ICOT Technical Report: TR-0828

TR-0828

An Insider's View of the FGCS Project

by
T. Chikayama

January, 1993

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

An Insider's View of the FGCS Project

Takashi Chikayama

Abstract

This article represents the author's personal view of the Japan's Fifth Generation Computer Systems project. The project started in 1982 and the author has been working in the project from its very beginning. The research history of the project is looked back and its outcome is reviewed from an insider's viewpoint.

1 Introduction

I joined the Institute for New Generation Computer Technology (ICOT), the research institute established for carrying out the FGCS project, in June 1982. It was almost immediately after its creation in April of the same year. All the researchers then gathered at ICOT were recruited from computer manufacturers and governmental research institutes (I was among these and was recruited from Fujitsu). Almost all these researchers, with very few exceptions, of which I was one, left ICOT after three to five years, either returning to their original places of employment or starting a new career, mostly in universities. Having been at ICOT for the entire project period of 10 years plus one additional year, I have gradually been making more critical decisions about research directions.

I started my research career at ICOT in design of a sequential logic programming language *à la* Prolog and its implementation, and then designing an operating system for logic programming workstations. As the main interest of the project shifted to concurrent logic programming, my research topics naturally changed to the design and implementation of concurrent languages and an operating system for parallel systems. The project has included various other research topics with which I have not been involved, for which I cannot comment fully here.

2 Before Things Really Started

2.1 The First Encounter

When the preliminary investigation of the FGCS project plan began in 1979, I was a graduate student at the University of Tokyo. Tohru Motooka, a professor at the university, was playing an important role in forming the project plan. I was invited to participate in one of many small groups to discuss various aspects of the project plan. That was my first chance to hear about this seemingly absurd idea of the "fifth generation" computer systems.

The project plan at that time was just too vague to interest me. The idea of building novel technologies for future computer systems seemed adequate, but it was not at all clear what such technologies should be. Our group was supposed to discuss how an appropriate software development environment for such a system should be designed, but the discussion was not much more than writing a sci-fi story. Both the needs and seeds of such a system were beyond our speculation, if not our imagination.

A few years later, the project plan became more concrete, committed to parallel processing and logic programming. My main research topic at the university was design and implementation of a Lisp dialect. Hideyuki Nakashima, one of my colleagues there, was an enthusiastic advocate of logic programming, and was strongly influenced by Koichi Furukawa, who was one of the key designers of the FGCS project plan. Nakashima was implementing his Prolog dialect on a Lisp system I had just implemented, and I assisted in this process. Although the Prolog language seemed interesting, I could not imagine how such a language could be implemented with the efficiency reasonable for practical use.

It actually took longer than expected, as is always the case; the first system was ready (with a barely useful software development environment for application users) at the end of 1984, two-and-a-half years after the project began. The development environment gradually matured to a reasonable level as its operating system went through several revisions. Two major revisions were subsequently made to the hardware, and the execution speed was improved by more than 30 times. The system, which had been used as the principal tool for software research until the first experimental parallel inference system was ready in 1988, is still being used heavily as personal workstations that simulate the parallel system.

3.1 Sequential Inference Machine: PSI

Without any doubt, the decision to develop such a "machine" had the same motivation as the Lisp machines developed at MIT and Xerox PARC. A DEC-2060 system was introduced in the fall of 1982, allowing us to use Edinburgh Prolog [2]. Its compiler was by far the most efficient system available at the time. However, when we started to solve larger problems, we soon found that the amount of computation needed exceeded the capacity of time-shared execution. Personal workstations specially devised for a specific language and with plenty of memory seemed to be the solution. Indeed they *were*, I think, for logic programming languages in 1982 when more sophisticated compilation techniques were not available.

Two models of personal sequential inference machines, called "PSI", were developed in parallel [23]. They had the same microprogram-level specification designed at ICOT, but with slightly different architectures. Two computer companies, Mitsubishi and Oki, manufactured different models. Such competition occurred several times during the project on different R&D topics. The word "competition" might not be quite accurate here. Although the top management of the companies might have considered them as competition, the researchers who actually participated in the projects gradually recognized that they were meant to be in collaboration. They exchanged various ideas freely in frequent meetings at ICOT.

