(2)

Vectorizing and Parallelizing Compiler Techniques for Block-Structured Languages

UEHARA Tetsutaro

November 1995

Abstract

With the recent progress in technology of semi-conductor devices and architectures, the performance of vector and parallel *supercomputers* have been remarkably advanced. Today they have become a fundamental tool of science which is widely used in various fields of science and engineering. Supercomputers manufacturers are providing automatic vectorizing and parallelizing compilers (or *supercompilers*) which generate suitable object codes from programs written in sequential languages. Since most of these compilers are for FORTRAN77 which has been commonly used for numerical applications, supercompilers for languages other than FORTRAN are indispensable to utilize supercomputers for non-numerical applications. Moreover, various features which have been considered to be specific to block-structured languages, such as pointers and recursive calls, are introduced in the new standard specification of FORTRAN called Fortran90. This thesis discusses various vectorizing and parallelizing compiler techniques which are effective not only for block-structured languages but also for Fortran90.

In Chapter 2, the performance of load/store instructions on each VP/PVP supercomputer is evaluated. In particular, the performance of vector indirect load/store instructions is precisely discussed since the performance of these operations are important not only for numerical applications but for non-numerical applications which treat pointers. We have developed a benchmarking program to evaluate this performance and show the obtained results.

In Chapter 3, a new method to vectorize and parallelize recursive procedures is proposed. Since recursive procedures are often seen in programs written in block-structured languages, this control should be also regarded as the targets for automatic vectorization and parallelization. We propose the breadth-first method and show that some recursive procedures will be greatly accelerated by this method. In particular, this method is effective for the recursive procedures which are the implementations of divide-and-conquer algorithms because this method utilizes the parallelism between recursive invocations of the procedures. We also discuss about the analysis to determine whether the target procedure can be vectorized/parallelized by this method or not. Since there is a difficulty for the determination when the procedure

refers global array variables in general, we propose a method which has a restriction but still effective in practical programs.

In Chapter 4, the overview of our automatic vectorizing and parallelizing compiler V-Pascal version 3 is presented. On the compiler, various analyzing, vectorizing and parallelizing techniques have been developed even for Algol-like features, such as alias analysis of pointers, vectorization/parallelization of while/repeat-until loops and recursive procedures. These techniques may also be useful for the full-fledged vectorizing/parallelizing compilers for the languages Fortran 90 and C.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Backgrounds

1.1.1 Supercomputers

With the recent progress in technology of semi-conductor devices and architectures, the peak performance of *supercomputers* have reached to tens or several hundreds GFLOPS (Giga Floating-point Operations Per Second). Their high performance enabled us to perform many billions of operations which are needed to solve various problems in fields of science and engineering, such as biology, energy research, high energy physics, astrophysics, aerospace engineering, seismology, petroleum exploration, image processing, pattern matching, VLSI design, tomography, scientific visualization and so on. Now supercomputers have become a fundamental tool of science.[54]

Architectures of today's supercomputers can be divided into three categories : *vector processor (VP), parallel vector processors (PVP)* and *massively parallel processors (MPP)*.

The main feature of VP machines is that they each have *vector-processing units* aside of the ordinal scalar processor. These units can perform vector operations which perform the same operations for each element of vectors (i.e. 1-dimensional arrays). Since these operations are performed by pipelined arithmetic units with quite high speed, the peak performance of a VP machine reaches up to several GFLOPS. A typical PVP machine is configured as a symmetric multi-processor system of several VPs which share the same main memory.[1] Today's high-end PVPs, such as Cray T90 and NEC SX-4, can be configured with up to 32 processors and their total peak performance is more than 50 GFLOPS. These machines have been widely used for computational science and engineering. In other words, these are conventional supercomputers. Table 1.1 shows a list of today's conventional supercomputers.

