PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

UNIREDII: The High Performance Inference Processor for the Parallel Inference Machine PIE64

Kentaro Shimada Hanpei Koike Hidehiko Tanaka
H.Tanaka Lab., Department of Electrical Engineering,
Faculty of Engineering, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan
E-Mail:{shimada,koike,tanaka}@mtl.t.u-tokyo.ac.jp

Abstract

UNIREDII is the high performance inference processor of the parallel inference machine PIE64. It is designed for the committed choice language Fleng, and for use as an element processor of parallel machines. Its main features are: 1) a tag architecture, 2) three independent memory buses (instruction fetching, data reading, and data writing), 3) a dedicated instruction set for efficient execution of Fleng, 4) multi-context processing for reducing pipeline interlocking and overhead of inter-processor synchronization. With the multi-context processing mechanism, the internal pipeline is shared by several independent instruction streams (contexts), and which contexts are to be executed is determined cycle by cycle. So, UNIREDII acts as a pipeline-shared MIMD processor. In this paper, several architectural features and the instruction set are explained. And performance measurement results by simulation are also presented. High performance (about 1M RPS[1] with 10MHz clock) is attained, and it is shown that the multi-context processing mechanism is very effective for improved performance.

1 Introduction

Committed choice languages are designed for efficient parallel execution of logic programs, but, because of their parallel and logic semantics, high performance is hardly achieved by conventional processors which are designed for sequential and procedural languages. Therefore we designed a dedicated processor for the element processor of the parallel inference machine PIE64 [Koike and Tanaka 1988], which we are now developing, and named UNIREDII. PIE64 executes committed choice language Fleng[Nilsson and Tanaka 1988], and has sixty-four processor elements.

For design decisions in UNIREDII, we paid special attention to the following points.

1. The processor architecture should be tuned for the execution of Fleng programs.
2. It must be equipped with the features of an element processor of parallel machines.

For the first point, we designed a dedicated instruction set for executing Fleng based on the experience of developing Fleng interpreters on workstations. For the second point, we proposed the multi-context processing mechanism for reducing inter-processor synchronization, and the independent coprocessor command bus to interconnect network interface processors and a process management processor.

UNIREDII is implemented in 1.2 μ CMOS gate array and driven with 10MHz clock. This clock rate is selected because UNIREDII must synchronize with the local memory bus, which in turn synchronizes with the network hardware of PIE64.

An overview of PIE64 is given in section 2. In section 3, the architecture including the multi-context processing mechanism and instruction set of UNIREDII are described. In section 4 and 5, some simulation results are presented and discussed. Finally, we conclude this paper in section 6.

2 PIE64

PIE64 is one of several parallel inference machines which executes parallel logic programming languages such as Fleng[Nilsson and Tanaka 1988] and KL1[Kimura and Chikayama 1987]. We designed the committed choice language Fleng for PIE64 so that it is easy to implement and easy to attain high performance. PIE64 has sixty-four processor elements, which are called inference units (IUs), and two independent interconnection networks (Process Allocation Network: PAN and Data Access/Allocation Network: DAAN)[Takahashi *et al.* 1991]. These interconnection networks are implemented as circuit switching, and have a special function of broadcasting load information for automatic load balancing so that each IU can select the minimum load IU automatically.

