Performance of the Decoupled ACRI-1 Architecture: the Perfect Club

Nigel Topham[1,†] and Kenneth McDougall[2,3]

[1] Department of Computer Science,
University of Edinburgh,
The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ,
Scotland, UK.
[2] The Advanced Computer Research Institute,
1, Boulevard Marius Vivier-Merle, 69443 Lyon,
France.
[3] Department of Mathematical and Computational Sciences,
University of St. Andrews,
The North Haugh, St. Andrews, Fife KY16 9SS,
Scotland, UK.

Abstract. This paper examines the performance potential of decoupled computer architectures on real-world codes, and includes the first performance bounds calculations to be published for the highly-decoupled ACRI-1 computer architecture. It also constitutes the first published work to report on the effectiveness of a decoupling Fortran90 compiler. Decoupling is an architectural optimisation which offers very high sustained performance through large-scale latency hiding. This paper investigates the applicability of access and control decoupling to real-world codes. We illustrate this with compiler-generated decoupling optimisations for the Perfect Club benchmark suite on the Advanced Computer Research Institute's ACRI-1 system, utilising the frequency of loss of decoupling (LOD) events as a measure of the effectiveness of decoupling to each code. We derive bounds for the performance of these codes and show that, whilst some exhibit performance roughly equivalent to that on vector computers, others exhibit considerably higher performance potential in a decoupled system.

1 Introduction

A computer's instruction set may be conceptually partitioned into subsets which may then be assigned to specialised hardware for execution. That these subsets are defined to maximise concurrence of hardware utilisation has meant that many different partitionings have been tried. Early high-performance computers such as the Cray-1 overlapped memory access and arithmetic operations to an extent, but the trend towards more explicit separation of the instruction sets has led to

---

[†] This research was supported by ESPRIT project P6253 and by UK EPSRC research grant number GR/K19723.

architectures such as PIPE [1], the Astronautics ZS-1 [2], and more recently the ACRI-1, which is the subject of this paper.

The ZS-1, PIPE and Wulf's WM architecture [3] are termed access decoupled. In such architectures, control transfers often require the synchronisation of the processing units. This occurs, for example, when a branch is dependent upon a comparison computed by the X-processor. A further architectural optimisation, termed control decoupling, is introduced in the Advanced Computer Research Institute's ACRI-1 architecture [4]. In a control decoupled architecture there are three independent units, responsible respectively for control, memory access and execution. The additional benefit of control decoupling is that the majority of control transfer decisions can be pre-computed, thus opening up many opportunities for preparing the access and execute pipelines for the operations which follow. Access and control decoupling are fully described elsewhere [4], however, a brief introduction to each is provided in sections 1.1 and 1.2.

Given the high degree of decoupling, the degree to which real-world applications can exploit decoupling is of prime importance. In common with many recent architectural innovations, the performance of the architecture is greatly influenced by the capabilities of the compiler. In that sense our analysis in this paper should be seen as a combined evaluation of the architecture and the pre-production version of the compiler for this architecture (scf90). Our analysis consists of compiler-driven measurements and profile-driven modelling of the Perfect Club suite of scientific programs [5].

As fully functional hardware for the ACRI-1 system was not available at the time of writing, our performance results are derived from frequency-domain profiling of the test programs on another system, combined with event-domain profiling from the pre-production compiler. This provides us with the capability to determine the position of events of interest within the source code, and to match this with run-time frequency information. Frequency information is architecture independent, and was obtained from program measurements on a Sun SPARC system. The event frequency measurements are presented in section 4. The dynamic event counts produced by this method then drive a simple linear model of execution time to produce bounds on execution times which are tight. These execution time bounds are presented in section 4.2.

1.1 Access Decoupling

In a decoupled access/execute (DAE) architecture, the processes of accessing memory and performing computations on values fetched from memory can be thought of as micro-threads implemented on separate asynchronous units. In ACRI-1 terminology, the Address Unit (AU) computes addresses and issues memory requests. The Data Unit (DU) receives data from memory, computes new values, and sends new data back to memory. The AU and DU execute a program containing the instructions that are specific to each unit. The only constraint on the AU and DU programs is that the order in which operand addresses are issued by the AU must match precisely the order in which operands are used by the DU.

The AU tags each memory fetch request with the location in a load queue within the DU where the incoming data will be buffered. This tag permits the physical memory system to process requests in any order; the DU load queue re-orders them as they arrive, ensuring that the AU-DU ordering constraint is always satisfied. In the ACRI-1 architecture there are two independent load paths to memory, and two independent load queues in the DU.

