The limitations of the tools available for the simulation of decoupled software and the potentially large size of real programs meant that it was impractical to analyze the behaviour of these real programs in their entirety via simulation. The annotated codes would grow enormously. The generation of traces would become time-consuming and require vast amounts of disk space for storage. Finally, and most importantly, the time required for the simulation itself would become unfeasibly large. It was instead decided to employ a number of techniques that would reduce the size of the traces generated without invalidating any of the results produced.

7.1.1 Trace reduction techniques

The first of these techniques involved the profiling of the program to be simulated to identify its most frequently executed routines, and to obtain a static approximation of its decoupling characteristics. Using this information, frequently executed sections of code with interesting decoupling behaviour (specifically, those which contain LOD events) were then annotated, either automatically or by hand. This produced a program with a much reduced degree of annotation that still reproduced the behaviour of the most frequently executed routines, and thus paid due attention to the presence of the most important decoupling events.

However, even this technique was insufficient to reduce the size of traces to manageable levels. To give an example, the TRFD program from the Perfect Club was minimally annotated using this technique and the resulting annotated program was run to generate traces. These trace files were compressed during generation using the `gzip` utility set on its highest compression setting. The resulting trace files consumed roughly 3 gigabytes of disc space in their compressed form. Their uncompressed size would be larger by at least an order of magnitude. In addition to the storage problems posed by such large traces, the time taken for the trace-driven simulation became prohibitive. The simulator used throughout this research is capable, at best, of around 10,000 simulated clock cycles per second for a typical decoupled architecture. For a simulation run containing billions of simulated instructions, simulation times of several weeks are typical even if it is assumed that the simulated architecture has perfect efficiency (that is, an instruction is issued on every sub-processor during every clock cycle and memory latency is unrealistically low). When the less-than-perfect behaviour of real programs is taken into account, coupled with the need to perform multiple runs of a simulation, even this reduced degree of annotation seems to be impractical.

Two additional steps were thus taken to reduce simulation time. The first involved modifying both the trace generation library and the simulator itself so that trace information was passed directly between the two, bypassing the need for large amounts of disk space to store traces.

This technique had both advantages and disadvantages. Since simulations were being run at off-peak hours on a dual-processor Sparcstation 20, the need to have two processes running together did not in itself impact performance. The total simulation time for a run of n experiments was also reduced. Previously this had been $t_g + n \times t_S$, where t_g was the time to generate the trace data and t_s was the average time for s single simulation run. With generation and simulation taking place in parallel, the total simulation time became $n \times \max(t_g, t_s)$. Since $t_g < t_s$ in the vast majority of cases, this reduced to $n \times t_s$, effectively removing the trace generation time from the equation. In the experiments performed in this chapter, where both trace generation time and simulation time were measured in days/weeks, this proved to be important.

The disadvantages of this technique were of a more technical nature. Due to the inherently decoupled nature of the trace streams being produced by the annotated program, the buffers provided by the operating system for the exchange of data through sockets/streams proved too small to prevent deadlock. This necessitated the provision of program-maintained buffering, which was implemented in the annotation library. This in turn required the implementation of a throttling algorithm, to prevent the program-maintained buffers from growing arbitrarily large, while still ensuring the continuing flow of data from trace generator to simulator. This did not completely eliminate the possibility of deadlock - where two units were out of step by a very large amount (that is, greater than the maximum instruction capacity of the buffer), deadlock could still occur. However, it was felt that the buffer size used (a maximum of 6 megabytes per processing unit) was enough to prevent this, and this did in fact prove to be the case. Unfortunately, the implementation of program-maintained buffering, which required the simulator to busy-wait on the sockets from which it was receiving stream data, increased the simulation overhead noticeably, reducing performance by perhaps 5%.

The final technique employed to reduce simulation time was perhaps the most drastic, and the only one of the three to noticeably effect how representative the results were of the behaviour of real programs. The large runtime of each simulation was due directly to the number of instructions that must be simulated. A substantial number of these instructions had little or no bearing on the program's decoupling behaviour or, if they did, affected only the degree of that decoupling, and not whether or not decoupling took place.

By removing from the trace all instructions except those that were directly concerned with decoupling, runtime could be reduced still further. Instructions concerned with decoupling include memory accesses, the issuing of instruction blocks by the control unit, and the passing of information from address and data units back to the control unit, or from the data unit to the address unit. Depending upon the specific processor architecture, subroutine calls and returns from subroutines may also need to be retained, since these can generate LODs in architectures where parameters are passed through memory.

While the use of this technique greatly reduced the number of instructions that had to be simulated, it did distort the program's behaviour in a number of ways. It was no longer possible to accurately determine the rate at which two processors decoupled or recoupled, since this is determined by the frequency with which decoupled instructions are executed. Since we eliminated the non-decoupled instructions that dictate the spacing of these instructions, both processors involved in a decoupled operation would proceed at the highest frequency possible, with *every* instruction that was issued being half of a decoupled pair. In the absence of data concerning the relative frequency at which decoupled instructions are issued, it was not possible to determine the decoupling or recoupling rate of the original program.

An additional effect of the elimination of non-decoupled instructions was an increase in the demand on resources in the decoupled data path. In the case of control decoupling this meant that the MIQ was accessed more frequently, potentially causing it to fill or empty with a greater frequency than in the real program. In the case of access/execute decoupling, the "pressure" on memory was increased, as memory accesses became more frequent. Unless the memory subsystem was adjusted to handle this increased pressure, measures of memory performance would have been distorted due to the LAQ and SAQ filling regularly (since the demands being made on memory exceeded its capabilities) and the LDQ emptying regularly (since the execute sub-processor would remove data more frequently).

7.2 Decoupling behaviour at the program level

The trace-reduction techniques described above were all used in the final experiment conducted in this research, which attempted to verify some of the small scale behaviours demonstrated in earlier experiments at the level of a real program. The aim of this experiment was to examine the effects of varying the memory latency on the behaviour of the program TRFD. As has already been indicated, a complete run of this program was several billion simulated cycles in length, even in ideal circumstances. However, by examining the decoupling profile of the first two billion cycles of a simulated run of TRFD, it was possible to determine that the first half billion simulated clock cycles alone contained a sufficient variety of program behaviours to be worthy of examination.

The architecture used in each run of the simulation was the same. All queue lengths were kept long (512 elements) to reduce the effects of queue stalls on performance. The memory subsystem consisted of 16 segments, each possessing 8 banks. This 128-way interleaved system was used, in conjunction with Rau's hashing algorithm, to reduce the effects of bank contention and to ensure that ample memory bandwidth was available. Each memory bank was equipped with a 128 element local queue to localize the effects of bank contention.

The high specification of the architecture used in the experiment was designed to eliminate as many subsidiary performance effects as possible. Memory latency, and the effect of this on decoupling performance, were the only factors being considered.

The same code annotation for TRFD was used in every case. Static analysis and profiling had shown that the `olda` subroutine within the program contained the vast majority of LOD events and was also where the program spent most of its time. Thus only this subroutine was annotated, in the minimally annotated form described in Section 7.1.1. No attempt was made to remove LOD events, or to hoist them to reduce their effects.

Three runs were made of the simulation. The only parameter varied each time was the memory latency. This was set first to 16, then 32, then finally 48 processor clock cycles. These figures were deliberately chosen to be relatively low, as it was considered undesirable for the simulation to encounter the situation where performance was affected by a lack of available memory bandwidth. It was felt that all three of these latencies, while varying substantially, would not tax the memory subsystem unnecessarily. Each run simulated the execution of the annotated TRFD for 0.5 billion cycles, and took around a week of simulation time to complete.

In addition to the static performance information gathered in previous experiments, the simulator was also altered to dynamically gather a number of performance characteristics, to provide a profile of the system's behaviour over time. Three such characteristics were dynamically measured.

The first of these was the degree of decoupling. A running total was maintained of the number of decoupled load and store operations issued on the access and execute units.. These totals were sampled every 1000 clock cycles (this being considered quite frequent in the context of a 0.5 billion cycle run) and the difference between the two was measured. This gave an estimate of the degree of decoupling between the access and execute units at that point, or how far ahead the access unit was running.

The second characteristic measured was derived from this information. As has been mentioned earlier, one of the side-effects of reducing the trace size was to greatly skew the recoupling/decoupling behaviour of the annotated program. However, there was still some value in calculating the rate of recoupling/decoupling for the annotated program, to see if this varied for different memory latencies. Such information could be gathered directly from a graph of the degree of decoupling, where the rate would be given by the gradient. However, the initial test runs that indicated the impracticality of complete simulation of real programs also demonstrated that this gradient was very difficult to determine visually. Even if the degree of decoupling varied sharply (say, by several hundred) between two sampling points, there might be half a million such sampling points for an experiment, and any display of this data would greatly compress the time axis, rendering all such gradients near vertical. For this reason the decoupling rate information was calculated at runtime, although it could also have been derived after simulation from the accumulated degree of decoupling data.

The final dynamic characteristic measured was that of effective memory latency. While the simulator already calculated the average effective latency over the complete simulation run, it was modified so that the effective latency information was reset at each sampling point, thus giving a measure of the average effective latency over the past 1000 clock cycles. The idea behind this was that it would now be possible to see the effect of losses of decoupling on effective latency. A loss of decoupling should, in theory, have caused a sudden jump in effective latency as the latency-hiding effects of decoupling were lost.

Once the three simulation runs had been performed, the results for each run in turn were collated. For such a huge volume of data (500,000 data points per measured characteristic per run) graphing of the results seemed to be the only way to visualize the program characteristics. Each characteristic was graphed for each run and, additionally, graphs were produced for each characteristic combining the data from all three runs.

