A Simulation Study of Decoupled Architecture Computers

JAMES E. SMITH, SHLOMO WEISS, AND NICHOLAS Y. PANG, MEMBER, IEEE

***Abstract*—Decoupled architectures achieve high scalar performance by cleanly splitting instruction processing into memory access and execution tasks. Several decoupled architectures have been proposed, and they all have two characteristics in common: 1) they have two separate sets of instructions, one for accessing memory and one for performing function execution. 2) The memory accessing task and the execution task communicate via architectural queues.**

These characteristics lead to pipelined computers that have the following advantages: 1) they can issue more than one instruction per clock period; 2) they can dynamically schedule instructions at runtime; 3) they are less sensitive to memory access delays than conventional architectures.

We present a simulation study of decoupled architectures. The simulation models are very detailed, with timing resolution to the clock period. The Lawrence Livermore Loops are used as the workload. We first describe a decoupled architecture based on the CRAY-1 scalar architecture. The sensitivity to memory access delays are studied by varying memory access time over a wide range of values. We show that performance improvements increase linearly over the scalar CRAY-1 as the memory access paths of both are lengthened. Then, we study queue lengths in decoupled machines, and show the affect of queue lengths on performance. Relatively short queues are shown to give optimum, or near-optimum, performance

***Index Terms*—Decoupled architectures, performance evaluation, pipelined processors, scientific computers, supercomputers.**

I. Introduction

HIGH-performance scalar computation is of fundamental importance, despite the increasing attention given to highly parallel methods that use vector processing and multiprocessor systems. In the realm of scientific supercomputers, evidence of the importance of scalar performance is given in [2]. Even for the best vectorizer studied by Arnold, automatic vectorization increases the total throughput of the CDC CYBER 205 by a factor of only 1.7. Furthermore, for the examples studied by Arnold, the scalar time dominates the total computing time: 60 percent scalar mode versus 40 percent vector mode. More recent results by Worlton [22] comparing the CRAY X-MP with various Japanese Supercomputers also serve to illustrate the importance of scalar performance in a scientific computing environment.

The importance of scalar processing goes beyond its significance in vector supercomputers. Cost considerations justify the existence of much less expensive processors that can achieve cost/performance comparable with, or superior to, supercomputers without incorporating an expensive vector unit. The current success of minicomputers and microcomputer-based workstations in scientific and engineering applications indicates the importance of this class of machines.

Scalar performance is also important in multiprocessor systems. Because of the relatively high cost of interprocessor communication, MIMD multicomputers tend to exploit parallelism at a higher level, namely at the level of tasks, rather than parallelism among individual instructions. Any increase in the performance of each of the processing elements, through concurrent execution of instructions from its own instruction stream, supplements the gain achieved through the MIMD configuration and is reflected in an increase of the overall system performance.

A. Decoupled Architectures

In the process of executing a program, a computer's instruction stream performs two interrelated tasks. The first is the memory access task which loads operands and stores results; this includes indexing and other addressing operations. The second is the execution task which operates on the data to produce results. The execution task is often considered to be the "useful" part of computation while the memory access task is often considered to be "overhead." Both are essential, however.

In most conventional architectures these two tasks "intermingle" in a single instruction stream, and high-performance implementations attempt to separate them to increase overlap. The computer's architecture significantly influences the degree to which this can be done, as well as the complexity of the underlying implementation. We give two examples to illustrate this point.

The access and execution tasks are very cleanly separated in most high-performance scientific computer systems, including the CDC 6600 [20], CDC 7600 [3], CRAY-1 [14], [7], the scalar unit of the CDC CYBER 205 [5], and the Denelcor HEP [16]. In these register-register, or RR architectures, functional unit operands come only from registers. To perform a floating point operation, one must first explicitly load both operands

Manuscript received October 26, 1984; revised April 25, 1985. This work was supported in part by the National Science Foundation under Grant ECS-8207277.

J. E. Smith is with the Astronautics Corporation of America, Madison, WI 53617, on leave from the Department of Electrical and Computer Engineering, University of Wisconsin, Madison, WI 53706.

S. Weiss is with Microelectronics and Computer Technology Corporation, Austin, TX 78759.

N. Y. Pang is with Amdahl Corporation, Sunnyvale, CA 94086.

IEEE Log Number 8609400.

into registers before the operation is performed, and then explicitly store the result register afterwards. All instructions are handled in a uniform way with all registers read at one stage in the pipeline. An operand cache is not essential for good performance because load instructions can be placed in the program ahead of when the data are needed. The memory access path is thus uniform and predictable, which simplifies interlocks.

As the second example, some architectures use a common set of registers for addressing and execution, while others use separate sets of registers for the two tasks. Perhaps the most notable examples of the latter are once again the CDC and Cray Research architectures. It is also true to a lesser extent in the IBM 360/370 architecture, and it is pointed out in [1] that separate floating point registers in the IBM 360/370 architecture leads to increased parallelism in high-performance implementations.

The class of architectures to be studied in this paper, decoupled architectures, are specifically designed to separate the memory access and execution tasks as much as possible, farther than the two methods described above. One main element of decoupled architectures is the use of two (or possibly more) instruction streams which may either be fetched from memory separately, or which may be split from a single stream early in instruction processing. This increases the maximum instruction issue bandwidth to two instructions per clock period rather than one instruction per clock period in conventional pipelined processors [9].

The second main element of decoupled architectures is communication through architectural queues. This tends to maximize independence of the access and execution instructions, and allows them to "slip" with respect to one another. The access instruction stream typically moves ahead of the execution stream, and when a program is looping, it often precedes the execution stream by at least one loop iteration. This is, in effect, a form of dynamic instruction scheduling which takes place at runtime.

In contrast, modern pipelined computers rely almost solely on static code scheduling done by the compiler. For example, in the CDC 7600 and CRAY-1 instructions issue in strict program order, and all code scheduling must be static. Static scheduling cannot be as good as dynamic scheduling because it is done with less knowledge of the runtime state of the machine. It does lead to simple pipeline control, however, which permits a very short clock period to be used.

In a decoupled architecture, dynamic scheduling does not come at the expense of additional control complexity [24]. This is because each instruction stream by itself is issued in strict program order with pipeline control that differs very little from a Cray-type implementation. The dynamic scheduling is only between the two streams. This constrains the extent of dynamic scheduling somewhat, but the dynamic scheduling that does take place reduces the most significant instruction issue delays.

Another important characteristic of decoupled architectures is a reduced sensitivity to memory access delays. This results from the ability of the access instructions to run ahead and fetch data in advance of when they are needed. Often the distance the access process can run ahead is only limited by the lengths of the queues. Hence, a slow memory access path can be compensated for by using longer queues. This last characteristic is likely to be important in VLSI systems where communication delay, especially with memory, is a critical factor in determining system performance.

