HARP: A VLIW RISC PROCESSOR

P. A. Findlay, S. A. Trainis, G. B. Steven and R. G. Adams

Hatfield Polytechnic, Hatfield, Herts AL10 9AB, UK.

Abstract

HARP (the HAtfield Risc Processor) is a reduced instruction set processor being developed at Hatfield Polytechnic, UK. The major aim of the HARP project is to develop a VLIW (Very Long Instruction Word) RISC (Reduced Instruction Set Computer) processor capable of a sustained instruction execution rate in excess of one instruction per cycle [1, 2], by the parallel execution of RISC type instructions.

Investigations to date support our hypothesis that this goal can be achieved by the development of an integrated processor-compiler pair designed to support low level parallelism identified by the compiler. This paper describes the HARP architecture and discusses those hardware features which will support parallel instruction execution.

Parallelism is provided in the hardware by multiple instruction pipelines which execute independent RISC-like instructions simultaneously. The principal techniques employed to exploit the available parallelism are efficient pipelining, register bypassing, optional register writeback and conditional execution of instructions.

Introduction

Current RISC implementations aim to complete an instruction every machine cycle by executing simple, easily decoded instructions in a streamlined pipeline[3]. However, this level of performance represents a maximum execution rate which is not achieved in practice. MIPS-X, for example, requires an average of 1.7 cycles per instruction when all overheads are included [4].

The HARP project is one of several recent research projects which have attempted to increase the instruction execution rate by specifying several operations (short instructions) in a single instruction word (long instruction). A number of the resulting architectures, for example, VLIW [5], WM [6], WARP [7], and Cydra [8], while detecting parallelism at compile time, are specifically aimed at scientific code and therefore require a floating point unit in addition to an integer unit. In contrast HARP aims to exploit the low level parallelism available in general purpose computations.

The HARP compiler selects short RISC type instructions which can be executed in parallel and packs them into long instructions, which are fetched from an instruction cache by the processor. The component short instructions are passed to a multiple pipelined structure for execution.

HARP uses several characteristic RISC features to achieve a sequential instruction rate of one instruction per cycle, these include:

1) Simple instruction formats which permit fast hardwired decoding

2) A small number of simple instructions. Each short HARP instruction carries out a single RISC-like operation.

3) A small number of addressing modes. HARP provides two addressing modes, base register indirect plus offset and base register indirect with index, from which typical CISC (Complex Instruction Set Computer) addressing modes can be synthesised.

4) A load and store architecture. Only load and store instructions reference memory, all other instructions operate on the contents of registers.

5) Efficient instruction pipelining. The HARP model uses a four stage instruction pipeline and a two instruction branch architecture, in which the result of a test or compare instruction is stored in a register or flag, and then the result is used by a separate branch instruction. The use of this branch architecture results in a branch delay of one cycle. Register bypassing allows the result of a short instruction to be used as an operand in any immediately following instruction.

6) Optimising compilers. The HARP compiler performs classical optimisations, reorders code to minimise the effects of data dependencies, and employs an extension of the delayed branch mechanism [9] to reduce the cost of branches.

Having addressed the problem of single cycle execution, the features of HARP which allow the processor to utilise the maximum amount of low-level parallelism inherent in a program include the following:

1) Multiple instruction pipelines. The model provides multiple instruction pipelines which execute independent short instructions in parallel. Each short instruction field of a long instruction is associated with a particular pipeline.

2) Conditional execution of short instructions. All short instructions may be conditionally executed. Conditional execution reduces the number of short branches in a program, thus increasing the average length of a basic block and hence the potential for parallelism. It also increases the number of instructions which can be placed in branch delay slots.

3) Optional register writeback. All computational and load instructions optionally change the contents of their specified result register, thus allowing the compiler to detect and prohibit unnecessary writes to the general purpose register file. This feature increases the potential for parallelism which is limited in implementation by a restriction on the number of parallel writes to the general purpose register file.

Instruction Set

The HARP instruction set has five types of short instruction: computational, relational, memory reference, Boolean and branch. All short instructions are 32-bits long. The short instruction format is shown in figure 1. All computational and relational instructions operate on signed and unsigned integers, and will evaluate to the following expression:

destination := *source1* op *source2*

where *source1* and *source2* are the values held by two of the general purpose registers. The *destination* is either a general purpose register (R0-R31, where R0 always holds zero and is read only) or, in the case of a relational instruction, a boolean register (B0-B7, where B0 always holds zero and is read only). Boolean instructions operate on the contents of the Boolean registers, and have the same format as the computational and relational instructions.