[1] RPS: Reduction Per Second

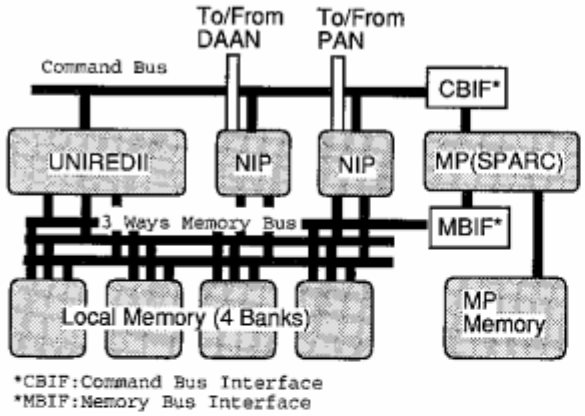

Figure 1: organization of the inference unit of PIE64

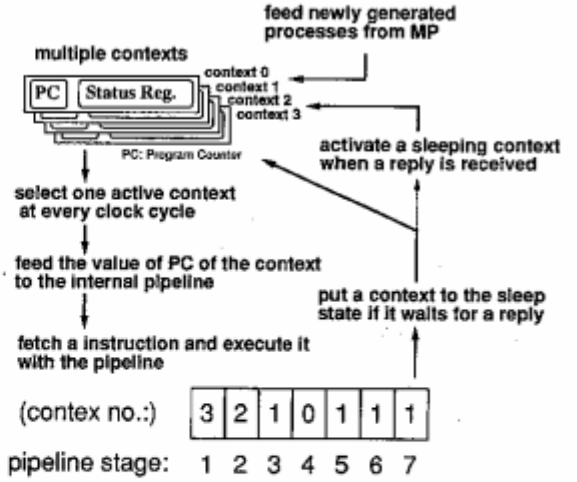

Figure 2: overview of the multi-context processing mechanism

Each IU has an inference processor UNIREDII, two[2] network interface processors(NIPs) [Shimizu *et al.* 1991], a management processor(MP), and a local memory of four banks (see figure 1). NIP, which is a dedicated processor as well as UNIREDII, performs inter-IU communicating/synchronizing functions in a form suitable for Fleng execution. The transmission throughput is 10M word per sec., namely 40MByte per sec. at one connection for each network. MP manages process scheduling, load distribution, and load balancing, and performs other functions such as system maintenance. We use a general purpose processor, SPARC, as MP to make process management flexible. Thus, in the IUs, we use functional parallelism by the three kinds of processor, where UNIREDII performs *computation*, NIP performs *communication and synchronization*, and MP performs *process management*.

In an IU, these three kinds of processors share the local memory, and access it through a three way memory bus which is driven with a 10MHz clock to synchronize with network access over NIPs. So we can get a throughput of 10M word (40MByte) per sec. each way, namely 120MByte per sec. in all. In addition, UNIREDII, NIPs, and MP communicate with each other through a coprocessor command bus using a specialized protocol for command-reply among these processors. The format of the coprocessor commands is determined with the Fleng data types taken into account.

[2]In practice, there are four network interface processor chips in one IU. Two of them act in master mode and can start the action of network connection while the other two act in slave mode and respond to the master NIP when requested.

3 UNIREDII Architecture

3.1 Overview

UNIREDII is a dedicated processor. It was designed for executing Fleng programs efficiently and to meet requirements of an element processor for parallel machines. Its main features are:

1. a tag architecture
2. three independent memory buses (instruction fetching, data reading, and data writing)
3. multi-context processing
4. a dedicated instruction set to execute Fleng programs efficiently

All instructions of UNIREDII have one word (32 bits) length, and are single-cycle instructions. Also its data types have a length of 32 bits, and consist of two mark bits for garbage collection, two tag bits, and twenty-eight bits of value part (pointer types); or two mark bits, six tag bits, and twenty-four bits of value part (constant types).

3.2 Multi-Context Processing

The internal pipeline of UNIREDII is shared by multiple instruction streams (contexts). UNIREDII executes them concurrently, and which contexts should be executed is determined cycle by cycle. In other words, UNIREDII acts as a pipeline-shared MIMD processor (see figure 2). Because Fleng is a parallel language which generates many instruction streams in parallel, we can expect to get enough instruction streams to fill the contexts of UNIREDII. Process scheduling is determined by the management processor, and UNIREDII starts a new process as one of the contexts by receiving a appropriate

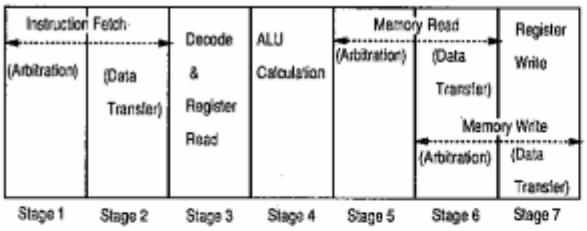

Figure 3: pipeline organization of UNIREDII

coprocessor command from MP. And when one of the contexts waits for a reply of some remote memory access, UNIREDII puts the context to sleep state and fills its pipeline with the other contexts dynamically.

