Appraisal of Parallel Processing Research at ICOT

E. Tick

CIS-TR-93-03
March 1993

Abstract

A major goal of the Fifth Generation Computer Systems project was the development of high-performance multiprocessors for symbolic applications. The key technologies envisioned as critical were fine-grain concurrent languages and custom microprocessors for directly executing those languages. In contrast to this top-down design approach were most other research laboratories, concentrating on further refining scalable architectures by speeding up communication. One main evolutionary trend was towards threaded architectures that supported latency hiding by fast task switching and low-cost message passing. This article attempts to compare these top-down and bottom-up approaches.

This article appears in *Future Generation Computer Systems*, North-Holland, vol. 9, no. 2, 1993.

Department of Computer and Information Science
University of Oregon

1 Introduction

This article critiques the Parallel Inference Machine (PIM) research within the Fifth Generation Computer Systems (FGCS) project conducted at ICOT over the period of 1982–1992. I will direct my attention almost exclusively to hardware (processor, memory, and network designs) and performance. Cost is also a critical factor, but one that is difficult to accurately evaluate, and is perhaps less important because all PIMs to date are experimental prototypes. The tack of this article is to compare the PIMs with other recent experimental architectures and research machines, specifically *not* commercial products. The latter is an unfair comparison primarily because the PIMs are symbolic, not numeric, processors, and the PIMs are not Unix-based, which limits their accessibility.

The key criterion in high-performance processor design (for single applications) is the *reduction* and/or *hiding* of latency, primarily that of memory. For a sequential microprocessor this requires some memory hierarchy, normally consisting of registers, multi-level caches, primary physical memory, and virtual memory. Since CPU technologies are growing in speed at a faster rate than memory (SRAM and DRAM) technologies [20], one can expect processors to have increasingly higher penalties, in cycles, when accessing memory levels far from the CPU. In a multiprocessors, latency reduction can be accomplished via sophisticated network routers (e.g., [12]) and relaxed cache coherence methods [39]. Router technology has improved over the past decade, making virtual shared-memory models efficient. By building dual-processor nodes that both compute and communicate concurrently, effective network bandwidth is greatly increased.

In distributed-memory multiprocessor design, latency hiding [3, 40] can be accomplished by "threading" together small grain tasks, quickly switching one task for another if the former task undergoes a long latency action [51, 56]. This can be viewed as increasing the throughput of thread execution. Switching tasks on remote memory accesses[1] requires some kind of tagging mechanism for returning the value to the requester.

To make threaded architectures work efficiently, a few constraints must be met. First, tasks must be plentiful, because substitutes must be available to switch in. Second, tasks must be lightweight, because the time to switch a task must be low, otherwise switching overhead will alleviate much (or all) of the gain of latency hiding. Plentiful and lightweight tasks are synergistic requirements because paradigms that decompose computation into its smallest components naturally derive many, many such components. However, third, tasks cannot be *too* lightweight, or else the frequency of task

[1]called a "split phase" memory transaction in dataflow parlance.

invocation (if not sufficiently supported by hardware) will incur excessive overhead reducing throughput: "medium" weight is best.

A fourth criterion, related to language design rather than architecture, is also considered (by some, including ICOT) critical for successful multiprocessing: efficient and correct means of task synchronization must be found, to ensure that parallel programs intended to be determinate *are* determinate. This is a key motivation behind applicative dataflow and logic languages.

Dataflow languages [4] and concurrent logic languages [48] were the first versatile programming languages to allow natural translation into threads with simple compilation. Dataflow architectures were handicapped because the granularity was too fine, until techniques where developed to efficiency invoke threads (removing token-matching store bottleneck), and thicken threads (partition the data-dependency graph to increase, up to an optimum, the required work per thread) [41, 58]. Both communities were handicapped by software engineering concerns: their perceived radical programming methodologies, and by continually changing, non-standardized languages. Furthermore, all commercial efforts in multiprocessor construction were based firmly on imperative languages with no implicit parallelism. To exploit the previous ideas for such languages requires sophisticated compilers: either to decompose a program into dynamically scheduled threads, or more complex still, into statically scheduled threads as in a VLIW machine, e.g., [7]. With time however, dataflow architectures have evolved to threaded architectures (e.g.,[10]), similar in concept but not in hardware organization, to the architectures originally proposed by ICOT for the PIM project.

