Decoupled Access/Execute Computer Architectures

JAMES E. SMITH
University of Wisconsin

An architecture for high-performance scalar computation is proposed and discussed. The main feature of the architecture is a high degree of decoupling between operand access and execution. This results in an implementation that has two separate instruction streams that communicate via architectural queues. Performance comparisons with a conventional scalar architecture are given, and these show that significant performance gains can be realized. Single-instruction-stream versions, both physical and conceptual, are discussed, with the primary goal of minimizing the differences with conventional architectures. This allows known compilation and programming techniques to be used. Finally, the problem of deadlock in a decoupled system is discussed, and a deadlock prevention method is given.

Categories and Subject Descriptors: B.2.1 [**Arithmetic and Logic Structures**]: Design Styles—*pipeline*; C.1.1 [**Processor Architectures**]: Single Data Stream Architectures—*pipeline processors*; C.4 [**Computer Systems Organization**]: Performance of Systems—*performance attributes*; C.5.1 [**Computer System Implementation**]: Large and Medium ("mainframe") Computers—*super (very large) computers*

General Terms: Design, Performance

Additional Key Words and Phrases: Pipelined computer systems, decoupled architectures, scalar processing

1. INTRODUCTION

Today's supercomputers are probably best known for their vector mode of operation where a single instruction can cause streams of data to flow through pipelined arithmetic units. Nevertheless, performance in the more traditional scalar mode of operation is recognized as being at least as important as vector performance, and a good balance between the two is what distinguishes the "second-generation" vector processors [15]. Perhaps the most striking example is the CDC CYBER 200 series of vector computers [16], which became commercially viable only after the scalar performance of their first-generation predecessor, the STAR-100 [14], was significantly upgraded.

This paper is a revised version of a paper that appeared in the Proceedings of the 9th Annual Symposium on Computer Architecture (Austin, Tex., Apr. 26–29). *SIGARCH Newsletter* (ACM) *10*, 3 (Apr. 1982), 112–119.

This work was supported by the National Science Foundation under Grant ECS-8207277.

Author's address: Department of Electrical and Computer Engineering, University of Wisconsin, Madison, WI 53706.

The need for balance between scalar and vector processing is very clearly articulated by J. Worlton in [25]. His discussion centers on what he terms "Amdahl's law," which appeared in [2] and which says that if a computer system has a high-speed mode of operation and a low-speed mode, then overall performance will be dominated by the low-speed mode unless the fraction of results generated in low-speed mode can be essentially eliminated. In modern supercomputer terms, the *high-speed mode* is vector operation, and the *low-speed mode* is scalar operation. Empirical evidence of the validity of Amdahl's law is substantial, with some benchmark results by Arnold [4] being a very good example.

In recent years there has been significantly more architectural and organizational innovation in the vector area than the scalar area. Furthermore, the vector mode of operation exploits parallelism that is characterized by minimal data dependencies. This is the same type of parallelism for which arrays of identical processors will be most effective. Hence, most of the proposed methods for exploiting very large-scale integration (VLSI) by bringing large numbers of identical processors to bear on a single problem will yield the best results on what are currently vectorizable problems. Unless there is innovation in processing scalar tasks containing little inherent parallelism, there is danger of a growing imbalance between high-speed and low-speed modes of computation and a limiting of overall system performance as predicted by Amdahl's law.

In this paper, we explore a class of computer architectures and underlying organizations that can substantially improve scalar performance. This is done by using two separate instruction streams that cooperate in executing the same scalar process; the two streams communicate via hardware queues. Instruction decode/issue logic in each stream is simple and requires nothing beyond the current state of the art as exemplified by the CRAY-1 and CYBER 205. Furthermore, demands on software for *scheduling* of code—a common practice in pipelined processors—are reduced. This paper discusses high-performance pipelined processors, as the preceding remarks suggest. Nevertheless, the basic principle can also be applied to much simpler and less expensive processors [23].

1.1 Design Issues for Pipelined Scalar Processors

Most scalar performance improvements made in the last fifteen years have been due to faster logic and memory technology, along with improved packaging and cooling methods. For example, the CRAY-1 [10], representing the state of the art in scalar as well as vector processing, performs scalar computation in fundamentally the same way as the CDC 7600 [5]. In fact, the CDC 6600 [24] and IBM 360/91 [3] were in many ways more sophisticated in their scalar organization than are more recent computers.

Both the CDC 6600 and IBM 360/91 processors attempted to maximize overlap by allowing instructions to be issued out of program sequence, but their complex issue methods have been abandoned by their respective manufacturers. There are at least three reasons for this retrenchment to simpler methods, and these reasons shed some light on the trade-offs involved in high-performance-

processor design:

(1) Complex instruction issue methods tend to require more complicated control, which means longer control paths and a slower clock rate. The reduced clock rate can offset performance gains due to the sophisticated issue schemes.

(2) Complex issue methods lead to more problems in hardware debugging and maintenance; errors are more difficult to reproduce due to less determinism in the order in which instructions are issued and executed.

(3) Some of the performance loss caused by simple issue methods can be compensated for by instruction scheduling by the compiler or assembly language programmer [20]. Of course, this places a significantly greater burden on software.

It has long been known that a practical impediment to scalar performance is that any straightforward instruction decoding/issuing scheme has some bottleneck through which instructions pass at the maximum rate of one per clock period. This bottleneck was first mentioned by Flynn [12] and will henceforth be referred to as the *Flynn bottleneck.* As previously mentioned, modern supercomputer implementations [10, 16] additionally constrain instructions to issue in program sequence.

The architectures discussed in this paper permit improved scalar performance in two important ways. First, the Flynn bottleneck is sidestepped by using two instruction streams. This effectively doubles the maximum available instruction bandwidth. Second, because hardware queues are used for communication between the instruction streams, the streams can "slip" with respect to each other. This leads to what is essentially dynamic scheduling of instructions, previously provided only by the sophisticated issue methods used in the CDC 6600 and IBM 360/91. Moreover, the instruction issue logic used in each instruction stream remains simple.

