TECHNICAL REPORT NO. 271

Inference Supercomputers

by

J. W. Mills, M. Burroughs, R. Wehrmeister, and D. Winkel

February 1989

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY

Bloomington, Indiana 47405-4101

Inference Supercomputers

by

J. Mills, M. Burroughs, R. Wehrmeister, and D. Winkel
Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 271

Inference Supercomputers

by

J. Mills, M. Burroughs,
R. Wehrmeister, and D. Winkel

February 1989

Inference Supercomputers

J. Mills, M. Burroughs, R. Wehrmeister, and D. Winkel
Computer Science Department
Indiana University
Bloomington, Indiana 47405-4101

Abstract

Inference supercomputers have received less attention from computer architects recently because relatively few supercomputer users require Lisp and Prolog, interest has grown in artificial neural networks, and the Japanese Fifth Generation project has encountered difficulties attaining its goals. Yet several important classes of problems can be solved using non-numeric algorithms such as unification, and would benefit from the development of powerful inference architectures. We suggest that attaching simple inference coprocessors to nodes of existing parallel computers is the most effective way to construct an inference supercomputer. The LIBRA processor being built at Indiana University is proposed as the prototype for such a node.

1. INTRODUCTION

The need for numeric computing which drove the development of early computers, and the increasing demands which led to their successors, spawned a class of supercomputers that are not well-suited as inference engines. The current problems with inference supercomputing are typified by automated theorem provers: at least one small class of problems (geometry theorems) is solvable both numerically and by using resolution, but the solutions are found approximately four orders of magnitude more slowly using resolution. The problem instances are identical, demonstrating the performance inequality between inference and numeric computing. Trainable architectures for pattern recognition (neural networks) offer one solution to certain types of inference, particularly when the problem may not be formulated precisely, but whether neural networks can be evolved as high performance symbolic computers is yet unknown. Just as there exist problems that can be well-stated numerically, and which no one would initially attempt to solve with a neural network (weather prediction, weapons simulations), so there exist well-formed classes of problems that can be solved algorithmically — but the algorithms are not numeric. Real-time expert systems, symbolic mathematics, theorem proving, logic programming, and scheduling problems that are not amenable to linear programming (job shop scheduling) all fit into this category. We suggest that the numeric supercomputer, the inference supercomputer, and the neural network represent architectures that are best suited for specific problem classes: as such, each architecture merits independent study.

2. WHAT IS INFERENCE SUPERCOMPUTING?

Although no inference supercomputers have yet been built, architectures comparable to one node in the proposed inference supercomputer have been or are being built in Europe, Japan and the United States. Although the inference supercomputer is just as much a von Neumann machine as a numeric supercomputer, the uses to which they are put place different demands on the hardware.

At a very basic level two computing paradigms exist:

Numeric computing performs arithmetic on numbers to calculate a result.

Inference computing defines relationships between objects to reach a conclusion.

Numeric computing is more constrained than inference computing, in the sense that the objects manipulated are less abstract (although their *meaning* may be as abstract as the problem requires, it does not affect the fact that the calculations are limited to numbers or vectors and arrays of numbers). The constraints necessary for numeric processing are relatively well-understood, and their study has produced algorithms and architectures for numeric, vector, and array processing. However, the algorithms and architectures for inference processing have only recently begun to emerge.

Current architectures for inference computing are derived from seminal work done by David Moon (the Symbolics 3600), and David H.D. Warren (the Warren abstract Prolog machine). Yet even these architectures have not been the result of rigorous evaluation of the needs of complex and powerful inference engines, but rather the demands of specific programming languages: Lisp for the Symbolics 3600, and Prolog for the Warren abstract machine (WAM). The architectural requirements of resolution, reduction, term-rewriting systems are more *global* than the requirements of a language. Consider this analogy: FORTRAN programs can be compiled to a single Motorola 68030 processor, but FORTRAN programs to solve weather prediction, ray tracing, and large systems of partial differential equations have requirements of scale that prevent their solution on the 68030. Having a hardware multiplier built into a single arithmetic logic unit is a far different thing than having an attached array processor. Similar requirements of scale turn inference computing into inference supercomputing: they must be isolated, identified and understood before an inference supercomputer can be built.

2.1 INFERENCE SUPERCOMPUTING IS MORE FUNDAMENTAL THAN KNOWLEDGE INFORMATION PROCESSING

Inference supercomputing is much less ambitious than knowledge information processing as defined in the Japanese Fifth Generation Computing effort (Moto-oka 1982). Inference supercomputing does not purport to give a solution to difficult problems in natural language recognition, VLSI design and machine intelligence. These problems may benefit from the availability of an inference supercomputer, but will require a much better understanding of the problem domain before solution is possible: for example, Prolog implementations today run at from 50,000 to 200,000 LIPS, yet computers still do not understand language, VLSI design has not been totally automated, and computers are not intelligent.