7.2.1 Results

The dynamic degree of decoupling results measured for the 16, 32 and 48 cycle latency runs of TRFD are shown in Figure 74, Figure 75 and Figure 76 respectively. It is immediately apparent from these graphs that while the overall trend in behaviour is clear, interpretation of the results is far from easy. Contrary to appearances, these are *not* bar graphs. Each datum has been plotted as a point, and the apparently solid areas of the graph are due to the high data density. With a horizontal

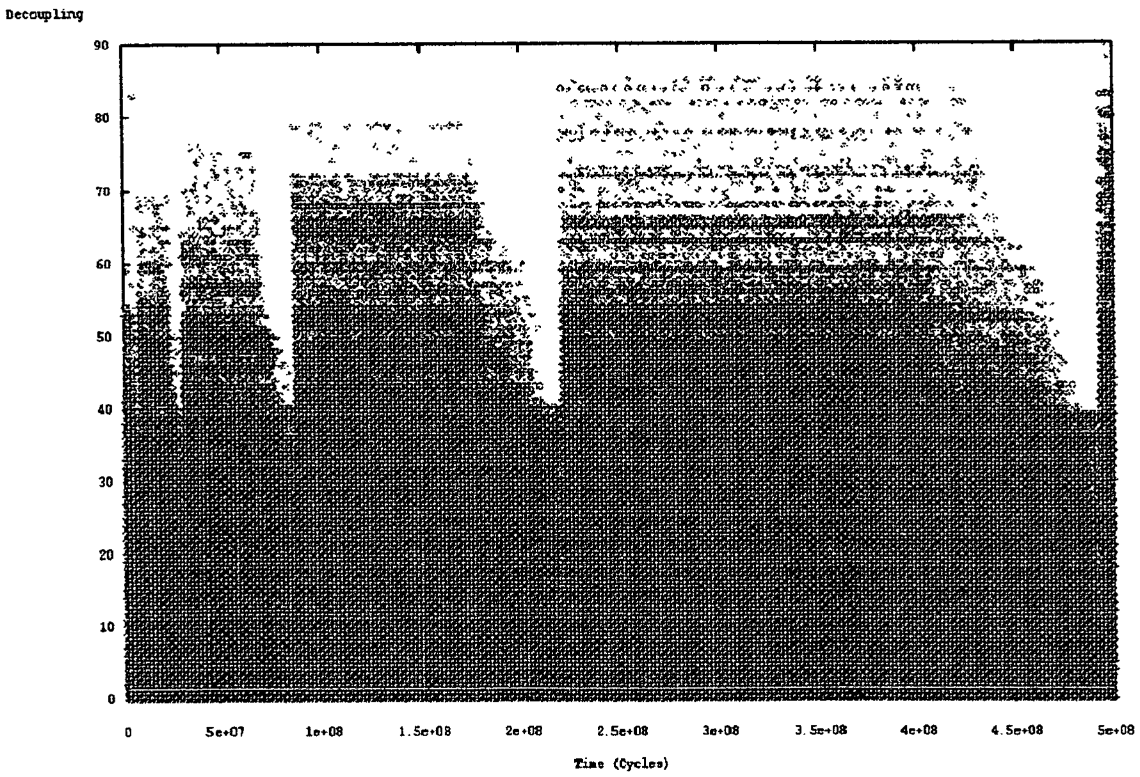


Figure 74 - Degree of decoupling for 16 cycle memory latency. (Ex. 10)

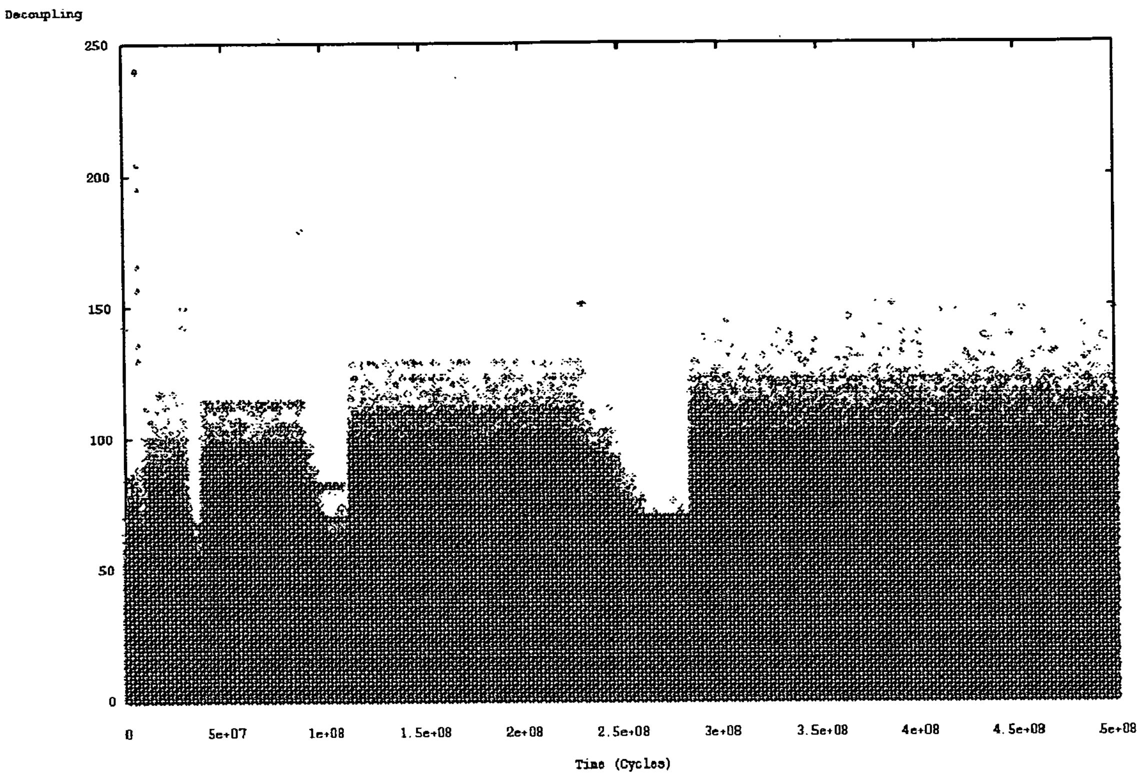


Figure 75 - Degree of decoupling for 32 cycle memory latency. (Ex. 10)

pixel resolution of around 1000 pixels, and 500,000 points to plot, each point along the x-axis must represent 500 data points, or 500,000 processor cycles. Where decoupling has varied greatly within this period of time, a solid vertical line results.

However, even if we greatly magnify the graph, clarity is not noticeably improved. For example, the first million cycles from the first run were examined in isolation, the graph produced appeared no more meaningful than those shown here.

The reason for this is due to the enormous gulf between the level at which changes in program behaviour can be accurately resolved, and the scope of the experiment. While 1000 cycles is a very short period of time in relation to the 0.5 billion cycles of the simulation, it is a very long time at the instruction level, long enough for the processor to issue a large number of loads (raising the decoupling level) then recouple without this showing in the measured data. In short, the required Nyquist frequency for the measurement of this information is at least a couple of orders of magnitude higher than that at which the measurements were made. It may even be the case that, to get a useably accurate representation of decoupling behaviour, we need to sample on a cycle-by-cycle basis. This, of course, would generate such enormous volumes of data as to render even a 0.5 billion cycle simulation impractical.

The incompatibility of these two requirements - to keep the data gathered to a reasonable size we need to sample relatively infrequently, but to get meaningful data we need to sample frequently appear to represent a severe problem to the analysis of the decoupling behaviour of real-life programs. It should be remembered that the benchmark chosen was one of the less demanding in the Perfect Club suite, and that the specific architecture used for the experiment was designed to be very fast indeed, with plentiful fast memory and large queues. The majority of real-world programs would experience this problem to an even greater degree, as would the use of a more realistic architecture.

Are the results completely meaningless, however? Not necessarily. While the data density renders individual data points almost useless, the large number of these points lets us draw some conclusions from these graphs. The inaccuracies introduced by the low sampling frequency are offset somewhat

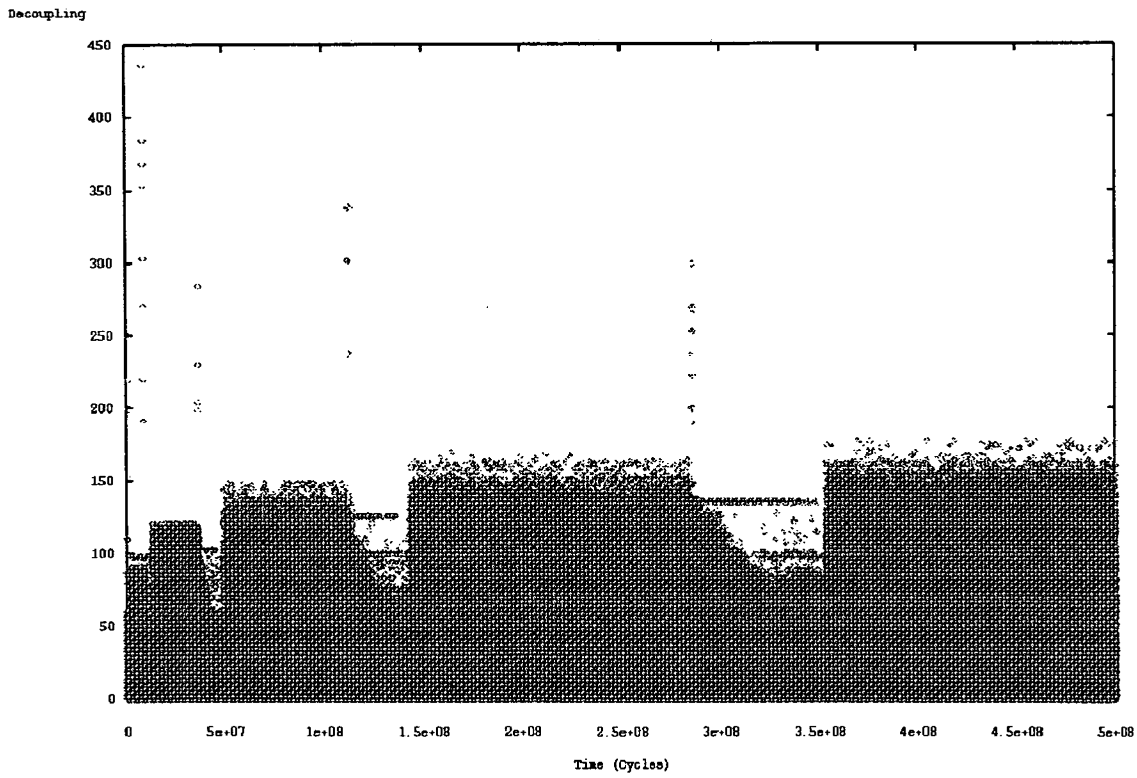


Figure 76 - Degree of decoupling for 48 cycle memory latency. (Ex. 10)

by the high number of data points, which has an averaging effect. For example, if we examine Figure 74 we can see several distinct phases to the program's behaviour. While it is, unfortunately, not possible to map these to specific program features (which exist on a much smaller time scale) we can see a number of periods of reasonably high decoupling, separated by brief bursts of low, or no decoupling. These distinct phases within the program are most probably the result of successive calls to `olda` by the main body of the program. We can tell that the degree of decoupling rarely exceeds 80 and that within each individual phase of the program certain levels of decoupling occur more frequently than others (and that for some inexplicable reason - possibly an artifact of the sampling - the degree of decoupling is never 2).

