Trace-driven Simulation of Decoupled Architectures

S. Manoharan*
Advanced Computer Research Institute
1 Boulevard Marius Vivier-Merle
69443 Lyon Cedex 03
France

N. P. Topham* A. W. R. Crawford†
Department Of Computer Science
The University Of Edinburgh
Edinburgh EH9 3JZ
United Kingdom

Abstract

The development of accurate trace-driven simulation models has become a key activity in the design of new high-performance computer systems. Trace-driven simulation is fast, enabling analysis of the behaviour of large application and benchmark programs on a new computer system.

We describe a trace-driven simulation engine for a decoupled processor architecture. We report on two ways of generating efficient execution traces of real programs for this engine: profiler-assisted and compiler-assisted, and also on the use of synthetic traces. Results are reported for execution on the simulation engine of traces from Linpack and one of the Perfect Club benchmarks, as well as synthesized traces.

1. Introduction

Any new complex architecture must be modelled and performance-tested before implementation. Such testing enables flaws to be corrected before the design process is far advanced. Performance models used in general practice are either analytical or simulation based. See [5] for an extensive survey of performance models and evaluation methodology. Analytical models are elegant and fast, but it is often necessary to make some simplifying assumptions to obtain tractable solutions using this technique. Accuracy of an analytical model depends mainly on the set of assumptions taken by the model. While analytical models have limitations on the number of features that can be modelled, simulation models can be built to an almost unlimited level of accuracy.

*Research funded by ESPRIT grant P6253.
†Supported by a UK SERC studentship.

Instruction level simulation studies of a decoupled architecture are reported in [7]. While being accurate, instruction level simulations are tied to a particular machine and instruction set. They are also time-consuming and thus are not particularly practical for large programs. Trace-driven simulations, on the other hand, lack the accuracy of instruction level simulations, but are less closely tied to a specific architecture and at least an order of magnitude faster than instruction level simulations. See [6] for an account of efficient tracing techniques.

In this paper, we report the development of a simulation engine, driven by execution traces for a decoupled processor [9]. We present ways of generating execution traces for the simulation engine and report some results obtained by executing both real and synthetic execution traces.

The paper is organised as follows: section 2 is a description of the architecture. Section 3 describes the simulation methodology. It defines the execution traces that drive the simulation engine, presents some of the forms of synthetic execution traces supported and describes some of key performance metrics examined during simulations. Section 4 describes some ways of generating execution traces from real programs and presents two trace generation processes: profiler-assisted and compiler-assisted. Section 5 reports some simulation results. The last section concludes with a summary.

2. The architecture

A decoupled processor [9] consists of an Access Unit (AU) and an Execution Unit (XU). See Figure 1(a) for a functional block diagram. The AU and XU operate on concurrent instruction streams from a program. The AU instruction stream is for generating addresses for the memory operations and for performing other

Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences, 1994

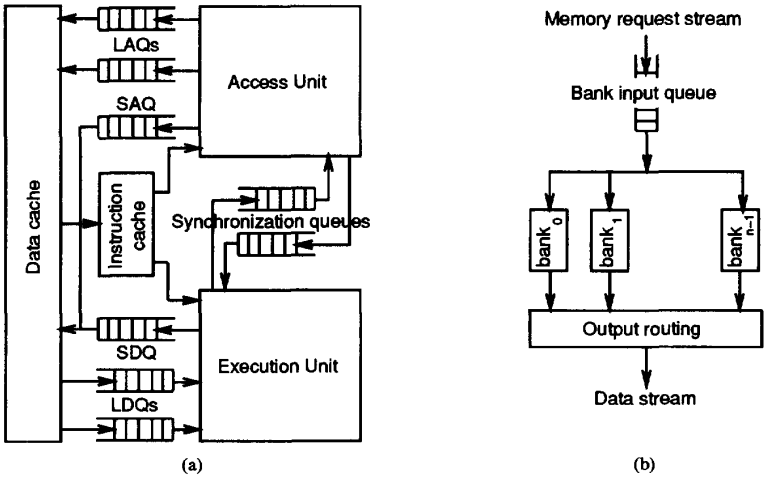

Figure 1: Functional block diagrams of (a) the processor and (b) a memory segment

integer operations; the XU instruction stream is for floating-point operations. The XU is capable of carrying out two pipelined floating-point operations each cycle. In general, the AU will be ahead of the XU in the program flow. When required, the AU and XU communicate and synchronize with each other through dedicated queues called synchronization queues. The processor interacts with the memory system via another set of queues.