But these VP/PVP machines are generally quite expensive. The main reasons are that they needs advanced technologies to achieve quite short clock-cycle of their devices, and that they need contrived memory units to obtain high load-store bandwidth to supply enough data to the high-speed arithmetic units. To achieve the comparable peak performance with reduced costs, MPP is proposed as a loosely-coupled multi-processor of RISC microprocessors. Recently these processors have remarkably progressed and achieved the performance of tens or hundreds MFLOPS (Mega Floating-point Operations Per Second). In a typical MPP machine, hundreds processors are coupled with a high-speed inter-connection network, therefore they can achieve the peak performance of several GFLOPS. These machines are still hard to utilize for general purpose, but considered to be effective for some specific problems which can be highly parallelized and need less data-transfer between the parallelized program sections.

1.1.2 Supercompilers for Supercomputers

The high performance of VP/PVP machines derives from their multiple vector pipelines with a very short pipeline period. These pipelines manage the vector operations which are 10 or 100 times faster than that by scalar processors. Since vector operations belong to parallel operations which are not familiar for ordinal users, it is very important to provide peculiar compilers which generate vector instructions automatically from programs written in conventional sequential programming languages. Now most of the manufacturers provide automatic vectorizing compilers for FORTRAN, which is commonly used in scientific

[1]Some newly announced vector-parallel machines (such as Fujitsu VPP series and NEC SX-4 multi-node models) are configured with distributed main memory, but we don't include them in our definition of PVP.

Table 1.1: Today's VP/PVP supercomputers

Vendor	Model	Maximum number of processors	Total peak performance (GFLOPS)
Cray Research[8]	T90	32	>60
Cray Research[8]	C90	16	16
Convex [5]	C4/XA	4	6.4†
Hitachi [25]	S-3800	4	32
Fujitsu	VP-2600E	1	5.5
NEC [27]	SX-4	32‡	64

† With single precision.
‡ For *single-node* (i.e. shared-memory) system.

Table 1.2: Today's MPP supercomputers

Vendor	Model	Processor element	Max. num. of nodes	Total peak performance (GFLOPS)
Cray Research [8]	T3D	DEC 21064(150MHz)	2048	300
Thinking Machine [41]	CM-5E	SuperSparc(40MHz)	1024	80
Convex [5]	SPP-1200/XA	HP PA-7200	128	30.7
Intel [17]	Paragon	i860XP(50MHz)	n/a†	0.075×nodes
IBM [16]	SP-2	POWER2‡	n/a*	0.267×nodes
NEC	Cenju-3	R4400SC	256	12.8
Ncube [35]	Ncube3	Original	10240	>1000
Hitachi	SR2001	Original	128	23

† The largest one installed at Sandia National Lab.(US) has 1840 nodes.
‡ PowerPC based nodes are also available.
* Announced as up to 128, but 512-nodes systems are also available.

and technological fields. Today's vectorizing compilers have high ability to find out vectorizable loops within programs and we can obtain quite high ratio of speedup. The techniques for automatic vectorization can be easily extended and applied to parallelization of loops and PVP machines can execute these loops effectively, therefore automatic vectorizing and parallelizing FORTRAN compiles for PVP machines are also provided. These compilers are called *supercompilers* and they have much contributed to utilize VP/PVP supercomputers in various fields of science. Thus VP/PVP supercomputers have been successful for scientific numerical computations.

On the other hand, fully-automatic parallelizing compilers for MPP machines are still hard to realize. While the techniques to recognize parallelizable parts of the target program for VP/PVP machines are also applicable for compilers for MPP, it is hard the compilers to decide the optimal distribution of data among processing nodes to reduce the amount of data-transfer. Therefore the current compilers provided by manufacturers are for parallel languages or conventional sequential languages with extension to be directed the data-distribution by users.

1.1.3 The Goal of This Research

For any type of supercomputer architectures, it is fairly important to provide supercompilers which assist users to enjoy the high performance of supercomputers. Most of current supercompilers provided by manufactures are for FORTRAN77 which is popular for numerical computation. However, FORTRAN77 is not effective for general purposes because it has rather poor control structures and data types. Therefore, it is indispensable to develop automatic vectorizing compilers for other programming languages to utilize supercomputers more efficiently. This is the motivation of this research. To extend the horizon of vector supercomputer usage, we were interested in developing automatic vectorizing/parallelizing compilers for block-structured languages, i.e. descendants of Algol60, which is equipped with more versatile control/data structures than FORTRAN77.