B. Paper Overview

We begin in the next section with a survey of decoupled architecture computers that have been proposed to date. This discussion is intended to highlight the similarities and thus to indicate the generality of the results given in this paper. Section III contains a general description of the specific decoupled architecture that serves as a basis for the simulation studies to follow. Section IV describes the simulation methodology used in the paper. Section V begins with a performance comparison to the CRAY-1, and contains a study of the sensitivity of performance to memory access delays. Section VI studies the effect of queue lengths on performance.

II. A Survey of Decoupled Architectures

A number of decoupled architectures are currently in various stages of development and investigation. Although the design details vary, the main feature common to these architectures is the partitioning of the machine into two or more units, each servicing its own instruction stream and interconnected by architectural queues.

Perhaps first of these, and the only one that is in production, is the MAP series of array processors developed by CSPI, and exemplified by the MAP-200 [6]. In Fig. 1, two major components of the MAP-200 are an "addresser," or APS, and an arithmetic processing unit or APU. Each of these units has its own instruction stream coming from its private program memory. These two programs cooperate during the execution of user tasks. The addresser performs address calculations and places addresses of memory data to be read or written into the read address FIFO (RAF) or write address FIFO (WAF), respectively. The memory transfer controller services the WAF and RAF queues by loading data addressed by the RAF into the APU's input queue, and storing data addressed by the WAF from the APU's output queue. The two instruction streams run at their own speed, with the queues absorbing excess data. The APS typically runs ahead and fetches data in advance of when it is needed by the APU as well as generating store addresses so that they are ready when data become available to be stored. Coordination for branches and memory hazards takes place via flags that are set and interrogated by APS and APU instructions.

For example, when the APS finishes a loop, it sets a flag and enters a "wait" state. When the APU processor is done with its corresponding loop, it clears the flag and allows the APS processor to continue. As pointed out in [6], this method is inefficient because it forces synchronization of the two processors at the end of each loop.

Developed independently and in the context of general-purpose scientific computers is a proposal for decoupled architectures in [17]. A specific decoupled architecture based on the CRAY-1 was introduced as a basis for performance

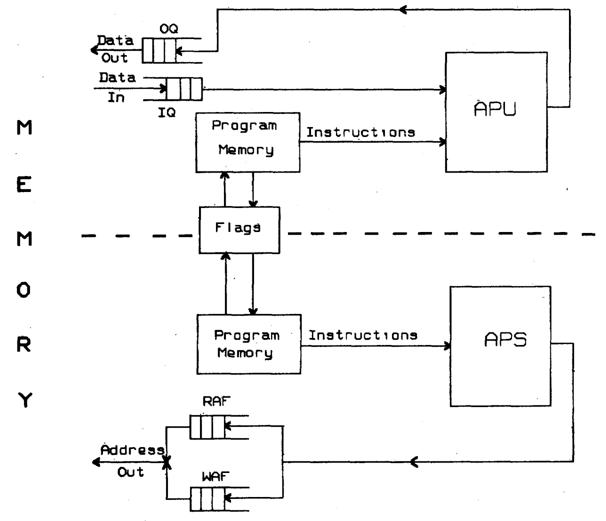


Fig. 1. The CSPI MAP-200 architecture.

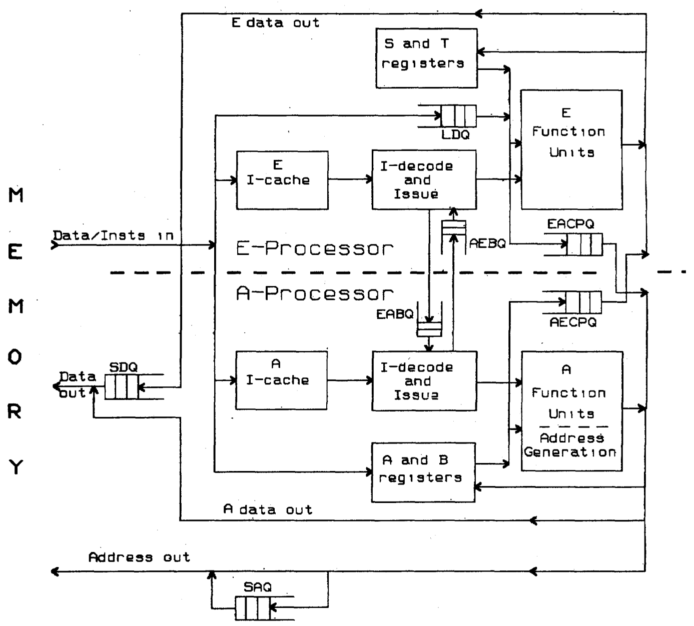


Fig. 2. The decoupled architecture studied in this paper.

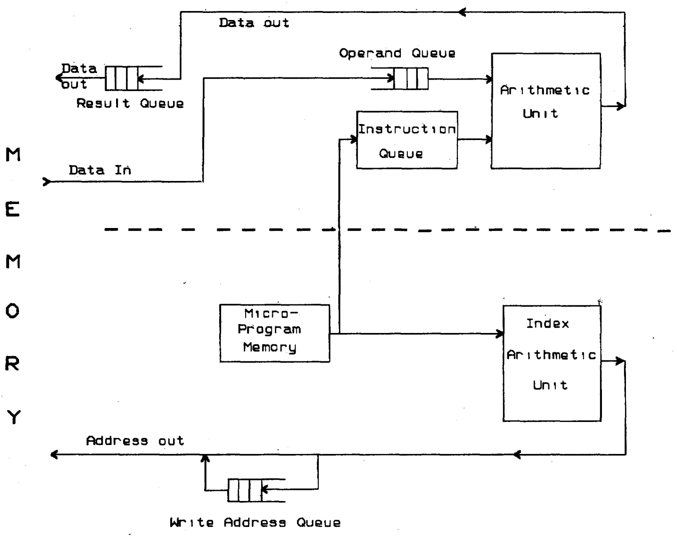


Fig. 3. The synchronous distributed processor (SDP) architecture.

comparisons. A slight variation of this same architecture will be used for the much more detailed simulation studies in this paper. Because this architecture will be described in detail in the next section, we give only a brief description here. Refer to Fig. 2. The architecture has two instruction streams just as the MAP processors have. However, they are stored in main memory and are fetched into instruction caches in the Access and Execution processors. The most significant difference with the MAP processors has to do with coordination of the two instruction streams for branching and resolution of memory hazards. In [17], queues are used to pass branch outcome information. This allows the same flexibility as is afforded by the data queues. Memory hazards are resolved automatically by the memory interface. Both of these features streamline the architecture by avoiding the use of flags which tend to inhibit overlap.

As will be pointed out later in this paper, decoupled architectures have several features that make them particularly appealing for VLSI implementations. One of the major advantages is reduced sensitivity to memory access delays. Consequently, PIPE [18], an architecture descended from [17], is being studied at the University of Wisconsin in the context of VLSI implementations.

