Hardware and Software Architectures for Efficient AI

Michael F. Deering

Fairchild Laboratory for Artificial Intelligence Research
Fairchild Camera and Instrument Corporation
4001 Miranda Avenue
Palo Alto, California 94304

Abstract

With recent advances in AI technology, there has been increased interest in improving AI computational throughput and reducing cost, as evidenced by a number of current projects. To obtain maximum benefit from these efforts, it is necessary to scrutinize possible efficiency improvements at every level, both hardware and software. Custom AI machines, better AI language compilers, and massively parallel machines can all contribute to efficient AI computations. However, little information is available concerning how to achieve these efficiences. A systematic study was undertaken to fill this gap. This paper describes the main results of that study, and points out specific improvements that can be made. The areas covered include: AI language semantics, AI language compilers, machine instruction set design, parallelism, and important functional candidates for VLSI implementation such as matching, associative memories, and signal to symbol processing for vision and speech.

1 Introduction

As AI software grows in complexity, and as AI applications move from laboratories to the real world, computational throughput and cost are increasingly important concerns.

In general, there are two motives for increasing the efficiency of computations. One is the need to obtain faster computation, regardless of cost. This may be due to explicit real-time constraints. It may also be due to current methods being taxed well beyond the limit of complexity or timely response. The other is when increases in computational efficiency are part of an overall effort to obtain a better cost/performance ratio. Both these motives arise within AI, and causes for each will be examined. Behind both, however, is usually the imperative of real world market pressures.

Opportunities for increased efficiencies in AI computations exist at every level. Improved instruction set designs combined with improved AI language semantics allow more powerful compiler optimizations to be performed. Concurrent machines allow parallel execution of Lisp and declarative constructs, raising issues of *and*, *or* and *stream* parallelism. Custom VLSI implementations for current AI performance bottlenecks are also possible, via devices such as hardware unifiers, associative memory, and communication hardware for coordinating parallel search. Many of these speed-ups are orthogonal and can potentially lead to multiplicative performance enhancements of several orders of magnitude. However, this is not always the case, as the optimizations can sometimes interfere. For example, some language optimizations may tend to serialize the computation, negating parallelism gains.

As part of an effort to design a massively concurrent architecture for AI computation (the Fairchild FAIM-1 project), a comprehensive study was done to determine potential throughput increases at various levels and their interactions. This paper will examine several results of this study.

2 Misconception

There are several misconceptions of what needs to be done to improve computational throughput for AI. Since most AI is done in Lisp, many believe the key is simply to make Lisp a few orders of magnitude faster. However this approach ignores potential speed-ups that may be easier to obtain elsewhere. Others see no reason to concentrate upon anything other than the fundamental problem of parallelism. This approach presumes routine solution of a very difficult problem: decomposing arbitrary AI computations to effectively use thousands of parallel processors. A problem with this is that most programs, even ones with a high degree of inherent parallelism, almost always have several serial bottlenecks. As an example, most parallel programs need to gather the result of one batch of parallel computations for reflection before generating the next batch. In many cases these serial sections will dominate the running time of the entire program. So one cannot ignore the issue of how to extract as much serial speed as possible from languages and machines. Otherwise it might be the case that, having built an expensive parallel machine hundreds of times faster than existing machines, a new compiler and/or microcode may make some existing serial machine even faster! The machine coded unifier in the Crystal AI language, for instance, is two orders of magnitude faster than the Lisp coded unifier in the predecessor PEARL AI language [Deering 81a].

3 Software: What can be done to help AI language implementations

3.1 Compile the language directly to machine code

Most "AI languages" per se are not complete computer languages, but packages of routines on top of an existing language (usually Lisp.) While this is a great way of rapidly prototyping a language, and results in an order of magnitude savings in development costs over a traditional full compiler, it does not lead to very efficient implementations. If to increase the speed of AI applications the extreme of building custom parallel processors is being considered, it is silly not to compile AI languages directly onto these processors. There is a large body of computer science knowledge on compilation that can be brought to bear, and great potential for performance increase. (Consider the 100x plus speed difference between most Lisp based Prolog interpreters and Warren's DEC-20 Prolog compiler [Warren 77].)