Two kinds of aim of the multi-context processing exist. 1) To reduce pipeline interlocking caused by pipelined executions of the instructions (intra-processor effect). 2) To reduce the cost of process switching due to remote memory access (inter-processor effect). Especially, the second effect is a very important feature for an element processor of parallel machines.

The most remarkable point of the multi-context processing mechanism of UNIREDII is that it can continuously execute instructions of only one context while the other contexts are sleeping (as shown at the stage from 5 to 7 in the figure 2). UNIREDII has a pipeline interlocking mechanism to keep dependency among instructions of one context. Other processors which have similar mechanisms, such as HEP[Jordan 1983], MASA [Halstead and Fujita 1988], and CPC[Shimuzu *et al.* 1989], do not have pipeline interlocking mechanism and cannot execute one instruction stream in continuous cycles. Therefore they slow down dramatically as the number of executable instruction streams decrease. That is not the case of UNIREDII.

Due to the restriction of available number of gates, the number of contexts is limited to four. But that is enough to get the full effect of multi-context processing, as we show later.

3.3 Hardware Organization

3.3.1 Pipeline Organization

Figure 3 shows the internal pipeline organization of UNIREDII. The pipeline consists of seven stages. The main reason why as many stages as seven are required is that, in an IU, all memory access needs two phases, one of which is the bus arbitration phase in which the four kinds of independent accesses from UNIREDII, two NIPs, and MP are arbitrated (see figure 1), and another is the data transfer phase in which memory access is actually performed, so that the memory access time hides the bus arbitration time. Thus, at the first and the second stages, UNIREDII fetches an instruction from the local memory, decodes it and reads registers at the third stage, executes it at the fourth stage, reads data from the memory at the fifth and the sixth stages, and writes data into the memory at the sixth and the seventh stages. Also registers are written at the seventh stage. UNIREDII has thirty-two general purpose registers per context, namely 128 general purpose registers in all. By means of this pipeline organization, UNIREDII makes efficient use of its three memory buses and can execute test-and-set type instructions which require two times of memory access (one read and one write) in one cycle without any pipeline holding because the data reading and writing buses are handled by different stages. These type instructions are very important for processing elements of parallel machines.

The effectiveness of the pipeline architecture is determined by when and where the pipeline interlock occurs. As for UNIREDII, the pipeline interlock will occur when not all contexts can be executed. In that case, taking a jump (at the fourth stage) will invalidate at most three following, already-fetched instructions of the same context at the first, the second, and the third stages. And reading registers (at third stage) which are destinations of load instructions executed within three preceding cycles will cause pipeline hold. When all four contexts can be executed, no pipeline interlock occurs because instructions of the same context are not executed and registers of the same context are not read in any four continuous cycles.

3.3.2 Mechanism of Remote Memory Access

In principle, memory-accessing instructions of UNIREDII can execute remote memory access automatically. UNIREDII has a special register which holds the IU identifier of six bits, and it compares the IU identifier field of the memory address (the upper six bits of the address of twenty-eight bits) with the IU-id register when executing memory access instructions. When they are not equal, UNIREDII issues a remote memory access command to NIP instead of accessing its local memory. The result of the remote memory access is sent back as a coprocessor reply from NIP.

UNIREDII should receive the replies of the remote memory access commands correctly. For this purpose, all general purpose registers of UNIREDII have a special bit indicating that they are waiting for replies or not. When an instruction reads the register whose reply wait bit is set before the reply is received, UNIREDII cancels the instruction and puts its context to sleep. And, after receiving the reply, UNIREDII wakes the context up and re-executes the canceled instruction.

