Aquarius

A. Despain, Y. Patt, V. Srini,
P. Bitar, W. Bush, C. Chien, W. Citrin, B. Fagin, W. Hwu
S. Melvin, R. McGeer, A. Singhal, M. Shebanow, and P. Van Roy

Computer Science Division
University of California Berkeley, CA 94720

1. Introduction.

1.1. Overview.

The Aquarius project [1] has, as the fundamental goal of its research, to establish the principles by which very large improvements in performance can be achieved in machines specialized for calculating difficult problems in design automation, expert systems, and signal processing. These problems are characterized by having substantial numeric and symbolic components. We are committed to the eventual design of a very high performance heterogeneous MIMD multiprocessor tailored to the execution of both numeric and logic calculations. Aquarius began in 1983. By 1985 we had completed and demonstrated the Aquarius I system [15] which was a small heterogeneous multiprocessor. Aquarius I achieved about an order of magnitude higher performance than had been achieved up to that time. For example, the Japanese Fifth Generation Computer 'PSI' had achieved 30 KLIPS in 1985. We are currently focusing on an experimental multiprocessor architecture (Aquarius II) for the high performance execution of Prolog that will contain 12 processors specialized for Prolog and others for a total of 16 processors.

1.2. Research Methodology

It is worth stating at the outset a number of key concepts which reflect our fundamental methodology for doing research in high performance knowledge processing systems. We believe in a research environment where systems evolve, taking advantage of contributions from a number of sources, both within and outside Berkeley.

Second, we believe that issues should be dealt with as quickly and inexpensively as possible: by *gadanken* experiments, if possible, else analyzing, else simulation, else emulation and finally, only if required, by constructing and analyzing machines.

Third, the nature of the high performance execution demands the effective utilization of enormous amounts of memory, coupled both loosely and tightly, it involves exploiting parallelism at both course and fine grain granularities, and it necessitates modularization of the system architecture to accommodate improvements in any element in the structure.

Fourth, we are interested in proving concepts, rather than engineering manufactured parts. Thus, we are interested in building experimental architectures which can then be transferred to sites more appropriate than us for fabrication to achieve higher performance and more reliable systems. We are interested in using as many standard components and buses as possible in the experimental machine. This will facilitate the rapid transfer of the architecture technology.

Fifth, we believe in working closely with government and industry. Government provides the basic support; we, at the University, provide the research ideas and industry provides the hardware. We are primarily supported by DARPA* and the California MICRO program. Our

*Sponsored by Defense Advanced Reseaarch Projects Agency (DoD), Arpa Order No. 4871, Monitored by Space & Naval Warfare Systems Command under Contract No. N00039-84-C-0089.

current industrial partners include DEC, NCR, ESL, Xenologic, Apollo, Mentor, Valid, Bellcore, and CSELT.

Finally, we believe in careful instrumentation for the purpose of measuring what we have done. This means simulating before we build hardware, and including test apparatus in the microcode and in the hardware to measure the executing system.

1.3. What We Have Done.

Our work has involved the studies, simulation, emulation, design and construction of two experimental processors especially capable of executing Prolog (the PLM and the VLSI/PLM). We are in the process of designing a third (the PPP) and a fourth (the SIMU engine). We have done theoretical work in unification. We have written a Prolog compiler, a static data dependency analyzer and semi-intelligent backtracking system. We have written large application programs in Prolog specifically design tools to aid in construction of our experimental engines.

1.4. Our current research.

The current focus of our research is in three basic areas, as explained below. All three are concerned with maximizing concurrency by exploiting key ideas in various aspects of computer systems architecture.

1.4.1. Evolution of Aquarius II.

First, we are reducing our PLM processor into a single chip Prolog attached processor for Aquarius II, a heterogeneous MIMD multiprocessor capable of executing concurrently applications with substantial logic and numeric components. This system will be an experimental evolution of Aquarius I, and as such, it will reflect the evolution of ideas which began with the PLM. The first step is the reduction of the PLM processor to a single chip. This is almost complete, and will be incorporated into a host computer system for evaluation. The next processor chip, the PPP will be the basic processing element for Aquarius II. It is currently under design.

1.4.2. Improved Microarchitectures.

Second, we are investigating the viability of new ideas in microarchitecture and new mechanisms for implementing Prolog with a concurrent model of execution. We believe it is still the case that we have yet to exploit concurrency at this level of the computation hierarchy. This work will contain both theoretical and experimental components. We plan to microprogram existing hardware as well as design special purpose functional units that will run concurrently. We are investigating various schemes for translation, interpretation and compilation of Prolog Code, to support this approach.

1.4.3. The Knowledge Processing Environment.

Third, we are continuing our research in developing an environment for knowledge processing. We have had demonstrated success in several aspects of that process already; for example, our theoretical work in unification, our Prolog compiler, our static data dependency analysis (SDDA) work, and our work on the ASP Silicon compiler. Finally, we are currently preparing a substantial suite of large Prolog benchmark programs to provide a more meaningful metric than current benchmark sets.