1.2 Related Work

The main ingredient in the architecture/organization discussed here is the separation of computer processing into two parts: access to memory to fetch operands and store results, and function execution to produce the results. This separation of tasks has been a particularly important feature of several high-performance computers, including those from IBM, Amdahl, CDC, and Cray Research.

The IBM and Amdahl high-performance implementations of the System 360/370 architecture [1, 8] almost invariably decompose the design into an I-*unit* and an E-*unit.* The I-unit fetches the single-instruction stream and makes the appropriate data access requests from memory; meanwhile, instruction control information is forwarded to the E-unit. The data from memory rejoin the control at the E-unit, and the instruction is executed.

CDC and Cray Research architectures further separate the memory access and function execution tasks. They use separate short precision registers for indexing and addressing and longer precision registers for floating-point data. There is a separate set of instructions for each set of registers to support the accessing and execution tasks.

The CSPI MAP-200 [7], an array processor, has pushed the degree of access and execution decoupling beyond that of any of the large-scale computers mentioned above. The MAP-200 uses two instruction streams—one for data access and the other for execution. In this paper the same basic scheme is used, but the MAP-200 architecture places much more of the burden for hardware coordination and synchronization on the software, as is often the case in array processors.

In related work, Hammerstrom and Davidson [13] proposed and studied a theoretical processor model that separates operand access from execution. Partly on the basis of this theoretical study, Pleszkun and Davidson [19] have recently proposed a *structured memory access* architecture that is somewhat reminiscent of the IBM and Amdahl I-unit/E-unit approach, except that in [19] both the I- and E-units can capture program loops. This results in what are effectively two instruction streams during loop execution. In addition, the structured memory access architecture employs several sophisticated techniques that reduce memory references for data.

Finally, architectures that have architectural queues but use a single-instruction stream have been proposed in [6, 22]. Architectures of this type are included in the discussion in Section 4.

1.3 Paper Overview

This paper begins with an overview of decoupled access/execute computer architectures. Then some specific implementation issues are discussed. These are handling of stores, conditional branches, and queues. All three of these are handled in ways intended to place the burden of synchronization and interlocking on the hardware. Next, results of a performance analysis of the 14 Lawrence Livermore Loops [17] are given. This is followed by a discussion of ways in which the two instruction streams of a decoupled access/execute architecture can be merged while retaining most, if not all, of the performance improvement. Finally, a discussion of deadlock, its causes, detection, and prevention, is given.

2. DECOUPLED ARCHITECTURE OVERVIEW

A decoupled access/execute, or simply *decoupled*, architecture is separated into two major functional units: the access processor, or A-*processor*, and the execute processor, or X-*processor* (Figure 1). Each of the processors has its own instruction stream. Hence the two instruction decode/issue units shown in Figure 1 are an essential part of the basic architecture. The instruction caches, on the other hand, are needed to increase instruction fetch bandwidth to match the increased instruction issue bandwidth that a decoupled architecture provides.

Each of the two processors has its own distinct set of registers. In the A-processor these are denoted as registers $A_0, A_1, \ldots, A_{n-1}$ and in the X-processor they are $X_0, X_1, \ldots, X_{m-1}$. The two processors do not necessarily have the same number of registers, nor do the registers have to be the same length.

The two processors execute separate programs that have a similar flowchart structure but perform two different functions. The A-processor performs all operations necessary for transferring data to and from main memory. That is, it does all address computation and performs all memory read and write requests.

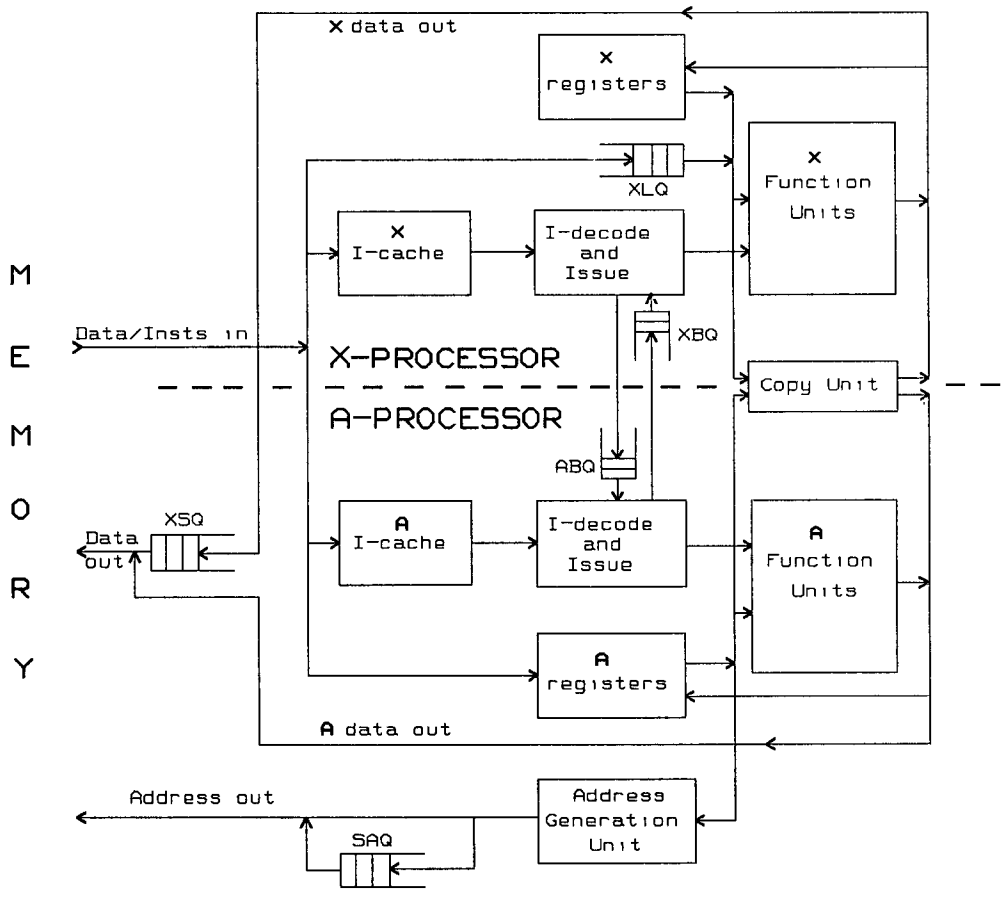

Fig. 1. Block diagram of a decoupled architecture.