3.2 Make sure that the language is compilable

Because most AI language implementations have been interpreters, issues of compilability generally have not been thought through. Language features that seemed efficient in an interpreted environment may be very slow when compiled, if they are compilable at all. A proper choice of features in light of a compiled environment will lead to more efficient program execution.

3.3 Add extensive libraries of useful routines

Another problem with many AI languages is the lack of general tools to support common applications. While it is argued that this allows the user to write his own customized tools (that may be very efficient), most users will do a much worse job than the language implementor could. For example, PEARL did not directly support any particular theorem proving or search system (such as forward and backward chaining), leaving the user to his own devices. But the MRS system [Genesereth 83], while it provides a convenient meta-level control for users to write their own search systems in, also provides a range of built-in search strategies, from backward chaining to full resolution theorem proving. The point is that an extensive library of well written routines of general use will speed the operation of typical user programs (not to mention their development).

4 Hardware: What can be done to help conventional computer instruction sets

It is often said that conventional computer instruction sets are not well suited for AI software, but there has been few attempts to quantify the reasons why. For older generation machines, severe address space limitations and lack of flexible pointer manipulation facilities are easy to point to [Fateman 78]. But what of the new more modern machines, such as the VAX, 68000, 16000 and RISC machines, and how do they compare with the custom Lisp machines? (Such as [Knight 81] and [Lampson 80].) To obtain insights into instruction set design, several Lisp systems and the fine details of their implementation were examined [Deering 84]. Several things were learned. It is very important to identify how rich of an environment one wishes to support. For example, contrary to many people's expectations, on a large application program, Franz Lisp [Foderaro 83] on a VAX-11/780 was not significantly slower than Zetalisp on a Symbolics 3600. The difference was that most all type checking and generic function capabilities were either turned off (by the programmer) or missing in Franz, and the overall environment was much poorer. Assuming that such things are not frills, the expense of providing them on different architectures was examined.

Flexible Lisp processing depends upon dynamic type checking and generic operations. Associating the data type directly with the data object means that the data type will always be at hand during processing, and this is the reason that tagged memory architectures are well suited to lisp processing. Because of this, the speed of various processors upon the generic Lisp task was dependent upon how fast they could effectively emulate a tagged memory architecture.

A number of experiments were performed to compare Lisp systems and processor instruction sets. As a representative sample, the timing results for a simple aggregate function incorporating some of the most common Lisp primitives (car, cdr, plus, function call/return) is shown in the table below:

Lisps vs. Processors on:			
(defun foo (x) (+ (car x) (cdr x)))			
Machine	Zetalisp	Franz	PSL
VAX	53.8μs	13.9μs	5.6μs
68000	65.2μs	43.6μs	5.8μs
68010	68.6μs	43.6μs	10.6μs
68020	29.75μs	27.9μs	4.6μs
MIT CADR	19.0μs	n/a	n/a
3600	6.4μs	n/a	n/a

More extensive benchmarks have borne out (very) roughly the same speed ratios. The variance exceeded 50%, but this was not unexpected. Slight modifications of the compilers or instruction sets produced similarly large changes in the speeds.

Existing Franz and PSL [Griss 82] compilers for the VAX and 68000's were used to compile foo. Type checking was turned off to obtain the fastest speeds. (Both PSL and Franz were told not to verify that the arguments of + were small integers, Franz did and PSL did not check for numeric overflow.) The timings figures were generated by examination of the assembly code produced and some actual machine timings. The timings of Zetalisp for the 3600 and CADR was taken by running existing systems. Zetalisp-like operations for the VAX and 68000's were hand coded, and the timings produced in the same way as those for PSL and Franz. The 68000 and 68010 were 10MHz no wait-state machines. The 68000 used 24 bit addresses, leaving the upper 8 data bit free for tag values. The 68010 used 32 bit addresses, and required the tags to be and-ed off before addresses could be used. The 68020 timings are estimates based upon the best available (but sketchy) preliminary performance data for a full 32 bit 16MHz machine with a small instruction cache.