1.5. Organization of this report.

The body of this report is organized in eight sections, each describing one of the key research problems we are currently working on. In section 2, we discuss our current efforts in the completion of a VLSI single chip Prolog processor. In section 3, we describe the extension of that chip to handle parallel processing of Prolog by replication of the basic Prolog processor. In section 4, we introduce another parallel processing model, the multiple functional unit approach. In section 5, we describe the research we are undertaking that is expected to pay dividends in synchronization of the full heterogeneous MIMD model. In section 6, we discuss our theoretical research in Unification. In section 7, we describe HPS, a new model of execution targeted for high performance microarchitectures. Finally, in section 8 we discuss the ASP compiler, a silicon compiler written entirely in Prolog. In section 9, we offer a few concluding remarks.

2. CMOS Chip for the PLM.

The design of the VLSI/PLM processor and its successor the PPP chip is one of the key projects in Aquarius. The objectives of the design are to achieve a cycle time that is the same as that of the TTL design **, to maintain compatibility with the TTL design at the microstate level, and to make the design modular so that some of the modules can be modified and the resulting chip can be used in a multiprocessor system. The performance objective of the chip presented a number of challenges. First of all the logic in more than 300 LSI and MSI chips occupying two hex size boards had to be put in a single chip. The next challenge is to achieve the 100 ns cycle time. The compatibility requirement of the chip with the TTL version means that at least 8 buses are needed in the data path to support 6 simultaneous register transfers and communicating address and data to memory. Additional buses are needed to avoid bus conflicts in data transfers if every block in the data path is not connected to each of the buses. Note that if each block is connected to every bus then the load capacitance on the bus will be high and it will increase bus transfer time. We have designed a chip, the VLSI/PLM, that meets the above objectives.

2.1. The Data Path.

The VLSI chip has three major units: data path, microsequencer, and ROM (containing microcode). The data path is the largest unit in the chip containing 20 blocks. It contains the special registers of the PLM in a register file, an ALU, counters of the PLM, and temporary registers for operands and results. The data path also has storage space for the top 16 entries of the pushdown list (PDL) and constants, hardware for PDL overflow and underflow detection, and hardware for detecting collisions between various segments of the data memory.

The design of the data path presented several challenges because of the 100 ns cycle time. The critical path involves reading operands from a register file, performing an ADD operation in the ALU and storing the result in a register. This involves designing a register file with a read access time of 30 ns or less, an ALU with a 40ns ADD time, and buses with a delay time of 5 ns. These constraints are based on the use of a two phase clock with a 10 ns nonoverlap time and that a maximum of 10 ns are needed to communicate the control point values. We have designed an ALU with parallel carry calculation circuit that has a worst case (VDD = 4.2 V and Temperature = 80 degrees Celsius) add time of 36 ns. The 28-bit counters in the data path have to increment or decrement and transfer the contents of the counter in 70 ns or less. The other time critical issue is the detection of PDL overflow or underflow in less than 30 ns after receiving control signals. The detection of PDL overflow or underflow has to be performed before phase 0 goes low since the calculation of the next microaddress is completed by this time.

2.2. The Microsequencer.

The microsequencer supplies the address of the next microinstruction to be executed. Its organization is similar to the TTL version of the PLM. It supports one level of microsubroutine and one level of interrupt. Two nine bit registers, microreturn pointer (urp) and control microreturn pointer (curp) are included in it to store return addresses. Fast microbranching is supported by partitioning the ROM into four pages and using logic to modify the two most significant bits (page bits) of the next microaddress seed. The micro page select (upage_select) logic modifies the page bits according to the current status and directives from the microinstruction.

The next microaddress is selected from different sources according to the current status and directives from the microinstruction in the micro program counter select (uPCselect) circuit. The

**The PLM processor was constructed with about 300 TTL chips.

potential sources for the next microaddress are: modified next address seed, new opcode, arg1 register, subroutine rom, microreturn register (urp), and control microreturn register (curp). Both upage_select and uPCselect circuits have been designed using the tree-height reduction method proposed by Kuck [16]

2.3. The ROM.

One of our goals is to design the ROM with a read access time of 40 ns. The NCR design team supplied the ROM as a macrocell. The ROM is organized as a NOR array with 128 rows and 640 columns. The 640 columns are divided into 160 groups with 4 columns in each group corresponding to the four pages. The least significant seven bits of the ROM address specify the row to be read. The most significant two bits specify the column.

The ROM uses a precharge scheme to reduce the read time. The reading takes place during phase 1 and the values of the 160 bits are supplied to the microinstruction register (MIR). The values of nine bits at the end of a word in the ROM are also supplied to output pad drivers for communicating them to the cacheboard.

2.4. Microcode Generation.

The microcode for the chip is generated from the microstate flow charts. The microcode is stored in the ROM. Almost 400 locations in the ROM are used up for the PLM instructions and initializing the chip. The remaining 112 locations are used for exception and interrupt handlers, microdiagnostics, and builtin functions.