In the process, it uses its own set of functional units and registers. Data fetched from memory are either used internally in the A-processor, or are placed in a first-in, first-out (FIFO) queue going to the X-processor. This is the X Load Queue, or XLQ. The X-processor removes operands from the XLQ as it needs them, uses them for computation with its functional units and registers, and places any results into a second FIFO queue, the X-Store Queue or XSQ.

The A-processor issues memory store instructions as soon as it can compute the store address; it does not wait until the store data are received via the XSQ. Store addresses awaiting X data are held in the Store Address Queue or SAQ. As a data item becomes available in the XSQ, it is paired with the first store address in the SAQ and is sent to memory. This pairing takes place automatically when the data become available.

The shared copy unit is used to provide a quick path for data in one processor's register to be passed to the other processor's registers. This is an alternative to passing data through main memory. When the data transfer $X_i \leftarrow A_j$ is required, an instruction is placed in each processor's program. The A-processor instruction

reads A_j and places it in a queue in the copy unit. The X-processor instruction reads the copy unit queue and places the content in X_i. A similar pair of instructions is used for an $A_i \leftarrow X_j$ copy. A full queue blocks issue of an instruction that places data in it. Similarly, an empty queue blocks issue of an instruction that reads it. At a minimum, there is a length-one queue (actually a simple register) in the copy unit for each type of copy. There might be instances when a longer queue would lead to better performance, but for simplicity of implementation length-one queues were used for generating the performance results given in Section 3.

In order for the A- and X-processors to track each other, they must be able to coordinate conditional jumps or branches. FIFO queues are also used for this purpose. These are the X-Branch Queue (XBQ) and A-Branch Queue (ABQ) in Figure 1.

Either processor could conceivably have the data necessary to decide a conditional branch. Consequently, each processor has a set of conditional branch instructions that use its own data. Each processor also has a Branch From Queue (BFQ) instruction that is conditional on the branch outcome at the head of its branch queue coming from the opposite processor. When a processor determines a conditional branch outcome, it places the outcome (*taken* or *not taken*) on the tail of the branch queue to the opposite processor. Thus conditional branch instructions appear in the two processors as pairs. If a conditional branch in the A-processor uses its own internal data, the conditional branch in the X-processor is a BFQ, and vice versa. A BFQ that finds its branch queue empty is blocked from issuing until an outcome arrives.

For performance reasons, it is desirable for the A-processor to determine as many of the conditional branches as possible. This reduces dependency on the X-processor and allows the A-processor to run ahead. Furthermore, if the A-processor is running ahead of the X-processor, branch outcomes in the XBQ can be used by the instruction-fetch hardware in the X-processor to reduce or eliminate instruction-fetch delays related to conditional branches; that is, it is as if the X-processor observes unconditional branches rather than conditional ones. Often, as when a loop counter is also used as an array index, it happens naturally that the A-processor determines conditional branches.

Throughout this paper, examples and comparisons are made with respect to the CRAY-1 architecture and organization. The CRAY-1 was chosen because it

(1) represents the state of the art in scalar performance,[1]
(2) already has an architecture that to some extent decouples access and execution,
(3) has a straightforward implementation that makes timing calculations relatively simple.

Example 1. Figure 2a is the first of the 14 Lawrence Livermore Loops (HYDRO EXCERPT) intended to benchmark FORTRAN performance [21]. Figure 2b is a compilation onto the CRAY-1 architecture; the compiled code is in CRAY

[1] The more recent CRAY X-MP [22] is nearly identical to the CRAY-1 in the way in which it handles scalar operations.

```
        q = 0.0
        Do 1 = 1, 400
1       x(k) = q + y(k)*(r*z(k + 10) + t*z(k + 11))
```

(a)

```
        A4 ← −400          • negative loop count
        A2 ← 0             • initialize index (k)
        A3 ← 1             • index increment
        X2 ← r             • load loop invariants
        X5 ← t             •      into registers
loop:   X3 ← z + 10, A2    • load z(k + 10)
        X7 ← z + 11, A2    • load z(k + 11)
        X4 ← X2*fX3        • r*z(k + 10) − floating multiply
        X3 ← X5*fX7        • t*z(k + 11)
        X7 ← y, A2         • load y(k)
        X6 ← X3 + fX4      • r*z(k + 10) + t*z(k + 11)
        A4 ← A4 + 1        • increment loop count
        X4 ← X7*fX6        • y(k)*(r*z(k + 10) + t*z(k + 11))
        A0 ← A4            • copy loop count for JAM
        x, A4 ← X4         • store into x(k)
        A2 ← A2 + A3       • increment index
        JAM loop           • branch if A0 < 0
```

(b)

```
          Access                          Execute
        A4 ← −400
        A2 ← 0
        A3 ← 1
        XLQ ← r                         X2 ← XLQ
        XLQ ← t                         X5 ← XLQ
loopa:  XLQ ← z + 10, A2        loopx:  X4 ← X2*fXLQ
        XLQ ← z + 11, A2                X3 ← X5*fXLQ
        XLQ ← y, A2                     X6 ← X3 + fX4
        A4 ← A4 + 1                     XSQ ← XLQ*fX6
        X, A2 ← XSQ                     BFQ loopx     • remove outcome from XBQ
        A0 ← A4
        A2 ← A2 + A3
        JAM loopa       • place outcome in XBQ
```

(c)

Fig. 2. (a) The first Lawrence Livermore Loop (HYDRO EXCERPT) (b) CRAY-1 compilation. (c) Decoupled architecture compilation.

assembly language with arrows inserted for readability. The scalar registers have been renamed X_0, X_1, etc., rather than S_0, S_1, etc., in order to make correspondence with the decoupled architecture more clear. Figure 2c contains the A- and X-programs for a decoupled architecture that is patterned after the CRAY-1; the major differences are that there are two instruction streams, and branch outcomes and data are communicated via queues.

2.1 Memory Interlocks

The memory system proposed here returns load data in the same order in which it is requested and does not include an operand cache. Both out-of-order memory loads and an operand cache could be used to implement a decoupled architecture. However, their absence considerably simplifies the necessary interlocks and is consistent with the CRAY-1 implementation.

