VSAM: A SIMULATOR-BASED DEBUGGER AND PERFORMANCE ANALYSIS TOOL FOR SAM

by

George Vodarek
B.Sc., Simon Fraser University, 1981

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in the School
of
Computing Science

Approval

Name:	George Vodarek
Degree:	Master of Science
Title of thesis:	VSAM: A Simulator-based Debugger and Performance Analysis Tool for SAM

Examining Committee:

Dr. J. G. Peters
Chair

Dr. R.F. Hobson
Senior Supervisor

Dr. J. J. Weinkam
Supervisor

Dr. R. Krishnamurti
External Examiner

Date Approved: July 27, 1995

Abstract

This thesis describes a *virtual* simulator-based software debugging and performance analysis system (VSAM) for the Structured Architecture Machine (SAM). SAM is a distributed-function multiprocessor computer designed to execute APL efficiently. The purpose of VSAM is to help researchers investigate the behavior of the SAM architecture and to support the exploration of alternative designs. Object-oriented techniques are used to represent the hierarchical structure of the hardware thereby facilitating instrumentation and modification of the architecture.

VSAM is implemented in C++ under OS/2 and utilizes multi-tasking extensively. The core of VSAM is a behavioral simulator of SAM. The simulator is a faithful functional model of SAM down to the register/bus component level. A full-featured debugger interface is provided for each processor. The debugger includes novel features for dealing with multiple processors, functional units, and data presentation. VSAM also provides a general instrumentation facility which uses OS/2 pipes to connect sensors embedded in the simulator to display windows.

The simulator design is discussed in detail and presented in the context of alternative simulation techniques and other microprocessor simulators. The use of VSAM is demonstrated on SAM benchmarks and the results are discussed.

Acknowledgments

I would like to thank Dr. Rick Hobson for his generous guidance, encouragement, and support during this work. I would also like to thank my wife Laurie Cooper for her support and love.

This work was funded by a Simon Fraser University Graduate Fellowship, the Simon Fraser University Center for System Science, Micronet Center of Excellence, and NSERC.

CONTENTS

List of Tables

List of Figures

1. Introduction

As computer architectures become more complex in order to achieve further gains in performance, a thorough understanding of the behavior of an architecture under working conditions is a prerequisite for further improvement. Software-based architecture simulation is an efficient and effective approach to analyzing and measuring the performance of existing and proposed systems. A detailed architecture simulator can be used to evaluate the feasibility of an architecture, to predict its performance, and evaluate the usability of the architecture from the point of view of the software that will need to be written for it.

While software based architecture simulators are easier to build than hardware, they present their own set of design challenges. The basic challenge is the task of capturing the functionality of the new system in software in an efficient and manageable way. The complexity and parallel nature of modern computer architectures makes this a significant exercise in software engineering. Beyond basic correctness, the simulator must be flexible enough to allow changes as the design evolves. The simulator representation must be sufficiently similar in structure to the hardware design to facilitate this. The simulator must be designed with measurement in mind, since one of its primary purposes is performance evaluation of the architecture. Structural similarity to the hardware will help greatly in this area, since the same types of events and objects that occur in the hardware will exist in the simulator. Ideally, the simulator must provide an array of performance analysis tools with which the designer can monitor the system. Finally, since software will need to be written and debugged for the new system, the simulator must provide an interface to software development facilities and to a debugger.

This thesis describes the design, implementation, and use of VSAM, a *virtual* software simulator for the Structured Architecture Machine (SAM) described in [HGT86]. SAM is a distributed-function multiprocessor computer designed to execute APL programs efficiently. A working prototype of SAM, called SAM-1, has been built and is described in [HHS92]. A rudimentary APL interpreter, named SAM APL, has been implemented for SAM, (see [Hos87]), and has demonstrated the capabilities of the architecture. Unfortunately, the prototype hardware is not an ideal performance analysis platform because it is difficult to instrument for detailed measurements and difficult to modify in order to test new ideas. The purpose of VSAM is to overcome these difficulties and to become an architectural workbench for further study of the Structured Architecture Machine and design of future machines.

VSAM is implemented in C++ under OS/2. Object-oriented techniques are utilized to represent the modular structure of the hardware components. Multiple processes are used to emulate the parallel nature of the hardware. VSAM is a machine code simulator that executes SAM-1 binary images, albeit much slower than SAM-1. The simulator is faithful in the representation of hardware components down to the level of registers and buses, and accurately emulates the movement of data at the microinstruction level. VSAM does not emulate the timing of sub-instruction events. A powerful debugger interface is provided to facilitate code testing and to explore the state of the various parts of the machine.

An integral part of VSAM is an instrumentation methodology for measuring and observing the behavior of SAM during execution of SAM software benchmarks. Instrument probes are embedded in the simulator at strategic points and send data to display tasks that present the results graphically on-line, and optionally save results and traces to files for off-line analysis. Several instruments have been developed including a call/return monitor, an execution profiler, and a processor utilization monitor. The design and implementation of the instruments is described in detail and their use is demonstrated on SAM benchmarks.

The remainder of this chapter examines related work on the design and use of architecture simulators. Chapter 2 is an overview of the SAM architecture and the SAM-1 prototype including a critical look at the software development system and development methodology. Chapter 3 describes the implementation of VSAM in detail. Chapter 4 looks at the performance analysis results of SAM benchmarks and discusses them.

1.1 Related Work

The benefits of simulating a system prior to implementation are described in many references including [BaC84], [Pre92], and [Fer78]. The main benefits are the ease with which a model can be built and modified, and the high degree of behavioral analysis such models enable. These benefits are particularly relevant in the study of parallel computer architectures due to the complexity of these systems and the difficulty of implementation. The importance of a sound evaluation methodology in performance analysis is articulated particularly well in [HeP90] and also in [Fer78]. The basic objectives of simulating a system for performance analysis are to identify the typical work-load of a system and then to observe and measure how well the system handles the work-load.

Two important design decisions in building an architecture simulator are the level of abstractness of the model and the implementation method. The level of abstractness refers to the granularity of the model with respect to the types of objects and events the designer works with. A highly abstract model may consider entire processors as basic building blocks, while a detailed model may be concerned with registers and bit transfers. The level of abstractness decision is based on the purpose of the simulation and the nature of the behavior to be observed. Detailed models provide the greatest flexibility and degree of detail, but are difficult to implement and generally very slow to execute. The implementation method must support the level of abstractness desired. A general-purpose programming language offers the greatest flexibility, but the least built-in support. Formal hardware description systems such as VHDL [Nav93] and Verilog [ThM91] are a good choice for detailed models. Discreet-event simulation systems such as SIMULA and GPSS are suitable for more abstract models. Hybrid solutions are also possible. In [Geo93] for example, the author describes the use of *processor libraries* as building blocks for a more abstract model.

Many reports of architecture simulation are available in the literature. Of particular interest are [And94], [Voi94], and [But94] which describe the use of architectural simulators for the PowerPC microprocessor recently released by IBM and Motorola. Two types of simulators are described: a detailed timing simulator and an instruction set simulator. The timing simulator models the internal organization of the microprocessor. It is intended for use by the microprocessor designers and by hardware designers of systems that will employ the microprocessor. It has been packaged as a Verilog module for use as a component of a Verilog simulation. The instruction set simulator executes actual instructions and is intended for software development for the PowerPC prior to availability of the hardware. The simulator has been interfaced to the Free Software Foundation's *gdb* debugger and a complete software development environment. The advantage of this approach is simultaneous delivery of a new microprocessor and software systems for it. A similar set of simulation tools for the Advanced Micro Devices' (AMD) 29K family of RISC processors is described in [TyD93].

An important class of architecture simulators are instruction set simulators described in [MAF91] and [HLT87]. These simulators allow designers to measure the use of various instructions and to evaluate proposed changes. RISC architecture designers made good use of such techniques to focus their designs on the actual work expected of their machines, [Pat85]. An important advantage of instruction set simulators over more detailed simulators, is ease of implementation and execution efficiency.

2. SAM Overview

This chapter describes the SAM architecture, the SAM-1 prototype, SAM APL, and the overall SAM software development environment. The purpose of this chapter is to familiarize the reader with SAM, describe the state of the SAM-1 prototype, and point out some of the difficulties with using it as an analysis platform.

2.1 SAM Architecture

The Structured Architecture Machine, (SAM), is a novel architecture designed to execute APL faster than general purpose architectures. SAM is described in detail in [HGT86] and [HHS92].

The basis for SAM is *A Directly Executable Language* called ADEL, described in [Hob84]. ADEL represents an APL program as a linear form that can be efficiently interpreted by SAM through the use of parallel processors and special purpose hardware. The structure of an ADEL instruction is a format code that identifies the instruction, operand references, and possibly an operator to be applied. The prototypical ADEL instruction is DLR which has the form:

DLR destination-operand left-operand right-operand operator

The APL expression

A←B+C

translates to the ADEL instruction

DLR A B C +

where the meaning of the operands A, B, and C is derived from context. The power of ADEL lies in the fact that the operands may in fact be large arrays! The set of ADEL formats was chosen through experimentation. Most formats are for data manipulation, but several formats are provided for branching and user-defined function invocation. The set of formats may change as the need arises.

The SAM execution model of ADEL partitions the work among three specialized execution units: the Environment Control Unit (ECU), the Program Management Unit (PMU), and the Data Management Unit (DMU). The SAM architecture is shown in Figure 2-1. The ECU is responsible for the user and host interfaces. It translates user input into ADEL which it passes to the PMU for execution. It also receives results from the DMU and displays them.

The PMU manages the execution of programs. It is responsible for storage of defined functions, and maintains the symbol table and the Contour Access Table (CAT) used to resolve operand references. During execution, the PMU looks after branching and function invocation and return. For data manipulation instructions, the PMU resolves operand references into data references, performs compatibility checks on the operands, and passes verified instruction to the DMU for execution.

The DMU executes data operations as instructed by the PMU. It manages the storage of data in memory via the Data Access Table (DAT) which is referenced by the instruction operands. The DMU checks the operands for compatibility and performs the specified action. The DMU receives its instructions from the PMU via a pair of instruction

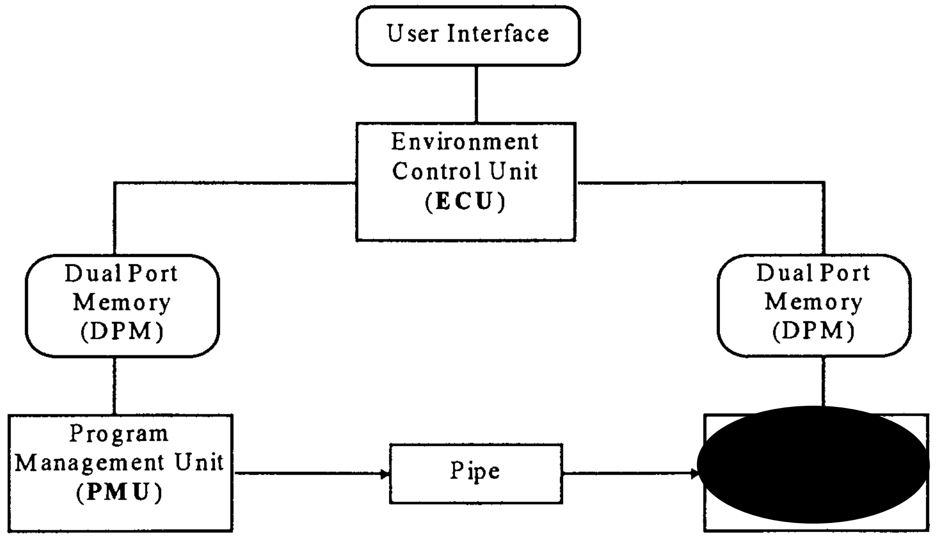


Figure 2-1: SAM Architecture

pipes which allows the PMU and DMU to overlap execution.

2.2 SAM-1 Prototype

SAM-1 is a prototype instantiation of SAM. The structure of SAM-1 is a general purpose host computer, (an IBM PC with DOS), that acts as the ECU, and two embedded custom processors called SAMjr, that are the PMU and DMU. The ECU executes a program that constitutes the user interface to the SAM application and that communicates with the PMU and DMU through a control bus interface and Dual Port Memory (DPM). The PMU and DMU communicate though a custom processor called the SJPM which implements the instruction pipe and operand compatibility checking.

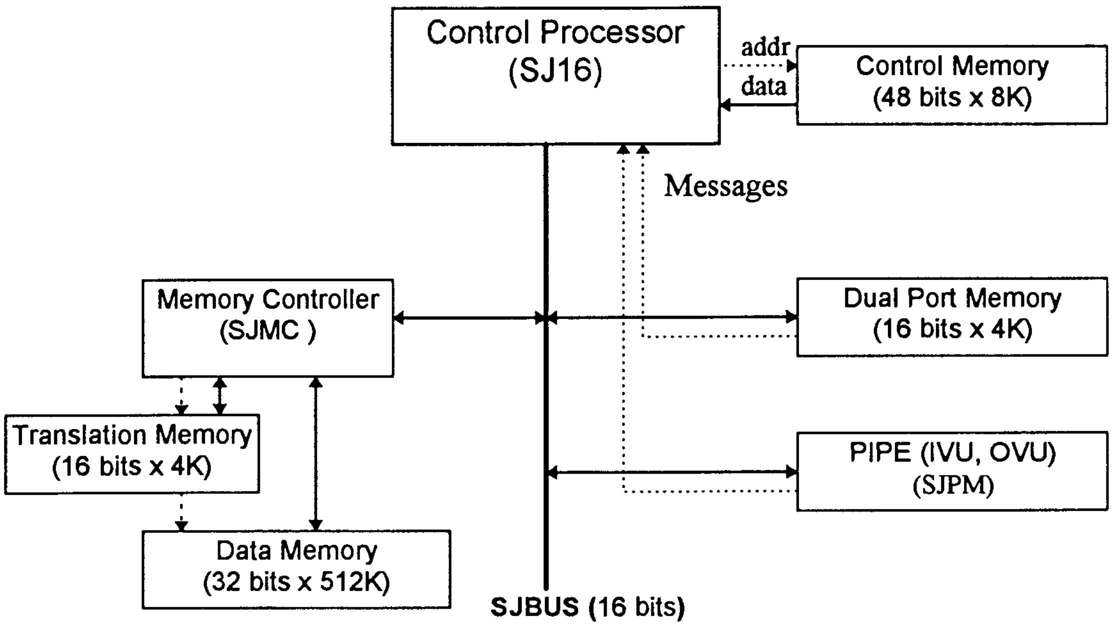

Control Processor (SJ16)
addr
data
Control Memory (48 bits x 8K)
Messages
Memory Controller (SJMC)
Dual Port Memory (16 bits x 4K)
Translation Memory (16 bits x 4K)
PIPE (IVU, OVU) (SJPM)
Data Memory (32 bits x 512K)
SJBUS (16 bits)

Figure 2-2: SAMjr Architecture

2.2.1 SAMjr

The PMU and DMU are two instances of a custom designed processor called SAMjr. SAMjr is a microprogrammable processor designed to be an efficient SAM building block. The architecture of SAMjr is shown in Figure 2-2 and is described in [Hob88B]. SAMjr consists of the SJ16 control microprocessor and co-processors including Dual Port Memory (DPM), segmented memory controller (SJMC), pipe interface (SJPM), and an optional auxiliary co-processor. The co-processors are connected to SJ16 via the SJBUS which is used both to send instructions in the form of Source/Destination (S/D) codes, and data. The SAMjr instruction format, shown in Table 2-1, includes the source and destination codes, thus allowing up to 2 co-processors to work in parallel with the SJ16 in any given instruction cycle.

2.2.2 SJ16

The SJ16 is a custom designed VLSI microprocessor with special features to facilitate SAM implementation. The architecture of the SJ16 is shown in Figure 2-3 and described in [Hob88]. The SJ16 micro-instruction consists of the fields shown in Table 2-2. Several operations may be specified in parallel including data manipulation, counter increment, and next address generation. The next address field is highly encoded and serves to specify literal data values, procedure calls, and conditional branching based on status register flags, external messages, or data bus values. A 256-way EXEC procedure call provides an efficient mechanism for ADEL format and operator decoding.