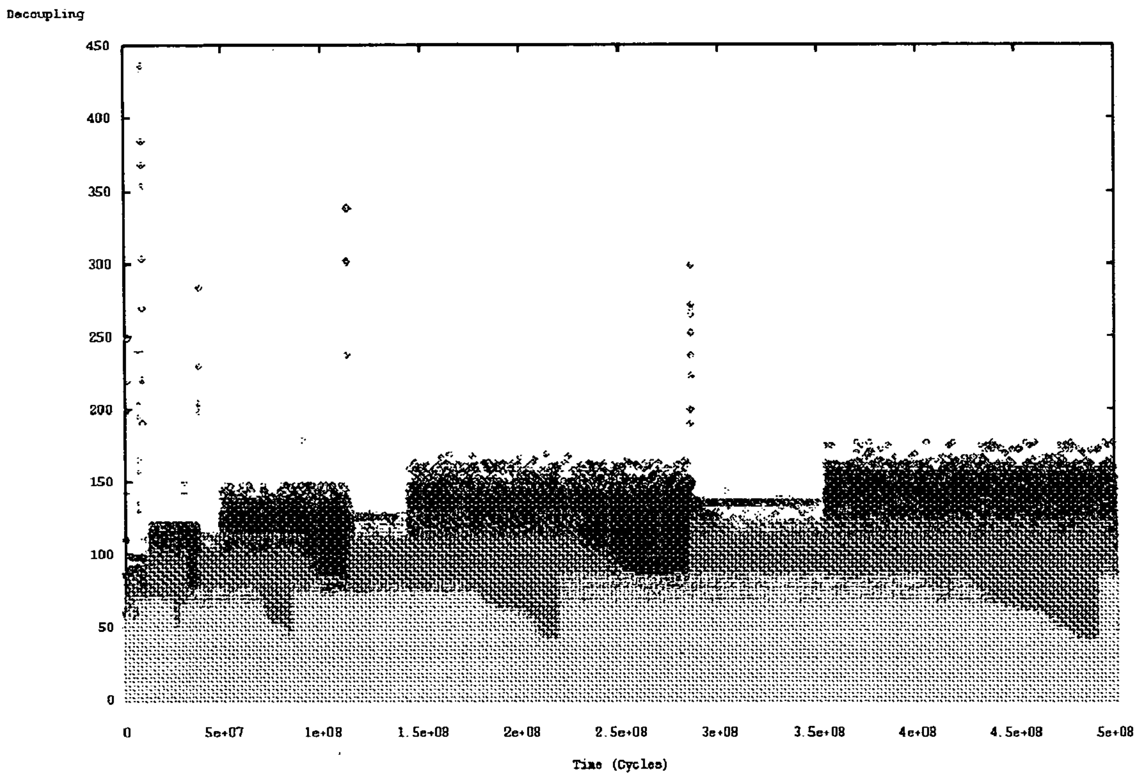


Figure 77 - Degree of decoupling for all three runs. (Ex. 10)

In isolation this information is of little use, but it becomes more meaningful when we examine the data for the three runs together. This is shown in Figure 77. It is clear from this graph that while a greater memory latency affects program behaviour strongly, the three data sets are similar, with each exhibiting the same basic features, albeit distorted by the different latencies. It can be seen that, unsurprisingly, increased memory latency increases the runtime of the program, with the various program phases starting later as the latency increases.

It is also quite noticeable that the maximum degree of decoupling achieved is higher for higher memory latencies. This is due to the manner in which data flows through the "memory pipeline" - the combination of LAQ, memory banks, and LDQ. Given the stripped down nature of the trace, both r_a, the rate at which the access unit generates addresses, and r_e, the rate at which the execute unit consumes data, tends towards unity.

When the memory pipeline is initially empty (either at the beginning of the program, or after an LOD event) it begins to fill at rate r_a. At this point the degree of decoupling is linked directly to r_a and the change in the degree of decoupling is given by $r_a \times t$, where t is the elapsed time in cycles. The degree of decoupling increases at the rate specified by r_a until such time as the first loaded value passes from memory to the LDQ. The degree of decoupling is then determined by both r_a and r_e - specifically by $(r_a - r_e) \times t$. In the case of the TRFD trace, most of the time both r_a and r_e are close to unity, so the degree of decoupling is essentially fixed. At some point (depending upon the speed and architecture of the memory subsystem) the LDQ may fill, which will tie r_a not to the rate at which the program produces data, but to the rate at which the memory subsystem accepts new load addresses. In the case of the highly overbanked system used in this experiment, this will approach unity (the shortfall being due to bank conflicts).

From this we can see that in the case of the TRFD experiment, peak decoupling will be reached immediately before the first value is returned from memory as the pipeline fills following a loss of decoupling. Since this time is dictated by the memory latency, it follows that peak decoupling in this situation is also dictated by the memory latency. The disparity between the memory latency in the example runs and the peak decoupling in each instance is due to the imperfect nature of the interleaved memory system. If bank conflicts occur they have no effect on r_a (in the presence of large bank queues, at least) but a substantial effect on the rate at which data arrives at the LDQ and thus on the effective r_e. From Figure 74 we can see that since the degree of decoupling reaches almost 90, bank conflicts must be occurring. It is difficult to estimate the precise number of bank conflicts that might cause this increase in the degree of decoupling since individual conflicts can be masked to a greater or lesser degree depending on how busy the memory pipeline is at the time and the pattern in which memory accesses are occurring.

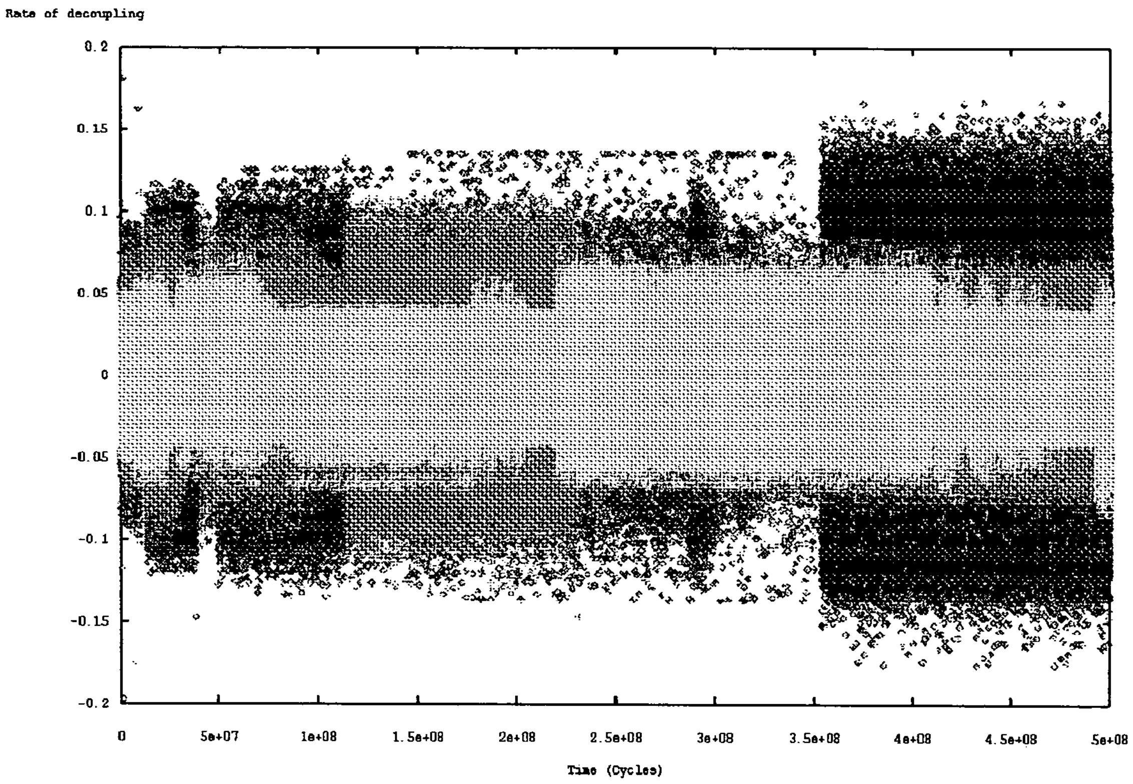


Figure 78 - Rate of decoupling for all three runs. (Ex. 10)

The measurements of the rate of decoupling (the data for all three runs is shown superimposed in Figure 78 with the lightest grey representing the shortest memory latency) yielded no great surprises, with areas showing a high degree of decoupling tending to have higher rates of decoupling and recoupling than others. In all cases the rates were somewhat low, peaking at around ± 0.1 decoupled operation per 1000 cycles. However, when it is considered that the maximum rate of decoupling possible is 1.0 (since it is not possible for the two units to separate by more than one load or store per cycle) these figures seem more reasonable, especially when it is remembered that a period of 1000 cycles might see several decoupling/recoupling sequences. Also of note is the way in which the periods of high decoupling are characterised by a (nearly) symmetrical pattern of decoupling/recoupling activity. This is not entirely unexpected, since anything else would indicate runaway decoupling or, alternatively, no decoupling at all. However, the symmetry of the pattern confirms the observation made earlier that these are in fact periods of fairly intensive decoupling/recoupling activity, the details of which are hidden due to the relatively low resolution of the experiment. It also suggests that the rate of recoupling is in most cases approximately equal to the rate of decoupling. This is explained by the rate of recoupling being governed by r_e, which like r_a is close to unity in the stripped-down annotation.

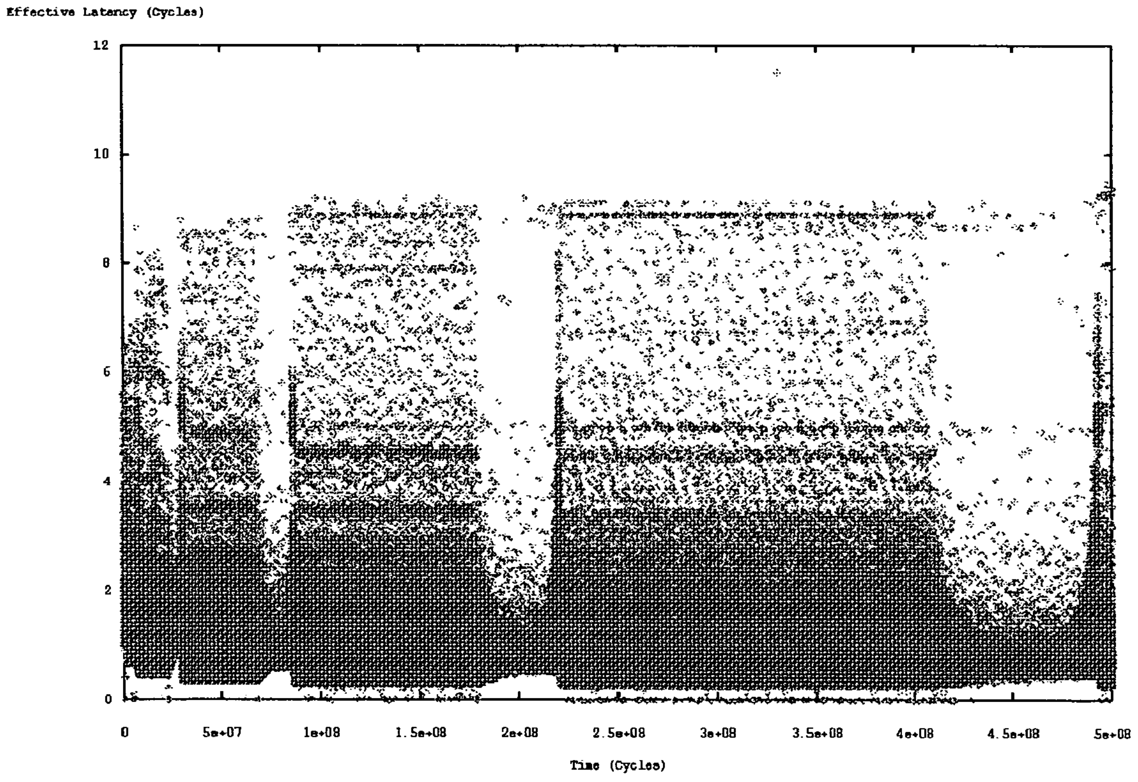


Figure 79 - Effective latency for 16 cycle memory latency. (Ex. 10)

How does one explain the apparent increase in the rate of decoupling when the memory latency increases, however? The rate at which the access unit issues loads is the same for all three latencies. This effect is due to the imprecision of the sampling technique used. The rate of decoupling that is calculated is not a precise measurement, but is obtained by comparing the level of decoupling at the start and end of each sample period and calculating the gradient. It has already been explained that higher memory latencies can lead to a higher degree of decoupling. If the two units are closely coupled at the start of a sample period, and at the end have decoupled to their limit, the higher memory latency run will show a higher rate of decoupling. This is misleading. In the simulation run with the lower memory latency, the peak degree of decoupling may have been reached some time before the end of the sample period. The fact that both simulation runs in fact demonstrated the same rate of decoupling is hidden "in the gaps" between the sample points.