Other experiments examined the architectural requirements for fast computation of some AI operations not directly supported by Lisp, in particular unification and associative search. When AI languages are fully compiled, these two functions many times become the computational bottlenecks. For traditional microprocessor instruction sets, the requirements of these operations turned out to be the same as for Lisp primatives: fast simulation of tagged architectures. More specifically, the instructions and capabilities that would make a conventional microprocessor better suited for Lisp (and Prolog, Krypton, MRS, PEARL, etc.) are:

- "Extract bit field and dispatch", an instruction to extract a sequence of bits from an operand, then add these bits to a dispatch table address, and jump indirect. This is necessary for rapid handling of tag values in generic operations, type checking, and for helping with unification.
- "Extract two bit fields, concat, and dispatch", an instruction for dispatching upon the context of *two* operands. (needed for the same reasons as the single argument version.)
- The memory address system of the processor should ignore the upper address bits of data addresses that are not otherwise in use. This allows the wasted space in 32 bit pointers to be used as a tag field.

In the Zetalisp-like code, more than 30% of the time on the 68000's was spent in emulating the bit field dispatch instructions. Stripping off the tag bits accounted for another approximately another 10%. It is therefore estimated that if the existing microprocessors had hardware support for these features, full type checking Lisps (like Zetalisp) could run almost twice as fast. These percentages come from hand implementing several Zetalisp primatives on current microprocessors. As an example, below the 68010 assembler code is shown for CAR. The number of processor clock cycles per instruction is shown in the left hand column. The boxed code will later be replaced by a single instruction.

Zetalisp car for 68010

```
        ; To take the car we do a few lines of in line code and
        ; then index jump to a subroutine. (Space for time.)
        ; The cons cell to take the car of is assumed in a0.

 ; dispatch to CAR subr based upon the tag in upper bits of a0
 4    movel a0,d2         ; put a copy of the arg into d2
 24   lsll   #8,d2        ; first 8 of: shift copy over by 9 bits
 10   lsll   #1,d2        ; last 1 of: shift copy over by 9 bits
 14   andl   #0x1F0,d2    ; and off non-tag (shifted over)
 18   jsr    CAR(d2)      ; branch to car table indexed by type
      ; At return, the car of the object is in a2

; The CAR subroutine.
CAR + DTP-CONS: ; CAR procedure entry point
                ;for normal cons cell.
; We will arrive here if the argument passed to car was of type
; "pointer to cons cell". Other objects passed to car => error

 ; follow the pointer to the car
 4    moveal    a0,d2           ; put a copy of the arg into d2
 14   andl   #0xFFFFFF,d2       ; and off tag
 4    moveal    d2,a2           ; put d2 into an address register
 12   moveal    (a2),a2         ; follow the car pointer.

 ; dispatch to TRANSPORT subr based upon the tag
 ; in the upper bits of a2
 4    movel a2,d2         ; put a copy into d2
 24   lsll   #8,d2        ; first 8 of: shift copy over by 9 bits
 10   lsll   #1,d2        ; last 1 of: shift copy over by 9 bits
 14   andl   #0x1F0,d2          ; and off non-tag (shifted over)
 10   jmp    TRANSPORT(d2) ; branch to car table
                          ; indexed by type.
                          ; The reason for this jump is to check
                      ; for possible invisible pointers, unbound, etc.
TRANSPORT + NORMAL: ; jump entry point for normal
                    ;cons cell contents
 8    rts           ; We're all done, return
 ———
174 clocks, @10MHz = 17.4µs
```

Now CAR for the 68010 will be recoded assuming two architectural refinements. First assume that the upper 7 bits of all addresses be ignored by the (virtual) memory system. Second, assume one additional instruction "extract bit field and dispatch". This instruction takes the bit field out of the second argument, as specified by the first argument (format: <#starting-bit, field-width>), adds it to the third argument (the jump table base address), and jump indirect through this address.

```
; Now the car routine is recoded using the new instructions:
; index jump to a subroutine.
; dispatch to CAR subr based upon the tag in upper bits of a0
22    extract-dispatch  <#26,#6>,a0,CAR

; The CAR subroutine.
CAR + DTP-CONS: ; CAR procedure entry point for
                ; normal cons cell.
; follow the pointer to the car
12    moveal (a0),a2     ; the upper 6 bits of a0 are ignored.
; dispatch to TRANSPORT subr based upon the tag
; in the upper bits of a2
22    extract-dispatch  <#26,#6>,a2,DISPATCH
TRANSPORT + NORMAL: ; jump entry point for normal
                    ; cons cell contents
 8    rts           ; We're all done, return
 ———
64 clocks, @10MHz = 6.4µs, 2.7 times faster
```