Table 2-1: SAMjr microinstruction fields

Field	**Purpose**
Source	Source Co-processor instruction
Destination	Destination Co-processor instruction
Instruction	Control processor instruction

Table 2-2: SJ16 microinstruction fields

Operation	Field	Function
Data	ABUS	ABUS source register
	BBUS	BBUS source register
	WT	write T register
	WA	write A register
	XS	select ALU or barrel shifter
	F	ALU function or shift count
	SF	sample ALU flags
Counter	COUNT	increment Counter register
Address	ACTL	next address control
	NEXT	next address value

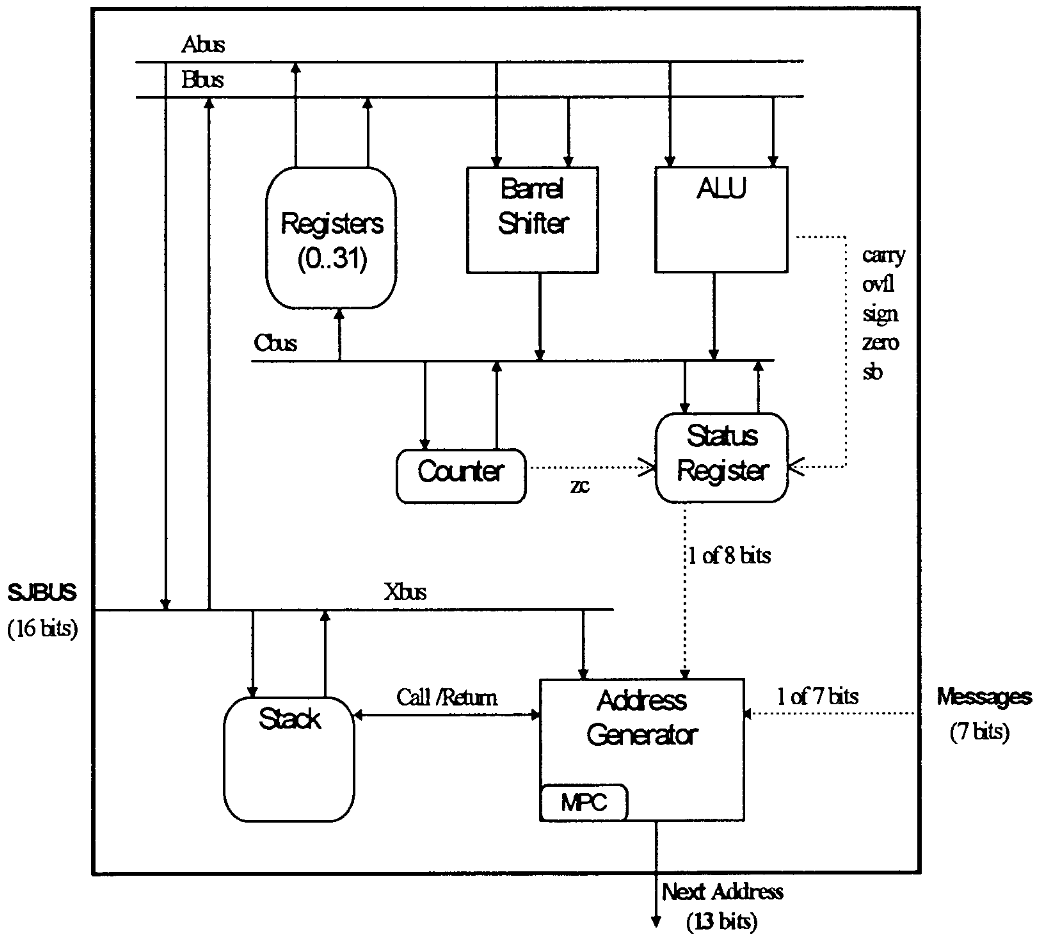


Figure 2-3: SJ16 Microprocessor Architecture

2.2.3 Dual Port Memory (DPM)

Dual Port Memory is used to pass data between the host and SAMjr. On the host side, Dual Port Memory is memory mapped. Each SAMjr gets a distinct DPM address range. On the SAMjr side, Dual Port Memory is accessed as a co-processor via Source/Destination codes. Two instructions are required by SAMjr to read and write data since a single bus is used for both address and data. Data in Dual Port Memory is word oriented.

2.2.4 SJMC

The SJMC is a custom VLSI Memory Controller which provides streamed access to segmented paged memory. The data streaming feature provides efficient access to logically sequential bytes or words in memory. Once a stream is started, it can deliver or receive a data item every clock cycle. The SJMC has 8 streams available.

The SJMC also implements a segmented paged memory system. A segment is a logically contiguous collection of a variable number of fixed size pages. An address translation memory translates logical addresses to physical memory addresses. The translate memory content is managed by the SAMjr memory management software. It is important to note that no virtual memory capability is provided directly by the hardware.

2.2.5 SJPM

The SJPM is a custom Pipe and Mail co-processor known generically as "the pipe." It is the communication medium between the PMU and DMU. It consists of the Instruction Verification Unit (IVU) which is attached to the PMU SAMjr SJBUS and the Operand Verification Unit which is attached to the DMU. The SJPM contains two FIFOs for instruction passing from IVU to OVU, a set of dual-port registers accessible to both sides, a state machine which controls execution, and error checking logic. Each end of the pipe has a distinct set of instruction codes. In addition, the OVU contains tag memory which is used for error checking.

In normal execution, the PMU waits for an empty FIFO, then loads format, operand, and operator bytes of an ADEL instruction into it, checks for errors and if none, releases the FIFO. It then does the same to the other FIFO. The DMU waits for a released FIFO, reads the contents, checks for errors, then executes the ADEL instruction.

Error checking is done on each ADEL instruction both by the IVU and OVU. Each operand has a tag which describes the data item. On the PMU side, the type of the operand is specified as one of: variable, constant, function, or reserved. The IVU uses a compatibility matrix to ensure that the types conform to the operation. For example, a constant or a function cannot be specified as the destination. On the DMU side, each operand has a tag that describes the operand shape (undefined, scalar, array, or reserved), and data type (character, boolean, integer, or floating point). The OVU uses compatibility matrices to ensure operand conformity. For example, adding a character and an integer is a domain error.

The SJPM registers are used to exchange status information and system values between the PMU and DMU. Since this is the only means of moving data from the DMU to the PMU, branch destination values are passed this way, as are error codes.

2.3 SEDIT the front-end program

SEDIT is the program that executes on the front-end host PC and interfaces among all the components of SAM-1. SEDIT is written in C and is based upon a multi-window visual text editor described in [Roc88] with specific enhancements for SAM. It is a conglomeration of previously distinct programs. SEDIT has three distinct roles:

1. User interface
2. Debug and control interface to the SAM hardware
3. SAM APL interface

The user interface uses separate windows for the APL interface and the debugger. The APL window is where the user enters APL code and views results. There are commands to manipulate code and data in the window and to read and write the window content to a file. When an APL function is defined or when the user specifies a line of APL to be executed, SEDIT translates the source into ADEL and sends it to the PMU. Results from the DMU, and error messages from the PMU or DMU are displayed.

The SAM debugger includes the following commands:

- switch between communication with the PMU and DMU
- load microcode image files into control store
- view and modify Dual Port Memory (DPM)
- view and modify SJ16 registers
- single step execution
- trace execution flow for short duration
- set, clear, and show breakpoints
- redirect DEBUG source to a file

The debugger represents a very primitive debugging facility, a factor that made software development of SAM microcode an arduous task. During the development of SAM APL, the programmer had to devise an ingenious system of dump-and-analyze techniques via Dual Port Memory (DPM) in order to monitor the internal activity of the machine. This "debug-by-remote-control" process not only frustrated development, but obscured the microcode design because many instructions and subroutines embedded in the system are included entirely for debugging purposes. This code slows down execution considerably, and is difficult to remove.

The control interface of SEDIT, called SAMIO, controls the execution of the SAMjr hardware via IO mapped command and data registers. It has the following primitives:

- stop and start the SAM clock
- retrieve the current micro-program counter value of a SAMjr unit
- single step a SAMjr unit
- restart execution of a SAMjr unit from a fixed address
- modify the control store of a SAMjr unit

SAMIO implements all the control of the SAM hardware, there is no debug monitor code in SAMjr. The functionality of the SAM debugger is implemented in terms of the above primitives. Access to SAMjr internal values is implemented by temporarily loading program fragments into control store, executing them, and then restoring the original control store contents. The debug code fragments use DPM to send data to the front-end. Breakpoints are implemented by changing the breakpoint location in control store to an instruction that just repeats itself. The detection of breakpoints is up to the user -- that is, the user is not notified explicitly when a breakpoint is reached. An execution tracing feature is implemented by single stepping the SAMjr unit and noting the program counter value.

2.4 SAM microcode development environment

This section describes the SAM microcode development environment and presents some of problems resulting from the software engineering methodology imposed by it.

The language used to program the SAMjr processor is a subset of APL called microAPL. MicroAPL was conceived during the early design stages of the SAM project as a high level microprogramming language that could be used to describe an architecture and simulate its execution by interpretation within an architectural support package

[Hob87]. While the language itself is a good medium for hardware description, the software engineering aspects of APL are not well suited to large projects such as SAM.

MicroAPL is an assembly language in that source statements correspond directly to machine instructions. A single statement can specify multiple microoperations which are executed in parallel. Microoperations correspond to the functionality of SAMjr and include co-processor actions, data manipulation, and branching. Statements are combined into subroutines which can be invoked via the CALL and EXEC operations. Subroutines are combined into a control store image which is stored as a file to be loaded into SAM.

The SAM development environment is implemented in a commercial APL system, (Manugistics APL*PLUS, [Man95]), on an IBM PC. MicroAPL subroutines are entered as APL functions which are stored as APL objects in a microcode database. The subroutines are compiled into an intermediate form which is also stored in the database. Images are generated from an image specification file that specifies the subroutines to be included and their absolute addresses. An APL workspace manages the database and performs compilation of subroutines and generation of images.

From a software engineering perspective, the SAM development environment has several shortcomings, most of them inherited from the APL environment. The primary problem is the lack of packaging. Subroutines exist on their own with no internal information on their relationships with other subroutines. The only grouping mechanism is the image specification file which simply enumerates the subroutines. There is no provision for documentation of relationships among functions and no hierarchical structuring mechanism. Furthermore, the APL syntax does not encourage liberal documentation at the code level. Finally, the APL syntax provides no structured programming constructs, leaving the programmer with a basic GOTO as the only branching mechanism.

The end result is a large database of tersely documented subroutines and very little structural information. The PMU and DMU programs that implement SAM APL consist of approximately 250 subroutines each. These are divided into roughly 10 images which correspond to broad categories such as Supervisor and Utilities as well as patch images that overlay previous code with new versions.

Several images can be further combined into a grand image in order to simplify the loading of images into SAM. The production version of SAM APL consists of a grand image overlaid by several patch images both for the PMU and DMU. Unfortunately, along the way, the content of the grand images was lost. That is, there is not a complete mapping from the subroutine database to the microcode that executes SAM APL. Because the original developer of SAM APL is gone, and the external documentation is not sufficient, it is not possible to recreate the generation of the SAM APL code at this time. One of the objectives of the simulator is to gather call information in order to facilitate the mapping processes. The inability to modify the SAM APL code was a major factor in the design of VSAM.

2.5 SAM APL

SAM APL is the application that runs on SAM. It is a basic APL interpreter that has been implemented to demonstrate the SAM prototype. The interpreter is described in [Hos87]. An overview is presented here.

SAM APL consists of three parts: the SEDIT program which handles the user interface and translates APL code into ADEL, the PMU which stores functions, controls execution, and manages the symbol table, and the DMU which stores and manipulates data objects. The DMU and PMU parts of SAM APL are implemented in microcode and manipulate the hardware directly.

The PMU part of SAM APL consists of the following modules:

- Diagnostic routines for communicating debugging information to the front-end via Dual Port Memory (DPM).

- A supervisor which gets control during startup. The supervisor initializes the PMU environment according to parameters passed from SEDIT via Dual Port Memory (DPM). It then initiates a protocol with SEDIT for defining new functions and program execution.
- A linker which incorporates new functions into the environment. This consists of storing the function code, and registering all identifiers and constants used by the function in the symbol table.
- An environment manager that maintains the Symbol Table (ST) and Contour Access Table (CAT) during function execution.
- A memory manager which manages the storage of PMU objects.
- Format subroutines that interpret ADEL instructions.
- Utility subroutines.

The basic algorithm of the PMU is:

1. Initialize environment.
2. Wait for a new function definition from SEDIT via Dual Port Memory (DPM).
3. Link the new function into environment.
4. If the new function type specifies that the function corresponds to a line of APL to be directly executed, then:
    5. Initiate pipe protocol with DMU.
    6. Execute the ADEL code for the new function.
    7. Wait for DMU to finish.
8. Goto 2.

APL function execution consists of executing the ADEL formats that comprise the function code. The IFETCH routine fetches the instructions and decodes them via an

EXEC call to the appropriate format subroutine. The format subroutine performs the actions appropriate to the format. Format types include data manipulation which is passed on to the DMU via the pipe, and execution control types which alter the instruction sequence.

Formats that perform conditional branches require a target value which is a data item stored in the DMU. The value is requested via a special DMU format which returns the value through the SJPM (Pipe) registers. The PMU is forced to wait for this value before it can continue. This is a major cause of delay in SAM APL execution as described in Chapter 4.

Before an instruction can be passed on to the DMU, the PMU must wait for a free pipe. Since there are two pipes, in general the PMU can load the next instruction while the DMU executes the last one. If the DMU gets behind, the PMU is held up.

The DMU part of SAM APL is an input driven program. After the initial startup processing, the DMU executes an IEXEC loop which gets instructions from the pipe and executes them by decoding the instruction format. Most of the formats executed by the DMU manipulate data. There are also formats for returning values to the PMU for branching, and for sending data to SEDIT via DPM which is how results get back to the user.

3. VSAM Implementation

This chapter describes the implementation of the VSAM simulator. It begins with an overview of VSAM including the objectives of the project and the implementation methodology. The major parts of VSAM are then described in detail in separate sections.

3.1 Overview

The motivation for VSAM was a need to observe and measure the performance of SAM-1 and future versions of SAM with the goal of assessing the efficiency of the architecture and identifying areas for possible improvements. The study began with the idea of instrumenting the SAM-1 prototype, however this turned out to be difficult for a number of reasons and was abandoned. After some consideration, a simulator-based approach was chosen for the following reasons:

- The process of replicating SAM would be a good way to learn the details of SAM and a motivation for compiling SAM documentation previously distributed in various forms and degrees of precision.
- A software version of SAM provides a flexible basis for further SAM research since it can easily be modified.
- A simulator is a better platform for observing architectural level behavior than hardware which is difficult to instrument and obscures design with detail.
- A simulator would allow observation of SAM APL "in situ", an important factor in light of the software development environment difficulties discussed in the previous chapter.
- A simulator would be a better platform for implementing a new software debugger interface for SAM since it is not encumbered by hardware interface limitations.

In order to allow the kind of observations desired, a detailed behavioral model of SAM-1 was constructed. The model is hierarchical in structure and corresponds closely to the structure of SAM-1 hardware. At the top level of the hierarchy, separate operating system tasks (processes) are used for the different units. At the bottom level of the hierarchy, microinstructions are directly executed and registers, busses, and memories are simulated. Each execution unit has its own user interface which provides execution control for the unit and gives access to the unit's data elements. Any part of the system can be instrumented by modifying the simulator software with probe instructions that send data to separate display processes.

The implementation platform for VSAM is C++ under OS/2. OS/2 was chosen for its multi-tasking capability and its DOS compatibility. Multi-tasking was clearly an appropriate way to simulate the multiple processors of SAM. DOS compatibility was important for continuity with the existing environment. Under OS/2 the APL-based SAM microcode development environment, SEDIT, and the simulator could all co-exist on a single machine. The initial implementation of VSAM is text based, but it was important to have a migration path to a future GUI version via the OS/2 Presentation Manager. C++ was a natural choice for the implementation language because of its object-oriented nature, and because SEDIT was already written in C. Object-oriented techniques turned out to be a good way to duplicate the modular structure of hardware, although little use was made of the class inheritance mechanism. All in all, OS/2 lived up to expectations and proved to be a good choice.

