TR-577

Processor Element Architecture for Parallel Inference Machine: PIM/p

by
A. Goto, T. Shinogi, T. Chikayama, K. Kumon
& A. Hattori (Fujitsu)

August, 1990

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

Processor Element Architecture for Parallel Inference Machine : PIM/p

Atsuhiro Goto[*1] Tsuyoshi Shinogi[*2] Takashi Chikayama[*1]

Kouichi Kumon[*2] Akira Hattori[*2]

*1: Institute for New Generation Computer Technology (ICOT)

4-28, Mita-1, Minato-ku, Tokyo 108, Japan

Phone: (03) 456-3193

*2: Fujitsu Limited

1015, Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan

List of Figures

List of Tables

Abstract

This paper describes the design of processor element architecture for the parallel inference machine prototype, PIM/p. Several innovative features are incorporated in the processor architecture to suit concurrent logic programming languages such as KL1. The processor's design is based on tagged architecture. With the variety of tag handling operations, instructions can be executed by one cycle pitch pipeline. Macro-call instructions are introduced to enable a lightweight subroutine call function for polymorphic operations required in execution of high level languages such as logic programming languages. This enables system designers to easily define high level instructions without losing the benefits of the pipelining mechanism. Dedicated instructions are introduced to support incremental garbage collection. Local coherent cache and optimized memory operations tailored to the memory access characteristics of KL1 can reduce common bus traffic in shared memory multiprocessors. In this paper, we describe the design decisions for these architecture features. The LSIs are now being fabricated using CMOS standard cell technology.

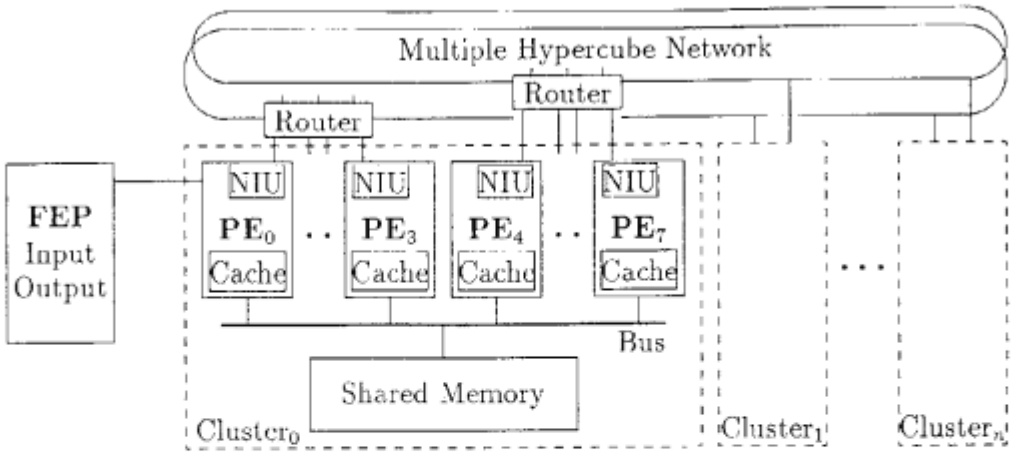

Figure 1: PIM/p overview

1 Introduction

In the Japanese FGCS project, the parallel inference machine systems are being developed based on a logic programming framework [9, 7]. Current interest in parallel logic programming stems from its declarative semantics which facilitates writing and debugging programs and removes most of the need for explicit uncovering and control of synchronization in concurrent programming.

KL1 [4], the kernel language of the parallel inference machine system, was designed based on GHC [23]. GHC has clear and simple semantics as a concurrent logic programming language, by which programmers can express important concepts in parallel programming, such as inter-process communication and synchronization.

We hope to realize very high execution performance for logic programming in KL1 to promote parallel logic programming application research. However, KL1 has features that make conventional machines unsuitable for efficient execution. Three of these features are: (1) unification is a *polymorphic* operation on, usually, dynamically constructed linked data structures; (2) the execution context, though small, is frequently switched because of data flow synchronization; and (3) single assignment feature demands high memory band width and efficient memory management scheme.

A parallel inference machine prototype, PIM/p, tailored to KL1, is now being developed, which is planned to include up to 512 processor elements. Hierarchical structure is introduced in the PIM/p. Eight processor elements form a cluster, communicating through shared memory over a common bus. The clusters are connected with one another by a multiple

hypercube packet switching network. This article presents the processor element design for PIM/p.

Some of the innovative features introduced in the PIM/p processor element architecture are: (1) light weight subroutine call function by macro-call instruction, which exploits the advantages both of hard-wired reduced instruction set computers and microprogrammable high-level instruction set computers; (2) architectural support for incremental garbage collection; and (3) local coherent cache and optimized memory operations tailored to KL1 parallel execution, which can reduce common bus traffic within shared memory multiprocessors, such as a PIM/p cluster.

This paper is organized as follows. The concurrent logic programming language, KL1, is briefly introduced in section 2, with its influences to the processor architecture. Section 3 and section 4 describe the design decisions for CPU and cache, and innovative architectural features tailored to KL1 programs. Section 5 presents the overview of processor element implementation. Finally conclusions are presented.

2 Characteristics of KL1

To understand the underlying motivations of the processor element design presented later in this paper, it is beneficial to review the concurrent logic programming language KL1, and its execution characteristics.

2.1 Brief Introduction to KL1 Abstract Machine

KL1 was initially specified as flat guarded horn clauses (FGHC) [23], and has been extended to be a practical language introducing meta-call and priority scheduling functions. The general purpose concurrent programming capability of KL1 is shown through the development of a self-contained operating system, PIMOS [4] for Multi-PSI systems [20, 14].

The KL1 execution is modeled as a partially ordered set of reductions wherein the initial user query (a set of goals) is reduced to the empty set. In KL1, as in Prolog, procedures are composed of sets of clauses with the same name and arity, of the form: $H : -G_1, ...G_m|B_1, B_2, ..., B_n$. where H is the head of the clause, G_i are guards, "|" is the commit, and B_j are the body goals. Execution proceeds by attempting unification between a goal (the caller) and a clause head (the callee), followed by guard unification. If these

unifications succeed, the procedure call "commits" to that clause (other candidate clauses are dismissed) and the input goal is reduced to the body goals in that clause. To make the KL1 goal reduction efficient, an abstract machine, KL1-B [12, 7], is developed, which is on a similar level of abstraction as WAM [24] for Prolog.