Each cycle, the AU can initiate two load requests and prepare a store, while the XU can consume two loads and issue a store. If the AU is far enough ahead of the XU in the program flow, load requests are serviced by the memory system before the XU requires them. At this stage, the units are said to be fully decoupled, therefore the XU need not stall waiting for a memory load. Communication and synchronization between the AU and XU may result in a loss of decoupling. When decoupling is lost, the XU may see a full memory latency for load requests. For a detailed discussion of loss of decoupling in scientific programs, see [2].

The memory system consists of four segments of interleaved memory banks and crossbar switches that connect the segments to the processor. The memory banks operate in parallel; thus memory requests, unless destined to the same bank, may not be serviced in the order of processor initiation. Load data queues (LDQs) within the processor re-order those memory requests that are returned out of order.

In addition to the LDQs, the processor has a store address queue (SAQ), which buffers partial stores to memory. A partial store is essentially the store address prepared by the AU. This address waits in the SAQ to be paired with a datum generated by the XU.

The AU passes the load and store addresses to appropriate address queues. There are two load address queues to buffer the two concurrent memory load requests. When passing a load address to a load address queue (LAQ), a position is reserved in the destination LDQ to store the corresponding data item. The ID of the destination LDQ and the position within that LDQ are tagged to each load address.

Store addresses are passed to the SAQ. To avoid potential read-after-write (RAW) hazards, each load address is checked against all the addresses in the SAQ before being placed in an LAQ. If a match is detected, then the load tags (the LDQ ID and position) are tagged to the matching SAQ position. When there is no room to add a load tag, the AU is stalled until the match is cleared.

The XU receives load data from the LDQs and passes store data to the store data queue (SDQ). A store datum from the SDQ is paired with an address from the SAQ and sent to memory. If the store has been tagged, a copy of the datum is short-circuited to the LDQs (as dictated by the load tags specified in the tagged store). Tagging of loads onto matching SAQ positions improves the system performance in two ways: (1) dependent loads no longer cause the AU to block, and (2) RAW loads are immediately available to the XU.

The memory banks are organised as segments, with each segment consisting of several concurrently access-

ible banks and a bank input queue. See Figure 1(b) for a functional block diagram. The segment has a single input port and a single output port and can service at most one memory request per cycle. The bank input queue uses the *lookahead* control scheme reported in [3]. It functions as follows: any memory request addressed to the segment is placed in the input queue. At every cycle the first item in the queue which is intended for a bank that is free is removed and sent to its bank. The removal of an item i from the input queue causes the items that follow i to move forward to fill the empty slot created by i.

The memory segments are connected to the processor by crossbar switches. They ensure that reads (writes) issued in cycle c and destined to bank b are not overtaken by writes (reads/writes) issued in cycles following c and destined to the same bank b.

3. Simulation methodology

The simulation engine is driven by execution traces and mimics the architecture described in Section 2. An execution trace is a sequence of architectural events extractable from programs executing on the architecture. These events are divided into three classes: computational, synchronization and memory operations.

Computational operations are carried out in the AU and the XU. Within the simulation engine these operations are denoted AU_Op and XU_Op respectively. The exact nature of these operations is left unspecified although each carries with it the time it takes to execute.

Synchronization operations are carried out on both the AU and XU, and must appear as pairs. For instance, the AU may issue an AU_Wait_XU instruction that causes it to wait for a result from the XU. The XU in turn may issue an XU_Signal_AU that passes a result to the AU. The passing of the result takes place via a queue, so an empty queue will cause the AU to block until the queue becomes non-empty while a full queue will cause the XU to block until the queue becomes non-full. A synchronization between the AU and XU is thus achieved. See Table 1 for a summary of the synchronization operations supported by the simulation engine.

Memory operations are always decoupled. The AU generates addresses and the XU either generates or consumes data. Therefore, memory operations appear in pairs. The AU part belongs to the AU instruction stream while the XU part belongs to the XU instruction stream. Valid memory operations within the simulation engine are summarized in Table 2. Each of the XU memory operations may have an implicit XU_Op associated with it. This implicit XU_Op accounts for the pipelined execution within the XU.