The AU is optimised to implement the most common operations on induction variables. Thus, it has a simple integer instruction set and instruction modes which permit operations of the form $mem = r_i = r_i + C$. In a single instruction an induction variable r_i can be incremented by some constant value (or the contents of a register) and the result can be stored back to the induction variable as well as being sent to memory as a load or store address. In the ACRI-1 architecture two load addresses and one store address can be computed and sent to memory in each cycle.

The memory of the ACRI-1 system is highly interleaved to provide the required bandwidth of two loads and one store per cycle (per processor). In addition, the bank selection logic implements a pseudo-random hashing function [6, 7] to ensure an even spread of addresses to banks; even in the presence of strides that would cause serious performance problems in traditional vector machines.

1.2 Control Decoupling

In the ACRI-1 architecture, control transfers are partitioned into two groups; those which implement leaf-level loop control (leaf-level loops are those without internal cycles), and those which implement all other control transfers. The AU and DU have the capability to implement simple looping constructs, and this permits the compiler to target leaf-level loop control directly on to the AU and DU. All remaining control transfers are executed by a third unit, the Control Unit (CU). Effectively the CU controls the sequencing of the program through its flow graph, dispatching leaf-level loops intact to the AU and the DU.

Control decoupled architectures share some similarities with vector processors, in which a scalar unit dispatches vector instructions to a vector load pipeline and vector arithmetic pipelines, however, the differences are significant. Firstly, the body of the leaf loop on the AU and the DU is derived directly from the source code without *any* need to vectorize. Secondly, the compiler's partitioning of code between units is driven by data dependencies and not by what instructions can or cannot be vectorized. Thirdly, there is a high degree of asynchrony between the three units, and this permits the CU, for example, to enqueue loop dispatch blocks for the AU and DU well in advance of their being executed. The CU is, in many ways, a natural (prefix) extension of the virtual pipeline connecting the AU to the DU through memory.

2 Overview of the ACRI-1 implementation

The ACRI-1 architecture is a high performance implementation of a control and access decoupled architecture. Each DU has two independent floating point units designed for a target clock period of 6ns. Each processor therefore has a floating point throughput design peak of 333 MFLOPS (64-bit precision). A node may contain up to 6 processors, for a peak floating point throughput of 2 GFLOPS. This paper addresses the performance of a single processor in this multiprocessor architecture; further information on the scalability implications of decoupling can be found in [8].

The ACRI-1 memory system comprises up to eight boards, each containing up to eight segments. Each segment may contain up to 16 independently addressable banks of DRAM. The memory boards are connected to the processors and I/O subsystems via a two-stage parallel network. Both the network and the memory boards contain request and response queues, and thus the round-trip latency for any particular request will depend to a certain degree on the memory loading. Register-transfer simulations of the network and memory subsystems have shown that latencies will be in the range 100 to 200 processor cycles, and will vary dynamically during program execution. In the execution time models presented in section 4.2, we use a nominal value of 150 cycles for the mean cost of uncoupling the DU from the CU and the AU. This uncoupling time is dominated by the round-trip delay of the memory system.

The ACRI-1 processor contains a cache which is used primarily for communication between the units, and as a level-2 instruction cache. The vast majority of memory operands are obtained directly from memory. Further information on the behaviour of caches in decoupled systems can be found in [9].

3 Losses of Decoupling (LODs)

In an access and control decoupled architecture, the CU "runs ahead" of the AU and DU, and the AU "runs ahead" of the DU. It is conceivable for the CU and the DU to be separated space and time by many thousands of program statements. When the system is fully decoupled, the AU will typically be ahead of the DU by a time equal to the largest memory latency experienced by any load operation since the last recoupling of the AU and DU. The CU, AU and DU therefore constitute an elastic pipeline, with the dispatch queue linking the CU to the AU and DU, and the memory system linking the AU to the DU.

In figure 1 the solid arrows show the typical direction of flow of information during decoupled execution. The broken arrows represent inter-unit dependencies which cause decoupling to temporarily break down, requiring the CU or AU to wait for operations to complete on a unit which is normally *downstream* in the system pipeline. These dependencies can be carried either through registers or memory locations. For example, a value in a DU register which the compiler knows to be needed by the CU will cause a register-based flow dependence from the DU to the CU. This will be satisfied by sending the value from the DU to the CU via a transfer queue. The CU must wait for this value, defining a synchronization point between DU and CU. This effectively flushes the memory pipeline and decoupling is lost. We term such a synchronization point a loss of decoupling (LOD).