For new, fully custom machine designs which are tailored specifically for AI, such features can all be built in. With a tagged architecture, many generic operations, such as "add", do not need to be dispatch subroutine calls. Rather the processor can examine the tags of the arguments to an add instruction, and if they are simple integers, directly perform the add. If the arguments are of a more exotic numeric type, the processor can generate a software interrupt to an appropriate routine. Further, for such designs it is very helpful to have a "smart" memory subsystem capable of rapidly chasing down indirect pointers (as on the PDP-10 and the custom Lisp machines). Additional customizations of a special AI instruction set design generally fall into the category of complete attached processors rather than just another instruction. This tactic has already been taken by many microprocessors whose floating point instructions are handled by what could be viewed as attached processors. The specific categories of important attached processors include: pipelined unifiers, associative memory sub-systems, multiprocessor communication packet switchers, and special signal processing chips for vision and speech.

Studies of a custom instruction set for the FAIM-1 machine indicate that not only can a single processor be designed that is memory bound by DRAM access delays, but that this is the case even when a large cache is employed. This is an important fact. It means that parallel machines sharing a single large common memory are a bad idea, there is not enough memory bandwidth to go around.

5 Parallelism: The great hope

Traditionally, concurrency has been viewed as a great method of obtaining increased computational power. In practice, however, designers continue to concentrate upon making single processor machines faster and faster. However, now that hard technological limits have been hit for serial processors, parallelism has become recognized as perhaps the only hope for further orders of magnitude performance increases. Unfortunately concurrency is not free as it brings new systems organizational problems to the fore.

The first conceptual problem with parallelism is the confusion between multi-*processing* and multi-*processors*. There are algorithms that are very elegantly expressed in terms of a set of cooperating processes (e.g. writers and readers), but these same algorithms have little or no inherent *parallelism* that can be exploited by parallel computers. Just because an algorithm can be expressed in concurrent terms is no guarantee that, when run on many parallel processors, it will run significantly faster than as separate processes on a single sequential machine.

The true measure of parallelism is how much faster a given program will run on n simple parallel processors compared to how fast it would run on a single simple processor, and for what ranges of n this is valid. The best one can hope for in principle is a factor of n speedup, but in practice this is rarely reached (due to overheads and communication contention). The maximum amount of speedup attained for a given program upon any number of parallel processors indicates the inherent parallelism of that program. Unfortunately, for most existing programs written in traditional computer languages, the maximum parallelism seems to be about 4 [Gajski 82]. This surprisingly low number is due to the style of programming enforced by the traditional languages. There are special purpose exceptions to this rule, and the hope is that non-traditional parallel languages will encourage more concurrent algorithms. Compilers for parallel machines can take advantage of techniques such as *and*, *or*, and *stream* parallelism if AI languages support concurrent control structures that will gives rise to them. But the jury is still out as to the amount of speed up such techniques can deliver.

Another problem in parallelism is failure to take the entire systems context into account. Before building a parallel machine one must not only simulate the machine but determine how to write large programs for it. This will reveal potential flaws in the machine before commencing with time consuming hardware development. If however the simulation does not properly take scheduling and technologically realistic hardware communication overhead into account, the timings produced will have little or no connection to reality.

Good examples of software systems that have not taken realistic hardware considerations into account are some of the parallel Lisps that have been proposed, such as [Gabriel 84]. These proposals point out places in Lisp-like processing where multiple processors could be exploited, but they do not analyze the overheads incurred. They usually assume that multiple processors are sharing a single large main memory where cons cells and other lisp objects are being stored. This is equivalent to assuming that memory is infinitely fast, which is just as un-realistic as assuming that processors are infinitely fast. The problem is that with current technology a *single* well designed Lisp processor could run faster than current mass memory technology could service it. Adding additional processors would thus not result in any throughput increase. There are several reasons why designers of parallel Lisps may have missed this fact. Perhaps one is that current 68000 Lisps are not memory bound. Another is the potential use of caches to reduce the required memory bandwidth to each processor. However even with caching, the number of processors that can be added is not unlimited; a 90% hit rate cache would allow only ten processors. What about the thousand processor architectures desired? Finally, experimental data shows that a *single processor* can run significantly faster than memory can service it: one must employ a cache just to keep a single processor running full tilt! The lesson is that processors are still much faster than (bulk) memories, and any sharing of data between multiple processors (beyond a few) must be done with special communication channels. In other words, MIMD machines with a single shared memory are a bad parallel architecture. This has important implications for some AI paradigms, such as Blackboard systems and Production systems that (in their current forms) rely upon memory for communication between tasks.