The synchronous distributed processor (SDP) [15], is a special-purpose processor designed for signal processing. It is illustrated in Fig. 3. Because of its special-purpose nature, an SDP program is stored in a microprogram memory. SDP contains two major units, the index arithmetic unit (IAU) and the arithmetic unit (AU). The primary task of the IAU is to compute effective addresses and to load and store data. The AU receives instructions from the IAU and performs operations on the operand stream. Communication between the AU and memory is handled through architectural queues. These result in reduced memory access delays because of the decoupling of the IAU and the AU.

A main difference between SDP and the decoupled architectures discussed thus far is that it has a single instruction stream coming from a microprogram memory. The instructions passed to the AU, however, are more like large subroutine calls than elementary operations. Hence, a mode of operation very similar to having dual instruction streams is achieved.

The Fortran oriented machine, or FOM [4], is an architecture proposed at IBM research; refer to Fig. 4. FOM has a single instruction stream that is fetched by an instruction dispatch unit. The instruction stream is split so that instructions are sent to a fixed point unit, a floating point unit, and an exchange/convert unit. The load/store unit gets its instructions indirectly through the fixed and floating point units. The instructions are queued in front of the units, and splitting can be done at a rate higher than one instruction per clock period. The load/store unit serves as an access processor, and the fixed and floating point units serve as execution processors. The exchange/convert unit serves the same function as the copy queues used in the architecture studied in this paper, but it has the additional capability of converting from fixed to

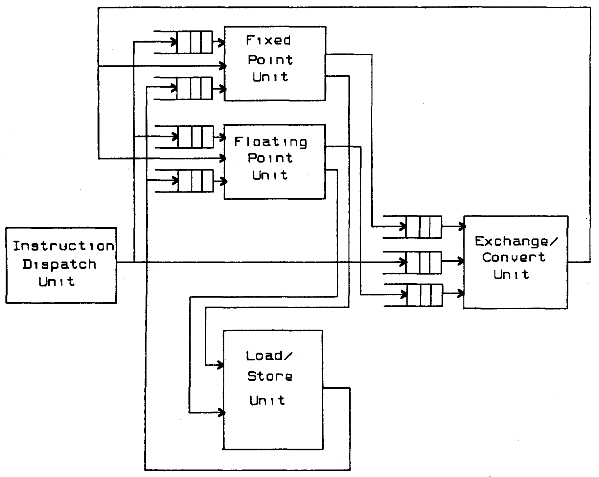


Fig. 4. The Fortran oriented machine (FOM) architecture.

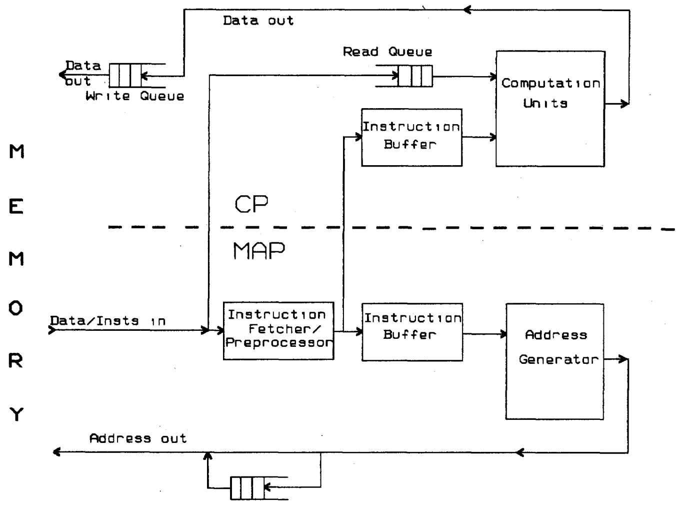


Fig. 5. The SMA architecture.

floating point. All communication among the units is via architectural queues, so that the instruction streams seen by each unit have the ability to "slip" with respect to one another.

SMA [13] is an architecture based on earlier theoretical work by Hammerstrom and Davidson [10]. SMA has a memory access processor (MAP), and a computation processor (CP); refer to Fig. 5. The communication between the two units is via architectural queues. The MAP fetches a single instruction stream for both processors, but blocks of instructions are passed to the CP. Hence, SMA bears some similarity to SDP. Both the MAP and CP can capture loops so that each effectively has its own instruction stream when loops are being executed. As with the other decoupled architectures, the access stream can run ahead of the execution stream. The MAP in the SMA is more elaborate than those in the other decoupled architectures and several mechanisms are included for efficient accessing of data structures. Control information, e.g., a branch terminating a loop, is passed from the MAP to the CP via a data queue.

The above survey shows that several decoupled architectures have been proposed, all with the same goal of partitioning data access and functional operations on the data. They all use architectural queues for communication between the two major processors, and either use multiple instruction streams directly or are capable of what is effectively a multiple stream mode of operation. While most of the decoupled architectures have been independently proposed, their similarities are much more significant than their differences. Hence, the remainder of this paper studies one decoupled architecture in depth, but it is felt that the results, at least qualitatively, extend to the others.

III. A Pipelined Decoupled Architecture

A. Architecture Overview

The decoupled architecture to be studied in this paper is closely related to the CRAY-1 scalar architecture, but uses two instruction streams. This is very natural because the CRAY-1 has its registers divided into 24-bit addressing registers (A and B) and 64-bit floating point registers (S and T). The CRAY-1 has a set of instructions for each type of register. In our architecture, these become the access, or A-processor instructions and execution, or E-processor instructions.

In Fig. 2, each of the processors has its own set of registers (we use A, B, S, and T registers to match their CRAY-1 counterparts). The processors exchange data and control information through a set of architectural queues. Data to be used by the E-processor are loaded by instructions issued by the A-processor, and are placed in the load data queue (LDQ). The E-processor consumes the data from the LDQ and executes instructions that use the data. The memory system handles all load requests in program sequence and returns them to the LDQ in the same order.

The A-processor also generates store addresses for the E-processor, and they are saved in the store address queue (SAQ) where they await data generated by the E-processor. When the E-processor has computed results needing to be stored to memory, it places the data in the store data queue (SDQ). When both the SDQ and the SAQ are nonempty, the operand at the head of the SDQ is matched with the address at the head of the SAQ, and the store address and data are sent to memory. Both the SDQ and SAQ follow a rigid FIFO discipline so that data are paired with the correct address. Allowing stores to wait until data are ready while letting subsequent loads pass through a memory does introduce the possibility of memory hazards. To resolve them, the architecture compares all load addresses to queued store addresses; if there is a match, the load must wait until the store is sent to memory, and the match goes away.

The LDQ is referenced by the E-processor as if it is a register. That is, the LDQ may be specified in any instruction source operand field instead of a register. One register designator is therefore reserved for this purpose, say $S7$. However, in this paper we will explicitly refer to it as LDQ for clarity. In order to access the LDQ as efficiently as possible, the LDQ may be used to supply both source operands for an instruction. In this case, the first and second LDQ elements are used, and issue is blocked if the LDQ contains fewer than two elements.