Toward the goal, we choose Pascal as the target language of our compiler. Pascal is a typical block-structured language which is widely spread for general purpose. Pascal has been used not only for educational purposes in classrooms but also for real-world problems, for example, the early TEXprocessor by D.E. Knuth, the vectorizing Fortran compiler construction by an American supercomputer manufacturer, and the implementation of an operating system for some Japanese minicomputers. Therefore the vectorization and parallelization techniques obtained through this research must be effective to execute various programs effectively in vector processors and parallel processors. The techniques will be also applicable to the other languages including Fortran90 which is the newest version of FORTRAN and is equipped with various control/data structures which have been considered to be specific to block-structured languages.

1.1.4 Outline of this thesis

This thesis discusses a way to construct an automatic vectorizing and parallelizing compiler for a block-structured language. The following three topics are included in this thesis.

In Chapter 2, the performance of vector load/store of vector supercomputers is evaluated with our newly developed benchmarking program. In particular, vector indirect load/store performance is precisely examined. One of the reason why we focused on the indirect addressing is that these instructions must be often used in non-numerical applications which contain data-structures constructed with pointers. Therefore these results are indispensable to construct a compiler for a block-structured language and to obtain optimal performance on VP/PVP machines in executing non-numerical applications. The other reason is that the vector-indirect addressing is also important for some numerical applications. The details will be described in the chapter.

In Chapter 3, we propose a new method to execute recursive procedures (and functions) on supercomputers. Generally, the targets for automatic vectorization and parallelization compilers are limited to `DO` loops of FORTRAN77. But block-structured languages have more versatile control structures to express iterations, such as **while** loops and **repeat/until** of Pascal. Moreover, recursive procedures can be used in these languages. Needless to say, these iterative control structures should be also vectorized/parallelized if possible. But there are not many researches to discuss this issue. In particular, there are few researches to treat recursive procedures. We propose a method to cope with the difficulty of automatic vectorization and parallelization of recursive procedures. The methods named *"breadth-first"* method is to utilize parallelism among two or more recursive calls and to execute them in parallel. Some procedures are accelerated more than 50 times by this method.

Chapter 4 describes the organization and the features of our automatic vectorizing/parallelizing Pascal compiler named **V-Pascal** Version 3. V-Pascal Ver.3 is designed as an automatic vectorizing and parallelizing compiler for PVP machines. The target language is a sequential block-structured language Pascal. Many techniques and methods we have implemented will be described later.

The last chapter, Chapter 5 is the conclusion of this thesis with some open questions and future works.

Chapter 2

Benchmarking Vector Indirect Load/Store Instructions

2.1 Introduction

In some fields of large-scaled numerical computations, the data should be expressed as huge random sparse matrices. We can enumerate some important examples such as Finite Element Method(FEM), Linear Programming(LP), Electric Circuit Simulation, Neural Network Simulation and so on. In processing these computations on VP/PVP machines, vector indirect load/store instructions are useful for both memory-space efficiency and the execution speed. In other words, the performance of the indirect vector accesses affects the overall performance of these computations.

We also believe that indirect vector accesses are effective for advanced automatic vectorization. For example, in some cases nested loops can be converted into non-nested loops by using list vectors, and allow longer vector lengths. Supercomputers with high performance tend to have large $\boldsymbol{N}_{half}$[15] values, so this conversion is useful for those machines that have good performance on vector indirect accesses. We have already implemented it in the early version of "V-Pascal" and have shown its usefulness[43].

Furthermore, to extend the applications of supercomputers beyond numerical computations, compilers or languages must support the data represented by pointers e.g, linked lists, trees etc. To vectorize applications, indirect vector load/store instructions are essential. These types of automatic vectorization must be supported on V-Pascal version 3 and then indirect vector load/store instructions will become more important in the field of vectorization.