The abstract KL1 architecture is summarized as follows [12, 17]. A goal is represented by a goal record, similar to a Prolog environment [21]. Reducible goal records are stored in a goal pool. A processor fetches a goal from the goal pool, and executes the compiled KL1-B code sequence corresponding to the goal attempting to commit to one of the clauses of its procedure. If a clause is committed to, the body instructions cause body goals to be created and put into the goal pool. If no clause is committed to, but one or more clauses are waiting for some variables being bound, the goal is *suspended*. When one of these variables is bound at some later time, the resumption routine is executed which restores the suspended goal(s) to the goal pool.

2.2 Conditional Execution Features in KL1

Dereference is required at the beginning of most unification instructions in KL1-B. In dereference, a register is first tested to see whether its content is an indirect pointer or not. If it is an indirect pointer, the cell pointed to is fetched into the register and its data type is tested again. Unification is performed depending on the data type.

Many instructions in KL1-B include run-time data type checks even after dereferencing. For example, the active unification between a KL1 variable, X (the contents are unknown), and a given structure, *Str*, has one of four kinds of actions, selected by the data type check: (1) when X is an unbound variable without suspended goals, the *Str* is assigned into the variable cell; (2) when X is an unbound variable with suspended goals, these suspended goals are resumed after the assignment to X; (3) when X matches the data type of *Str*, general unification for elements of both is performed; and (4) otherwise, the unification fails.

Consequently, most instructions in KL1-B include run-time data type checks. The actions that follow the run-time type check are very different. How to implement these polymorphic operations is one of the key issues in the processor design for concurrent logic programming languages. Therefore, tagged architecture is chosen as the base of the PIM/p processor element, and tag conditional macro-call instructions are introduced to perform polymorphic operations in KL1-B, which will be discussed in section 3.

2.3 Incremental Garbage Collection by MRB

KL1 is a concurrent language with no side effects. Destructive memory assignment is not allowed at the KL1 language level. Therefore, naive implementations of KL1 tend to consume memory area very rapidly, so that garbage collection must occur frequently. The locality of memory references is supposed to be very low during garbage collection because most garbage collection schemes [5] walk around wide memory area. As a result, cache misses and memory faults occur often. In sequential Prolog [24], this problem is not very serious because of the backtracking feature. However, as concurrent logic programming languages have no backtracking, an efficient garbage collection method with high memory reference locality is important in KL1 implementation.

Incremental garbage collection by multiple reference bit (MRB) [3] is introduced in KL1 architecture. MRB is a one bit tag in a pointer to show whether the referenced data object is possibly referenced from other data objects (*on-MRB*) or not (*off-MRB*). When a pointer to a data object has *off-MRB*, the corresponding memory area can be reclaimed after reading its contents, because there will be no other paths to the data. The reclaimed memory area is usually linked to free-lists for reallocation. As an optimization, the reclaimed memory area can be reused immediately with its contents.

Contents of a data object is read during unification. Therefore, KL1 compiler detects places where cells possibly become garbages, and inserts garbage collection instructions at appropriate places. Unification in KL1 may produce a chain of variable cells containing indirect pointers. These indirection cells with *off-MRB* can be reclaimed during dereferencing.

The locality of memory references can be raised using MRB incremental garbage collection, because memory areas that have recently been read are likely to be reclaimed and reused, instead of allocating new areas at completely irrelevant address. The MRB is also used to implement the constant time stream merging and array updating in KL1 programs. For example, an array element can be destructively updated without destroying the logic programming semantics, when the array is referenced by an *off-MRB* pointer. These features must be very important for KL1 to be a general purpose programming language.

MRB information maintenance and incremental memory management includes conditional execution with bit manipulation. This is a costly operation for conventional machines, because MRB information has to be maintained in each unification. Therefore, the MRB scheme in KL1 architecture requires low-level architectural support.

3 CPU Architecture Design

As discussed in section 2, KL1 has some features that are difficult to implement efficiently on conventional computers. These include polymorphic operations and incremental garbage collection. In this section, the key issues in CPU architecture design tailored to KL1 are discussed.

3.1 Alternatives for KL1-B implementation

Unifications include polymorphic operations for a variable cell whose type is not known until run-time. In addition, the incremental garbage collection by MRB is embedded in dereferencing. Therefore, tagged architecture is vital to efficient implementation of KL1.

Most Prolog machines, such as PSI [15], have been implemented as a high-level instruction set computers with microprogram, that is as WAM [24] interpretation by microprogram. However, the KL1-B interpretation by microprogram has the following disadvantages. First, it would be difficult to make full use of micro-instruction fields, because the actions of each KL1-B instruction are determined by run-time data type checks as in section 2.2. Next, the data type check often selects to proceed to the next instruction without any operations or with just a simple operation. Therefore, when each KL1-B instruction is interpreted by microprogram, the cost for dispatching to microprogram from a fetched instruction will be relatively large.

On the other hand, recently, tagged architecture has been incorporated into reduced instruction set computers (RISC) [11], taking advantage of compile-time optimization and low cost in hardware design. However, that architecture has the following disadvantages in KL1 implementation.

When KL1-B instructions are expanded by low-level RISC instructions, the static code size of compiled programs will be very large. In addition, these compiled programs possibly include many branch instructions. This is because most KL1-B instructions involve polymorphic operation. As a result, instruction cache misses occur often and common bus traffic may increase in tightly-coupled multiprocessors with local coherent cache [2], such as a PIM/p cluster (see section 4). Software simulation in [13] found the two times and four times expanded compiled code of original KL1-B code causes a 15% and 70% increase in the common bus traffic of a PIM/p cluster, so that the total performance of a cluster will degrade

by 5% and 30% [10]. Certainly, high-level instruction set computers with microprogram are advantageous to reducing common bus traffic.

