Presented at the Seventh ACM International Conference on Supercomputing,
Tokyo Japan, July 19-23 1993.

The Effectiveness of Decoupling

Peter L. Bird
ACRI,
Lyon
France

Alasdair Rawsthorne,
University of Manchester,
United Kingdom

Nigel P. Topham,
University of Edinburgh
United Kingdom

Abstract

This paper examines the effectiveness of decoupling as an optimization technique for high-performance computer architectures. Decoupled access execute architectures are described, and the concept of *control decoupling* is introduced and justified. A description of a highly-decoupled architecture is given, and a metric for the effectiveness of decoupling on particular programs, the Loss of Decoupling frequency is introduced. Finally, a number of real benchmark programs are examined and the applicability of decoupling them is analyzed.

1 Introduction

A number of papers have discussed the architectural optimization *decoupling* over the last decade (see [2], [6], [7] and [8]). This paper introduces control decoupling, a further technique for increasing performance, and attempts to identify the class of programs over which decoupling is an effective technique to achieve high performance.

The instruction set of modern computers (see, for example, reference [5]) is partitioned into three classes of instruction: control, memory accessing and data operations. The decoupled architectures that have been described to date, for example ZS-1[2], PIPE[6] and WM[8], decouple between the latter two classes, using "Decoupled Access/Execute," in which the addresses for memory references are generated in advance of the execution of data-related instructions. This means that memory read operations can be initiated many cycles before the read data is required for execution, and the latency of main memory read operations (or cache operation) can be hidden. Decoupled Access/Execute is described in detail below, in section 2.

Less conventional is the use of *control decoupling*. This architecture feature is introduced in order to maximize the use of main memory bandwidth in an implementation, by exploring the control-flow graph of a program ahead of the time at which computation is required: this enables requests for packets of computation to be queued ahead of the time at which they are required, so that when one computation has finished, another is ready to take its place on the relevant unit. Control decoupling is described in detail below, in section 3.

In order to determine how valuable these optimizations are, in section 5 we introduce a conceptual framework for program events that cause decoupling to break down: these Loss of Decoupling events are of central importance to understanding the performance of decoupled architectures. In section 6 we discuss briefly the performance impact of Loss of Decoupling events. In section 7, we conduct an examination of a range of popular benchmark programs in order to understand the prevalence of these events.

2 Access Decoupling

In the architecture described here, Access Decoupling is implemented by partitioning user instructions into two classes, memory accessing and user arithmetic.

The memory accessing instructions are run on a special-purpose processor, the Address Processor (AP), whose function is optimized for the purposes of producing regular patterns of addresses, such as found in numeric programs. The most widely used instruction in the AP is an addition operation, which adds register plus register or register plus immediate, writing the result to another register and initiating either a memory read or a memory write operation. Apart from simple additions, the AP has no other data-handling capability: it has no general multiplication, division or logical operations.

The user arithmetic instructions are also run on a special-purpose processor, the Data Processor (DP). This has a full set of integer and floating-point arithmetic and logical operations, but no memory addressing instructions at all.

The AP and DP are connected by two types of queues: the Load Data Queue (LDQ) and the Store Address Queue (SAQ). Entries in these queues are made when the AP generates memory addresses, but the two types of queues are used substantially differently. In the architecture described in this paper there are two independent LDQs connected to the two memory read ports, and one SAQ to drive the single memory write port. The relationships of these processors and their queues can be seen in Figure 1 below.

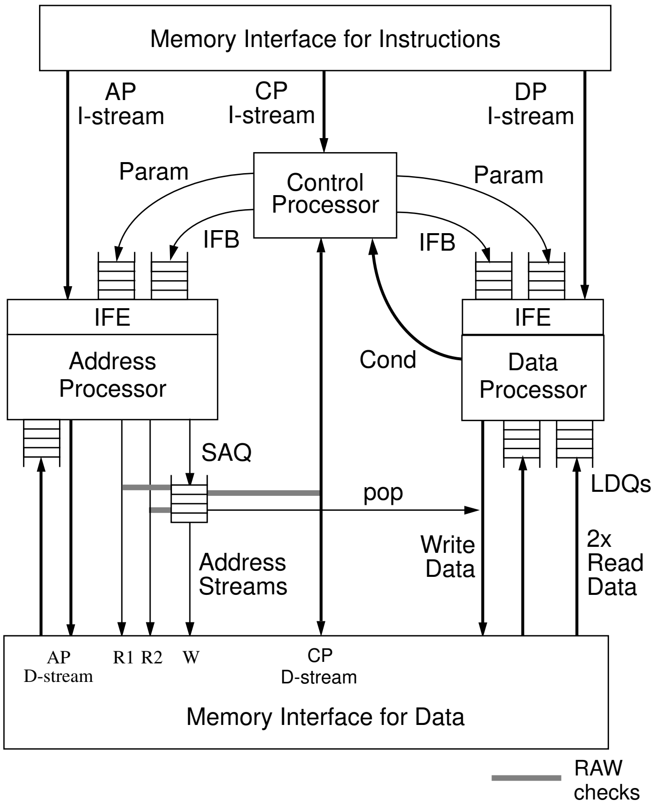

Figure 1. Processor Block Diagram

When the AP generates a memory request for a Load operation, the next free element in the relevant Load Data Queue is allocated and marked "pending". A request for a memory read at that address is sent to the memory interface. When the read data is returned from memory, it is placed into the queue element reserved at the time of the request, and the element is marked as "valid".

The Load Data Queues are available as source operands to instructions running in the DP, and an instruction that reads a queue suspends until the top entry in the queue is marked "valid", and then pops this element. (In fact, the queues are mapped to two general-purpose registers.) In this manner, every read memory access is carried out by two instructions, one on the address processor to initiate the request, and one on the data processor to access the data.

Store operations are handled differently. When an address for a store operation is generated by the AP, the address itself is written to the next free element of the Store Address Queue, and marked "valid". Store operations are initiated in the DP, where every arithmetic instruction contains a "store" bit. When this bit is set, the result of the instruction is sent as data to main memory, in addition to being written to a general-purpose register. When an instruction with its "store" bit set completes, the oldest entry in the Store Address queue is popped, and used as the address for a memory write operation with the data generated by the instruction.

In this way, the AP may proceed through a program, keeping ahead of the place where the DP is executing. This has the beneficial effect of initiating memory read operations early, reducing the impact of memory latency on execution time. In a fully decoupled program, once the decoupling between processors has been established, the execution time is insensitive to latency, provided that the main memory offers sufficient bandwidth to support the request rate generated by the AP.

The purpose of generating store addresses earlier than performance arguments would require is to ensure correct functionality: the instruction-set specification of the architecture defines that memory operations have their semantics defined by the order in which read and write operations are initiated by the AP. If the address of a write operation, for example, is used as the address of a subsequent read operation before the data for the write has been generated in the DP, a comparator detects that the read cannot proceed, and the AP is stalled until the condition can be resolved.

3 Control Decoupling

Control Decoupling is a further optimization that permits a Control Processor (CP) to execute yet further in advance of the AP. The first step is to give the Address and Data processors the capability of running program inner loops, that is of running a body of instructions a number of times, determined either by a loop-count, or terminated in a data-dependent manner. Apart from simple loops, the Address and Data Processors have no other control capability: they have no conditional jumps or subroutine call instructions.

All major control functionality, that is non-inner loop control, subroutine call and return, and dispatching inner loops to the Address and Data Processors, is concentrated in the CP. Since user computation is carried out in the DP, the CP does not need floating-point capabilities, but it does provide a full set of logical operations, integer addition and subtraction, integer multiplication (for array index calculations inner loops have their index arithmetic strength-reduced), integer division (for loop normalization), and a full set of comparison and conditional and unconditional branch operations that would be familiar to the programmer of any conventional RISC instruction set. Memory accessing (both load and store operations) are provided conventionally in the CP instruction set.

In order to reduce the interaction between Control and Data Processors, the DP supports an elaborate conditional-execution scheme, with a full set of comparison operations on integers and floating point operands, conditional execution of any of its instruction set, and a comprehensive set of condition combination instructions, which permit the compilation of nested `if..then..else` statements into guarded execution.

The CP invokes operations of the Address and DP using a special instruction, which dispatches a unit of work called an Instruction Fetch Block (IFB) to one of them. An IFB contains a pointer to the first instruction, a length field, specifying the number of instructions to issue, and a loop count, identifying how many times to issue these instructions. Instruction Fetch Blocks are enqueued by the CP for both the Address and Data Processors: when a processor has finished issuing instructions from one IFB, it can proceed to issuing instructions from the next block, if there is one in the queue, without delay. A set of parameter queues is provided to allow the CP to pass data items to the Address and Data Processors.

In this manner, the normal operation of the system is that the CP is executing instructions from the later part of a program. At the same time, the AP is executing instructions from an earlier part of the program, and the DP is executing from a still earlier part. In this way, all three processors are fully decoupled.

The benefit of Control Decoupling is that while one inner loop is running on the AP, preparatory work for subsequent loops, such as loop count calculation and array subscript arithmetic, may be carried out in parallel, ensuring that no time is lost in the AP between inner loop bodies.

A description of events during program execution that cause this decoupling to break down is found below, in section 5.

3.1 CP Memory Ordering

All addresses produced by the CP are compared against all pending DP write operations, whose addresses are held in the Store Address Queue, and the CP is stalled if any conflict arises. Nevertheless, a logical inconsistency may arise if the CP tries to read data that will be written by the DP by an earlier part of the program, if the address has not yet been generated because the AP has not yet reached this part of the program.

In this system, it is the function of the compiler to eliminate such inconsistencies by preventing the CP from decoupling from the AP when an access of this type is possible.

4. Decoupled Execution: an Example

The diagram in Figure 1 shows an overall picture of the control- and address- decoupled machine. The way in which the two forms of decoupling occur can be explained by reference to an example. Consider the code fragment below :

```
      DO 20 I = 1, M
        DO 10 J = 1, N
          A(I, J) = B(J) * S + C(I,J)
10      CONTINUE
20    CONTINUE
```

The inner loop, in J, is executed autonomously in the Address and Data Processors: in each iteration, the AP initiates a memory read for `B(J)` and `C(I,J)` and a queues the write address for `A(I,J)`, and the DP multiplies one Load Data Queue element by the register holding `S`, adds another Load Data Queue element and stores the result. The AP relies on the availability of stride values for the `A(I,J)` and `C(I,J)` accesses to avoid the need to do full multiplications within the loop.

The CP implements the loop in I, and prepares the Instruction Fetch Blocks and stride values for the AP and the DP. These are passed to these Processors via the queues, enabling the CP to iterate around the loop in I without re-synchronizing with the Address or DP on each iteration.

5 When does Decoupling Break Down?

Figure 2 gives a framework for discussing the influence of program events on both control and access decoupling. The broad arrows show how the decoupling optimization is successful when the CP is transferring information to the AP, and when the AP is transferring information to the DP. The numbered arrows, in the reverse direction, represent inter-processor dependencies that can interfere with decoupling by requiring that the CP or AP wait for a later unit

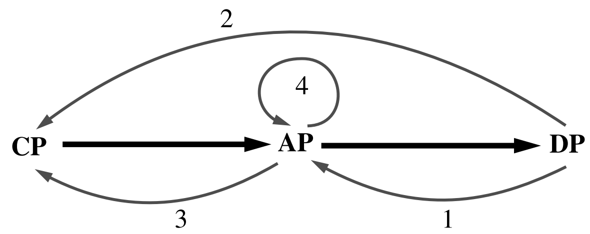

Figure 2. Events that May Destroy Decoupling

to "catch up". We call each of these events a "Loss of Decoupling"; each numbered arc in Figure 2 corresponds to exactly one type of LOD event.

5.1 Computed Index Operations

Arc 1 in Figure 2 represents the case of the AP needing to wait for the DP before initiating further operand fetches: this case arises when a value computed in the DP must be conveyed back to the AP to take part in address formation. A program fragment causing such an event is illustrated below:

```
      DO 10 I = 1, N
        X[I] = ...
        IX = ...X[I]
        Y[IX] = ...
10    CONTINUE
```

This case is rare across the full range of numeric codes: it arises in "Particle-In-Cell" codes, where a real particle position is calculated and then quantized into a polygonal grid, and in Monte Carlo codes, in which a discrete item is selected using a real-valued random number generator.

5.2 Conditional Control Flow Operations

Arc 2 in Figure 2 shows the CP needing to wait for a condition to be evaluated in the DP before it can issue further instructions: at first glance, this event might seem to occur every time a conditional statement is executed. However, since the DP can implement conditionals internally, through the medium of conditional execution, this event only occurs when a condition causes a major change of sequence in the Control Flow behaviour of the program, for example, when a program conditionally calls a subroutine, or conditionally executes an entire loop body. A program fragment illustrating this type of event is shown below:

```
      DO 10 I=1, 42
        X[I] = ...
10    CONTINUE
      IF (X[42] .NE. 0) CALL RENORM
```

Other types of program fragments which cause this type of LOD event are loops containing conditional exits, and conditional subroutine returns. Again, this type of event is rare in most numeric codes, and most occurrences prove to be highly predictable using a branch prediction scheme, giving the compilation system a good method for minimizing the performance impact of these events.