Both of the models of sequential inference machines had microcoded interpreter for graph-encoded programs. An alternative research direction which put more effort on static analysis and optimized compilation was not considered seriously. Running such a research project in parallel with the development of the hardware systems might have yielded less costly solutions. However, given a short period of time and few human resources with compiler-writing skills, we had to commit ourselves to pursue a single method.

3.2 The First Kernel Language: KL0

My first serious job in the project was designing the first version of the kernel language, "KL0". This language was, in short, an extended Prolog. Some nonstandard control primitives were introduced, such as mechanisms for delaying execution until variable binding or for cleaning up side effects on backtracking, but they were only minor additions that did not affect the basic implementation scheme of Prolog.

We decided to write the whole software, including the operating system, in this language. This was partly for clearing up the common misunderstanding that logic

When I finished my doctoral thesis in March 1982 and was looking for a job opportunity, Motooka kindly recommended me to work at ICOT. Without any particular expectation in research topics, hoping to do something interesting without too much restriction, I accepted his proposal.

2.2 Joining the Project

The FGCS project was organized so that one central research institute, ICOT, could decide almost all the aspects of the project, except for the size of its budget. The Japanese government (Ministry of International Trade and Industry [MITI] to be more specific), funded the project, but MITI officers never forced ICOT to change the research direction in order to promote the Japanese industry more directly. This has been true throughout the 11 years of the project period. Although the grand plan of the project was already there, it was still vague enough to leave enough freedom to the ICOT researchers.

One of the consequences of this situation was that when the research center was founded with some 30 researchers in June 1982 nobody had concrete research plans. The core members of the project, including Kazuhiro Fuchi, Toshio Yokoi, Koichi Furukawa and Shunichi Uchida who had participated in the project's grand plan, held meetings almost daily to develop a more detailed plan. Common researchers such as I had no mission for about a month but to read through a heap of research papers on vast related areas. Voluntary groups were formed to study those papers. Also, we tried out Prolog programming with implementations on PDP-11 and Apple-II, that were the only available systems to us at that time.

My greatest surprise in course of this study was that the researchers gathered there had almost no experience of symbolic processing, with only a small number of exceptions. Only few had experienced design and implementation of any language systems or operating systems either. It was not that ICOT's selection of researchers was inappropriate — there were so few in Japan with experiences in these areas. The level of the computer software research in Japan was far behind the United States and Western Europe at that time, especially in the basic software area.

In early July, a more concrete research plan was finished and several research groups were formed to tackle specific topics. I joined the group to design the first version of the "kernel language".

The idea of the "kernel language" has been characteristic of the project. The research and development were to be started with the design of a programming language, followed by both hardware and software research toward its efficient implementation and effective utilization. There have been two versions of the kernel language, KL0 and KL1, and this process repeated in the project. The design I started in 1982 was that of KL0, which was a sequential logic programming language.

3 Sequential Inference Systems

One of the first subprojects planned was to build so-called "personal sequential inference machines". The development effort was an attenpt to provide a comfortable software research environment in logic programming languages as soon as possible.

programming languages could not be practical for real-world software development. We thought, on the contrary, that using a symbolic processing language was the easiest way to build a decent software environment for the new machine.

For writing a whole operating system, extensions to control low-level hardware, such as the I/O bus or page-map hardware, were also made. The memory space was divided into areas private to each process for Prolog-like stacks and areas common to all the processes where side effects were allowed. Side-effect mechanisms were much enhanced than Prolog. Interprocess communication was effected through such side effects.

The resultant language had high descriptive power, but was somewhat like a medley of features of various languages. I did not mind it because, although the language had all the features of Prolog, it was supposed to be a low-level machine language, rather than a language for application software developers.

3.3 An Object-Oriented Language: ESP

A group headed by Toshio Yokoi was designing the operating system for the sequential inference machines. Their rough design of the system was based on object-oriented abstraction of system features. After finishing the design of KL0, I was requested to join a team to design a higher level language with object-oriented features.

Through several meetings discussing the language features, a draft specification of the language named "ESP" was compiled [3]. I wrote an emulator of its subset on Edinburgh Prolog in the summer of 1983, in about one week when the DEC-2060 was lightly loaded as most of other users were away on summer vacation. This emulator was used extensively later in early development phases of the operating system.