The microinstructions are 160 bits long in the chip compared to 144 in the TTL version of PLM. This is because the number of buses in the chip and the implementation of PDL are different from the TTL version. There are also additional blocks in the chip to handle heap/stack and stack/trail collisions. The chip also has additonal circuits for testing.

2.5. Simulation.

The chip has been a complex one to design because of the 100 ns cycle time and the number of simultaneous data transfers that must be done in a cycle. We used functional simulation to verify the data transfers between the blocks in the chip and the computation of next microinstruction address. The timing simulation has been done with estimated capacitances to determine the delay through the blocks. The timing information is used in redesigning the blocks to achieve the 100ns cycle time.

The simulation efforts used a "start small" approach. Each block in the data path and microsequencer is functionally simulated by applying all possible values for the control inputs coming from microengine (MIR). The functional simulation of the ALU is carried out using two programs. All the functions of the ALU are exercised by using a given operand for the A and B inputs of the ALU in the first program. The add and subtract operations of the ALU are performed for a set of patterns by the second program. Exhaustive functional simulation of the MDR and Regfile blocks could not be done because of the large number of control inputs from MIR (20 to MDR and 16 to Regfile).

We are simulating the entire chip for functional correctness and timing. A set of programs written in C supply patterns to the pads from benchmark programs. Another set of C programs generate patterns for the units in the chip based on the patterns supplied to the pads.

3. The Parallel Prolog Processor

Many schemes have been proposed for executing Prolog in parallel on a multiprocessor, but all seem inadequate as candidates for a realizable parallel Prolog system. Almost all parallel execution models for Prolog have been designed without a vision of an underlying architecture, making them ill-suited for practical implementation. Typical features of these models include the creation of huge numbers of processes and a refusal to consider the costs associated with multiprocessing.

Parallel execution models do exist that are suitable for implementation, but these are discouragingly weak and inflexible in their use of concurrency. Typical features of these models include the utilization of only one type of parallelism and forcing the user to compute all solutions to a problem.

Another problem with the state of current research in parallel execution models for Prolog is that simulation results are extremely scarce. Researchers understand how to implement AND and OR parallelism, for example, but the costs and benefits associated with them have yet to be measured.

We intend to build a high performance computer suitable for executing parallel Prolog programs: the Aquarius multiprocessor. Because of the aforementioned deficiencies of the execution models for parallel Prolog currently being considered by the research community, we believe that none of them are adequate for our needs. Instead, we are constructing a new execution model for Prolog: the Parallel Prolog Processor model. The PPP model attempts to combine the power of the more abstract models with a concern for the cost/benefit tradeoffs inherent in any realizable multiprocessor system that these models lack. We intend to simulate the PPP very carefully, in an attempt to understand the merits of the various types of concurrency recoverable from Prolog, and to understand the architectural issues involved in supporting them.

The PPP model is expected to support AND parallelism, OR parallelism, the overlapped production and consumption of variable bindings from Prolog goals, and intelligent backtracking. The consistency check problem of AND parallelism is avoided by using static data dependency analysis to guarantee that AND processes never bind shared variables. The multiple binding problem of OR parallelism is solved using the "hash window" technique, first suggested by Warren. The support for intelligent backtracking is basically an extension of the work of J. H. Chang to a parallel execution environment, in which static data dependency analysis is used to pass additional information to an intelligent compiler.

In summary, we believe that present parallel execution models for Prolog are inadequate, because they were not designed with computer architecture in mind or because they are not powerful and flexible enough. We are designing a new model for the Aquarius multiprocessor system, which we hope will address both these issues. We also hope that detailed simulation will begin to address the lack of experimental results of parallel execution models for Prolog. It is expected that the PPP model will be implemented in the PPP processor currently under design.

4. A Multifunctional Unit Approach to Parallelism.

There are several forms of parallelism in Prolog. Another project we are working on exploits unification parallelism and also overlaps some book-keeping operations with unification. Measurements on benchmarks run on the Berkeley PLM indicate that from 40% to 60% of the total execution time is spent on unification and another 20% is spent on book-keeping. This means that even if we reduced the unification time and book-keeping time to zero we would be limited by a factor of 5 speedup. However, we expect to get a speedup of between two and three.

The architecture consists of a control unit which sequentially dispatches unification operations, one per argument of the clause head, to unification units. The unification units may also load arguments and also dereference arguments for escapes. The unifications are dynamically scheduled since the time required for unifications is indeterminate at compile time. Static scheduling would have reduced hardware utilization and effective parallelism.

Parallel unification requires synchronized access to unbound variables which may also be accessed by other unification units. While static data dependency analysis can be used to order execution of unifications, this provides a worst case schedule which does not exploit as much parallelism as could be exploited with dynamic scheduling and the hardware synchronization mechanism provided by our architecture. The hardware complexity is not unduly increased since, unlike unbound variables shared by two subgoals (AND parallelism), unbound variables shared within a clause head may be bound in any order, irrespective of the order of appearance of the arguments in the clause head. Simple hardware locks are provided by means of a set of dereference locks and write-once registers.