Operation	Explanation
AU_Wait_XU	AU waits for result from XU
XU_Signal_AU	XU passes result to AU
XU_Wait_AU	XU waits for result from AU
AU_Signal_XU	AU passes result to XU

Table 1: Permissible synchronization operations

Operation	Explanation
AU_XU_Load XU_Load	XU load decoupled by AU
AU_XU_Store XU_Store	XU store decoupled by AU
AU_XU_LoadLoad XU_LoadLoad	XU loads decoupled by AU
AU_XU_LoadStore XU_LoadStore	XU load and store decoupled by AU
AU_XU_LoadLoadStore XU_LoadLoadStore	XU loads and store decoupled by AU

Table 2: Permissible memory operations

3.1. Internal execution traces

Execution traces are generated either internally or externally. Internal traces are synthetic and the simulation engine has the provision to synthesize three forms of traces: uniformly distributed random, strides and saxpy. These synthetic traces are tuned to bring to light certain behavioural patterns specific to the architecture.

A uniformly distributed random trace stresses the memory system evenly. Executing such traces therefore presents a best-case behaviour of the memory system. This behaviour is expected with real execution traces when a suitable address remapping strategy is employed to randomize the memory reference pattern [8].

Strides are used for examining the behaviour of the interleaved memory system in the presence and absence of address remapping. The bandwidth achievable by a sequentially interleaved memory, in the absence of address remapping, is given by:

$$\frac{n}{\gcd(n, s)}$$

where gcd stands for the greatest common divisor, n is the number of banks and s is the stride. Address remapping attempts to make the bandwidth stride-insensitive so that the full memory bandwidth will be achievable.

Saxpy is a kernel taken from Linpack. It produces three streams having unit strides: two load streams and a store stream. Thus saxpy examines the behaviour of the system when the memory is stressed to the maximum extent. The saxpy kernel consists of the following loop:

```
DO I = 1, N
   Y[I] = A * X[I] + Y[I]
ENDDO
```

The internal synthetic traces are useful for understanding the memory system behaviour under different synthetic loads. The loading factor and the synchronization frequency of the synthetic traces can be tuned to suit experiments. The loading factor refers to the rate at which the address references are issued. A 70% loading means the processor handles, on average, 14 loads and 7 stores per 10 cycles. The synchronization frequency refers to the rate at which the AU depends on the XU for address generation.

3.2. Performance metrics

The simulation engine produces an extensive statistics report at the end of each simulation. The statistics include average and maximum lengths of various queues within the processor and memory system, utilization of various resources, measurements of various bottlenecks (such as queue blockages) within the system and cumulative latencies at the output points of each architectural component. The prime performance metrics reported by the simulation engine are as follows:

- Perceived latency is the latency of a load as seen by the XU. It is the average number of cycles the XU has to wait for a load item to become available.
- Through latency is the full-trip latency of a load request measured between the points in time at which the AU issued the address and the XU consumed the data.
- Execution unit efficiency is the number of flops performed by the XU per cycle and is expressed as a percentage of peak flops achievable.
- Loss of decoupling (LOD) penalty is the average number of cycles lost due to an LOD event.

4. Execution traces from programs

Any execution trace generated elsewhere can be executed on the simulation engine as long as it adheres to the syntax of the engine trace format. Traces extracted from real programs are valuable for determining the system performance on specific workloads. This section reports on the generation of execution traces from programs. We describe two trace generation processes: profiler-assisted and compiler-assisted.

4.1. Profiler-assisted trace generation

Profiler-assisted trace generation is based on the observation that, in most scientific application programs, the majority of execution time is spent in relatively small sections of the code. These small sections of code are hand-annotated to produce execution traces. A profiler-assisted trace generation process uses the following steps.

Profiling. A standard profiler is used to identify the sections of the program that are heavily used. The profiling is generally done on a machine other than the target machine. Thus we make the important assumption that critical sections on the host machine remain critical on the target machine.

Annotation. Output routines that print out the trace information are inserted manually into the critical sections. Insertion of these output routines does not change the semantics of the program. The annotations inserted assume that all possible optimizations have been made.