As elegantly summarized by Arvind and Iannucci [3], the two key architectural features necessary for massive multiprocessors are the ability to hide latencies and to efficiently synchronize tasks.[2] These two criteria are at odds in Arvind's fine-grained model where latency hiding occurs at the lowest level of instruction operands, perhaps with reservation registers [57]. Thus to allow more execution concurrency within the instruction stream, more registers are needed, and therefore task switching (e.g., due to an external interrupt) time increases because the processor state is larger. This is the general idea in the HEP [51] and Tera [56] machines: processes are composed of threads that are rapidly and frequently swapped to hide all latencies.

The ICOT PIMs are founded on a more uniform model of medium-grain tasks, i.e., Horn clauses synchronized on logic variables. The custom processors developed for the PIMs do not rapidly swap tasks to hide all latencies — any latency hiding is at a higher level, and controlled by software. For example, on a cache miss, the PIM CPUs all idle

[2]Synchronization can incur task suspension, for which task switching is favored over busy waiting: I therefore loosely refer to "task switching" as the action (and its related cost) most closely associated with synchronization.

until the request is satisfied from memory. However, for a reference located on another node/cluster, the requesting task is suspended by a microcode handler. In some sense, the PIM model is more similar to threads in Mach [6], than to threaded architectures. However, PIM tasks may tend to be smaller than Mach's because they do not have associated stacks (since they are not derived from sequential execution), although PIM tasks can have peripheral data structures for suspension management.

This article is organized as follows. In Section 2 an overview of the PIMs is presented. Sections 3 and 4 describe the PIM/m and PIM/p designs in more detail. Section 5 presents an overview of alternative experimental multiprocessors based on threaded architectures. Conclusions are summarized in Section 6.

2 Overview of PIM Hardware Technology

The main components of the PIM designs include the following (see Hirata *et al.* [21] and Taki [54] for summaries of these machines):

- memory management: concurrent logic programs have a high memory bandwidth requirement because of their single assignment property and lack of backtracking. ICOT has developed an integrated solution for garbage collection at three levels within the PIMs. Locally, incremental collection is performed with approximative reference counting. Specifically, Chikayama's Multiple Reference Bit (MRB) is incorporated in each data word [8]. Distributed data is reclaimed across clusters via their export tables. Finally, if local memory is filled, a parallel stop & copy garbage collector is invoked in the cluster [23]. The effort in designing and evaluating alternative garbage collection methods is one of the most extensive of all ICOT projects (e.g., [38, 24, 17, 33]), primarily because the problem was recognized several years ago. However, compilation techniques, such as *statically* determining instances of local reuse (e.g., [53]), were not explored.[3] The tradeoffs between static analysis time *vs.* runtime overhead are still open questions for such techniques.

- scheduling: concurrent logic languages have inherently fine-grain process structures. The advantage is exploitable parallelism, but the disadvantage is the potential of a thrashing scheduler. The PIMs rely on explicit inter-cluster scheduling with *goal pragma* (an attribute telling where the goal should be executed), and implicit (automatic) load balancing within a cluster. Furthermore, goals can be

[3]The PIM compilers use MRB in conjunction with hardware support to *dynamically* reuse memory, at some cost in execution complexity compared to static methods.

assigned priorities, which steer load balancing in a nonstrict manner. Although functional mechanisms have been completed at ICOT, extensive evaluation has not yet been conducted. Higher-level programming paradigms, such as *motifs* [15], have not been designed to alleviate the complexity of user development and modification of pragma.

- meta-control: the PIMs are based on Flat Guarded Horn Clauses (FGHC, and a superset called KL1) which by virtue of its simplicity cannot reflect upon failure and does not have atomic tell unification. Failure of conditional tests in guards cause execution to try alternative clauses; however, body goal failure is terminal. Thus erroneous user-program failure will propagate through an unprotected operating system also written in FGHC. Atomic tell unification is allows a richer variety of guard tests that can make bindings conditional upon eventual commitment. Without this capability (which is very complex to implement), output bindings can be made only *after* commit (see [49] for more details). These features are desirable in operating systems, although perhaps less so for application programs. To make up for this loss, ICOT developed protected tasks called *shoen* [22], similar to work by I. Foster [14]. Functional mechanisms for shoen management have been implemented at ICOT over the past four years [34, 42, 43], although empirical measurements of the operating system running applications programs have not yet been made. Intra-cluster mechanisms include foster-parent (a shoen's local proxy) termination detection and deadlock detection; inter-cluster mechanisms include weighted throw counts for shoen termination detection and intelligent resource management [21, 25]. Without further empirical data, it is difficult to judge the effectiveness of these mechanisms for reducing runtime overheads.