Therefore, we have been investigating the performance of vector indirect access on vector supercomputers but enough data has not been accumulated to date. Some existing benchmarks showed the performance of applications where indirect vector load/store instructions were used but we wanted to know the "pure" performance of these instructions, because the precise information about these instructions are very important in code-generation of compilers. This chapter describes a newly developed benchmarking program for this purpose and obtained some interesting results.

2.2 Factors Dominating the Performance of Indirect Vector Access

This section describes the factors dominating the performance of indirect vector load/store instructions. Generally an indirect vector access is slower than a direct vector access for the same amount of data. There are two main reasons for this.

1. Complications in the behavior of load/store pipelines as a result of indirect accesses.
2. Effect of bank conflicts.

The following sections describes these two factors.

2.2.1 Behavior of Indirect Vector Operations

This section discusses the behavior of indirect vector access and compare it with that of direct vector access.

Let us start with the direct operations. A sample of a vector direct load instruction is written in assembly code as follows.

```
vload BASE, STRIDE, vr1
```

In this statement `vr1` means a vector register and `BASE` means an address. `STRIDE` expresses the distance between each data elements on the memory. When expressing the *i-th* element of `vr1` as `vr1(`i`)`, the semantics of the instruction are presented as follows.

> for each $i \in \{1,2,...,$<*vector length*>$\}$,
> let `vr1(`i`)` to the content of the address
> `BASE+STRIDE`×(*i-1*)×<*size of each element*>

Note that the 3rd line shows the effective address for each element.

Turning to indirect vector access a sample of an indirect vector load instruction is as follows.

```
viload BASE, vr1, vr2
```

The semantics of this instruction are as follows.

> **for each** $i \in$ **{1,2,...,**<*vector length*>**},**
> **let** `vr2(`i`)` **to the content of the address**
> `BASE+vr1(`i`)`×<*size of each element*>

As shown above, the semantics are similar. The difference is seen only in the calculation of effective addresses. In fact, complications in the behavior of indirect vector access comes from this difference. We considered the detailed complications as follows.

1. An indirect vector operation must treat two vector registers and the set-up sequences may be more complicated.
2. Another data path for list vectors is needed between vector registers and each load/store pipeline, but an implementation without this path is also possible, at the cost of throughput.

3. In a direct vector operation, effective addresses of the array elements are estimated when the vector unit is set up. But in indirect vector operations, they are known only at run-time. This means that it is impossible to foresee the occurrences of bank conflicts or memory-protection exceptions. Moreover, when some elements in the list vector are of the same value, the stores must be serialized to guarantee the sequential order of writes[1]. Therefore complicated pipeline controls are needed lowering the throughput of the pipeline.

Note that the complications are separated into two types. Some of them affect the instruction set-up time and others affect the throughput. In the above list, 1 belongs to the former, 2 belong to the latter and 3 may belong to the both.

2.2.2 Bank Conflicts

Bank interleaving is one of commonly used techniques to utilize slow memory chips in high-performance computer systems. As supercomputer systems are of quite high-performance, even faster S-RAM chips are sometimes regarded as slow. Therefore most VP/PVP supercomputers have memory units interleaved with many banks. Another common technique is caching but it will not work well in supercomputers which tend to have large working sets in memory. Accordingly caching is rarely used in VP/PVP machines.

A simple model of bank interleaving is shown in Figure 2.1. On a memory chip, there is a delay between giving an address and accessing the location. It is called **Access Time**(and represented as T_a). Also it takes a time (called **Recovery Time** represented as T_r) from the finish of one access and before the next one is possible. We call $T_a + T_r$ the **Continuous Access Time**[2] and represent it as T_c. Interleaving with *N*-banks seemingly decreases T_c/N if an access is for *stride-1*. But if two accesses to the same memory occur with an interval shorter than T_c, the latter access will be delayed until T_c passes. This is called a

[1]The S-820 and S-3800 have another indirect store instruction which does not guarantee the order. See Section 2.4.2.

[2]Actually this value differs between read and write accesses.