Trace generation. The annotated program is run to produce its execution trace.

4.2. Compiler-assisted trace generation

While annotating sources by hand is practical for reasonably short routines, the process of hand annotating complete programs is both time-consuming and prone to errors. For this reason, we decided that an annotation tool should be developed that would be capable of annotating source code automatically while still producing traces of comparable accuracy to those produced by hand.

The annotation tool that was developed, and whose use is illustrated in Figure 2, is based on the Sigma toolkit [4]. This was originally intended for analysis, restructuring and parallelization of Fortran sources,

but proved adaptable to our needs. It provides a complete Fortran 90 parser (known as `cfp`) that produces output in the form of a database representing the parse tree for the input program which can be traversed and manipulated using the provided library functions.

The toolkit seemed ideal for our requirements, allowing rapid development of a source analysis and modification tool by distancing the application programmer from the source itself and presenting the program under analysis in an easily manipulated abstract form.

There were two principal requirements for the annotation tool:

- That the annotation should be semantically "invisible".
- That the traces generated were close to those that would be generated by good hand annotation or by tracing the output of a production level compiler.

Generating naïve traces based purely on a single parse tree traversal produces a valid execution trace. However, the trace corresponds to that generated by tracing the execution of the output of an extremely stupid non-optimizing compiler. Production of more accurate annotation requires that additional analysis of the source be performed, similar to that done by an optimizing compiler. This in turn requires that a compromise be made between the speed and accuracy of annotation. Unfortunately, producing traces that closely approximate those produced by an optimizing compiler would have required that we actually perform many of those optimizations, considerably increasing the size, complexity and runtime of the annotator. Since the combination of automatic annotation and trace-driven simulation was seen as a quick alternative to producing a complete compiler and a full instruction level simulator, having the annotator spend considerable amounts of time performing optimizations was not seen as desirable.

In the end, a limited set of optimizations were implemented. The most important of these is common sub-expression elimination. Other optimizations include constant folding, address arithmetic simplification and several optional optimizations such as support for guarded execution.

4.2.1. Annotator structure

The annotator makes four traversals of the parse tree. The first two of these restructure the parse tree into a more easily annotated yet semantically equivalent normal form. The third pass performs a simple form of common sub-expression elimination. This could have been merged with the fourth pass but was kept separate for portability reasons. On the fourth and final pass, the annotator builds up an intermediate representation of the behaviour of each basic block. The internal representation used is straightforward and general and consists of pairs or triples of lists of operation tuples (as described in Section 3), with each list corresponding to one of the processing units that makes up the decoupled processor architecture.

This pass of the annotator corresponds approximately to the code generator of a compiler and performs two functions. First, depending on the required trace information, the internal representation of the trace for a basic block may be compacted, simplified or otherwise manipulated. A reasonable analogy here would be peephole optimization of an intermediate representation to take advantage of features of a specific target architecture. Secondly, this pass manipulates the Sigma database to add new statements to the parse tree that will generate trace information when the modified source is regenerated and run.

This method of annotation has proved to be highly flexible and is easily parameterized, allowing one tool to produce trace-generating sources comparable in quality to those produced by hand annotation for architectures using arbitrary combinations of several architectural features.

5. Some simulation results

This section presents some typical results obtained from running the simulation engine. During the design phase, simulation experiments help to make both architectural and design decisions. The results we show in this section arise from the following questions:

- Is there any performance gain if one uses an address remapping scheme for the memory?
- What are the optimum sizes of SAQ, LDQ and the bank input queue?

5.1. Effect of address remapping

Sequential interleaving is a commonly used memory addressing mechanism. If a system has m memory segments with b banks per segment, the lowest $\log m$ bits of a memory address A determine the segment ID of A and the next lowest $\log b$ bits determine the bank ID of