5.3 Control/Data Aliases

Arc 3 in Figure 2 shows the CP needing to wait for the AP to "catch up". This event needs to be generated when the compiler detects a possible read-after-write hazard between a CP read and a write by either the AP or DP. A fragment of code illustrating this event is shown below.

```
      DO 10 I = 1, N
        IV[I] = ... 10
      CONTINUE
      DO 20 I = 1, IV(100)
        ...
20    CONTINUE
```

The loop bound for the second loop is required before it can be dispatched: however, since the bound may be computed by the previous loop, the CP must be prevented from reading `IV(100)` before its new value has been computed. In the ACRI system, it is safe for the CP read to proceed as soon as it is known that all write addresses for the first loop have been generated, since a run-time "alias with outstanding write" check is carried out on all entries in the Store Address Queue takes place, and a read that does conflict is stalled until the data is generated by the DP.

5.4 Sparse and Pointer Operations

Arc 4 in Figure 2 shows an AP-AP dependency, which is significant when the AP needs to wait for an AP memory read to finish before it may initiate a further memory access. A program fragment illustrating this event is shown below:

```
      DO 10 I = 1, N
        ...A[IX[I]]
10    CONTINUE
```

In this fragment, the AP needs to fetch the value of the `IX` element before it can fetch (or generate a write address for) the A array element. This is typical code for sparse vector or sparse matrix operation, and while it is clearly necessary to optimize this for these applications, the occurrence of this events in non-sparse numeric codes is again rare.

Other types of inter-unit dependencies are much less significant than the four identified above: a DP-DP arc, for example, would indicate that a subsequent operation depended on a previous operation, both in the DP: this is a case that occurs in all pipelined machines, and is resolved by a combination of pipeline forwarding, code generation and stalling. A CP-CP arc represents a control-control dependency, of the type that occurs in conventional RISC microprocessors. Again, performance is maximized in these circumstances using conventional architectural techniques: caching minimizes the impact of memory latencies and compiler code scheduling maximizes the overlap of operations.

6 The Cost of an LOD

When the AP is ahead of the DP by an amount of time which is greater than the memory latency, each LDQ pop performed by the DP adds nothing to the program execution time. In effect the latency associated with reading that piece of data is zero. It is therefore useful to talk about the *perceived latency*, as the mean number of cycles the DP must stall each time it tries to pop an item off an LDQ. In the absence of LODs, the perceived latency (after an initial start-up period) is always zero. This is, of course, an ideal situation and in practice the dependencies outlined in section 5 lead to the occasional re-coupling of the AP and DP (as well the CP and the AP and/or DP). To assess the impact of LOD events on the performance of the system we can use a simplistic LOD-penalty model.

Let t_{min} be the idealized execution time of a program when memory latency is zero, let $\bar{P}_{lod}$ be the mean penalty incurred in the DP whenever an LOD occurs, and let N_{lod} be the number of LODs that occur in the program. Naturally, the total execution time $t = t_{min} + \bar{P}_{lod} \cdot N_{lod}$. We can think of the mean LOD penalty as somewhat equivalent to the start-up time of a vector operation, although there are good reasons to believe that LODs are strictly less frequent than vector start-ups. Following on from our definition of t we can say that an efficient decoupled system will have a low LOD penalty and requires a compiler which optimizes for minimal LOD frequency.

7 The Frequency of LODs

To examine the frequencies of the various types of Loss of Decoupling events, we turned to the Perfect Club benchmark suite which contains 12 programs chosen from a range of different supercomputer applications areas, running on problem sets which are small enough to enable investigation on workstation-sized computing environments.

The methodology we adopted was to profile these benchmarks using conventional Unix tools (`prof`, Sun's `tcov`, and Mips's `pixie` programs). In common with many other applications, they show significant instruction locality, in that a small number of routines in each program contributes a large fraction of the execution time. We identified the areas of the programs that dominate the computation, and examined those routines for the syntactic causes of Loss of Decoupling events. Where these events were identified, we describe the impact of the Loss of Decoupling, and suggest ways that the compiler or applications programmer might reduce this impact.

The analyses for six benchmarks are presented here: they are SPICE, a circuit simulation package, OCEAN, an oceanographic modelling program, BDNA, a molecular dynamics program which computes interactions between DNA molecules and an ionic solution, DYFESM, a finite-element package, MDG, a program which simulates the dynamics of water molecules, and QCD, which performs Monte Carlo simulations of quantum chromodynamics. We do not attempt a systematic study of Loss of Decoupling frequency in this paper: this is currently on-going work and will be reported in a subsequent publication.

7.1 Analysis of SPICE

The statements most frequently executed in the SPICE benchmark occur within subroutine `DCDCMP`. This routine, whose purpose is to perform an LU factorization of the matrix giving the coefficients of the circuit, is called by the circuit solver. The functioning of the routine, part of which is illustrated in Figure 3, is significantly obscured by the data representation used and by the fact that SPICE uses an internal memory management package. The code fragment shown in the figure searches for an element located at `(i,j)` in the coefficient matrix, and adjusts its value when it is found. It is clear that a large number of addressing computations are necessary to support each data operation: these are inevitable with the data representation chosen, which allocates elements of the matrix in a vector with no direct mapping of the matrix row and column number to the position in the vector. This sparse allocation is, in turn, inevitable given the constraints of the problem: the matrix represents information between different ‘‘nodes’’ in the circuit, and is necessarily largely full of zeros since most nodes are not connected to most other nodes. In this fragment, however, nearly 3.3 million index array references take place in order to perform 1.4 million data references.

```
                C      LOCATE ELEMENT (I,J)
                C
 343536 ->     135 IF (J.LT.I) GO TO 145
 207553 ->         LOCIJ=LOCC
1014234 ->     140 LOCIJ=NODPLC(IRPT+LOCIJ)
                   IF (NODPLC(IROWNO+LOCIJ).EQ.I)
                       GO TO 155
 806681 ->         GO TO 140
 135983 ->     145 LOCIJ=LOCR
 622430 ->     150 LOCIJ=NODPLC(JCPT+LOCIJ)
                   IF (NODPLC(JCOLNO+LOCIJ).EQ.J)
                       GO TO 155
 486447 ->         GO TO 150
 343536 ->     155 VALUE(LVN+LOCIJ)=VALUE(LVN+LOCIJ)-
                     VALUE(LVN+LOCC)*VALUE(LVN+LOCR)
               160 LOCC=NODPLC(JCPT+LOCC)
                     GO TO 130
 113670 ->     170 LOCR=NODPLC(IRPT+LOCR)
                   IF (IPIV.LE.0) GO TO 125
    270 ->         NODPLC(NUMOFF+I)=NODPLC(NUMOFF+I)-1
                   GO TO 125
```

Figure 3. Locating sparse array elements (SPICE)

This routine is essentially non-decouplable in any reasonable computer structure, since every index array reference causes a control transfer before the routine commits to making a data reference. It is very hard to see any compiler-implemented transformation that would improve this, and the only alternative for users wishing to get significant speedups would be to re-code the algorithm using a more sympathetic data structure, perhaps taking advantage of the much larger amounts of physical memory that are available on machines more recent than when SPICE was originally written.

The situation with the next most frequent group of statements, is much more healthy. There are no address recurrences, and no control recurrences in this routine: the control flow is independent of all addressing and all data arithmetic. This routine therefore decouples fully, in spite of the fact that the loop counts are small. This routine would not benefit greatly from vectorization on a different architecture, but can exploit decoupling. It is responsible for some 2.4 million floating point operations in this benchmark.

The third group of most frequently executed instructions is within function `MEMPTR`, whose purpose is to validate a ‘pointer’ (actually an array subscript) within the SPICE internal memory management package. The core of this routine is shown in Figure 4.

This is a good example of a loop with a premature exit, and since the mean loop count is 66, it benefits well from decoupling. This is despite the fact that it contains no floating-point operations and performs a function that is traditionally regarded as non-numeric.

```
  9055 ->       MEMPTR=.FALSE.
                LTAB=LOCTAB
                LOCPNT=LOCF(IPNTR(1))
                DO 20 I=1,NUMBLK
605843 ->         IF (LOCPNT.NE.ISTACK(LTAB+4)) GO TO 10
  8891 ->         IF (IPNTR(1)*ISTACK(LTAB+5).NE.
              1         ISTACK(LTAB+1)) GO TO 10
  8891 ->           MEMPTR=.TRUE.
                    GO TO 30
596952 ->   10    LTAB=LTAB+NTAB
            20  CONTINUE
  9055 ->   30  RETURN
```

Figure 4. Extract from MEMPTR (SPICE)

Two groups of frequent statements are concerned with copying array elements (in `COPY4`) and zeroing data (in `ZERO8`). These are trivially decouplable.

The final group of statements worth considering lie within the `INTGR8` routine, which performs numeric integration. This routine contains no loops, but again, it is perfectly decouplable, since there are no control or addressing recurrences.

In summary, decoupling appears to be a valuable performance optimization over most of the SPICE benchmark, but it is prevented from full effectiveness by the chosen data representation in one kernel routine.

7.2 Analysis of OCEAN

The OCEAN benchmark is interesting in that the two assignments that are most frequently executed (166 million times) are straightforward array copying operations. It is possible that this arises because the benchmark has been ‘scaled down’ to a reasonable size: the full size production code may have a different ratio of computation to copying.

The most intensive computation occurs inside a complex FFT routine, a fragment of which is shown in Figure 5. Each inner-most loop has a high loop count, and no address recurrences to prevent full exploitation of decoupling. The computation of the array index `JS` can be strength-reduced to a single addition within the loop body, and even the major control transfers (the two arithmetic IF statements on `JL`) may be evaluated while previous loops are continuing to execute, achieving full control decoupling in addition to the access/execute decoupling.

```
   26330 ->       JLI=I2K/2+1
                  DO 109 JL=1,I2K
  385391 ->        IF(JL-1) 102,102,104
   26330 ->   102  EXJ=(1.,0.)
                   DO 103 JJ=JL,NPTS,I2KP
  385391 ->         DO 103 MM=1,MTRN
32826979 ->          JS=(JJ-1)*NSKIP+(MM-1)*MSKIP+1
                     H=DATA(JS)-DATA(JS+I2KS)
                     DATA(JS)=DATA(JS)+DATA(JS+I2KS)
                     DATA(JS+I2KS)=H
              103  CONTINUE
   26330 ->        GO TO 109
  359061 ->   104  IF(JL-JLI) 105,107,105
             C
             C INCREMENT JL-DEPENDENT EXPONENTIAL FACTOR
             C
  336780 ->   105  EXJ=EXJ*EXK
                   DO 106 JJ=JL,NPTS,I2KP
  722562 ->         DO 106 MM=1,MTRN
58901658 ->          JS=(JJ-1)*NSKIP+(MM-1)*MSKIP+1
                     H=DATA(JS)-DATA(JS+I2KS)
                     DATA(JS)=DATA(JS)+DATA(JS+I2KS)
                     DATA(JS+I2KS)=H*EXJ
              106  CONTINUE
  336780 ->        GO TO 109
   22281 ->   107  EXJ=CMPLX(0.,SGN1)
                   DO 108 JJ=JL,NPTS,I2KP
  190671 ->         DO 108 MM=1,MTRN
16218819 ->          JS=(JJ-1)*NSKIP+(MM-1)*MSKIP+1
                     H=DATA(JS)-DATA(JS+I2KS)
                     DATA(JS)=DATA(JS)+DATA(JS+I2KS)
                     DATA(JS+I2KS)=CMPLX(-SGN1*HH(2),
                        SGN1*HH(1))
              108  CONTINUE
  385391 ->   109 CONTINUE
```

Figure 5. Extract from complex FFT routine (OCEAN)

Code inside subroutine `ACAC` accounts for the second largest amount of computation: again, the loops are simply nested, per-forming straightforward computation on array elements indexed by simple strided subscripts, with no recurrences, ensuring that no loss of decoupling occur.

7.3 Analysis of BDNA

BDNA calculates dynamic interactions between organic and non-organic molecules in a complex polarized environment. The vast majority of computation time is spent in subroutine ACTFOR which calculates the interaction between each possible pair of atoms in the environment. The most frequent statements are shown in Figure 6. These calculate the distances between every pair: the array IND is set up to point to every atom that is within 8 Angstroms of the atom I, and a huge body of code (332 lines containing 265 addition and subtractions, 137 multiplications, 23 divisions, 14 square roots and 13 exponentials) is run over that set of atoms. Although this second loop executes with a mean loop count of less than 27, the fact that the loop body is so large means that accesses to IND can be successfully pre-queued by the CP, and hence a potential loss of decoupling point is avoided.

```
                     DO 100 I=1,NSP
                        :
                        :
                     DO 235 J=1,I-1
 5621250 ->             IND(J)=0
                        JNS=(J-1)*ISIT
                        XD=X0(I)-X0(J)
                        YD=Y0(I)-Y0(J)
                        ZD=Z0(I)-Z0(J)
                        XDT(J)=XD-2.D0*
                     1     ALENGT*DBLE(INT(XD*ALENGM))
                        YDT(J)=YD-2.D0*
                     1     ALENGT*DBLE(INT(YD*ALENGM))
                        ZDT(J)=ZD-2.D0*DBLE(INT(ZD))
              C O-O
                        DXS=XDT(J)+SX(INS+1)
                        DYS=YDT(J)+SY(INS+1)
                        DZS=ZDT(J)+SZ(INS+1)
                        RX=DXS-SX(JNS+1)
                        RY=DYS-SY(JNS+1)
                        RZ=DZS-SZ(JNS+1)
                        RSQ=RX*RX+RY*RY+RZ*RZ
                        IF(RSQ.GE.RCUTS) GO TO  235
  196892 ->                IND(J)=1
 5621250 ->   235    CONTINUE
    7495 ->          L=0
                     DO 236 J=1,I-1
 5621250 ->             IF(IND(J).EQ.0) GO TO 236
  196892 ->                L=L+1
                           IND(L)=J
 5621250 ->   236    CONTINUE
```

Figure 6. Extract from ACTFOR (BDNA)

A second group of statements, executed 4.76 million times, relates all interactions between water and DNA molecules: all pairs are considered, without screening by distance. This loop (not shown) is again large (70 statements), containing 67 additions and subtractions, 61 multiplications, 3 divisions and 3 square roots. This loop is fully decouplable.