3.2 The model

An important decision in the design of VSAM was the nature of the model and the user and instrumentation interfaces. Initial research concentrated on a powerful visual approach. What was envisioned was a kind of animated hierarchical architecture block diagram that would allow the user to watch the system during execution and to zoom in and out on specific components as desired. As the view zoomed in, more detailed structural components would be visible and execution would be divided into steps appropriate to the view level. As execution proceeded, the diagram would show the current values of components and present an overall sense of the flow of data and control in the system. The view level and the rate of execution would be under direct control of users, allowing them to focus on the interesting parts of the machine and program. Execution could be stopped and component values modified. Instrumentation would be achieved by attaching probes to the object of interest and hooking them up to various instruments.

While very appealing, the visual approach proved to be far too ambitious given the time and resources available. It was also not necessary for the immediate goals. With the visual approach as a general guiding principle, a more pragmatic approach was chosen.

The hierarchical structure was maintained, but instead of a unified visual interface, VSAM uses separate text windows to control and access the state of the individual units. The unit windows are the debug and control interfaces to the SAMjr simulators. All of the SAMjr components are accessible through commands. Execution control and monitoring is also affected through the unit windows. Instrumentation is achieved by modifying the simulator code at the appropriate location with instructions that send data to an instrument process.

An important step in simulating a system is the verification of the model accuracy in representing the system. In the case of VSAM, verification was achieved through execution of identical code in the SAM prototype and VSAM. The same input problem was specified for both, and the results were compared. This was done with several benchmarks which thoroughly exercised all parts of the machine. The verification process was in fact part of the VSAM debugging process. It was an exciting moment when VSAM was able to add two numbers and give the correct result!

3.3 VSAM Architecture

VSAM consists of a number of cooperating OS/2 sessions. (A session is a process with a display window and a virtual keyboard.) The main session is VSAM, an administrative session that creates the various resources such as shared memory, pipes, and semaphores which are used by other sessions. VSAM also creates the other sessions and stops them when it terminates. The other sessions are SEDIT, VPMU, and VDMU. SEDIT is the front-end user interface program. VPMU and VDMU are instances of VSAMjr, the SAMjr unit simulator, corresponding to the PMU and DMU. The VSAM architecture is shown in Figure 3-1. This figure can be compared with Figure 2-1 which shows the SAM architecture.

OS/2 provides inter-process communication via semaphores, pipes, and shared memory. See [IBM94] for details. Semaphores can be event semaphores which allow synchronization, or mutual exclusion (mutex) semaphores for protected access to shared resources. Pipes are a type of point-to-point connection designed for client-server communication. Shared memory gives multiple processes access to the same memory.

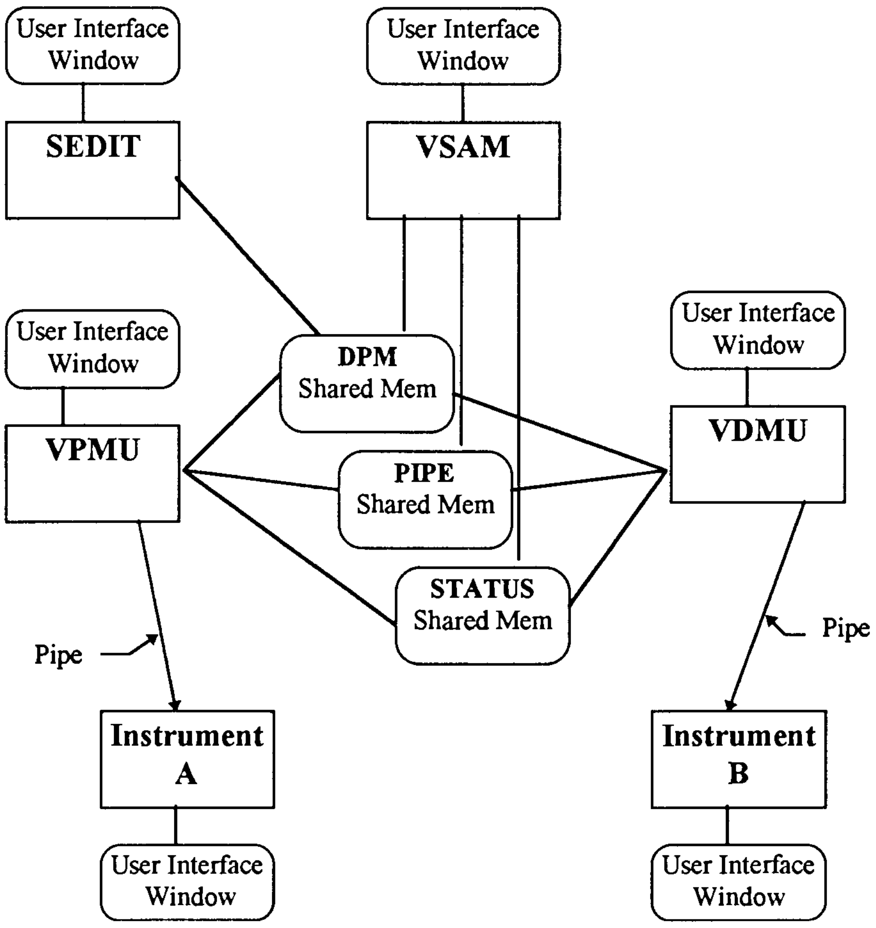


Figure 3-1: VSAM Architecture

Semaphores are used throughout VSAM. Pipes are used between the units and the VSAM main session for instruction execution control. Pipes are also used to connect instrument probes to the instrument process. Shared memory is used to implement DPM, and SJPM. A Status shared memory was added late in the project to aid instrumentation.

The VSAM session establishes the working environment for VSAM and controls overall execution. The session provides a user interface which is intended to give access to global data structures and system parameters. Currently, the interface only provides commands to pause and resume system execution, and to terminate VSAM. The VSAM session uses command line parameters which determine how the system is initialized. One set of these parameters can specify that any of the SEDIT, VPMU, and VDMU sessions can be executed under the C++ debugger (Borland TD) which allows for the debugging of the session software. Other VSAM command line parameters specify command source files to be executed by the units during system startup. After initialization, the VSAM session executes a loop which coordinates the execution of instructions by the VSAMjr units, handles user commands, and provides a place to attach instrumentation probes. An outline of the VSAM session main procedure follows:

```
void main( int argc, char *argv[] )  // VSAM main procedure.
{
      //--- Initialize system

      ::SysClock = 0;
      UserMsg( "Starting VSAM Master initialization." );
      SJMP_Create_smem();
      Create_SeditSem();
      Create_StartupSem();
      CreateStatus();
      ...

      // Parse command line args and start other sessions.

      ...
      UserMsg( "Start SEDIT session..." );
      if ( debug_sedit )
            StartDebugSession( "c:\\agv\\sedit\\SEDIT.EXE", sedit_args,
                               SEDIT_sessionID, SEDIT_processID );
      else
            StartSession( "c:\\agv\\sedit\\SEDIT.EXE", sedit_args,
                          SEDIT_sessionID, SEDIT_processID );
      UserMsg( "SEDIT session started!" );
      ...
```

```
        //--- Execute loop until user stop or error stop

        int sender;
        msg_type mtype;

        int utilz_counter = 0; // Utilz instrument.

        ::StepMode = 0;
        int exit = 0;
        while( !exit ) {
              if ( MSG_GetAny( mtype, sender ) )
                    if ( exit = ProcessMessage( mtype, sender ) )
                          continue;
              if ( kbhit() ) { // Invoke user interface
                    vm_user_action action = vuser();
                    exit = action == VMU_END;
                    ::StepMode = action == VMU_STEP;
              }

              // Utilz instrument probe code.
              if (++utilz_counter >= Utilz_sample_period ) {
                    utilzc_send( ::SysClock, GetStatus_PMUwait(),
                                                          GetStatus_DMUwait() );
                    utilz_counter = 0;
              }
        }

        //--- Stop all sessions and exit

        UserMsg( "Stopping unit sessions!" );
        MSG_Send( MSG_STOP, UNIT_VPMU );
        UnitStatus[UNIT_VPMU] = US_STOP;
        MSG_Send( MSG_STOP, UNIT_VDMU );
        UnitStatus[UNIT_VDMU] = US_STOP;
}
```

SEDIT, the front-end program for SAM, has been ported from DOS to OS/2. It executes as a separate session and communicates with the PMU and DMU via Dual Port Memory (DPM). The control interface of SEDIT, SAMIO, is disabled in VSAM. The only debugger commands that work are the Dual Port Memory display and modify commands. The functionality of SAMIO and the SEDIT debugger has been moved to the VSAMjr units described below.

The moving of the control functionality from SEDIT to the VSAMjr units uncovered interesting time dependencies that were not anticipated at design time. In retrospect, more control functionality should have gone into the VSAM session user interface rather than the VSAMjr units, particularly startup and execution control. A command sequence at the VSAM level could have specified the timing dependencies contained in SEDIT. As it was, a number of semaphores were added strictly to maintain execution order. The sources of these dependencies were initialization protocols among the units that utilized DPM locations and registers in the SJPM Pipe unit as signals. It turns out that during startup, SEDIT must finish initialization before the DMU starts, and the PMU must wait for the DMU. Several DPM locations are also used as startup parameters. These dependencies are not inherent in the design of SAM, but were obviously added during SAM-1 implementation. They were not anticipated during the partitioning of function of the VSAM simulator and were only discovered during simulator debugging.

The APL interface of SEDIT has been left as is. Unfortunately the APL character set has not been implemented for OS/2. In DOS, SEDIT modified the display character generator to implement APL characters. The same approach does not appear possible in OS/2. The result is that the special APL characters show up in OS/2 version of SEDIT as strange symbols. It is not however difficult to interpret the display and it was not deemed a high priority to achieve the translation at this time. Various approaches are feasible, including turning SEDIT into an OS/2 Presentation Manager application which would support arbitrary fonts.

The PMU and DMU are implemented as separate sessions consisting of the VSAMjr simulator, a user interface for debugging and unit control, and an execution control interface to the VSAM session. The sessions are called VPMU and VDMU, and are nearly identical except for minor details relating to specific differences between the PMU and DMU such as Dual Port Memory and the SJPM Pipe. Execution proceeds one instruction at a time with the two units kept synchronized by the VSAM session. The purpose of the synchronization is to maintain predictable behavior of the simulator during debug sessions. If one of the units is stopped by a breakpoint, for example, the other unit will wait before executing the next instruction. The synchronization is implemented by a message protocol via pipes between the units and VSAM. When a unit is ready to execute the next instruction it sends a READY message and waits for an EXECUTE message. It turns out that this co-ordination is a large source of OS/2 overhead due to the process switching involved. Because of the length of the initial startup code, it was decided to de-couple the units during startup and let them run at full speed. The subsequent speed up in execution speed of each unit was at least a factor of 10. This suggests that another mechanism such as a pair of event semaphores may be a better way to implement the synchronization of the units. One semaphore would indicate that a unit is ready, and the other would correspond to the EXECUTE message. This scheme avoids the costly process switch to the VSAM session.

The VSAM control interface of the VPMU and VDMU sessions is contained in the main function of the unit sessions. The interface consists of establishing access to shared resources, initialization, and then proceeding with execution under the control of the VSAM session instruction execution protocol. Initialization includes local variable settings and also execution of startup commands from the user interface, possibly through a specified command source file. An outline of the VPMU and VDMU session main function follows:

```
void main( int argc, char *argv[] )  // VPMU or VDMU session main.
{
      // Initialize session
      MSG_Init(); // Establish comm with VSAM
      VDPM_Init(); // Establish DPM access
      SAMjr.SJMP.Init(); // Establish SJMP_MUTEX
      OpenStatus(); // Status Smem
      ...

      int msg = 0;
      while( msg != MSG_STOP ) {

            // Reset VSAMJR
            SysClock = 0;
            SAMjr_PC = 3;
            SAMjr_PC_old = 0;
            SAMjr.CP.Reset();
            SAMjr.SMem.Reset();
            SAMjr.DPM.Reset();
            VDPM_Reset();
            StepMode = 0;
            BreakMode = 0;
            Reset_tracepoints();
            ...

            // Unit Start up - ini file according to command line args

            char *startupfile = VSAMJR_UNIT ".INI";
            if ( argc > 1 )
                  if ( *argv[1] == '-' )
                        startupfile = 0;
                  else
                        startupfile = argv[1];
            if ( UnitStartUp( startupfile ) == 'Z' )
                  msg = MSG_RESET;
            SysClock = 0;

            // Execute instructions...
            while( msg != MSG_STOP && msg != MSG_RESET ) {

                  // Go to user if step, breakpoint, or user input
                  if (   ::StepMode
                        || ::BreakMode && Test_breakpoint( SAMjr_PC )
                        || ::BreakDPMmode && Test_DPMbreak()
                        || ::BreakCallMode && Test_CallBreak()
                        || ::BreakPipeMode && Test_PipeBreak()
                        || ...
                        || kbhit() )
                    UserBreak();

                  // Signal "Ready to execute instruction" to VSAM
                  MSG_Send( MSG_READY );

                  // Wait for message from VSAM;  process user input if any
                  while( MSG_NULL == (msg = MSG_Get()) )
                        if ( kbhit() )
                              UserCommand();
                    // Carry out VSAM message
                    if ( msg != MSG_EXEC )
                        break;

                    // Execute instruction
                    ProcessAddress(::SAMjr_PC);
                    Add_tracepoint(::SAMjr_PC);
                    ::SAMjr_PC_old = ::SAMjr_PC;
                    ::SAMjr_PC = SAMjr.Execute( ::SAMjr_PC );
                    if ( ::SAMjr.SimBreak() )
                        SimBreak();

                    // Increment System Clock
                    ::SysClock++;
        }
}
```

3.4 The VSAMjr simulator

The VSAMjr simulator structure closely resembles the SAMjr hardware. Essentially, the SAMjr design was implemented in software instead of hardware. It is interesting to note that the software version was much easier to build, but executes about 1000 times slower than the hardware.

This similarity in structure is deliberate for the following reasons:

- Ease of development - the simulator was built directly from the hardware specifications and ambiguities were resolved by inspecting the hardware.
- Ease of documentation - the same documentation that applies to the hardware applies to the simulator. Also, the simulator implementation and the hardware complement each other in documenting SAM.
- Ease of verification - the simulator implementation is easy to verify step by step by comparison to the hardware.
- Ease of instrumentation - instrumenting the simulator is analogous to instrumenting the hardware. The same objects and events are involved in both.

- Ease of modeling future modifications to SAM architecture - since the simulator and hardware are nearly identical, the designer can try out proposed hardware changes on the simulator and evaluate their effectiveness.

Object oriented techniques were applied to package the various components of the simulator into neat modules with well-defined interfaces. This closely represents the component nature of hardware. Generally, there is a one-to-one mapping between the hardware components and object classes representing them. The VSAM classes with their nesting and a brief explanation are:

samjr	SAMjr unit simulator
sjinst	SAMjr microinstruction decoding auxiliary class
cp_sj16	SJ16 Control Processor simulator
cpstack	SJ16 stack class
smem	SJMC Memory Controller simulator
dpm	Dual Port Memory simulator
sjpm	SJPM (Pipe) simulator including the IVU and OVU

The highest level class is samjr which stands for the SAMjr processor. In VSAM it is instantiated as the PMU and DMU. The definition is:

```
class samjr {
      CMem_instr  CMem[CMEM_SIZE];  // Control memory
      cp_sj16     CP;               // Control Processor
      smem        SMem;             // Segmented memory
      dpm         DPM;              // Dual port memory
      sjmp        SJMP;             // Pipe chip: IVU or OVU
public:
      SAMADDR Execute( SAMADDR address );
};
```

The only method defined for samjr is Execute() which takes an address as input and returns the next address to be executed. As a side effect, Execute() modifies internal state. Execution of SAMjr is achieved by the following code:

```
samjr SAMjr;
SAMWORD PC;
PC = 3;  // SAMjr always starts executing at 3.
While( 1 )
      PC = SAMjr.Execute( PC );
```

There is no defined termination condition for SAM. In case of an error, SAMjr usually ends up in a tight loop in an error subroutine so that the error may be detected by the user.

An important part of the simulator is the handling of sub-microinstruction events. The SAMjr instruction cycle is divided into 4 phases called T1 to T4. Events within SAMjr are co-ordinated with respect to these phases. Some important events are:

- during T4, the next microinstruction is fetched, and the Source and Destination codes are placed on SJBUS for co-processors to latch. Part of the Source/Destination code is a select field which activates only the specified co-processor.
- during T2, the selected source co-processor outputs its value onto SJBUS, and the selected destination co-processor latches it.
- data operations in the Control Processor start at T3

The simulator emulates the data flow of SAMjr, but not the actual timing. The simulator instruction cycle has the following sequence:

1. Fetch the next instruction.
2. Invoke the Source processing part of the specified co-processor with the Source code as a parameter. Store the return value in variable sjbus.
3. Invoke the Destination processing part of the specified co-processor with the Destination code and the value of sjbus as parameters.
4. Invoke the Control Processor execution processing function with the value of sjbus as a parameter.