As for load/store interlocks, every high-performance processor must resolve hazards involving the sequencing of memory reads and writes; a decoupled architecture is no exception. As mentioned earlier, memory addresses for stores may be computed well in advance of when the data are available. These addresses are held in the SAQ, and as store data are placed on the XSQ, they are lined up with their addresses in the SAQ and are sent to memory. The issuing of store instructions before data become available is an important factor in improving performance because it reduces pipeline blockages.

One problem that arises, however, is that a load instruction might use the same memory location (address) as a previously issued, but not yet completed, store. The solution in [7] is to provide the programmer with interlock flags to block instruction issuing when there is any danger of a load bypassing a store to the same location.

An alternative, but slightly more expensive, solution that relieves the programmer (or compiler) of inserting interlocks is to compare each newly issued load address with all the addresses in the SAQ. If there is a match, then the load should be held (and all subsequent loads should be held, possibly by blocking their issue) until the match condition goes away. This associative compare would be a limiting factor on the size of the SAQ, but a size of 8–16 addresses seems feasible and would probably be adequate. This solution can be made more elaborate by "short-circuiting" store data to the load path when the load address matches a store address in the SAQ.

A second problem is the sequencing of stores involving A-processor and X-processor data. If two such stores are to the same location, they should be performed by the A-processor in program sequence. This can be accomplished by strictly forcing all stores to leave the A-processor in order. This might mean buffering an A-processor store address and data behind some earlier store address that is awaiting X-processor data. This could be incorporated into the SAQ mechanism. An alternative would be to treat A-register stores in the same way as the loads. A store address for an A-register store could be compared with the SAQ and blocked if there is a match. With this scheme, A-register stores could pass X-processor stores waiting in the SAQ, provided there are no address conflicts.

2.2 Queue Architecture and Implementation

In order to simplify the instruction set of a decoupled architecture, we reserve a register designator to represent the queue heads and tails. In this way queue operands can be referred to just as registers are, and no special instructions or addressing modes are needed to access the queues. In particular, in the X-processor instruction set we let the highest numbered X-register, X_{n-1}, refer to the XLQ head when it is in a source field, and the XSQ tail when it is in a

destination field. For example,

$$XSQ \leftarrow XLQ + X2$$

takes the first data word off the X load queue, adds to it the content of register X2, and places the result in the XSQ, to be stored in memory. In an 8-register X-processor ($X_0 - X_7$), we could also have used

$$X_7 \leftarrow X_7 + X_2$$

to achieve the same end.

In the A-processor, the instruction

$$XLQ \leftarrow \text{"address"}$$

causes data to be loaded into the tail of the XLQ; the destination can be interpreted from the opcode, and a destination register field is unused with respect to the CRAY-1 counterpart because the CRAY-1 can load into any scalar register. Similarly, a store can be denoted: "address" ← XSQ, although strictly speaking this instruction places the address in the SAQ, where it waits for data to arrive via the XSQ.

A more general alternative is to give the processor access to the top two (or more) elements of a queue, when, for example, the top two elements of a queue are both operands for an add or multiply. In this case, one could use separate register designators for each position in the queue to be accessed. For example, X_{n-1} and X_{n-2} could be the first two elements in the XLQ. The instruction $X_1 \leftarrow X_{n-1} + X_{n-2}$ adds the first two members of the XLQ. A third alternative is to use one register designator for the queues, but to interpret an instruction such as $X_1 \leftarrow X_{n-1} + X_{n-1}$ to mean that the top two elements on the XLQ are to be added. For a noncommutative operation like subtraction, a convention would have to be used so that the elements come off the queue in a predefined order. This method has the advantage of using only one register designator for queues, but it adds somewhat to control complexity because the issue of such an instruction would have to be blocked if either the queue is empty or has only one element. This third alternative is the one used in the simulations reported in Section 3.

The use of registers as queue heads and tails also suggests a convenient and efficient implementation. The XLQ can be implemented as a standard circular buffer held in a register file. A *head counter* and a *tail counter* point to the elements of the register file that are at the head of the queue and at the tail.

The method of using register designators to specify queue operands simplifies testing queues for full and empty conditions. In a typical pipelined processor, for example, the CRAY-1, a set of flip-flops, one for each register, is used to coordinate the reading and writing of registers so that register contents are read and updated in correct sequence. When there issues an instruction that changes a register's contents, the corresponding flip-flop is set to designate the register as being reserved. Any subsequent instruction using the register as an input or output operand encounters the reserved bit and is blocked from issuing. After an instruction that modifies a register completes, the reserved bit is cleared, and any instruction blocked by the bit is allowed to issue.

The queues can be checked for empty/full status by exactly the same reserved bits. For example, if the XLQ is empty, an X_{n-1} reserved bit in the X-processor is set so that any instruction needing an operand for the XLQ is blocked. Similarly, if the XSQ is full, then a second X_{n-1} reserved bit in the X-processor is set so that any instruction needing to place data into the queue is blocked.

2.3 Queue Timing

To support the performance results to be given in the next section, it is necessary to show that data accesses involving the XLQ will be as fast as those involving X registers. What follows assumes a CRAY-1 technology, although one can make a similar argument for most other technologies. The key point is that the queues are implemented as registers.

In the CRAY-1, when an instruction is latched in CIP (the instruction issue register), the operand register designators are latched on the register modules. In fact, CIP is distributed around the machine, and a portion of CIP containing the register designators resides on the register modules. During the next clock period, while an issue decision is being made, the register files are read. If the instruction issues, then the register designators are overwritten as the next instruction is loaded into CIP.

The XLQ could be implemented with the same parts as the X-register files. The designator used to address the queue files are held in the head counter, rather than coming from register designator fields in instructions. In a queue implementation, the head counter could reside on the same modules as the queue register files. As an instruction issue decision is being made, the XLQ can be read exactly as the X registers are. If the instruction issues, then the head counter can be incremented. Since the counter can be completely set up prior to the *issue* signal that causes it to be incremented, the actual change in the counter's state takes no longer than the time required to load the latches. A similar argument can be made for loading the XLQ from memory and incrementing the tail counter.