The language model was simple. Each object corresponds to a separate axiom database of Prolog. The same query might be answered differently with different axiom sets, as different objects behaves differently on the same method in other object-oriented languages. This allowed programming in small to be done in the logic programming paradigm and programming in large in the object-oriented paradigm.

3.4 SIMPOS

More detailed design of the operating system followed. Actual coding and debugging of the system began in the fall of 1983 using the implementation on Edinburgh Prolog, by a team of some 30 programmers gathered from several software companies. The hardware of PSI arrived at ICOT on Christmas day, and the development of the microcoded interpreter, which had also been done on emulators, was continued on the physical hardware. In July 1984, the operating system, named SIMPOS, first began working on the machine. In the course of the development, I gradually had become the virtual leader of the development team.

Even in its first version, SIMPOS had a full repertoire of a personal operating system: multitasking, files, windows (which were not so common at that time), networking, and so on. The first version, however, was awfully slow. It took several seconds to display a character on a display window after pressing a keyboard key.

Following our analysis of this slug, a thorough revision of the system was carried out. The microcoded language implementation and the compiler, especially the object-oriented features, were considerably improved, making the same program run about three times faster. The operating system also went through a complete revision in the kernel, the I/O device handlers, the window system, and so forth. Algorithms and data structures were changed everywhere. Task allotment to processes were also changed. This considerable amount of change made the system run almost two orders of magnitude faster. The revision took less than three months and was ready to exhibit at the conference FGCS'84 in the beginning of November [4].

The system before the revision already had several tens of thousands of lines of ESP. The high level features of ESP helped considerably in carrying out such a major revision in such a short period of time. The object-oriented features, especially its flexible modularization power, allowed major changes without taking too much care on details. Like other symbolic processing languages, explicit manipulation of memory addresses is not allowed in ESP (except for in the very kernel of the system) and ranges of indexes to arrays are always checked. This made bugs in rewritten programs much easier to find.

A very important byproduct of the development of SIMPOS was training of logic programming language programmers. For most of the programmers participating in the development of SIMPOS it was their first experience to write a logic or an object-oriented programming language. Many, probably nearly half of them, had not experienced any large-scale software development before. For some, ESP was the first language to program in. Those programmers who acquired programming skills during this development effort played important roles in various software development later in the project.

3.5 Software Systems on PSI

The original PSI machine ran at about the same speed as Edinburgh Prolog on DEC 2060. The large main memory (80 MB max.) allowed much larger programs to run. Being a personal machine, users were not bothered by other time-sharing users. Limitation in computational resource, one of the largest obstacles in advanced software research, was greatly relaxed.

From 1985 and on, the PSI machine, and its successors PSI-II and -III, have been used heavily in software research. The largest software on PSI was its operating system SIMPOS. It went through many revisions and added more and more functionalities, including debugging and performance-tuning facilities, on ever-increasing users' demands. It now has more than 300,000 lines of ESP code.

Not only the operating system but also other basic software systems were built up on PSI and SIMPOS. A database system Kappa based on a nested relational model was probably the largest such system. Higher level programming language systems, such as a language based on situation semantics CIL or a constraint based language CAL, were also built.

Numerous experimental application systems were also built on PSI. A natural language processing system, DUALS played an important role in showing to the people outside the community what a logic programming system can do. Many

expert systems and expert system shells were developed, based on a variety of reasoning technologies. At its maximum, probably more than two hundred people were conducting their research using PSI or its successors within the project [14].

3.6 PSI-II and -III

Near the end of 1985, we decided to develop a new model of PSI based on a more sophisticated compilation scheme proposed by David H. D. Warren [22]. Its experimental implementation on PSI by Hiroshi Nakashima ran more than twice as fast as the original implementation. A new machine called PSI-II was designed and became operational near the end of 1986. SIMPOS were ported to the machine relatively easily. This model went through minor revisions for faster clock speed later and its final version attained more than 400 KLIPS, about 10 times faster than the original PSI. As the machine clock was as low as 6.67MHz, this meant that one inference step needed 16 microprogram steps.

Another major revision was made during 1989 and 1990, which resulted in the third generation of the system, PSI-III. At this time, Unix was already recognized as the common basis of virtually all research systems. Thus, the PSI-III system was built as a back-end processor, rather than a stand-alone system. The operating system, however, was ported to the new hardware almost without modification, replacing I/O device drivers with a communication server to the front-end part. The system attained 1.4 MLIPS at the clock rate of 15MHz. One inference needed only 11 steps.