A further groups of statements, executed 150 thousand times, calculate interaction between water molecules and dissolved ions: other statements in the program are executed much more rarely.

Analysis of this program demonstrates that decoupling can be an effective technique in programs that contain extremely large loop bodies, even if these loops are accessing sparsely stored array elements.

7.4 Analysis of DYFESM

The DYFESM program performs two-dimensional finite element structural analysis using the Explicit Leap Frog method. A large proportion of the execution time is spent in a small number of subroutines. When profiled on a SUN Sparc system, using prof, the time spent in the top four routines accounts for over 85% of the execution time, and on an Alliant FX/80 these same routines account for over 93% of the execution time [3].

7.4.1 Subroutine MATMUL

The matmul subroutine, shown in Figure 7, accounts for around 60% of the execution time of DYFESM when executed on a scalar processor such as that found in a SPARCStation. This is an inherently vectorizable routine, and for example accounts for less than 37% of the execution time on an Alliant FX/80.

The routine contains a triple-nested set of DO loops, which perform a matrix multiplication as a linear combination of columns. The only statement which could possibly interfere with the decoupling of the inner loop is the statement:

```
IF(TEMP.EQ.0.) GOTO 300
```

The intent of this statement is to prevent unnecessary computations from taking place when the multiplier (TEMP) is zero. In fact, TEMP is rarely zero. However, even with this statement in, no loss of decoupling need occur. If all of the non-leaf loops, and all scalar statements outside of non-leaf loops, are executed on the CP, then we can be sure of avoiding any dependency that might cause a loss of decoupling. Here we are assuming that the compiler can detect that there is no overlap between the B and C arrays.

This is one example of the case where, in a multiply-nested loop structure, there is no loss of decoupling on a branch provided that there is no loop-carried dependence from a leaf-loop computation to an outer (non-leaf) scalar computation.

```
                           DIMENSION A(L,M), B(M,N), C(L,N)
   48048 ->              DO 400 K = 1, N
   48048 ->                DO 100 I = 1, L
 2110108 ->                  C(I,K) = 0.
                 100       CONTINUE
   48048 ->                DO 300 J = 1, M
 2162160 ->                  TEMP = B(J,K)
                             IF (TEMP .EQ. 0.) GOTO 300
 2160216 ->                  DO 200 I = 1, L
94871592 ->                    C(I,K) = C(I,K) + A(I,J)*TEMP
                 200         CONTINUE
 2162160 ->      300       CONTINUE
   48048 ->      400     CONTINUE
   48048 ->              RETURN
                         END
```

Figure 7. The MATMUL routine (DYFESM)

7.4.2 Subroutine CHOSOL

The CHOSOL routine, shown in Figure 8, solves $Ax = b$ by Cholesky decomposition. The forward solve phase contains a doubly-nested loop structure. The body of the inner loop consists of a single statement containing a scalar recurrence. This recurrence can be pipelined by promoting SUM to a vector via a conventional scalar expansion transformation. The computation of `B(I)` then becomes

```
B(I) = B(I) - SUM(1) - SUM(2) - ... - SUM(S1)
```

The entire forward solve phase decouples perfectly. However, the value of `B(I)` defined in iteration `I` of the outer loop is then used in iterations `I+1` through `N`. The normal process of pre-loading values for `A(,)` and `B()` lead to Read-After-Write hazards in the Load Address and Store Address queues of the AP - particularly during the early iterations when `I` is small compared with the decoupling distance, and the dynamic flow distance short. In the architecture model assumed in this paper, such memory-RAW hazards are detected by the associative match circuitry in the SAQ and tagged. When the corresponding store data is produced it is automatically forwarded to the appropriate LDQ at the correct position in the queue. This bypass mechanism prevents the compiler having to insert an algorithmic LOD after the completion of each inner loop, which is what would effectively happen in a vector machine.

```
                  SUBROUTINE CHOSOL(A, N, B)
           C
                   DIMENSION A(N,N), B(N)
           C
           C      --- FORWARD SOLVE ---
           C
  34034 ->        DO 53 I=2,N
 844844 ->           SUM = 0.
                     DO 51 L=1,I-1
12496484 ->              SUM = SUM + A(L,I)*B(L)
             51      CONTINUE
 844844 ->           B(I) = B(I) - SUM
             53   CONTINUE
           C
           C      --- DIVIDE BY DIAGONAL ---
           C
  34034 ->        DO 55 I=1,N
 878878 ->           B(I) = B(I)*A(I,I)
             55   CONTINUE
           C
           C      --- BACK SOLVE ---
           C
  34034 ->        DO 80 I = N-1, 1, -1
 844844 ->           SUM = 0.
                     DO 60 L=I+1,N
12496484 ->              SUM = SUM + A(I,L)*B(L)
             60      CONTINUE
 844844 ->           B(I) = B(I) - SUM
             80   CONTINUE
  34034 ->        RETURN
                  END
```

Figure 8. The CHOSOL routine (DYFESM)

7.4.3 Subroutine MNLBYX

The third most prevalent section of the program is subroutine `mnlbyx` This comprises a pair of quadruply-nested loops, with each loop again containing a scalar recurrence. The first inner loop decouples very straightforwardly, but the second (DO 20...) contains a subscripted index in the form of M(I,M1(K)+J,N). The indirection enables the matrix M(I,J,K) which is symmetric upon interchange of I and J, to be stored in a compressed form.

The effect this has on decoupling depends on how the compiler decides to treat the references to M1(K) and M(I,M1(K)+J,N). If the AP reads M1(K), waits until the value arrives from memory, and then computes the address for M(I,M1(K)+J,N) before reading the correct location, then decoupling will be lost. However, there are three ways around this problem:

1. Let the CP prefetch the values of `M1(K)`,
2. Let the AP issue non-blocking loads to the `M1` vector within the AP's inner loop.
3. Implement an address cache the AP so that the average latency for accessing subscripted indices is reduced to a tolerable level.

Any one of these solutions can be used to maintain Access/Execute decoupling throughout this subroutine. Again, as we saw with the previous example, there is a potential memory-RAW hazard on the store to `M(I,J,K)` and subsequent reads from `M` during later iterations.

```
                  SUBROUTINE MNLBYX(M, X, MX)
                   COMMON /INDEX/ M1(ZNNPED)
                  REAL M(NNPES,NNPED*(NNPED+1)/2,3),
                  *      MX(NNPES,NNPED,3),
                 *      X(NDDF,NNPED)
            C
   8008 ->        DO 50 N = 1, 3
  24024 ->         DO 40 I = 1, NNPES
  96096 ->          DO 30 J = 1, NNPED
            C
 864864 ->            SUM = 0.
                      M1J = M1(J)
                      DO 10 K = 1, J
4324320 ->             JK = M1J+K
                       SUM = SUM + M(I,JK,N) * X(3,K)
              10      CONTINUE
            C
 864864 ->            DO 20 K = J+1, NNPED
3459456 ->             JK = M1(K) + J
                       SUM = SUM + M(I,JK,N) * X(3,K)
              20      CONTINUE
            C
 864864 ->            MX(I,J,N) = SUM
              30    CONTINUE
  96096 ->    40   CONTINUE
  24024 ->    50  CONTINUE
   8008 ->        RETURN
                  END
```

Figure 9. The MNLBYX routine (DYFESM)

7.4.4 Subroutine MATMUT

The fourth most prevalent section of code in DYFESM is subroutine `MATMUT`. This performs a matrix transpose multiplication, which from the point of view of decoupling behaves exactly as a conventional matrix multiplication. Needless to say, this subroutine decouples effortlessly.

7.5 Analysis of MDG

The program called MDG in the Perfect Club is a molecular dynamics modelling application which simulates the behaviour of water molecules. On a Cray Y-MP this program is 87.7% vectorizable [3]. It spends most of its time in two routines: `INTERF` and `CSHIFT`, but also makes a significant use of the `SQRT` and `EXP` functions.

7.5.1 Subroutine INTERF

The `INTERF` subroutine calculates inter-molecular interaction forces in three dimensions. For the most part it decouples very well, but there are two places where loss of decoupling appears to be unavoidable.

In the calculation of inter-molecular forces, a test is made to find out if the distance over which an interation occurs is greater than some threshold. If the test is true for all possible interactions on a molecule, then the code which computes forces is skipped. We can see this occurring in the statement:

```
IF(KC.EQ.9) GO TO 1100
```

This is executed 5,923,953 times. The value of KC is computed in the Execution Unit, within the immediately preceding loop. There is no loss of decoupling within the loop which computes KC, since we can use "if-conversion" to turn the statement:

```
IF (RS(K).GT.CUT2) KC=KC+1
```

into a guard computation followed by a guarded increment. However, converting the conditional jump to label 1100 into guarded execution would be difficult and possibly counter-productive since the guarded region is large, and not executed in approximately 37% of the cases. This is a situation where run-time information can be extremely useful to a compiler -- the decision about whether to do if-conversion is a pragmatic one, and depends on dynamic program behaviour. A similar structure occurs later on in the program, and a further 586,530 loss of decoupling events accrue.

```
 5923953 ->          JW1=JW1+NATOMS

                     DO 1110 K=1,9
53315577 ->           RS(K)=XL(K)*XL(K)+YL(K)*YL(K)+
                     *        ZL(K)*ZL(K)
               1110   IF(RS(K).GT.CUT2) KC=KC+1
  5923953 ->          IF(KC.EQ.9) GO TO 1100
  3723689 ->          DO 1120 K=1,14
52131646 ->            FF(K)=0.0D0
               1120   CONTINUE
  3723689 ->          IF(RS(1).GE.CUT2) GO TO 10
  3085663 ->             FF(1)=QQ4/(RS(1)*SQRT(RS(1)))
                     *          +REF4
                        VIR=VIR+FF(1)*RS(1)
  3723689 ->     10   DO 1130 K=2,5
14894756 ->            IF(RS(K).GE.CUT2) GO TO 11
12352784 ->             FF(K)=-QQ2/(RS(K)*SQRT(RS(K)))
                     *          -REF2
                        VIR=VIR+FF(K)*RS(K)
14894756 ->      11    IF(RS(K+4).GT.CUT2) GO TO 1130
12357670 ->             RL(K+4)=SQRT(RS(K+4))
                        FF(K+4)=QQ/(RS(K+4)*RL(K+4))
                     *            +REF1
                        VIR=VIR+FF(K+4)*RS(K+4)
14894756 ->    1130   CONTINUE
  3723689 ->          IF(KC.NE.0) GO TO 20
  2450444 ->           RS(10)=XL(10)*XL(10)+YL(10)*
                     *          YL(10)+ZL(10)*ZL(10)
                       RL(10)=SQRT(RS(10))
                       FF(10)=AB1*EXP(-B1*RL(10))
                     *          /RL(10)
                       VIR=VIR+FF(10)*RS(10)
                       DO 1140 K=11,14
  9801776 ->            FTEMP=AB2*EXP(-B2*RL(K-5))
                     *          /RL(K-5)
                        FF(K-5)=FF(K-5)+FTEMP
                        VIR=VIR+FTEMP*RS(K-5)
                        RS(K)=XL(K)*XL(K)+YL(K)*YL(K)
                     *          +ZL(K)*ZL(K)
                        RL(K)=SQRT(RS(K))
                        FF(K)=(AB3*EXP(-B3*RL(K))-AB4
                     *          *EXP(-B4*RL(K)))/RL(K)
                        VIR=VIR+FF(K)*RS(K)
               1140    CONTINUE
  5923953 ->   1100   CONTINUE
```

Figure 10. Extract from INTERF routine (MDG)

7.5.2 Subroutine CSHIFT

The `CSHIFT` subroutine checks two interacting water molecules to see if they need to be shifted to within half the length of the molecular bounding box. It is a very straightforward piece of code, with no loss of decoupling events. The only loop contains a single `IF` statement. This would be if-converted into a guard evaluation followed by sequence of guarded instructions. No branch instructions need to be executed within this subroutine.

```
 19531449 ->         XL(1)=XMA-XMB
                     XL(2)=XMA-XB(1)
                     XL(3)=XMA-XB(3)
                     XL(4)=XA(1)-XMB
                     XL(5)=XA(3)-XMB
                     XL(6)=XA(1)-XB(1)
                     XL(7)=XA(1)-XB(3)
                     XL(8)=XA(3)-XB(1)
                     XL(9)=XA(3)-XB(3)
                     XL(10)=XA(2)-XB(2)
                     XL(11)=XA(2)-XB(1)
                     XL(12)=XA(2)-XB(3)
                     XL(13)=XA(1)-XB(2)
                     XL(14)=XA(3)-XB(2)
                     DO 100 I=1,14
273440286 ->           IF(ABS(XL(I)).GT.BOXH) XL(I)=XL(I)
                     *     -SIGN(BOXL,XL(I))
273440286 ->  100    CONTINUE
 19531449 ->         RETURN
                     END
```

Figure 11. The CSHIFT routine (MDG)

7.5.3 Loss of Decoupling Frequency in MDG

This program is perhaps unusual for a scientific application, in that the most frequently executed subroutine contains a loss of decoupling. However, even when that happens, the relative frequency of loss of decoupling is still low. According to the definition of MFLOPS for this program, there are over 3.4 billion floating point operations alone. Any processor capable of issuing one floating point add and one floating point multiply per cycle will therefore have an execution time greater than 1.7 billion cycles, and in practice a number of effects will conspire to extend the minimum execution time somewhat beyond that. We can immediately state that the smallest average interval between loss of decoupling events in this program can not be less than 1.7E9/6.5E6 = 262 cycles.