The SDQ is also referred to as a register, and can be used for an instruction's result instead of a register. The same reserved register designator (*S*7) can be used for the SDQ and LDQ without ambiguity. That is, when it specifies a source operand the LDQ is referenced; when it specifies a destination operand, the SDQ is referenced.

In some programs it is necessary to pass data directly from one of the processors to the other. Two copy queues are used for this purpose. These are the *A* to *E* copy queue (AECPQ) and the *E* to *A* copy queue (EACPQ). In order to effect a copy, there needs to be an instruction for each processor. For example, to copy from an *A* register in the A-processor to an *S* register in the E-processor, an A-processor instruction copies from the *A* register to the AECPQ. The corresponding E-processor instruction copies from the AECPQ to the *S* register. The copy queues are not absolutely essential because copies could take place through main memory. Nevertheless, in the few cases where they are used, especially for an *E* to *A* copy, they can have a significant effect on performance.

For synchronizing branch instructions, two branch queues (EABQ and AEBQ) are used. Each processor has its own set of conditional branch instructions. The processor that has the data to determine the outcome of a branch will encounter a conditional branch instruction, execute it, and place the branch outcome (taken or not taken) on the tail of the branch queue. Its coprocessor will encounter a "branch from queue" (BFQ) instruction that is the mate of the conditional branch instruction and will read the head of the branch queue to determine to take its branch. The A-processor usually determines the results of conditional branches; for example, it performs loop counting functions. Hence, it typically leads the E-processor, and continues to run ahead even after a conditional branch has been encountered. One bit is sufficient to indicate if a branch is taken or not, and thus the branch queues are relatively inexpensive to implement because each stage is only one-bit wide.

In the simulations to follow, we assume that the decoupled architecture is implemented with pipelining to the same degree as the CRAY-1. That is, all the pipelines are the same length as in the CRAY-1, and the clock period is the same. For this to be possible, the control logic, instruction issue logic in particular, must be no more complicated. Consequently, we assume all instructions issue in order. The checks for queues empty or full can be integrated into the register "busy bit" scheme used in the CRAY-1. An empty LDQ as a source operand and a full SAQ or SDQ as a destination operand cause instruction issue to block just as a busy register does.

It is also assumed that accessing a queue takes no longer than accessing a register. This can be done if the queues are held in register files used as circular buffers, and are addressed with head and tail counters rather than register designators coming from instructions. A more complete discussion of this issue can be found in [19]. Nevertheless, results given later will show that performance is not degraded in any significant way if the memory access path is increased by a clock period or two.

```
        q = 0.0
        do 3 k = 1, 1000
    3   q = q + z(k) * x(k)
```

(a)

```
      A1  ←  -1750      . initialize loop counter
      A3  ←  1          . index increment
      A2  ←  1          . initialize index
      S6  ←  0          . q = 0.0
3:    S3  ←  z, A2      . load z(k)
      S2  ←  x, A2      . load x(k)
      A1  ←  A1 + 1     . increment loop counter
      A0  ←  A1         . transfer loop counter to A0
      A2  ←  A2 + A3    . increment index
      S1  ←  S2 *f S3   . z(k) * x(k)
      S6  ←  S6 +f S1   . add above to q
      JAM    3          . branch if A0 < 0
      q   ←  S6         · store q
```

(b)

```
        Access Processor           Execute Processor
        A1  ← -1750
        A3  ← 1
        A2  ← 1                        S6  ← 0
3a:     LDQ ← z, A2            3e:     S1  ← LDQ *f LDQ
        LDQ ← x, A2                    S6  ← S6 +f S1
        A1  ← A1 + 1                   BFQ   3e
        A0  ← A1                       SDQ ← S6
        A2  ← A2 + A3
        JAM   3a
        SAQ ← q
```

(c)

Fig. 6. Lawrence Livermore Loop 3. (a) FORTRAN code for Lawrence Livermore Loop3 (inner product). (b) Assembly code for CRAY-1S. (c) Assembly code for decoupled architecture.

B. Program Example

Fig. 6 illustrates a program example for the decoupled architecture. The Fortran source for Lawrence Livermore Loop 3 is in Fig. 6(a). Fig. 6(b) shows a CRAY-1 compilation; we have inserted arrows to improve readability. The decoupled architecture compilation is in Fig. 6(c).

IV. Simulation Methodology

A. Simulator Structure

A number of simulation tools have been developed for this study [12]. They are written in *C* and run on a VAX®-11 under the UNIX® operating system. The tools are capable of simulating program execution, producing instruction and data traces, and providing extensive performance-related data by displaying resource usage and conflicts on a clock-by-clock basis. The tools are primarily table driven so that experiments can be carried out by varying numerous timing parameters. The basic approach we use is to divide the simulation task into architecture and implementation phases. Thus, CRAY-1 simulation is done by two simulators, one used to simulate function and the other to simulate performance.

The table-driven functional simulator takes CRAY-1 machine language programs and simulates them to generate trace files. These trace files contain all the information necessary to

® VAX is a trademark of Digital Equipment Corporation.

® UNIX is a trademark of AT&T Bell Laboratories.

determine processor performance. The performance simulator takes a trace file and a table containing timing information and produces either a detailed timing history for each instruction and/or a timing summary for the entire program. The table that drives the simulation contains interlock conditions that must be checked at issue time as well as the times required by the major functions in the CRAY-1.

In addition to the CRAY-1 simulators, two decoupled simulators have been constructed. The relationship of the decoupled functional simulator and the decoupled performance simulator is similar to that of the CRAY-1 simulators.

B. Simulation Workload

Performance is measured by simulating the first 14 Lawrence Livermore Loops [11]. These are excerpts from large Fortran programs that have been judged to provide a good measure of large scale computer performance. The loops were first compiled using the CRAY Fortran (CFT) optimizing compiler [8], version 109h. Since we are interested in scalar performance, the CFT compiler was run with the "vectorizer" turned off so that no vector instructions were produced. When the vectorizer is on, half of the 14 loops contain a substantial amount of vector code, and half remain scalar. Although all the loops were compiled with scalar instructions, we will refer to the ones that can be vectorized by the CFT compiler as "vectorizable" and to the others as "nonvectorizable."

For some of our simulation results, we separate the vectorizable and the nonvectorizable loops. This is because vector architectures and decoupled architectures are in a sense orthogonal (one could use both in the same computer). Separating the vector and scalar results allows one to see the improvement in both the context of a vector supercomputer and a scientific superminicomputer without vectors. In general, a decoupled architecture performs best on loops that have characteristics that make them vectorizable. If a loop can be vectorized then there is a high degree of independence among the loop iterations. Also, the data access mechanism is independent of the execution mechanism.