4 Parallel Inference Systems

From the very beginning of the project, the second version of the kernel language was planned to combine parallel computation and logic programming. Parallel hardware research was going on simultaneously. These two groups, however, did not interact well in the early years of the project, resulting in several parallel Prolog machines and a language design that did not fit on them. Later, the language and hardware design activities became much better orchestrated under the baton of KL1.

4.1 Early Parallel Hardware Systems

Some argued that much parallelism could be easily exploited from logic programming languages because both AND and OR branches can be executed in parallel. With some experience in cumbersome interprocess synchronization, I was quite skeptical about such an optimistic and simplistic claim. Yes, a high degree of parallelism was possibly there, but exploiting that parallelism could be counterproductive; making everything parallel means making everything slow, probably spoiling all the benefits of parallelism.

The parallel hardware research began, however, despite the skepticism. As far as pure Prolog is concerned, the easiest parallelism to exploit was the OR parallelism because no sharing of data is required between branches once the whole environment is copied. Some of the systems successfully attained reasonable speedup, although the physical parallelism was still small.

The next thing to do was to implement a fuller version of Prolog, since the descriptive power of pure Prolog was quite limited. The implementation was a difficult task. To do that efficiently actually required a considerable amount of effort later in the Aurora OR-parallel Prolog project [17]. Our language processing technology was not yet at that level. OR parallel hardware research ceased around 1985 and was displaced by committed-choice type AND parallel research.

4.2 Pre-GHC Days

The first concurrent logic programming language I learned was the Relational Language by Keith Clark and Steve Gregory [9]. When I read the paper in 1982, I liked it because the language did not try to blindly exploit all the available parallelism, but confined itself to the dataflow parallelism. The idea there seemed quite revolutionary. I thought the language implementation would be much easier than naive parallelization of Prolog and parallel algorithms could be easily expressed in the language. But should a language for describing algorithms be called a logic programming language?

The most loudly trumpeted advantage of logic programming was that the programmers have only to describe *what* problem to solve, not *how*.. At that time, in the summer of 1982, I was still a beginner in Prolog programming. I did not yet recognize fully that, even in Prolog, I had to describe algorithms. Anyway, I was too busy designing the sequential system and soon stopped thinking about it.

Near the end of 1982, Ehud Shapiro visited ICOT with his idea of Concurrent Prolog (CP). During his stay, he refined the idea and even made its subset implementation on Edinburgh Prolog[18], which worked very slowly but anyway allowed us to try out the language. The language design considerably generalized the idea in the Relational Language by allowing partially defined data structures to be passed between processes. The object-oriented programming style in CP proposed later by Shapiro and Akikazu Takeuchi [19] showed that the enhanced descriptive power would be actually useful in practical programming.

The language feature attracted people at ICOT most might be its syntactic similarity to Prolog, which the Relational Language did not have. This look-alikeness was inherited later in PARLOG and then in GHC. This may have been the main cause of the widespread misunderstanding that concurrent logic programming languages are parallel versions of Prolog.

In 1983 Clark and Gregory proposed their new language, PARLOG [10]. Its design seemed to have been greatly influenced by CP. A crucial difference was that, the argument mode declaration allowed more static program analysis, making it much easier to implement nested guard structures.

When ICOT invited Shapiro, Clark, Gregory, and Ken Kahn, who was also interested in the area, all at the same time, we had time to discuss various aspects of those languages. These discussions contributed considerably in deciding later research directions. I was an outsider at that time, but enjoyed the discussions. Basic ideas of some of the features incorporated in KL1 implementations occurred to me during the discussions, such as automatic deadlock detection by the garbage collector [16].

I already had become sure enough through my experience of Prolog programming

that we cannot avoid describing algorithms even in logic programming languages. When the basic design and the development time table of SIMPOS were more or less established, I could find some time for my participation in the design of CP implementation.

After FGCS'84 held in November, Kazunori Ueda, then at NEC, started examining the features of CP, especially its synchronization mechanism by *read only* variables and atomic unification in detail. His conclusion was that, to make the semantics clean enough, the language implementation would become much more complicated than was expected. That led him, at the very end of the year, to a new language with much simpler and cleaner semantics, later named the Guarded Horn Clauses (GHC) [20].