Passing data to the floating point functional units from the XLQ in addition to X registers means there needs to be multiplexing into the functional units. But, there is already considerable functional unit input multiplexing in the CRAY-1. Each floating point functional unit can receive inputs from X registers, vector registers, and all the other floating-point functional units (to support vector chaining).

To summarize, the time required to load the address latches for the queue register files is the same as the load address latches for the X register files; the time required to read the files is also the same, and the functional unit input multiplexing is already there. Of course, one could only determine the exact time required for accessing the XLQ by implementing it. Nevertheless, the preceding argument is strong enough to support the assumption that XLQ and X register file timing are the same, as is done in the next section.

2.4 Interrupts and Traps

Interrupts and traps are an important consideration in any highly parallel computer, including those with decoupled architectures. Here, the term *interrupt* means any externally caused interruption, whereas a *trap* is caused by some

condition detected in the processor itself. For example, a request for service by an I/O processor is through an interrupt, whereas a floating-point overflow may cause a trap. A *precise* interrupt or trap occurs when the process state is saved so that it can be restarted following service of the interrupt or trap.

Precise interrupts are supported by virtually every processor in existence, the only exceptions being some array processors. (Note that by our terminology the IBM 360/91 provided precise interrupts but did not always provide precise traps.) To support precise interrupts in a pipelined processor, it is only necessary to stop instruction issue and wait until all executing instructions complete. This is also the approach to be used in a decoupled architecture. After instruction execution has ceased, the two program counters need to be saved by hardware. Registers and queues can be saved by hardware, software, or some combination. The use of queues might lead to saving more state information than in a conventional architecture, but this is not necessarily so; a decoupled architecture may need fewer general purpose registers.

Providing precise traps is typically more difficult than precise interrupts. An approach often used is to force the instructions to complete (and modify the process state) in order. This approach has been used in all the high-performance IBM 360/370 implementations following the IBM 360/91. A similar approach could be applied to a decoupled architecture, but to each of the instruction streams separately. This approach may also lead to reduced performance because fast instructions cannot pass slow ones.

The CRAY-1 allows instructions to complete out of order and does not provide precise traps. In order to provide a fair performance comparison, the decoupled implementation described in this paper does the same.

One of the major reasons for providing precise traps is to support page faults in a virtual memory system. Because page faults involving instruction fetches can be handled in much the same way as interrupts (stop instruction issue and wait), they pose no particular problem. Page faults involving data can be made precise in a decoupled architecture by forcing the A instructions to finish in order and by treating the page fault like an interrupt in the X-processor; the X instructions may still finish out of order. Hence, in a decoupled architecture it may be possible to support virtual memory with less performance loss than with a conventional architecture.

3. PERFORMANCE STUDY

In this section, estimates of possible performance improvement with decoupled architecture are made. These focus on scientific programs and use the CRAY-1 for comparison, as discussed prior to Example 1. To get single instruction stream estimates, the 14 Lawrence Livermore Loops [21] were compiled onto the CRAY-1 scalar architecture. The actual object code generated by the CRAY FORTRAN compiler (CFT) was used as a guide, so the level of code optimization and scheduling is realistic. Because of our interest in scalar performance, the CFT *vectorizer* was turned off. The only change made in the CFT scalar code deals with the way index values are manipulated. The CFT compiler keeps the indices in S and T registers, increments them in S registers, and copies them into A registers for indexing. This is not essential and is a quirk of the current version of the CFT compiler [personal communication from D. Hendrickson, Cray

Research, Inc]. In order to provide a fair comparison with a decoupled architecture, the CFT code was changed so that indices are held and incremented in A registers. This was one of the originally intended uses of the A register as described in [10], and it results in about a 10 percent performance improvement.

The 14 loops represent a mix of programs, some of which are vectorizable, and some of which are not. Loop 1, shown in Figure 2, happens to be vectorizable. Loops 4, 5, 6, 8, 11, 13 and 14 are not vectorized by the CFT compiler.

All timing was done with a CRAY-1 simulator [18] and a decoupled architecture simulator developed at the University of Wisconsin. Execution times were calculated in clock periods, using the number of clock periods required by the CRAY-1 for each operation; for example, a load from memory is 11 clock periods, a floating-point add is 6, and a floating-point multiply is 7. Loads, stores, and conditional branches are assumed to require two clock periods to issue, as in the CRAY-1. In [21] loads, stores, and conditional branches in a *simplified* CRAY-1 model were assumed to issue in only one clock period; also branch instruction timing was assumed to be optimum, and memory bank conflicts were ignored. These and other simplifications resulted in the differences in performance figures given in [21] and the more accurate simulator-derived figures given here.

The decoupled compilation was extracted directly from the CFT compilation. No further optimization was done, except for the manipulation of loop index values discussed earlier. Each of the two instruction streams was assumed to issue in strict program sequence, just as in the CRAY-1. The register X_{n-1} is used to designate queue heads and tails, as discussed earlier. As argued in Section 2.3, the time needed to communicate through a queue should be no longer than to communicate through a register. This was assumed in making the time estimates given below.

Figure 3a shows the steady-state timings for the HYDRO EXCERPT loop. The CRAY-1 takes 41 clock periods for each pass through the loop (36 clock periods to get through the loop, plus 5 more for the taken branch at the bottom).

Figure 3b shows the timings for the Access and Execute programs in the decoupled version. In this program, the A-processor decides all the conditional branches and computes all addressing information itself. This means the A-processor is never delayed by the X-processor. The timings given in Figure 3 reflect the steady-state situation where the X-program always finds a nonempty XLQ, although initially the X-program will be held up waiting for its operands.

The A-processor can make each pass through its loop in 16 clock periods (including the 5 for the taken branch). The X-processor takes 20 clock periods and would lag behind the A-processor. Nevertheless, after the first two passes through its loop (where there is a wait by the X-processor for the XLQ) the computation proceeds at the steady-state rate of 20 clock periods per iteration—slightly over twice the speed of the single stream version.