- unification: unification is peculiar to logic programs, and somewhat controversial, in the general computing community, in its utility. Concurrent logic programs reduce general two-way unification into *ask* and *tell* unifications, which correspond more directly to importation and exportation of bindings in imperative concurrent languages. Still, logical variables cause two serious problems.

  First, variables are overloaded to perform synchronization. This is both the beauty and horror of concurrent logic languages. The programmer's model is simplified by implicit synchronization on variables, i.e., if a required input variable arrives with no binding, the task suspends. Furthermore, if at any time that variable receives a binding (anywhere in the machine), the suspended task is resumable. Implementing this adds overhead to the effective binding time, primarily because mutual exclusion is required during both binding and suspension/resumption management.

Second, in a distributed environment, optimizing data locality over a set of unifications of arbitrary data structures is a difficult problem. Message passing mechanisms defining import/export tables and protocols were developed [22], but little empirical analysis has been published.

Compilation techniques to determine runtime characteristics of logical variables, such as "hookedness" [60] and modes [59], and exploit them to minimize suspensions or memory consumption, have not yet been implemented in the current PIM compilers.

The ICOT research schedule followed the development of the personal inference machines (PSI-I,II,III) [55, 35], followed by mockup PIMs (Multi-PSI-V1/2, built of PSI-IIs), and finally the various PIMs: PIM/p (Fujitsu) [29], PIM/m (Mitsubishi) [36], PIM/i (Oki) [46], PIM/c (Hitachi) [32], and PIM/k (Toshiba) [5]. A great deal of credit must go to ICOT's central management of these efforts, based on a virtual machine instruction set called PSL used to describe a virtual PIM (VPIM) running on a Sequent Symmetry [54]. VPIM was shared (with slight modifications) by most of the member organizations, making design verification feasible. These designs are summarized in Taki [54]. Highly parallel execution of dynamic and non-uniform (but explicitly *not* data-parallel) applications is cited as the target of the project. The major design decisions were made for MIMD execution, of a fine-grain concurrent logic programming base language, on scalable distributed-memory multiprocessors.

In the following sections, I review PIM/m and PIM/p because they represent the main hardware efforts of the FGCS project.

3 PIM/m

PIM/m, derived from the Multi-PSI testbed, is a 16x16 mesh of processors, where wormhole routing is used for communication. Each node is built from three custom VLSI parts: CPU, cache, and router. The node has 80 Mbytes of local memory and split I/D caches of 5 Kbytes and 20 Kbytes respectively. The CPU directly executes KL1 [26] in a five-stage pipeline, under microcode control. The pipeline has an interesting feature of dereferencing operands within the fourth pipe stage (interacting with memory). No published data is yet available concerning the performance of the pipeline or network. In this section I will attempt to extrapolate some gross conclusions from experiments on the Multi-PSI [34], since the two machines use the same operating system, PIMOS [9].

PIMOS, and the PIMs, do not attempt to hide the latency of remote accesses at a low level. The hardware design philosophy surrounding the PIM effort was to exploit faster communication, not to speed up task switching within what are now termed "multi-threaded" architectures. However, some latency hiding naturally occurs. First, a task is forked by an explicit *pragma*, which sends a `throw_goal` message to a remote node in PIM/m. Input data arguments to the task, if complex, are not sent, but indirectly referenced. Note also that goal priority can be specified in a pragma, giving some pseudo real-time control.

To access these complex arguments, the callee must send a `read` message, and receive an `answer_value` message. This lazy approach hopes to reduce overall communication, for example when the forked task invocation can proceed without referring to (all of) its input parameters. There has been no empirical evidence published showing if this method is superior to the alternative wherein the entire task and its data are thrown initially. It appears as if static analysis, to determine critical arguments for eager communication, would be beneficial. This protocol is too complex if not supported by fast and seamless communication (a fast network/processor interface). However, in some sense, the PIM/m is meant as a network testbed for PIM/p, where latencies are reduced by clustering processors. There the software is responsible for keeping inter-cluster communication to a minimum.

Within the body of a task, term unification is used to match and create data structures. In general, KL1 allows active unification of arbitrary terms, although programs very rarely unify anything more than a term to an unbound variable. Latency is hidden by treating body unifies as bonefide goals, not inlining them. A unify goal will either immediately execute locally, or if an operand is a remote reference, the goal itself is sent as a message requesting action. (Note that unification almost always acts as assignment, and if implemented properly, none of the overheads of full unification should be visible to the execution). As in the previous case, this method of latency hiding has been implemented, but not stressed, in the PIM designs. For example, extremely fast task switching has not been achieved, which is critical for the latency to be effectively hidden. Methods such as dynamic fattening of messages (increase message size, up to an optimum, to effectively offset task switch overhead) in data-parallel C [19] have not been implemented.