Data can only be transferred between memory and CPU registers by employing either a load or a store instruction. The format of the memory reference instructions is as follows:

```
   LOAD destination,offset(source1)      STORE offset(source1),source3
or LOAD destination,(source1,source2)    STORE (source1,source2),source3
```

where *source3* is the contents of a general purpose or Boolean register. The address for such an instruction is computed by ORing *source1* with *source2* or an *offset* [10]. Load and store instructions can transfer either 32-bit words or bytes. Load Boolean and store Boolean instructions are used to transfer Boolean values.

A branch instruction tests a specified Boolean register (*Bsrc1*). The format of a branch instruction is:

OP *Bsrc1,label*

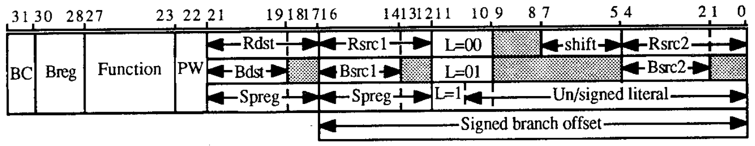
Where OP can either be branch if *Bsrc1* is true or branch if *Bsrc1* is false. Testing B0 will always or never produce a branch.

Conditional Execution of Short Instructions

Short HARP instructions, including conditional branches, can be conditionally executed. The execution of a specified short instruction will depend on the value held by a specified Boolean register, e.g.

TB1 ADD R1,R2,R3

If Boolean register B1 is true (logic 1), then add the contents of registers R2 and R3, and store the result in register R1. Unconditional execution is achieved by omitting the condition.and is in reality implemented by testing B0.



Key

BC	Boolean for conditional execution
Breg	Boolean reg. to test for conditional execution
Function	Instruction function code
PW	Permit write-back to register file
Rdst/ Bdst	General purpose/Boolean destination register
Rsrc1/ Bsrc1	General purpose/Boolean register containing 1st operand

Rsrc2/ Bsrc2		General purpose/Boolean register containing 2nd operand
L=	00	Inter-register format
	01	Long-literal format (32-bits)
	1X	Short-literal format
Shift		Shift control code
Spreg		Special purpose register. Used for load and store special purpose register
▨		Bits not used in some formats

Figure 1 : HARP Short Instruction Format

The following example illustrates how conditional execution can reduce the number of short branches, hence increasing the potential for parallelism within a basic block. Consider the following program fragment:

```
IF      R1 > R2
THEN    R1 := R1 - R3
ELSE    R2 := R2 + R3
(* End IF *)
R3 := R4
R4 := R4 + 2
```

Translated into unconditional sequential HARP code :

```
       GT   B1,R1,R2    (* B1 holds Boolean result *)
       BF   B1,else
       NOP              (* No operation *)
       SUB  R1,R1,R3    (* THEN *)
       BF   B0,out      (* Branch always to "out" *)
       NOP              (* No operation *)
else   ADD  R2,R2,R3    (* ELSE *)
out    MOV  R3,R4       (* R3 := R4 *)
       ADD  R4,R4,#2    (* R4 := R4 + 2 *)
```

Translated into conditionally executed sequential HARP code :

```
       GT   B1,R1,R2    (* B1 holds Boolean result *)
TB1    SUB  R1,R1,R3
FB1    ADD  R2,R2,R3
       MOV  R3,R4       (* R3 := R4 *)
       ADD  R4,R4,#2    (* R4 := R4 + 2 *)
```

Translated into conditionally executed parallel HARP code :

```
       GT   B1,R1,R2
TB1    SUB  R1,R1,R3; FB1   ADD  R2,R2,R3; MOV R3,R4
       ADD  R4,R4,#2
```

Implementation Issues

HARP CPU Architectural Model

An ELLA behavioural simulation [11] for the HARP architectural model [12] has been produced. This simulation is being used to assess the validity of compiled code and the effectiveness of the novel architectural features. The ELLA simulation provided the means to test and evaluate a number of HARP implementations derived from an abstract HARP architectural model with a variable number of pipelines and pipeline functions. This process has revealed a number of architectural flaws, which were resolved before the hardware specification was defined [13]. The VLSI integrated version of HARP (iHARP) is a subset of the HARP abstract model, and is being designed using Seattle Silicon ChipCrafter [14] software, which provides a relatively high level of design abstraction through the use of data-path components, compiled cells and logic synthesis.

iHARP

The iHARP CPU architecture is shown in figure 2 and contains a number of major blocks:

1) Four parallel pipelines.
2) Boolean unit.
3) Two memory address calculation units.
4) Branch unit.