All 14 of the original Lawrence Livermore Loops were simulated as just described. The results are given in Table I. The speedup is computed by dividing the CRAY-1 clock periods by the decoupled architecture clock periods. The speedups vary between 1.1 and 2.1. The average speedup is 1.58. For the 7 nonvectorizable loops the average speedup is 1.41. The higher speedups for loops that are vectorizable occur because the loop iterations are independent, and the

Issue Time (clock period)		
0	loop:	$X_3 \leftarrow Z + 10, A_2$
2		$X_7 \leftarrow Z + 11, A_2$
11		$X_4 \leftarrow X_2 * fX_3$
13		$X_3 \leftarrow X_5 * fX_7$
14		$X_7 \leftarrow y, A_2$
20		$X_6 \leftarrow X_3 + fX_4$
21		$A_4 \leftarrow A_4 + 1$
26		$X_4 \leftarrow X_7 * fX_6$
27		$A_0 \leftarrow A_4$
33		$X, A_2 \leftarrow X_4$
35		$A_2 \leftarrow A_2 + A_3$
36		JAM loop

(a)

Issue Time		*Access*
0	loop*a*:	$XLQ \leftarrow Z + 10, A_2$
2		$XLQ \leftarrow Z + 11, A_2$
4		$XLQ \leftarrow y, A_2$
6		$A_4 \leftarrow A_4 + 1$
7		$X, A_2 \leftarrow XSQ$
9		$A_0 \leftarrow A_4$
10		$A_2 \leftarrow A_2 + A_3$
11		JAM loop*a*

(b)

Issue Time		*Execute*
0	loop*x*:	$X_4 \leftarrow X_2 * fXLQ$
1		$X_3 \leftarrow X_5 * fXLQ$
8		$X_6 \leftarrow X_3 + fX_4$
14		$XSQ \leftarrow XLQ * fX_6$
15		BFQ loop*x*

(c)

Fig. 3. Performance comparison for the first Lawrence Livermore Loop. (a) CRAY-1 machine code and issue timing. (b) Access processor machine code and issue timing. (c) Execute processor machine code and issue timing.

access/execute decoupling is complete. Loops 13 and 14 show the least amount of speedup. This is because data need to be passed from the X-unit to the A-unit. This severely restricts the freedom of the A-unit to run ahead and fetch data in advance of when it is needed. As for the other loops that show relatively low performance, the major reason is that they store data at the bottom of the loop, which are subsequently reloaded when the top of the loop is entered. Memory loads are then delayed because X-unit results must be computed first. Furthermore, because we chose a straightforward memory implementation where the data are not short-circuited, there is additional delay while the data are stored and then reloaded.

Table I. Performance Simulation Results for the 14 Lawrence Livermore Loops

Loop	CRAY-1	Decoupled	Speedup
1	41	20	2.1
2	60	32	1.9
3	26	14	1.9
*4	38	20	1.9
*5	62	48	1.3
*6	64	47	1.4
7	81	56	1.4
*8	200	118	1.7
9	96	58	1.7
10	101	63	1.6
*11	27	21	1.3
12	27	17	1.6
*13	146	135	1.1
*14	149	126	1.2
Average			1.58

[a] All times are in clock periods per loop iteration. Loops that cannot be vectorized by the CFT compiler are marked with an asterisk (*).

4. SINGLE INSTRUCTION STREAM ARCHITECTURES

Although the dual instruction stream decoupled architecture is conceptually simple and leads to straightforward implementations, it does suffer some disadvantages. For the most part, these are due to the human element—the programmer and/or compiler writer must deal with two interacting instruction streams. The programmer problem can be overcome if a high-level language is used. However, this forces the work onto the compiler.

A disadvantage of secondary importance is that two separate instruction caches and decode/issue units are needed, one for each instruction stream. This hardware cost problem can probably be partially alleviated by duplicating the same design for both instruction fetch/decode units.

In this section we briefly outline solutions to the above problems that

(1) physically merge the two instruction streams into one, or
(2) conceptually merge the two instruction streams for the purpose of programming and compilation, but physically separate the instruction streams before the program is loaded into memory.

The simplest way to physically achieve a single instruction stream is to *interleave* the instructions from the two streams. Let $a_1, a_2, \ldots, a_n$ be the sequence of instructions in the A-program and let $x_1, x_2, \ldots, x_m$ be the sequence of instructions in the X-program. An interleaving consists of merging the two sequences into one so that

(1) if a_i precedes a_j in the original A-program, then a_i precedes a_j in the interleaved sequence;
(2) if x_i precedes x_j in the original X-program, then x_i precedes x_j in the interleaved sequence;

(3) if a_k and x_l are corresponding branch instructions in the two sequences, that is, a conditional branch and the corresponding branch from queue or two corresponding unconditional branches, then a single branch instruction is placed in the interleaved sequence, which satisfies the precedence constraints (1) and (2) for both a_k and x_l.

Since the two sequences are interleaved, a bit can be added to each nonbranch instruction, say as part of the opcode, to indicate the stream to which it originally belonged. After instructions are fetched from memory and decoded, the bit can be used to guide instructions to the correct processor for execution. Queues in front of the processors can be used to hold the decoded instructions so that the processors retain the freedom to "slip" with respect to each other. With this scheme, only one program counter is required, and the BFQ instructions are no longer needed.

Eliminating the BFQ instructions may reduce the execution time of the X-program and improve overall performance. In [22] a single-stream architecture that retains the Flynn bottleneck is studied. In some cases, performance is reduced as compared with a two-instruction-stream processor due to the Flynn bottleneck. In other cases, performance is improved because BFQ instructions are removed.

A disadvantage of this method is that the instruction pipeline may be lengthened so that branch performance may be degraded. Also, it should be noted that this approach may reintroduce the one instruction per clock period bottleneck in the instruction fetch/decode pipeline. These would in some instances result in reduced performance. However, it is also true that if one could split the two instruction streams at a rate higher than one instruction per clock period, then the Flynn bottleneck would still be widened. Because the splitting operation is so simple, this is very likely to be possible.

Example 2. An interleaving of the HYDRO EXCERPT program is shown in Figure 4. The processor to which each instruction belongs is noted in parentheses. This particular interleaving places an instruction sending data via a queue immediately before the instruction in the other processor that receives the data. In addition, we use "X_7" as the convention for denoting the queue heads and tails.