Sophisticated distributed garbage collection mechanisms have been developed for PIMOS and PIM/m, e.g.,[22], the details of which are beyond the scope of this paper. This is one area that the ICOT research has far surpassed that within the U.S., probably because garbage collection has only recently become an issue in the U.S. (after seminal work for Lisp and SmallTalk) with the increasing popularity of C++ and ML.

The PIM/m router chip is dedicated to communication among nodes and does no part of message packet (de)construction. One conclusion of Nakajima's analysis [34] was that the message-handling costs in Multi-PSI (and I assume PIM/m) are very high. For example, forking a task with three arguments (65 byte packet) takes 419 (microcode) cycles. Sending a read message and receiving the answer value (for a list pair) takes 117 and 397 cycles respectively. These measurements do not include network transfer time. The Multi-PSI microprocessor, a PSI-II [35] with 200 nsec cycle time, is sufficiently slow to make network performance assessments dubious. For example, Nakajima [34] showed that the overdesigned 5 Mbyte/second network is scalable up to 1000 PEs (i.e., the network should not significantly degrade overall performance up to this saturation point). His benchmarks showed, however, that on 64 PEs (full system), about 20% of the processor cycles were idling (see comments below).

PIM/m CPUs have a 65 nsec cycle time, but are pipelined, making direct comparison to PSI-II difficult. Nakashima *et al.* [36] presented measurements that PIM/m ran 1.5–4.8 times faster than Multi-PSI on one processor. The average speedup of their four benchmarks is 2.5, attributing the poor showing to inefficient task switching that takes "dozens of cycles for execution in the E[xecution] stage" of the pipeline. The PIM/m uses the same network as Multi-PSI. Nakashima observed multiprocessor efficiencies on three large problems ranging from 82%–85%. The benchmarks chosen for that study were a double-edged sword: on the one hand, they could produce any amount of parallelism given large enough inputs, allowing for the demonstration of speedups.[4] On the other hand, given unlimited potential parallelism, the PIM/m was still losing up to 18% efficiency.

Since communication bandwidth was sufficient, the consistent results in both these studies are mainly attributable to two effects: 1) inherent inefficiency at latency hiding, and 2) load balancing and cold start effects. Although the benchmarks measured were highly parallel, in some, process structure implied cold start/finish effects that could account for a portion of the idle cycles. Furthermore, certain benchmarks used software-implemented on-demand load balancing that might have been imperfect. Further studies are needed to determine the exact proportion of the various effects.

4 PIM/p

The PIM/p [29] consists of a six-dimensional hypercube, each node of which is a shared-memory cluster of eight processors (for a total of 512 processors). The PIM/p organization is similar in concept (but not cache coherency protocol) to DASH [30], although

[4]Without, I may add, regard for solving the problem efficiently.

the PIM/p is very much customized to execute KL1. Latency is further reduced, with respect to PIM/m, by the cache-based shared-memory clusters. Each processor has two 64 Kbyte I+D snoopy caches.

The PIM/p custom microprocessor pipeline has an interesting feature allowing dynamic insertion of instruction sequences triggered by macro instructions. This is useful for implementing high-level operations, such as dereference, with less code expansion. The pipeline however executes a single instruction stream, and no support of multiple contexts to speed task switching is given. Recent architectural development of "active messages," e.g., [13, 44], are an interesting comparison to the PIM/p pipeline. An active message is essentially a message that can be sent directly to a remote processor's pipeline and executed directly. The EM-4's [44, 47] "fused pipeline" is an example of such a scheme. The advantages over conventional message passing protocols can be significant: messages require no interpretive overhead. It should be noted however that the EM-4 is primitive compared to the PIMs in that it does not support memory synchronization in hardware (i.e., logical variables or I-structures), resulting in very poor performance for many naturally-parallel algorithms.

Each PIM/p processor is supported by a network interface processor that constructs packets, sends packets to the router, and receives packets from the router. Actually, each cluster has two wormhole routers, allowing for two hypercube networks, doubling bandwidth. No evaluation of the network has been published. In general, the two-level hierarchy presents an irregular model to mapping and load balancing algorithms that has not yet been conclusively solved by ICOT (see [16, 32, 52]).