Each co-processor has a source processing and destination processing part. This includes the Control Processor which performs literal, register, and stack input and output during the source/destination phase. The Control Processor also has a process part that executes the rest of the microinstruction.

The samjr Execute() function controls the order of events within a microinstruction:

```
SAMADDR samjr::Execute( SAMADDR CMem_addr ) // Execute an instruction
{                                           // Return next addr to execute.
      SAMADDR next_addr;
      SAMWORD SJBUS;

      // Fetch instruction
      sjinstr cur_inst = CMem[ Cmem_addr];

      // Source processing
      switch ( cur_inst.Source_unit() ) {
            case CP_UNIT:
                  SJBUS = CP.Source( cur_inst );
                  break;
            case DPMem_UNIT:
                  SJBUS = DPM.DPM_source( cur_inst.Source() );
                  break;
            case SMem_UNIT:
                  SJBUS = SMem.Source( cur_inst.Source() );
                  break;
            case SJMP_UNIT:
                  #ifdef PMU
                        SJBUS = SJMP.IVU_source( cur_inst.Source() );
                  #else DMU
                        SJBUS = SJMP.OVU_source( cur_inst.Source() );
                  #endif
                  break;
      }
      // Destination processing
      switch ( cur_inst.Dest_unit() ) {
            case CP_UNIT:
                  CP.Dest( cur_inst, SJBUS );
                  break;
            case DPMem_UNIT:
                  DPM.DPM_dest( cur_inst.Dest(), SJBUS );
                  break;
            case SMem_UNIT:
                  SMem.Dest( cur_inst.Dest(), SJBUS );
                  break;
            case SJMP_UNIT:
                  #ifdef PMU
                        SJMP.IVU_dest( cur_inst.Dest(), SJBUS );
                  #else DMU
                        SJMP.OVU_dest( cur_inst.Dest(), SJBUS );
                  #endif
                  break;
      }
      // CP processing
      next_addr = CP.Process( CMem_addr, cur_inst, SJBUS );
      return next_addr;
}
```

Since a co-processor cannot be both a source and destination, it will at most be called once during a cycle. The source and destination parts must complete all processing for that cycle in the single call. The source or destination code which is the instruction to be executed by the co-processor is passed to the co-processor as a parameter. The typical implementation of a co-processor is demonstrated below in a simplified form of SMem. Note that in the case of SM_S_SCLR which flushes a data stream and SM_D_SRB which initiates a stream, further processing is required to complete the instruction.

```
SAMWORD smem::Source( const SAMBYTE source_code )
{
      SAMWORD dataout;
      int stream = sdcode_stream( source_code );
      switch ( sdcode_function( source_code ) ) {
            case SM_S_SORS:
                  dataout = SOffset[stream]; break;
            case SM_S_SBRS:
                  dataout = SBase[stream]; break;
            case SM_S_SSN:
            //...
            case SM_S_SCLR:
                  dataout = 4 - SStatus[stream].BA;
                  AOL = Make_SM_address( SBase[stream], SOffset[stream] );
                  Mem_write_request( AOL, SStatus[stream].BS, stream,
                                          SStatus[stream].W );
                  break;
            }
      return dataout;
}
SAMWORD smem::Dest( const SAMBYTE dest_code, const SAMWORD datain )
{
      int stream = sdcode_stream( dest_code );
      switch ( sdcode_function( dest_code ) ) {
            case SM_D_SBRD:
                  SBase[stream] = datain;
                  break;
            //...
            case SM_D_SRB:
                  SOffset[stream] = datain + 4;
                  SStatus[stream].M = MBYTE;
                  for ( i = 0; i < 4; i++ ) SStatus[stream].W[i] = 0;
                  SStatus[stream].BS = 0;
                  SStatus[stream].BA = datain & 0x3;
                  SStatus[stream].BF = 0;
                  AOL = Make_SM_address( SBase[stream], datain );
                  Mem_read_request( AOL, SStatus[stream].BS, stream );
                  break;
      }
      return datain;
}
```

The SJ16 control processor class is defined as:

```
class cp_sj16 {
public:
      SAMWORD Register[CP_NUM_REGISTERS];
      cpstack Stack;

      SAMWORD Source( sjinstr );                    // Source processing
      void Dest( sjinstr, SAMWORD datain );         // Destination processing
      SAMADDR Process( SAMADDR address,             // Execution processing
                       sjinstr,
                       const SAMWORD input_bus );
private:
      SAMWORD alu( SAMWORD a, SAMWORD b, int fn, SAMWORD &flags );
      SAMWORD xshift( SAMWORD a, SAMWORD b, int xcount );
      SAMADDR agen( SAMADDR mpc, sjinstr ci, SAMWORD bus );
};
```

The Control Processor (CP) data members are the register file and the stack. The register file includes the general purpose registers as well as the special registers Counter, IO, T, and Status. The stack is a simple LIFO stack implemented by the cpstack class. The public methods for cp_sj16 are Source(), Dest(), and Process() which implement the source, destination and execution part of the SAMjr instruction cycle. Source() handles literal specification, and output of register or stack values onto the bus. Dest() handles input of stack values from the bus. Process() performs the movement of data among the SJ16 internal registers and processing units, and the generation of the next microinstruction address. The internal methods invoked within Process() are alu(), xshift(), and agen() which implement the ALU, barrel shifter, and next address generation logic components respectively. These functions are relatively straightforward though somewhat tedious in detail.

The Dual Port Memory (DPM) co-processor is implemented by the class dpm defined as:

```
class dpm {

      SAMWORD in_data, out_data;  // DPM data latches

public:
      void Reset();
      SAMWORD DPM_source( SAMBYTE source_code );
      void DPM_dest( SAMBYTE dest_code, SAMWORD sjbus );
};
```

This definition hides the details of the shared memory implementation of DPM. The details consist of a separate named shared memory for each unit session, and a named mutex semaphore used to protect access to the shared memory. The names are constructed from the values of preprocessor constants.

The SJMC memory controller is implemented by the class smem defined as:

```
class smem {

      SAMBYTE SMem[SMEM_SIZE]; // Segmented memory - bytes
      SAMWORD TMem[TMEM_SIZE]; // Translate memory - words

      SAMWORD SOffset[SMEM_STREAMS];  // Offset registers
      SAMWORD SBase[SMEM_STREAMS];    // Base registers
      smbuff SBuff[SMEM_STREAMS][2];  // Stream buffers
      SM_stream_status SStatus[SMEM_STREAMS];  // Stream status bits
      SMADDR AOL; // Address Output Latch
public:
      void Reset();
      SAMWORD Source( const SAMBYTE source_code );
      SAMWORD Dest( const SAMBYTE dest_code, const SAMWORD );
private:
      void Mem_read_request( SMADDR addr, BIT bs, int stream );
      void Mem_write_request( SMADDR addr, BIT bs, int stream, BIT w[] );
};
```

Class smem contains the segmented and translate memory data structures, as well as the various data buffers and status bits required. The public methods Source() and Dest() implement the controller behavior. The methods accept the instruction byte as a parameter and proceed to decode it into the stream number and specific action code. All actions that take place in the hardware during the rest of the instruction cycle are performed by smem.Source() and smem.Dest() before they return.

The SJPM Pipe co-processor is more complex than the other co-processors since it connects the PMU and DMU. In VSAM, this is achieved by dividing the co-procesor into IVU functions which are part of the PMU session, OVU functions which are part of the

DMU session, and the SJPM shared memory which is accessed by these functions. Access to the shared memory is protected by a mutex semaphore. The class and supporting definitions are:

```
struct sjmp_shared_data {

      // sjmp registers - shared
      SAMWORD sjmp_regs[SJMP_REGISTERS];

      // Pipe status flags
      BIT Is, Ir, Os, Or;  // State bits - shared
      BIT Pe, ISe, Ie, Oe; // IVU flags
      BIT OSe, Oep;        // OVU flags

      // SJMP FIFOs - shared
      SAMWORD FIFO[2][SJMP_FIFO_SIZE];
      int FIFO_wcount[2];
      int FIFO_maxwcount[2]; // OVU uses this to read the FIFO.

      // IVU syntax tag registers
      int Idest, Ileft, Iright;

      // OVU Tag memory
      SAMWORD TagMem_addr;
      SAMBYTE TagMem[TAG_MEM_SIZE]; // lower 4 bits are valid data.

      // OVU Semantic tags
      int Dtag, Ltag, Rtag;  // lower 4 bits are valid data.
      BIT Lv, Rv;            // valid tag bits
};

class sjmp {
      HMTX sjmp_mutex;
      sjmp_shared_data *sd;
public:
      SAMWORD IVU_source( const SAMBYTE source_code );
      void IVU_dest( const SAMBYTE dest_code, const SAMWORD datain );
      int IVU_msg( int msg );

      SAMWORD OVU_source( const SAMBYTE source_code );
      void OVU_dest( const SAMBYTE dest_code, const SAMWORD datain );
      int OVU_msg( int msg );
private:
      void SM_in();
      void SM_out();
      SAMWORD IVU_Cmat();
      SAMWORD IVU_status_word();
      int Dest_shape_tags();
      int Dest_type_tags();
      int Dest_tags();
      int OVU_CMat();
      BIT OVU_Oe();
      SAMWORD OVU_status_word_1();
      SAMWORD OVU_status_word_2();
};
```

3.5 The VSAMjr debugger

The purpose of the VSAMjr debugger is to control the execution of the simulator and to give the user access to the state of the simulated machine so that the correctness of the executing microcode can be determined. The debugger uses a command line interface with three groups of commands:

1. Control of the environment including loading of control memory, scripting, and general session control.
2. Execution control via breakpoints and single stepping.
3. Object access to data elements, state values, memory contents, and execution history.

The debugger is an integral part of the SAMjr simulator. Since it must have access to internal elements of the simulator, many access and display functions were added to the basic simulator classes. (These were left out of the previous SAMjr simulator discussion for conciseness.) The debugger is the user interface to the VSAMjr program which executes as the PMU and DMU. The debugger is invoked by VSAMjr during startup, when a breakpoint is reached, or when the user enters input into the debugger window. The debugger is only invoked between SAMjr instructions which are indivisible from the user point of view. The VSAMjr main execution loop checks for breakpoints and user input before it executes an instruction. In the following (simplified) code fragment from VSAMjr, the functions UserBreak(), UserCommand() and SimBreak() invoke the user interface. The function SimBreak() is used to signal special conditions such as invalid machine operations.

```
while( msg != MSG_STOP && msg != MSG_RESET ) {

      // Go to user if step, breakpoint, etc., or user input
      if (      ::StepMode
            || ::BreakMode && Test_breakpoint( SAMjr_PC )
            || ...
            || kbhit() )
            UserBreak();

      // Signal "Ready to execute instruction" to VSAM
      MSG_Send( MSG_READY );

      // Wait for msg from VSAM; process user input if any
      while( MSG_NULL == (msg = MSG_Get()) )
            if ( kbhit() )
                  UserCommand();

      // Carry out VSAM message
      if ( msg != MSG_EXEC )
            break;

      // Execute instruction
      ProcessAddress(::SAMjr_PC);
      Add_tracepoint(::SAMjr_PC);
      ::SAMjr_PC_old = ::SAMjr_PC;
      ::SAMjr_PC = SAMjr.Execute( ::SAMjr_PC );
      if ( ::SAMjr.SimBreak() )
            SimBreak();

      // Increment System Clock
      ::SysClock++;
}
```

The debugger syntax was kept very simple for ease of implementation. Commands were added during development of VSAM as need arose. The basic format is a single character which determines the type of command, followed by optional characters for modifiers, followed by optional parameters. For example, the memory command demonstrates the complete syntax. It is a highly overloaded command since it provides access to three types of memory. It has the following forms:

```
MS [s] -- display the status of SMem for stream s (or all streams)
MDD    -- display the DPM data latch value
M{D|S|T}{V|C|F} [addr[{-addr|,count}]][=value] -- View/Change/Fill memory
```

The last form requires further explanation. After the M, the first modifier is the memory specifier -- one of: dual port memory (DPM), segmented memory (SMem), or translate memory (TMem). The next modifier is the action, one of view, change, or fill. Next come the parameters which specify the address range in various forms, and an optional value. For example, the command MDV 1000,20 displays 20 values of the DPM from address 1000. The command MSF 0-100=0 fills the first 100 locations of segmented memory with zeros. The memory command also has an interactive mode which steps through memory and allows the user to change only selected values.

Most commands are much simpler. The I command, for example has the form:

```
I[{+|-}] [a]
```

which shows a disassembled view of control memory from the specified address, or relative to the previous I command if + or - is specified.

A novel feature of the VSAMjr debugger is the use of color to highlight key data objects in a complex display such as the register file which consists of 32 registers, 4 of which are dedicated. The foreground color indicates whether the register has changed since the last time it was displayed by using yellow for changed and white for not changed. The background indicates special status such as the IO, Counter, Status, and T register which each get a dedicated color, and the target register of the last instruction. This has turned out to be a very effective technique and represents the first step to a graphical interface that would allow the user to organize the display in a meaningful way.

Breakpoints are an important feature of a debugger. The standard type of breakpoint specifies a break when a given address is about to be executed. In the VSAMjr debugger, these breakpoints are implemented by keeping a list of breakpoint addresses and checking this list at the start of each instruction. This is less efficient than the usual method of modifying the instruction, but it has the advantage of leaving control memory pristine. The debugger also has breakpoints that examine the Source/Destination codes and stop on instructions that use specified units. This is a valuable feature for debugging co-processor software. Other breakpoint type features include a break on function call and return, the execution stack display, and a trace of the last dozen executed instructions.

A couple of features that did not get implemented due to their complexity, but would have been very useful are datapoints and reverse execution. Datapoints cause execution to break upon access to specified data objects. In VSAM, data objects could be various machine registers and flags, as well as locations in dual port memory and segmented memory. One possible implementation approach is to maintain a list of all datapoints in effect, and search this list for each data object accessed by each instruction. This approach seems straightforward in concept, but does require interpretation of each instruction in the context of various register values, particularly in the case of segmented memory where buffering is taking place. This would probably incur a significant performance penalty. An alternative approach would be to give data objects the responsibility of knowing when the object is a datapoint, and detecting when the datapoint is triggered. This would reduce overhead for each instruction, but would require considerable modification to the simulator.

Reverse execution allows the user to back up from the current instruction to determine the events that led to it. This would be particularly useful in conjunction with breakpoints and datapoints, especially if the user could then modify some value and proceed with forward execution. The basic problem in reverse execution is that all the changes precipitated by each instruction must be reversible, and must be recorded during execution. Besides the performance and storage costs of this approach, reversibility may be limited by cascading changes.

During the development of VSAM, the debugger was used in reverse to the usual order of things. The program was assumed to be correct; it was the simulator that was being debugged. The process is essentially the same - the program is executed and the change in the state of the machine is monitored - except that the simulator program is itself run in a debugger, (in our case the Borland C debugger), and monitored. This gets particularly complex when multiple instances of the simulator are running each with its own (Borland) debugger as in the case of the DMU and PMU. Despite the large number of windows involved and the processing overhead, OS/2 was able to support this mode of debugging, and the technique proved quite effective.

3.6 Instrumentation

Since one of the primary motivations for building VSAM was instrumentation, the system includes a simple yet powerful instrumentation methodology. The instrumentation design goals were:

- flexibility and extensibility
- ease of instrument hook-up and take-down
- low impact on simulator design
- execution efficiency in space and time
- close analogy to hardware instrumentation methods such as logic probes

A generic instrument consists of three parts: the probe, the connection, and the display. The probe is the sensor that is directly attached to the object being measured. In the case of VSAM, the probe is a piece of software that is embedded in the simulator code. The probe software obtains the values of relevant variables and/or activities and sends them to the display unit via the connection. In VSAM, we chose OS/2 pipes as the method of connection based on the flexibility and simplicity of the pipe model. The display is an arbitrarily complex program that reads the probe data from the pipe, processes the data, and outputs it in some way. The output may be in the form of a visual display in a window, a file in trace or processed form, or both. The display program may be a fixed display type or may require user input for control.