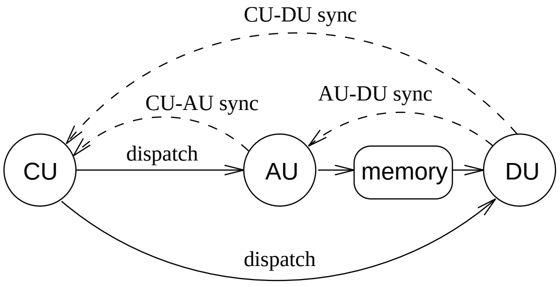


Fig. 1. Pipeline flow in a control and access decoupled architecture

When the DU defines a location in memory and there is a reaching use of that location on the CU, then a read-write hazard exists between the CU and the DU through that location. The compiler detects all such hazards and inserts explicit synchronization operators at appropriate points in the code to force the CU to wait for the hazard to be resolved. This results in the DU and the CU becoming synchronized, so this is also a form of LOD. Both of the above types of LOD are termed algorithmic LODs, as their presence is a direct result of the structure of the program. In the tables which follow, such LODs are labelled A-LODs.

The calling standard of the ACRI-1 defines a common shared stack for the CU, the AU and the DU. This has implications for data synchronization across function call and return boundaries. When a function is called, and the CU and DU are decoupled, the CU will reach the stack frame manipulation code before the DU finishes using the current stack frame. To prevent stack frame corruption, the CU must wait for the DU to reach the call point before the stack frame is modified. Again, this constitutes an LOD. In the tables which follow, we term this a "call LOD" (or C-LOD). A similar situation may occur if the units are uncoupled at a function return point. Again, the CU would like to relinquish a stack frame, but the DU may not yet have finished using some of the local variables declared in that frame. The CU must therefore wait for the DU to reach the return point, and then perform the return sequence. This is a "return LOD" (or R-LOD).

Calls to external routines, such as I/O and timer calls, must also be synchronized. However, by appropriate engineering it is possible to avoid LODs across the call boundaries with intrinsic functions (information on the use and modification of parameters is more clearly specified, and intrinsics do not modify global variables). In this analysis, LODs due to calls to external routines are termed "eXternal LODs" (or X-LODs). In this analysis intrinsic functions do not lead to LODs.

To summarize, when a program is executing in a decoupled mode the perceived memory latency is zero; effectively the entire physical memory has an access time equivalent to the processor's register file, and latency is completely hidden. However, when an LOD occurs, the penalty is significant. Our model assumes a nominal penalty of 150 cycles, and this means that the frequency of LODs is of paramount importance when determining the expected application performance. We now present measurements of the LOD frequencies in the Perfect Club, and in section 4.2 we use these frequencies to predict bounds on the execution time of the Perfect Club programs on the ACRI-1 architecture.

4 Perfect Club LODs

The Perfect Club consists of 13 Fortran programs, ranging in size from 509 to 18,545 lines of source code. Each program was compiled using `scf90` and the resulting dump of the intermediate representation was analyzed to determine the type and source code position of each LOD. This information was combined with run-time statistics obtained by profiling each program with `tcov` on a Sun SPARC architecture to obtain an accurate prediction of execution frequency for each individual LOD in the compiled program. Table 1 below details the number of each type of LOD that will be executed for each program in the Perfect Club.

Program	A-LODs	C-LODs	XC-LODs	R-LODs	Total LODs
ADM	117,353	2,773,739	80,003	2,914,874	5,885,969
SPICE	6,452,950	187,681	277,420	1,042,677	7,960,728
QCD	833,543	6,340,047	201	6,648,301	13,822,092
MDG	6,868,049	19,535,152	176	19,535,153	45,938,530
TRACK	593,825	658,826	26,679	652,929	1,932,259
BDNA	67,158	134	814	166	68,272
OCEAN	320,808	154,920	1,366	152,940	630,034
DYFESM	145,362	375,897	60,760	382,131	964,150
MG3D	1,974,962	3,640,446	778,518	2,083,671	8,477,597
ARC2D	7,158	2,918	26,303	2,819	39,198
FLO52Q	24,541	21,053	21,634	12,620	79,848
TRFD	256,632	14	42	15	256,703
SPEC77	4,814,795	435,328	810	447,578	5,698,511

Table 1. Loss of Decoupling (LOD) frequencies for the four categories of LOD in the Perfect Club.

4.1 Optimizing LOD frequency

Of particular interest is the dominance of call and return LODs in some of the programs. Such LODs can be optimized away relatively easily by subroutine inlining. Table 2 shows how the overall LOD counts change in six programs which were deemed candidates for inlining. The inlining was performed by KAP, and the resulting code was re-compiled and re-profiled to measure the alteration in LOD frequencies. Most programs experienced a significant reduction in overall LOD count, even though some experienced increased A-LOD counts.