Table 1: the instruction set of UNIREDII

Dereference	derf	dereference	dfcl	dereference and check list
	dcll	dereference and check list and load car	dfcv	dereference and check vector
	dcvl	dereference and check vector and lod top	dfcc	dereference and check constant
Execute	exec	execute	exel	execute on list
	exll	execute on list and load car	exev	execute on vector
	exvl	execute on vector and load top	exct	execute on constant
Manipulate	cpir	copy if remote	cprr	copy if remote with register
Structure	cvtp	check vector top	cvtr	check vector top with register
Load /	ld	load	tgld	tagged load
Store	ldst	load and store	tlds	tagged load and store
	st	store	sudf	store undefind code
	stim	store immediate		
Active	bind	bind variable	bdim	bind with immediate
Unification	cvos	check variable order and swap		
Heap Allocation	allc	allocate		
Flow	jump	jump	call	call
Control	jcmp	jump on compare	jncp	jump on not compare
	jtag	jump on tag	jntg	jump on not tag
	jrmt	jump on remote pointer	jloc	jump on local pointer
	jcc	jump on flag condition	stop	stop
Coprocessor	cpcm	send coprocessor command		
Garbage	ldsm	load and store mark	stmm	store with modified mark
Collection	jmrk	jump on mark/stop condition		
Set	setc	set constant	sett	set mark and tag
	seta	set alternative pointer	setf	set condition flags
	clrf	clear condition flags	gfc	get flag condition
Arithmetic	add	add	adc	add with carry
and Logical	sub	subtract	sbb	subtract with borrow
	and	bit-wise and	or	bit-wise or
	xor	bit-wise exclusive or	rol	rotate left
	ror	rotate right	rcl	rotate with carry left
	rcr	rotate with carry right	shl	shift left
	shr	shift right	asr	arithmetic shift right
Management	spid	set pid register	pidt	preset pid and tag
	shp	set heap pointer	lhp	load from heap pointer
	exhp	exchange heap pointer set	stp	set scratch area pointer
	ltp	load from scratch area pointer	stf	store flags
	ldf	load from flags		

3.4 Instruction Set

Table 1 shows the instruction set of UNIREDII. We describe several notable instructions of it in the following subsections.

3.4.1 Dereference Instructions

The dereference instructions are most characteristic of the instruction set of UNIREDII. They dereference links of a variable and get the value of the variable. To dereference one link per cycle, they recursively jump to themselves when the value of the operand register of them is a pointer to a variable. In that case, they read the content of the address of the variable, write its value into the same operand register, and jump to themselves. As a result, they have dereferenced one link. By means of using this mechanism in addition to the cycle-by-cycle context switching, instructions of the other contexts can be executed even while a dereference loop is processed.

One significant point of the dereference instructions is that they have special ability for committed choice languages. That is the suspension register mechanism. In head matching of committed choice languages, a goal may suspend when a component of its arguments is an unbound variable while the corresponding component of the current clause head is not a variable. After trying all alternative clauses and committing no clauses, the goal really suspends. Therefore when a goal may suspend at one of alternative clauses, the variable which caused the suspension must be recorded to hook the goal by some suspension stack mechanisms until all alternative clause are tried. In the case of UNIREDII, the variable is recorded in the suspension register (general purpose register R30) by the dereference instructions.

There are two kinds of effect of the suspension register mechanism. First, the suspension stack, which is in memory in the case of other similar processors [Kimura and Chikayama 1987], can easily be implemented in reg-

```
append([H | T], X, Y) :- Y = [H | Z], append(T, X, Z).
append([], X, Y)      :- X = Y.

append:
        seta    $suspend, ap
        dcll    r1, $2, r4        ; [H |
$1:
        tgld    [r1 + 1], r1      ; T]
        allc    s, lst, 2, r5     ; [
        st      r4, [r5], %tmp    ; H |
        sudf    [r5 + 1], r6      ; Z]
        bind    r5, r3, r7        ; Y = [ H | Z]
        jntg    r7, udf, $check1
        mov     r6, r3            ; Z)
        exll    r1, $1, r4        ; [H |
$2:
        dfcc    r1, nil, $fail    ; []
        cvos    r2, r3, r2, r3    ; X = Y.
        bind    r2, r3, r7
        jntg    r7, udf, $check2
        succ    gtp, mp
        stop
```

Figure 4: an example of compiled codes (append)

isters so that the suspension check is speeded up. Second and more important, when the head matching is deterministic, as is often the case with real programs, once a goal suspends at one of the alternative clauses, the goal suspends after all. Therefore no suspension stack in memory is necessary in that case. The suspension register mechanism also speeds up this case.