An example of an implemented VSAM instrument is Callvue which captures subroutine calls and returns executed in the SAMjr microcode. This instrument was the first one built and was extremely useful during the debugging of VSAM. The Callvue display shows the names of subroutines as they are called in an indented call tree. The Callvue probe is attached to the SAMjr simulator in the next address generation module of the SJ16 control processor. If the next address action is a call or return, the probe sends a record down the pipe. The record specifies the current address, whether a call or return, and if a call, the target address. The display part of Callvue translates call addresses into subroutine names via a load map file and displays the name and address positioned according to the current call nesting level. Return records are only used to decrease the call level.

Callvue information is also used to build a dynamic call profile of how many times each subroutine was called and by whom. The call information is accumulated in the "calls" matrix where each element M[i][j] counts the number of times subroutine i calls subroutine j. The "calls" matrix is stored in a file at the end of a run. It is processed off-line to produce a histogram of often called subroutines. The transpose of the "calls" matrix corresponds to the "is-called-by" matrix where each element M[i][j] counts how many times subroutine i was called by subroutine j. The sum of a given row of the "is-called-by" matrix corresponds to the total number of times a subroutine was called. A dynamic call tree (as opposed to a static one) can be obtained from the "calls" matrix by following the call chain for each subroutine. The question "who calls subroutine i" can be answered from the "is-called-by" matrix. This can be very useful when a subroutine needs to be modified. Yet more information about subroutine relationships can be obtained by computing the transitive closure of the two call matrices to obtain a "uses" and "is-used-by" view of the software. The later tools are particularly important for software archeology - the process of trying to understand a software system from the bottom up, usually required when no design documentation is available.

Another useful instrument is the unit utilization trace tool called Utilz. The purpose of Utilz is to show the state of the PMU and DMU over time. Utilz shows when a unit is busy or waiting, and if waiting, it shows what the unit is waiting for. This is an important tool for assessing the degree of parallelism in the system, and determining the causes of stalls. The Utilz probe is embedded in the VSAM control module where it samples both the PMU and DMU status at once. This approach was chosen in order to explore the instrumentation methodology. Utilz is an example of a sampled tool. The probe only samples information every n cycles in order to reduce the overhead. The value of n is currently set as a compile constant in the probe.

In general, a VSAM instrument consists of the probe module and the display program connected by a pipe, configured in a client-server relationship with the display program as the server and the probe as the client. The display program establishes the pipe and waits for the probe to connect and start sending data. The display program must be started before the probe attempts to connect. If the probe fails to connect, it assumes that the display program is not present and effectively turns off the instrument. Generally the display program is configured to accept multiple simulation sessions.

The display program can be display-only with no user input, or fully interactive. The probe can be a passive probe which simply sends a one-way stream of data, or it could interact with the display via a bi-directional pipe. Such an active probe would contain local intelligence regarding when and what to sample. To date only simple instruments with write-only displays and passive probes have been built for VSAM. An example of where an active probe would make sense is a probe whose sampling rate can be changed dynamically by the display unit.

The probe module is linked into the simulator. It consists of general routines for connecting to the pipe and packaging data for transmission, as well as specific routines that gather the data and interface with the display program. Calls to the probe routines are inserted directly into the simulator code at strategic points, either in the VSAMjr instruction execution loop or within specific simulator components. This invasive approach allows arbitrary instrumentation flexibility, but does require that care be taken not to disturb the environment. Since the probe code is usually quite straightforward, this has not been a problem with the instruments implemented so far. For example, the Callvue instrument probe is inserted into the cp_sj16.Process() function after the next address has been determined. The probe code is shown below. The code that connects the Callvue probe to the Callvue instrument is contained in the VSAMjr unit main() function.

```
SAMADDR cp_sj16::Process( SAMADDR ci_addr, sjinstr ci,
                          const SAMWORD input_bus )
{
      // The data movement part
      ...
      // COUNT processing - ZC flag is updated at end of cycle ...
      ...
      // Compute next address
      SAMADDR next_addr = agen( ci_addr, ci, input_bus );
      if ( ci.x(9) == 0 && ci.x(10) == 0 || ci.actl() == ACTL_EXEC )
            Stack.Push( ci_addr+1 );

      // Update flags
      ...

      // Callvue instrument probe code!   Send call/ret msg.

      if ( ci.x(9) == 0 && ci.x(10) == 0 || ci.actl() == ACTL_EXEC )
            callvuec_call( next_addr );
      else if ( ci.actl() == ACTL_RETURN )
            callvuec_ret();

      return next_addr;
}
```

The instrument display program is an independent session in OS/2. It receives data from the probe, processes it, and displays it in an appropriate format. The Callvue display program, for example, receives call and return messages from the probe. The call target address which is contained it the call message is translated into a subroutine index and the name of the subroutine is displayed on the screen indented to the current call level. The call level is incremented for calls, and decremented for returns. Since Callvue must also increment the Calls matrix for the appropriate subroutines, a simple call stack is maintained in order to know who the caller was. The display program may also produce permanent files to store results for off-line analysis. In the case of Callvue, the Calls matrix is output to a file at the end of a benchmark execution run.

An important issue in the design of instruments is the definition of important events which act as triggers to start and stop data collection and reset. One convenient but not very useful event is the startup of the instrument itself. A more useful event is the connection of a probe to the pipe for a session. Generally, the instrument should reset itself at this time. An example of this is in Callvue, where upon probe connection the call level and all entries in the call matrix are set to 0. Other important events are instrument specific and must be specified in the probe-display protocol. An example of this is in

Profile, where a reset record can be sent by the probe to the display program, instructing it to reset its counters in preparation for a measurement run. It turns out to be useful to reset the display at the start of a trigger rather than at the end (e.g., probe disconnection) in order to leave the display for viewing by the user.

To enable sophisticated instrumentation, a rich choice of system status indicators must be available to instrument probes. The status must be globally available, and must be easily modified as new requirements are encountered. In VSAM, the Status shared memory was added just for this purpose. The need arose during the construction of the dynamic execution profile instrument. The profile desired was of the execution phase of a benchmark. Because SAM APL links new functions into the environment including the special immediate execution function that results in execution, there is a lot of activity on either end of the actual execution. The start trigger for the profile was to be when the PMU began executing the code for the immediate execution function. The end trigger was when the DMU began transferring the result to the ECU. These events were most easily localized by execution address, so the simulator was modified to set a flag in Status shared memory when the appropriate addresses were executed. The Profile probe watches these flags and only samples during the relevant time.

In retrospect, the use of the Status shared memory should have been incorporated into the simulator design as a central mechanism for posting important events and general exchange of data among different components of VSAM. The shared memory would contain two types of data, a fixed set of status flags that would describe the overall state of VSAM, and a flexible named message mechanism which would provide for arbitrary communication among cooperating components. Access to Status would be protected by a mutex semaphore. The contents of Status would need to be accessible to the VSAMjr debugger for full flexibility. The fixed status information could in fact be used to implement the instruction execution synchronization between the PMU and DMU which is currently implemented via pipes. The fixed information would be very useful for instrumentation and general debugging. Some care would need to be taken to ensure that Status information is kept well organized and coherent.

4. Benchmark Analysis Results

This chapter presents the results of the execution analysis of two APL benchmark programs, a scalar implementation of Quicksort called QSS, and a vector implementation called QSV. The purpose of this chapter is to demonstrate the use of VSAM to analyze the execution behavior of SAM and SAM APL. Detailed interpretation of the results is beyond the scope of this work.

The Quicksort benchmarks were selected because they have been previously written by Hoskin [Hos87] to run in SAM APL and have been discussed in [HHS92] and [CNS89]. The programs are shown in Appendix A. They are particularly interesting since they offer a direct comparison of scalar and vector implementations of the same problem. Both versions are recursive and use the algorithm of dividing the input vector into two parts based on a pivot value, and calling themselves to sort each part. The primary difference between the two benchmarks is in the divide step. The scalar version, QSS, manipulates the elements as scalars and uses the traditional swapping approach. The vector version, QSV, uses the APL vector function Select (/) to extract all elements less than and greater than the pivot. The scalar version performs a lot of branching and copying of single values. The vector version performs copying of vectors.

The analysis data was collected by the Callvue, Profile, and Utilz instruments of VSAM. Data was collected only while SAM was actually executing the benchmarks and does not include the translation from APL to ADEL, nor the linking of new functions into the environment. Data collection began when the immediate execution function corresponding to the line of APL to be evaluated began execution, and ended when the results were available for display. While each instrument has an on-line display, in-depth analysis was performed off-line by importing the trace files into Microsoft Excel [Mic94]. The benchmarks were executed under the same conditions with identical sampling rates and input. The input size was 10 elements. This is a relatively small size for a sort benchmark, but is sufficient to demonstrate the process. The execution speed of a fully instrumented VSAM is rather slow. The 10 element benchmarks each took approximately an hour of elapsed time to run. The results reported in [HHS92] show that the vector benchmark performance is always better than the scalar version, and that the advantage grows with input size.

The first part of each benchmark run constructs the input vector with a balanced distribution. This is not directly relevant to the analysis, except that it shows up as part of the run in the time profile graphs, for example Figure 4-2. Since the input generation program was nearly identical for both benchmarks, this phase of the run serves as an informal reference point among the time series graphs, although they are not exactly the same. Since the phase is nearly identical in both cases, it does not affect the results significantly.

A side-by-side comparison of some execution statistics is summarized in Table 4-1. The table shows that QSV executed in 26% less time than QSS, and executed significantly fewer subroutine calls. While the PMU utilization was higher in QSS than QSV, the DMU utilization was lower. This is a reflection of the greater amount of interpretive overhead incurred by the scalar version which executed more lines of APL, more APL function calls, and more APL branches. The vector version, on the other hand, moved more data within the DMU. The PMU and DMU overlap is the fraction of time that both units were busy. Surprisingly these figures are very close. Overall, the vector benchmark is more efficient

Table 4-1: Comparison of QSS and QSV execution statistics

	QSS	QSV	% Decrease
APL function Calls	25	18	28%
APL lines executed	122	84	31%
APL branches	91	35	62%
SAM clock cycles	48,350	35,650	26%
PMU microcode subroutine calls	5,346	3,952	26%
DMU microcode subroutine calls	11,702	4,851	59%
PMU subroutines invoked (of 219)	79	79	0%
DMU subroutines invoked (of 252)	107	104	3%
PMU utilization	60%	46%	23%
DMU utilization	74%	90%	-22%
PMU and DMU overlap	35%	37%	-6%

in each of the APL, SAM, and real-time domain.

4.1 Utilization

As a first look at how SAM spends its time, the Utilz instrument was designed to record the busy/waiting state of both units during execution. The PMU has three types of waits:

1. Waiting for a free instruction pipe FIFO to fill. This occurs when the PMU gets ahead of the DMU and both FIFOs are full.
2. Waiting for a branch destination value from the DMU.
3. Waiting for the DMU to finish executing the last instruction and sending results to the front-end. This wait only occurs at the end of immediate execution and is ignored in the rest of this discussion.

The DMU has only one type of wait, waiting for a full instruction FIFO to execute. The Utilz trace file records the state of each unit. The file can be analyzed in a number of ways.

Figure 4-1 is a summary of the unit busy/waiting state over the whole benchmark. It was obtained by totaling the trace file for each type of state. The figure clearly shows that the scalar benchmark took longer to execute. It also shows that the scalar benchmark used the PMU heavily, while the vector benchmark used the DMU heavily. This is consistent with the discussion in the previous section.

Figures 4-2 and 4-3 show unit utilization as a time profile. These figures were obtained by grouping the Utilz trace file into 50 equal units of time, and calculating the busy/waiting state distribution for each group. The time profiles show that there are definite phases to the program. The first phase is input generation which lasts about one fifth of the run and is identical in both benchmarks. It is marked by a sudden increase in the QSS PMU Pipe Wait increase in the top part of Figure 4-2, presumably caused by the overhead of copying the result vector from the generation function to the sort function.

The next phase in the QSS benchmark is the split of the entire input vector which involves a lot of branching in the PMU and scalar data movement in the DMU. This phase is identifiable in the top part of Figure 4-2 as many branch waits in the PMU and relatively high DMU utilization. As the vectors get shorter the graph gets erratic, but the lower DMU utilization is discernible. The QSV benchmark behavior in Figure 4-3 is somewhat easier to see. DMU utilization is consistently high due to the amount of vector copying. On the PMU side the amount of busy time increases as the recursion winds up and the vectors get shorter, then decreases as the recursion unwinds and the result vector gets built.

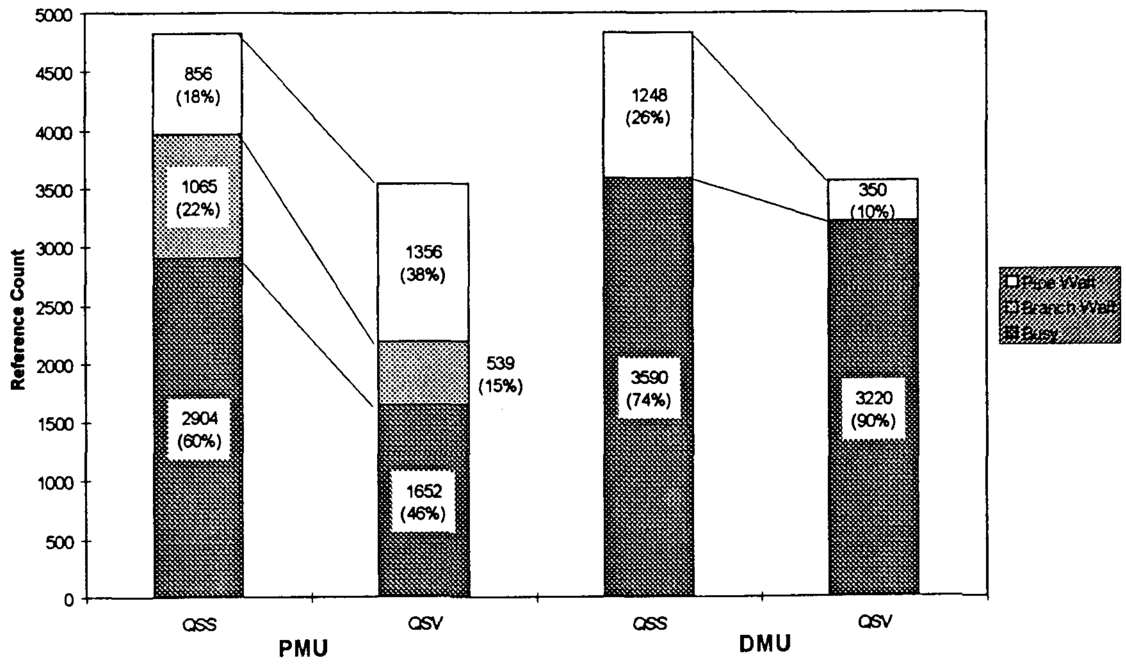
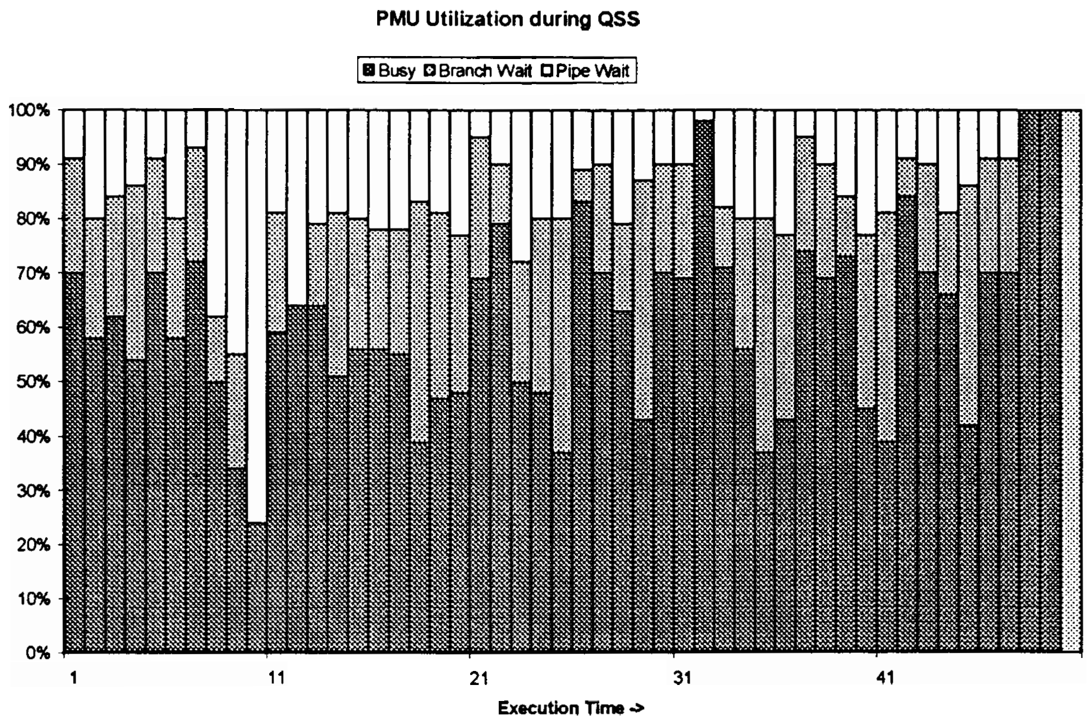