From the above example, it can be seen that we are very close to a conventional architecture, which uses different registers for addressing and functional unit execution, that is, the CDC and CRAY architectures. The only difference is that the only X register that may be loaded or stored is X_7 (recall that X_7 is really representing the XLQ or XSQ).

The interleaved architecture appears to be so similar to conventional architectures that many standard compiler techniques can probably be used. Only two special rules regarding the use of X_7 must be followed in order to follow the underlying queue discipline:

(1) After X_7 is loaded, it must be used once and only once.
(2) After data are put into X_7 by an execute instruction, they must be stored.

```
        A4 ← −400          (A)
        A2 ← 0             (A)
        A3 ← 1             (A)
        X7 ← r             (A)
        X2 ← X7            (X)
        X7 ← r             (A)
        X5 ← X7            (X)
loop:   X7 ← z + 10, A2    (A)
        X4 ← X2*fX7        (X)
        X7 → z + 11, A2    (A)
        X3 ← X5*fX7        (X)
        X6 ← X3 + fX4      (X)
        X7 ← y, A2         (A)
        X7 ← X7*fX6        (X)
        A4 ← A4 + 1        (A)
        x, A2 ← X7         (X)
        A0 ← A4            (A)
        A2 ← A2 + A3       (A)
        JAM Loop
```

Fig. 4. An interleaved version of the first Lawrence Livermore Loop.

If one physical instruction stream is to be used, the opcodes are *marked* by the compiler so they can be split at run time.

If two instruction streams are to be used, then as a final step the compiler can pull apart the two instruction streams, by inserting BFQ instructions.

From the preceding discussion it is apparent that many standard compilation techniques can be used for decoupled architectures. Furthermore, the code scheduling problem usually found in a pipelined computer is somewhat reduced because the ability of the A-processor to run ahead of the X-processor results in dynamic code scheduling by the hardware.

5. DEADLOCK

In a decoupled architecture, an instruction can be blocked from issuing if it needs data (or a branch outcome) from a queue and the queue is empty, or if it needs to place data (or a branch outcome) into a queue and the queue is full (this include copy queues). Deadlock occurs if both instruction streams are simultaneously blocked for either of the above reasons. An example of this is shown in Figure 5. The first instruction in each stream is blocked by an empty queue; the next instruction in both streams places data in the other's queue. Note that we are assuming that instructions in a given stream are constrained to issue in program sequence, as has been the case throughout this paper.

Deadlock detection and prevention are both important problems. Deadlock can be detected by simply determining when instruction issue is blocked in both processors owing to full or empty queues, possibly after a delay to ensure that no data are in transit. This should be flagged as a program error, and the program should be purged.

Deadlock prevention is more complicated. We give a sufficient condition for deadlock-free operation and show that meeting the sufficient condition occurs quite naturally.

Access	*Execute*
⋮	⋮
BFQ L1	$X_3 \leftarrow$ XLQ
XLQ $\leftarrow A_3$, disp	JAM L2
⋮	⋮

Fig. 5. A deadlock situation in a decoupled architecture.

Consider the instruction streams as they flow through the processors. Both data and control information (in the form of branch outcomes) are passed via queues. For each data or control transfer through a queue, there is an instruction in one processor that sends the data or control item, and an instruction in the other processor that receives the item. We refer to the instruction that sends a particular item i as *SEND i*, and the instruction that receives it as *RECEIVE i*. For example, in Figure 5, "XLQ $\leftarrow A_3$, disp" sends a data item i that is received by the instruction "X3 $\leftarrow$ XLQ." Hence, the first instruction is SEND i and the second is RECEIVE i. A branch instruction is a SEND i and its corresponding branch from a queue is a RECEIVE i. In general, if an instruction is both a SEND and a RECEIVE, then it can be considered to be a RECEIVE immediately followed by a SEND. For the discussion that follows, we assume queues of length one since this is the most restrictive case.

An interleaving of instructions (Section 4) is defined to be *proper* if the instruction causing SEND i immediately precedes the instruction causing RECEIVE i for all data or control transfers i. The interleaving shown in Figure 4 is proper.

THEOREM. *If the A and X processor instruction streams can be properly interleaved, then deadlock cannot occur.*

PROOF. Deadlock occurs when both instruction streams are blocked due to queues being empty or full. To be more specific, a SEND instruction can be blocked if its queue is full, or a RECEIVE instruction can be blocked if its queue is empty. One such deadlock situation is shown below.

	A-stream	X-stream	
blocked $\rightarrow$	RECEIVE i	RECEIVE j	$\leftarrow$ blocked
	⋮	⋮	
	SEND j	SEND i	

Here, both streams are blocked with empty queues waiting to RECEIVE, while the matching SENDs occur later in the streams. Figure 5 is an example of the preceding situation. We see, however, that a proper interleaving of the two streams is impossible: if SEND i is placed ahead of RECEIVE i, as it must be, then because RECEIVE j precedes SEND i in the X-stream, it must also precede SEND j in any interleaving.

There are three other cases to be considered; these are shown below.

	A-stream	X-stream	
blocked →	RECEIVE i	SEND j	← blocked
	⋮	⋮	
	RECEIVE j	SEND i	

	A-stream	X-stream	
blocked →	SEND j	RECEIVE i	← blocked
	⋮	⋮	
	SEND i	RECEIVE j	

	A-stream	X-stream	
blocked →	SEND i	SEND j	← blocked
	⋮	⋮	
	RECEIVE j	RECEIVE i	

For each of these three other cases, it can be seen that a proper interleaving is impossible. We have just examined all four of the possible deadlock situations and have shown that a proper interleaving is impossible in each case. Using the contrapositive completes the proof.

Code compiled as in Section 4 automatically has this property, since it is compiled so that the static code sections are properly interleaved before being separated.

The program in Figure 4 is a proper interleaving for the HYDRO EXCERPT compilation, so the program must be deadlock free. Turning to Figure 5, it can be seen that it is impossible to interleave the A- and X-programs properly, and the preceding sufficient condition is not satisfied.

6. CONCLUSIONS

It has been shown that a decoupled architecture has the following advantages.