Several combined instructions exist among the dereference instructions. The dereference-and-check-list (dfcl) instruction checks the dereferenced value to determine whether it is a pointer to a list or not, and the dereference-and-check-list-and-load-car (dcll) instruction reads the car part of the list if the dereferenced value is a pointer to a list. Similarly the dereference-and-check-vector (dfcv) instruction checks the dereferenced value to determine whether it is a pointer to a vector, and so on. These instructions are capable of a two-way jump, one for suspension and the other for pointer type check failing. The jump addresses are given by the offset value from the instruction itself and the alternative pointer register (general purpose register R28).

Another kind of combined instruction is that the execute instructions, which execute tail recursions, are combined with those dereference instructions so that they optimize the tail recursion and the consequent head matching sequence.

3.4.2 Arithmetics and Bit-wise Logic Instructions

The arithmetic and bit-wise logic instructions of UNIREDII are very similar to those of conventional processors. They exist for compiling such things as built-in arithmetic predicates. One difference between those of

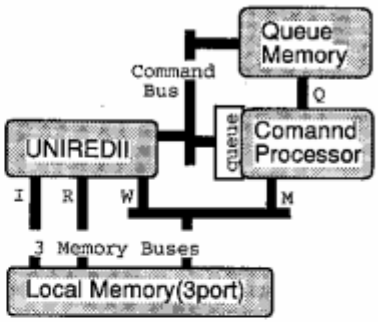

Figure 5: the simulation model used for the evaluation

Table 3: the number of clock cycles which are necessary for emulating co-processor commands issued by UNIREDII

command	description	to	reply	num.of cycles
newgoal	enqueue a new goal	MP	no	1
endreduce	end of a reduction	MP	no	1
suspend	suspend a goal	MP	no	7
deref	dereference a variable	NIP	yes	16
bind	bind a variable	NIP	yes	17
read2	read a remote list	NIP	yes	19
activates	activate a goal	NIP	no	8

UNIREDII and those of conventional processors is that those of UNIREDII check the tag part of the operands and set the tag error flag bit of the flag register according to the value of the tags[3]. Therefore there are several switches of those instructions to deal with various tag types. For example, the add.i instruction (.i switch on) adds an integer to another integer (otherwise set the tag error flag), the add.p instruction adds an integer to a pointer, and the add.b instruction adds total 32-bits to 32-bits and does not change the tag error flag.

3.4.3 An Example of Compiled Code

Now, we present an example of compiled code of UNIREDII in figure 4. It is the code compiled from a deterministic append program. In the tail recursion loop (between label $1 and $2 in the figure), there are only eight instructions. Therefore UNIREDII can execute the append program at a maximum rate of 1.25 million reductions per second with the clock rate of 10 MHz.

4 Simulation Results

4.1 Simulation Model

Figure 5 shows a simulation model used for the evalu-

[3]To simplify the hardware, there are no tag error trap mechanisms in UNIREDII.

Table 2: several aspects of the sample programs which are revealed by the simulation

program	append 100	nreverse 30	qsort 50	primes 100	8 queens
total clock cycles	1435	5427	8162	41068	656011
times of reduction	101	496	380	726	38878
times of suspension	0	29	122	103	558
num.of executed instructions	816	4858	7747	39352	647933
instructions per reduction	8.08	9.79	20.39	54.20	16.67
clock cycles per instruction	1.759	1.117	1.054	1.044	1.012

ation. We evaluated UNIREDII as a single, independent processor, and emulated the coprocessor commands issued by UNIREDII with the imaginary command processor shown in Figure 5. In a real IU of PIE64, these commands are processed by MP and NIPs. Table 3 shows the number of cycles which are necessary for emulating the commands with the command processor. As for network access commands, the number of cycles in the table is determined based on the NIP's performance from [Shimizu *et al.* 1991]. In addition, we used an independent queue memory for queuing newly spawned goals in the simulation model. This roughly corresponds to the MP memory in figure 1. The goal scheduling strategy with this queue memory is LIFO (Last-In/First-Out).

4.2 Performance with Sample Programs

First, we evaluate UNIREDII's performance under the condition that there is no remote memory access. We use, as the sample programs, append 100 (deterministic append of a list of length 100), nreverse 30 (naive reverse of a list of length 30), qsort 50 (quick sort of a list of length 50), primes 100 (generation of prime numbers up to 100), and 8 queens (the 8-queen problem). Table 2 shows some aspects of the sample programs, and figure 6 shows the performance with the sample programs. These are measured with 10 MHz clock.