Figure 4-1: PMU and DMU utilization summary for QSS and QSV



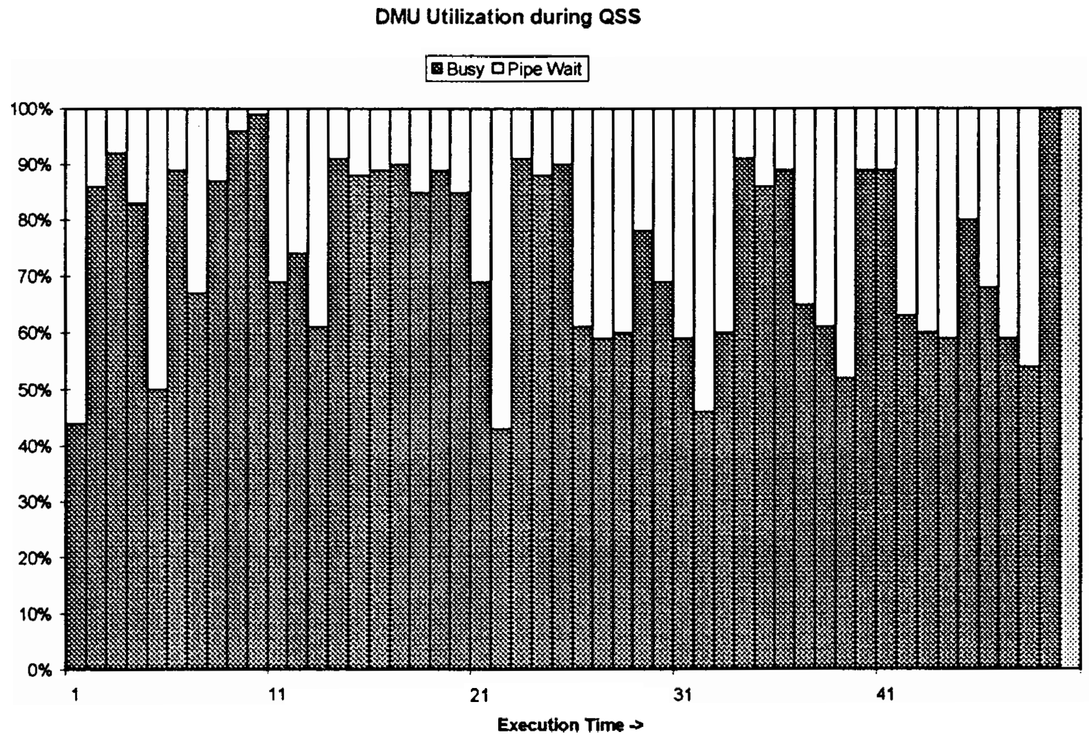
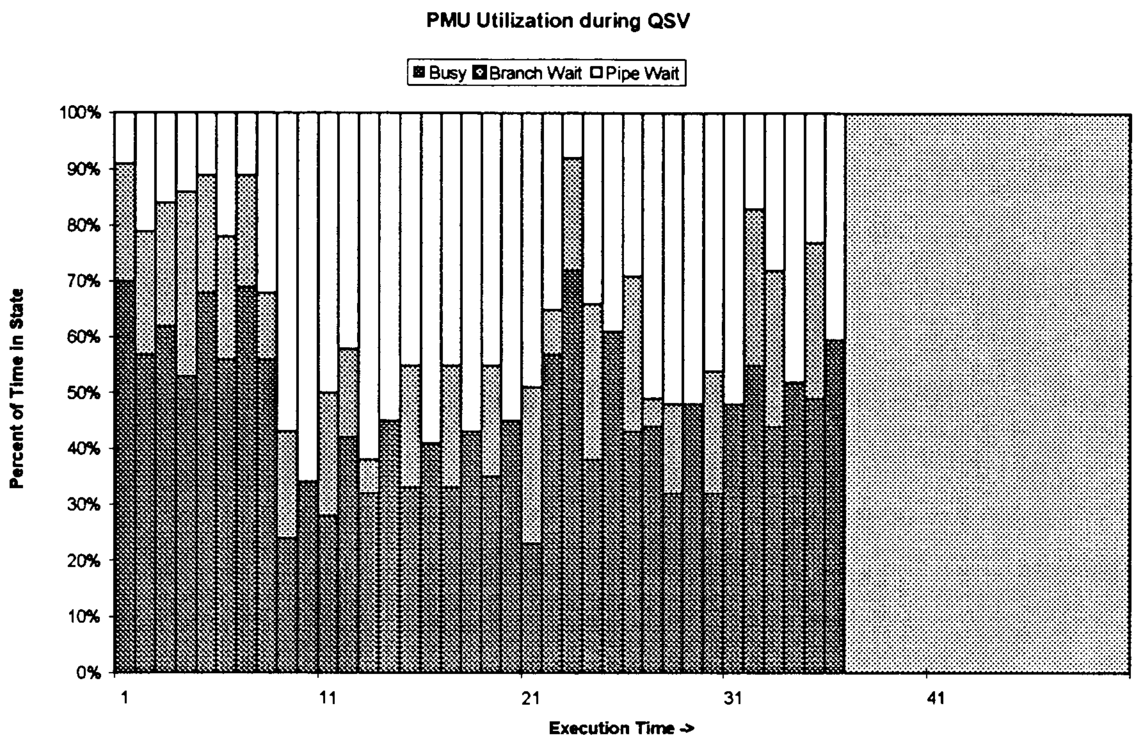


Figure 4-2: PMU and DMU Utilization Time Profile during QSS



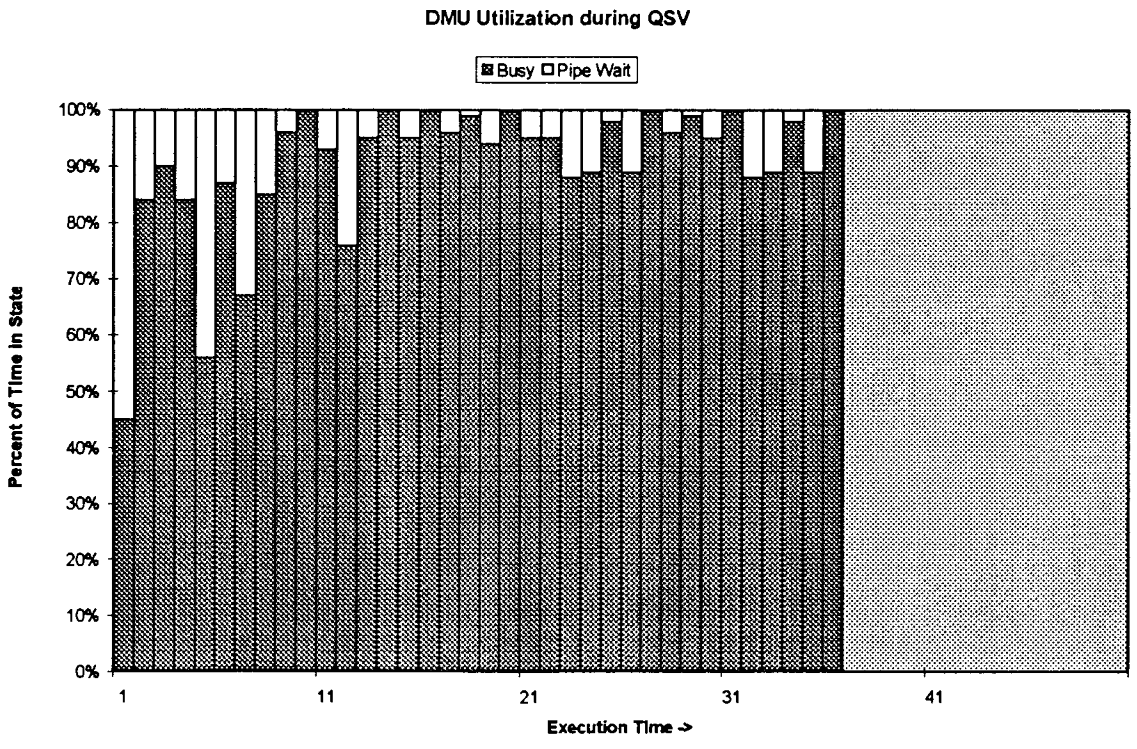


Figure 4-3: PMU and DMU Utilization Time Profile during QSV

A significant efficiency problem arises from APL's peculiar branching mechanism which uses data values as control flow targets. Since the target line of a branch statement is a data item in the DMU, the PMU must request the value from the DMU. The D4SNDSTK instruction directs the DMU to send the top of stack value to the PMU via the SJPM (Pipe) registers. The PMU must wait until the value is delivered. Since this empties the instruction pipe, a significant overhead is incurred for each branch. Figure 4-4 demonstrates the branching overhead in a time trace which shows the wait state of both units for a small period of time at the start of the benchmark execution. The PMU must first wait for a free FIFO to send the stack request, then wait for the stack value, then resume execution at the new address. The DMU is idle while it waits for the PMU to resume the instruction stream. The branching delay is clearly more significant in the scalar version of the benchmark which does far more branches than the vector version (91 vs.

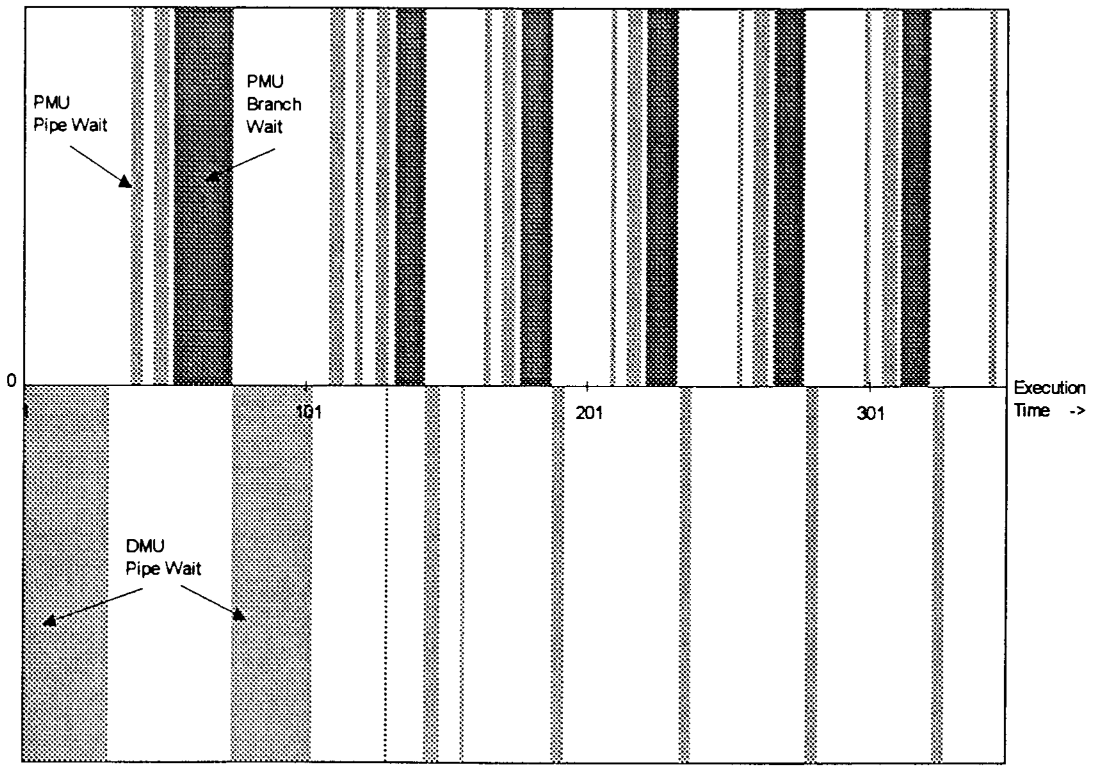


Figure 4-4: Utilz trace showing the effect of APL branching

35). The higher Branch Wait counts and lower DMU Utilization for the QSS benchmarks shown in Figure 4-1 are a symptom of this effect. Branching is always a source of stalls in pipelined computers, but the problem is particularly acute in APL since the branch instruction is so general. The same mechanism serves for unconditional branches, conditional branches, and function returns. Only conditional branches need to incur this overhead. The other two types should be treated separately by assigning ADEL formats for them. APL*PLUS III [Man94] has added structured programming constructs to APL (if-then-else, and loops) which would allow the use of branch prediction techniques.

4.2 Execution Profile

The Profile instrument is used to obtain the distribution of execution time among the subroutines of SAM APL. Profile produces a trace file of the subroutine that was executing in the PMU or DMU during periodic execution sampling. From this trace, an execution profile of the program can be constructed by subroutine and by module. Figure 4-5 shows a summary of the profile for QSS and QSV grouped by module. This figure and the subsequent time profile figures, Figure 4-6 and Figure 4-7, correspond to the utilization figures described in the previous section. The profile figures show the relative execution times of the different components of the SAM APL software. As the summary Figure 4-5 shows, QSS took longer to execute and spent more time in every module except PMU Pipe Wait and DMU Memory Management. Since the PMU looks after branching and function invocation, during QSV, the PMU has much less to do, which explains the higher waiting time. The PMU is simply waiting for the DMU to finish an instruction, so that it can load the next instruction into the pipe. The DMU is kept much busier in QSV copying intermediate vectors, which explains the higher Memory manager use. Because the operands in QSV are vectors rather than the scalars that QSS moves, the DMU executes fewer instructions, but more work is done. This is precisely the design goal of the Structured Architecture Machine which attempts to reduce interpretive overhead.

The time profiles show how work is distributed among SAM APL components over the duration of execution. As with the utilization figures, the first phase of each benchmark is the generation of the input vector. The profiles show that a significant part of the PMU's work time is spent in the Environment and Linker modules which look after function invocation. The DMU spends a large part of its time in the Data Access Table (DAT) and Memory Manager modules. During the vector benchmark, the DMU Memory Manager shows a dramatic increase in use compared to the scalar benchmark. The format routines directly account for a small portion of the execution time.