No further optimization or code scheduling was done in addition to that of the CFT compiler. A minor change to the code as generated by the CFT compiler was necessary, however. As a practical matter in a decoupled architecture, transfer of data or control information from the E to the A-processor will force the A-processor to wait for the E-processor. This at least temporarily halts prefetching by the A-processor and reduces performance. Such a forced delay of the A-processor cannot always be eliminated. Loop counting, however, is one case where it can be avoided because incrementing a loop counter can be entirely done by the A-processor. This is, in fact, one of the stated functions of the A registers in [7]. For rather obscure reasons, CFT generates code so that loop counters are held in A registers, but are copied into S registers to be incremented before being copied back into A registers for loop testing. Therefore, the code generated by the CFT has been manually changed so that all the loop counting operations are done in A registers. Both the CRAY-1 and the decoupled architecture execute code that has been modified in this way.

The workload for the decoupled architecture was generated by manually splitting the object code for each kernel into two instruction streams. Register reallocation was sometimes necessary, because register $S7$ has special meaning in the decoupled machine (see Section III-A above). A new instruction, BFQ (branch from queue) was introduced by using a CRAY-1 opcode that never appears in the Lawrence Livermore Loops. The processor that has the information to determine the result of a branch was given the conditional branch instruction from the original instruction stream; a BFQ was inserted in its coprocessor code. Load and store instructions are placed so that they always follow the queue discipline.

Performance has been calculated in millions of floating point operations per s, or megaflops. The number of floating point operations in each of the loops was determined by actually counting the number of floating point operations generated by CFT in each of the kernels, not by inspecting the Fortran code. We have assumed a 12.5 ns clock period, identical to that of the CRAY-1. Memory bank conflicts are not considered.

V. Performance Comparison to Conventional Architectures

A. CRAY-1 Comparison

The CRAY-1 instruction set is designed for efficient pipeline processing. Being a register-register architecture, only load and store instructions access memory. The rest of the instructions take operands from registers. The CRAY-1 forces instructions to issue strictly in program order. If an instruction is blocked from issuing due to a conflict, all instructions following it in the instruction stream are also blocked, even if they have no conflicts.

In the CRAY-1, only one parcel (16 bits) is read from the instruction buffers per clock period. Many instructions are only one parcel long, although the load, store, and branch instructions are two. This means that the maximum instruction issue rate of the CRAY-1 is one parcel per clock period. To keep comparisons fair, we assume that each of the instruction streams in the pipelined decoupled architecture we are studying have the same constraint. Hence, although we often talk about the limitation of one instruction per clock period, it is actually one parcel per clock period in this paper.

Because the maximum rate of instruction issue is one of the topics we are studying, we define the *issue bound* to be the maximum CRAY-1 performance given the most favorable data dependence conditions. That is, it assumes that the only limitation is set by the maximum rate at which instructions can be issued. The issue bound assumes that a one parcel instructions consumes one clock period, a two parcel instruction consumes two clock periods, and a taken branch consumes five clock periods. The loops that are vectorizable are marked with an "*."

For the study in this section we assume unbounded queues for the decoupled architecture. We show in the next section, however, that in a decoupled architecture short queues are sufficient to achieve performance very close to that given here. For the functional units, we have assumed similar times as in the CRAY-1 (e.g., floating point multiply takes seven clock periods, floating point add six, etc.). We have also assumed that one parcel is issued per clock period, again in accordance with the CRAY-1. Obviously, in a decoupled machine two instruction streams are processed in parallel, and therefore two parcels may be issued simultaneously; as already pointed out, this is one of the advantages of a decoupled architecture.

Table I shows the results of our initial simulations. Overall, the decoupled computer gives a speedup of 1.56 as compared to the scalar CRAY-1. Speedups for the vectorizable and nonvectorizable loops are 1.72 and 1.40, respectively. This confirms our earlier observation that the vectorizable loops should provide better performance because of the independence of the access and execution tasks. When compared to the issue bound, we see that the issue bound can be exceeded by some of the loops, and these tend to be vectorizable ones.

B. Sensitivity to Memory Path Length

One of the arguments for a decoupled architecture is a reduced performance sensitivity to memory access delays. In this section, we given simulation results that support this claim. Figs. 7, 8, and 9 summarize a series of performance simulations made with memory access time varying from 5 clock periods to 32 clock periods in increments of 3. For reference, the CRAY-1 has a memory access time of 11 clock periods for scalar memory references. Shown on the graphs are decoupled computer performance and CRAY-1 performance. For reference, the instruction issue bound is also shown. Fig. 7(a) graphs performance over all 14 loops. Fig. 7(b) shows the speedups for the decoupled computer as compared to the original CRAY-1. Figs. 8 and 9 give separate performance results for the vectorizable and the nonvectorizable loops, respectively.

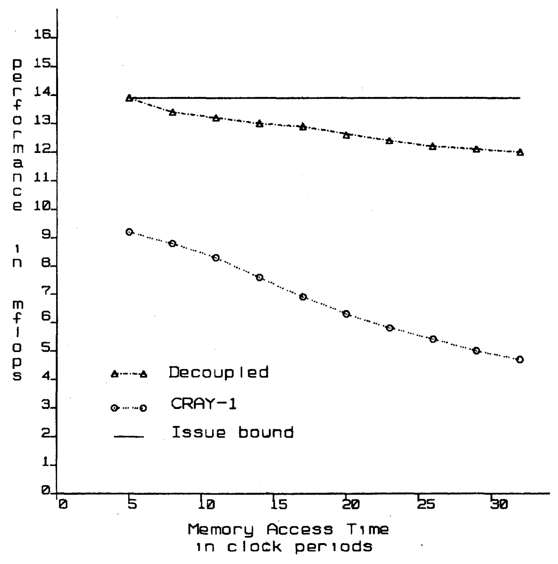
Fig. 8 shows that the decoupled architecture is better than the issue bound on the vectorizable loops. Fig. 9 shows that for the nonvectorizable loops it falls below the issue bound, so that on average, as shown in Fig. 7, the decoupled architecture performs under the issue bound.

Fig. 7(b) shows that the speedup of the decoupled system relative to the CRAY-1 increases almost linearly with memory path length. Thus the performance improvement for 5 clock periods access time is 1.48 and it grows to a 2.54 speedup when the access time is 32 clock periods. Fig 8(b) shows a considerable speedup for the vectorizable loops, as expected. Fig. 9(b) shows speedup for nonvectorizable loops that is not as great as for the vectorizable loops, but is significant nevertheless.