A faster inference engine cannot reduce the worst-case complexity of any NP-hard problem, but it *can* improve the ability to find solutions to those problems that are computationally feasible (as many such instances of NP-hard problems are). To use an automated theorem prover as an example, an increase in execution speed of 10^5 could reduce the time neceLIBRAry to find a proof from 2 hours 45 minutes to less than 1 second. This suggests the true importance of fast inference processing: the improvement of existing applications which can be expected if faster inference engines are available.

2.2 DESIGNING A PARALLEL INFERENCE SUPERCOMPUTER

No matter how well a programming system is understood, designing an architecture for that system will highlight incorrect design assumptions and less well-understood interactions within the system. Our proposed inference supercomputer is based on the design of two RISC architectures for logic programming which have influenced researchers in this area in the United States and Europe.

The LOW RISC and the LIBRA architectures are seminal designs that have taken the highly successful RISC methodology, applied it to logic programming, and identified principles that allow them to execute Prolog faster than any other architecture. The LOW RISC is being modified by Prasenjit Biswas of Southern Methodist University for use as a processor node in a parallel Prolog machine. The DLM, designed by Frank McCabe of Imperial College and built by British Aerospace, has many features derived from the LOW RISC and some similar to the LIBRA. The performance of the DLM is at the lower range of the estimated LIBRA performance, indicating that the higher estimates for performance of the LIBRA design (which includes architectural support for logic programming not present in the DLM) are reasonable.

3. Potential performance of inference supercomputers

The performance of symbolic supercomputers varies with the architecture of the machine, its degree of parallelism, and the implementation technology. In Figure 1 the estimated performance of five approaches to a symbolic supercomputer are compared. From this comparison, the approach most likely to produce an inference supercomputer is to extend the LIBRA architecture as a loosely coupled multiprocessor, implementing it in ECL or GaAs that have a short switching time, and from which a cache and at least the first level of main memory can be constructed.

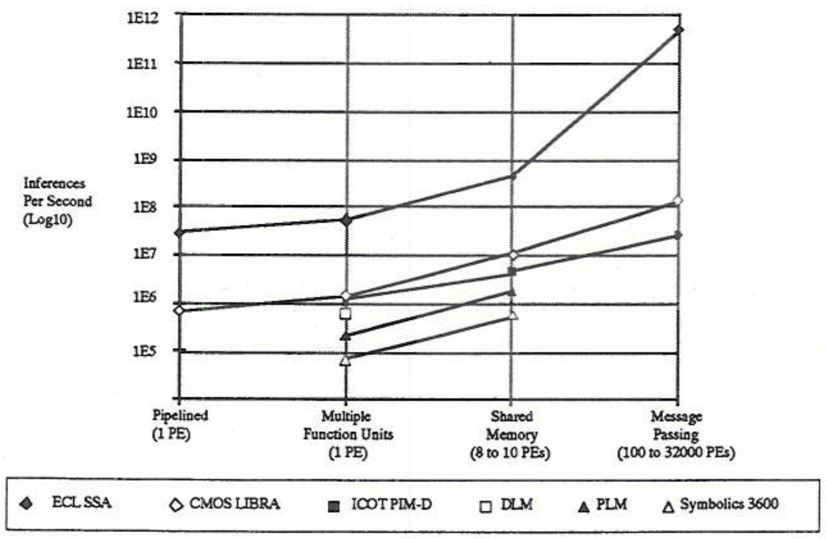

Figure 1. Estimated performance of various inference supercomputers

4. LIBRA: THE BASIC INFERENCE ARCHITECTURE

4.1 DESCRIPTION

A brief list of the LIBRA's features follows:

a. Multiple individual memories. Code, heap/stack, cache, and trail memories are described in [3].

b. 40-bit tagged architecture, with 3-bit inference type, 2-bit mark & reverse garbage collect flags, and a 35-bit value which can be further typed for use with 32-bit coprocessors [1, 2].

c. Independent next address calculation. The instruction address

d. Partial unification

e. Tag-controlled caching

f. Bounds check for trail checking.

g. "Sticky overflow" detection: bounds checking before stacks collide.

h. a "fail-trail" ALU to off load the burden of backtracking and trailing, moving this outside the cache to increase the cache hit rate.