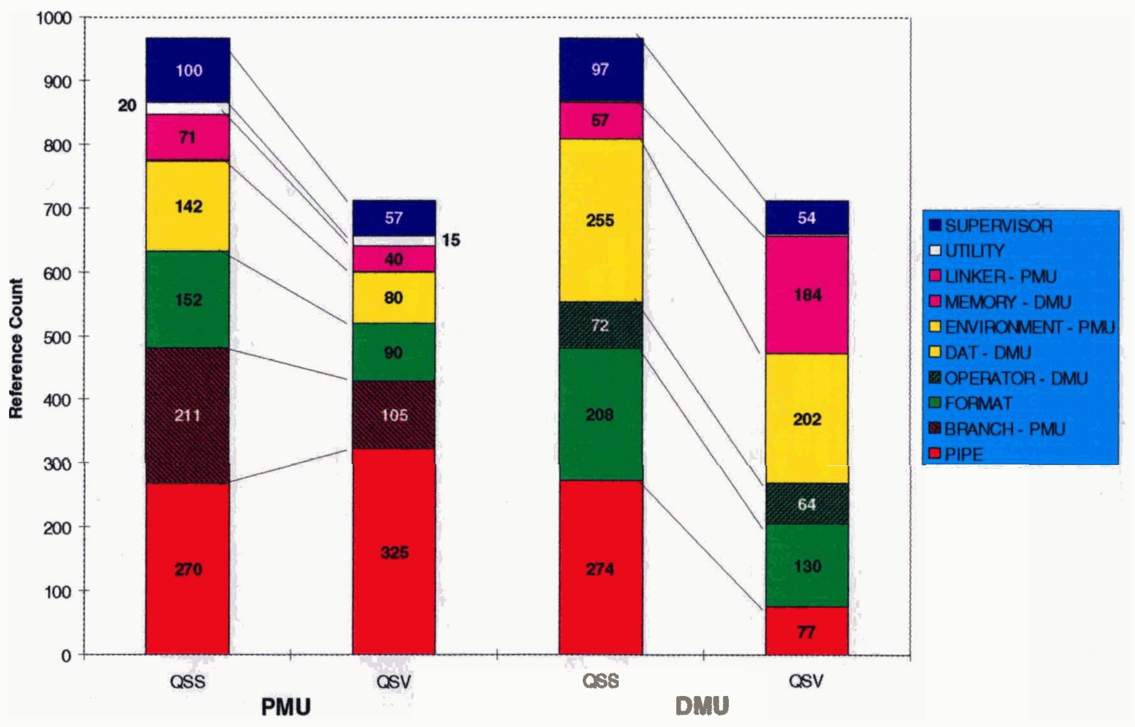
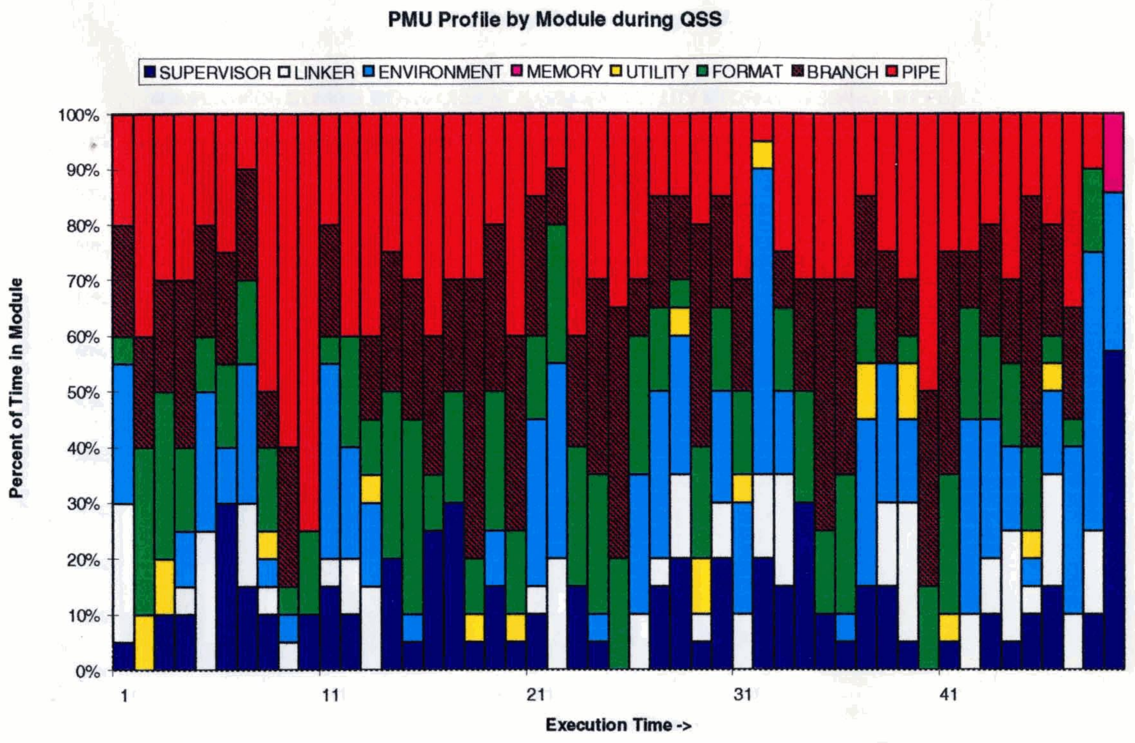


Figure 4-5: PMU and DMU Execution Profile Summary by Module for QSS and QSV



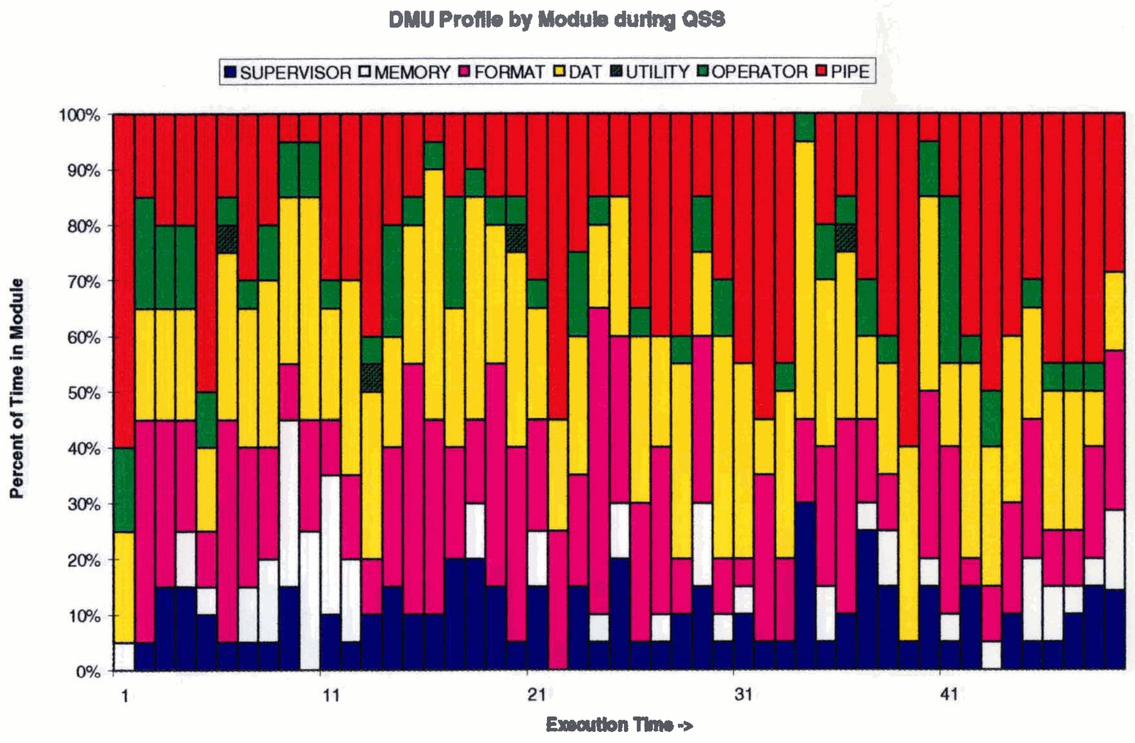
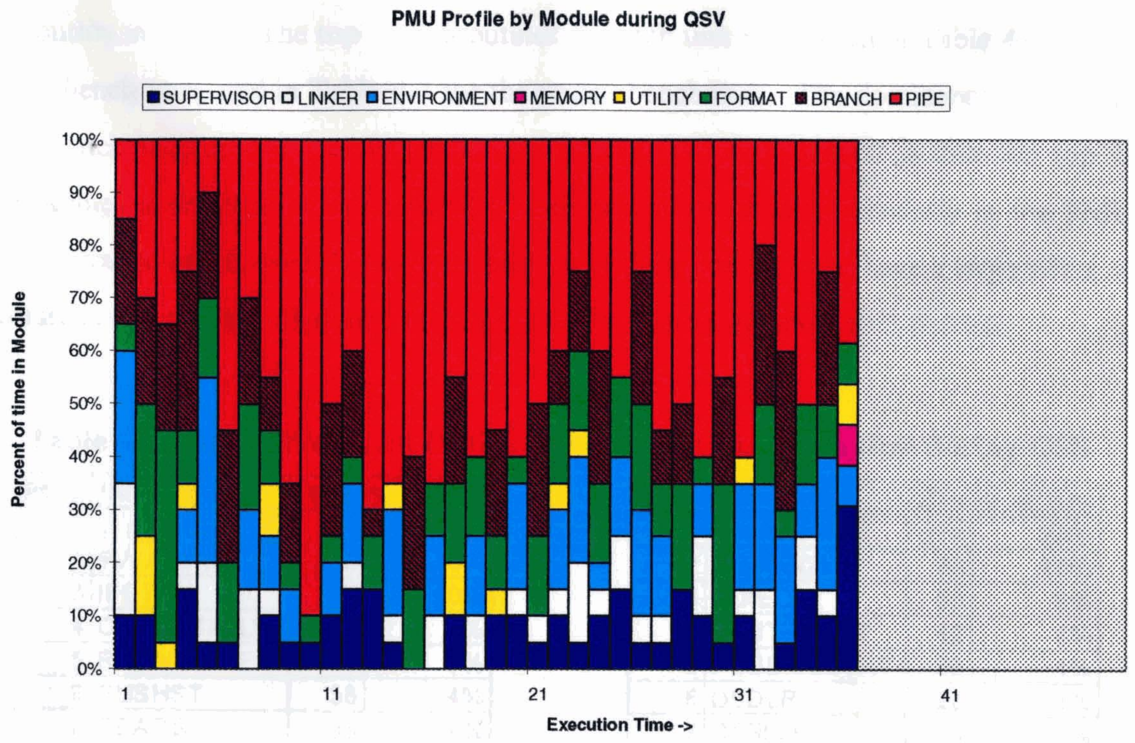


Figure 4-6: Profile of PMU and DMU during QSS benchmark



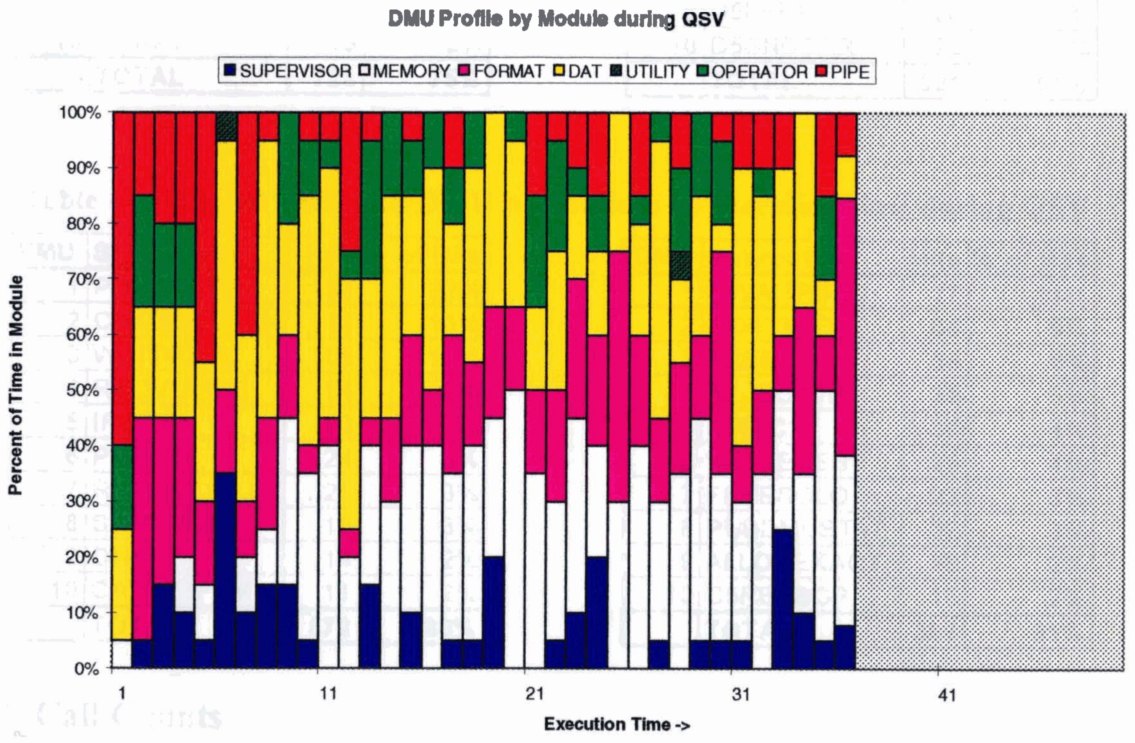


Figure 4-7: Profile of PMU and DMU during QSV benchmark

The Profile instrument gathers statistics on the relative execution times for each subroutine in a unit. The top 10 subroutines for each unit are shown in Table 4-2 for the scalar benchmark, and in Table 4-3 for the vector benchmark. The dominance of the pipe and branch wait subroutines is obvious. The next point of interest is that the top 10 subroutines account for a large portion of the execution time, particularly in the PMU. The relevance of this point is that optimization of these routines will have a large effect on overall execution time. Subroutine usage is further discussed in the next section.

Table 4-2: Top 10 PMU and DMU subroutines by execution time during QSS

PMU	Subroutine	Hits	% of Hits
1	WAIT4CLR	211	22%
2	PWAITPIPE	123	13%
3	IFETCH	97	10%
4	CHKDSTATUS	78	8%
5	RELIVUPIPE	69	7%
6	PUSHST	38	4%
7	READST	36	4%
8	STARTSTS1	35	4%
9	POPST	20	2%
10	CHKPT	19	2%
	TOTAL	**726**	**75%**

DMU	Subroutine	Hits	% of Hits
1	DWAITPIPE	274	28%
2	COPYN2D	93	10%
3	IEXEC	59	6%
4	WRITEDEST	42	4%
5	PUBDMUSTAT	38	4%
6	D5DLR	28	3%
7	D5RIGHT	25	3%
8	OVUCAST1	22	2%
9	D5LEFT	20	2%
10	D5SNDSAR	20	2%
	TOTAL	**621**	**64%**

Table 4-3: Top 10 PMU and DMU subroutines by execution time during QSV

PMU	Subroutine	Hits	% of Hits
1	PWAITPIPE	158	22%
2	CHKDSTATUS	113	16%
3	WAIT4CLR	105	15%
4	RELIVUPIPE	54	8%
5	IFETCH	53	7%
6	PUSHST	24	3%
7	READST	21	3%
8	STARTSTS1	19	3%
9	CHKPT	14	2%
10	CATGETFRM	11	2%
	TOTAL	**572**	**80%**

DMU	Subroutine	Hits	% of Hits
1	DWAITPIPE	77	11%
2	COPYN2D	66	9%
3	FREE	56	8%
4	ALLOCATE	35	5%
5	IEXEC	30	4%
6	WRITEDEST	25	4%
7	FREESOLO	23	3%
8	PUBDMUSTAT	22	3%
9	ALLOCEXACT	16	2%
10	O5REVBOP	16	2%
	TOTAL	**366**	**51%**

4.3 Call Counts

The Callvue instrument produces a Calls matrix in which the value of element Calls[i,j] is the number of times subroutine i calls subroutine j. There are many ways to use this matrix, but for the purposes of performance analysis, the most useful is to sum the rows of the transpose of Calls, producing a table of how many times each subroutine is called. When this table is sorted in decreasing order of the number of times called, a profile of subroutine invocation is obtained. The call profile is significantly different from the execution profile. For example, a routine such as IFETCH which is the main instruction processing loop in the PMU, is only called once, but executes for the duration of the run and accumulates 10% of the execution time. The top 10 subroutines for each unit for the vector benchmark are shown in Table 4-4, with the corresponding execution weight shown for comparison. RECVCONST is a very short DMU subroutine called by DWAITPIPE. Although it accounts for a large number of calls, it accumulates no execution time.

Table 4-4: Top 10 called subroutines in the PMU and DMU during QSV

PMU	Subroutine	% of Calls	% of Time
1	CHKDSTATUS	37%	16%
2	CHKPT	8%	2%
3	PWAITPIPE	6%	22%
4	RELIVUPIPE	5%	8%
5	STARTSTS1	5%	3%
6	P5RIGHT	4%	1%
7	P5NXTL	3%	1%
8	P5RSTPRG	3%	1%
9	P4NXTL	2%	1%
10	READST	2%	3%
	TOTAL	74%	58%

DMU	Subroutine	% of calls	% of Time
1	RECVCONST	17%	0%
2	PUBDMUSTAT	6%	3%
3	DWAITPIPE	5%	11%
4	COPYN2D	4%	9%
5	WRITEDEST	4%	4%
6	TRACE	4%	0%
7	D5DEST	4%	2%
8	D5RIGHT	4%	2%
9	D5LEFT	3%	2%
10	CHKB4FREE	2%	2%
	TOTAL	52%	34%

Perhaps the best use of call count information is to understand how many times particular events took place during a run. For example, since the subroutine UDFCALL is called when an APL function is invoked, the call count for UDFCALL indicates the number of APL function invocations. Similarly, the subroutine P4NXTL is called at the end of each APL line and P4BSTK for branching. Table 4-5 shows some interesting call counts for the two benchmarks. The subroutine STARTSTS1 is called to start the ADEL instruction stream from PMU Segmented Memory. It represents a significant delay which could perhaps be reduced through the use of multiple instruction streams. WRITEDEST is a DMU DAT subroutine which copies a temporary result to its destination.