The compiled programs in RISC-like instructions can, of course, be kept small by using small conditional subroutine calls. However, subroutine calls cost much in conventional methods. Therefore, to use only the best features of both RISC and high level instruction computers, we aimed at designing a processor which facilitates efficient conditional subroutine call function on data tag, accompanied by a RISC-like instruction set. As a result, the PIM/p processor element instruction set includes RISC-like instructions and the efficient one-level subroutine call function by tag conditions, presented in the following subsections.

3.2 Basic Instructions

The processor element of PIM/p has two kinds of instructions, external and internal. *External instructions* are mainly used to represent compiled codes of user programs, while *internal instructions* are used to define high level instructions as presented later in section 3.4. Most of these instructions are RISC-like instructions, in the sense that they can be executed by one cycle pipeline. However, there are almost 100 varieties of instructions, more than other RISC processors. This is because instructions for tag handling and dedicated instructions for KL1 are added as presented later.

Table 1 shows the form of basic instructions. Basically, ALU instructions have three register operands (one of the source operands may be a short immediate value). In memory access instructions, the memory location is specified by a register and an immediate offset. The sub-opcode *sub-op* can specify the transferred data width, which can be 8, 16, 32 bits, 32 bits with an 8-bit tag, or 64 bits. 64-bit data is loaded to (or stored from) two neighboring registers. As will be shown in section 5.3, branching costs three additional cycles. Thus one-cycle delayed branch instructions and conditional skip instructions are provided.

3.3 Tagged Architecture

Taking practical KL1 implementation into consideration, 40-bit (8-bit tag + 32-bit data) registers and tag branch instructions are provided in CPU. The MRB is assigned in one of the 8 bit tag.

As discussed in section 2.2, most unification includes a multi-way branch based on the

Table 1: Form of Basic Instructions

ALU instructions		
ALU-op	Rs1, Rs2, Rd	Rd ←Rs1 *op* Rs2;
ALU-op	Rs1, imm, Rd	Rd ←Rs1 *op* imm;
Memory access instructions		
read *sub-op*	Rd, Ra, ofst	Rd ←M[Ra+ofst];
write *sub-op*	Rs, Ra, ofst	Rs →M[Ra+ofst];
Branch instructions		
jump *cond*	Rt, mask8, (imm8,) ofst	if condition is true, PC←PC+ofst
(delayed jump)		
jump_and_link	Ra, Rd, retofst, ofst	Rd←PC+retofst, PC←Ra+ofst;
(delayed jump_and_link)		
skip *cond*	Rt, mask8, (imm8)	if condition is true, skip next

Table 2: Tag conditions in the PIM/p processor instructions

XOR, Not-XOR	tag(Rt) = imm8, or not
OR, Not-OR	tag(Rt) | mask8 = all 1, or not
AND, Not-AND	tag(Rt) & mask8 = all 0, or not
XORmask, Not-XORmask	(tag(Rt) & mask8) = imm8 or not

KL1 data type. Some Prolog machines, such as the PSI [21], have a hardware-supported multi-way branch function. The processor element of PIM/p does not have such hardware. This is because: (1) it is costly to adopt a hardware-supported multi-way branch to a pipeline processor; and (2) branches taken in run-time are biased; not all possibilities are chosen by equal chances. The PIM/p instruction set has only a two-way tag condition in macro-call instructions and in tag branch instructions, but various tag conditions can be specified in the instructions as follows.

The tag conditions can be specified as bit-wise logical operations between tag of a register Rt and 8 bit tag value imm8 in the instruction, as in Table 2. The (Not-)XOR checks whether the tag of Rt matches imm8. In addition to these exact match conditions, tag conditions to examine only specified bits in the tag of a register are provided. The mask8 values is

used to specify the bit field in the tag of Rt. The (Not-)OR conditions examines whether the specified bits are all one, while the (Not-)AND examines whether they are all zero. The (Not-)XORmask examines if the specified bits matches imm8. By these tag conditions, various group of data types, as well as the combination with MRB, can be specified in two-way branch instructions, such as jump, skip, and macro-call instructions.

In the processor element of PIM/p, various hardware flags, such as the condition code of ALU operation or an interrupt flag, can be accessed as the tag of dedicated registers. Therefore, these flags can also be examined just as KL1 data type.

3.4 Macro-call function

A macro-call instruction can be regarded as a *lightweight* subroutine call with tag conditions, whose form is:

MCall *cond*, R1, R2/imm8, R3/imm8, i-Addr

where i-Addr is the entry address of the internal instruction memory; R1, R2, and R3 are the register numbers. R2 and R3 can be 8 bit immediate values (imm8). The macro-call instruction first tests the data type of a register, given as its operand R1, then it will or will not invoke its macro-body in the internal instruction memory (IIM) depending on the result of the test. The contents of these registers as well as the immediate values can be accessed through *indirect access registers* and *indirect value registers* in the macro-body as presented later.

The macro-bodies stored in the internal instruction memory are written in *internal instructions* by system designers. Here, most of both external and internal instructions are common. Therefore, system designers can easily specify a high-level instruction, using one kind of RISC-like instructions instead of complicated micro-instructions in conventional computers. Considering the difficulty in making full use of long micro-instructions, this scheme is advantageous to system designers. In addition, the specification of a high-level instruction is very flexible, because a macro-body can include subroutine calls in external instructions stored in main (shared) memory as well as subroutine calls in the internal instruction memory.

One of the overheads in usual subroutine calls is the branching cost both for call and return. As presented later in section 5.3, the tag condition for macro-call is tested at the second stage of four stage pipeline. When the condition is true, the program counter for ex-

ternal instructions is frozen, and the execution stream is switched to the internal instructions by putting the entry address (`i-Addr`) in the internal program counter. Therefore, the cost to invoke the macro-body is only one additional cycle, while usual jump instructions cost three additional cycles to take the branch. The cost for returning from macro body is also minimized as follows. Each internal instruction has an additional bit, called *eoi*, to specify the exiting point from macro-body, so that the execution of macro-body can finish at any non-branch instruction. When the internal instruction with *eoi* is put into the pipeline, the external instruction follows without branching costs, melting the external program counter.