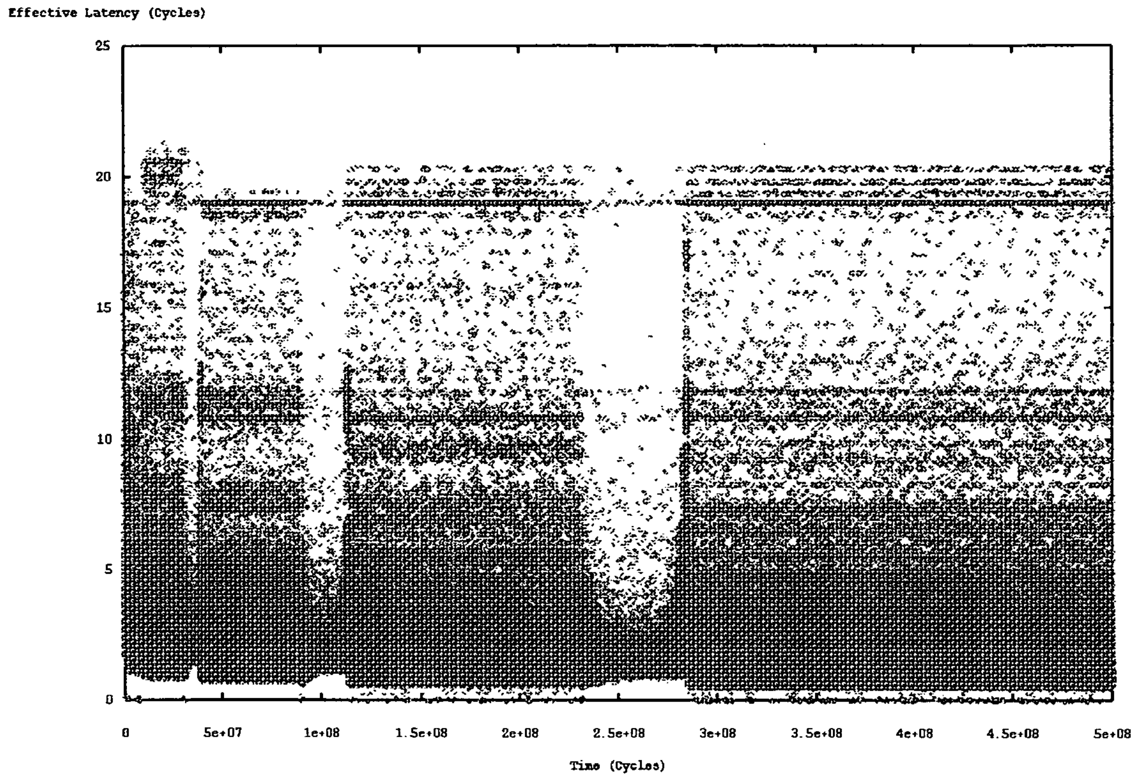


Figure 80 - Effective latency for 32 cycle memory latency. (Ex. 10)

The final dynamic measurement made was that of the effective latency, as seen by the execution unit. This differed slightly from the other characteristics measured in that the value obtained at each sample point was the average of all latencies measured since the previous sample. This is more accurate and was possible because it involves only one of the work units - calculating the other statistics (which involve two units) measured on a per memory access basis would have required some changes in the simulator architecture which were not easily made.

The behaviour of the effective latency over time is an interesting one. Periods of low effective latency match up not with periods of high decoupling, but with troughs in the degree of decoupling graph. Similarly, high effective latency appears to accompany a high degree of decoupling. This is illustrated in Figure 79, Figure 80 and Figure 81, which show the effective memory latency for each of the three simulation runs. All three sets of data are shown superimposed in Figure 82, which clearly illustrates a strong correlation between actual memory latency and the highest effective memory latencies.

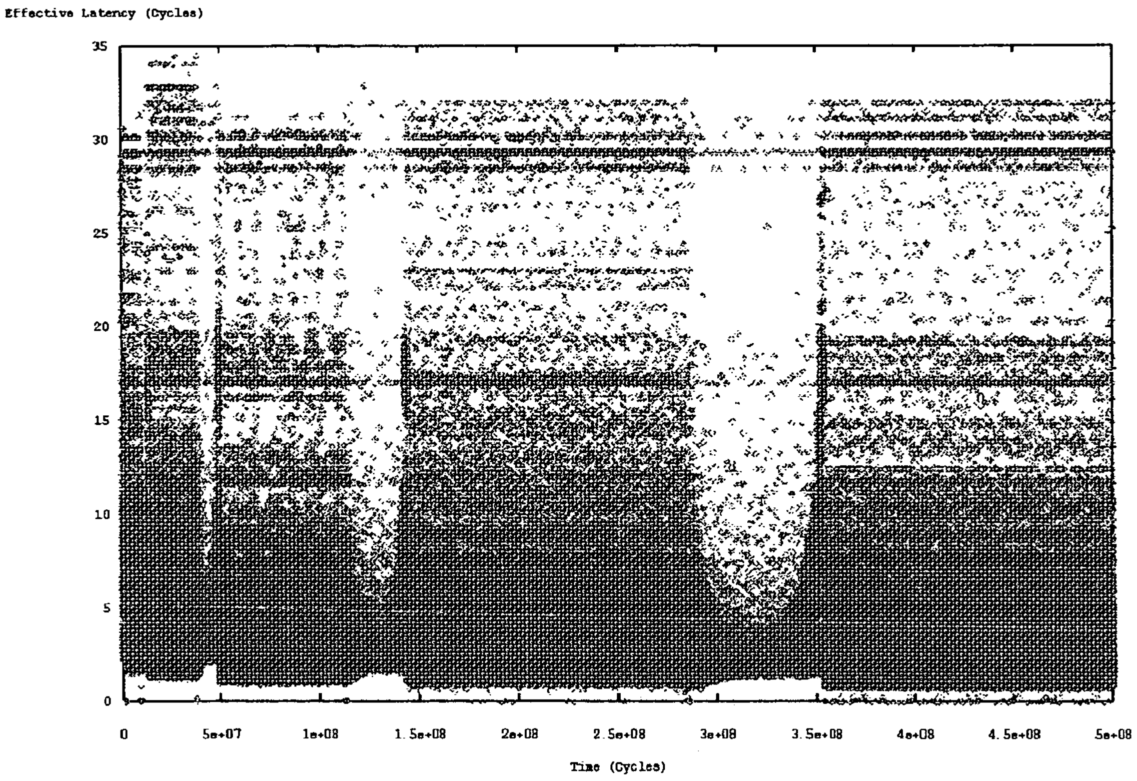


Figure 81 - Effective latency for 48 cycle memory latency. (Ex. 10)

The data gives an indication of what the degree of decoupling information shown in Figure 74 through Figure 76 actually indicates. It is not, as would be assumed, a measure of where the program is poorly and badly decoupled. As has already been observed, the low sampling frequency is such that individual decoupling or recoupling events cannot be detected. The expected peak degree of decoupling for a system with ample memory bandwidth and r_a and r_e is directly related to memory latency as has already been shown. Thus, one would expect the degree of decoupling in Figure 74 to remain around 16, perhaps with occasional drops where the program was poorly decoupled. It has already been indicated that higher degrees of decoupling result when bank conflicts allow the access unit to race still further ahead of the execute unit. What the peaks and troughs on figures such as Figure 74 are showing is not whether the program is poorly or well decoupled, but the frequency with which bank conflicts are occurring. Whether the program is well or poorly decoupled is hidden, both in the large expanses of program time between sample points, and in the solid areas of the decoupling graph.

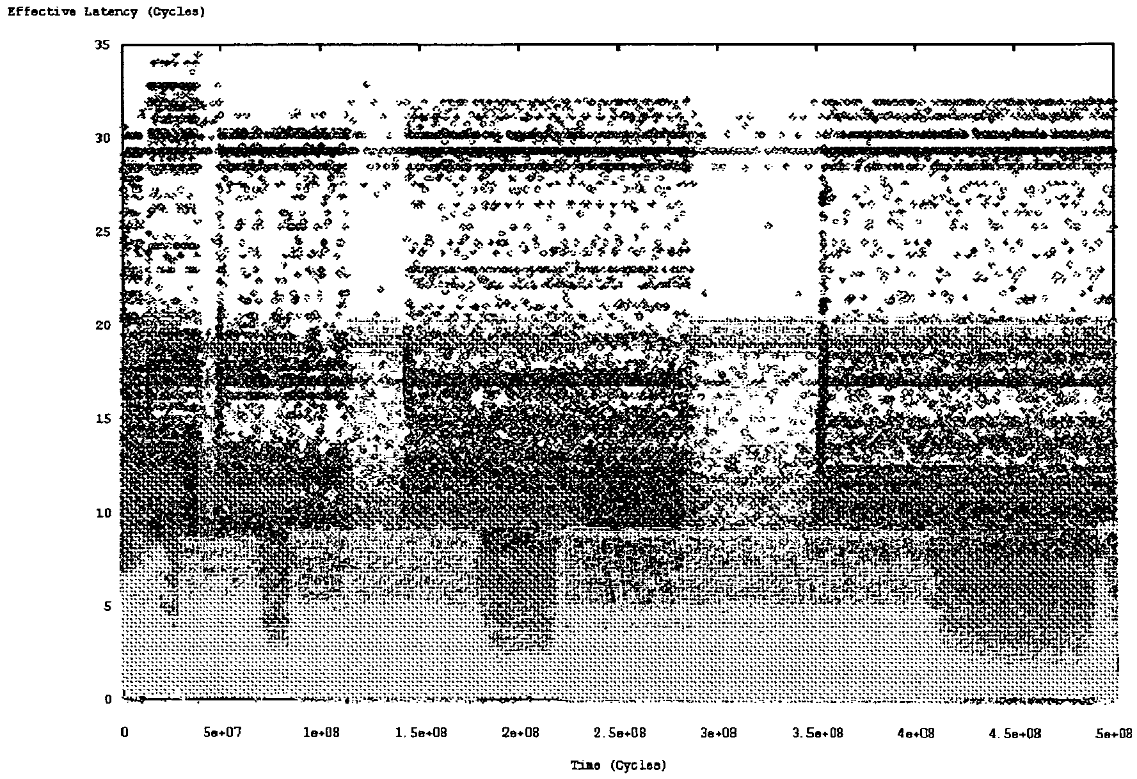


Figure 82 - Effective memory latency for all three runs. (Ex. 10)

This interpretation of the degree of decoupling information matches the effective latency measurements well. Where bank conflicts occur, the degree of decoupling increases for reasons already explained, and the effective latency increase as the bank interleaving becomes ineffective and the memory pipeline latency becomes visible (even if only for a single load) to the execution unit. The fact that the effective latencies remain consistently below the actual latencies for the systems being simulated is due to the averaging that takes place between each sample point. The increase in effective memory latency caused by a bank conflict is more localised than that caused by a loss of decoupling. The latter flushes the memory pipeline and the full latency shows in all subsequent loads until the memory pipeline fills again. Where a bank conflict occurs, an increase in latency is visible only to loads directly involved in the conflict - those that are held up in the bank's queue. The distribution of arrival times at a single bank (discussed in Chapter 4) and the distribution of loads over the various banks can all serve to reduce the average effective latency between sample points.