As expected, the performance drops with increasing memory path length. However, as Fig. 7 shows, the overall performance penalty due to a long memory path is far lower for the decoupled architecture than for the CRAY-1. As expected, decoupled performance is higher for the vectorizable loops. For the vectorizable loops, performance is virtually

TABLE I
PERFORMANCE COMPARISON OF A DECOUPLED ARCHITECTURE AND THE CRAY-1 PERFORMANCE IS MEASURED IN MEGAFLOPS

Loop	Issue Bound	Performance CRAY-1	Performance Decoupled	Speedup
1*	15.98	7.79	15.92	2.04
2*	14.28	11.98	22.36	1.87
3*	11.42	6.15	11.30	1.84
4	8.31	4.22	7.85	1.86
5	13.72	7.74	9.98	1.29
6	13.72	7.49	10.19	1.36
7*	27.19	15.74	22.73	1.44
8	20.06	14.14	25.05	1.77
9*	20.53	14.09	23.26	1.65
10*	10.72	7.11	11.36	1.60
11	5.33	2.96	3.79	1.28
12*	5.33	2.96	4.65	1.57
13	9.44	5.98	6.49	1.09
14	14.56	8.03	9.44	1.17
Mean	13.61	8.31	13.17	1.56



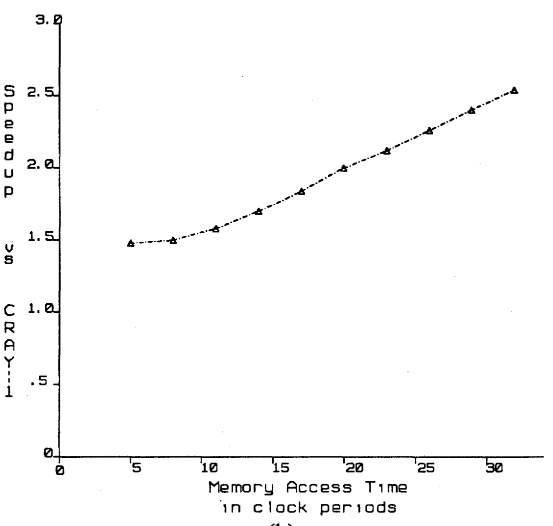


Fig. 7. Performance and speedup for all 14 Lawrence Livermore Loops. (a) Performance. (b) Speedup.

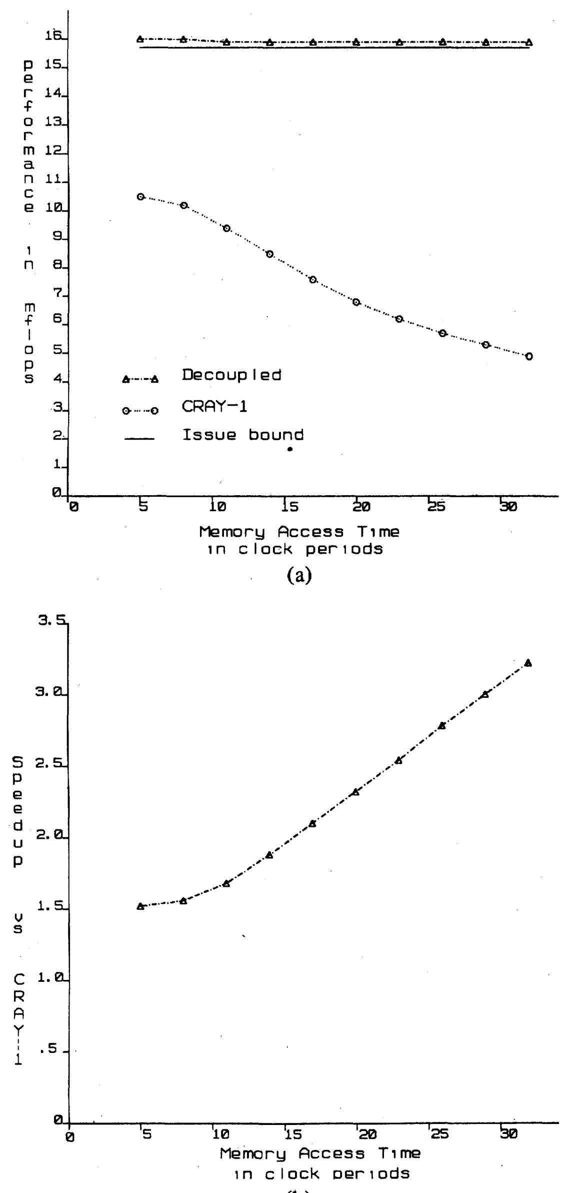


Fig. 8. Performance and speedup for the seven vectorizable loops. (a) Performance. (b) Speedup.

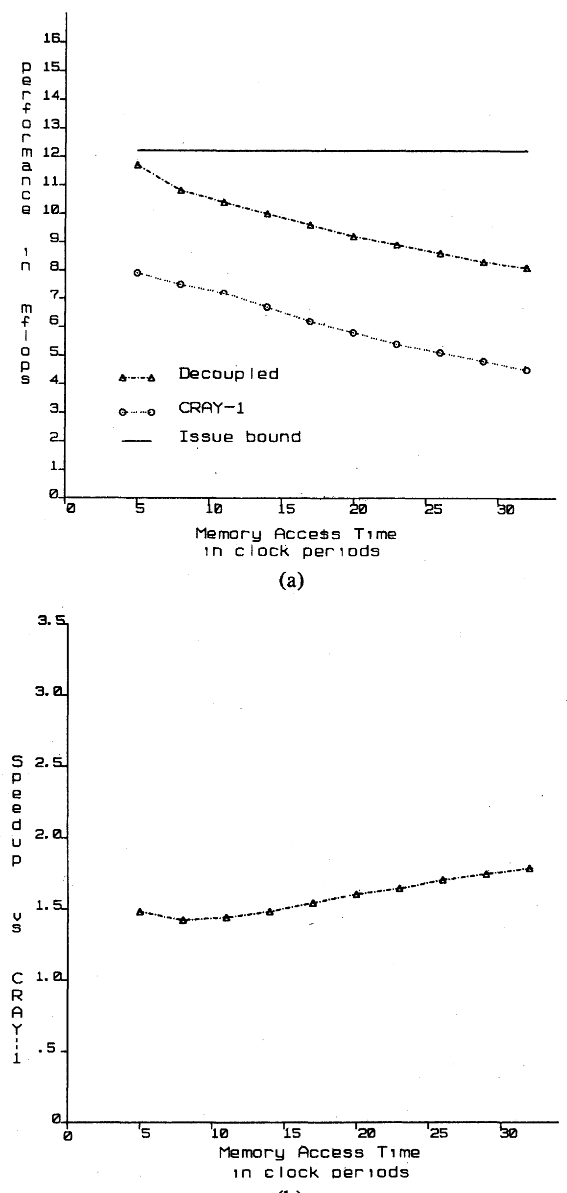


Fig. 9. Performance and speedup for the seven nonvectorizable loops. (a) Performance. (b) Speedup.

independent of the memory access delay over the range studied. For the nonvectorizable loops there is increased sensitivity to memory access delay, but the overall performance remains better than with the CRAY-1 (as shown by the speedup curve).

As an interesting aside, we see that reducing the CRAY-1 memory access time to five clock periods results in a performance improvement of only 10.23 percent. This indicates the kind of improvement that could be achieved if a data cache were used (assuming a 100 percent hit rate).