Another overhead is the cost for the arguments passing to and from the subroutine bodies. To avoid the arguments passing costs, two kinds of virtual registers are provided. *Indirect value registers* are used to get the operand of the macro-call instruction as an immediate value, and *indirect access registers* are used to access the contents of the register that is specified in the macro-call operand. Each of these virtual registers corresponds to the operand position of the macro-call instruction. Therefore, the arguments of a macro-call can be efficiently passed to and from its macro-body.

3.5 Support for Dereference and MRB Garbage Collection

As discussed in section 2.3, garbage collection support is one of the most important issues in parallel inference machines. The PIM/p instruction set includes several instructions tailored to MRB garbage collection.

In MRB incremental garbage collection, each variable cell or structure is allocated from a free list. When reclaimed, its memory area is linked to a free list. To support these free list operations, the `Push` and `Pop` instructions listed in Table 3 are provided. `Push` links a cell to the free list, and `Pop` allocates it from the free list, in one machine cycle. `PushTag` and `PopTag` put a new tag in the register. For example, allocation of a list cell referenced by "LIST" tag can be done by one instruction:

```
PopTag Rd, Ra, ofst, LIST
```

The MRB of each pointer and data object has to be maintained correctly in all unification instructions. Here, the most primitive operation is MRB maintenance during dereferencing. In dereferencing, the MRB of the dereferenced result should be *off* if and only if MRBs of both the pointer and the cell are *off*. In this case, the indirect word cell can be reclaimed immediately because the indirect word cell has no other reference paths to it. Two dedicated

Table 3: Instructions for dereferencing and MRB garbage collection

Instruction	*Operands*	*Comment*
`Push`	`Rs, Ra, ofst`	M[Ra+ofst] ←Rs, Rs ←Ra;
`PushTag`	`Rs, Ra, ofst, imm8`	M[Ra+ofst] ←Rs,
		data(Rs) ←(Ra), tag(Ra) ←imm8 ;
`Pop`	`Rd, Ra, ofst`	Rd ←Ra, Ra ←M[Ra+ofst];
`PopTag`	`Rd, Ra, ofst, imm8`	Rd ←Ra, tag(Rd) ←imm8,
		Ra ←M[Ra+ofst];
`ReadOrMRB`	`Rd, Ra, ofst`	Rd←M[Ra+ofst],
		mrb(Rd) ←mrb(Ra) | mrb(old Rd);
`Deref`	`Rd, Ra, ofst`	Rd ←Ra, Ra ←M[Ra+ofst],
		mrb(Ra) ←mrb(Ra) | mrb(old Ra);

instructions, `ReadOrMRB` and `Deref`, support this operation. `ReadOrMRB` accumulates both the address register's MRB and the destination register's MRB, then sets the result in the destination register. `Deref` performs MRB accumulation along with the `POP` operation. This means that:

```
Deref Reg, Ptr
```

saves the pointer `Reg` to the dereferenced cell to another register, `Ptr`, then reads the contents into `Reg` with MRB accumulation. Therefore, succeeding instructions can reclaim the cell referenced by `Ptr` by examining the MRB of `Reg`.

These instructions can minimize the costs of free-list operations and dereferencing with MRB management in PIM/p.

4 Cache Architecture Design

PIM/p has a hierarchical structure, as shown in Figure 1. Eight processor elements (PEs) form a cluster, communicating through shared memory (SM) over a common bus. Processor elements within each cluster share one address space, so that they can quickly communicate by reading or writing shared memory. However, KL1 programs require high memory bandwidth because data structure manipulations dominate whole computation rather than

arithmetic computation. Thus, we designed local coherent cache optimized for the memory access characteristics of KL1.

4.1 Motivations for Cache Design

As discussed in section 2.1, a processor executes goal reductions of a relatively small granularity (compared to procedural languages). Thus, focusing on the parallel processing within a PIM/p cluster, there are significant differences in KL1 memory referencing characteristics from the characteristics of conventional multiprocessor systems such as Symmetry [18].

First, memory write frequency is higher than in conventional languages. Memory access characteristics of KL1 benchmarks, gathered by simulation, indicate that data write frequency is 36% (see [8] for details).

Next, the processors communicate more often with each other, through the logical variables, than the usual parallel processing on the Symmetry system, because parallel goals share logical variables. In addition, it is necessary to communicate for scheduling KL1 goals. Thus it is important for a locally parallel cache to have an efficient cache-to-cache data transfer mechanism as well as to work as a shared global memory cache.

Finally, there are many exclusive accesses to communicate through shared logical variables. The frequency of locking shared data in the KL1 execution is relatively high. However, we can expect that exclusive memory accesses seldom conflict with each other [17].

4.2 Cache Protocol

Copyback cache protocols have been proved effective for reducing common bus traffic in shared-memory multiprocessors for procedural languages, as shown by Goodman [6] and Archibald [1], among others. Thus the basis for the PIM/p cache is a copyback protocol. Local coherent cache protocols, such as [1, 2, 16, 19], use both invalidation and broadcast to ensure all caches are consistent. Invalidation reduces common bus traffic when the frequency of shared block write accesses is low, while broadcast is better when many processors frequently write data to the same shared blocks [1]. Considering the single-assignment feature of KL1, most logical variables are shared by only two KL1 goals. Thus broadcasting is not necessary for most programs, and invalidation suffices.

Table 4: Optimized memory access instructions

Instruction	*Operation*
`read_invalidate`	After cache misses, the source cache block is invalidated. Otherwise, the same as Read.
`read_purge`	After CPU reads, the cache block is purged. The shared blocks in other caches are also purged.
`exclusive_read`	For the last word in a cache block, same as Read_Purge. Otherwise, the same as Read_Invalidate.
`direct_write`	If cache misses at block boundary, write data into cache without fetching from memory. Otherwise, ordinary memory write.
`lock_read`	Lock a memory word, then read the content.
`write_unlock`	Memory write, followed by unlock.
`unlock`	Unlock a memory word.

4.3 Local Coherent Cache Optimized for KL1

The PIM/p cache protocol is similar to Illinois protocol [16], but has several memory operations optimized for KL1 as described in Table 4.