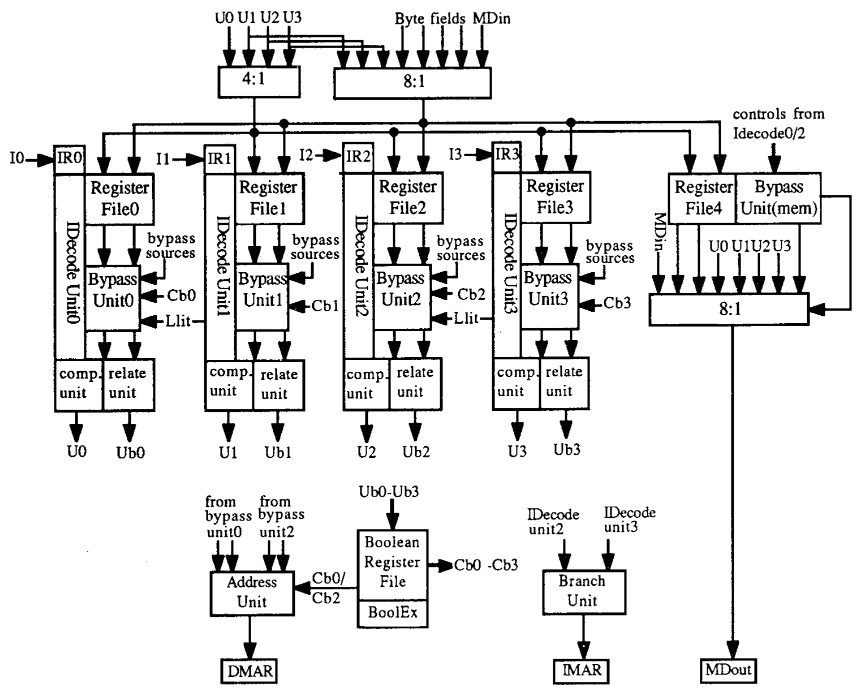
The system is provided with data and instructions from separate external caches. The data port has a 26-bit address bus and a 32-bit data bus,which implies a maximum main memory of 256Mbytes. The instruction port has a 24-bit address bus and a 64-bit multiplexed data bus. Since the instruction port is read only it can operate at approximately twice the speed of the bi-directional data port and so it is possible to read a long instruction in two 64-bit blocks. This also substantially reduces the number of I/O pins, thus easing packaging constraints. The general purpose register file has 32 32-bit registers, with R0 holding the read-only value zero. The Boolean register file contains eight 1-bit registers, with B0 holding the read-only value zero.

The four computational units operate in parallel and apply standard 32-bit arithmetic and logic operations to data from the general purpose register file. Literal values may also provide the computational units with data directly from the instruction register. Boolean values are generated by integer comparisons within each pipeline or by standard logical operations computed in a separate Boolean Unit attached to pipeline two. Values stored in the Boolean Register File are used to control the conditional execution of short instructions.

A long instruction may contain a maximum of two branch instructions which are usually executed on mutually exclusive conditions. However, if two branch instructions are scheduled to execute on the same condition, then pipeline three always takes priority. The branch unit will compute the next sequential long instruction address and any branch targets. The appropriate next long instruction address is selected by testing the appropriate Boolean register(s). Similarly, up to two memory reference instructions can be placed in the same long instruction, providing they are always executed on mutually exclusive Boolean conditions, since there is only one data memory port. The address unit allows parallel computation of two data addresses using values from the general purpose register file and/or immediate values from the instruction register

The table below summarises how iHARP internal functions are allocated to the four pipelines:

	Pipeline 0	Pipeline 1	Pipeline 2	Pipeline 3
1	Computational	Computational	Computational	Computational
2	Relational	Relational	Relational	Relational
3	Memory ref.	-	Memory ref.	-
4	-	-	Boolean	-
5	-	-	Branch	Branch
6	-	-	Special purpose	-
L	-	32-bit literal	-	32-bit literal



Key To Principal Signals and Registers

U0 - U3	Computational results
Ub0 - Ub3	Relational results
Cb0 - Cb3	Conditional execution Boolean
I0 - I3	Short instruction no.
Llit	32-bit long literal
IMAR	Instruction memory address register

DMAR	Data memory address register
MDin	Memory data input
MDout	Memory data output
bypass sources	U0-U3, Literals, MDin

Figure 2: HARP CPU Architecture

Pipelines zero and two are able to use the 32-bit literal from the succeeding pipeline.

The iHARP instruction pipeline has been developed from the MIPS-X pipeline [15,16]. iHARP has a 4-stage pipeline [17], which operates as follows:

IF	Fetch long instruction word from instruction cache.
RF	Instruction decode. Fetch registers from general purpose register file and Boolean register file. Calculate branch addresses in branch unit. Calculate memory addresses in address units.
ALU/MEM	ALU operation for computational or relational instructions. Calculate Boolean result in Boolean unit. Wait for data from memory for a load instruction. Write Boolean or relational result to the Boolean register file. Output data for a store instruction.
WB	Write computational or relational instruction result to general purpose register file. Write Boolean load instruction result to Boolean register file. Write data load instruction result to general purpose register file.

Figure 3 shows the timing of the Boolean register file reads and writes. The IF cycle has been omitted for clarity, and each cycle of the instruction pipeline is divided into two phases. All Boolean reads (conditional execution Boolean, store data and Boolean sources) occur in phase two of the RF cycle. The timing of Boolean data writes is dependent on the instruction being executed. Boolean values are not bypassable.

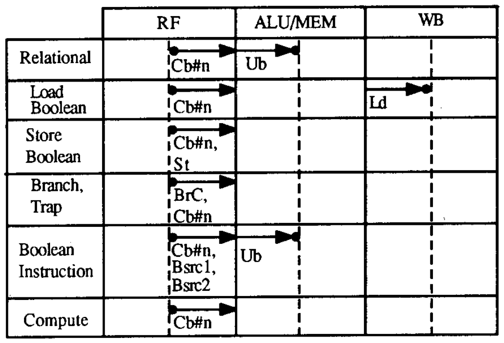
Figure 4 shows the timing of the general purpose register file reads and writes. This diagram shows when data can be bypassed to an instruction and when bypassed data can be accepted from an instruction.

General Purpose Register Files

The silicon compiler provides a RAM block which will allow two parallel reads or two parallel writes. These read/write restrictions could result in a bottleneck, as the register file may need to provide several operands to different pipelines. The problem of multiple reads can be resolved by replicating the general purpose register file, i.e. four identical register files [18], each with two read ports. The problem of multiple writes can be resolved by ensuring that each long instruction generates a limited number of parallel writes. This limiting of parallel writes, is based on two observations.

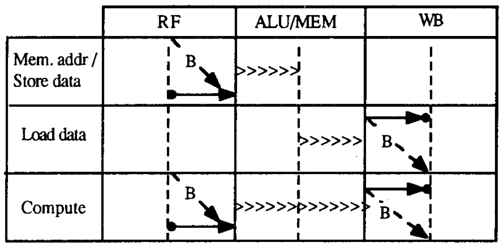
1) A significant number of iHARP instructions e.g. relational, store, branch and Boolean instructions do not generate a write to the register file, and a number of instructions only produce one write-back at run time, as the compiler frequently generates mutually exclusive execution conditions.