Within a cluster, unique architecture features include dynamic garbage collection of data cells based on binary reference counting [8, 38], and split I/D copyback caches with protocols that can reduce bus traffic by exploiting properties of the KL1 software implementation [18]. The utility of these features has been evaluated, but extensive comparisons to alternatives has yet to be done.

Published measurements [29] of the PIM/p are for eight PEs configured either within a single cluster or as eight nodes of the hypercube. A single, simple benchmark was measured (a multiple-solution search with sufficient parallelism), showing 93.8% and 91.0% efficiency on the two configurations, respectively. This is not sufficient data to extrapolate the performance of real applications on a full system, especially since raw execution breakdowns are not given.

5 Other Approaches

The recent evolution of dataflow architectures into threaded architectures occurred concurrently with the hardware design and development of the PIMs within the FGCS Project. It helps to clarify the PIM designs in light of the recent excitement [13, 37, 47]. There are several experimental machines serving as examples of threaded architectures: Alewife, EM-4, J-machine, *T, TAM, and Tera. The concepts are roughly the same in all, as outlined in Section 1. Monsoon [41] and Tera [56], in the spirit of the HEP [51], swap threads each cycle.[5] The EM-4 [44] and J-machine [11] implement types of active messages wherein there is essentially no overhead to received and execute a message. Alewife [1] and *T [37] incorporate multiple contexts built within the commercial SPARC and Motorola 88110 microprocessors, respectively. In this section, I further compare the TAM [10] threaded architecture proposal with the KL1 architecture [26].

Both TAM and KL1 have threads represented by activation frames (called "goals" in KL1) holding local variables. TAM however can represent more than one thread within a frame. The distinction is that each procedure (called a "code block" in TAM) may be split into multiple threads, whereas in KL1 there is a single thread. Because KL1 has recursion and no iteration (as there is, for example, in Id [2]), the threads end up being approximately the same small granularity. For example, in both systems, nonlocal array references effectively suspend. In TAM, an array reference is compiled as an explicit thread, whereas in KL1, the reference is through a system-builtin body goal that can suspend. There is equal flexibility in both models with respect to the compilation of arithmetic expressions. In KL1, arithmetic can be split out as a separate task to increase concurrency, or generated in-line to increase granularity.

Further differences occur in synchronization and scheduling. At a high level, thread synchronization is similar among the models: TAM uses I-structures and KL1 uses logical variables. At a lower level, TAM gains efficiency by exploiting the functionality of its source languages, in addition to compiler analysis. The number of "synchronizers" needed to fire a thread is determined statically, and the synchronization count is atomically decremented as arguments (I-structures) are bound. The KL1 systems do not attempt mode analysis to exploit this information, and thus goals can potentially resume and resuspend multiple times before all necessary arguments are present (furthermore, resumption always starts at a single entry point). Also, TAM retains the information representing the call tree, i.e., the parents and children relationships among the active frames. In all KL1 systems, this information is lost, and frames/goals

[5]This is not precisely accurate for Monsoon: a thread will retain control of the pipeline because no additional long latency actions can occur, as guaranteed by the compiler.

are kept in pools represented by queues.

TAM introduces another level of abstraction, called a *quantum*, which is a group of threads that actively execute from the register set until none of the group are enabled. The hope is that a compiler can do intelligent register allocation across threads in a quantum. KL1 has no such concept, and currently no compilers do global register allocation on this form. There is a related abstraction called a *time slice* wherein a KL1 thread cannot pass control tail-recursively too many times in succession — the system will force suspension, enabling more fair scheduling.

Empirical performance analyses of TAM are at more preliminary stages than those of KL1, so comparison is difficult. In summary, the models have more similarities than differences. The PIM design work did not focus on compiler analyses, which are necessary to derive the functionality of programs allowing more streamlined execution.

6 Conclusions

The FGCS PIM organizations are more closely related to the threaded architectures recently evolved from dataflow, than has been documented in the literature by either group.[6] The KL1 language is in some sense "threaded" because it is founded on small-grain tasks that synchronize on data dependencies. Threaded architectures in "conventional" parallel processing derived from the unconventional dataflow processor effort, going strong for the past several years. With respect to hardware, the key motivations were to speed up network communication and to improve the efficiency of the network/processor interface. Seminal research in the former direction include the J-machine [11], and in the latter direction include the EM-4 [44, 47] and Monsoon [41].

With respect to software, the key motivation was to thicken the too-small granularity of dataflow packets into "threads" of sequential computation triggered by limited inputs, and producing output(s). Sarkar [45], Culler [10], Traub [58] and others pursued *compile-time* techniques to thicken the threads. Although ICOT did not address these issues, others in the logic programming community have been, e.g., [27, 31].