When LODs are close together in time, the associated penalty is likely to be close to the mean memory access time (plus epsilon), but if LODs are widely spaced out in time, then the associated penalty will be closer to the maximum memory access time. Thus, a program with clustered LODs will fare better than a program with evenly-spaced LODs. In MDG the LODs are well spaced out, and will probably experience a comparatively high penalty.

7.6 Analysis of QCD

The QCD program performs a Monte Carlo simulation for quantum chromodynamics using the Pseudo Heat-bath algorithm. On a CRAY Y-MP this program has been measured at a little over 4% vectorizable [1], but a hand-tuned Y-MP/832 version has been benchmarked at 270.9 MFLOPS compared with the baseline compiler version (same machine) which runs at just 13.0 MFLOPS [4]. There are nominally 2.59 billion floating point operations in the benchmarked run for this program.

7.6.1 Subroutine MULT

The `MULT` subroutine contains 18 complex scalar expressions, and this is one of the main reasons that this program vectorizes poorly. However, there are no algorithm structures which could lead to loss of decoupling events, and so we must conclude that this routine will decouple completely. Any problems with LODs during execution of this routine must occur in the calling context just prior to the call to `MULT`.

The DAG for this subroutine contains no common sub-expressions, but many multiple uses of input values. For example, each element of A() and B() is used six times. Also, there are many independent operations. Whilst all additive operations depend on some multiplication operation, there are many independent add and subtract operations within the 18 independent expressions. The number of additive and multiplicative operations is balanced at a ratio of approximately 1:1 (108 multiply and 90 add or subtract operations). A super-scalar code schedule for an Execution unit with one adder and one multiplier would have a makespan of slightly more than 108 cycles (actually, it would be 108 cycles plus the adder pipeline length). Therefore, a decoupled architecture executing this routine could achieve an execution rate of approximately 1.75 flops/clock, assuming that the program is adequately decoupled on entry to the subroutine.

7.6.2 Subroutine SYSLOP

The second most prevalent subroutine is SYSLOP. This is quite a lengthy routine which systematically calculates Wilson loops for SU(3) theory in 3+1 dimensions.

It has a structure from which it is possible, but quite difficult, to remove all loss of decoupling points. The outer loop is a WHILE loop, implemented with tests and GOTO statements. However, the body of the WHILE loop contains a number of nested IF statements with relatively unbalanced THEN and ELSE clauses.

At the outer-most nest level of IF statements we find what is essentially large CASE statement. The determinant of the CASE is an integer variable IC which is assigned at the head of the WHILE loop by reading it from an array. That is something which can be done in advance by the Control Unit, since the address is not determined by any Execution Unit results. On close inspection it becomes apparent that all the code in the WHILE loop, except the calls to mult, cpymat and udag ought to be executed on the CP - including the assignments to SETFLG at the leaf-level within the IF tree. Dependence analysis indicates that there are no dependencies from the calls of mult, cpymat and udag, to any of the subsequent CP computations. If code is partitioned in this way, then all potential loss of decoupling points are removed. This does, however, place a significant load on the CP, which then requires a floating-point arithmetic capability.

If the CP computations within the WHILE loop take longer to execute than the calls to mult, cpymat and udag, then the CP will be the bottleneck. Otherwise the computation will proceed at the rate determined by mult, cpymat and udag, and we have seen that in the case of mult the rate is close to peak. Here is a situation in which control decoupling provides a very significant advantage compared with the two-way (simple Access-Execute) decoupling found in machines such as the ZS-1 [2].

```
2215936 ->      1 NN = NN+1
                  IC = PTR(NN)
                  IF(IC.EQ.14) GOTO 2
              C
1753088 ->        IF(IC.LE.4) THEN
 303104 ->          IND = IC
                    PU  = SITES+ROT(IND)
                    IF(SETFLG.EQ.0) THEN
 180224 ->            CALL CPYMAT(FTEMP(1,INDEX+1),
                                  U1(PU+1),18)
                      SETFLG = 1
                    ELSE
 122880 ->            CALL CPYMAT(FILMAT,U1(PU+1),18)
                      TINDEX = 1-INDEX
                      CALL MULT(FTEMP(1,INDEX+1),FILMAT,
                                FTEMP(1,TINDEX+1))
                      INDEX  = TINDEX
                    ENDIF
 303104 ->          COORD(IND) = MOD(COORD(IND)+1,
                                     LATT1(IND)+1 )
                    IF(COORD(IND).EQ.0) THEN
  37888 ->            SITES = SITES-MOV(IND)*LATT1(IND)
                    ELSE
 265216 ->            SITES = SITES+MOV(IND)
                    ENDIF
              C
 303104 ->        ELSEIF(IC.LE.8) THEN
 495616 ->          IND = IC - 4
                    IF(COORD(IND).EQ.0) THEN
  61952 ->            COORD(IND) = LATT1(IND)
                      SITES = SITES+MOV(IND)*LATT1(IND)
                    ELSE
 433664 ->            COORD(IND) = COORD(IND)-1
                      SITES = SITES-MOV(IND)
                    ENDIF
 495616 ->          PU = SITES+ROT(IND)
                    IF(SETFLG.EQ.0) THEN
 176128 ->            CALL CPYMAT(FILMAT,U1(PU+1),18)
                      CALL UDAG(FILMAT,FTEMP(1,INDEX+1))
                      SETFLG = 1
                    ELSE
 319488 ->            CALL CPYMAT(EXTRA,U1(PU+1),18)
                      CALL UDAG(EXTRA,FILMAT)
                      TINDEX = 1-INDEX
                      CALL MULT(FTEMP(1,INDEX+1),FILMAT,
                              FTEMP(1,TINDEX+1))
                      INDEX  = TINDEX
                    ENDIF
                     :
                     :
                     :
              C
 442368 ->        ENDIF
1753088 ->        GOTO 1
              C
 462848 ->      2 CONTINUE
```

Figure 12. Extract from the SYSLOP routine (QCD)

7.6.3 Subroutine PRNSE2

One subroutine which appears to cause problems for a decoupled architecture is PRNSE2. This contains a very deeply nested loop structure (6 loops deep), with an IF statement at the inner-most level. This can spell trouble for a decoupled machine, but in this case the body of the THEN part is substantial enough so that the loop trip time for the CU computation for the inner-loop ought to

```
                DO 2 I=0,2
  73728 ->       DO 2 P=0,2
 221184 ->        DO 2 J=0,2
 663552 ->         DO 2 Q=0,2
1990656 ->          DO 2 K=0,2
5971968 ->           IF(EPSILO(I+1,J+1,K+1).NE.0) THEN
1327104 ->            DO 3 R=0,2
3981312 ->             IF(EPSILO(P+1,Q+1,R+1).NE.0) THEN
 884736 ->             FAC = EPSILO(I+1,J+1,K+1)
                 .          *EPSILO(P+1,Q+1,R+1)
                   TOT(1)  = TOT(1)+ FAC*U1(1,3*I+P+1)
                 .         *U2(1,3*J+Q+1)*U3(1,3*K+R+1)
                   TOT(1)  = TOT(1)- FAC*U1(2,3*I+P+1)
                 .         *U2(2,3*J+Q+1)*U3(1,3*K+R+1)
                   TOT(1)  = TOT(1)- FAC*U1(1,3*I+P+1)
                 .         *U2(2,3*J+Q+1)*U3(2,3*K+R+1)
                   TOT(1)  = TOT(1)- FAC*U1(2,3*I+P+1)
                 .         *U2(1,3*J+Q+1)*U3(2,3*K+R+1)
                   TOT(2)  = TOT(2)+ FAC*U1(1,3*I+P+1)
                 .         *U2(1,3*J+Q+1)*U3(2,3*K+R+1)
                   TOT(2)  = TOT(2)+ FAC*U1(1,3*I+P+1)
                 .         *U2(2,3*J+Q+1)*U3(1,3*K+R+1)
                   TOT(2)  = TOT(2)+ FAC*U1(2,3*I+P+1)
                 .         *U2(1,3*J+Q+1)*U3(1,3*K+R+1)
                   TOT(2)  = TOT(2)- FAC*U1(2,3*I+P+1)
                 .         *U2(2,3*J+Q+1)*U3(2,3*K+R+1)
                   ENDIF
3981312 ->       3  CONTINUE
1327104 ->           ENDIF
5971968 ->       2 CONTINUE
```

Figure 13. Extract from PRNSE2 (QCD)

be shorter than the AP and DP parts. This means that the CP uses its control decoupling at the inner-most loop level to pre-compute the `IF` conditions and dispatch the inner-most blocks. Observing the execution profile information, we see that the `IF` evaluates TRUE in only 88 out of 398 cases (approximately 22% of the time). So, on average, the CP must go around the inner loop 4.5 times for each dispatch of the inner loop to the AP and DP. It will help greatly if the `EPSILO` array can be cached “close” to the AP and DP, and accessed also by the CP.

An alternative way to remove LODs is to re-structure the loop (a typical hand optimization). This could be done by splitting the loop structures into two: the first would compute a vector of boolean conditions, and the second would read those conditions and decide whether to compute the inner-loop body. Note, that guarded execution does not help in this case, since the code body is large and rarely executed, but branch prediction coupled with speculative dispatch operations is potentially useful optimization.

7.6.4 Loss of decoupling in QCD

Under the assumptions that the CP has floating point capability and that the potential LODs in `PRNSE2` are overcome, there will be very few LODs in QCD. It is worth noting that even the optimized (single processor) version of QCD only attains a performance of 44 MFLOPS on the CRAY Y-MP. This is mostly due to scalar register pressure, and the consequent register spill operations (accounting for approximately 27% of all operations).

8 Conclusions

We have presented control decoupling, a technique for extending the benefits of decoupling to a higher level of abstraction than in previously described decoupled architectures. The principal attraction of control decoupling is that the control flow graph of a program an be searched by the CP in advance of the AP and DP so that events which would otherwise cause an LOD in a purely Access/Execute decoupled architecture do not necessarily disrupt the flow through the AP-memory-DP pipeline. In many cases speculative traversal of the control flow graph of a program by the CP will further improve performance: many control decisions are highly predictable, and so the speculative dispatch of work to the AP and DP is likely to be rewarded.

We describe how particular features of source programs cause loss of decoupling in a three-way decoupled system, and how they negatively impact processor performance, and we examine a range of benchmark programs for the dynamic incidence of these events.

We conclude from this evidence that decoupling is a very powerful technique for minimizing the impact of memory latency, and that it is applicable to a wider range of programs than other architectural optimizations. In particular, we have shown that syntactic LOD events do not always occur at points in a program where one expects to find a vector start-up penalty in a vector machine. As a loss of decoupling event has a penalty somewhat similar in magnitude to a vector start-up, we suggest that control-decoupled architectures offer potentially much higher efficiencies than existing vector machines.

9 References

[1]. Vajapeyam S., Gurindar S.S., Wei-Chung H., “An Empirical Study of the CRAY Y-MP Processor using the Perfect Club Benchmarks”, ASPLOS 1991 (I think).

[2]. J.E. Smith et al., “The ZS-1 Central Processor”, in Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, Palo Alto, CA,1987, pp. 199-204.

[3]. G. Cybenko, L. Kipp, L. Pointer, and D. Kuck, “Supercomputer Evaluation and the Perfect Benchmarks”, in Proc. International Conference on Supercomputing, 1990.

[4]. G. Cybenko, “Supercomputer Performance Trends and the Perfect Benchmarks”, Supercomputing Review, April 1991.

[5]. G.Kane, “MIPS RISC Architecture”, Prentice Hall, Englewood Cliffs, N.J., 1988.

[6]. Goodman J.R., Hsieh J.T., Liou K., Plezkun A.R., Schechter P.B., Young H.C., “PIPE: A VLSI Decoupled Architecture”, 12th Annual International Symposium on Computer Architecture, 1985, Boston, MA, pp. 20-27.

[7]. Smith J.E., “Decoupled Access/Execute Computer Architecture”, ACM Transactions on Computer Systems, Vol. 2, No. 4, November 1984, pp. 289-308.

[8]. Wulf, Wm. A., “Evaluation of the WM Architecture”, Proceedings of the International Symposium on Computer Architecture, Gold Coast, Australia, May 1992, pp. 382-390.

Presented at the Seventh ACM International Conference on Supercomputing,
Tokyo Japan, July 19-23 1993.