As for append 100, the performance is comparatively low because, for spawning no sub-goals, the number of active contexts in the program does not exceed one and so the multi-context processing mechanism does not work. In this case, the pipeline interlock occurs frequently and therefore the performance is degraded in spite of only eight instructions in its reduction loop. Figure 7 shows the average number of active contexts about the sample programs. In the figure, more than three contexts are active in average about the other four programs. Consequently we can get enough effect of the multi-context processing in these programs.

Another example of low performance is that of primes 100 because there are no multiplier/divider units in UNIREDII and it takes long time to carry out the divisions which that program requires through integer additions and subtractions. According to table 2, there are more than fifty instructions per reduction in that program, and this is over twice as big as in other programs such as quick sort 50 and 8 queens. This is because it takes about 120 instructions to perform an integer division which is required in primes 100. For other similar programs which require multiplication and/or division of integer and/or floating point, low performance is also expected. But, because the management processor has its own FPU (floating point unit) in the IUs of PIE64, UNIREDII can pass such calculation to the MP and can concentrate on reducing goals. However, the evaluation has not been done yet.

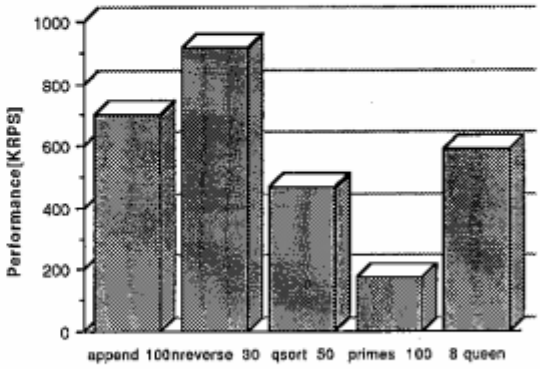

Figure 6: performance with the sample programs

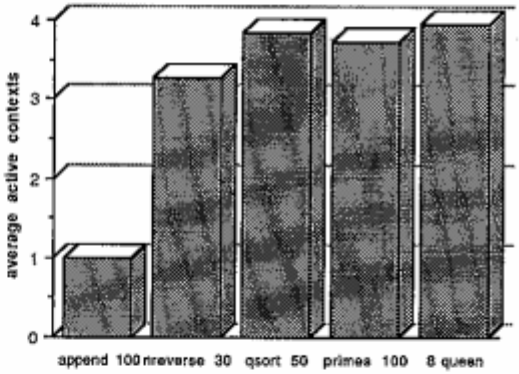

Figure 7: average number of active contexts in the sample programs

4.3 Tolerance of Remote Access Latency

To evaluate tolerance of remote memory access latency, we incorporated a pseudo-remote access mechanism in

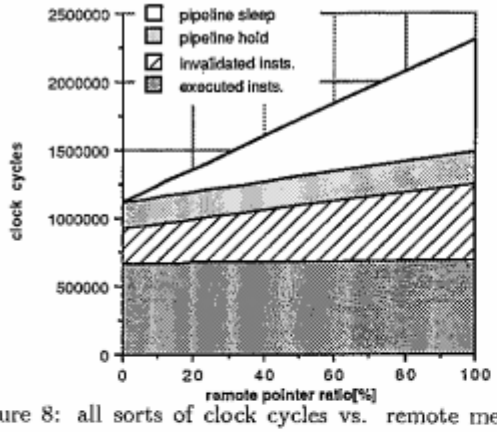

Figure 8: all sorts of clock cycles vs. remote memory access (8 queens, the maximum number of contexts = 1)

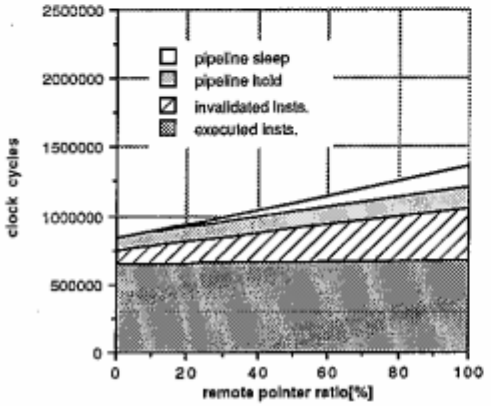

Figure 9: all sorts of clock cycles vs. remote memory access (8 queens, the maximum number of contexts = 2)