Specialized microprocessor architectures had been developed for several experimental machines that support fast switching among threads, usually involving having multiple contexts in registers. ICOT did not pursue this approach to microprocessor design, instead placing emphasis on direct execution of symbolic, concurrent languages. Hand in glove with the evolution of fine-grain architectures of all types was the refinement of network interfaces that support fast message passing. The state-of-the-art is illus-

[6]Interestingly, H. Tanaka's group at the University of Tokyo, closely involved with ICOT, *has* developed a link: the PIE64 machine [28, 50].

trated by machines such as the EM-4 where active messages are merged directly into pipeline execution. ICOT did not lead, but rather followed, in the design of network and network-interface architectures.

By pushing a top-down design approach, ICOT has tackled some problems in great depth that other research groups have only recently, or not yet, started to explore. These include: 1) dynamic garbage collection; 2) dynamic load balancing; and 3) symbolic applications development. However, the current slew of PIMs do not achieve their potential performance because of several design flaws: 1) lack of compiler analysis to exploit functionality (or can be viewed as insistence on sticking with KL1 instead of a moded/strongly-typed form of the language); 2) lack of sophisticated "fused" pipelines handling both incoming messages as well as local instruction stream; 3) lack of multiple context support for fast task switching, and 4) reliance on custom microcoded microprocessors instead of leveraging generic microprocessors.

It is somewhat unfair to make the comparison with current threaded architectures because the PIMs designs are circa 1988. Interestingly, the PIMs are built on a strong foundation of an applicative, concurrent language (as opposed to multiprocessors built for imperative, sequential languages). It should be possible to leverage this main strength in further refining the PIM compiler and processor designs to achieve very high performance.

Acknowledgements

The author is supported by an NSF Presidential Young Investigator award, with matching funds from Sequent Computer Systems Inc. I thank Xiaoxing Zhong for many insightful discussions on parallel processing and architecture. Zena Ariola, Bart Massey, and Katsuto Nakajima gave helpful comments on early drafts of this paper.

References

[1] A. Agarwal, B-H Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *International Symposium on Computer Architecture*, pages 104–114. Seattle, IEEE Computer Society Press, May 1990.

[2] Arvind and K. Ekanadham. Future Scientific Programming on Parallel Machines. *Journal of Parallel and Distributed Computing*, 5(5):460–493, October 1988.

[3] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *DFVLR: Conference on Parallel Processing in Science and Engineering*. Bonn-Bad Godesberg, June 1987.

[4] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4), October 1989.

[5] S. Asano, S. Isobe, and H. Sakai. The Unique Features of PIM/k: A Parallel Inference Machine with Hierarchical Cache System. Technical Report TR-767, ICOT, 1-4-28 Mita, Minato-Ku Tokyo 108, Japan, April 1992.

[6] D. L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, pages 35–43, May 1990.

[7] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWARP: An Integrated Solution to High-Speed Parallel Computing. In *International Conference on Supercomputing*, pages 330–339, Orlando, November 1988. IEEE Computer Society Press.

[8] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *International Conference on Logic Programming*, pages 276–293. University of Melbourne, MIT Press, May 1987.

[9] T. Chikayama, H. Sato, and T. Miyazaki. Overview of the Parallel Inference Machine Operating System PIMOS. In *International Conference on Fifth Generation Computer Systems*, pages 230–251, Tokyo, November 1988. ICOT.

[10] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 164–175, Santa Clara, April 1991. IEEE Computer Society Press.

[11] W. J. Dally. The J-Machine System. In P. Winston and S. A. Shellard, editors, *Artificial Intelligence at MIT: Expanding Frontiers*, pages 536–569. MIT Press, 1990.

[12] W. J. Dally and C. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[13] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *International*

Symposium on Computer Architecture, pages 256–266. Gold Coast, IEEE Computer Society Press, May 1992.

[14] I. Foster. Logic Operating Systems: Design Issues. In *International Conference on Logic Programming*, pages 910–926. University of Melbourne, MIT Press, May 1987.

[15] I. Foster and S. Taylor. A Compiler Approach to Scalable Concurrent Program Design. Technical Report MCS-P306-0492, Argonne National Laboratory, 1992.

[16] M. Furuichi, K. Taki, and N. Ichiyoshi. A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI. In *SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 50–59, Seattle, March 1990. ACM Press.