The Effectiveness of Decoupling

Peter L. Bird
ACRI,
Lyon
France

Alasdair Rawsthorne,
University of Manchester,
United Kingdom

Nigel P. Topham,
University of Edinburgh
United Kingdom

Abstract

This paper examines the effectiveness of decoupling as an optimization technique for high-performance computer architectures. Decoupled access execute architectures are described, and the concept of *control decoupling* is introduced and justified. A description of a highly-decoupled architecture is given, and a metric for the effectiveness of decoupling on particular programs, the Loss of Decoupling frequency is introduced. Finally, a number of real benchmark programs are examined and the applicability of decoupling them is analyzed.

1 Introduction

A number of papers have discussed the architectural optimization *decoupling* over the last decade (see [2], [6], [7] and [8]). This paper introduces control decoupling, a further technique for increasing performance, and attempts to identify the class of programs over which decoupling is an effective technique to achieve high performance.

The instruction set of modern computers (see, for example, reference [5]) is partitioned into three classes of instruction: control, memory accessing and data operations. The decoupled architectures that have been described to date, for example ZS-1[2], PIPE[6] and WM[8], decouple between the latter two classes, using "Decoupled Access/Execute," in which the addresses for memory references are generated in advance of the execution of data-related instructions. This means that memory read operations can be initiated many cycles before the read data is required for execution, and the latency of main memory read operations (or cache operation) can be hidden. Decoupled Access/Execute is described in detail below, in section 2.

Less conventional is the use of *control decoupling*. This architecture feature is introduced in order to maximize the use of main memory bandwidth in an implementation, by exploring the control-flow graph of a program ahead of the time at which computation is required: this enables requests for packets of computation to be queued ahead of the time at which they are required, so that when one computation has finished, another is ready to take its place on the relevant unit. Control decoupling is described in detail below, in section 3.

In order to determine how valuable these optimizations are, in section 5 we introduce a conceptual framework for program events that cause decoupling to break down: these Loss of Decoupling events are of central importance to understanding the performance of decoupled architectures. In section 6 we discuss briefly the performance impact of Loss of Decoupling events. In section 7, we conduct an examination of a range of popular benchmark programs in order to understand the prevalence of these events.

2 Access Decoupling

In the architecture described here, Access Decoupling is implemented by partitioning user instructions into two classes, memory accessing and user arithmetic.

The memory accessing instructions are run on a special-purpose processor, the Address Processor (AP), whose function is optimized for the purposes of producing regular patterns of addresses, such as found in numeric programs. The most widely used instruction in the AP is an addition operation, which adds register plus register or register plus immediate, writing the result to another register and initiating either a memory read or a memory write operation. Apart from simple additions, the AP has no other data-handling capability: it has no general multiplication, division or logical operations.

The user arithmetic instructions are also run on a special-purpose processor, the Data Processor (DP). This has a full set of integer and floating-point arithmetic and logical operations, but no memory addressing instructions at all.

The AP and DP are connected by two types of queues: the Load Data Queue (LDQ) and the Store Address Queue (SAQ). Entries in these queues are made when the AP generates memory addresses, but the two types of queues are used substantially differently. In the architecture described in this paper there are two independent LDQs connected to the two memory read ports, and one SAQ to drive the single memory write port. The relationships of these processors and their queues can be seen in Figure 1 below.

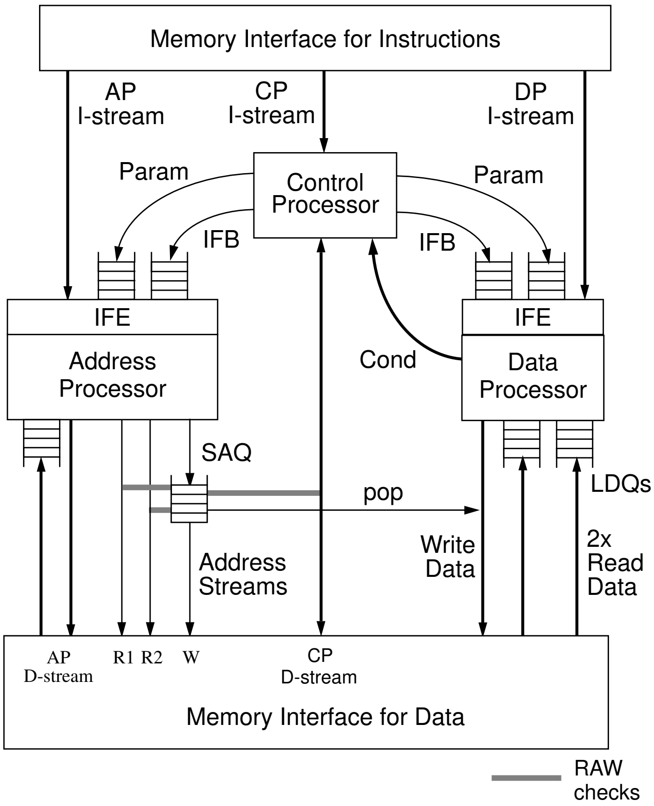

Figure 1. Processor Block Diagram

When the AP generates a memory request for a Load operation, the next free element in the relevant Load Data Queue is allocated and marked "pending". A request for a memory read at that address is sent to the memory interface. When the read data is returned from memory, it is placed into the queue element reserved at the time of the request, and the element is marked as "valid".

The Load Data Queues are available as source operands to instructions running in the DP, and an instruction that reads a queue suspends until the top entry in the queue is marked "valid", and then pops this element. (In fact, the queues are mapped to two general-purpose registers.) In this manner, every read memory access is carried out by two instructions, one on the address processor to initiate the request, and one on the data processor to access the data.

Store operations are handled differently. When an address for a store operation is generated by the AP, the address itself is written to the next free element of the Store Address Queue, and marked "valid". Store operations are initiated in the DP, where every arithmetic instruction contains a "store" bit. When this bit is set, the result of the instruction is sent as data to main memory, in addition to being written to a general-purpose register. When an instruction with its "store" bit set completes, the oldest entry in the Store Address queue is popped, and used as the address for a memory write operation with the data generated by the instruction.

In this way, the AP may proceed through a program, keeping ahead of the place where the DP is executing. This has the beneficial effect of initiating memory read operations early, reducing the impact of memory latency on execution time. In a fully decoupled program, once the decoupling between processors has been established, the execution time is insensitive to latency, provided that the main memory offers sufficient bandwidth to support the request rate generated by the AP.

The purpose of generating store addresses earlier than performance arguments would require is to ensure correct functionality: the instruction-set specification of the architecture defines that memory operations have their semantics defined by the order in which read and write operations are initiated by the AP. If the address of a write operation, for example, is used as the address of a subsequent read operation before the data for the write has been generated in the DP, a comparator detects that the read cannot proceed, and the AP is stalled until the condition can be resolved.

3 Control Decoupling

Control Decoupling is a further optimization that permits a Control Processor (CP) to execute yet further in advance of the AP. The first step is to give the Address and Data processors the capability of running program inner loops, that is of running a body of instructions a number of times, determined either by a loop-count, or terminated in a data-dependent manner. Apart from simple loops, the Address and Data Processors have no other control capability: they have no conditional jumps or subroutine call instructions.

All major control functionality, that is non-inner loop control, subroutine call and return, and dispatching inner loops to the Address and Data Processors, is concentrated in the CP. Since user computation is carried out in the DP, the CP does not need floating-point capabilities, but it does provide a full set of logical operations, integer addition and subtraction, integer multiplication (for array index calculations inner loops have their index arithmetic strength-reduced), integer division (for loop normalization), and a full set of comparison and conditional and unconditional branch operations that would be familiar to the programmer of any conventional RISC instruction set. Memory accessing (both load and store operations) are provided conventionally in the CP instruction set.

In order to reduce the interaction between Control and Data Processors, the DP supports an elaborate conditional-execution scheme, with a full set of comparison operations on integers and floating point operands, conditional execution of any of its instruction set, and a comprehensive set of condition combination instructions, which permit the compilation of nested `if..then..else` statements into guarded execution.

The CP invokes operations of the Address and DP using a special instruction, which dispatches a unit of work called an Instruction Fetch Block (IFB) to one of them. An IFB contains a pointer to the first instruction, a length field, specifying the number of instructions to issue, and a loop count, identifying how many times to issue these instructions. Instruction Fetch Blocks are enqueued by the CP for both the Address and Data Processors: when a processor has finished issuing instructions from one IFB, it can proceed to issuing instructions from the next block, if there is one in the queue, without delay. A set of parameter queues is provided to allow the CP to pass data items to the Address and Data Processors.

In this manner, the normal operation of the system is that the CP is executing instructions from the later part of a program. At the same time, the AP is executing instructions from an earlier part of the program, and the DP is executing from a still earlier part. In this way, all three processors are fully decoupled.

The benefit of Control Decoupling is that while one inner loop is running on the AP, preparatory work for subsequent loops, such as loop count calculation and array subscript arithmetic, may be carried out in parallel, ensuring that no time is lost in the AP between inner loop bodies.

A description of events during program execution that cause this decoupling to break down is found below, in section 5.

3.1 CP Memory Ordering

All addresses produced by the CP are compared against all pending DP write operations, whose addresses are held in the Store Address Queue, and the CP is stalled if any conflict arises. Nevertheless, a logical inconsistency may arise if the CP tries to read data that will be written by the DP by an earlier part of the program, if the address has not yet been generated because the AP has not yet reached this part of the program.

In this system, it is the function of the compiler to eliminate such inconsistencies by preventing the CP from decoupling from the AP when an access of this type is possible.

4. Decoupled Execution: an Example

The diagram in Figure 1 shows an overall picture of the control- and address- decoupled machine. The way in which the two forms of decoupling occur can be explained by reference to an example. Consider the code fragment below :

```
      DO 20 I = 1, M
        DO 10 J = 1, N
          A(I, J) = B(J) * S + C(I,J)
10      CONTINUE
20    CONTINUE
```

The inner loop, in J, is executed autonomously in the Address and Data Processors: in each iteration, the AP initiates a memory read for `B(J)` and `C(I,J)` and a queues the write address for `A(I,J)`, and the DP multiplies one Load Data Queue element by the register holding `S`, adds another Load Data Queue element and stores the result. The AP relies on the availability of stride values for the `A(I,J)` and `C(I,J)` accesses to avoid the need to do full multiplications within the loop.

The CP implements the loop in I, and prepares the Instruction Fetch Blocks and stride values for the AP and the DP. These are passed to these Processors via the queues, enabling the CP to iterate around the loop in I without re-synchronizing with the Address or DP on each iteration.

5 When does Decoupling Break Down?

Figure 2 gives a framework for discussing the influence of program events on both control and access decoupling. The broad arrows show how the decoupling optimization is successful when the CP is transferring information to the AP, and when the AP is transferring information to the DP. The numbered arrows, in the reverse direction, represent inter-processor dependencies that can interfere with decoupling by requiring that the CP or AP wait for a later unit

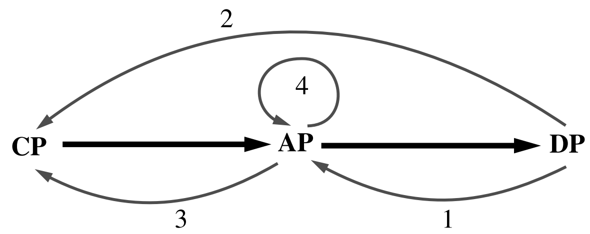

Figure 2. Events that May Destroy Decoupling

to “catch up”. We call each of these events a “Loss of Decoupling”; each numbered arc in Figure 2 corresponds to exactly one type of LOD event.

5.1 Computed Index Operations

Arc 1 in Figure 2 represents the case of the AP needing to wait for the DP before initiating further operand fetches: this case arises when a value computed in the DP must be conveyed back to the AP to take part in address formation. A program fragment causing such an event is illustrated below:

```
      DO 10 I = 1, N
        X[I] = ...
        IX = ...X[I]
        Y[IX] = ...
10    CONTINUE
```

This case is rare across the full range of numeric codes: it arises in “Particle-In-Cell” codes, where a real particle position is calculated and then quantized into a polygonal grid, and in Monte Carlo codes, in which a discrete item is selected using a real-valued random number generator.

5.2 Conditional Control Flow Operations

Arc 2 in Figure 2 shows the CP needing to wait for a condition to be evaluated in the DP before it can issue further instructions: at first glance, this event might seem to occur every time a conditional statement is executed. However, since the DP can implement conditionals internally, through the medium of conditional execution, this event only occurs when a condition causes a major change of sequence in the Control Flow behaviour of the program, for example, when a program conditionally calls a subroutine, or conditionally executes an entire loop body. A program fragment illustrating this type of event is shown below:

```
      DO 10 I=1, 42
        X[I] = ...
10    CONTINUE
      IF (X[42] .NE. 0) CALL RENORM
```

Other types of program fragments which cause this type of LOD event are loops containing conditional exits, and conditional subroutine returns. Again, this type of event is rare in most numeric codes, and most occurrences prove to be highly predictable using a branch prediction scheme, giving the compilation system a good method for minimizing the performance impact of these events.