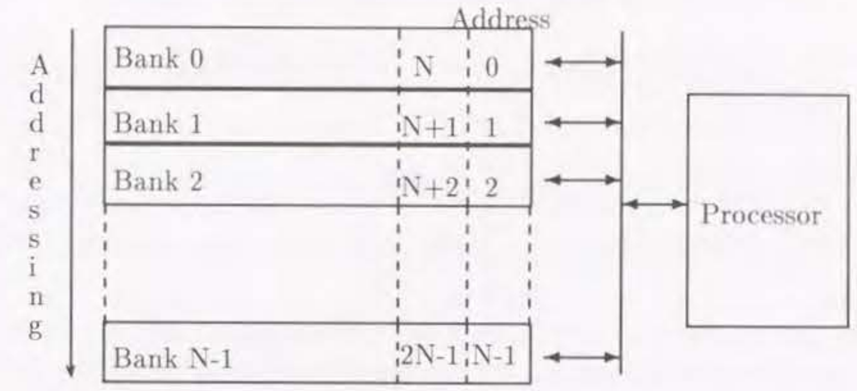


Figure 2.1: Bank Interleaving

Bank Conflict. Obviously, in indirect vector accesses, bank conflicts occur more frequently than in direct vector accesses and their frequency dominates the performance of indirect vector accesses.

In the simple model (shown in Figure 2.1), expressing the pipeline period of the load/store pipeline as T_p, the following equation characterizes the frequency of the occurrence of bank conflicts.

$$s = T_p \times N/T_c$$

s means that this memory unit allows accesses for *stride-s* without any bank conflicts. The greater s becomes, the less is the frequency of conflict.

Actually as most memory architectures are not simple as this. As described in Section 2.4.2, some machines have much more complicated memory units, in particular, in multi-processor systems. But s is still useful as a criterion of performance. N and T_c still remain as the dominating factors of the performance of indirect vector accesses.

2.2.3 Performance Factors and their Compound Effects

As mentioned in the former sections, the performance of indirect accesses is affected by the following factors.

1. Set-up times for indirect vector load/store instructions.
2. Throughput of each load/store pipeline in indirect vector operation.
3. The number of load/store pipelines and of processors.
4. The frequency of bank conflicts and their penalties.

These factors should be examined in benchmarking. We also have to note the compound effects of these factors. In some systems, the penalties of bank conflicts in indirect accesses seem to be worse than that in direct accesses. It is considered to be caused from the difficulty of pipeline control under irregular occurrences of bank conflicts.

2.3 The Benchmarking Program

This section describes our new benchmarking program for indirect vector load/store instructions.

2.3.1 Our Main Goal

Our main goal is to evaluate the performance of vector indirect addressing on each machine. The most specific feature of our benchmark is to evaluate both the throughput of indirect addressing and the effect of bank conflicts. We insist that it is quite important to evaluate them separately while existing older benchmarks try to evaluate them at once.

Assume that a machine show enough performance of the throughput but it is seriously decelerated by bank conflicts. Such machine may show poor performance in older benchmarks, but this machine is still available when an application with vector indirect accesses is carefully programmed to avoid bank conflicts. It is also possible for a compiler to generate efficient objects for advanced vectorization with indirect accesses described in Section 2.1 if the compiler has an accurate information of the penalties of bank conflicts on the machine.

2.3.2 Benchmarking Program

To take an accurate measurement assembly coding is ideal, but it is hard to program each benchmark in assembler, because the architectures of some machines are proprietary. Therefore our benchmarking program is written in FORTRAN77 and it causes some problems. The main problem of our program is that it cannot evaluate performance of vector direct/indirect load instructions exactly. The reason is as follows.

1. We cannot generate vector load instructions without generating vector store instructions. For example, a FORTRAN statement `A(I)=B(I)` includes both of vector load and store instuctions and we cannot get rid of the store. On the other hand, we can easily generate a vector store insruction alone by a Fortran statement `A(I)=1.0D0`.
2. "<Execution time of `A(I) = B(I)`> − <execution time of `A(I) = 1.0D0`>", does not always mean the execution time of the vector load instruction because of the chained behaviour of vector pipeline units in the target machine.