In normal write operations, a fetch-on-write strategy is used. However, it is not necessary to fetch the contents of shared global memory when a new cache block is allocated for a new data structure. For example, in KL1, new data structures are created dynamically on the top of the heap area when the free lists for those structures are empty. To accomplish this, the `direct_write` instruction is introduced to avoid useless swap-in from shared memory. The *direct_write* instruction can also be useful as stack pushing operations in WAM-based architectures.

In KL1 parallel architectures, interprocessor communication (such as for goal distribution) uses a shared message buffer. In this case, swap-in and swap-out of meaningless data can be avoided by invalidating the sender's cache block after a cache-to-cache transfer and by purging the receiver's cache block after the receiver finishes reading. To accomplish this, the `exclusive_read`, `read_invalidate`, and `read_purge` instructions are introduced.

These new memory access instructions can reduce common bus traffic by avoiding useless

swap-in and swap-out operations. Cache simulations [8] indicate that these optimizations reduce bus traffic by 40–50% with respect to an unoptimized system. Direct write affords 35–45% reduction and other optimizations only 5% reduction. From the evaluation in [22], we believe these optimizations will prove effective on other parallel logic programming architectures as well.

Lock operations are essential in shared global memory architectures. The KL1 language processor uses lock operations for heap and communication area accesses [17]. The frequency of locking and unlocking shared data is high. The simulation result in [8] shows more than 5% of all memory accesses. However, actual lock conflicts seldom occur [17]. Therefore, it is effective to introduce a hardware lock mechanism that has less overhead when there are no lock conflicts.

The PIM/p cache enables a *lightweight* lock and unlock operation by using the cache block status, lock address registers, and busy-wait locking scheme. When CPU issues a lock command to its cache to attempt a `lock_read` instruction, the cache checks the corresponding address tag and status tag. If the address hits and its status is *exclusive*, the address can be locked without using the common bus. The locked address is held in a lock address register. When other processor attempts to access the locked address, the access itself is automatically postponed until unlocked. This lock protocol is effective for reducing the bus traffic of lock/unlock operations: for KL1, no bus cycles are needed for the high percentage of lock reads hitting in exclusive blocks and unlocks to non-waiting locks.

5 Processor Element Implementation

A PIM/p processor element will be implemented on a single board, which includes CPU, internal instruction memory (IIM), cache system, and two co-processors: a network interface unit (NIU) and a floating point processor unit (FPU), as shown in Figure 2. The target of the basic machine cycle is 50 nanoseconds. The LSIs are now being fabricated by CMOS standard cell technology that can include up to 80K gates.

The PIM/p has a 4G-byte global virtual address space on each cluster. KL1 data is represented by 40-bit word (an 8-bit tag and 32-bit data). Normal KL1 data is placed by 40-bit KL1 tagged data in aligned 64-bit words in the PIM/p memory system, while instructions and some data structures, such as strings or floating point numbers, are placed on a byte

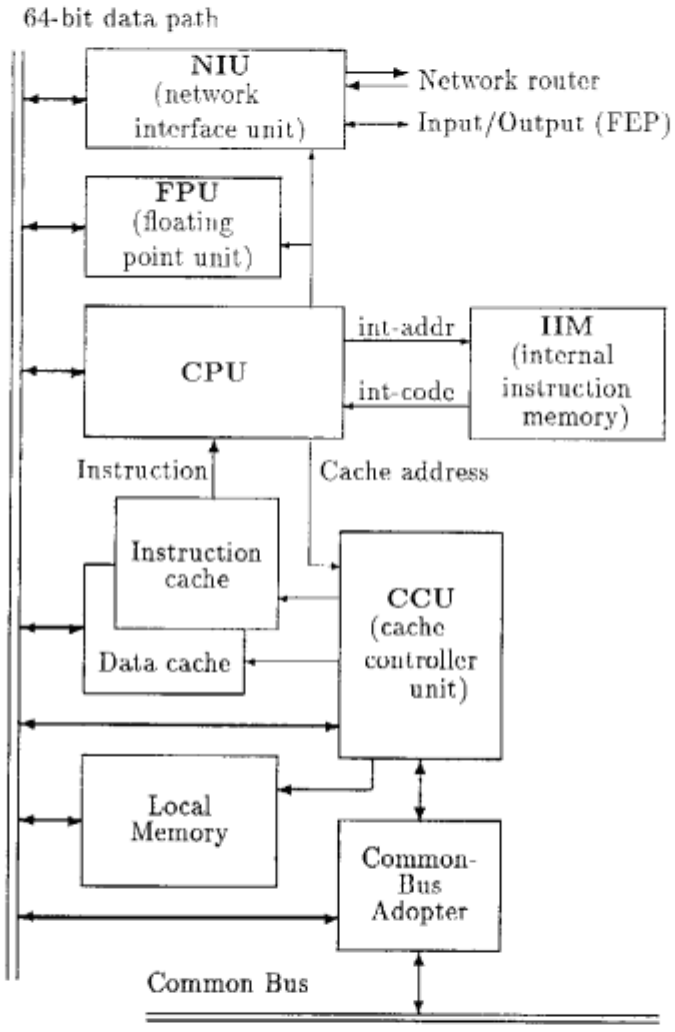

Figure 2: PIM/p Processor Element Configuration

boundary.

5.1 Cache System

The processor element includes two caches: an instruction cache and a data cache. The instruction cache supplies the instruction buffer in CPU with external instruction stream in parallel with data accesses by CPU. The contents of both cache memories are identical, so that, in a branch instruction, CPU can fetch a branch target instruction from the data cache as shown in section 5.3.

The cache controller unit (CCU) manages both the instruction cache and the data cache. The cache address array would be updated by both commands from the CPU and a common bus. To avoid the access conflict, the common bus adopter has a copy of cache address array with cache block status.