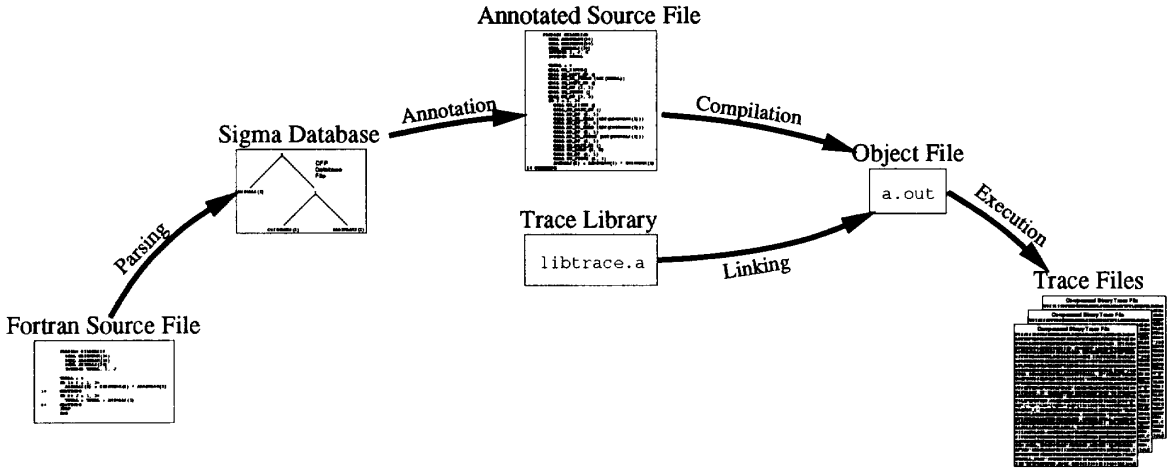

Figure 2: The automatic annotation process

A. As has been pointed out in Section 3.1, this mechanism performs poorly when the input address stream has certain strides, since with certain stride patterns some banks are hit more frequently than others, resulting in bank conflicts and a large memory latency.

An address remapping scheme is a transformation mechanism that makes an address stream practically stride-insensitive. For an example, see Rau's irreducible polynomial method [8]. We ran execution traces, produced using a profiler-assisted process, of Linpack and BDNA (from the Perfect Club suite [1]), with and without Rau's address remapping scheme. Table 3 illustrates relative execution times of Linpack and BDNA with and without address remapping. When there is no address remapping, a normal sequential interleaving is used.

Execution trace	Remapping	No remapping
LINPACK	1.00	0.99
BDNA	1.00	1.59

Table 3: Relative execution times with and without address remapping

Table 3 shows that the BDNA benchmark, with many strides that are not optimal for a sequential interleaving, experiences a significant improvement in performance when Rau's address remapping is used. The Linpack benchmark, with mainly unit strides that are optimal for sequential interleaving, suffers a slight loss in performance under the address remapping. This leads us to conclude that the address remapping makes "bad strides" perform well, while having little effect on "good strides". We believe that without extensive tracing of real applications it would be difficult to assess conclusively the benefits or otherwise of address remapping.

5.2. Queue size determination

Given the generally superior behaviour of Rau's address remapping, we may assume for this experiment that remapping is performed on all address streams. If this is so, the resulting address streams closely resemble a uniformly distributed random address stream. We therefore use a uniform execution trace to find the optimum sizes of SAQ, LDQ and the bank input queue. Figures 3 and 4 show the results.

We chose a typical high-performance processor cycle time of 5 ns and a range of typical DRAM bank cycle times (130–165 ns). Across this range of bank cycle times, a bank input queue size of 8–16 achieves close to 100% performance. We were able to show (see Figure 3) that bank input queue size is relatively insensitive to bank cycle time.

Figure 4(a) shows the effect that SAQ and LDQ sizes have on the degree of decoupling. The goal of decoupling is to achieve a memory system with an effective latency of zero cycles. Figure 4(b) shows the actual perceived latency of the system as a function of queue sizes and bank cycle time. The region of the graph where the perceived latency is zero corresponds to queue sizes which permit full decoupling. Figure 4(a) shows how the perceived latency translates into XU efficiency. The performance plateau,