5.3 Control/Data Aliases

Arc 3 in Figure 2 shows the CP needing to wait for the AP to “catch up”. This event needs to be generated when the compiler detects a possible read-after-write hazard between a CP read and a write by either the AP or DP. A fragment of code illustrating this event is shown below.

```
      DO 10 I = 1, N
        IV[I] = ... 10
      CONTINUE
      DO 20 I = 1, IV(100)
        ...
 20   CONTINUE
```

The loop bound for the second loop is required before it can be dispatched: however, since the bound may be computed by the previous loop, the CP must be prevented from reading `IV(100)` before its new value has been computed. In the ACRI system, it is safe for the CP read to proceed as soon as it is known that all write addresses for the first loop have been generated, since a run-time "alias with outstanding write" check is carried out on all entries in the Store Address Queue takes place, and a read that does conflict is stalled until the data is generated by the DP.

5.4 Sparse and Pointer Operations

Arc 4 in Figure 2 shows an AP-AP dependency, which is significant when the AP needs to wait for an AP memory read to finish before it may initiate a further memory access. A program fragment illustrating this event is shown below:

```
      DO 10 I = 1, N
        ...A[IX[I]]
 10   CONTINUE
```

In this fragment, the AP needs to fetch the value of the `IX` element before it can fetch (or generate a write address for) the A array element. This is typical code for sparse vector or sparse matrix operation, and while it is clearly necessary to optimize this for these applications, the occurrence of this events in non-sparse numeric codes is again rare.

Other types of inter-unit dependencies are much less significant than the four identified above: a DP-DP arc, for example, would indicate that a subsequent operation depended on a previous operation, both in the DP: this is a case that occurs in all pipelined machines, and is resolved by a combination of pipeline forwarding, code generation and stalling. A CP-CP arc represents a control-control dependency, of the type that occurs in conventional RISC microprocessors. Again, performance is maximized in these circumstances using conventional architectural techniques: caching minimizes the impact of memory latencies and compiler code scheduling maximizes the overlap of operations.

6 The Cost of an LOD

When the AP is ahead of the DP by an amount of time which is greater than the memory latency, each LDQ pop performed by the DP adds nothing to the program execution time. In effect the latency associated with reading that piece of data is zero. It is therefore useful to talk about the *perceived latency*, as the mean number of cycles the DP must stall each time it tries to pop an item off an LDQ. In the absence of LODs, the perceived latency (after an initial start-up period) is always zero. This is, of course, an ideal situation and in practice the dependencies outlined in section 5 lead to the occasional re-coupling of the AP and DP (as well the CP and the AP and/or DP). To assess the impact of LOD events on the performance of the system we can use a simplistic LOD-penalty model.

Let t_{min} be the idealized execution time of a program when memory latency is zero, let $\bar{P}_{lod}$ be the mean penalty incurred in the DP whenever an LOD occurs, and let N_{lod} be the number of LODs that occur in the program. Naturally, the total execution time $t = t_{min} + \bar{P}_{lod} \cdot N_{lod}$. We can think of the mean LOD penalty as somewhat equivalent to the start-up time of a vector operation, although there are good reasons to believe that LODs are strictly less frequent than vector start-ups. Following on from our definition of t we can say that an efficient decoupled system will have a low LOD penalty and requires a compiler which optimizes for minimal LOD frequency.

7 The Frequency of LODs

To examine the frequencies of the various types of Loss of Decoupling events, we turned to the Perfect Club benchmark suite which contains 12 programs chosen from a range of different supercomputer applications areas, running on problem sets which are small enough to enable investigation on workstation-sized computing environments.

The methodology we adopted was to profile these benchmarks using conventional Unix tools (`prof`, Sun's `tcov`, and Mips's `pixie` programs). In common with many other applications, they show significant instruction locality, in that a small number of routines in each program contributes a large fraction of the execution time. We identified the areas of the programs that dominate the computation, and examined those routines for the syntactic causes of Loss of Decoupling events. Where these events were identified, we describe the impact of the Loss of Decoupling, and suggest ways that the compiler or applications programmer might reduce this impact.

The analyses for six benchmarks are presented here: they are SPICE, a circuit simulation package, OCEAN, an oceanographic modelling program, BDNA, a molecular dynamics program which computes interactions between DNA molecules and an ionic solution, DYFESM, a finite-element package, MDG, a program which simulates the dynamics of water molecules, and QCD, which performs Monte Carlo simulations of quantum chromodynamics. We do not attempt a systematic study of Loss of Decoupling frequency in this paper: this is currently on-going work and will be reported in a subsequent publication.

7.1 Analysis of SPICE

The statements most frequently executed in the SPICE benchmark occur within subroutine `DCDCMP`. This routine, whose purpose is to perform an LU factorization of the matrix giving the coefficients of the circuit, is called by the circuit solver. The functioning of the routine, part of which is illustrated in Figure 3, is significantly obscured by the data representation used and by the fact that SPICE uses an internal memory management package. The code fragment shown in the figure searches for an element located at `(i,j)` in the coefficient matrix, and adjusts its value when it is found. It is clear that a large number of addressing computations are necessary to support each data operation: these are inevitable with the data representation chosen, which allocates elements of the matrix in a vector with no direct mapping of the matrix row and column number to the position in the vector. This sparse allocation is, in turn, inevitable given the constraints of the problem: the matrix represents information between different ‘‘nodes’’ in the circuit, and is necessarily largely full of zeros since most nodes are not connected to most other nodes. In this fragment, however, nearly 3.3 million index array references take place in order to perform 1.4 million data references.

```
           C      LOCATE ELEMENT (I,J)
           C
 343536 ->    135 IF (J.LT.I) GO TO 145
 207553 ->        LOCIJ=LOCC
1014234 ->    140 LOCIJ=NODPLC(IRPT+LOCIJ)
                  IF (NODPLC(IROWNO+LOCIJ).EQ.I)
                      GO TO 155
 806681 ->        GO TO 140
 135983 ->    145 LOCIJ=LOCR
 622430 ->    150 LOCIJ=NODPLC(JCPT+LOCIJ)
                  IF (NODPLC(JCOLNO+LOCIJ).EQ.J)
                      GO TO 155
 486447 ->        GO TO 150
 343536 ->    155 VALUE(LVN+LOCIJ)=VALUE(LVN+LOCIJ)-
                    VALUE(LVN+LOCC)*VALUE(LVN+LOCR)
              160 LOCC=NODPLC(JCPT+LOCC)
                    GO TO 130
 113670 ->    170 LOCR=NODPLC(IRPT+LOCR)
                  IF (IPIV.LE.0) GO TO 125
    270 ->        NODPLC(NUMOFF+I)=NODPLC(NUMOFF+I)-1
                  GO TO 125
```

Figure 3. Locating sparse array elements (SPICE)

This routine is essentially non-decouplable in any reasonable computer structure, since every index array reference causes a control transfer before the routine commits to making a data reference. It is very hard to see any compiler-implemented transformation that would improve this, and the only alternative for users wishing to get significant speedups would be to re-code the algorithm using a more sympathetic data structure, perhaps taking advantage of the much larger amounts of physical memory that are available on machines more recent than when SPICE was originally written.

The situation with the next most frequent group of statements, is much more healthy. There are no address recurrences, and no control recurrences in this routine: the control flow is independent of all addressing and all data arithmetic. This routine therefore decouples fully, in spite of the fact that the loop counts are small. This routine would not benefit greatly from vectorization on a different architecture, but can exploit decoupling. It is responsible for some 2.4 million floating point operations in this benchmark.

The third group of most frequently executed instructions is within function `MEMPTR`, whose purpose is to validate a ‘pointer’ (actually an array subscript) within the SPICE internal memory management package. The core of this routine is shown in Figure 4.

This is a good example of a loop with a premature exit, and since the mean loop count is 66, it benefits well from decoupling. This is despite the fact that it contains no floating-point operations and performs a function that is traditionally regarded as non-numeric.

```
  9055 ->        MEMPTR=.FALSE.
                 LTAB=LOCTAB
                 LOCPNT=LOCF(IPNTR(1))
                 DO 20 I=1,NUMBLK
605843 ->          IF (LOCPNT.NE.ISTACK(LTAB+4)) GO TO 10
  8891 ->          IF (IPNTR(1)*ISTACK(LTAB+5).NE.
              1          ISTACK(LTAB+1)) GO TO 10
  8891 ->            MEMPTR=.TRUE.
                     GO TO 30
596952 ->   10     LTAB=LTAB+NTAB
            20   CONTINUE
  9055 ->   30   RETURN
```

Figure 4. Extract from MEMPTR (SPICE)

Two groups of frequent statements are concerned with copying array elements (in `COPY4`) and zeroing data (in `ZERO8`). These are trivially decouplable.

The final group of statements worth considering lie within the `INTGR8` routine, which performs numeric integration. This routine contains no loops, but again, it is perfectly decouplable, since there are no control or addressing recurrences.

In summary, decoupling appears to be a valuable performance optimization over most of the SPICE benchmark, but it is prevented from full effectiveness by the chosen data representation in one kernel routine.

7.2 Analysis of OCEAN

The OCEAN benchmark is interesting in that the two assignments that are most frequently executed (166 million times) are straightforward array copying operations. It is possible that this arises because the benchmark has been ‘scaled down’ to a reasonable size: the full size production code may have a different ratio of computation to copying.

The most intensive computation occurs inside a complex FFT routine, a fragment of which is shown in Figure 5. Each inner-most loop has a high loop count, and no address recurrences to prevent full exploitation of decoupling. The computation of the array index `JS` can be strength-reduced to a single addition within the loop body, and even the major control transfers (the two arithmetic IF statements on `JL`) may be evaluated while previous loops are continuing to execute, achieving full control decoupling in addition to the access/execute decoupling.

```
   26330 ->        JLI=I2K/2+1
                   DO 109 JL=1,I2K
  385391 ->         IF(JL-1) 102,102,104
   26330 ->   102   EXJ=(1.,0.)
                    DO 103 JJ=JL,NPTS,I2KP
  385391 ->          DO 103 MM=1,MTRN
32826979 ->           JS=(JJ-1)*NSKIP+(MM-1)*MSKIP+1
                      H=DATA(JS)-DATA(JS+I2KS)
                      DATA(JS)=DATA(JS)+DATA(JS+I2KS)
                      DATA(JS+I2KS)=H
              103   CONTINUE
   26330 ->         GO TO 109
  359061 ->   104   IF(JL-JLI) 105,107,105
            C
            C INCREMENT JL-DEPENDENT EXPONENTIAL FACTOR
            C
  336780 ->   105   EXJ=EXJ*EXK
                    DO 106 JJ=JL,NPTS,I2KP
  722562 ->          DO 106 MM=1,MTRN
58901658 ->           JS=(JJ-1)*NSKIP+(MM-1)*MSKIP+1
                      H=DATA(JS)-DATA(JS+I2KS)
                      DATA(JS)=DATA(JS)+DATA(JS+I2KS)
                      DATA(JS+I2KS)=H*EXJ
              106   CONTINUE
  336780 ->         GO TO 109
   22281 ->   107   EXJ=CMPLX(0.,SGN1)
                    DO 108 JJ=JL,NPTS,I2KP
  190671 ->          DO 108 MM=1,MTRN
16218819 ->           JS=(JJ-1)*NSKIP+(MM-1)*MSKIP+1
                      H=DATA(JS)-DATA(JS+I2KS)
                      DATA(JS)=DATA(JS)+DATA(JS+I2KS)
                      DATA(JS+I2KS)=CMPLX(-SGN1*HH(2),
                          SGN1*HH(1))
              108   CONTINUE
  385391 ->   109 CONTINUE
```

Figure 5. Extract from complex FFT routine (OCEAN)

Code inside subroutine `ACAC` accounts for the second largest amount of computation: again, the loops are simply nested, per-forming straightforward computation on array elements indexed by simple strided subscripts, with no recurrences, ensuring that no loss of decoupling occur.

7.3 Analysis of BDNA

BDNA calculates dynamic interactions between organic and non-organic molecules in a complex polarized environment. The vast majority of computation time is spent in subroutine ACTFOR which calculates the interaction between each possible pair of atoms in the environment. The most frequent statements are shown in Figure 6. These calculate the distances between every pair: the array IND is set up to point to every atom that is within 8 Angstroms of the atom I, and a huge body of code (332 lines containing 265 addition and subtractions, 137 multiplications, 23 divisions, 14 square roots and 13 exponentials) is run over that set of atoms. Although this second loop executes with a mean loop count of less than 27, the fact that the loop body is so large means that accesses to IND can be successfully pre-queued by the CP, and hence a potential loss of decoupling point is avoided.

```
                    DO 100 I=1,NSP
                       :
                       :
                    DO 235 J=1,I-1
 5621250 ->            IND(J)=0
                       JNS=(J-1)*ISIT
                       XD=X0(I)-X0(J)
                       YD=Y0(I)-Y0(J)
                       ZD=Z0(I)-Z0(J)
                       XDT(J)=XD-2.D0*
                    1    ALENGT*DBLE(INT(XD*ALENGM))
                       YDT(J)=YD-2.D0*
                    1    ALENGT*DBLE(INT(YD*ALENGM))
                       ZDT(J)=ZD-2.D0*DBLE(INT(ZD))
              C O-O
                       DXS=XDT(J)+SX(INS+1)
                       DYS=YDT(J)+SY(INS+1)
                       DZS=ZDT(J)+SZ(INS+1)
                       RX=DXS-SX(JNS+1)
                       RY=DYS-SY(JNS+1)
                       RZ=DZS-SZ(JNS+1)
                       RSQ=RX*RX+RY*RY+RZ*RZ
                       IF(RSQ.GE.RCUTS) GO TO  235
  196892 ->               IND(J)=1
 5621250 ->   235   CONTINUE
    7495 ->         L=0
                    DO 236 J=1,I-1
 5621250 ->            IF(IND(J).EQ.0) GO TO 236
  196892 ->               L=L+1
                          IND(L)=J
 5621250 ->   236   CONTINUE
```

Figure 6. Extract from ACTFOR (BDNA)

A second group of statements, executed 4.76 million times, relates all interactions between water and DNA molecules: all pairs are considered, without screening by distance. This loop (not shown) is again large (70 statements), containing 67 additions and subtractions, 61 multiplications, 3 divisions and 3 square roots. This loop is fully decouplable.