In general, a larger cache is necessary to maintain a high hit-ratio. The simulation in [8] shows at least 64K bytes capacity is necessary for KL1. However, it is preferable to give up forming a large cache by enlarging the cache block size. This is because the simulation results in [8] have also found that a cache block larger than four tagged words causes an increase in shared blocks between caches in parallel execution of KL1, so that mutual cache invalidation may increase. On the other hand, it is found difficult to provide an address array for 64K bytes of 32-byte block (four tagged words), because the size of the cache address array is restricted by the LSI capacity of the cache controller unit (CCU). Through these observations, we designed the following cache system. The capacity of both the instruction and data caches is 64K bytes. The CCU has a block status tag for each 32-byte block, and an address tag for each two blocks, that is, every 64 bytes. Our simulation result also shows that that scheme does not decrease the performance so much compared to a full 32-byte block cache of the same capacity.

5.2 Registers

The CPU in the processor element includes 32 general-purpose registers, several dedicated registers, indirect value registers, and indirect access registers (see section 3.4). These registers are specified by a 6-bit register specifier in most instructions. Each general-purpose register has an 8-bit tag and 32-bit data.

Table 5: Pipeline Stage and its Operation

	ALU operation	*Memory access*	*Branch*
D	Decode	Decode / register read (address)	Decode / register read (address)
A	–	Operand address calculation	Branch address calculation
T	Register read	Cache address access	Cache address access
B	ALU operation / register write	Cache data access / (register write)	Cache data access / condition test

The dedicated registers include a condition code register and a slit-check register (see sections 5.4). Most flags, such as the condition code, are placed in the tag part of the dedicated registers, and can be tested by the tag-branch instructions.

In addition to the above registers, NIU and FPU have several co-processor registers, which are handled only by co-processor interface instructions.

5.3 CPU Execution Pipeline

The CPU has two instruction streams, one is from the instruction cache, and the other is from the internal instruction memory (IIM). The CPU uses an instruction buffer and a four-stage pipeline, **D A T B**, to attempt to issue and complete an external instruction every cycle. External instructions are either four or six bytes long, so that the instruction buffer has a hardware aligner. Each internal instruction requires two additional stages, preceding stage **D**, to set the internal instruction address (stage **S**) and to fetch the instruction (stage **C**).

Table 5 shows the pipeline stages and corresponding operation. General-purpose registers are updated only at the last **B** stage, thereby avoiding write conflicts. Internal forwarding is done by hardware so that the result of a register-to-register instruction can be used by the next instruction even though that result has not yet been written to the general registers.

In a branch instruction to an external instruction, the branch target instruction is fetched at stage **B** in the same way as memory read instructions. Therefore, ordinary branch instructions may cost three additional cycles to branch. Delayed branch instructions can avoid one of the three cycles by executing an effective instructions.

Most tag branch instructions test their condition at stage **B**. However, macro-call instructions and internal branch instructions test their condition at stage **A**. Figure 3 shows

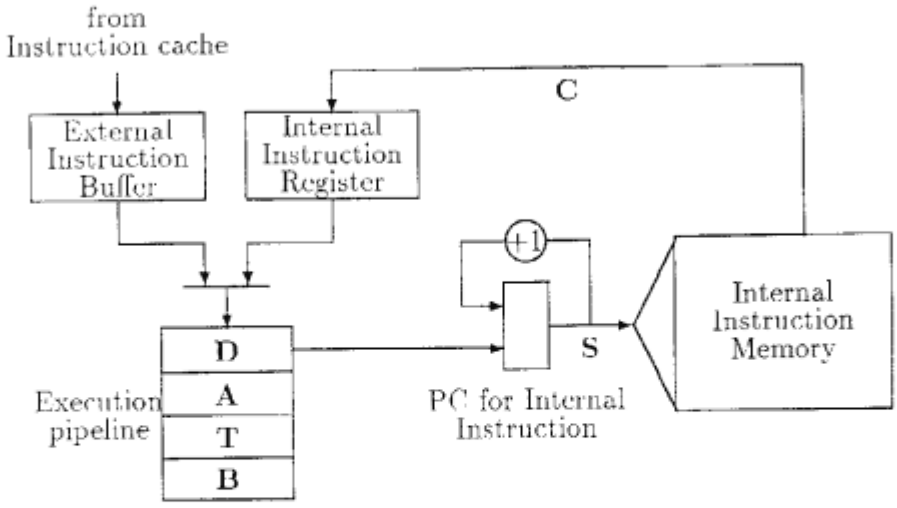

Figure 3: Macro-call Instruction Mechanism

the invocation mechanism of the macro-call instruction, and Figure 4 shows their pipelining features. A macro-call instruction puts the entry address in the program counter for internal instructions and initiates the internal instruction fetch (stage **S**) at its stage **D**, then tests its condition at stage **A**. When the condition is true, the program counter for external instructions is frozen at this point, cancelling the next external instruction. Therefore, a macro-call instruction costs only one additional cycle to invoke a subroutine in the internal instruction memory. In addition, delayed macro-call instructions are provided to avoid the penalty. Return from macro-call, that is, return from internal instructions to external instructions, can be indicated by a one-bit flag: *eoi*, in each internal instruction except for branch instructions. When an internal instruction with *eoi* is put into the pipeline, the instruction stream is switched back to external at stage **D**, and the external instruction frozen by the previous macro-call instruction follows without waiting cycles. (See Figure 4.)

5.4 Slit-check and Interrupt

Various events may arise asynchronously during KL1 execution, such as: other processors requires a garbage collection of shared memory. However, the actions corresponding to these events are delayed until a current goal reduction finishes, even if the event occurred during a goal reduction. This is because garbage collection is difficult to start during a goal reduction. So, they may be delayed until after the goal reduction finishes. The detection of these events at the end of goal reduction is called *slit-checking.*

When the condition is true:

D	A	(condition test at A)					: macro-call instruction
	D	(cancelled)					: next external instruction
S	C	D	A	T	B		: first internal instruction
	S	C	D	A	T	B	: second internal instruction

When the condition is false:

D	A	(condition test at A)					: macro-call instruction
	D	A	T	B			: next external instruction
		D	A	T	B		: external instruction

End of macro body:

S	C	D	A	T	B		: internal instruction *eoi*
	S	C	(cancelled)				: internal instruction
		S	(cancelled)				: internal instruction
			D	A	T	B	: next external instruction

Figure 4: Pipelining Features of Macro-call Instruction

The processor element of PIM/p incorporates a hardware mechanism for *slit-checking* as well as ordinary interrupts for debugging and error detections. A hardware interrupt, in general, causes automatic save of program status, slit-checking does not. Each processor element has flag registers, each of which can keep an individual event, such as signals from other processors and network packet arrival. The slit-checking mechanism has an additional flag to show whether any events has happened or not, which can be tested by one conditional branch instruction. Therefore, the KL1 language processor can detect the normal but asynchronous events by itself at appropriate point. On general purpose computers, the slit-checking might be implemented using normal interrupt mask/unmask operations and a cumbersome interrupt handler. It would cost too much for the KL1 system. By incorporating the hardware slit-checking mechanism, the processor element can avoid frequent mask/unmask operations and interrupt handling overhead.