Access to heap and trail pointers are also synchronized. In addition the control unit performs choicepoint and trail buffering in order to reduce memory traffic. In our initial simulations we provide for up to four unification units. We feel that more units will not be useful because most prolog clause heads do not contain a large number of arguments and additional units will cause contention for the memory and a shared "distribution bus" which interconnects the control unit with all the unification units. Most of the available parallelism would be achievable with two unification units. A prefetch unit prefetches and buffers instructions for the control unit.

5. Fast Synchronization for Shared-Memory Multiprocessors

Our research here focuses on fast synchronization among the processors of a shared-memory system, in particular the Aquarius multiprocessor system. We have identified major subtopics in this domain, analyzed mechanisms proposed in the literature, and devised what we suggest are promising innovations. We are now evaluating the speedup potential of these options for the Aquarius memory system. Our performance evaluation uses stochastic modeling and simulation, and we are currently developing a stochastic model for this purpose. More specifically, our analysis of the issues, along with the innovations that we have proposed, may be summarized as follows.

5.1. Busy-Wait Locking, Waiting, Unlocking

The issues in *broadcast synchronization schemes for caches* have been analyzed, and new methods for busy-wait locking, waiting, and unlocking are introduced. The *lock/unlock* scheme allows busy-wait locking and unlocking to occur in zero time, eliminating the need for test-and-set; while the *wait* scheme eliminates all unsuccessful retries from the switch, and allows a process to work while busy-waiting. These methods for busy-wait locking, waiting, and unlocking also integrate *processor* atomic read-modify-write instructions and *programmer/compiler* implementations of atomic, busy-wait-synchronized operations under the same mechanism, and improve the performance of both approaches to atomic operations.

5.2. Sleep-Wait and Service-Request Queuing (for High-Contention Atomic Operations)

Fast queuing operations on priority queues, including the sleep-wait operations P and V, can be executed by VLSI hardware, whose structure, function, and management have been proposed. This introduces a paradigm for *VLSI implementation of high-contention atomic read-modify-write operations*. The paradigm will virtually eliminate switch traffic in the execution of such operations, as well as speed up the operations themselves tremendously.

6. Parallel Unification Scheduling in Prolog.

Unification is the fundamental operation in Prolog. Measurements have indicated that some Prolog interpreters spend over 50% of their time performing unifications [15] and data from the PLM simulator indicate similar results. Thus, in looking for areas in which Prolog execution speed may be increased, unification is a prime candidate.

Since the current motivation of the Aquarius project is to design a processor to exploit all available parallelism in a Prolog program, the approach taken here to speeding up unification is to exploit inherent parallelism. Unfortunately, there is a well-known result stating that, in the worst case, no parallel algorithm for unification exists which is substantially better than the best sequential algorithm[14], so the approach currently being taken is to divide unifications (particularly Prolog head unifications) into smaller tasks, then identify those tasks which may be safely performed in parallel. The tasks chosen map directly onto the unification instructions of the PLM instruction set. The safety criterion is twofold: first, two unification operations may not be unified simultaneously if they refer to the same memory location. Second, a structure argument may not be unified before the structure's functor.

In order to determine which operations may be safely executed in parallel, a static analysis of the Prolog program is performed before it is compiled. The analysis is a refined version of J-H Chang's static data-dependency analysis (SDDA) technique[12]. The result of the analysis is an estimate, for each procedure in the Prolog program, of the coupling relationships between the arguments to the procedure as it is called, and an estimate of the structure of any of the non-atomic arguments, along with estimates of the coupling relationships between arguments of those structures. Since a procedure may be called in several different ways, the estimate (or **entry mode**) is a worst-case generalization of all possible entry modes.

In order to improve the worst-case generalizations, the Prolog program is automatically transformed so that each procedure may be called in at most one way. If a procedure in the original is called in two ways, for example, from two different calling sites, a copy is made of the called procedure, and one calling site will call the original procedure while the other site will call the copy of the procedure. In this way, exact, rather than worst-case, information exists for each procedure, and better schedules can be generated.

Once data-dependency information has been derived, the unification operations are scheduled. There are three basic principles in scheduling. First, all operations in a parallel unification step must complete before the next step can begin. Second, no two operations in a scheduling step may reference variables which are coupled to each other. Third, no structure arguments may be unified before the functor of the structure is unified. If one operation in a parallel unification step fails, the entire unification is considered to have failed. Since generating an optimal unification schedule can be shown to be NP-complete, any of a number of good heuristic algorithms may be used[13].

After a schedule is generated, parallel unification instructions may be generated. These unification instructions perform the work of several sequential PLM instructions. Each unification operation is dispatched to one of an array of homogeneous unification processors. When all of the processors complete their unifications, the next parallel instruction is executed.