The details of these reasons are described in [48].

Our program is carefully designed to avoid unexpected effects caused by optimization of compilers. The main technique to avoid them is to use some variables of which value cannot be assumed by compilers. Our program is also designed to take execution times of each benchmarking item twice and reports the latter value because instruction/data cache of scalar units may affect the execution time. The details of our benchmarking techniques are also described in [48].

The items examined in our benchmarking are as follows. (In the following, `A` and `B` are arrays of `REAL*8`[3]. `L` is an array of `INTEGER*4`.)

- The execution times of

  ```
  A(I) = B(I)
  ```

  with vector lengths $2^4, 2^6, 2^8, ..., 2^{18}$ are used to estimate the throughput of load/store pipelines on vector direct load/store operations,

[3]8bytes/element

and are also used for comparison with the results of indirect load/store operations.

- The execution time of

  ```
  A(I) = B(I)
  A(I) = 1.0D0
  ```

  at vector lengths 2^{18} are used to estimate the throughput on direct accesses.

- The execution time of

  ```
  A(I) = B(I*STRIDE)
  A(I*STRIDE) = 1.0D0
  ```

  for strides $1, 2, 3, ..., 512$ and vector lengths $2^4, 2^6, 2^8$ are used to estimate the time T_c of the memory unit.

- The execution time of

  ```
  A(I) = B(L(I))
  A(L(I)) = 1.0D0
  ```

  for each `L` as

  ```
  L(I) = ((I*STRIDE).MOD.(2**18))+1
  ```

  for strides $1, 2, 3, ..., 512$ and vector lengths $2^4, 2^6, 2^8, ..., 2^{18}$ are used to estimate the throughput on vector indirect accesses when bank conflicts occur periodically.

- The execution time of

  ```
  A(I) = B(L(I))
  A(L(I)) = 1.0D0
  ```

  with `L` initialized as shown below. (In these expressions, `N` means the vector length.)

  - `L1(I) = I`.
  - `L2(I) = N-I+1`.
  - `{L3(I)} = {`*random permutation of* $1, 2, ..., N$ `}`.
  - `L4(I) = <`*random number from* $[1, N]$`>`.
  - `L5(I) = 1`.
  - `LS(I) = I.MOD.X, X`$\in \{2, 3, 4, ..., 10\}$

  for vector lengths $2^4, 2^6, 2^8, ..., 2^{18}$. This includes the influence of bank conflicts that occur when addressing is either regular or random.

If a VP has 2 load pipes and 1 store pipe, it becomes easy to evaluate the throughputs of indirect load and store instructions. The statement `A(I)=B(I)` needs only 1 load pipe and 1 store pipe for direct load and store instructions while the statement `A(I)=B(L(I))` needs 2 load pipes and 1 store pipe, and for both of these statements all of the pipelines are organized into a chain and can be work simultaneously. Therefore, if the throughput of the load pipe for both of indirect and direct accesses are the same and no bank conflict occurs, the execution times of these statements can be assumed to be almost the same. Similarly, if the throughputs of indirect and direct store are the same, the execution time of the statements `A(I)=1.0D0` and `A(L(I))=1.0D0` can be almost the same. If the throughput of load or store pipes for indirect accesses is much lower than that for direct access, the pipe for the indirect access becomes the bottleneck of the whole operation and the execution time shows the throughput for indirect accesses.

Note that `A`, `B` and `L` are allocated continuously in the memory. It means that `A(I)`, `B(I)`, `L(2*I-1)` and `L(2*I)` of a specific value of `I` are all on the same bank. This allocation may be also taken into consideration when we evaluate the result in detail.

2.4 Experimental Results

2.4.1 Performance parameters

We show the following performance parameters as a summary from the results of our benchmarking program.

Firstly, as the parameter of throughput of load/store pipelines, we show the following values.

- Data transfer rate with the following statements
  - `A(I)=B(I)`
  - `A(I)=1.0D0`

  at each vector length of $2^8, 2^{18}$.