[17] A. Goto, Y. Kimura, T. Nakagawa, and T. Chikayama. Lazy Reference Counting: An Incremental Garbage Collection Method for Parallel Inference Machines. In *International Conference and Symposium on Logic Programming*, pages 1241–1256. University of Washington, MIT Press, August 1988.

[18] A. Goto, A. Matsumoto, and E. Tick. Design and Performance of a Coherent Cache for Parallel Logic Programming Architectures. In *International Symposium on Computer Architecture*, pages 25–33. Jerusalem, IEEE Computer Society Press, May 1989.

[19] P. J. Hatcher and M. J. Quinn. *Data-Parallel Programming on MIMD Computers.* MIT Press, Cambridge MA, 1991.

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

[21] K. Hirata, R. Yamamoto, A. Imai, H. Kawai, K. Hirano, T. Takagi, K. Taki, A. Nakase, and K. Rokusawa. Parallel and Distributed Implementation of Concurrent Logic Programming Language KL1. In *International Conference on Fifth Generation Computer Systems*, pages 436–459, Tokyo, June 1992. ICOT.

[22] N. Ichiyoshi, K. Rokusawa, K. Nakajima, and Y. Inamura. A New External Reference Management and Distributed Unification for KL1. In *International Conference on Fifth Generation Computer Systems*, pages 904–913, Tokyo, November 1988. ICOT.

[23] A. Imai and E. Tick. A Shared-Memory Multiprocessor Garbage Collector and its Evaluation for Committed-Choice Logic Programs. In *Symposium on Parallel and Distributed Processing*, pages 870–877. Dallas, IEEE Computer Society Press, December 1991.

[24] Y. Inamura, N. Ichiyoshi, K. Rokusawa, and K. Nakajima. Optimization Techniques Using the MRB and Their Evaluation on the Multi-PSI/V2. In *North American Conference on Logic Programming*, pages 907–921. Cleveland, MIT Press, October 1989.

[25] Y. Inamura and S. Onishi. A Detection Algorithm of Perpetual Suspension in KL1. In *International Conference on Logic Programming*, pages 18–30. Jerusalem, MIT Press, June 1990.

[26] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *International Symposium on Logic Programming*, pages 468–477. San Francisco, IEEE Computer Society Press, August 1987.

[27] A. King and P. Soper. Schedule Analysis of Concurrent Logic Programs. In *Joint International Conference and Symposium on Logic Programming*, pages 478–492. Washington D.C., MIT Press, November 1992.

[28] H. Koike and H. Tanaka. Multi Contex Processing and Data Balancing Mechanism of the Parallel Inference Machine PIE64. In *International Conference on Fifth Generation Computer Systems*, pages 970–977, Tokyo, November 1988. ICOT.

[29] K. Kumon, A. Asato, S. Arai, T. Shinogi, A. Hattori, H. Hatazawa, and K. Hirano. Architecture and Implementation of PIM/p. In *International Conference on Fifth Generation Computer Systems*, pages 414–424, Tokyo, June 1992. ICOT.

[30] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *International Symposium on Computer Architecture*, pages 148–159. Seattle, IEEE Computer Society Press, May 1990.

[31] B. C. Massey and E. Tick. Sequentialization of Parallel Logic Programs with Mode Analysis. In 4^{th} *International Conference on Logic Programming and Automated Reasoning*, St. Petersburg, July 1993. Submitted.

[32] T. Nakagawa, N. Ido, T. Tarui, M. Asaie, and M. Sugie. Hardware Implementation of Dynamic Load Balancing in the Parallel Inference Machine PIM/c. In

International Conference on Fifth Generation Computer Systems, pages 723–730, Tokyo, June 1992. ICOT.

[33] K. Nakajima. Piling GC: Efficient Garbage Collection for AI Languages. In *IFIP Working Conference on Parallel Processing*, pages 201–204. Pisa, North Holland, May 1988.

[34] K. Nakajima. Distributed Implementation of KL1 on the Multi-PSI. In P. Kacsuk and M. Wise, editors, *Implementation of Distributed Prolog*, pages 311–332. John Wiley & Sons, Ltd., Sussex England, 1992.

[35] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine: PSI-II. In *International Symposium on Logic Programming*, pages 104–113. San Francisco, IEEE Computer Society Press, August 1987.

[36] H. Nakashima, K. Nakajima, S. Kondo, Y. Takeda, Y. Inamura, S. Onishi, and K. Masuda. Architecture and Implementation of PIM/m. In *International Conference on Fifth Generation Computer Systems*, pages 425–435, Tokyo, June 1992. ICOT.