(1) It permits high-speed instruction issue; up to two instructions can be issued per clock period. This is twice the rate for traditional single-instruction stream processors that contain the Flynn bottleneck, that is, some bottleneck in the instruction fetch/decode path where instructions can pass at only one per clock period.

(2) It permits a constrained type of out-of-order instruction issue that allows data access operations to be done well in advance of when they are needed, but without resorting to complicated methods that have been used in the past for out-of-order issue.

(3) Unpredictable (and long) memory access delays can be smoothed and possibly hidden. This is particularly important in modern multiprocessor systems where memory bandwidth is a critical resource.

Decoupled architectures are relatively new, and many variations are possible. Although large-scale implementations are emphasized here, application to smaller

scale systems also seems to be in order. A decoupled architecture offers opportunities for VLSI implementations where *coprocessors* have become an accepted technique for improving performance [9]. By using features of a decoupled architecture, overlap of coprocessors may be significantly increased and effects of memory access delays may be reduced.

ACKNOWLEDGMENT

The author would like to thank Nicholas Pang for developing the performance simulators used in this study and David Anderson for his assistance in obtaining CFT object listings for the Livermore Loops. The author would also like to thank David Patterson, the referees, and the editor for their numerous comments and helpful suggestions.

REFERENCES

1. AMDAHL CORPORATION. *Amdahl 580 Technical Introduction.* Amdahl Corporation, Sunnyvale, Calif. 1982.
2. AMDAHL, G. The validity of the single processor approach to achieving large-scale computing capabilities. In *Proceedings of AFIPS Spring Joint Computer Conference* (Atlantic City, N.J., Apr. 18–20), AFIPS Press, Reston, Va., pp. 483–485.
3. ANDERSON, D.W., SPARACIO, F.J., AND TOMASULO, R.M. The IBM System/360 Model 91: Machine philosophy and instruction handling. *IBM J. Res. Devel. 11*, 1 (Jan. 1967), 8–24.
4. ARNOLD, C.N. Performance evaluation of three automatic vectorizer packages. In *1982 International Conference on Parallel Processing* (Bellaire, Mich., Aug.) IEEE, New York 1982, pp. 235–242.
5. BONSEIGNEUR, P. Description of the 7600 computer system. *Comput. Group News 3*, 5 (May 1969) 11–15.
6. BRANTLEY, W.C., AND WIESS, J. Organization and architecture tradeoffs in FOM. In *IEEE International Workshop on Computer Systems Organization*, (New Orleans, La. Mar. 29–31), IEEE, New York, 1983, pp. 139–143.
7. COHLER, E.U., AND STORER, J.E. Functionally parallel architecture for array processors. *Computer 14*, 9 (Sept. 1981), 28–36.
8. CONNORS, W.D., FLORKOWSKI, J.H., AND PATTON, S.K. The IBM 3033: An inside look. *Datamation 25*, 5 (May 1979), 198–218.
9. Co-Processor cooperates with 8 or 16-bit microprocessors. *Electron. Des. 28*, (Mar. 1980), p. 19.
10. CRAY RESEARCH. *CRAY X-MP Series Mainframe Reference Manual.* Cray Research, Inc., Chippewa Falls, Wis. 1982.
11. CRAY RESEARCH. *CRAY-1 Computer System Hardware Reference Manual.* Cray Research, Inc., Chippewa Falls, Wis. 1976.
12. FLYNN, M.J. Very high-speed computing systems. In *Proceedings of the IEEE 54*, 12 (Dec. 1966), 1901–1909.
13. HAMMERSTROM, D.W., AND DAVIDSON, E.S. Information content of CP memory referencing behavior. In *Proceedings of the 4th Annual Symposium on Computer Architecture* (Mar.). IEEE, New York, 1977, pp. 184–192.
14. HINTZ, R.G., AND TATE, D.P. Control data STAR-100 processor design. In *Proceedings of the IEEE Comcon 1972*, IEEE, New York, (Sept.). 1972, pp. 1–4.
15. KOZDROWICKI, E.W., AND THEIS, D.J. Second generation of vector supercomputers. *Computer 13*, 11 (Nov. 1980), 71–83.
16. LINCOLN, N.R. Technology and design tradeoffs in the creation of a modern supercomputer. *IEEE Trans. Comp. C-31*, 5 (May 1982), 349–362.
17. MCMAHON, F.H. FORTRAN CPU performance analysis. Lawrence Livermore Laboratories, Livermore, Calif., 1972.
18. PANG, N.R., AND SMITH, J.E. CRAY-1 simulation tools. Tech. Rep. Electrical and Computer Engineering Dept., Univ. of Wisconsin, Madison, Wis. Dec. 1983.

19. PLESZKUN, A.R., AND DAVIDSON, E.S. A structural memory access architecture. In *1983 International Conference on Parallel Processing* (Bellaire Mich., Aug. 23–26) IEEE, New York, 1983.
20. RYMARCZYK, J. Coding guidelines for pipelined processors. In Symposium on Architectural Support for Programming Languages and Operating Systems. *ACM SIGARCH Comput. News 10*, 2 (Mar. 1982), 12–19.
21. SMITH, J.E. Decoupled access/execute computer architectures. In *Proceedings of the 9th Annual Symposium on Computer Architecture* (May), 1982.
22. SMITH, J.E., AND KAMINSKI, T.J. Varieties of decoupled access/execute architectures. In *Proceedings of the 20th Allerton Conference* (Oct.). Univ. of Illinois, Monticello, Ill. 1982, pp. 577–586.
23. SMITH, J.E., PLESZKUN, A.R., KATZ, R.H., AND GOODMAN, J.R. PIPE: A high performance VLSI architecture. In *IEEE International Workshop on Computer Systems Organization* (Mar.). IEEE, New York, 1983, pp. 131–138.
24. THORNTON, J.E. *Design of a Computer—The Control Data 6600.* Scott, Foresman, Glenview, Ill., 1970.
25. WORLTON, J. Supercomputers: The philosophy behind the machines. *Computerworld* IN DEPTH Sect. (Nov. 9, 1981), In depth 1–14

Received June 1983; revised April 1984; accepted June 1984