2) iHARP uses register bypassing to remove the potential delays caused by computational and load instructions. Data computed or loaded in the ALU/MEM stage of the short instruction, is then written back to the destination register in phase one of the WB stage. This data cannot be used until phase two of the RF stage of the next instruction. The delay can be removed by preventing the initial computational result from being written to the register file, and passing it directly to the ALU/MEM stage of the



Key	
Cb#n	Condtional execution boolean
Ub	Boolean result
St	Store Boolean data
Ld	Load Boolean data
BrC	Branch condition
Bsrc1	Boolean source 1
Bsrc2	Boolean source 2
●—▶	Read data
—▶●	Write data

Figure 3 : Boolean Data Read / Write Timing



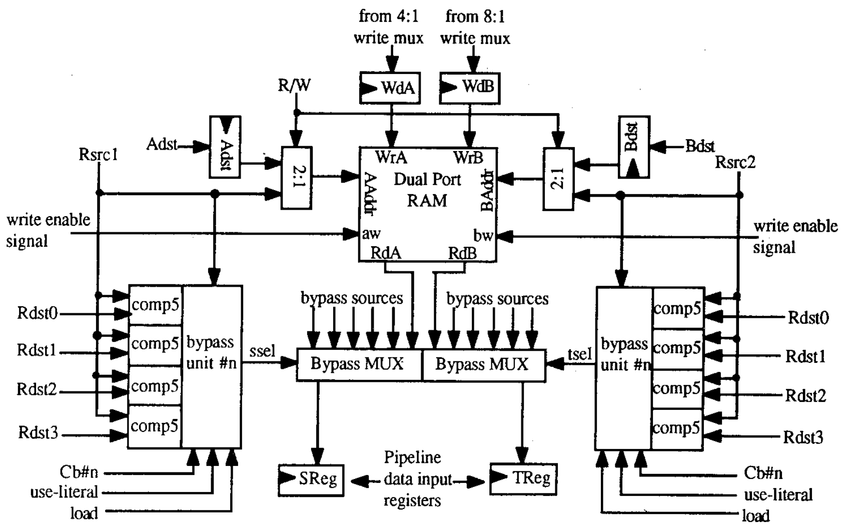
Key	
B, ●—▶	Read register file or accept bypassed data
—▶●, B'	Write to register file or bypass result to another pipeline
>>>>	Compute result

Figure 4 : General Purpose Register Read/Write Timing

next instruction. Many RISC ALU operands are obtained via the register bypass logic. If such operands are redundant after the current instruction, there is no need to write them back, although they are allocated register addresses. The only restriction to this operation is during exception processing, where iHARP enters a special state that writes-back all of the pipeline registers to the allocated addresses in two cycles.

Figure 5 illustrates the architecture of one instance of the fully bypassable register file [13]. The registers WdA and WdB contain data to be written to the register file. This data is selected from a number of sources by two write-back multiplexers (top of figure 2).

In order to determine whether data should be read from the register file or bypassed from another pipeline, each of the four current destination addresses (Rdst0 - Rdst3) are compared with the two possible read addresses (Rsrc1 and Rsrc2). Permit bypass controls generated in the instruction decode units are used



Key To Signals and Registers

Rdst0 - Rdst3	Destination regs.
Rsrc1 - Rsrc2	Source regs.
WrA - WrB	Write ports
RdA - RdB	Read ports
AAddr - BAddr	Address ports
aw - bw	Write signals
Adst - Bdst	Selected dest. regs.
WdA - WdB	Write data regs.

Cb#n	Conditional execution boolean
use-literal	Use short or long literal data
load	Use MDin data
bypass sources	U0 - U3, Literals, MDin

Note : #n = 0 to 3

Figure 5 : Fully Bypassable Register File

to qualify the results of the five bit address comparisons. These qualified comparisons produce eight bypass signals that are used by the bypass control blocks. These blocks generate the signals ssel and tsel which select data that bypasses the register file. Possible bypass sources are pipeline results , memory data, long and short literals. If there is no bypass match for either Sreg or Treg sources, then data read from the register file is passed to the pipeline data input registers.

Summary

This paper has described the iHARP hardware architecture with particular reference to those features which support parallel pipelined instruction execution. Notable features of iHARP which significantly increase performance are:

1) RISC like instruction set.
2) Compact four-stage pipeline.
3) An ORed indexed addressing mechanism.
4) Load and store architecture with simple addressing modes.
5) Optional conditional execution of all short instructions.
6) Complete register bypassing.

The success of the project is dependent on how successfully the compiler schedules instructions in parallel. Short benchmark tests indicate that instruction scheduling will reduce the instruction count of sequential iHARP code by 55 %. This performance requires approximately two short instructions scheduled in every long instruction. Such results suggest that an iHARP chip will be capable of achieving a sustained execution rate in excess of one instruction per cycle.