VI. Simulation Study of Queue Lengths

An important design parameter of a decoupled architecture is the depth of the various architectural queues. We now consider the importance of queue lengths as a function of memory access path length. For the purpose of the simulation, most of the queues that appear in the decoupled architecture described in Section II have been grouped in pairs, according to their function. The only exception is the LDQ. Thus, four sets of experiments were run varying the queue lengths of each of the following groupings:

AEBQ-EABQ
AECPQ-EACPQ
SDQ-SAQ
LDQ.

When the length of a particular queue pair (or the LDQ) is varied, the other queues are assumed to be of unbounded length.

The above partitioning was done for obvious reasons. For example, an address in the SAQ is matched with an operand in the SDQ when sent to memory, and having different length store queues is probably not justified. One could perhaps find some justification for making the branch queues different

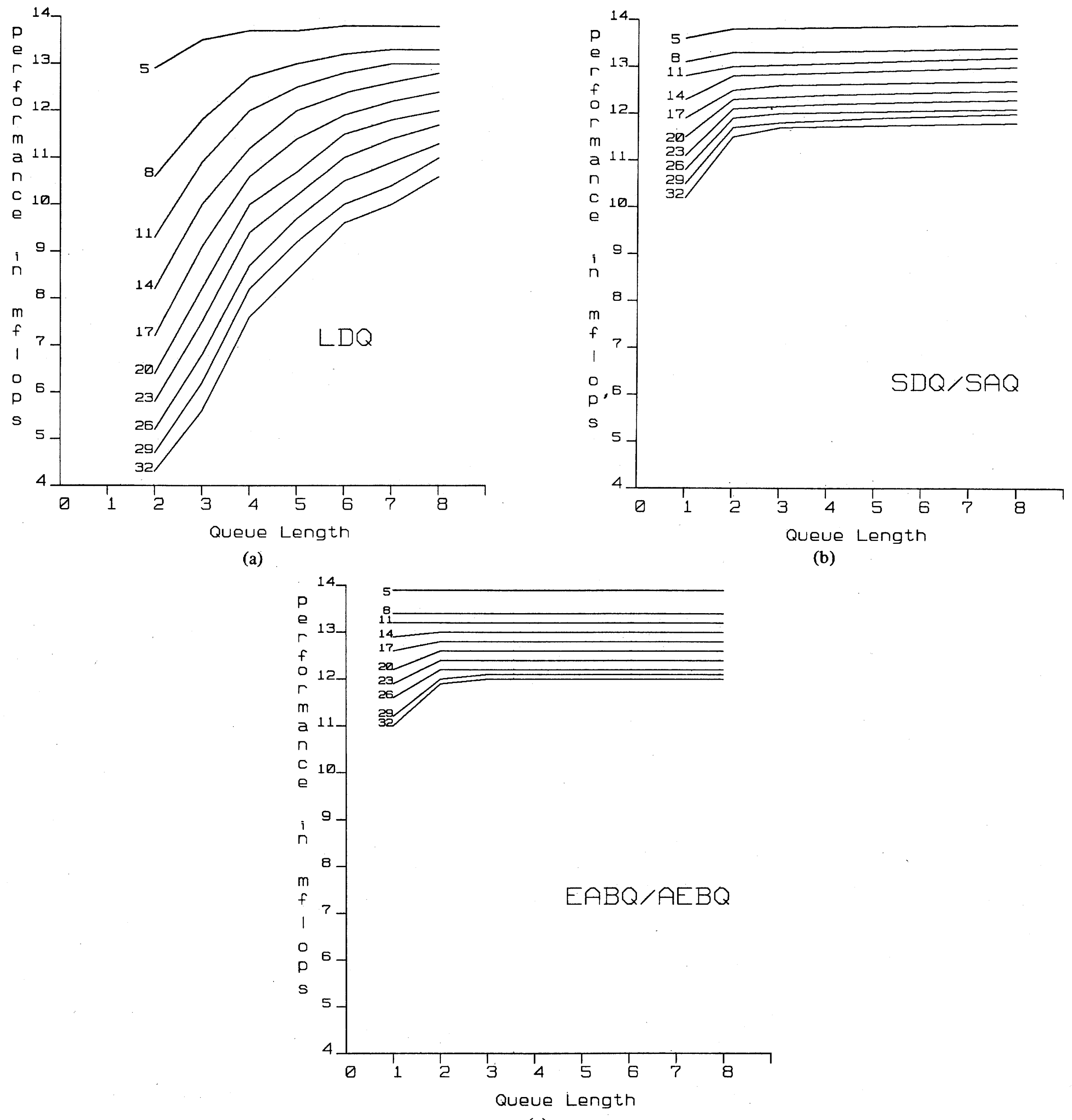


Fig. 10. Performance versus queue lengths for various memory access times (in clock periods).

lengths because one would expect most branches to be determined by the A-processor. Therefore, the AEBQ might be more active than the EABQ.

Notice that the initial length of most queues is one, with the exception of the LDQ that begins at two. Some instructions, e.g., "$S1$ <− LDQ + LDQ," may require two operands from the LDQ. Using a length of one queue would result in deadlock.

Fig. 10 shows performance as a function of queue length for various memory path lengths. For copy queues, no performance improvement has been observed for queue depth greater than one, independent of the memory path length. The branch queues behave similarly for memory path lengths up to 11 clock periods. For longer memory paths there is an advantage in increasing the number of stages in the queue. This appears to indicate that in some of the loops the A-processor runs more than one iteration ahead of the E-processor when the memory path exceeds 11 clock periods. And when the memory access time becomes 29 clock periods, the A-processor is running as much as three loop iterations ahead. For a few of the kernels this is a significant advantage. However, the mean performance does not gain more than 9.2 percent even for the longest memory path simulated (32).

For store queues, performance becomes increasingly dependent on the availability of a second queue stage as the memory path gets longer. No significant speedup has been observed for deeper store queues.

The simulation results for the load queue are the most interesting. For a short memory path (five clock periods), two stages are sufficient. However, as the memory access time gets longer, performance becomes critically dependent on the depth of the LDQ.

For a given memory path length one can optimize the various queue depths. For example, with memory access time 11 clock periods, as in the CRAY-1, one might choose the following queue depths:

AEBQ–EABQ:	1
AECPQ–EACPQ:	1
SDQ–SAQ:	2
LDQ:	4.

The above queue lengths are sufficient to achieve 90 pecent of the maximum performance. Extending the LDQ to length 8 leads to 98 percent of the maximum performance. Such short queues point to fairly simple and inexpensive implementations of decoupled architectures.

VII. Summary and Conclusions

We have presented simulation results for a decoupled access/execute architecture and compared them to the CRAY-1. Overall, there is over a 50 percent improvement over the scalar CRAY-1, assuming the same memory path as in the CRAY-1. As the length of the memory path increases, the performance improvement becomes greater. If the memory access time grows to 32 clock periods, the improvement is over two-and-one-half times.