7.2.2 Conclusions

The most important conclusion reached from this experiment was that, for general measures of behaviour at least, program-level examination via simulation is impractical. The experimenter is faced with a choice between collecting prohibitively large volumes of data, or collecting data at a lower frequency and losing a great deal of the more useful information contained therein.

The use of a relatively low sampling frequency for the characteristics measured in this experiment obscured much of its detailed behaviour. That the resulting information can only be interpreted in the form of graphs gives some indication of the way in which only high-level trends can be examined. It was most disappointing that no clear mapping could be found between program features and the behaviour of the simulation, with the only visible structure being the differentiation of successive calls to the annotated subroutine in question.

More meaning might have been gleaned from this high-level data had the trace generation and simulation tools supported some form of higher-level annotation, perhaps allowing the passing of markers from trace to simulator to indicate when different phases of program execution had been entered, such as top-level loops, or entry to/exit from annotated subroutines.

However, it is still likely that even with this high-level annotation, little in the way of meaningful information could be extracted from a program-level simulation. This is illustrated by Figure 83. This shows the results of running a specially annotated version of TRFD. In this, each of the basic blocks within the `olda` subroutine had been assigned a unique numeric ID, and this was output whenever that block was entered. The accumulated block IDs thus gave a map of where program activity lay over time. The figure shows only the first million basic blocks in the program's execution. The limited display resolution means that each displayed point corresponds to approximately 1000 recorded block executions. As the figure shows, at this level it is extremely difficult to map program activity onto program structures, other than perhaps to spot successive calls to the subroutine as a whole.

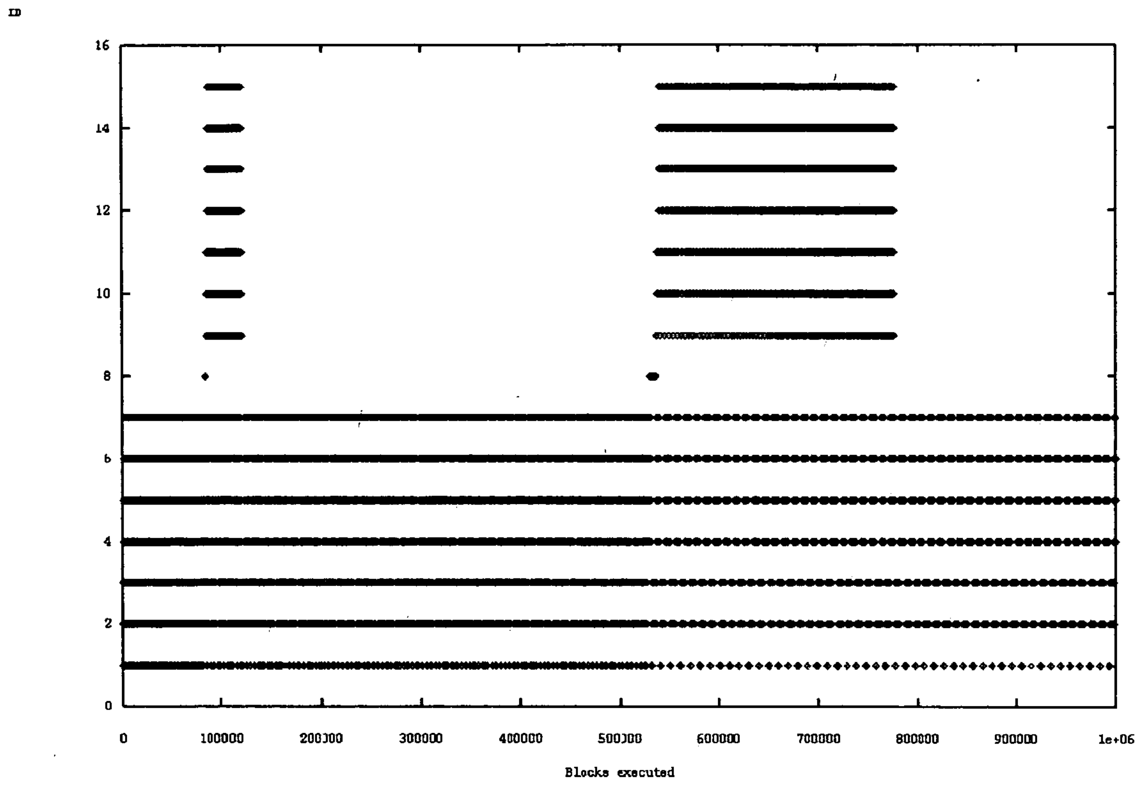


Figure 83 - Block execution activity for Ex. 10.

It should also be considered that the typical basic block in the minimally annotated TRFD used in the main experiment contained at most four or five instructions per unit. This, coupled with the low effective latency shown in the likes of Figure 79, means that the typical basic block execution time is unlikely to have exceeded thirty of forty clock cycles. Thus, Figure 83 might represent 40,000,000 clock cycles of the program's execution, or less than one tenth of the program execution time that was measured in the main experiment. When one imagines the data shown in this figure compressed to a tenth of its width, the scope of the problem of detecting program behaviour becomes even clearer - it is just too wide for analysis. When significant events can occur within a window of tens of clock cycles, spotting them within a 500,000,000 cycle simulation becomes even more difficult.

Despite the problems of program-level simulation, some results were obtained from this experiment. The effects of memory access patterns on effective latency were shown to be of some significance, and some basic results regarding the factors that determine the maximum rate of decoupling were shown. However, the minor significance of these results (which had been suspected previously, and could doubtless have been demonstrated through the use of simple kernels) did not warrant the large amount of computation that went into obtaining them.

7.2.2.1 An alternative to program-level simulation

That the raw, "brute force" simulation method failed to provide much insight into program-level behaviour does not, however, mean that it is impossible to determine such details. One possible alternative technique, which will be outlined here briefly, would use a combination of analysis and simulation to produce a model of program-level behaviour with greatly reduced computational requirements and more insight into the mapping of behaviour onto program structures.

In the majority of the experiments performed during the course of this research, simple program kernels with one, or sometimes two, modes of behaviour were examined. As has been seen, the jump between this and the complex multi-modal behaviour of real programs is a large one. An intermediate solution to this problem would use a constructive approach. All basic blocks fall into a small number of categories, depending on their decoupling characteristics. A basic block may potentially increase access-execute decoupling (if $r_a > r_e$), reduce decoupling ($r_a < r_e$), leave decoupling unchanged ($r_a = r_e$) or cause a loss of decoupling. Similar characteristics determine the decoupling between the control unit and each of the work units.

Given this information for individual basic blocks, flow control analysis can be used to determine which blocks follow which in the program being analyzed. Simulation of simple bi-modal kernels for all possible combinations of basic block decoupling characteristics can be used to determine how transitions between different block behaviours affect decoupling. From this, it may be possible to derive a simple model that associates with each basic block or basic block transition some notion of how it is likely to affect decoupling. Program-level decoupling behaviour can then be determined by constructing basic blocks into ever larger units, with associated decoupling information.

The estimate given would admittedly be static, and highly dependent on accurate profiling. However, since overall program behaviour would be synthesized bottom-up, it might prove easier to determine detailed behaviour since such behaviour would be identified at the level it first became apparent, rather than by attempting to "delve down" into a very general top-level collection of data. Alternatively, a greatly simplified annotation/simulation approach could be used where each basic block was annotated to output some description of the way in which it affected decoupling, and the resulting "trace" fed into a simulator that could keep track of current decoupling levels, memory subsystem state, and so forth. While this would suffer some of the shortfalls of the approach taken in this experiment at a program level, the annotation of the program in terms of the information in which we are interested (decoupling characteristics) rather than at an instruction level would both reduce the cost of simulation and aid interpretation. For example, if a large nested loop was known to have highly predictable decoupling behaviour, it could be so annotated, removing the need to simulate the execution of individual iterations.

Unfortunately, time constraints have prevented a more detailed examination of this possible approach to the examination of program-level decoupling.

8. Conclusions & Future Work

8.1 Conclusions

In this thesis, the performance of a variety of fully decoupled architectures has been examined. The factors which determine the performance of these architectures depends largely on the type of decoupling with which we are concerned. The performance of access/execute architectures depends on the total memory throughput, the length of the LDQ and the frequency and cost of loss of decoupling events. The former determines whether the memory subsystem is able to keep the data sub-processor supplied with load packets at the same rate that the address sub-processor supplies them. Since the memory data path is in essence the combination of two decoupled processes - one connecting the address sub-processor to memory, the other connecting memory to the LDQ - the memory subsystem should not act as a bottleneck in the larger decoupled system formed from the two decoupling processes. In the systems examined, the banked and interleaved memory subsystem had to have sufficient banks that it could cope with a load address packet arriving on every clock cycle and be able to supply the data sub-processor with a load data packet at the same rate. This typically requires that the number of banks match the latency of a memory bank divided by the processor clock speed.

The other requirement was imposed by the load tagging system used. This required that each active load be assigned a unique tag which would allow it to be re-ordered in the LDQ upon arrival. The number of active loads was thus limited by the number of available tags which was in turn set by the LDQ size. Generally speaking, the longer the LDQ the better, although increasing the LDQ length beyond the total "capacity" of the memory sub-system produced no gains in performance.

Access/execute decoupling was found to be lost in a number of situations where data has to be shared between the two work sub-processors. This typically occurred when indirection or list following was performed, or when a variable used by the data sub-processor was also required for use in an address calculation.

The characteristics determining control decoupling were similar, although it was not possible to examine these in quite as great a degree of detail due to limitations in the simulation techniques used. The length of the MIQ was found to limit the degree of control decoupling possible in a similar manner to the LDQ. If it should prove necessary to implement some form of re-ordering scheme for the results of expanding a meta-instruction, the length of the instruction buffer into which sub-processor level instructions are to be delivered may also be of some import. Once again, it is important that the memory sub-system have sufficient bandwidth that the work sub-processor can be kept supplied with instructions.

Control decoupling was lost both in situations similar to those encountered for access/execute decoupling, where a variable is shared between two units, and in certain fixed control-flow situations, where a decision about the execution of the program had to be made on one or other of the work processors. It was also found that a loss of control decoupling implicitly caused a loss of access/execute decoupling.

A number of techniques for reducing the effects of LODs were examined. Some of these, such as the replication of code across two or more sub-processors and the provision of dedicated paths to memory, were common to both forms of decoupling. Others, such as strip decoupling and guarded execution were specific to one or other form of decoupling.

The techniques for the elimination or reduction of LODs examined fell into two primary categories - software based and hardware based. The former, although often limited in application and effectiveness, were easily implemented on a basic architecture and are thus likely to be of use across a wide range of decoupled architectures. Hardware techniques, however, require that special provision be made in the architecture. While these techniques were generally at least as good as equivalent software techniques and often better, their potential cost and their additional complexity may well prove prohibitive.