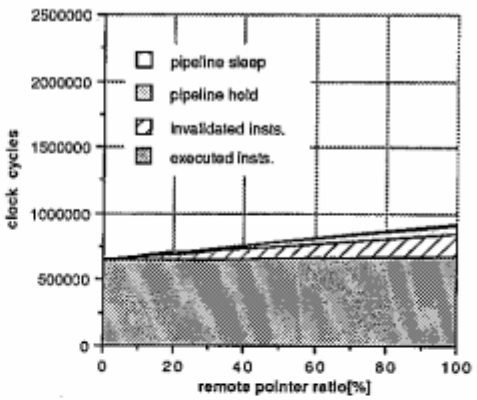

Figure 10: all sorts of clock cycles vs. remote memory access (8 queens, the maximum number of contexts = 4)

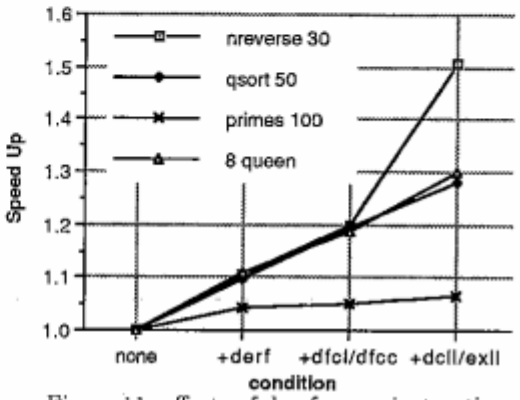

Figure 11: effects of dereference instructions

the simulator in spite of the single processor model of it as shown in figure 5. In more detail, we change the value of the IU-identifier field of the pointers included in every goal when reduction of the goal starts or resumes after suspension, with the probability which we call *remote pointer ratio*. Remote memory access commands issued by UNIREDII are emulated by the command processor shown in figure 5 with cycles listed in table 3. Under these conditions, we varied the maximum number of the contexts from one to four, and measured the clock cycles required by all sorts of the pipelined execution of instructions using the 8 queens program. Results are shown in figure 8 to 10. In these figures, the lowest part (shadowed) of the graph represents the number of executed instructions, the second part (hatched) represents the number of invalidated instructions by some jumps, the third part (lightly shadowed) the number of cycles while the internal pipeline of UNIREDII holds, and the fourth, uppermost part (white) the number of cycles while the pipeline are sleeping because, waiting for some replies, no contexts can be executed.

In figure 8, the multi-context processing mechanism of UNIREDII is not activated because the maximum number of the active contexts is set to one. Therefore the pipeline sleeping time (the white part of the graph) can not be hidden and becomes longer and longer as the remote memory access increases. Moreover, the pipeline hold time and the amount of invalidated instructions are great because the pipeline interlock occurs frequently.

In the other two figures (figure 9 and 10), the multi-context processing mechanism works and works more effectively as the number of contexts increase. The pipeline sleeping time is least in the figure 10 and the pipeline interlock (the pipeline hold and the instruction invalidation) hardly occurs in that figure. They become a little longer as the remote memory access increases because the average number of the active contexts decreases. Figure 9 shows an intermediate state between figure 8 and 10.

4.4 Effects of Dedicated Instructions

Finally, we present the effect of the dereference instructions, which are most characteristic of the instruction set of UNIREDII. Figure 11 shows the speed up about four sample programs (naive reverse 30, quick sort 50, primes 100, 8 queen) without the dereference instructions (the dereference instructions are resolved into more basic instructions), with only the basic dereference (derf)

instruction, with the dereference-and-check-list/constant (dfcl / dfcc) instruction, and with the all combined instructions such as the dereference-and-check-list-load-car / execute-on-list-load-car (dcll/exll) instruction, respectively.