The decoupled architecture also allows some of the loops to surpass the instruction issue bound, the maximum possible performance with a conventional pipelined implementation. This tends to support the argument that a decoupled architecture provides a means for avoiding the one instruction issue per clock period bottleneck.

The performance simulations also show that loops that can be executed with vector instructions show the greatest performance improvements with the decoupled machine. Nevertheless, the improvements for the non-vectorizable loops are still significant, and a decoupled architecture may still be justified even if the architecture also supports vector instructions.

We have also presented a simulation study of architectural queue lengths. It is shown that as the memory path increases performance becomes critically dependent on the length of the branch, store and load queues. The length of the load queue has the greatest impact on performance. No performance improvement has been observed for copy queues longer than one stage, even for very long memory paths (up to 32 clock periods). An important result of our study is that short queues are sufficient to achieve performance close to the maximum available with unbounded queues.

References

[1] D. W. Anderson, F. J. Sparacio, and F. M. Tomasulo, "The IBM System/360 model 91: Machine philosophy and instruction-handling," *IBM J. Res. Develop.*, vol. 11, pp. 8–24, Jan. 1967.

[2] C. N. Arnold, "Performance evaluation of three automatic vectorizer packages," in *Proc. Int. Conf. Parallel Processing*, 1982, pp. 235–242.

[3] P. Bonseigneur, "Description of the 7600 computer system," *Computer Group News*, pp. 11–15, May 1969.

[4] W. C. Brantley and Joseph Weiss, "Organization and architecture tradeoffs in FOM," presented at IEEE Int. Workshop Comput. Syst. Organization. New Orleans, LA, Mar. 1983.

[5] *CDC CYBER 200 Model 205 Computer System Hardware Reference Manual.* Control Data Corp., Arden Hills, MN, 1981.

[6] E. U. Cohler and J. E. Storer, "Functionally parallel architectures for array processors," *Computer*, vol. 14, pp. 28–36, Sept. 1981.

[7] *CRAY-1 Computer Systems, Hardware Reference Manual.* Cray Research, Inc., Chippewa Falls, WI, 1979.

[8] *CRAY-1 Computer Systems, Fortran (CFT) Reference Manual.* Cray Research, Inc., Chippewa Falls, WI, 1982.

[9] M. J. Flynn, "Very high-speed computing systems," *Proc. IEEE*, vol. 54, pp. 1901–1909, Dec. 1966.

[10] D. W. Hammerstrom and E. S. Davidson, "Information content of CPU memory referencing behavior," in *Proc. 4th Ann. Symp. Comput. Architect.*, Mar. 1977, pp. 184–192.

[11] F. H. McMahon, *Fortran CPU Performance Analysis.* Lawrence Livermore Laboratories, 1972.

[12] N. Pang and J. E. Smith, "CRAY-1 simulation tools," Dep. Elec. Comput. Eng., Univ. Wisconsin, Madison, Tech. Rep. ECE-83-11, Dec. 1983.

[13] A. R. Pleszkun, "A structured memory access architecture," Coord. Sci. Lab., Univ. Illinois, Urbana, Comput. Syst. Group Rep. CSG-10, Oct. 1982.

[14] R. M. Russell, "The CRAY-1 Computer System," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 63–72, Jan. 1978.

[15] R. R. Shivley, "Architecture of a programmable digital signal processor," *IEEE Trans. Comput.*, vol. C-31, Jan. 1982.

[16] B. J. Smith, "A pipelined, shared resource MIMD computer," in *Proc. 1978 Int. Conf. Parallel Processing*, Aug. 1978, pp. 6–8.

[17] J. E. Smith, "Decoupled access/execute computer architectures," *Ninth Ann. Symp. Comput. Architect.*, Apr. 1982, pp. 112–119.

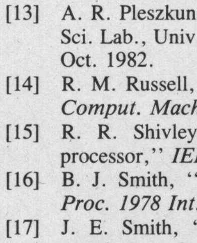
[18] J. E. Smith, A. R. Pleszkun, R. H. Katz, and J. R. Goodman, "PIPE: A high-performance VLSI architecture," presented at IEEE Int. Workshop Comput. Syst. Organization, Mar. 1983.

[19] J. E. Smith, "Decoupled access/execute computer architectures," *ACM Trans. Comput. Syst.*, vol. 2, pp. 289–308, Nov. 1984.

[20] J. E. Thornton, *Design of a Computer—The Control Data 6600.* Glenview, IL: Scott, Foresman, 1970.

[21] S. Weiss and J. E. Smith, "Instruction issue methods in pipelined supercomputers," presented at 11th Ann. Int. Symp. Comput. Architect., June 1984.

[22] J. Worlton, "Understanding supercomputer benchmarks," *Datamation*, pp. 121–130, Sept. 1984.

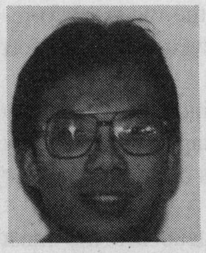


James E. Smith received the BS, MS, and Ph.D. degrees from the University of Illinois, Urbana, in 1972, 1974, and 1976, respectively.

Since 1976, he has been on the faculty of the University of Wisconsin, Madison, where he is an Associate Professor in the Department of Electrical and Computer Engineering. He spent the Summer of 1978 with the IBM T. J. Watson Research Center, Yorktown Heights, NY, and from September 1979 to July 1981 he worked for the Control Data Corporation, Arden Hills, MN. While at CDC he participated in the design of the CYBER 180/990. He is currently on leave from the University of Wisconsin, working for the Astronautics Corporation of America on the design of a large-scale scientific computer system.

Shlomo Weiss received the B.S. degree from the Technion—Israel Institute of Technology, Haifa, Israel, in 1973 and the Ph.D. degree from the University of Wisconsin, Madison, in 1984.

He is currently working on the VLSI CAD Program on design data management, for the Microelectronics and Computer Technology Corporation, Austin, TX. Previously, he has held positions with the Israel Aircraft Industry, Koor Systems, Ltd., and Motorola Israel Ltd. He spent the summer of 1982 with Xerox PARC, Palo Alto, CA. His research interests include computer-aided design for VLSI systems and high-performance computing.



Nicholas Y. Pang (S'80–M'83) received the B.S. degree (summa cum laude) in electrical engineering from Arizona State University, Tempe, in 1981, and the M.S. degree in electrical and computer engineering from the University of Wisconsin, Madison, in 1983.

From 1982 to 1983 he was a Research Assistant for the Department of Electrical and Computer Engineering, University of Wisconsin, Madison, working on simulation models for the CRAY-1 and Decoupled architectures. He is currently working for Amdahl Corporation, Sunnyvale, CA, on the design of the CPU Storage Unit of a high-performance computer. His present interests lie in the area of vector processors and multiprocessor systems.

Mr. Pang is a member of the IEEE Computer Society, Phi Kappa Phi, Tau Beta Pi, and Eta Kappa Nu.