Table 4-5: Call counts for various PMU and DMU subroutines for QSS and QSV

	Call Count	
Subroutine	**QSS**	**QSV**
UDFCALL	25	18
P4NXTL	122	84
P4BSTACK	73	27
P4BRABS	28	8
P4*	564	315
D4*	422	238
O4*	210	174
WRITEDEST	335	209
STARTSTS1	349	186
PWAITPIPE	422	238
DWAITPIPE	421	237

The relative frequency of the instruction formats and operators is an important clue to the work done by SAM. Since each format and operator are implemented as separate subroutines, their call counts correspond to their frequency. Tables 4-5, 4-6, and 4-7 compare the instruction and operator subroutine counts for QSS and QSV.

Table 4-6: PMU instruction format subroutine call counts for QSS and QSV

Subroutine	Calls	% of Calls
P4NXTL	122	22%
P4BSTACK	73	13%
P4SSmR	73	13%
P4SLR	61	11%
P4LSUBR	32	6%
P4BRABS	28	5%
P4DGETSS	28	5%
P4LLR	28	5%
P4SLS	22	4%
P4COMENT	17	3%
P4DGETSR	17	3%
P4SLuR	16	3%
P4DLR	12	2%
P4IXASN	12	2%
P4SLuS	9	2%
P4SmR	3	1%
P4DLmS	2	0%
P4LLmR	2	0%
P4SLmS	2	0%
P4DfS	1	0%
P4DLS	1	0%
P4DmR	1	0%
P4HALT	1	0%
P4QADOUT	1	0%
24	**564**	**11%**

Subroutine	Calls	% of Calls
P4NXTL	84	27%
P4SLR	28	9%
P4BSTACK	27	9%
P4SSmR	27	9%
P4DGETSR	15	5%
P4SLS	15	5%
P4SLuR	15	5%
P4SmR	15	5%
P4LLR	14	4%
P4DGETSS	12	4%
P4DLmR	12	4%
P4DSmR	12	4%
P4BRABS	8	3%
P4DLmS	8	3%
P4LSUBR	6	2%
P4DLS	4	1%
P4COMENT	3	1%
P4SLuS	3	1%
P4LLmR	2	1%
P4SLmS	2	1%
P4DmR	1	0%
P4HALT	1	0%
P4QADOUT	1	0%
23	**315**	**8%**

Table 4-7: DMU instruction format subroutine call counts for QSS and QSV

Subroutine	Calls	% of calls
D4DLR	123	29%
D4SNDSTK	73	17%
D4SSmR	73	17%
D4LSUBR	32	8%
D4DGETSS	28	7%
D4UDFRET	25	6%
D4DGETSR	17	4%
D4SLuR	16	4%
D4IXASN	12	3%
D4SLuS	9	2%
D4SmR	3	1%
D4DLmS	2	0%
D4LLmR	2	0%
D4SLmS	2	0%
D4DfS	1	0%
D4DLS	1	0%
D4DmR	1	0%
D4EXIT	1	0%
D4QADOUT	1	0%
19	**422**	**4%**

Subroutine	Calls	% of calls
D4DLR	57	24%
D4SNDSTK	27	11%
D4SSmR	27	11%
D4UDFRET	18	8%
D4DGETSR	15	6%
D4SLuR	15	6%
D4SmR	15	6%
D4DGETSS	12	5%
D4DLmR	12	5%
D4DSmR	12	5%
D4DLmS	8	3%
D4LSUBR	6	3%
D4DLS	4	2%
D4SLuS	3	1%
D4LLmR	2	1%
D4SLmS	2	1%
D4DmR	1	0%
D4EXIT	1	0%
D4QADOUT	1	0%
19	**238**	**5%**

Table 4-8: DMU operator subroutine call counts for QSS and QSV

Subroutine	Calls	% of calls
O4CMPRSS	73	35%
O4ADD	31	15%
O4GRTR	23	11%
O4SUBT	23	11%
O4EQUAL	16	8%
O4GREQ	13	6%
O4LESS	12	6%
O4LTEQ	9	4%
O4CATN8	6	3%
O4SHAPE	3	1%
O4MIOTA	1	0%
11	210	2%

Subroutine	Calls	% of call
O4CMPRSS	39	22%
O4GRTR	36	21%
O4LTEQ	29	17%
O4ADD	19	11%
O4CATN8	18	10%
O4SHAPE	15	9%
O4SUBT	9	5%
O4DROP	6	3%
O4EQUAL	2	1%
O4MIOTA	1	1%
10	174	4%

5. Conclusions

The goal of this work was to develop a method for observing and analyzing the behavior and performance of parallel computers. The VSAM performance analysis tool described in this thesis accomplishes this goal for the Structured Architecture Machine (SAM), a distributed-function multiprocessor computer. The SAM-1 prototype has been simulated in software and its performance on existing benchmarks was measured. The design and implementation of VSAM is described herein and the results of the benchmark measurements are presented.

A simulator-based approach to performance analysis was chosen after initial experiments with the prototype hardware showed it to be difficult to instrument and generally hard to work with. Simulation is a proven method of analyzing the behavior of a complex system. The main benefits of simulation are the degree of control that can be exercised over the simulated system, and the ease with which the system can be changed to explore the effects of proposed alterations of the system. Simulation is also an excellent platform for measurement because of the direct access provided to all internal objects and events.

The main challenge of the simulator-based approach was the specification of the complex SAM architecture in a software model. A detailed and structurally accurate model was required in order to measure the execution of benchmark programs written for the prototype. The VSAM model was written in C++ and runs under OS/2. The object-oriented nature of C++ was exploited to represent the modular structure of the SAM hardware. The multi-processing capability of OS/2 was used to partition SAM processors into separate processes. The resulting system closely resembles the structure of the hardware, and can be readily modified to explore alternative architecture configurations such as adding more processing units or extending the capabilities of the SAM components.

VSAM runs about 1000 times slower than the SAM-1 prototype. A large part of this difference is due to the multiprocessing overhead of OS/2 which could probably be significantly reduced by a better inter-process communication strategy. Late in the project, the STATUS shared memory was added in order to make available the status of various system components for instrumentation. In retrospect, the shared memory should have been a central part of VSAM both for instrumentation and general process control. In the current control scheme, the PMU and DMU processes co-ordinate execution of instructions through the VSAM process via commands passed through OS/2 pipes. This is very inefficient and unnecessarily complex. The STATUS shared memory could contain flags that achieve the same effect without the overhead of switching context to the VSAM process. As more processors and instruments are added to VSAM, the STATUS shared memory will become an important central feature.

The current user interface to VSAM is text oriented. While this is efficient from the implementation point of view, a graphical interface would better serve the VSAM user, particularly when VSAM is used for debugging SAM software. The main problem is the large number of commands a user must be familiar with, and the large amount of information that is presented to them. A graphical interface would allow users to focus on parts of the system that are changing and of direct interest to the problem at hand. VSAM makes some attempts at helping the user through the use of color to highlight changed values and these were found to be very effective. A graphical interface could also be very useful for observation of the system during execution. For example, animation could be used to indicate changes in the system, and instruments could be attached directly to elements to be monitored. The initial design of VSAM was graphically oriented, but this proved to be very difficult to implement and was abandoned as too ambitious under the circumstances.

Overall, VSAM represents a good start at an architectural modeling tool. Given the complexity of modern computer systems, such modeling tools are an essential part of the design process. Future graduate projects could expand the capabilities of VSAM as the need arises.

6. Appendix

This appendix shows the SAM APL source code for the Quicksort scalar QSS and vector QSV benchmarks. The code may look peculiar due to SAM APL limitations.

QSS is a shell which calls QSSINNER to do the work. The variable ANS is used as a global place-holder for the data being sorted.

```
∇ ANS←LEF QSS ARG; RYT
[1]   ANS←ARG
[2]   RYT←+/ρANS
[3]   LEF←1
[4]   ARG←0+LEF QSSINNER RYT
    ∇
```

QSINNER splits the vector to be sorted into two parts based on the pivot, and calls itself recursively to sort each part.

```
∇ PTR←LEF QSSINNER RYT;T
[1]   PTR←13
[2]   →(LEF≥RYT)/0
[3]   PTR←(LEF QSSPLIT RYT)
[4]   T←(LEF QSINNER PTR-2)
[5]   T←(PTR QSINNER RYT)
    ∇
```

QSSPLIT arranges the data vector ANS[LEF] to ANS[RYT] so that upon return lower values are to the left of PTR, and higher values to the right.

```
∇ PTR←LEF QSSPLIT RYT;VAL
[1]    VAL←ANS[LEF]
[2]    PTR←LEF+1
[3]    →(PTR>RYT)/8
[4]    →(VAL<ANS[PTR])/10
[5]    ANS[LEF]←ANS[PTR]
[6]    LEF←LEF+1
[7]    →2
[8]    ANS[LEF]←VAL
[9]    →0
[10]   →(PTR=RYT)/8
[11]   →(VAL>ANS[RYT])/14
[12]   RYT←RYT-1
[13]   →10
[14]   ANS[LEF]←ANS[RYT]
[15]   ANS[RYT]←ANS[PTR]
[16]   RYT←RYT-1
[17]   →6
    ∇
```

QSV is the vector version of Quicksort. It calls itself recursively twice, once with the values less than or equal to the pivot, and then with the values greater than the pivot. It then catenates the two results and the pivot in proper order.

```
∇ ANS←LEF QSV RARG
[1]    ANS←RARG
[2]    →(2>⍴RARG)/0
[3]    LEF←RARG[1]
[4]    RARG←1↓RARG
[5]    ANS←LEF,2 QSV (LEF≤RARG)/RARG
[6]    ANS←(2 QSV (LEF>RARG)/RARG),ANS
    ∇
```

QARG builds an input vector of the specified length with a balanced order.

```
∇ VEC←PIVOT QARG LEN
[1]    →(LEN>2)/5
[2]    VEC←⍳LEN
[3]    →0
[4]    PIVOT←(LEN DIV 2)
[5]    VEC←(2 QARG PIVOT-1)
[6]    VEC←PIVOT,VEC,PIVOT+VEC
[7]    →(LEN=⍴VEC)/0
[8]    VEC←VEC,LEN
    ∇
```

7. Glossary

ADEL	A Directly Executable Language
CAT	Countour Access Table
CP	Control Processor
DAT	Data Access Table
DMU	Data Management Unit
DPM	Dual Port Memory
FIFO	First In First Out
IVU	Instruction Verification Unit
OVU	Operand Verification Unit
PMU	Program Management Unit
SAM	Structured Architecture Machine
SAMjr	a unit of SAM consisting of a microprocessor and co-processor
SJ16	a custom VLSI microprocessor for SAMjr
SJMC	SAMjr Memory Controller
SJPM	SAMjr Pipe and Mail processor
SMem	Segmented Memory
TMem	Translate Memory
VSAM	Virtual SAM
VSAMjr	Virtual SAMjr

8. Bibliography

[And94] Anderson, W., "An Overview of Motorola's PowerPC Simulator Family", *Communications of the ACM*, Vol. 37, No. 6, June 1994, pp. 64-69.

[BaC84] Banks, J. and Carson, J.S., *Discreet-Event System Simulation*, Prentice-Hall, 1984.

[BuO86] Butler, J.M. and Oruc, A.Y., "A Facility for Simulating Multiprocessors", *IEEE Micro*, Oct. 1986, pp. 32-44.

[But94] Butt, F., "Rapid Development of a Source-Level Debugger for PowerPC Microprocessors", *ACM Sigplan Notices*, Vol. 29, No. 12, Dec. 1994, pp. 73-77.

[CNN89] Ching, W., Nelson, R., Shi, N., "An Empirical Study of the Performance of the APL370 Compiler", *APL 89 Conference Proceedings*, August 1989, New York, pp. 87-93.

[Fer78] Ferrari, D., *Computer Systems Performance Evaluation*, Prentice-Hall, 1978.

[Geo93] George, A.D., "Simulating Microprocessor-Based Parallel Computers Using Processor Libraries", *Simulation 60:2*, Feb. 1993, pp. 129-134.

[HeP90] Hennessy, J.L. and Patterson, D.A., *Computer Architecture A Quantitative Approach*, Morgan Kaufmann Publishers Inc., 1990.

[HHS92] Hobson, R.F., Hoskins, J., Simmons, J., Spilsbury, R., "SAM-I: a Prototype Machine for Dynamic, Array-oriented Programming Languages", *IEE Proceedings*, Vol. 139, Pt. E, No. 4, July 1992, pp. 335-347.

[HIM91] Hollingsworth, J.K., Irvin, R.B., Miller, B.P., "The Integration of Application and System Based Metrics in a Parallel Program Performance Tool", *ACM Sigplan Notices*, Vol. 26, No. 1, 1991, pp. 189-199.

[HLT87] Huguet, M., Lang, T., Tamir, Y., "A Block-and-Actions Generator as an Alternative to a Simulator for Collecting Architecture Measurements", *ACM Sigplan Notices*, Vol. 22, No. 7, 1987, pp. 14-25.

[Hob84] Hobson, R.F., "A Directly Executable Encoding for APL", *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 3, July 1984, pp. 314-332.

[Hob86] Hobson, R.F., Microprogramming Tools in an APL Environment, Technical Report, (LCCR TR 87-14), School of Computing Science, Simon Fraser University, 1986.

[Hob88] Hobson, R.F., "High-level Microprogramming Support Embedded in Silicon", *IEE Proceedings,* Vol. 135, Pt. E, No. 2, March 1988, pp. 73-81.

[Hos87] Hoskin, J., *An APL Subset Interpreter for a New Chip Set*, Master's Thesis, School of Computing Science, Simon Fraser University, 1987.

[IBM92] IBM Corp., *OS/2 2.0 Control Program Programming Guide*, Que, 1992.

[Knu72] Knuth, D.E., "An Empirical Study of FORTRAN Programs", *Software - Practice and Experience*, Feb. 1972, pp. 105-133.

[MAF91] Mills, C., Ahalt, S., Fowler, J., "Compiled Instruction Set Simulation", *Software - Practice and Experience*, Vol. 21(8), Aug. 1991, pp. 877-889.

[Man94] Manugistics, *APL*PLUS III Language Reference Manual*, 1994.

[MeM88] Melamed, B. and Morris, R.J.T., "Visual Simulation: The Performance Analysis Workstation", *Computer,* Aug. 1988, pp. 87-94.

[Mic93] Microsoft Corporation, *Microsoft Excel User's Guide Version 5.0,* 1993.

[MKO88] Miyata, M., Kishigami, H., Okamoto, K., Kamiya, S., "The TX1 32-Bit Microprocessor: Performance Analysis, and Debugging Support, *IEEE Micro,* Apr. 1988, pp. 37-46.

[Nav93] Navabi, Z., *VHDL Analysis and Modeling of Digital System,* McGraw-Hill Inc., 1993.

[Pat85] Patterson, D.A., "Reduced instruction set computers", *Communications of the ACM,* Vol. 28, No. 1, 1985, pp. 8-21.

[Pre92] Pressman, R.S., *Software Engineering A Practitioner's Approach,* Third Edition, McGraw-Hill Inc., 1992.

[Roc88] Rochkind, M.J., *Advanced C Programming for Displays*, Prentice-Hall, 1988.

[Str91] Stroustrup, B., *The C++ Programming Language*, Second Edition, Addison-Wesley Publishing Company, 1991.

[TyD93] Typaldos, M.D. and Deneau, T., "Interoperability of RISC Debugger Tools", *Computer Design.*

[ThM91] Thomas, D.E. and Moorby, P., *The Verilog Hardware Description Language*, Kluwer Academic Publishers, 1991.

[Voi94] Voith, R.P., "The PowerPC 603 C++ Verilog Interface Model", *Proceedings of Spring Compcon '94*, San Francisco,