In the figure, the speed up of the basic dereference instruction is about 10 % except in the primes 100 program, in which the majority of the executed instructions are arithmetic ones. In addition, the combined instructions have their effect as shown, and the total effect of these instructions is about 30% except primes 100. Therefore it can be said that the dereference instructions have a great effect.

5 Discussion

In the previous subsection, we present the effect of the dereference instructions and the combined ones. One point is that they are not such complicated instructions. In the hardware design, the instruction decoder does not include the critical path which actually determines the maximum clock rate of UNIREDII. The critical path is included in reading general purpose register file and ALU calculation. Moreover, all of the instructions of UNIREDII are single-cycle instructions because they jump to themselves recursively when they need more cycles to complete their action, as described before in section 3.4.1.

Owing to these dedicated instructions, we can compile Fleng programs so that the number of executed instructions are minimized. As the result, we can achieve high performance though the clock rate is comparatively slow, 10 MHz.

Finally, we shall mention the effect of the multi-context processing of UNIREDII. As well as reducing overhead of inter-processor synchronization, we can reduce pipeline interlock with it so that we can turn the pipeline of UNIREDII into an interlock-free one.

6 Conclusion

We have described the architecture of the inference processor UNIREDII, and evaluated some aspects of it. We got a performance of about 1 MRPS with 10MHz clock, and made certain that the multi-context processing of UNIREDII has a big effect on reducing pipeline interlocking and on reducing overhead of the remote memory access latency. In future, we will evaluate it by larger, real application programs. And, of course, we will make the real UNIREDII chip work as PIE64 system element.

Acknowledgements

We specially thank Prof. J.A.Robinson for much helpful advice. And we also thank the members of the group SIGIE in our laboratory, namely Tadashi Saito, Eiichi Takahashi, Minoru Yoshiada, Takeshi Shimizu, Yasuo Hidaka, Jun'ichi Tatemura, Hidemoto Nakada, Kei Yamamoto, Hajime Maeda, Shougo Shibauti, and Takashi Matsumoto. This work was supported by Grant-in-Aid for Specially Prompted Research (No.62065002), and is now supported by Grant-in-Aid for Encouragement of Young Scientists (No.03001269) of the Ministry of Education, Science and Culture.

References

[Jordan 1983] Jordan,H.F.: *Performance Measurements on HEP - A Pipelined MIMD Computer*, Proc. of the 10th Annual International Symposium on Computer Architecture, pp.207-212, ACM (1983)

[Halstead and Fujita 1988] Halstead,R. and Fujita,T.: *MASA:A Multithreaded Processor Architecture for Parallel Symbolic Computing*, Proc. of the 15th International Symposium of Computer Architecture, pp.443-451, IEEE (1988)

[Shimuzu *et al.* 1989] Shimizu,K., Goto,E., and Ichikawa,S.: *CPC (Cyclic Pipeline Computer) – An Architecture Suited for Josephson and Pipelined-Memory Machines*, Transactions on Computers, Vol.38, No.6, pp.825-832, IEEE (1989)

[Kimura and Chikayama 1987] Kimura,Y. and Chikayama,T.: *An Abstract KL1 Machine and Its Instruction Set* Proc. of the 1987 Symposium on Logic Programming, pp.468-477 (1987)

[Nilsson and Tanaka 1988] Nilsson,M. and Tanaka,H.: *Massively Parallel Implementation of Flat GHC on the Connection Machine*, Proc. of Fifth Generation Computer Systems 1988, pp.1031-1040, ICOT (1988)

[Koike and Tanaka 1988] Koike,H. and Tanaka,H.: *Multi-Context Processing and Data Balancing Mechanism of the Parallel Inference Machine PIE64*, Proc. of Fifth Generation Computer Systems 1988, pp.970-977, ICOT (1988)

[Takahashi *et al.* 1991] Takahashi,E., Shimizu,T., Koike,H., and Tanaka,H.: *A Study of a High Bandwidth and Low latency Interconnection Network in PIE64*, Proc. of Pacific Rim Conference on Communications, Computers and Signal Processing, pp.5-8, IEEE (1991)

[Shimizu *et al.* 1991] Shimizu,T.,Koike,H.,and Tanaka,H.: *Details of the Network Interface Processor for PIE64*, (in Japanese) SIG Reports on Computer Architecture, 87-5, IPSJ (1991)