This is not to say that there are not opportunities for spreading Lisp like processing across hundreds of processors. There are many techniques other than a single shared memory system for connecting processors. More realistic areas of research are the spreading of parallel inference computation via techniques of *and*, *or*, and *stream* parallelism. The point is that all of these techniques incur some overhead, and one cannot simply solve the parallel computation problem by saying that arguments to functions should be evaluated in parallel. One must first study hardware technology to determine what at what grain sizes parallelism is feasable, and then figure out how to make AI language compilers decompose programs into the appropriate size pieces.

6 Generic AI problems for custom VLSI

One of the main hopes for more efficient computation in the future is the use of custom VLSI to accelerate particular functions. The ideal functions for silicon implementation should currently be bottlenecks in AI systems, and generic to many AI tasks. Four classes of operations were identified that fit this description: symbolic matching of abstract objects, semantic associative memory, parallel processor communication, and signal to symbol processing. Each will now be examined in detail.

6.1 Matching and Fetching

The concept of *matching* two objects is a general and pervasive operation. Most AI languages define one or more *match* functions on their structured data types (such as frames.) Some of these match functions are very ad-hoc (thus supposedly flexible), but others are sub- or super-sets of unification. If significant support for matching is to be provided in hardware, the match function must have well defined semantics.

When a match function is applied to a data base of objects, the operation is called *fetching*. In this case matching becomes the inner loop operation, and this is a context in which matching should be optimized. An ideal solution would integrate matching circuitry in with memory circuitry, so that fetching would become a memory access of a content addressable memory (CAM). The choice of match function is critical. To obtain reasonable memory densities, the relative silicon area of match circuitry cannot overwhelm that of the memory circuitry. Unfortunately, full unification and more complex match functions require too much circuitry to be built into memory cells. But if a formal subset of unification could be built in, then the CAM could act as a pre-filter function for unification.

The primary source of unification complexity is the maintenance of the binding environment. The match function of *mock unification* resembles full unification, except that all variables are treated as "don't cares", and no binding list is formed. It is the most powerful subset of unification that is state-free. Because of this, mock unification is a suitable candidate for integration into VLSI memory. We name associative memory systems that utilize mock unification as their match function CxAM's: ConteXt Addressable Memories.

From a hardware point of view, designing associative memory architectures involves a resource tradeoff between processing and memory: the more hardware devoted to "matching" the more data that can be examined in parallel, leading to faster search time per bit of storage. But conversely, the more matching hardware there is, the smaller the amount of hardware that can be devoted to data memory, and the lower the density of the associative memory. The data path widths of the match hardware is also a factor in making these tradoffs. Therefore associative memories can be rated by their storage density (bits stored per unit silicon area) and search throughput (bits searched per unit time per unit silicon area).

We examined two classes of associative memory in which the match function is mock unification. One integrated the matching circuitry in with memory circuitry, the other was hash based. Hashing was considered because in many applications in the past software hashing has dominated CAM technology [Feldman 69]. In more detail the two classes are:

1 Brute force search. The contents of a memory is exhaustively searched by some number of parallel match units. For this class of search a custom VLSI mock unification memory architecture was designed.

2 Hashing. Objects to be fetched are hashed, and then the collision list is serially searched by a match unit. A proposed VLSI implementation of PEARL's hashing scheme (called the HCP: Hash Co-Processor) served as an embodiment of hash based searching. In this system the bit storage is conventional DRAM.