[37] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *International Symposium on Computer Architecture*, pages 156–167. Gold Coast, IEEE Computer Society Press, May 1992.

[38] K. Nishida, Y. Kimura, A. Matsumoto, and A. Goto. Evaluation of MRB Garbage Collection on Parallel Logic Programming Architectures. In *International Conference on Logic Programming*, pages 83–95. Jerusalem, MIT Press, June 1990.

[39] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 24(6):52–60, August 1991.

[40] G. M. Papadopoulos, A. P. W. Bohm, and A. T. Dahbura. Multithreaded Computer Systems. In *International Conference on Supercomputing*, pages 772–775. Minneapolis, IEEE Computer Society Press, November 1992.

[41] G. M. Papadopoulos and D. E. Culler. Monsoon: an Explicit Token-Store Architecture. In *International Symposium on Computer Architecture*, pages 82–91. Seattle, IEEE Computer Society Press, May 1990.

[42] K. Rokusawa and N. Ichiyoshi. A Scheme for State Change in a Distributed Environment Using Weighted Throw Counting. In *Interational Parallel Processing Symposium*, pages 640–645, Beverly Hills, March 1992. IEEE Computer Society Press.

[43] K. Rokusawa, N. Ichiyoshi, T. Chikayama, and H. Nakashima. An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems. In *International Conference on Parallel Processing*, pages 18–22, Penn State, August 1988.

[44] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *International Symposium on Computer Architecture*, pages 46–53. Jerusalem, IEEE Computer Society Press, May 1989.

[45] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* MIT Press, Cambridge MA., 1989.

[46] M. Sato, K. Kato, K. Takeda, and T. Oohara. Exploiting Fine Grain Parallelism in Logic Programming on a Parallel Inference Machine. Technical Report TR-676, ICOT, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, August 1991.

[47] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and Y. Koumura. Thread-Based Programming for the EM-4 Hybrid Dataflow Machine. In *International Symposium on Computer Architecture*, pages 146–155. Gold Coast, IEEE Computer Society Press, May 1992.

[48] E. Y. Shapiro. A Subset of Concurrent Prolog and Its Interpreter. Technical Report TR-003, ICOT, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, January 1983.

[49] E. Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[50] K. Shimada, H. Koike, and H. Tanaka. UNIRED-II: The High Performance Inference Processor for the Parallel Inference Machine PIE64. In *International Conference on Fifth Generation Computer Systems*, pages 715–722, Tokyo, June 1992. ICOT.

[51] B. J. Smith. A Pipelined, Shared Resource MIMD Computer. In *International Conference on Parallel Processing*, pages 6–8. IEEE Computer Society Press, August 1978.

[52] M. Sugie, M. Yoneyama, N. Ido, and T. Tarui. Load-Dispatching Strategies on Parallel Inference Machines. In *International Conference on Fifth Generation Computer Systems*, pages 987–993, Tokyo, November 1988. ICOT.

[53] R. Sundararajan, A. V. S. Sastry, and E. Tick. Variable Threadedness Analysis for Concurrent Logic Programs. In *Joint International Conference and Symposium*

on Logic Programming, pages 493–508. Washington D.C., MIT Press, November 1992.

[54] K. Taki. Parallel Inference Machine PIM. In *International Conference on Fifth Generation Computer Systems*, pages 50–72, Tokyo, June 1992. ICOT.

[55] K. Taki, K. Nakajima, H. Nakashima, and M. Ikeda. Performance and Architectual Evaluation of the PSI Machine. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 128–135. Palo Alto, IEEE Computer Society Press, October 1987.

[56] M. R. Thistle and B. J. Smith. A Processor Architecture for Horizon. In *International Conference on Supercomputing*, pages 35–41. Orlando, IEEE Computer Society Press, November 1988.

[57] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[58] K. R. Traub and D. E. Culler. Global Analysis for Partitioning Non-Strict Programs into Sequential Threads. In *Conference on Lisp and Functional Programming*, pages 324–334. San Francisco, ACM Press, 1992.

[59] K. Ueda and M. Morita. A New Implementation Technique for Flat GHC. In *International Conference on Logic Programming*, pages 3–17. Jerusalem, MIT Press, June 1990.

[60] K. Yanoo. An Optimizing Compiler for a Parallel Inference Language. In H. Tanaka, editor, *Annual Report of the Research on Parallel Inference Engine*, pages 71–94. University of Tokyo, April 1992. (in Japanese).