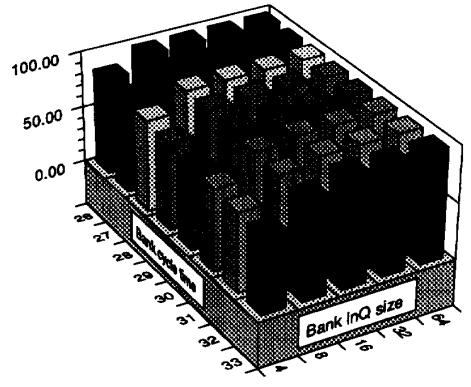

Figure 3: XU efficiency vs bank input queue size and cycle time

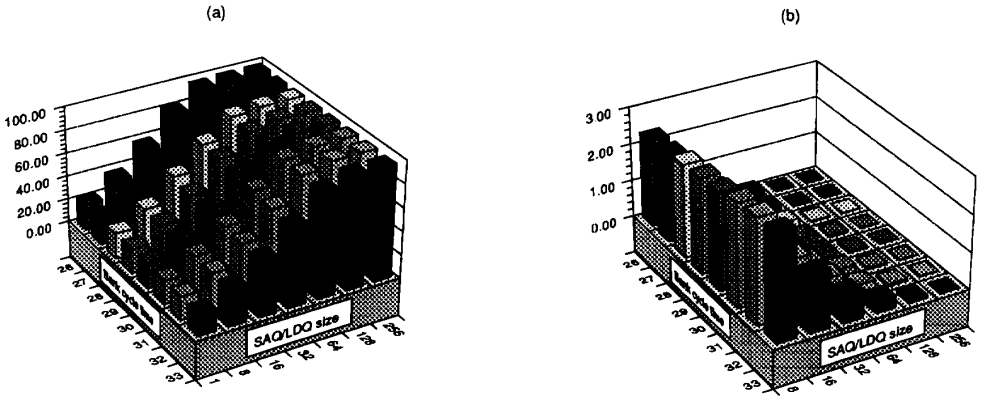

Figure 4: (a) XU efficiency and (b) perceived latency vs SAQ/LDQ size and bank cycle time

at around 100%, represents a trade-off space within which any combination of queue sizes and bank cycle times will yield maximum performance.

6. Conclusions

In this paper we have reported the development of a simulation engine, driven by execution traces, for a decoupled processor architecture. We have also presented two ways of generating execution traces for the simulation engine, and shown some results obtained by executing both real and synthetic execution traces.

A number of novel architectural techniques can be assessed quantitatively on real applications using profiler-assisted and compiler-assisted tracing techniques. We believe that the behaviour of complex nonlinear systems, such as decoupled architectures, requires extensive simulations on target application classes to expose the performance space. Even if it is sometimes difficult to obtain accurate absolute values for performance (due mainly to model inaccuracies) the insight into the behaviour of the system gained by such simulations is normally extremely valuable.

References

[1] M. Berry *et al.*, "The Perfect Club benchmarks: Effective performance evaluation of supercomputers," *The International Journal of Supercomputer Applications*, vol. 3, pp. 5–40, 1989.

[2] P. L. Bird, A. Rawsthorne, and N. P. Topham, "The effectiveness of decoupling," in *Proceedings of the 7th ACM International Conference on Supercomputing*, (Tokyo, Japan), 1993.

[3] P. L. Bird, N. P. Topham, and S. Manoharan, "A comparison of two memory models for high performance computers," in *Proceedings of the 2nd Joint International Conference on Vector and Parallel Processing: CONPAR 92 - VAPP V (LNCS 634)*, (Lyon, France), pp. 399–404, Springer-Verlag, September 1992.

[4] D. Gannon *et al.*, "Sigma II: A toolkit for building parallelizing compilers and performance analysis systems," in *Programming Environments for Parallel Computers* (N. P. Topham, R. Ibbett, and T. Bemmerl, eds.), North Holland, 1992.

[5] P. Heidelberger and S. S. Lavenberg, "Computer performance evaluation methodology," *IEEE Transactions on Computers*, vol. 33, no. 12, pp. 1195–1220, December 1984.

[6] J. R. Larus, "Efficient program tracing," *Computers*, vol. 26, no. 5, pp. 52–61, May 1993.

[7] W. Mangione-Smith, S. G. Abraham, and E. S. Davidson, "The effects of memory latency and fine-grain parallelism on Astronautics ZS-1 performance," in *Proceedings of the 23rd Annual Hawaii International Conference on System Sciences*, pp. 288–296, 1990.

[8] B. R. Rau, "Pseudo-randomly interleaved memory," in *The 18th International Symposium on Computer Architecture*, pp. 74–83, May 1991.

[9] J. Smith, "Decoupled access/execute architectures," *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 289–308, November 1984.