An examination of their actual costs would require a more detailed examination of decoupling, using full programs rather than the reduced kernels that were used in these experiments. The frequency with which the various varieties of LOD occur and the actual decoupling available would have to be determined for a number of the programs which would comprise a typical workload for a decoupled processor. Detailed investigation would also need to be made into the hardware cost of the techniques investigated, and whether the performance gains they allow would be justified.

In this thesis the behaviour of fully decoupled architectures with specific regard to decoupling and loss of decoupling has been examined. The results presented here detail the basic behaviour of this class of architectures. Determining their behaviour in real-life situations is a more complex task and one that may prove to be determinable only in an empirical manner. However, it is hoped that the material presented here can form a basis on which future studies can build, and which may lend a more solid foundation to empirical results. It is regrettable that attempts to derive a useful analytical model proved abortive. It may be that some method exists by which a tractable and exact analytical model can be obtained. However, this is unlikely to be easy to derive and may prove so complex as to be of limited practical use when compared with simulation techniques.

8.2 Some possible future work

While the bulk of this thesis has been concerned with an examination of the basic behaviours of fully decoupled architectures and straightforward solutions to some of their problems, a number of areas for possible future work arose during the course of the research. These are presented below, in rough form only.

8.2.1 Instruction caching mechanisms for decoupled architectures

Both instruction and data caching rely on two principles - those of spatial and temporal locality - to improve performance. The first of these states that 90% of memory accesses are to only 10% of the memory used by a program (and this can be exploited by arranging for that 10% to be available in a fast cache) and that locations accessed in the recent past will be near to those we'll access in the near future, while the second states that if we've accessed a particular memory location in the recent past, we're likely to do so again in the near future. Yet both of these are little more than (good) heuristics used to predict the future pattern of accesses to the cache and are limited by the fact that they attempt to predict the future based only on past behaviour. If it were possible to know the future behaviour of a program as well as the past, cache performance could be improved.

The asynchronous nature of the sub-processors in a fully decoupled architecture can potentially make just such knowledge available. Since in most circumstances the control sub-processor is capable of "running ahead" of the data and address sub-processors, the meta-instruction queue will usually contain a partial future history of the execution of the corresponding sub-processor. Also, since each meta-instruction contains information that completely specifies the basic block (not only the start address, but also the length and in some cases the number of times the block will be executed) this "limited precognition" covers both temporal and spatial information. The execution order of the next *n* blocks is known in advance, as well as their memory requirements. In a system that allows loop iteration counts to be specified in the meta-instruction, the meta-instruction queue may contain the future history of the sub-processor for hundreds of thousands of clock cycles in advance.

With this information available, it may be possible to design a cache controller that can either obtain an extremely high instruction cache hit rate for a given cache size, or provide an acceptably high hit rate from a much smaller instruction cache. The information in the MIQ may be used to preload basic blocks long before they are actually required and to dispose of them as soon as they are no longer needed, allowing available cache memory to be managed far more efficiently and improving the hit rate. The additional information provided by the MIQ may also make it possible to explore new cache designs taking advantage of the unit of program execution being a basic block rather than a single instruction.

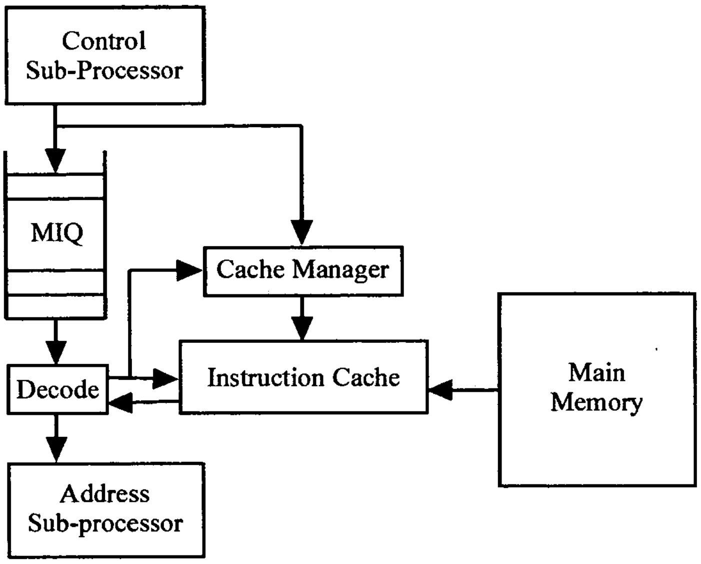


Figure 84 - An instruction cache preload mechanism.

Figure 84 shows the block diagram of a possible fetch engine architecture that attempts to perform cache preloading. Meta-instructions, in addition to being queued in the MIQ, are also routed to the cache manager. This unit, through mechanisms which will be outlined later, checks to see if there is room for the block in the instruction cache and, if there is, begins to preload it into the instruction cache from memory.

As meta-instructions reach the head of the queue they are passed to a block of decode logic. This breaks the meta-instruction down into a stream of sub-processor operations and fetches the appropriate instructions from the cache which it passes directly to the sub-processor (in this case, the address sub-processor). When the decode logic finishes processing a meta-instruction it informs the cache manager so that it may, if necessary, free the cache space used by the basic block.

By recording the information in each meta-instruction as it arrives and noting the completion of each meta-instruction as the decode logic finishes with it, the cache manager is able to maintain a record of which blocks should be in the cache and when they can be safely removed.

Since a decoupled processor deals with programs at a slightly higher level than a conventional architecture (the basic unit of execution from the control sub-processor/MIQ point of view being a basic block), it may be possible to take advantage of this. In a conventional architecture there is no guarantee that the next instruction fetched into an instruction cache will actually be used. The probability that a fetched instruction will actually be used is high until instruction l (where l is the average length of a basic block) of the basic block is fetched, then the likelihood of use drops away. Typically a well-designed instruction cache will fetch instructions in blocks of approximately l (or some multiple of l) instructions, hoping that a single cache line load will suffice to load the entire basic block. If the block turns out to be longer than l instructions in length, further cache miss penalties may be paid but this is usually considered preferable to fetching instructions which will never be executed. When the block length is shorter than l internal fragmentation will occur in the instruction cache, wasting space.

When blocks are specified by meta-instructions, however, the probability distribution changes. For a block of length l, instructions 1 to l are certain to be executed while instruction $l+1$ will never be executed. Thus it should in theory be possible to arrange the cache in such a way that no redundant instructions are loaded. In practice this is unlikely to be the case since, regardless of whether a conventional or novel cache architecture is used, it is likely that the memory$\rightarrow$ cache bus will be some multiple of the word size in width and that internal fragmentation may still occur because of this. However, this fragmentation may be considerably less than that brought about by the loading of an entire cache line.

It is the duty of the cache manager to do its best to ensure that each basic block is in memory by the time the corresponding meta-instruction is decoded, that the cache memory is used efficiently. To do this, the cache manager must ensure that each block is present only once in the cache, that there is minimal fragmentation (internal or external), that no preloaded block is overwritten until it has been used and that cache space allocated to a basic block is freed once that block has been executed.

The above requirements suggest that the cache manager design may be quite complicated. It will need to maintain some record of the current contents of the cache, either in its own internal structures or via access to a cache TLB. It will require direct access to the meta-instruction queue itself and may require that it can interrogate the MIQ associatively. Finally, it will also need to cooperate closely with the fetch engine to establish when the execution of each meta-instruction has completed.

If the cache manager were as complex as indicated above, careful experimentation would be required to establish whether the performance improvements produced by the use of a “smart” high-level cache management scheme outweighed the high hardware cost. It may well be the case that a straightforward conventional instruction cache would function adequately in most situations and that the silicon area that the cache manager would require would be better used to increase the size of the I-cache.

8.2.1.1 Multi-threaded decoupling

In Chapter 2, the subject of multi-threading was briefly touched on. To recap, this technique involves the use of multiple hardware contexts with a rapid context switch to “bulletproof” a processor against long memory latencies. Whenever an unavoidable long delay is encountered, the processor switches contexts to another hardware thread and starts executing that instead, reactivating the previous thread only when the delay has passed.

Although this technique has a number of problems, such as the limitations it places on per-thread throughput and the large areas of silicon required to maintain multiple contexts, it is a perfectly viable alternative to decoupling as a means of obtaining latency tolerance. In fact, since the two techniques are orthogonal, there is no reason that they cannot be combined.

An architecture that combined both techniques would use decoupling to reduce the frequency with which context switches must be performed. The situations that might cause context switches would differ from processor to processor, roughly as follows:

Control

The control sub-processor can stall for two main reasons - being unable to store a meta-instruction into a full MIQ and waiting for a value to return from one of the work processors.

In the former situation multi-threading provides no advantages. The queue is full, and switching contexts will not cause it to empty (unless multiple queues were provided - this seems wasteful though). If there are many control sub-processor instructions between each meta-instruction issue some advantage may be gained from switching context, but it is unlikely that there will be sufficiently many to justify the cost of the context switch. However, a control LOD seems to provide an ideal opportunity to use multi-threading. If the control sub-processor attempts to read one of the CFQs and finds it empty it switches context and continues execution in that context.

This will require some extra information to be passed around the system. Meta-instructions issued to the work-processors will need to contain a tag identifying the context in which they were issued and this information will in turn be used to ensure that the work processor are in the appropriate context while executing that block. Additionally some mechanism will be required to ensure that as values are added to the CFQ by the work processors the threads which were stalled awaiting their arrival are awakened ready for execution.

In more sophisticated architectures (i.e. Levels 2 and 3) multi-threading could be extended to mask the latency of control sub-processor self-loads that miss the cache.

Address

The address sub-processor can stall for at least six reasons. It can be waiting for a value to be passed via the TRQ from the data sub-processor, it can be waiting for a meta-instruction from the control sub-processor to be expanded by the fetch engine, or it can be stalled by trying to store a datum into the CFQ, TRQ, LAQ or SAQ. There is little that can be done about the situation where the instruction fetch has not yet completed and it is probably best that no context switch occurs in this situation since the next thread activated will likely find itself in a similar quandary. The stall resulting from a $DU \rightarrow AU$ LOD seems like an ideal opportunity for a context switch, however, and it would seem reasonable to perform one whenever a read from the TRQ is attempted and no datum is present.

As for the other situations in which a stall can occur, the desirability of performing a context switch on these will depend largely on their frequency and the distance between operations which may cause them. Switching contexts because the LAQ is full will have little benefit if the next thread that is activated immediately tries to store to the LAQ and is itself suspended. The practicality of switching contexts in these situations is directly proportional to the distance between instructions that can cause stalls of this type.

As with the control sub-processor, multi-threading could also be used in more sophisticated architectures to mask the latency of address sub-processor self-loads.

Data

The data sub-processor can stall if no instructions are available, if it is awaiting a datum from the TRQ or LDQ, or if it tries to write the SDQ or TRQ and finds it full. Little can be done in the first situation while the second seems a good candidate for a context switch and the appropriate actions for the third type of stall will depend, as for the other sub-processors, on the distance between operations of this type.

Implementing multi-threading on a decoupled architecture will require a considerable amount of additional hardware. By far the largest proportion of this will be accounted for by the replicated register files. Each sub-processor will also require status information for each hardware thread that it possesses regarding whether it is running, stalled, ready or inactive. The format of meta-instructions, memory packets and even data packets passed through the TRQ or CFQ will also need to be extended to include information indicating which context the packet is part of.