- Data transfer rate with the following statements
  - `A(I)=B(L(I))`
  - `A(L(I))=1.0D0`

  for `L(I)=I` at vector length of 2^{18}.

Secondly, we show the following values to estimate the averaged throughput of load/store pipelines under the consideration to the effects of bank conflicts.

- Averaged data transfer rate with the following statements

```
A(I) = B(I*S)
A(I*S) = 1.0D0
```

  at `S` $=1,2,3,...,512$ and vector length 2^8.

- Averaged data transfer rate with the following statements

```
A(I) = B(L(I))
A(L(I)) = 1.0D0
```

  for strides $1,2,3,...,512$ and vector length 2^{18}.

Thirdly, to evaluate the worst case of the data transfer rate caused by bank conflicts, we show the following values. These values imply the case where all of the memory accesses are to the same bank.

- Data transfer rates with the following statements

```
A(I) = B(I*S)
A(I*S) = 1.0D0
```

  at `S` =512 and vector length 2^8.

- Data transfer rates with the following statements

```
A(I) = B(L(I))
A(L(I)) = 1.0D0
```

  with `L(I)=512*(I-1)` and vector length 2^{18}.

Lastly, we show the following strides to evaluate the performance of memory systems. The smaller these values are, The more serious are the influence of bank conflicts.

- The minimum number of stride which causes the performance which is less than the half performance of the maximum with the following statements.

```
A(I) = B(I*S)
A(I*S) = 1.0D0
```

  (We call this number S_{half} in this thesis.)

- The minimum number of stride which causes as the same performance as that of the worst case with the following statements.

```
A(I) = B(I*S)
A(I*S) = 1.0D0
```

  (We call it S_{worst} in this thesis.)

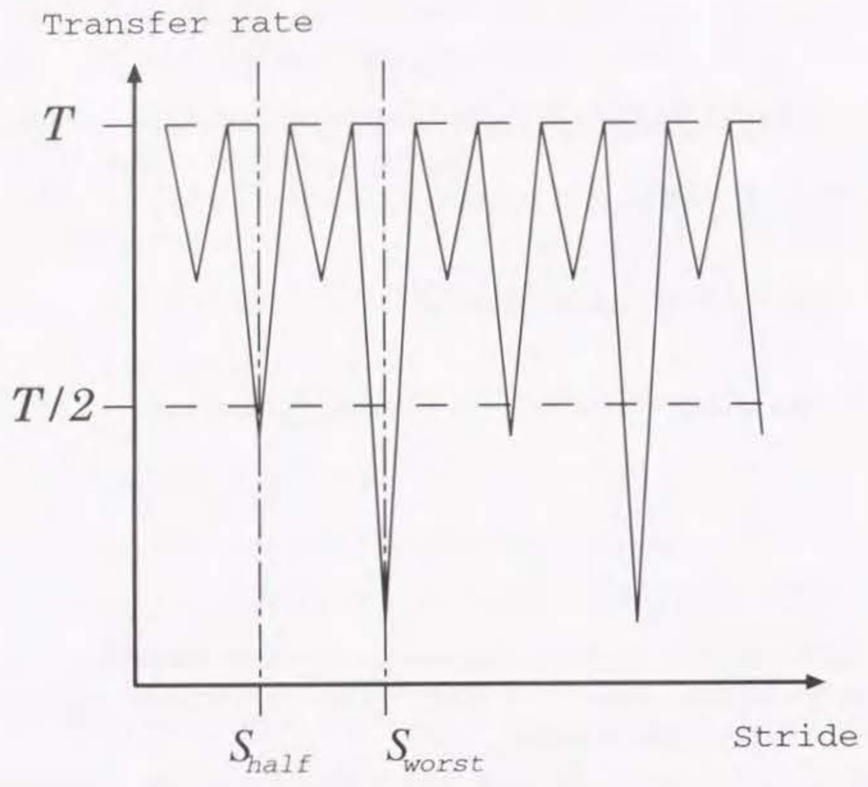


Figure 2.2: A sample of S_{half} and S_{worst}.