A further groups of statements, executed 150 thousand times, calculate interaction between water molecules and dissolved ions: other statements in the program are executed much more rarely.

Analysis of this program demonstrates that decoupling can be an effective technique in programs that contain extremely large loop bodies, even if these loops are accessing sparsely stored array elements.

7.4 Analysis of DYFESM

The DYFESM program performs two-dimensional finite element structural analysis using the Explicit Leap Frog method. A large proportion of the execution time is spent in a small number of subroutines. When profiled on a SUN Sparc system, using prof, the time spent in the top four routines accounts for over 85% of the execution time, and on an Alliant FX/80 these same routines account for over 93% of the execution time [3].

7.4.1 Subroutine MATMUL

The matmul subroutine, shown in Figure 7, accounts for around 60% of the execution time of DYFESM when executed on a scalar processor such as that found in a SPARCStation. This is an inherently vectorizable routine, and for example accounts for less than 37% of the execution time on an Alliant FX/80.

The routine contains a triple-nested set of DO loops, which perform a matrix multiplication as a linear combination of columns. The only statement which could possibly interfere with the decoupling of the inner loop is the statement:

```
IF(TEMP.EQ.0.) GOTO 300
```

The intent of this statement is to prevent unnecessary computations from taking place when the multiplier (TEMP) is zero. In fact, TEMP is rarely zero. However, even with this statement in, no loss of decoupling need occur. If all of the non-leaf loops, and all scalar statements outside of non-leaf loops, are executed on the CP, then we can be sure of avoiding any dependency that might cause a loss of decoupling. Here we are assuming that the compiler can detect that there is no overlap between the B and C arrays.

This is one example of the case where, in a multiply-nested loop structure, there is no loss of decoupling on a branch provided that there is no loop-carried dependence from a leaf-loop computation to an outer (non-leaf) scalar computation.

```
                      DIMENSION A(L,M), B(M,N), C(L,N)
   48048 ->         DO 400 K = 1, N
   48048 ->           DO 100 I = 1, L
 2110108 ->             C(I,K) = 0.
              100     CONTINUE
   48048 ->           DO 300 J = 1, M
 2162160 ->             TEMP = B(J,K)
                        IF (TEMP .EQ. 0.) GOTO 300
 2160216 ->             DO 200 I = 1, L
94871592 ->               C(I,K) = C(I,K) + A(I,J)*TEMP
              200       CONTINUE
 2162160 ->   300     CONTINUE
   48048 ->   400   CONTINUE
   48048 ->         RETURN
                    END
```

Figure 7. The MATMUL routine (DYFESM)

7.4.2 Subroutine CHOSOL

The CHOSOL routine, shown in Figure 8, solves $Ax = b$ by Cholesky decomposition. The forward solve phase contains a doubly-nested loop structure. The body of the inner loop consists of a single statement containing a scalar recurrence. This recurrence can be pipelined by promoting SUM to a vector via a conventional scalar expansion transformation. The computation of `B(I)` then becomes

```
B(I) = B(I) - SUM(1) - SUM(2) - ... - SUM(S1)
```

The entire forward solve phase decouples perfectly. However, the value of `B(I)` defined in iteration `I` of the outer loop is then used in iterations `I+1` through `N`. The normal process of pre-loading values for `A(,)` and `B()` lead to Read-After-Write hazards in the Load Address and Store Address queues of the AP - particularly during the early iterations when `I` is small compared with the decoupling distance, and the dynamic flow distance short. In the architecture model assumed in this paper, such memory-RAW hazards are detected by the associative match circuitry in the SAQ and tagged. When the corresponding store data is produced it is automatically forwarded to the appropriate LDQ at the correct position in the queue. This bypass mechanism prevents the compiler having to insert an algorithmic LOD after the completion of each inner loop, which is what would effectively happen in a vector machine.

```
                      SUBROUTINE CHOSOL(A, N, B)
           C
                       DIMENSION A(N,N), B(N)
           C
           C          --- FORWARD SOLVE ---
           C
  34034 ->            DO 53 I=2,N
 844844 ->               SUM = 0.
                         DO 51 L=1,I-1
12496484 ->                  SUM = SUM + A(L,I)*B(L)
              51         CONTINUE
 844844 ->               B(I) = B(I) - SUM
              53      CONTINUE
           C
           C          --- DIVIDE BY DIAGONAL ---
           C
  34034 ->            DO 55 I=1,N
 878878 ->               B(I) = B(I)*A(I,I)
              55      CONTINUE
           C
           C          --- BACK SOLVE ---
           C
  34034 ->            DO 80 I = N-1, 1, -1
 844844 ->               SUM = 0.
                         DO 60 L=I+1,N
12496484 ->                  SUM = SUM + A(I,L)*B(L)
              60         CONTINUE
 844844 ->               B(I) = B(I) - SUM
              80      CONTINUE
  34034 ->            RETURN
                      END
```

Figure 8. The CHOSOL routine (DYFESM)

7.4.3 Subroutine MNLBYX

The third most prevalent section of the program is subroutine `mnlbyx` This comprises a pair of quadruply-nested loops, with each loop again containing a scalar recurrence. The first inner loop decouples very straightforwardly, but the second (DO 20...) contains a subscripted index in the form of M(I,M1(K)+J,N). The indirection enables the matrix M(I,J,K) which is symmetric upon interchange of I and J, to be stored in a compressed form.

The effect this has on decoupling depends on how the compiler decides to treat the references to M1(K) and M(I,M1(K)+J,N). If the AP reads M1(K), waits until the value arrives from memory, and then computes the address for M(I,M1(K)+J,N) before reading the correct location, then decoupling will be lost. However, there are three ways around this problem:

1. Let the CP prefetch the values of `M1(K)`,
2. Let the AP issue non-blocking loads to the `M1` vector within the AP's inner loop.
3. Implement an address cache the AP so that the average latency for accessing subscripted indices is reduced to a tolerable level.

Any one of these solutions can be used to maintain Access/Execute decoupling throughout this subroutine. Again, as we saw with the previous example, there is a potential memory-RAW hazard on the store to `M(I,J,K)` and subsequent reads from `M` during later iterations.

```
                    SUBROUTINE MNLBYX(M, X, MX)
                     COMMON /INDEX/ M1(ZNNPED)
                    REAL M(NNPES,NNPED*(NNPED+1)/2,3),
                     *      MX(NNPES,NNPED,3),
                   *      X(NDDF,NNPED)
              C
   8008 ->          DO 50 N = 1, 3
  24024 ->           DO 40 I = 1, NNPES
  96096 ->            DO 30 J = 1, NNPED
              C
 864864 ->              SUM = 0.
                        M1J = M1(J)
                        DO 10 K = 1, J
4324320 ->               JK = M1J+K
                         SUM = SUM + M(I,JK,N) * X(3,K)
                10      CONTINUE
              C
 864864 ->              DO 20 K = J+1, NNPED
3459456 ->               JK = M1(K) + J
                         SUM = SUM + M(I,JK,N) * X(3,K)
                20      CONTINUE
              C
 864864 ->              MX(I,J,N) = SUM
                30     CONTINUE
  96096 ->      40   CONTINUE
  24024 ->      50  CONTINUE
   8008 ->          RETURN
                    END
```

Figure 9. The MNLBYX routine (DYFESM)

7.4.4 Subroutine MATMUT

The fourth most prevalent section of code in DYFESM is subroutine `MATMUT`. This performs a matrix transpose multiplication, which from the point of view of decoupling behaves exactly as a conventional matrix multiplication. Needless to say, this subroutine decouples effortlessly.

7.5 Analysis of MDG

The program called MDG in the Perfect Club is a molecular dynamics modelling application which simulates the behaviour of water molecules. On a Cray Y-MP this program is 87.7% vectorizable [3]. It spends most of its time in two routines: `INTERF` and `CSHIFT`, but also makes a significant use of the `SQRT` and `EXP` functions.

7.5.1 Subroutine INTERF

The `INTERF` subroutine calculates inter-molecular interaction forces in three dimensions. For the most part it decouples very well, but there are two places where loss of decoupling appears to be unavoidable.

In the calculation of inter-molecular forces, a test is made to find out if the distance over which an interation occurs is greater than some threshold. If the test is true for all possible interactions on a molecule, then the code which computes forces is skipped. We can see this occurring in the statement:

```
IF(KC.EQ.9) GO TO 1100
```

This is executed 5,923,953 times. The value of KC is computed in the Execution Unit, within the immediately preceding loop. There is no loss of decoupling within the loop which computes KC, since we can use “if-conversion” to turn the statement:

```
IF (RS(K).GT.CUT2) KC=KC+1
```

into a guard computation followed by a guarded increment. However, converting the conditional jump to label 1100 into guarded execution would be difficult and possibly counter-productive since the guarded region is large, and not executed in approximately 37% of the cases. This is a situation where run-time information can be extremely useful to a compiler -- the decision about whether to do if-conversion is a pragmatic one, and depends on dynamic program behaviour. A similar structure occurs later on in the program, and a further 586,530 loss of decoupling events accrue.

```
5923953 ->          JW1=JW1+NATOMS

                    DO 1110 K=1,9
53315577 ->          RS(K)=XL(K)*XL(K)+YL(K)*YL(K)+
                    *        ZL(K)*ZL(K)
               1110  IF(RS(K).GT.CUT2) KC=KC+1
 5923953 ->          IF(KC.EQ.9) GO TO 1100
 3723689 ->          DO 1120 K=1,14
52131646 ->           FF(K)=0.0D0
               1120  CONTINUE
 3723689 ->          IF(RS(1).GE.CUT2) GO TO 10
 3085663 ->              FF(1)=QQ4/(RS(1)*SQRT(RS(1)))
                    *          +REF4
                       VIR=VIR+FF(1)*RS(1)
 3723689 ->       10  DO 1130 K=2,5
14894756 ->           IF(RS(K).GE.CUT2) GO TO 11
12352784 ->            FF(K)=-QQ2/(RS(K)*SQRT(RS(K)))
                    *          -REF2
                       VIR=VIR+FF(K)*RS(K)
14894756 ->       11   IF(RS(K+4).GT.CUT2) GO TO 1130
12357670 ->            RL(K+4)=SQRT(RS(K+4))
                       FF(K+4)=QQ/(RS(K+4)*RL(K+4))
                    *            +REF1
                       VIR=VIR+FF(K+4)*RS(K+4)
14894756 ->  1130    CONTINUE
 3723689 ->          IF(KC.NE.0) GO TO 20
 2450444 ->           RS(10)=XL(10)*XL(10)+YL(10)*
                    *         YL(10)+ZL(10)*ZL(10)
                      RL(10)=SQRT(RS(10))
                      FF(10)=AB1*EXP(-B1*RL(10))
                    *         /RL(10)
                      VIR=VIR+FF(10)*RS(10)
                      DO 1140 K=11,14
 9801776 ->            FTEMP=AB2*EXP(-B2*RL(K-5))
                    *         /RL(K-5)
                       FF(K-5)=FF(K-5)+FTEMP
                       VIR=VIR+FTEMP*RS(K-5)
                       RS(K)=XL(K)*XL(K)+YL(K)*YL(K)
                    *        +ZL(K)*ZL(K)
                       RL(K)=SQRT(RS(K))
                       FF(K)=(AB3*EXP(-B3*RL(K))-AB4
                    *         *EXP(-B4*RL(K)))/RL(K)
                       VIR=VIR+FF(K)*RS(K)
               1140   CONTINUE
 5923953 ->  1100    CONTINUE
```

Figure 10. Extract from INTERF routine (MDG)

7.5.2 Subroutine CSHIFT

The `CSHIFT` subroutine checks two interacting water molecules to see if they need to be shifted to within half the length of the molecular bounding box. It is a very straightforward piece of code, with no loss of decoupling events. The only loop contains a single `IF` statement. This would be if-converted into a guard evaluation followed by sequence of guarded instructions. No branch instructions need to be executed within this subroutine.

```
 19531449 ->        XL(1)=XMA-XMB
                    XL(2)=XMA-XB(1)
                    XL(3)=XMA-XB(3)
                    XL(4)=XA(1)-XMB
                    XL(5)=XA(3)-XMB
                    XL(6)=XA(1)-XB(1)
                    XL(7)=XA(1)-XB(3)
                    XL(8)=XA(3)-XB(1)
                    XL(9)=XA(3)-XB(3)
                    XL(10)=XA(2)-XB(2)
                    XL(11)=XA(2)-XB(1)
                    XL(12)=XA(2)-XB(3)
                    XL(13)=XA(1)-XB(2)
                    XL(14)=XA(3)-XB(2)
                    DO 100 I=1,14
273440286 ->          IF(ABS(XL(I)).GT.BOXH) XL(I)=XL(I)
                    *     -SIGN(BOXL,XL(I))
273440286 ->  100 CONTINUE
 19531449 ->        RETURN
                    END
```

Figure 11. The CSHIFT routine (MDG)

7.5.3 Loss of Decoupling Frequency in MDG

This program is perhaps unusual for a scientific application, in that the most frequently executed subroutine contains a loss of decoupling. However, even when that happens, the relative frequency of loss of decoupling is still low. According to the definition of MFLOPS for this program, there are over 3.4 billion floating point operations alone. Any processor capable of issuing one floating point add and one floating point multiply per cycle will therefore have an execution time greater than 1.7 billion cycles, and in practice a number of effects will conspire to extend the minimum execution time somewhat beyond that. We can immediately state that the smallest average interval between loss of decoupling events in this program can not be less than 1.7E9/6.5E6 = 262 cycles.