Preliminary characterisation of pipeline one (the least complex pipeline), including its' general purpose register file, indicates an area of less than 17 sq. mm., with an approximate speed of 10MHz. The design database is complete, and layout, timing analysis and simulation will begin in the near future. It is hoped that an iHARP prototype chip will undergo tests in 1992.

Acknowledgements

The HARP project is supported by UK S.E.R.C Research Grant GR/F 88018. Part of this work was also supported by a S.E.R.C Research Studentship. The authors gratefully acknowledge the contributions of the other members of the HARP team. In particular they would like to thank Dr. Brian Johnson for his help with the design of the iHARP chip.

References

[1] R. G. Adams, S. M. Gray and G.B Steven, "Utilising Low Level Parallelism in General Purpose Code: The HARP Project", Microprocessing and Microprogramming, vol. 29, no. 3, pp. 137-149, October 1990.

[2] G.B Steven, S. M. Gray and R. G. Adams, "HARP: A Parallel Pipelined RISC Processor", Microprocessors and Microsystems, vol. 13, no. 9, pp. 579-587, October 1989.

[3] D. A. Patterson, "Reduced instruction set computers," Comm. ACM , vol. 28, no. 1, pp. 8-21, January 1985.

[4] P. Chow and M. Horowitz, "Architectural tradeoffs in the design of MIPS-X", Proceedings of 14th Annual International Symposium on Computer Architecture, June 1987, pp. 300-308.

[5] J. A. Fisher, "Very Long Instruction Word Architectures and the ELI-512", 10th Annual International Symposium on Computer Architecture, June 1983, pp. 140-150.

[6] W. A. Wulf, The WM Computer Architecture Definition and Rationale, Technical Report, Computer Science Department, University of Virginia, Charlottesville, USA, 1988.

[7] R. Cohn, T. Gross, M. Lam and P. S. Tseng, "Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor", ASPLOS III, Boston, April 1989, pp. 2-14.

[8] B. R. Rau, D. W. L. Yen, W. Yen and R. A. Towle, "The Cydra-5 Departmental Supercomputer: Design Philosophies, Decisions and Trade-offs", Computer, January 1989, pp. 12-25

[9] T. R. Gross and J. L. Hennessy, "Optimising Delayed Branches" Proceedings of MICRO-15 15th Annual Workshop on Microprogramming, October 1982, pp 114-120.

[10] G. B. Steven, "A novel effective address calculation mechanism for RISC microprocessors", ACM Computer Architecture News, vol. 16 no. 4, pp. 150-156, September 1988.

[11] G. J. Green, A Simulation of a HARP Architecture in ELLA, Computer Science Technical Report no. TR-104, School of Information Sciences, Hatfield Polytechnic, Hatfield, UK, July 1990.

[12] G. B. Steven and S. M. Gray, Specification of a Machine Model for the HARP Architecture and Instruction Set - Version 3, Computer Science Technical Report no. TR-117, School of Information Sciences, Hatfield Polytechnic, Hatfield, UK, January 1991.

[13] S. A. Trainis, A Device Specification for the iHARP Processor, Computer Science Technical Report no. TR-118, School of Information Sciences, Hatfield Polytechnic, Hatfield, UK, February 1991.

[14] Seattle Silicon Corporation, ChipCrafter Designers Handbook, 1990.

[15] P. Chow, MIPS-X Instruction Set and Programmer's Manual Technical Report no. CSL-86-289, Computer Systems Laboratory, Stanford University, CA, USA, May 1986.

[16] M. Horowitz, P. Chow, D. Stark, R. T. Simoni, A. Salz, S. Przybylski, J. L. Hennessy, G. Gulak, A. Agarwal and J. M. Acken, "MIPS-X: A 20-MIPS peak, 32-bit microprocessor with on-chip cache", IEEE Journal of Solid State Circuits, vol. SC-22 no. 5, pp 790-799, October 1987.

[17] S. M. Gray, Considerations in the Design of an Instruction Pipeline for a Reduced Instruction Set Computer, Computer Science Technical Report no. TR-83, School of Information Sciences, Hatfield Polytechnic, Hatfield, UK September 1988.

[18] J. Labrousse and G. A. Slavenburg, "CREATE-LIFE : A Modular Design Approach for High Performance ASIC's", Digest of Papers COMPCON 1990, IEEE Computer Society Press, February 1990, pp. 427-433.