Important Features of the LIBRA Architecture

Because a RISC represents the essence of the operational semantics of a source language, its performance will depend on the distillation of those semantics into the architecture. For example, the RISC II is not adequate for Smalltalk and Lisp: instead, the SOAR was designed for Smalltalk-80 (Ungar 1986) and the SPUR for Lisp (Taylor et. al. 1986). The development of two language specific architectures confirms Tick and Warren's objections to a general purpose RISC for Prolog: the RISC II architecture cannot efficiently perform some operations that Prolog requires. But these objections do not hold for a language specific RISC. Much of this research consisted of creating a RISC architecture tailored to Prolog. In one sense this dissertation describes "just another RISC", but in another sense it does not: to solve the problems of branch frequency and code density five concepts from computer architecture were extended to tagged architectures, introducing design techniques that could be used to improve the symbolic processing capability of general purpose computer architectures:

1. Generic instructions, such as a single add for integer, long integer and floating point data, are generalized to replaceable instructions, instructions whose operation consists of any non-replaceable instruction from the architecture's instruction set as determined by the type of its operands. The LIBRA architecture includes a single-cycle partial unify replaceable instruction that may perform either a nop, a store, a call or a branch, thus condensing three to five tag checking instructions into one. The partial unify instruction can eliminate as many as 30% of the subroutine calls performed by a general purpose RISC running Prolog.

2. Microcode addressing is typically provided or modified by flags, fields in the microinstruction, and a microprogram counter. A new technique is introduced in the LIBRA: concatenated-tag microcode addressing. Using this technique, the microcode is viewed as a two-dimensional array of control words. The two tags saved from the previous instruction's operands are used to index the microcode array, and select the control word for a replaceable instruction.

3. Cache data management typically deals with what is removed from a cache. But some data, particularly pointer chains, can cause a cache to be flushed unneceLIBRArily. Tag-controlled caching is a technique that prevents transient data, such as intermediate elements in a pointer chain, from entering a cache, increasing the stability of other data in the cache.

4. Scoreboarding allows the efficient use of a machine resource after all neceLIBRAry conditions are met. Generalized scoreboarding manages the previous use of a resource as well, avoiding the later use of a resource at an inconvenient time. If an operation can be divided so that a resource can be used in advance, then the scoreboard can mark pre-processed data as well as data waiting to be processed. The LIBRA architecture uses generalized scoreboarding by performing bounds checks during a load on unbound variables. If the bounds checks show that the unbound variable needs to be trailed if it is instantiated later, the register loaded is marked by setting a trail-check flag.

5. Conditional instruction execution is used to implement preferred branches and is also used with numeric conditions to control the execution of every instruction in the Acorn RISC Machine (Acorn Computers Limited 1986, 1987). The LIBRA architecture extends this concept by adding the symbolic conditions used in Prolog, allowing operations which formerly needed several test-and-branch instructions to be coded instead as a sequence of instructions, all of which are conditionally nops. This reduces the branch frequency of the LIBRA, and improves its ability to use interleaved memory.

4.2 INSTRUCTION SET

Prolog implementations quite often spend more time checking to see if a basic operation such as unification or trailing must be done than they take to do it. A unification may not be recursive; a variable may not need to be trailed; an argument may already be dereferenced; but the checks to determine that the operation is not needed appear to be unavoidable. However, in the Logical Inference Balanced RISC Architecture (LIBRA), designed to execute Prolog, two features were included to avoid recursive unification, and hide the time spent doing trail checking. When the LIBRA executes a load instruction, it checks unbound variables against internal bounds registers, and stores the result of the check in a status bit associated with each register. Later, if the unbound variable is bound, the status bit is used to conditionally execute a trailing instruction [Mills 88a]. The LIBRA also has a single-cycle partial unification instruction which uses the tags of the two operands being unified to select microcode to perform the correct unification operation: do nothing, bind, branch to fail, or call a recursive unification subroutine [Mills 88b].

Class		Instruction		Operands
Arithmetic	0	if cond3: IMM	{sc}	Hilmm18
	1	if cond3: ADD	{sc}	r1, r2, r3
	2	if cond3: ADD	{sc}	r1, Lolmm11, t3: r3
	3	if cond3: SUB	{sc}	r1, r2, r3
	4	if cond3: SUB	{sc}	r1, Lolmm11, t3: r3
Memory	5	if cond3: LD	{sc}	r1, Lolmm11, r3
	6	if cond3: ST	{sc}	r1, Lolmm11, r3
	7	if cond3: ST	{sc}	r1, Lolmm11, t3: r3
	8	if cond3: POP	{sc}	[R0 - R3], r3
	9	if cond3: PUSH	{sc}	[R0 - R3], {t2:} r2, {t3:} r3
	A	if cond3: PUSH	{sc}	[R0 - R3], t2: Lolmm11, t3: r3
Control	B	if cond3: SWITCH	[r1 | src2]	fail: b7, fail: b7, fail: b7
	C	if cond3: IF	{not}	cond5, b19
	D	if cond3: UNIFY	{sc}	r1, r2, b3, b7, b7
	E	if cond3: CALL	{sc}	[R28 - R31], b22
	F	if cond3: RETURN	{sc}	[R28 - R31]