Work on this project will involve construction of a data-dependency analyzer and procedure splitter, a scheduler, and a parallel unification simulator. Once these are constructed, data will be gathered from a number of benchmarks to determine the amount of available parallelism in unification and the speedup to be gained from employing parallel unification. An analyzer has been constructed which does not perform structure analysis (this will be added shortly) and design has been completed on the scheduler. Preliminary data derived from the analyzer indicates that there is at least a parallelism of approximately 2:1 in most head unifications when structure information is not incorporated in the analysis, and that this parallelism should increase when structure analysis is added. This indicates that the speed of unification may be at least doubled.

7. HPS, A New Microarchitecture.

Our research in high performance computing involves exploiting concurrency at all levels of implementation. At the microarchitecture level, this work has resulted in a new model of execution, restricted data flow, which we believe has great potential for implementing very high performance computing engines for both numeric and symbolic computations. We are calling our microengine HPS, which stands for High Performance Substrate, to emphasize the notion that this model should be useful for implementing very dissimilar ISA architectures.

Our model of the microengine is a **restriction** on classical fine granularity data flow. It is not unlike that of Dennis [4], Arvind [5], and others, but with some very important differences, as discussed in [6]. The most important distinction is that, unlike classical data flow machines, only a small subset of the entire program is in the HPS microengine at any one time. We define the **active window** as the set of ISP instructions whose corresponding data flow nodes are currently part of the data flow graph which is resident in the microengine. As the active window moves through the dynamic instruction stream, HPS executes the entire program.

7.1. The Importance of Local Parallelism

It is important to emphasize the importance of local parallelism to our choice of execution model. Indeed, we chose this restricted form of data flow specifically because our studies have shown that the parallelism available in typical sequential control flow instruction streams is highly localized. We argue that, by restricting the active instruction window, we can exploit almost all of the inherent parallelism in the program while incurring very little of the synchronization costs which would be needed to keep the entire program around as a total data flow graph.

7.2. Potential Limitations of Other Approaches.

We believe that an essential ingredient of high performance computing is the effective utilization of a lot of concurrency. Thus we see a potential limitation in microengines that are limited to one operation per cycle. Similarly, we see a potential limitation in a microengine that underutilizes its bandwidth to either instruction memory or data memory. Finally, although we appreciate the advantages of static scheduling, we see a potential limitation in a microengine that purports to execute a substantial number of operations each cycle, but must rely on a non-run-time scheduler for determining what to do next.

7.3. Our Approach

7.3.1. The Three Tier Model.

We believe that irregular parallelism in a program exists both locally and globally. Our mechanism exploits the local parallelism, but disregards global parallelism. Our belief is that the execution of an algorithm should be handled in three tiers. At the top, where global parallelism can be best identified, the execution model should utilize large granularity data flow, much like the proposal of the CEDAR project [7]. In the middle, where forty years of collected experience in computer processing can be exploited probably without harm, classical sequential control flow should be the model. At the bottom, where we want to exploit local parallelism, fine granularity data flow is recommended. Our three tier model reflects our conception that the top level should be algorithm oriented, the middle level sequential control flow ISP architecture oriented, and the bottom level microengine oriented.

7.3.2. Stalls, Bandwidth, and Concurrency

We believe that a high performance computing engine should exhibit a number of characteristics. First, all its components must be kept busy. There must be few stalls, both in the flow of information (i.e., the path to memory, loading of registers, etc.) and in the processing of information (i.e., the functional units). Second, there must be a high degree of concurrency available, such as multiple paths to memory, multiple processing elements, and some form of pipelining, for example.

In our view, the restricted data flow model, with its out-of-order execution capability, best enables the above two requirements, as follows: The center of our model is the set of node tables, where operations await their operands. Instruction memory feeds the microengine at a constant rate with few stalls. Data memory and I/O supply and extract data at constant rates with few stalls. Functional units are kept busy by nodes that can fire. Somewhere in this system, there has to be "slack." The slack is in the nodes waiting in the node tables. Since nodes can execute out-of-order, there is no blocking due to unavailable data. Decoded instructions add nodes to the node tables and executed nodes remove them. The node tables tend to grow in the presence of data dependencies, and shrink as these dependencies become fewer. Meanwhile, our preliminary measurements support, the multiple components of the microengine are kept busy.

7.4. The HPS Model of Execution

An abstract view of HPS is as follows: Instructions are prefetched from a static instruction stream via the use of a branch predictor into the dynamic instruction stream, as shown at the top of the figure. From there, instructions are fetched according to the fetch control unit described below, decoded, and presented for merging.

This discussion implies that the instruction stream is taken from a sequential control flow ISP architecture. This is not a necessary part of the HPS specification. In fact, the HPSm model directly processes multinode words (i.e., the nodes of a directed graph), which are produced as the target code of a (for example) C compiler [8], while an HPS implementation of the VAX would process sequential control flow native mode VAX instruction stream. What is necessary is that, for each instruction, the output of the decoder which is presented to the Merger for handling by HPS is a data flow graph.

A very important part of the specification of HPS is the notion of the active instruction window. Unlike classical data flow machines, it is not the case that the data flow graph for the entire program is in the machine at one time. We define the active window as the set of ISP instructions whose corresponding data flow nodes are currently being worked on in the data flow microengine.