4.3 Guarded Horn Clauses

When Ueda proposed GHC, the group designing KL1 almost immediately adopted it as the basis of KL1, in place of CP. Although I cannot deny the possibility of the "not invented here" rule slightly affecting the decision in a national project, the surprisingly simpler and cleaner semantics of GHC was the primary reason.

GHC was much more welcomed than CP by language implementors. Those who had not found any reasonable implementation schemes of CP felt much more relaxed. Only a few months later, Shunichi Uchida and Kazuo Taki initiated a group to plan an experimental parallel system, connecting several PSI machines, to make an experimental implementation of GHC, which was called Multi-PSI [13].

I was still feeling uneasy with its ability to express nested guards. Arbitrarily nested environments were required to implement them correctly, in which variables of the innermost environment and outer environments must somehow be distinguished.

In the fall of 1985, partly under the influence of the Flat version of CP adopted as the basis of the Logix system developed at Weizmann Institute [12], the KL1 group decided not to include nested guards in the language, that made it Flat GHC. This decision allowed me to start considering further details of the implementation with Toshihiko Miyazaki and others, although my principal mission was still the sequential system.

The last few months of the year might have been the hardest period for those who had been engaged in the parallel Prolog hardware development. After examining the rough sketch of Flat GHC implementation, the leaders of the group, Shunichi Uchida and Atsuhiro Goto, decided that this language was simple enough for efficient implementation and descriptive enough for a wide range of applications. The development of parallel Prolog machines was stopped and new project of building parallel hardware that support a language based on Flat GHC was started.

4.4 MRB and My Full Participation

Based on experimentations with the first version of Multi-PSI, building a more powerful and stable parallel hardware was planned, called Multi-PSI V2. For the processing elements, the second generation of PSI, PSI-II, was chosen. From this stage (i.e., from 1986) I was more fully involved in the parallel systems research as

SIMPOS was approaching to its maintenance phase. My active motivation was that I thought I solved the last remaining difficulty of efficient implementation.

Logic programming languages are *pure* languages in that once a value holder (a variable or an element of a data structure) gets some value, it will remain constant. It is impossible to *update* an element of an array. What one can do is to make a copy of an array one element differing from the original. A straightforward implementation of actually copying the whole array was, of course, not acceptable. Representing arrays with trees, allowing logarithmic time accesses, would not be satisfactory either. Without constant time array element accesses, computational complexity of already existing algorithms would become larger — massively parallel programs written in such a language would be beaten by sequential programs with large enough problem size. Henry Baker's shallow binding method [1] or similar methods proposed for Prolog matched the basic requirements, but the constant overhead associated with those methods seemed unbearable for the most basic primitives.

In early 1986, I heard that a constant time update scheme was designed by a group in a company cooperating with ICOT. I talked with them and found a crucial oversight in their algorithm, but the basic idea was excellent. If there was no other references to an array except that used as the original of the updated array, destructive update would not disturb the semantics. While the idea was simple, the algorithm of keeping the single reference information where I found the bug was rather complicated, as we had to cope with shared logical variables.

After several days of considering how to fix the bug, I reached a solution, later named the multiple reference bit (MRB) scheme [6]. Only one-bit information in pointers, rather than data objects, was needed for MRB, which was especially beneficial for shared memory implementation, since no memory accesses were needed for reference counting. It was also suited for hardware support.

In later years, as static analysis of logic programs prospered, static reference count analyses were also studied, yielding reasonable results. But this dynamic analysis by MRB suited well to the human resource we had at ICOT. The lack of compiler experts has always been a problem with the project. If we had tried static analysis method at that time, the language implementation would not have been completed in that short period of time.

4.5 KL1 and Multi-PSI V2

Near the end of 1986, a group was formed to investigate details of the language implementation on Multi-PSI. Weekly meetings of this group continued for about two years and the discussion there was the hottest I know of at ICOT. The final design of KL1 [21] was decided here and most of the proposed ideas were actually implemented on Multi-PSI [14].

Most of the implementation issues were on optimization schemes, many based on MRB. The principle there was to make single reference cases as efficient as possible and have multiple reference cases handled correctly but less efficiently. This decision later was proved to be reasonable through later programming experiences since the single reference programming style was found to be not only efficient but also more readable. Some similar languages designed more recently even enforce data structure references to be single.