The combination of both latency tolerance techniques seems to overcome some of their limitations. The primary failing of decoupling is the high penalty incurred when a loss of decoupling occurs. The use of multi-threading may allow this penalty to be masked. Multi-threading's weaknesses include the need to maintain a large number of hardware contexts due to a high rate of context switching and low per-thread throughput. If context switches occur only on LOD events and these are less frequent than the sort of event that would cause a context switch in a conventional multi-threaded material (i.e. a cache miss) fewer contexts will be required and each thread will run for a longer time before forcing a context switch.

Since the LOD cost is high, even when compared with the cost of a cache miss, it may also be practical to use low-cost context switching mechanisms that do not require multiple copies of the register set. It may be that only part of the register set is replicated and the rest is switched out to memory. While this is potentially expensive, its cost may be considerably lower than the total LOD penalty.

8.2.1.2 Scheduling

The scheduling of threads is complicated by the fact that, if we take the arrival of the "missing" datum that caused a thread to stall in the first place to be the event that causes it to be rescheduled or marked as ready to run, conflicts may occur. In a conventional multi-threaded architecture context switches are usually provoked by cache misses. The thread remains suspended until all the other ready threads have had their turn, by which time the requested data will hopefully have been loaded into the cache and execution may proceed. The arrival of the requested data results in the thread being rescheduled.

This approach is perfectly acceptable for sub-processor self-loads and even (should we extend the multi-threading to cope with this) stalls due to unavailable instructions where the threads do not have to be rescheduled in any particular order for progression to be made. However, it is *not* particularly well suited to stalls caused by accesses to "ordered" data structures such as the LDQ, TRQ and CFQ. In these cases it would seem more efficient to schedule threads on the basis of the context of the next available datum. This can be seen as a variation on coarse-grained dataflow where the arrival of data actives a thread (which remains active until it encounters another stall) rather than a single instruction.

However, if a sub-processor uses this data driven scheduling approach, what do we do when the data at the head of two queues (for example the LDQ and TRQ) are in different contexts? Whatever scheduling algorithm was adopted would have to ensure fairness, progression and be deadlock-free - the standard requirements of any multi-programming system.

Staying with the subject of deadlock, while a system with a single load path and a single store path as described in this thesis would not be likely to deadlock, more complex decoupled architectures with multiple load paths could experience problems. For an example of this see Figure 85. The two LDQs are shown. Each entry in each LDQ is labelled with the context in which the load packet belongs. Also shown are the next instructions from two hardware threads, contexts 1 and 2. Assume that the arrival of a load packet at the head of L0 has activated context 1. The processor checks to see if both the requested operands are available. They are not, since the head of L1 is a context 2 packet. Context 1 is suspended and context 2 is activated. This also checks to see if both operands are available. They are not, since the head of L0 is a context 1 packet. The processor deadlocks either because no other thread can get at operands "below" the head or because contexts 1 and 2 alternately try to acquire both operands and fail.

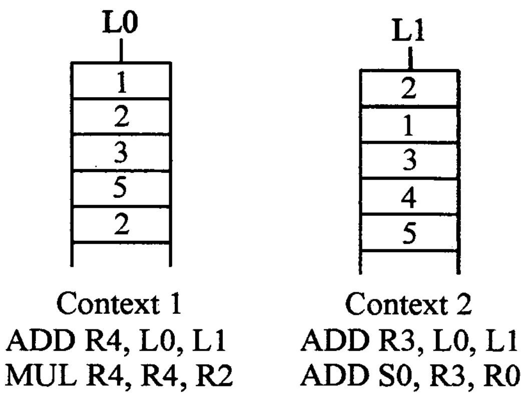


Figure 85 - Deadlock in a multi-threaded dual load path decoupled architecture.

However, it *may* be possible to avoid deadlock if it is possible for an instruction to read only one of its operands (and set a flag accordingly) before stalling. If this were the case, the context 1 instruction could read the head of L0 and would then suspend since its second operand was not available. However, when context 2 was then activated it would be able to read both its operands. This would in turn move the second operand of the context 1 instruction to the head of the L1 queue, allowing it to be read when context 1 is next scheduled.

Unfortunately a similar and less easily avoided deadlock problem can occur with stores, even on a single load/store path architecture. Stores are only sent to memory when both data and address halves of the store packet have been produced. With a single-threaded architecture it is guaranteed that the data and address parts of the packet will arrive in the correct order (even though the address half of the store may arrive before the data half). This is not the case with a multi-threaded architecture, where the fact that the two sub-processors may be in different contexts at the same time means that the store address queue could fill with store addresses from context 1 while the data sub-processor is in context 2. As soon as the data sub-processor produces the data half of a store the machine will deadlock. No further addresses can be added and the data cannot be stored to memory because there is no matching address in the queue.

The elimination of the SDQ may present a solution, although it is not an elegant one. When the data sub-processor generates the data half of a store packet, this is compared associatively (matching on context ID) with the SAQ. If an address from this context is present, it is matched with the data and sent to memory. If no address is present, the thread is suspended and the next thread begins execution. This prevents deadlock but has a number of shortcomings. It may affect runtime adversely (since the two contexts will effectively become sequentialised) and the hardware to associatively match against the SAQ and return the address of the *first* matching entry would be quite complicated.

A further problem arises when we consider the decoding of meta-instructions. A multi-threading decoupled architecture would associate with each meta-instruction the tag of the context in which it was issued. However, there is no guarantee that, when the meta-instruction at the head of the MIQ is decoded and expanded into a stream of sub-processor instructions, the appropriate sub-processor will be in that context. As has been described above, context switching would most likely be performed in a data-driven manner on the data sub-processor and whenever a LOD event was encountered on the address sub-processor.

This suggests that the sub-processor instruction fetch engine will need to keep the first meta-instruction *for each possible context* ready since there is no guarantee that the context ID of the meta-instruction at the head of the MIQ will match that of the processor. This may be implemented through the use of multiple (shorter) MIQs but seems to be an expensive solution. It also complicates the more sophisticated instruction cache management schemes outlined in Section 8.2.1.

One possible approach to these problems might be to split the decoding of meta-instructions into two phases. In the first phase, meta-instructions are placed in the MIQ by the control processor. At this point they may either be sorted by context ID (obtained from the control processor) into separate MIQs, or left unsorted in a single unified MIQ. This information is used by the fetch engine to preload the instruction cache. Once this has been done, meta-instructions are passed to a second set of MIQs. When the sub-processor requests instructions, the appropriate meta-instruction is decoded and the cache accessed.

Since meta-instructions in the second set of queues have already been pre-loaded into the instruction cache, keeping these queues short in no way reduces the amount of lookahead available. The lookahead is still determined entirely by the first (set of) queues. The cache pre-loading mechanism is invisible to the sub-processor since it sees only those meta-instructions that have already pre-loaded the cache.

It remains the case, however, that even this approach requires a considerable amount of additional hardware. This may not be sufficient to justify its use. A considerable amount of research would be required to determine what quantifiable benefits, if any, would emerge from the use of multi-threading on a decoupled architecture.

8.2.1.3 Non-specialised decoupled processing

Many of the problems of the fully decoupled architectures examined in this thesis can be traced, albeit indirectly, to the specialisation of the three sub-processors and, indeed, to the fact that there are only three sub-processors.

The advantages of decoupled processing lie in the building of a "high-level pipeline", splitting the operations to be performed in a program into three separate phases, control, addressing and data manipulation. By running the control sub-processor ahead of the other two, and the address sub-processor ahead of the data sub-processor, the three sub-processors form a pipeline with the processing of a datum (or series of data) starting on the control sub-processor, being passed to the address sub-processor and finishing up on the data sub-processor. While the address sub-processor is operating on a datum, the control sub-processor may be working on the next datum to be processed and the data sub-processor on the previous.

Problems occur when the required high-level pipeline is more complex. The most simple pipeline, described above, can be written as $C \rightarrow A \rightarrow D$, with each datum flowing from control to address to data sub-processors. However, if the required high-level pipeline is along the lines of $C \rightarrow A \rightarrow A \rightarrow D$ (it requires an indirection) or $C \rightarrow A \rightarrow D \rightarrow C \rightarrow A \rightarrow D$ (the processing of the datum requires a control decision), we find that loss of decoupling occurs since the required pipeline differs from that for which the architecture is optimised ($C \rightarrow A \rightarrow D$) and the pipeline is, in effect, being asked to run backwards in places.

It has been shown that some of these problems can be eliminated by the technique of strip decoupling, which effectively time-slices the address sub-processor, allowing it to function as two stages in the pipeline. However, this technique is limited in its applicability, can cause deadlocks, and also requires that the combined execution time of the two pipeline stages on the address sub-processor be less than that of the single stage on the data sub-processor for decoupling to occur.

It appears to be the case that many of the problems of decoupled architectures arise from there being insufficient units of the correct type to efficiently implement the required "high-level pipeline". It would certainly be possible, for example, to produce an architecture containing two address sub-processors that could implement the pipeline required for a single level of indirection efficiently and with no loss of decoupling. However, unless such a machine was to be used solely for the computation of a class of problems that performed a great deal of single-level indirection, it would be unnecessarily specialised, and the second address sub-processor would be idle for much of the time.

Non-specialised decoupled processing would attempt to improve performance by providing an architecture from which a wider variety of high-level pipelines can be constructed. Rather than three dedicated sub-processors, an NSDP machine would provide a number of configurable processing elements, similar to that shown in Figure 86. This element is a generalisation of the specialised units of a fully decoupled architecture, capable of acting as a data sub-processor, an address sub-processor or a control sub-processor. A number of these elements, connected to a flexible interconnection network, could be configured to provide the required high-level pipeline for any particular epoch of program execution.

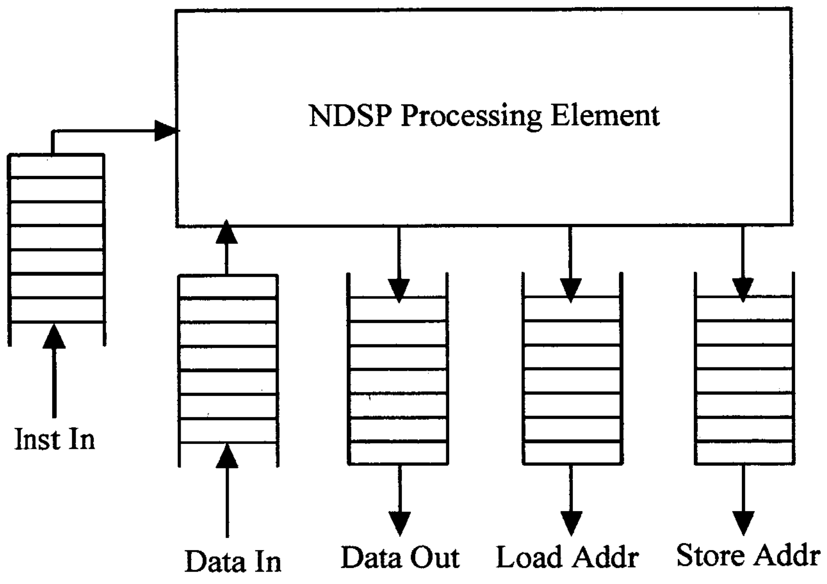


Figure 86 - NDSP processing element.