As the instruction window moves through the dynamic instruction stream, HPS executes the entire instruction stream. Parallelism which exists within the window is fully exploited by the microengine. This parallelism is limited in scope; ergo, the term "restricted data flow."

The Merger takes the data flow graph corresponding to each ISP instruction and, using a generalized Tomasulo algorithm to resolve any existing data dependencies, merges it into the entire data flow graph for the active window. Each node of the data flow graph is shipped to one of the node tables where it remains until it is ready to fire.

When all operands for a data flow node are ready, the data flow node fires by transmitting the node to the appropriate functional unit. The functional unit (an ALU, memory, or I/O device) executes the node and distributes the result, if any, to those locations where it is needed for subsequent processing: the node tables, the Merger (for resolving subsequent dependencies) and the Fetch Control Unit (for bringing new instructions into the active window). When all the data flow nodes for a particular instruction have been executed, the instruction is said to have executed. An instruction is retired from the active window when it has executed and all the instructions before it have retired. All side effects to memory are taken care of when an instruction retires from the active window. This is essential for the correct handling of precise interrupts [9].

The instruction fetching and decoding units maintain the degree of parallelism in the node tables by bringing new instructions into the active window, which results in new data flow nodes being merged into the data flow node tables. The branch predictor is very important to this scheme, since (unlike the piecewise data flow model of Requa and McGraw [10], for example) we allow out-of-order execution to take place across branch boundaries.

Current Work and Concluding Remarks

Our current research is taking HPS along four very different tracks, as we attempt to understand the limits of this microarchitecture. One major track, particularly relevant to Aquarius, is our study of the potential of HPS for implementing a a Prolog processor.

In this work, we are addressing several issues that are particularly relevant to the HPS model of execution. For example, if HPS is to implement a sequential control based ISP architecture, then there are decoding issues, including the question of a node cache, which need to be decided. Second, HPS requires a data path that (1) has high bandwidth and (2) allows the processing of very irregular parallel data. These two objectives usually suggest contradictory designs. Third, HPS needs a scheduler which can determine, in real-time, which nodes are firable and which are not. Fourth, the out-of-order execution of nodes requires additional attention to the design of the memory system, the instruction retirement and repair mechanisms, and the I/O system. These issues are discussed in [11].

8. Advanced Silicon Compiler In Prolog.

The purpose of the Advanced Silicon compiler in Prolog (ASP) project is twofold. We wanted a system that would rapidly generate good microprocessor designs as a tool for the architectural research being pursued by the Aquarius project. We also wanted to understand the benefits and liabilities of using Prolog for large software systems in general and CAD in particular.

8.1. Decomposition of the Silicon Compilation Problem.

A full behavior-to-silicon compiler is a complex undertaking; as computer scientists, we sought a tractable decomposition of the problem. Following Gajski, we decompose the silicon compilation problem into three abstract problem domains, ordered more or less hierarchically.

The top level of our system is the behavioral domain. This level generates a set of logical devices, controlled by a finite state machine, from an input specification written in Prolog. The devices (and the finite state machine) are generated with performance and area constrained. Both standard compiler techniques and hardware-specific knowledge are used in this process.

The second level is the circuit or functional domain. The purpose of this domain is to present the programs of the behavioral domain with an abstract class of idealized devices. Hence, this level attempts to synthesize, place, and route the finite state machine and logical devices generated by the behavioral level into a sticks-and-stones description of the circuit. This level encompasses the traditional tasks of state assignment, logic synthesis, transistor sizing, place and route, module generation and layout. Currently, the core of this level is in place with our Topolog Module Generator.

The third level is the geometric or structural domain. The purpose of this domain is to present the programs of the functional domain with an abstract class of idealized elements, or a sticks-and-stones virtual-grid abstraction of the actual mask layers involved in fabricatable design. This domain encompasses the traditional tasks of compaction, river routing, and device-level simulation. These tasks are accomplished by the Sticks-Pack component of ASP.

Clearly there is some interaction between the levels. No layout generator can ignore the constraints inherent in technology, such as, for example, the richer connectivity of one level of metal. Similarly, the machine generator must set reasonable constraints on the performance or area of a given logical device; demanding a 1-ns 32-bit ALU will not produce such a device.

Interfaces are entirely procedural; lower levels are regarded as providers of information and services to the next level above.

8.2. Status of ASP

ASP is not yet complete. We have demonstrated the generation of a data path somewhat simpler than that of the PLM. We expect to automatically generate the complete VLSI/PLM Processor design within the next two years.

9. Conclusions

The Aquarius project is less than half completed, yet a number of interesting results have been obtained and several interesting systems have been demonstrated (see also [17-47]). We have freely distributed to other research groups some of our tools and systems such as our Prolog Compiler and PLM Simulator. Several of our systems have been improved by our industrial partners and are either offered or will be soon offered to the public.

In summary we believe we have made major contributions with our on-going research. We face a series of difficult research issues. but have launched a multilevel attack to get these issues resolved. The central, top-level, issue still remains: How to achieve a large improvement in computer system performance through concurrent execution.