The discussion in the group was not confined to implementation issues. Some aspects of the specification of KL1, especially on metalevel features, were also investigated. Although the KL1 language design group had already proposed that KL1 should have the metacall feature similar to one in PARLOG [8], it only had qualitative execution control mechanisms, while more quantitative mechanisms such as priority and resource control were needed as the parallel kernel language. It was reasonable, I think, to define details of such metalevel features at the implementation group, as they could not be clearly separated from implementation issues.

Load distribution was made explicit by enabling the program to specify the processor to execute goals. This decision seems to have been appropriate, as we are still struggling to design good load distribution policies and it would have resulted in disaster if the language implementation tried to automatically distribute the load within large-scale multiprocessor systems.

Data placement, on the other hand, was made automatic. Thanks to the side effect-free nature of the language, data structures can be moved to any processors and arbitrarily many copies can be made. This simplified the design considerably.

Features for building up a reasonable software development environment, such as primitives for program tracing and executable code management, were also added. These additions were designed so that the basic principles of the language, such as the side effect-free semantics, were not disturbed. Otherwise, the implementation would have been much more complicated, disabling various optimization schemes.

As a whole, the design of the language and its implementation was rather conservative. We chose a design we could be sure to implement without much problems and gave up our ambition to be more innovative. We had to provide a reasonable development environment to allow enough time for parallel software research within the project period.

The hardware development went on in parallel with the language implementation meetings at Mitsubishi and the hardware arrived at ICOT at the end of 1987. It had 64 processors with 80MB of main memory each, connected by a mesh network. The development of KL1 implementation on the hardware continued.

4.6 PIMOS

When the design of the basic meta-level primitives was completed, a team to develop the operating system for parallel inference systems, PIMOS, was formed in 1987. Given the well-considered language features, the design of PIMOS was not very difficult.

As the real parallel hardware was expected to be ready much later, we needed some platform of the operating system development. Although an implementation of GHC upon Prolog by Ueda was available, its performance was too low for large-scale program development and many newly added features of KL1 for system programming were lacking. A team led by Miyazaki made a fuller pseudoparallel implementation in C, called PIMOS Development Support System (PDSS) to fill up the needs.

Coding and debugging of PIMOS were done by a team of about 10 participants using PDSS, until the language system on Multi-PSI V2 became barely operational at the end of the summer of 1988.

As we expected, but nevertheless to our surprise, the operating system developed on the pseudoparallel PDSS could be ported immediately on to the real parallel hardware. With multiple processors, the execution order of processes on Multi-PSI was completely different from PDSS. In theory, the dataflow synchronization feature of GHC was expected to avoid any synchronization problems. But with my own experience in developing a multitask operating system SIMPOS, I was ready to encounter annoying synchronization bugs. On physically parallel hardware on which scheduling-dependent bugs were hard to reproduce, debugging should be much more difficult than on single-processor multitask systems. All the bugs we found were those of the language implementation except for a few problems of very high-level design.

This experience clearly showed us the merit of using a concurrent logic programming language. In a parallel processing environment, not only a limited number of system programmers, but also application programmers have to solve synchronization problems. The dataflow synchronization mechanism can reduce the burden almost entirely. The language implementation might be much more difficult than for sequential languages with additional communication and synchronization features, but the results of the effort can be shared by all the software developers using the language.

After about two months, the language implementation and the operating system on Multi-PSI V2 became stable. We could exhibit the system at FGCS'88 held in the beginning of December with several preliminary experimental application software systems [7].

4.7 Application Software Development

With its 64 processors, Multi-PSI V2 ran more than 10 million goal reductions per second at its peak. This figure was not outstanding, being only about ten times faster than Prolog on main frame machines, but good enough to invite some of the application researchers to parallel processing. Several copies of Multi-PSI V2 were manufactured in the following years and used heavily in parallel software research in various areas [15].

For about a year or two, we heard many complaints from those who were accustomed to Prolog and ESP. The lack of automatic backtracking made the language completely different from Prolog, while their syntactic similarity prevented some from easily recognizing the difference. Many tried to write their programs in Prolog-like style, recognizing after the debugging struggle that the language did not provide automatic search features and they had to write their own. Then they reluctantly started writing search algorithms. This often resulted in much more sophisticated searches than Prolog's blind exhaustive search. They also had to consider how these searches could be parallelized in an efficient way. The language lured Prolog programmers to the strange world of parallel processing with its syntactic decoy.