6 Conclusion

This paper describes the design of processor element architecture for the parallel inference machine prototype, PIM/p. The execution features of the concurrent logic programming language, KL1, were observed, and its architectural issues were discussed. The innovative processor architecture for KL1 and with its design decisions were presented. The processor is designed based on tagged architecture. With the variety of tag handling operations, instructions can be executed by one cycle pipeline. Macro-call instructions are introduced to enable lightweight subroutine call function for polymorphic operations in unification, so that system designers can easily define high level instructions. Dedicated instructions are introduced to support incremental garbage collection embedded in KL1 unifications. Local coherent cache and optimized memory operations tailored to the memory access characteristics of KL1 are designed, which can reduce common bus traffic within shared memory multiprocessors. These features incorporated in the processor architecture can be expected to suit other concurrent logic programming languages. The LSIs are now being fabricated by CMOS technology.

Acknowledgement

We wish to thank all of the PIM research members both at ICOT and Fujitsu Limited. Especially we thank ICOT researchers: Mr. A. Matsumoto, and Mr. T. Nakagawa, Mr. K. Nakajima, and Fujitsu researchers: Mr. S. Arai and Mr. A. Asato, for their useful comments. We also wish to thank Mr. H. Murano in Fujitsu Limited for his help in developing the LSIs and his useful comments. Finally, we would like to thank ICOT Director, Dr. K. Fuchi, the chief of the fourth research section, Dr. S. Uchida, the general manager of Information Processing Division in Fujitsu Laboratories, Mr. J. Tanahashi, and the manager of Artificial Intelligence Laboratory in Fujitsu Laboratories, Mr. H. Hayashi, for their valuable suggestions and guidance.

References

[1] J. Archibald and J. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transaction of Computer Systems*, 4(4):273–298, 1986.

[2] P. Bitar and A. M. Despain. Multiprocessor cache synchronization. In *Proc. of the 13th Annual International Symposium on Computer Architecture*, pages 424–433, June 1986.

[3] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 276 293, 1987.

[4] T. Chikayama, H. Sato, and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. of the International Conference On Fifth Generation Computing Systems 1988*, Tokyo, November 1988.

[5] J. Cohen. Garbage Collection of Linked Data Structures. *ACM Computing Surveys*, 13(3):341–367, Sept. 1981.

[6] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proc. of the 10th Annual International Symposium on Computer Architecture*, pages 124–131, 1983.

[7] A. Goto et al. Overview of the Parallel Inference Machine Architecture (PIM). In *Proc. of the International Conference On Fifth Generation Computing Systems 1988*, pages 208 – 229, Tokyo, Japan, November 1988.

[8] A. Goto, A. Matsumoto, and E. Tick. Design and Performance of a Coherent Cache for Parallel Logic Programming Architectures. In *16th Annual International Symposium on Computer Architecture*, pages 25 – 33, Jerusalem, Israel, May 1989.

[9] A. Goto and S. Uchida. Toward a High Performance Parallel Inference Machine –the Intermediate Stage Plan of PIM–. In *Future Parallel Computers*, pages 299–320. LNCS 272, Springer-Verlag, Pisa, Italy, 1986.

[10] A. Hattori, T. Shinogi, K. Kumon, and A. Goto. Architecture of Parallel Inference Machine: PIM/p. In *JSPP'89*, pages 107–114. IPSJ, Feb. 1989. (in Japanese).

[11] M. Hill et al. Design decisions in SPUR. *IEEE Computer*, 19(11):8–24, November 1986.

[12] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 468–477, 1987.

[13] A. Matsumoto et al. Locally Parallel Cache Designed Based on KL1 Memory Access Characterestics. TR 327, ICOT, 1987.

[14] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, and T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 436–451, Lisboa, June 1989.

[15] H. Nakashima and K. Nakajima. Hardware architecture of the sequential inference machine: PSI-II. In *Proceedings of 1987 Symposium on Logic Programming*, pages 104–113, San Francisco, 1987.

[16] M.S. Papamarcos and J.H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348–354, 1984.

[17] M. Sato et al. KL1 Execution Model for PIM Cluster with Shared Memory. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 338–355, 1987.

[18] Inc. Sequent Computer Systems. *Sequent Guide to Parallel Programming*, 1987.

[19] L.C. Stewart et al. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8), August 1988.

[20] Y. Takeda et al. A Load Balancing Mechanism for Large Scale Multiprocessor Systems and its Implementation. In *Proc. of the International Conference On Fifth Generation Computing Systems 1988*, Tokyo, November 1988.

[21] K. Taki et al. Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI). In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 398–409, Tokyo, 1984.

[22] E. Tick. Performance of Parallel Logic Programming Architectures. TR 421, ICOT, 1988.

[23] K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog - Collected Papers*, pages 140–156. MIT Press, 1987.

[24] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, Artificial Intelligence Center, SRI, 1983.