When LODs are close together in time, the associated penalty is likely to be close to the mean memory access time (plus epsilon), but if LODs are widely spaced out in time, then the associated penalty will be closer to the maximum memory access time. Thus, a program with clustered LODs will fare better than a program with evenly-spaced LODs. In MDG the LODs are well spaced out, and will probably experience a comparatively high penalty.

7.6 Analysis of QCD

The QCD program performs a Monte Carlo simulation for quantum chromodynamics using the Pseudo Heat-bath algorithm. On a CRAY Y-MP this program has been measured at a little over 4% vectorizable [1], but a hand-tuned Y-MP/832 version has been benchmarked at 270.9 MFLOPS compared with the baseline compiler version (same machine) which runs at just 13.0 MFLOPS [4]. There are nominally 2.59 billion floating point operations in the benchmarked run for this program.

7.6.1 Subroutine MULT

The `MULT` subroutine contains 18 complex scalar expressions, and this is one of the main reasons that this program vectorizes poorly. However, there are no algorithm structures which could lead to loss of decoupling events, and so we must conclude that this routine will decouple completely. Any problems with LODs during execution of this routine must occur in the calling context just prior to the call to `MULT`.

The DAG for this subroutine contains no common sub-expressions, but many multiple uses of input values. For example, each element of `A()` and `B()` is used six times. Also, there are many independent operations. Whilst all additive operations depend on some multiplication operation, there are many independent add and subtract operations within the 18 independent expressions. The number of additive and multiplicative operations is balanced at a ratio of approximately 1:1 (108 multiply and 90 add or subtract operations). A super-scalar code schedule for an Execution unit with one adder and one multiplier would have a makespan of slightly more than 108 cycles (actually, it would be 108 cycles plus the adder pipeline length). Therefore, a decoupled architecture executing this routine could achieve an execution rate of approximately 1.75 flops/clock, assuming that the program is adequately decoupled on entry to the subroutine.

7.6.2 Subroutine SYSLOP

The second most prevalent subroutine is `SYSLOP`. This is quite a lengthy routine which systematically calculates Wilson loops for SU(3) theory in 3+1 dimensions.

It has a structure from which it is possible, but quite difficult, to remove all loss of decoupling points. The outer loop is a WHILE loop, implemented with tests and `GOTO` statements. However, the body of the WHILE loop contains a number of nested `IF` statements with relatively unbalanced `THEN` and `ELSE` clauses.

At the outer-most nest level of IF statements we find what is essentially large CASE statement. The determinant of the CASE is an integer variable `IC` which is assigned at the head of the WHILE loop by reading it from an array. That is something which can be done in advance by the Control Unit, since the address is not determined by any Execution Unit results. On close inspection it becomes apparent that all the code in the WHILE loop, except the calls to `mult`, `cpymat` and `udag` ought to be executed on the CP - including the assignments to `SETFLG` at the leaf-level within the `IF` tree. Dependence analysis indicates that there are no dependencies from the calls of `mult`, `cpymat` and `udag`, to any of the subsequent CP computations. If code is partitioned in this way, then all potential loss of decoupling points are removed. This does, however, place a significant load on the CP, which then requires a floating-point arithmetic capability.

If the CP computations within the WHILE loop take longer to execute than the calls to `mult`, `cpymat` and `udag`, then the CP will be the bottleneck. Otherwise the computation will proceed at the rate determined by `mult`, `cpymat` and `udag`, and we have seen that in the case of `mult` the rate is close to peak. Here is a situation in which control decoupling provides a very significant advantage compared with the two-way (simple Access-Execute) decoupling found in machines such as the ZS-1 [2].

```
2215936 ->     1 NN = NN+1
                 IC = PTR(NN)
                 IF(IC.EQ.14) GOTO 2
           C
1753088 ->       IF(IC.LE.4) THEN
 303104 ->         IND = IC
                   PU  = SITES+ROT(IND)
                   IF(SETFLG.EQ.0) THEN
 180224 ->           CALL CPYMAT(FTEMP(1,INDEX+1),
                                 U1(PU+1),18)
                     SETFLG = 1
                   ELSE
 122880 ->           CALL CPYMAT(FILMAT,U1(PU+1),18)
                     TINDEX = 1-INDEX
                     CALL MULT(FTEMP(1,INDEX+1),FILMAT,
                               FTEMP(1,TINDEX+1))
                     INDEX  = TINDEX
                   ENDIF
 303104 ->         COORD(IND) = MOD(COORD(IND)+1,
                                    LATT1(IND)+1 )
                   IF(COORD(IND).EQ.0) THEN
  37888 ->           SITES = SITES-MOV(IND)*LATT1(IND)
                   ELSE
 265216 ->           SITES = SITES+MOV(IND)
                   ENDIF
           C
 303104 ->       ELSEIF(IC.LE.8) THEN
 495616 ->         IND = IC - 4
                   IF(COORD(IND).EQ.0) THEN
  61952 ->           COORD(IND) = LATT1(IND)
                     SITES = SITES+MOV(IND)*LATT1(IND)
                   ELSE
 433664 ->           COORD(IND) = COORD(IND)-1
                     SITES = SITES-MOV(IND)
                   ENDIF
 495616 ->         PU = SITES+ROT(IND)
                   IF(SETFLG.EQ.0) THEN
 176128 ->           CALL CPYMAT(FILMAT,U1(PU+1),18)
                     CALL UDAG(FILMAT,FTEMP(1,INDEX+1))
                     SETFLG = 1
                   ELSE
 319488 ->           CALL CPYMAT(EXTRA,U1(PU+1),18)
                     CALL UDAG(EXTRA,FILMAT)
                     TINDEX = 1-INDEX
                     CALL MULT(FTEMP(1,INDEX+1),FILMAT,
                               FTEMP(1,TINDEX+1))
                     INDEX  = TINDEX
                   ENDIF
                    :
                    :
                    :
           C
 442368 ->       ENDIF
1753088 ->       GOTO 1
           C
 462848 ->     2 CONTINUE
```

Figure 12. Extract from the SYSLOP routine (QCD)

7.6.3 Subroutine PRNSE2

One subroutine which appears to cause problems for a decoupled architecture is `PRNSE2`. This contains a very deeply nested loop structure (6 loops deep), with an `IF` statement at the inner-most level. This can spell trouble for a decoupled machine, but in this case the body of the `THEN` part is substantial enough so that the loop trip time for the CU computation for the inner-loop ought to be shorter than the AP and DP parts. This means that the CP uses its control decoupling at the inner-most loop level to pre-compute the `IF` conditions and dispatch the inner-most blocks. Observing the execution profile information, we see that the `IF` evaluates TRUE in only 88 out of 398 cases (approximately 22% of the time). So, on average, the CP must go around the inner loop 4.5 times for each dispatch of the inner loop to the AP and DP. It will help greatly if the `EPSILO` array can be cached “close” to the AP and DP, and accessed also by the CP.

```
                DO 2 I=0,2
  73728 ->       DO 2 P=0,2
 221184 ->        DO 2 J=0,2
 663552 ->         DO 2 Q=0,2
1990656 ->          DO 2 K=0,2
5971968 ->           IF(EPSILO(I+1,J+1,K+1).NE.0) THEN
1327104 ->            DO 3 R=0,2
3981312 ->             IF(EPSILO(P+1,Q+1,R+1).NE.0) THEN
 884736 ->             FAC = EPSILO(I+1,J+1,K+1)
           .                *EPSILO(P+1,Q+1,R+1)
                       TOT(1)  = TOT(1)+ FAC*U1(1,3*I+P+1)
           .                 *U2(1,3*J+Q+1)*U3(1,3*K+R+1)
                       TOT(1)  = TOT(1)- FAC*U1(2,3*I+P+1)
           .                 *U2(2,3*J+Q+1)*U3(1,3*K+R+1)
                       TOT(1)  = TOT(1)- FAC*U1(1,3*I+P+1)
           .                 *U2(2,3*J+Q+1)*U3(2,3*K+R+1)
                       TOT(1)  = TOT(1)- FAC*U1(2,3*I+P+1)
           .                 *U2(1,3*J+Q+1)*U3(2,3*K+R+1)
                       TOT(2)  = TOT(2)+ FAC*U1(1,3*I+P+1)
           .                 *U2(1,3*J+Q+1)*U3(2,3*K+R+1)
                       TOT(2)  = TOT(2)+ FAC*U1(1,3*I+P+1)
           .                 *U2(2,3*J+Q+1)*U3(1,3*K+R+1)
                       TOT(2)  = TOT(2)+ FAC*U1(2,3*I+P+1)
           .                 *U2(1,3*J+Q+1)*U3(1,3*K+R+1)
                       TOT(2)  = TOT(2)- FAC*U1(2,3*I+P+1)
           .                 *U2(2,3*J+Q+1)*U3(2,3*K+R+1)
                       ENDIF
3981312 ->       3  CONTINUE
1327104 ->            ENDIF
5971968 ->       2  CONTINUE
```

Figure 13. Extract from PRNSE2 (QCD)

An alternative way to remove LODs is to re-structure the loop (a typical hand optimization). This could be done by splitting the loop structures into two: the first would compute a vector of boolean conditions, and the second would read those conditions and decide whether to compute the inner-loop body. Note, that guarded execution does not help in this case, since the code body is large and rarely executed, but branch prediction coupled with speculative dispatch operations is potentially useful optimization.

7.6.4 Loss of decoupling in QCD

Under the assumptions that the CP has floating point capability and that the potential LODs in `PRNSE2` are overcome, there will be very few LODs in QCD. It is worth noting that even the optimized (single processor) version of QCD only attains a performance of 44 MFLOPS on the CRAY Y-MP. This is mostly due to scalar register pressure, and the consequent register spill operations (accounting for approximately 27% of all operations).

8 Conclusions

We have presented control decoupling, a technique for extending the benefits of decoupling to a higher level of abstraction than in previously described decoupled architectures. The principal attraction of control decoupling is that the control flow graph of a program an be searched by the CP in advance of the AP and DP so that events which would otherwise cause an LOD in a purely Access/Execute decoupled architecture do not necessarily disrupt the flow through the AP-memory-DP pipeline. In many cases speculative traversal of the control flow graph of a program by the CP will further improve performance: many control decisions are highly predictable, and so the speculative dispatch of work to the AP and DP is likely to be rewarded.

We describe how particular features of source programs cause loss of decoupling in a three-way decoupled system, and how they negatively impact processor performance, and we examine a range of benchmark programs for the dynamic incidence of these events.

We conclude from this evidence that decoupling is a very powerful technique for minimizing the impact of memory latency, and that it is applicable to a wider range of programs than other architectural optimizations. In particular, we have shown that syntactic LOD events do not always occur at points in a program where one expects to find a vector start-up penalty in a vector machine. As a loss of decoupling event has a penalty somewhat similar in magnitude to a vector start-up, we suggest that control-decoupled architectures offer potentially much higher efficiencies than existing vector machines.

9 References

[1]. Vajapeyam S., Gurindar S.S., Wei-Chung H., “An Empirical Study of the CRAY Y-MP Processor using the Perfect Club Benchmarks”, ASPLOS 1991 (I think).

[2]. J.E. Smith et al., “The ZS-1 Central Processor”, in Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, Palo Alto, CA,1987, pp. 199-204.

[3]. G. Cybenko, L. Kipp, L. Pointer, and D. Kuck, “Supercomputer Evaluation and the Perfect Benchmarks”, in Proc. International Conference on Supercomputing, 1990.

[4]. G. Cybenko, “Supercomputer Performance Trends and the Perfect Benchmarks”, Supercomputing Review, April 1991.

[5]. G.Kane, “MIPS RISC Architecture”, Prentice Hall, Englewood Cliffs, N.J., 1988.

[6]. Goodman J.R., Hsieh J.T., Liou K., Plezkun A.R., Schechter P.B., Young H.C., “PIPE: A VLSI Decoupled Architecture”, 12th Annual International Symposium on Computer Architecture, 1985, Boston, MA, pp. 20-27.

[7]. Smith J.E., “Decoupled Access/Execute Computer Architecture”, ACM Transactions on Computer Systems, Vol. 2, No. 4, November 1984, pp. 289-308.

[8]. Wulf, Wm. A., “Evaluation of the WM Architecture”, Proceedings of the International Symposium on Computer Architecture, Gold Coast, Australia, May 1992, pp. 382-390.