Figure 2. Instruction set for minimal balanced RISC architecture

4.3 PERFORMANCE

Although the LIBRA was originally designed for logic programming, it is also an efficient Lisp architecture. Lisp and Prolog execute similar basic operations but Lisp is faster due to inefficiencies in the abstract machine used to implement Prolog (Tick, *Lisp and Prolog Memory Performance*, Stanford Tech. Report 86-291). Because the LIBRA provides single-cycle tag handling, function call, stack access and unification instructions using a MIPS-like approach to the datapath and only 6.8K bits of microcode, it provides better support for Common Lisp than the MIPS without the disadvantages of the Symbolics 3600 or the TI Explorer (Steenkiste and Hennessy, Lisp on a RISC: Characterization and Optimization, *IEEE Computer*, July 1988). An ECL LIBRA operating at 1.6 GHz with an 8 word instruction prefetch buffer and 32 word stack buffer could execute TAK in 0.000069 seconds, more than 600 times faster than any other Lisp implementation (Gabriel, *Performance and Evaluation of Lisp Systems*). Its Prolog performance is similar: measured in megaLIPS, a single ECL LIBRA could attain 174 megaLIPS. The LIBRA's performance was verified at Motorola by simulating an extended 88000 based on the LIBRA, and concepts introduced in the LIBRA have been validated by the success of the British Aerospace DLM (a Lisp/Prolog machine resembling the LIBRA, influenced by Mills' earlier LOW RISC design).

5. Extending the LIBRA to a CRAY-like architecture

Very similar to the LIBRA, but uses additional functional units to perform head unification in parallel. Pre-fetching of multiple instruction streams on a branch (may have one stream for each different type of object, and it is possible that more than one stream may be similar. Example: unification that succeeds only with a defined constant or a variable may prefetch streams for the constant unification code, the variable unification code, and the failure (or backtracking) code.

6. Attaching the LIBRA to a message-passing architecture

The LIBRA attached to a message passing host may have varying degrees of complexity, as different host architectures may require different approaches to add symbolic processing capability.

Varying the basic architecture yields the following possibilities:

- standard cell that can be included in host VLSI design as functional unit
- tightly coupled accelerator retrofitted to host node
- loosely coupled accelerator retrofitted to host node

• separate processing element communicating to host via thin wire connection

7. The LIBRA Implementation Project at Indiana University

The LIBRA has been described, and earlier versions simulated (88000e, LOW RISC II), but the design must be implemented to verify its performance. We are undertaking that implementation at Indiana University, approaching it from three directions.

7.1 Bipolar microprogrammed version

We are building a LIBRA prototype out of AMD 29000 horizontal function slices. This single node will provide information about the extensibility of LIBRA functions at the microcode level.

7.2 CMOS VLSI

A CMOS VLSI version of the LIBRA is planned. Initially the LIBRA and MIPS architecture and instruction set will be compared, then those instructions not available in the MIPS extracted. The combined LIBRA and MIPS instruction set will be simulated, and the LIBRA tuned by modifying its datapath. This will result in a basic architecture which can be translated into an equational specification according to techniques developed by Steve Johnson.

The architectural requirements of Common Lisp and Prolog on tightly coupled (shared memory) and distributed (message passing) parallel architectures will be analyzed. Preliminary research indicates that compiling Lisp to a tightly coupled multiprocessor results in numerous small functions per node (W. L. Harrison, *Compiling Lisp for Evaluation on a Tightly Coupled Multiprocessor*, CSRD Rpt. No. 565, University of Illinois Center for Supercomputing Research & Development). Small functions linked by conditional branches are optimally executed by the LIBRA due to its orthogonal conditional instruction execution. Experiments with a message passing architecture (the Connection Machine, R. Stevens, Argonne National Laboratory) suggest that a massively parallel architecture could be improved by addition of a unification operation. Simulations of the LIBRA will be performed on the BBN Butterfly at Indiana University to determine the dynamic frequency of LIBRA instructions. Using this information a variety of reduced LIBRA's will be extracted. A reduced LIBRA that unifies two 32-bit words and emits an instruction to the host node based on the result of the unification is one possibility for an LIBRA standard cell.

7.3 ECL Heterogeneous Gate Array

Motorola is interested in building the ECL LIBRA, and this effort is most likely to result in a powerful, single-node Cray-like processor.

References

[1] Mills, J. 1987. Coming to grips with a RISC.

[2] Mills, J. 1988. LIBRA: A High-Performance Balanced RISC Architecture for Prolog, Ph.D. dissertation, ASU, August 1988.

[3] Mills, J. 1986,1987,1988. Technical description and patent claims for the LOW RISC and LIBRA computers.

[4] Short, B. 1987. Extending a reduced instruction set computer to support Prolog.

[5] Short B. 1987. Use of instruction set simulators to evaluate the LOW RISC.

INDIANENSIS UNIVERSITATIS SIGILLUM

LUX ET VERITAS

MDCCCXX