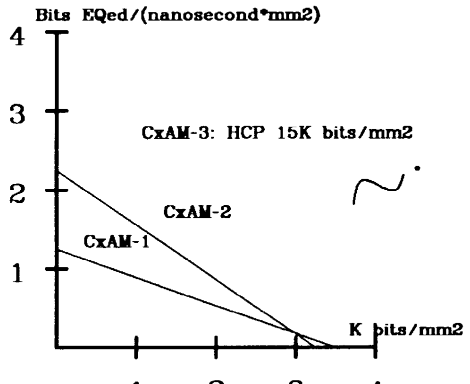

Figure 1

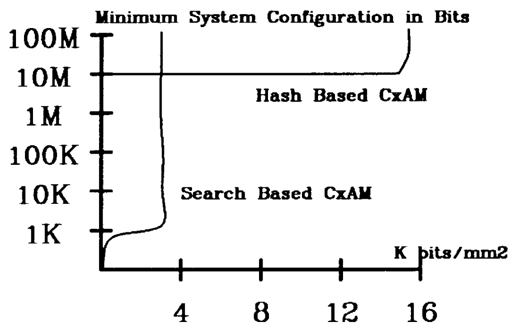

Figure 2

Figures 1 and 2 display graphs of CxAM design space trade offs. In figure 1 the range of bit and search power densities are displayed. The hash based CxAM has a single operating point because the fetch time is essentially independent of memory size, as is the density. The search based CxAM has a variable range because one can vary the relative proportions of storage and processing in such architectures. The two lines represent two different search based architectures. One has inherently better bit density, but over most of the design space this advantage is negated by an inherently worse search throughput. However neither design completely dominates the other, a choice between the two will depend upon the relative storage density - match throughput balance desired. In figure 2 the defect of the hashing CxAM is displayed. The minimum usable size system is too large for some applications.

Thus the trade-offs between these two schemes turn out to be in density and minimum usable size. As a representative data point, both techniques could perform a mock-unification of their entire local memory contents for an average query (an S-expression of length 16) in 5μs. The density of the search based CxAM was about eight times worse than that of conventional single transistor DRAM. The hashing scheme utilized conventional DRAMS, and so had high density. But the minimum configuration of a hash based CxAM memory system utilizing standard 256K DRAMs is 10 megabits, where as the search based CxAM can be configured for much smaller system storage sizes.

This extreme high speed of 5μs portends very efficient systems for those bottlenecked by data base fetch time. But which technique should be used is very dependent upon grain size. If one were constructing a large non-parallel machine, a bank of HCP's and conventional DRAMS would work well. But for an array of small grain processors with on-chip memories, the search based CxAM approach is more tractable.

By combining a CxAM with software based routines, a range of tailored matching services can be provided, with sliding power-price/throughput trade-offs. The design of the FAIM-1 machine provides an example of this. For each of thousands of processors, there is parallel CxAM hardware for mock unification, a single (pipelined) serial hard-wired full unifier, and software support for post-unification matching features (attached predicates and demons). With such a hardware/software hierarchy, simple matches (like Lisp's equal) will run fast, whereas more complex matching services (such as KRL's [Bobrow 77]) would cost more in time due to the software component.

In summary, matching is a common operation ripe for VLSI implementation, but the complexity of match functions varies by orders of magnitude. Below a simple list of match operations and data types are arranged in order of complexity. Successful high performance AI machines will have to carefully decompose these function into hardware and software components.

Match Hierarchy	
Match Operation	Object Type
Compare Instruction	32 bit data object
Lisp EQ Function	Atomic Lisp Objects
Lisp EQUAL Function	S-Expressions
Mock Unification	S-Expression with don't cares
Unification	S-Expression with Matching Variables
Unification & Predicates	S-Expression with Variables/Predicates
Arbitrary User Code	Arbitrary User Representation Objects

6.2 Parallel Processor Communications

As mentioned several times previously, when utilizing a number of processors in parallel, they cannot communicate objects and messages by sharing a large common memory. Some sort of special message passing (and forwarding) hardware is absolutely essential for efficient handling of the traffic. In many general purpose parallel processors, interprocessor communication is *the* computational bottleneck.

6.3 Signal to Symbol Processing

Despite all attention given to speeding up high level symbolic computation, within some AI applications the main processing bottleneck has been in the very low level processing of raw sensory data. Within many vision systems 90% or more of the run time may be incurred in the initial segmentation of the visual scene from pixels to low level symbolic constructs [Perkins 78]. Moreover limitations of the higher level vision processing usually are traceable to an inadequate initial segmentation [Deering 81b]. Similar problems arise in many speech systems. In such cases one should look to special purpose VLSI processors to directly attack the problem. Examples include special image processing chips, such as [Kurokawa 83], and speech chips, such as [Burleson 83]. As array processors have shown us, for these special processors to be usable by programmers, they need to be very well integrated with the other hardware and software components of the system, and as transparent as possible to the programmer. As most AI programmers are not good microcode hackers, one is in trouble if this is the only interface with a special device.