Acknowledgements

Many have contributed to the research we are doing on the Aquarius project. We first acknowledge our past and present co-workers: Ashar Butt, Tep Dobry, J.H. Chang, Michael Carlton, Gino Cheng, Ron David, Bruce Holmer, Jonathan Pincus, Nasser Lone, Tam Nguyen, John Swensen, Jerric Tam, Herve Touati, Erling Wold, Robert Yung, James Wilson, Jiajuin Wei, Weijin Dai, Robert Williams. Second, we gratefully acknowledge the industrial support from DEC, NCR, ESL, Xenologic, Apollo, Mentor, Valid, Bellcore, and CSELT.

References.

[1] Alvin M. Despain, and Y.N. Patt,"The Aquarius Project," *Digest of Papers, COMPCON84* San Francisco, February, 1984

[2] Bitar, P., "Fast synchronization for shared-memory multiprocessors," Dec. 1985. *TR 85.11, Research Institute for Advanced Computer Science,* NASA Ames Research Center, MS 230-5, Moffett Field, CA 94503.

[3] Bitar, P., Despain, A.M., "Multiprocessor cache synchronization," *13th Int'l Symp. on Computer Architecture,* 1986.

[4] Dennis, J. B., and Misunas, D. P., "A Preliminary Architecture for a Basic Data Flow Processor," *Proceedings of the Second International Symposium on Computer Architecture,* 1975, pp 126-132.

[5] Arvind and Gostelow, K. P., "A New Interpreter for Dataflow and Its Implications for Computer Architecture," *Department of Information and Computer Science, University of California, Irvine, Tech. Report 72,* October 1975.

[6] Patt, Y.N., Hwu, W., and Shebanow, M.C., "HPS, A New Microarchitecture Rationale and Introduction," *18th Annual Microprogramming Workshop,* Asilomar, CA, December, 1985.

[7] Gajski, D., Kuck, D., Lawrie, D., Sameh, A., "CEDAR -- A Large Scale Multiprocessor," *Computer Architecture News,* March 1983.

[8] Shebanow, M.C., Patt, Y.N., Hwu, W., and Melvin, S.W., " A C Compiler for HPS I, a Highly Parallel Execution Engine," *Hawaii International Conference on System Sciences - 19,* Honolulu, HI, January, 1986.

[9] Anderson, D. W., Sparacio, F. J., Tomasulo, R. M., "The IBM System/360 Model 91: Machine Philosophy and Instruction - Handling," *IBM Journal of Research and Development,* Vol. 11, No. 1, 1967, pp. 8-24.

[10] Patt, Y.N., Melvin, S.W., Hwu, W., and Shebanow, M.C., "Critical Issues Regarding HPS, a High Performance Microarchitecture, *Proceedings of the 18th International Microprogramming Workshop,* Asilomar, CA, December, 1985.

[11] J.-H. Chang, "High Performance Execution of Prolog Programs Based on a Static Data Dependency Analysis", Ph.D Thesis, University of California, Berkeley, October 1985.

[12] W. Citrin, "Parallel Unification Scheduling in Prolog," Ph.D Thesis, University of California, Berkeley, est. September, 1987.

[13] C. Dwork, P.C. Kanellakis, and J. C. Mitchell, "On the Sequential Nature of Unification," *The Journal of Logic Programming,* June, 1984.

[14] N. S. Woo, "A Hardware Unification Unit: Design and Analysis," *Proceedings of the 12th Intl. Symposium on Computer Architecture,* New Orleans, June 1985.

[15] T.P. Dobry, A.M. Despain, Y.N. Patt, "Performance Studies of a Prolog Machine Architecture," *Proceedings of the 12th Intl. Symposium on Computer Architecture,* June 1985.

[16] Kuck, David, " *The Structure of Computers and Computations*", Wiley 1978.

[17] Alvin M. Despain, and Y.N. Patt, "Aquarius -- A High Performance Computing System for Symbolic/ Numeric Applications" *COMPCON 1985,* February, 1985.

[18] Peter Van Roy, "A Prolog Compiler for the PLM", *University of California, Berkeley,* November 1984.

[19] Wayne Citrin, Peter Van Roy, Alvin M. Despain, "A Prolog Compiler," *HICSS-19* January 1986.

[20] Jung-Herng Chang and Alvin M. Despain, Doug de Groot, "Semi-Intelligent Backtracking of Prolog Based on A Static Data Dependency Analysis," *Logic Programming Conference,* July, 1985.

[21] Wayne Citrin, Peter Van Roy, Alvin M. Despain, "Compiling Prolog for the Berkeley PLM," *HICSS-19* January 1986.