Program	A-LODs	C-LODs	XC-LODs	R-LODs	Total LODs	% LOD Reduction
MDG	6,868,931	3,703	171	3,704	6,876,509	85
ADM	541,681	326,366	80,730	325,647	1,274,424	78
QCD	2,477,507	2,582,316	195	3,040,984	8,101,002	41
FLO52Q	47,980	0	9	1	47,990	40
SPICE	5,752,231	47,236	149,451	493,395	6,442,313	19
SPEC77	4,831,837	350,224	28,792	362,719	5,573,572	2

Table 2. Reductions in LOD frequencies due to subroutine inlining by KAP.

4.2 Perfect Club Performance

The execution time T_p for program p on the ACRI-1 system is modelled relatively accurately by equation 1,

$$T_p = \phi(C_{du} + nd) \tag{1}$$

where C_{du} is the number of execution cycles of the DU[1], n is the number of LODs executed, d is the nominal penalty induced by each LOD, and ϕ is the clock period of the machine. Without actually executing a program is it hard to predict values for C_{du}, but it is possible to define a lower bound $\overline{C}_{du}$ as $f_p/2$, where f_p is the number of floating point operations executed by program p. This must be an absolute minimum execution time for any processor with two floating point pipelines. This permits us to define execution time as follows:

$$T_p \geq \phi\left(\frac{f_p}{2} + nd\right) \tag{2}$$

In fact, in the ACRI-1 architecture only pipeline startup and shutdown delays on the DU will introduce any discrepancy between $\overline{C}_{du}$ and C_{du}. As the DU contains hardware support for modulo-scheduled software pipelining, we expect startup and shutdown costs to be relatively low, though it is still too early to present definite figures. Figure 2 compares the lower bound on ACRI-1 cycles to completion, for each program in the Perfect Club, with the measured cycles to completion on the Cray Y-MP C90 (from [10]).

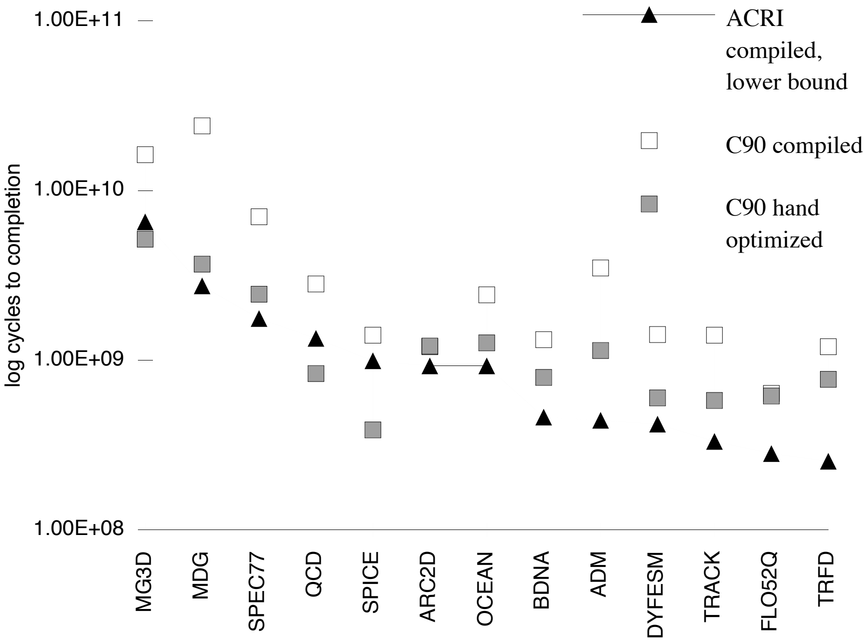
Figure 3 illustrates how the lower bound on cycles to completion translates into an upper bound on MFLOPS execution rates for each of the Perfect Club programs, given a target clock period of 6ns.

5 Conclusions

In this paper we have presented the most recently available performance data the ACRI-1 supercomputer which is currently under development. Our main aim in so doing has been to highlight the execution efficiency that derives from

---

[1] We assume that the DU code defines the critical path through the program. In most scientific applications this is a safe assumption.


1.00E+11
1.00E+10
1.00E+09
1.00E+08
log cycles to completion
ACRI compiled, lower bound
C90 compiled
C90 hand optimized
MG3D
MDG
SPEC77
QCD
SPICE
ARC2D
OCEAN
BDNA
ADM
DYFESM
TRACK
FLO52Q
TRFD