7 Conclusion

Opportunities for increased efficiency are present at all levels of AI systems if we only look, but to obtain the orders of magnitude throughput increases desired all these potential improvements must be made. We must make hard trade offs between traditional AI programming practices and the discipline necessary to construct algorithms than can make effective use of large multiprocessors. We must compile our AI languages, and these compilers must influence instruction set design. Key computational bottlenecks in AI processing must be attacked with custom silicon. There is a real need to use concurrency at all levels where it makes sense, but the overhead must be analyzed realistically.

Acknowledgments

The author would like to acknowledge the contributions of members of the FAIM-1 project: Ken Olum for his collaboration on the instruction set benchmarks, Ian Robinson and Erik Brunvand for their VLSI CxAM designs, and Al Davis for overall architectural discussions.

REFERENCES

[Bobrow 77] D. Bobrow and T. Winograd, "An overview of KRL-0, a knowledge representation language," *Cognitive Science*, Vol. 1, No. 1, 1977.

[Burleson 83] "A Programmable Bit-Serial Signal Processing Chip," SM Thesis, MIT Dept. of Electrical Engineering and Computer Science, 1983.

[Deering 81a] M. Deering, J. Faletti and R. Wilensky, "PEARL — A Package for Efficient Access for Representations in LISP," in *Proc. IJCAI-81*, Vancouver, B.C., Canada, Aug. 1981, pp. 930-932.

[Deering 81b] M. Deering and C. Collins, "Real-Time Natural Scene Analysis for a Blind Prosthesis," in *Proc. IJCAI-81*, Vancouver, B.C., Canada, Aug. 1981, pp. 704-709.

[Deering 84] M. Deering and K. Olum, "Lisp and Processor Benchmarks," unpublished FLAIR Technical Report, March 1984.

[Fateman 78] R. Fateman, "Is a Lisp Machine different from a Fortran Machine?," *SIGSAM* Vol. 12, No. 3, Aug. 1978, pp. 8-11.

[Feldman 69] J. Feldman and P. Rovner, "An Algol Based Associative Language," *Commun. ACM*, Vol. 12, No. 8, Aug. 1969.

[Foderaro 83] J. Foderaro, "The Franz Lisp System," unpublished memo in *Berkeley 4.2 UNIX Distribution*, Sept. 1983.

[Gabriel 84] R. Gabriel and J. McCarthy, "Queue-based Multi-processing Lisp", preprint, 1984.

[Gajski 82] D. Gajski, D. Pradua, D. Kuck and R. Kuhn, "A Second Opinion on Data Flow Machines and Languages," *IEEE Computer*, Vol. 15, No. 2, Feb. 1982, pp. 58-69.

[Genesereth 83] M. Genesereth, "An overview of Meta-Level Architecture," in *Proc. AAAI-83, Washington, D.C., 1983.*

[Griss 82] M. Griss and E. Benson, "Current Status of a Portable Lisp Compiler," *SIGPLAN*, Vol. 17, No. 6, in *Proc. SIGPLAN '82 Symposium on Compiler Construction*, Boston, Mass., June. 1982, pp. 276-283.

[Knight 81] T. Knight, Jr., D. Moon, J. Holloway and G. Steele, Jr., "CADR", MIT AI Memo 528, March 1981.

[Kurokawa 83] H. Kurokawa, K. Matsumoto, M Iwashita and T. Nukiyama, "The architecture and performance of Image Pipeline Processor," in *Proc. VLSI '83*, Trondheim, Norway, Aug. 1983, pp. 275-284.

[Lampson 80] B. Lampson and K. Pier, "A Processor for a High-Performance Personal Computer," *Proc. 7th Symposium on Computer Architecture*, SigArch/IEEE, La Baule, May 1980, pp. 146-160.

[Perkins 78] W. Perkins, "A model based vision system for industrial parts," *IEEE Trans. Comput.*, Vol. C-27, 1978, pp. 126-143.

[Warren 77] D. H. Warren, "Applied Logic — Its Uses and Implementation as a Programming Tool," Ph.D. Dissertation, University of Edinburgh, 1977, Available as Technical Note 290, Artificial Intelligence Center, SRI International.