並列推論マシン PIM/p の要素プロセッサアーキテクチャ

Processor Element Architecture for Parallel Inference Machine : PIM/p

後藤 厚宏[*1], 篠木 剛[*2], 近山 隆[*1], 久門 耕一[*2], 服部 彰[*2]

Atsuhiro Goto[*1], Tsuyoshi Shinogi[*2], Takashi Chikayama[*1], Kouichi Kumon[*2], Akira Hattori[*2]

*1: 新世代コンピュータ技術開発機構

*1: Institute for New Generation Computer Technology (ICOT)

*2: 富士通株式会社

*2: Fujitsu Limited

概要

本稿では、並列推論マシン PIM/p の要素プロセッサのアークテクチャおよびその設計方針ついて述べる。本プロセッサにおいては, 並列論理型言語に適したアーキテクチャを採用した。プロセッサはタグアーキテクチャに基づいて設計されており, 豊富なタグ操作機能を含め, マシン命令は 1 サイクルパイプラインによって実行することができる。KL1 のユニフィケーションは, 動的データ型判定の結果によってそれに続く操作が決まる。このような多義性のある操作を効率良く実装するために, マクロ命令と呼ぶ, コストの小さいサブルーチン呼びだし機能を導入した。これにより, ユニフィケーションに相当する高機能命令の設計が容易になる。また, 並列論理型言語にとって重要な実行時のガーベジコレクションを支援する専用マシン命令を用意した。さらに, KL1 のメモリ参照特性の解析に基づいて設計した共有メモリ用局所キャッシュと最適化キャッシュ命令を導入し, 共有バストラヒックの低減を計った。現在, プロセッサを構成する LSI を CMOS スタンダードセル技術によって製造中である。

並列推論マシン PIM/p の要素プロセッサアーキテクチャ

Processor Element Architecture for Parallel Inference Machine : PIM/p

後藤 厚宏 *1, 篠木 剛 *2, 近山 隆 *1, 久門 耕一 *2, 服部 彰 *2

Atsuhiro Goto*1, Tsuyoshi Shinogi*2, Takashi Chikayama*1, Kouichi Kumon*2, Akira Hattori*2

*1: 新世代コンピュータ技術開発機構

*1: Institute for New Generation Computer Technology (ICOT)

*2: 富士通株式会社

*2: Fujitsu Limited

1. はじめに

ICOT の第五世代コンピュータプロジェクトでは, 数 100 台またはそれ以上の要素プロセッサを結合した並列推論マシンの研究開発を進めている。 PIM/p は, 密に結合された 8 台の要素プロセッサからなるクラスタをネットワークによって階層的に接続した並列推論マシンのパイロットマシンである。

並列推論マシンの核言語 KL1 は, 純粋な並列論理型言語である GHC に基づいて開発された言語である。 KL1 のような並列論理型言語は, 宣言的な並列プログラミングを可能とし, 詳細かつ直接的な同期操作の記述が不要となるため, 今後の大規模並列ソフトウェア研究において重要な役割を担うと考えられる。しかし, KL1 は: (1) 基本となるユニフィケーションは動的データ型判定に基づく多義的操作である; (2) 並列処理されるゴール (プロセス) の粒度が小さく, それらの間のデータ依存性によって頻繁に実行コンテキストが切り換わる; (3) 単一代入の性質により, 高いメモリ参照能力と効率的なメモリ管理機能を必要とする, といった従来の計算機では適合し難い性質を持っている。本稿では, 以上のような並列論理型言語の特性に適合することができる並列推論マシン PIM/p の要素プロセッサのアークテクチャおよびその設計方針ついて述べる。

2. RISC 指向命令とマクロ機能

並列論理型言語のユニフィケーションは, 動的データ型判定の結果によってそれに続く操作が決まる。並列論理型言語向きのプロセッサにおいては, このような多義性のある操作をいかにして支援するかが重要となる。動的データ型判定のためには, まず, タグアー

キテクチャが基本となる。近年，タグアーキテクチャをRISC方式の命令で実現するプロセッサも提案されている。ただし，KL1の単一化のような多義的な操作を実現しようとすると，コンパイルコードが大きくなってしまい，キャッシュミスが頻発する恐れがある。特に，密結合マルチプロセッサでは，共有バストラヒックが増加し，全体性能の低下に結び付き易いことがわかった。そこで，PIM/pの要素プロセッサは，このような多義性のある操作を効率良く実装するために，マクロ命令と呼ぶ，コストの小さいサブルーチン呼びだし機能を導入した。

要素プロセッサの各命令は，豊富なタグ操作機能を含め，4段のパイプラインを用いて，(50ナノ秒を目標とする) 1マシンサイクルに1命令の割合で実行できる。KL1のユニフィケーションのような高機能命令はマクロ命令を利用して実装する。マクロ命令は引数として指定されたレジスタのデータ型判定に基づき，小さいコストでマクロの本体の条件呼び出しができる。マクロの本体は通常の命令とほぼ同様の内部命令によって記述でき，マイクロプログラム方式に比べて実装が用意である。また，マクロの本体は各プロセッサの内部命令メモリから供給されるため，マクロ本体の実行中は共有メモリからの命令フェッチが節約できる。

3. ガーベジコレクションの支援

KL1はデータの破壊的書き換えを許さない単一代入言語であるため，メモリ管理の役割が従来の計算機にも増して重要となる。MRB方式は，データセルへのポインタに，そのデータへの参照数を示す1ビットのフラグ(MRB)を付加することにより，実行時にメモリ回収，再利用を行うガーベジコレクション方式である。MRB方式では，メモリ参照の局所性が高いという利点を持つが，通常のプロセッサでは，1ビットのMRB情報の管理が効率良くできない。そこで，PIM/pの要素プロセッサでは，MRB方式のガーベジコレクションを支援する命令を用意している。

4. 並列論理型言語向き一貫性キャッシュの設計

PIM/pのクラスタのような密結合マルチプロセッサ上におけるKL1のメモリ参照特性の評価に基づいて，並列論理型言語向きの並列一貫性キャッシュを設計し，要素プロセッサに導入した。KL1は，その単一代入性によりメモリへの書き込み比率が高いため，無効

化方式のライトバック型プロトコルを基本とした。ここで，各クラスタにおいてプロセッサ台数に比例した並列処理性能を得るためには，共有バスネックの回避が必要である。そこで，バスの転送能力を高くするとともに，KL1 の並列処理の特性を活かしてバストラヒックを節約するキャッシュコマンドを用意した。

また，共有メモリ上の変数を介した通信ではメモリアクセスの排他制御機構が重要である。そこで，各プロセッサのキャッシュのブロック状態を利用したロック機構を設け，低コストの排他制御を可能としている。