[22] T.P. Dobry, Y.N. Patt, A.M. Despain, "Design Decisions Influencing the Microarchitecture for a Prolog Machine," *MICRO 17 Proceedings, R October 1984.*

[23] Carl C. Ponder and Yale Patt, "Alternative Proposals for Implementing Prolog Concurrently and Implications regarding their respective Microarchitectures," *Proceedings of the 17th Annual Microprogramming Workshop,* October 1984

[24] T.P. Dobry, J.H. Chang, A.M. Despain, Y.N. Patt, "Extending a Prolog Machine for Parallel Execution," *HICSS Proceedings 19,* January 1986

[25] Barry Fagin and Alvin M. Despain, "Goal Caching in Prolog," *HICSS Proceedings* January 1986

[26] Barry Fagin, Yale Patt, Vason Srini, Alvin Despain, "Compiling Prolog into Microcode: a Case Study Using the NCR/32-000," *MICRO 18 Proceedings,* December 1985.

[27] Jonathan Pincus, Alvin M. Despain, "Delay Reduction Using Simulated Annealing," *Design Automation Conference* 1986

[28] Patrick McGeer, William Bush, Jonathan Pincus, Alvin Despain, "Design Considerations for a Prolog Silicon Compiler," *Submitted to Design Automation Conference* 1986

[29] Nasser N. Lone, "A Replacement Algorithm for Prolog Goal Caches," *University of California, Berkeley,* December, 1985.

[30] Ashar Butt, Alvin Despain, "Cell Design in Prolog," *University of California, Berkeley* August, 1985

[31] Wayne Citrin, "A Comparison of Indexing and Term Re-Ordering: Two Methods for Speeding Up the Execution of Prolog Programs," Unpublished.

[32] A.M. Despain, Y.N. Patt, T.P. Dobry, J.H. Chang, W.Citrin "High Performance Prolog, The Multiplicative Effect of Several Levels of Implementation," *COMPCON,* March, 1986

[33] Yale N. Patt, Wen-mei Hwu, Stephen Melvin, Michael Shebanow, Chien Chen, Jiajuin Wei, "Experiments with HPS, a Restricted Data Flow Microarchitecture for High Performance Computers," *COMPCON,* March, 1986

[34] Wen-mei Hwu, Steve Melvin, Mike Shebanow, Chien Chen Jiajuin Wei, Yale Patt, "An HPS Implementation of VAX, Initial Design and Analysis," *HICSS-19,* January 1986.

[35] Wen Mei Hwu, Yale N. Patt, "HPSm A High Performance Restricted Data Flow Architecture Having Minimal Functionality," *13th International Symposium on Computer Architecture* Tokyo, Japan, June 1986.

[36] Alvin M. Despain, Vason Srini, Yale Patt, Barry Fagin, "Architecture Research for Prolog Based on NCR/32," University of California, Berkeley, 1986

[37] Alvin M. Despain, "A High Performance Hardware Architecture for Design Automation," University of California & Xenologic Inc. *Aerospace Applications of Artificial Intelligence Conference,* Oct, 1986

[38] Yale N. Patt, "Several Implementations of Prolog, the Microarchitecture Perspective," *Proceedings of the IEEE International Conference on Systems, Man & Cybernetics,* October, 1986

[39] Yale N. Patt, Stephen Melvin, "A Microcode-Based Environment for Non-Invasive Performance Analysis," *Proceedings of the 19th Microprogramming Workshop,* October, 1986.

[40] Yale N. Patt, Stephen W. Melvin, Wen-mei Hwu, Michael Shebanow, Chien Chen, Jiajuin Wei "Run-Time Generation of HPS Microinstructions from a VAX Instruction Stream," *Proceedings of the 19th Microprogramming Workshop,* October 1986.

[41] Jeff Gee, Stephen W. Melvin, Yale N. Patt, "The Implementation of Prolog via VAX 8600 Microcode," *Proceedings of MICRO 19,* New York, October 1986.

[42] Wen Mei Hwu and Yale N. Patt, "Design Choices for the HPSm Microprocessor Chip," *Proceedings of the 20th Annual Hawaii International Conference on System Sciences,* Kona, Hawaii, January 1987.

[43] Stephen W. Melvin, Yale Patt, "A Clarification of the Dynamic/Static Interface," *Proceedings of the 20th Annual Hawaii International Conference on Systems Sciences,* Kona Hawaii, January 1987.

[44] Patrick McGeer, Robert K. Brayton, "Efficient, Stable Algebraic Operations on Logic Expressions," Submitted to VLSI 1987, August 1987

[45] Wen-Mei Hwu and Yale N. Patt, "Checkpoint Repair for Out-of-order Execution Machines," *Proceedings, 14th Annual International Symposium on Computer Architecture,* Pittsburgh, Pennsylvania, June 1987

[46] John Swensen and Yale Patt, "Fast Temporary Storage for Serial and Parallel Computation," *Proceedings, 14th Annual International Symposium on Computer Architecture,* Pittsburgh, Pennsylvania, June 1987

[47] Jeff Gee, Stephen Melvin and Yale Patt "Advantages of Implementing Prolog by Microprogramming a Host General Purpose Computer," *Proceedings, 4th International Conference on Logic Programming,* Parkville, Victoria, Australia, May 1987