Another typical difficulty seems to have been in finding a good programming style in the language with too much freedom. The object-oriented style [19] became recognized as the standard style later. Designing programs in KL1 became synonymous with designing process structures.

Load distribution with decent communication locality is the key to efficient parallel computation. Load distribution strategies that made success for some particular problems were compiled into libraries and distributed with the operating system [11], accelerating the development of many application systems.

Although some application systems attained almost linear speedup with 64 processors rather easily, others needed more efforts to benefit from parallel execution. Some needed a fundamental revision in the algorithm level; some could run only a few times faster than on a single processor; some seemed to have attained reasonable speedup, but when certain changes in the algorithm successfully improved the overall performance, the speedup figure went down considerably. Parallel algorithm study with much more realistic assumptions on the hardware than PRAM is one of the most important areas of study in the future.

4.8 Parallel Inference Machines

In parallel with the development of the experimental Multi-PSI V2, design of more powerful parallel inference machines, PIMs, was going on by a team headed by Goto and later by Keiji Hirata. I participated in this hardware project only to a limited extent, but I may have influenced on the grand design of the language implementations on PIMs considerably.

Five different models were planned, corresponding to five different computer manufacturers. This decision had a more political than pure scientific basis. Five different models not only required more funding but also incurred considerable research management overhead. On the other hand, it may have been quite effective in diffusing the technology to Japanese industry.

As it was still difficult to find many language implementation experts, we decided to use basically the same implementation, Virtual PIM (VPIM), for four out of five models to minimize the effort. The one remaining model inherited the design from the implementation on Multi-PSI V2. VPIM was written in a language called the PIM System Language (PSL), which is a small subset of C with extensions to control low-level hardware features. The idea was that the same implementation should be ported to all the models by only writing PSL compilers. VPIM was developed at ICOT using a PSL compiler built on Sequent Symmetry. The responsibility of porting it to each model was on each manufacturer.

The first model of PIM to have become available in mid-1991 was PIM/m, the successor of Multi-PSI V2 by Mitsubishi. It had up to 256 processors with its peak speed of more than 100 million reductions per second (i.e., about ten times faster than Multi-PSI V2). PIMOS and many application software developed on Multi-PSI were ported to PIM/m without much effort, as the programming language was identical. Almost all of the software that showed near-linear speedup on Multi-PSI V2 also did so on PIM/m.

Early in 1992, another model, PIM/p, manufactured by Fujitsu, got ready for software installation. This system had up to 64 *clusters* each with 8 processors sharing the main memory through coherent cache memory. Load distribution within clusters was automatic by the language implementation, while it was still programmed among clusters. This made the scheduling quite different from Multi-PSI or PIM/m, but PIMOS and application software were ported without much of a

problem except for hardware and language implementation problems, to be solved in time for exhibition at FGCS'92 [5]. Again we thanked the language for its dataflow synchronization.

5 Conclusion

There have been pros and cons on the outcome of the project.

One might argue that the project was a failure as it could not meet its goal described in the project's grand plan, such as a computer that can speak like a human. The grand plan didn't actually say such a computer can be realized within ten years. The goal of the project was to establish the basic technologies needed for making such a dream come true. I consider such dreams a much better excuse than Star Wars to obtain funding for basic research.

The FGCS project was the first scientific Japanese national projects conducted by MITI. All the projects carried out before FGCS started, and many that followed, were aiming primarily at promoting industry. This may have been greatly due to the intransigent character of the project leader, Kazuhiro Fuchi. The results of the project may not be immediately commercialized. But we have not been aiming at such a short-term goal. Our goals are much longer-term: technologies that will be indispensable when even personal computers can have multimillion processors.

One of the weak points of our project was, as mentioned earlier, the shortage of human resource in basic software technology. If we had three times as many researchers who could design a programming language and who could write optimizing compilers, the design of the parallel inference machines might have been much different. At least, more ambitious systems (several of them) could have been designed. The language system was not our ultimate goal. Research in parallel software architecture was a much more important goal. For securely providing a development environment for parallel software research with the limited human resources, we chose one safe route, which, I believe, was the best choice.

As a whole, I think the project was quite successful. It made considerable contribution to the parallel processing technologies, especially in programming language and software environment design. Research in parallel software for knowledge processing has only begun, but without the project, there would have been nothing.