Each element is connected to the interconnection network by a number of queues. These queues (and the network) can contain a number of different packet types. Load packets differ little from those in a conventional decoupled processor, and specify the address in memory to load from, as well as an LDQ position. However, in an architecture where the data read from memory may potentially be delivered to any one of a large number of processing elements, it will also be necessary to specify the identity of the unit to which the data element is to be delivered.

The implementation of decoupled stores would require that the interconnection network be configurable in such a way that elements acting as address sub-processors and those acting as data sub-processors could be paired up to produce store packets. How this might be efficiently implemented has not yet been investigated.

Other required packet types would include a direct transfer packet, allowing an element to transfer a value directly to the Data In queue of another element, and a meta-instruction packet, allowing an element configured as a control sub-processor to send meta-instructions to its nominated work units.

The Data In queue would be used to store both incoming load data packets and direct transfer packets. In a straightforward pipeline of processing elements there would be no problem with packets from different sources becoming mixed, since each element would have at most one source and one sink. The distribution of signals regarding tag availability (one feature with requires relatively close co-operation between two units in a conventional decoupled architecture) may pose a problem here, but could be achieved relatively easily in small or medium-scale NSDP systems through the use of a dedicated 1-bit-per-element bus, with a pulse on the appropriate line indicating the removal of an element from the Data In queue of a particular processing element.

The Data Out queue would be used store the data portion of store packets, outgoing direct transfer packets and meta-instruction packets. As has already been indicated, the implementation of a decoupled store may cause some problems due to it requiring the "linkage" of two processing elements to produce complete store packets, but the other two types of packet are unlikely to cause major difficulties. Both would pair their data with a destination address (containing only a processing element ID - no memory address would be required since no memory would be involved) taken from the Load Address queue of the same processing element.

The Load Address queue is perhaps misleadingly named, since it would be used to buffer both the addresses and destinations for loads and the destinations for direct transfers and meta-instructions. The implementation of decoupled loads on a machine of this type would be straightforward, with load address packets being removed from the Load Address queue and sent to the memory subsystem in much the same way as in a conventional decoupled architecture. However, after a load request had been fulfilled by the memory subsystem, the load data packet would be routed (using information provided in the original load address packet) to the Data In queue of the appropriate processing element.

To perform a direct transfer, the address part of a transfer packet would be combined with the appropriate data part of the packet from the Data Out queue and the result routed directly to the Data In queue of the destination processing element. A similar procedure would be followed for the delivery of a meta-instruction packet, with the important difference that the packet thus assembled would be routed not to the Data In queue but to the Instruction In queue of the destination PE.

The above assignment of roles to queues is only a very approximate one, designed to give a rough idea as to how an NSDP architecture would load, store and transfer data. Depending on the relative cost of components, the differing sizes of the various packet types and their relative importance, it may be more desirable to have queues dedicated to a particular packet type.

Each processing element would be equipped with an instruction fetch engine capable of operating in two distinct modes. It would be able either to fetch instructions directly, functioning in the same manner as a conventional processor, or expand arriving meta-instructions and fetch the instructions thus specified for execution. Thus any processing element may function either independently, or as a slave sub-processor to another element.

With all of the above implemented, the individual processing elements that would make up an NSDP architecture would be capable of behaving as conventional uniprocessors, or as any of the three types of sub-processor that makes up a fully decoupled architecture. Additionally, their greater flexibility and number would allow the construction of more complex high-level pipelines by chaining suitably configured units together. This would include not only pipelines implementing indirection or control decisions, but also more complex pseudo-pipelines or multiple pipelines.

Take, for example, the implementation of an array operation for which the data part of the loop body had half the execution time of the address part. In a conventional decoupled architecture this would result in the data sub-processor spending much of its time idle and its effective iteration time would drop to that of the address sub-processor. However, it might be possible to configured an NSDP in such a way that two processing elements were assigned the address sub-processor role, each generating half of the addresses for the array. The receiving processing element, possibly using a context switching mechanism as described in Section 8.2.1.1 to distinguish between load data packets from the two address sub-processors, would then be kept busy.

Additionally, a system with $3n$ processing elements could be configured as n separate fully decoupled processors, each processing $\frac{1}{n}^{\text{th}}$ of the array. A configurable NSDP architecture would combine the flexibility and power of an equivalent shared memory multiprocessor with the latency tolerance of a fully decoupled architecture. Although no research has yet been done into the practicality of such an architecture, or whether the performance improvements thus achieved would outweigh the technical problems that such an architecture would face, such a generalised approach to decoupling may offer an alternative to more conventional shared memory multiprocessors.

Bibliography

[A92] *Alpha Architecture Handbook*, Digital Equipment Corporation, 1992.

[Aho86] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Arv91] Arvind, L. Bic, T. Ungerer, "Evolution of Dataflow Computers". *Advanced Topics In Data-Flow Computing*, Prentice-Hall 1991. pp 3-33.

[Ben91] M.E. Benitez and J.W. Davidson, "Code Execution for Streaming: An Access/Execute Mechanism". *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991. pp. 132-141.

[Ber89] M. Berry et al, "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers". *Technical Report CSRD 827*, Center for Supercomputing Research and Development, Illinois, May 1989.

[Bird91] P.L. Bird, U.F. Pleban, N.P. Topham and H. Scheuer, "Semantics Driven Computer Architecture". *Proceedings of the. International Conference on Parallel Computing*, September 1991.

[Bird93] P.L. Bird, A. Rawsthorne, & N.P. Topham, "The Effectiveness of Decoupling". *Proceedings of the International ACM Conference on Supercomputing '93*, July 1993.

[Cyb90] O. Cybenko, L. Kipp, L. Pointer and D. Kuck, " Supercomputer Performance Evaluation and the Perfect Benchmarks". *Proceedings of the International Conference On Supercomputing 1990.*

[Dub94] Dubey, P.K.,Arvind Krishna and Flynn, M.J. "Analytical Modelling of Multithreaded Pipeline Performance". *Proceedings Of The 27th Annual Conference On System Sciences, Vol I.*, 1994. pp. 361-367.

[Chen95] T.-F. Chen, "An Effective Programmable Prefetch Engine for On-Chip Caches". *Proceedings of the 28th International Symposium on Microarchitecture*, December 1995.

[Cra96] S.P. Crago and A.M. Despain, "A High-Performance, Hierarchical Decoupled Architecture". *ACAL-TR-96-07*. November, 1996.

[Far91] M. Farrens and A. Pleszkun, "Overview of the PIPE Processor Implementation". *Proceedings of the 24th Annual Hawaii International Conference on System Sciences*, 1991.

[Far93] M. Farrens, P. Nico and P. Ng, "A Comparison of Superscalar and Decoupled Access/Execute Architectures". *Proceedings of the 26th Annual International Symposium on Microarchitecture*, 1993.

[Gan92] D. Gannon, "SIGMA II: A Tool Kit for Building Parallelizing Compilers and Performance Analysis Systems". *IFIP Transactions A-11, Programming Environments for Parallel Computing.* North-Holland, 1992.

[Goo85] J.R. Goodman, J.-T. Hsieh, K. Liou, A.R. Pleszkun, P. Schechter and H.C. Young, "PIPE: A Decoupled Architecture For VLSI". *Proceedings of the 12th International Symposium on Computer Architecture*, May 1985. pp. 20-27.

[Gru96] W. Grünewald, T. Ungerer, "Towards Extremely Fast Context Switching in a Block-Multithreaded Processor". *Proceedings of the 22nd Euromicro Conference*, September 1996. pp 592-599.

[HP90] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, 1990.

[Ibb82] R.N. Ibbett, *The Architecture Of High-Performance Computers.* Macmillan, 1982.

[Jou89] N.P. Jouppi, D.W. Wall: "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines". *ASPLOS*, 1989: pp. 272-282.

[Kur94] L. Kurian, P.T. Hulina and L.D. Caraor, "Memory Latency Effects In Decoupled Architectures". *IEEE Transactions on Computers*, October 1994. pp. 1129-1139.

[Kur91] K. Kurihara, D. Chaiken, and A. Agarwal, "Latency Tolerance through Multithreading in Large-Scale Multiprocessors". *Proceedings of the International Symposium on Shared Memory Multiprocessing, 1991.* pp. 91-101.

[Lau94] J.P. Laudon, *Architectural and Implementation Tradeoffs for Multiple-Context Processors.* Ph.D. Thesis, Stanford University Technical Report No. CSL-TR-94-634, September 1994.

[Mang91] W. Mangione-Smith, S.G. Abraham and E.S. Davidson, "A Performance Comparison Of The IBM RS/6000 And The Astronautics ZS-1". *Computer*, Jan. 1991. pp. 39-46.

[Moss94] P.D. Mosses, "A Tutorial On Action Semantics". Course notes for tutorial given *at FME '94 (Formal Methods Europe)*, October 1994.

[Pr88] S.A. Przybylski, A. M. Horowitz and J. L. Hennessy, "Performance Tradeoffs in Cache Design". *Proceedings of the 15th Symposium on Computer Architecture,* 1988. pp. 290-298.

[Rau91] B. R. Rau, "Pseudo-Randomly Interleaved Memory", *Proceedings of the 18th International. Symposium on Computer Architecture*, May 1991. pp. 74-83.

[Sk92] I. Sklenar, "Prefetch Unit For Vector Operations On Scalar Computers". *ACM SIGARCH*, Sept 1992. pp. 31-37.

[Smit82] J.E. Smith, "Decoupled Access/Execute Computer Architectures". *Proceedings Of The 9th Annual Symposium on Computer Architecture*, 1982.

[Smit84] J.E. Smith, "Decoupled Access/Execute Computer Architecture". *ACM Transactions on Computer Science Vol 2*, 1984. pp. 289-308

[Smit87] J.E. Smith, G.E. Dermer, B.D. Vanderwarn, S.D. Klinger, C.M. Rozewski, D.L. Fowler, K.R. Scidmore and J.P. Laudon, "The ZS-1 Central Processor". *ACM SIGARCH*, 1987.

[Smit89] J.E. Smith, "Dynamic Instruction Scheduling And The Astronautics ZS-1". *Computer*, July 1989. pp. 21-35.

[Smth82] A. Smith, "Cache Memories". *ACM Computing Surveys*, March 1982.

[Top95] N.P. Topham and K. McDougall, "Performance of the Decoupled ACRI-1 Architecture: the Perfect Club". *Proc. HPCN - Europe*, May 1995, LNCS 919, Springer-Verlag. pp. 472-480.

[Tri82] K.S. Trivedi, *Probability & Statistics With Reliability, Queuing And Computer Science Applications*. Prentice-Hall, 1982.

[Tys92] G. Tyson, M. Farrens and A. Pleszkun, "MISC: A Multiple Instruction Stream Computer". *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 1992.

[Wal88] J. Walrand, *An Introduction To Queueing Networks*. Prentice Hall, 1982.

[Wol92] W.A. Wolf, "Evaluation of the WM Architecture". *Proceedings of the 19th Annual Symposium on Computer Architecture*, 1992. pp. 382-390.

[You88] H.C. Young, "Code Scheduling Methods For Some Architecture Features In PIPE". *Microprocessors and Microprogramming*, January 1988. pp. 39-63.