Further refinements of the design of the kernel language, its implementation both in compilation scheme and hardware are needed, but they are relatively minor issues. The most important research topic of the future, I believe, is in the design of parallel algorithms. The largest achievement of the project was showing a way to build a platform for such research activities.

Acknowledgment

The author would like to thank all those who participated in the project and research in the related areas.

References

[1] Henry Baker. Shallow binding in LISP 1.5. *Communications of the ACM*, 21(7), 1978.

[2] David L. Bowen, Lawrence Byrd, Fernando C. N. Pereira, Luis Moniz Pereira, and David H. D. Warren. *DECsystem-10 Prolog User's Manual*, November 1983.

[3] Takashi Chikayama. Unique features of ESP. In *Proceedings of FGCS'84*, pages 292–298, 1984.

[4] Takashi Chikayama. Programming in ESP — experiences with SIMPOS. In Kazuhiro Fuchi and Maurice Nivat, editors, *Programming of Future Generation Computers*, pages 75–86. North-Holland, New York, New York, 1988.

[5] Takashi Chikayama. Operating system PIMOS and kernel language KL1. In *Proceedings of FGCS'92*, pages 73–88, Tokyo, Japan, 1992.

[6] Takashi Chikayama and Yasunori Kimura. Multiple reference management in flat GHC. In *Proceedings of 4th International Conference on Logic Programming*, 1987.

[7] Takashi Chikayama, Hiroyuki Sato, and Toshihiko Miyazaki. Overview of the parallel inference machine operating system (PIMOS). In *Proceedings of FGCS'88*, pages 230–251, Tokyo, Japan, 1988.

[8] Keith Clark and Steve Gregory. Notes on systems programming in PARLOG. In *Proceedings of FGCS'84*, pages 299 306, 1984.

[9] Keith L. Clark and Steve Gregory. A relational language for parallel programming. In *Proceedings of ACM Conference on Functional Languages and Computer Architecture*, pages 171–178, 1981.

[10] Keith L. Clark and Steve Gregory. Parlog: A parallel logic programming language. Research Report TR-83-5, Imperial College, 1983.

[11] Masakazu Furuichi, Kazuo Taki, and Nobuyuki Ichiyoshi. A multi-level load balancing scheme for or-parallel exhaustive search programs on the multi-PSI. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 50–59, March 1990.

[12] Michael Hirsh, William Silverman, and Ehud Shapiro. Computation control and protection in the logic system. In Ehud Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, pages 28–45. The MIT Press, Cambridge, Massachusetts, 1987.

[13] Nobuyuki Ichiyoshi, Toshihiko Miyazaki, and Kazuo Taki. A distributed implementation of flat GHC on the multi-PSI. In *Proceedings of 4th International Conference on Logic Programming*, Cambridge, Massachusetts, 1987. MIT Press.

[14] ICOT. *Proceedings of FGCS'88.* ICOT, Tokyo, Japan, 1988.

[15] ICOT. *Proceedings of FGCS'92.* ICOT, Tokyo, Japan, 1992.

[16] Yu Inamura and Satoshi Onishi. A detection algorithm of perpetual suspension in KL1. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 18–30. The MIT Press, 1990.

[17] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora or-parallel system. *New Generation Computing*, 7:243–271, 1990.

[18] Ehud Shapiro. A subset of Concurrent Prolog and its interpreter. ICOT Technical Report TR-003, ICOT, 1983.

[19] Ehud Shapiro and Akikazu Takeuchi. Object oriented programming in Concurrent Prolog. ICOT Technical Report TR-004, ICOT, 1983. Also in New Generation Computing, Springer-Verlag Vol.1 No.1,1983.

[20] Kazunori Ueda. Guarded Horn Clauses: A parallel logic programming language with the concept of a guard. ICOT Technical Report TR-208, ICOT, 1986.

[21] Kazunori Ueda and Takashi Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, December 1990.

[22] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, 1983.

[23] Minoru Yokota, Akira Yamamoto, Kazuo Taki, Hiroshi Nishikawa, and Shunichi Uchida. The design and implementation of a personal sequential inference machine: PSI. ICOT Technical Report TR-045, ICOT, 1984. Also in New Generation Computing, Vol.1 No.2, 1984.