See Figure 2.2 as an example of S_{half} and S_{worst}.

The results are shown in Table 2.2 at the end of this report. To express the performance, we use a unit MDwords/sec[4]. This unit is useful for comparison to MegaFLOPS[5] but note that it is not always equal to the transfer rate between memory and the processor.

We also show some additional values which can be got from the above values labeled (U),(V),(W) and (X) in the Table 2.2. These values are calculated by the expressions written in the first column of the table.

The value of (U),(V) implies the relative throughput of load/store

[4]Mega Doublewords per second. Doubleword means the size of REAL*8

[5]Mega FLoating-OPerations per Second

pipelines in indirect accesses compared to that in direct accesses.

(W) and (X) are helpful to evaluate the robustness to bank conflicts on each machine, but we should note that (W) is measured by the vector length of 2^8 which is too short for some machines to evaluate the throughput. We should also note that (X) is measured by indirect accesses and it may cause some compounded effects of bank conflicts and indirect accesses.

The results in Table 2.2 are also shown in Figure 2.4, Figure 2.4 and Figure 2.6.

2.4.2 Target Machines and the Results

Table 2.1 shows the performance and hardware facilities of each target machine.

The features and some notices for each machine are described below with the results.

Hitachi S-820/80

The S-820/80[6] is an old type machine announced in 1987. Now this series of machines are obsolete and replaced by the S-3000 series.

One of the specific features of this machine is a support of a special instruction of indirect vector store, called **VIST**[13]. As this instruction does not preserve the sequential order of the store, the elements of the list vector must be differ from each other, but its throughput is higher than that of the normal indirect store instruction called **VSTX**[13]. We have added the results using VIST in Table 2.2 in parentheses. According to the results labeled (F) and (J) in Table 2.2, VIST instruction shows about 80% better performance than VSTX instruction shows. VIST is not only used through user directives but also used by our V-Pascal compiler to utilize the instruction.

We also have to note that (B) does not show the maximum data transfer rate in direct accesses. The transfer rate for stride-2 access shows 563 MDwords/sec and is about 20% better than that for stride-1. The performance for any other odd number of stride is no better

[6]The S-820/80 was settled at Computer Centre, The University of Tokyo. This machine has been replaced by the S-3800/480 in Feb. 1993.

than 350 MDwords/sec. It is not obvious why this machine shows better performance of the direct store only for stride-1 and stride-2. It may have to do with the complicated bank interleaving in the memory unit of the S-820.

Since the S-820 was equipped with only two pipes for memory access, the statement `A(I)=B(L(I))` cannot be processed in a chain of pipelines. Taking the pipeline organization of the S-820 in consideration, it is assumed that `A(I)=B(L(I))` will be executed as follows.

1. The processor loads `L` and `B` with chaining.
2. Then it stores `A` while it loads next `L`.
3. Lastly it loads `B` and stores `A` with chaining.

Note that 2∗ <*vector length of strip mining*> elements of data are processed by these 3 stages which takes the same time with `A(I)=B(I)` to transfer 3∗ <*vector length of strip mining* >. Therefore the throughput of `A(I)=B(L(I))` is at most 66% of that of `A(I)=B(I)` even if the throughput of the indirect load is the same as that of the direct load. According to the value of (U), the performance parameter for indirect load accesses (E) is 64% of that for direct load access (C). Thus it is assumed that the throughput of indirect loads on the S-820 is as the same as that of direct loads. The throughput of indirect stores is not so good as that of direct loads even if VIST instructions are used, but the absolute performance is still kept to high.

As the values of direct load (A) and direct store (B) is almost the half of indirect load (C) and indirect store (D), 2^8 is not considered to be enough for the vector length to evaluate the effects of bank conflicts and the value of (W) is not reliable as the parameter of robustness to the bank conflicts. Taking (X) as the parameter, the S-820 is consider to be robust to the bank conflicts in indirect addressing and will show good performance even for randomized list vectors. The values of (X) for the SX series seem to be better than that of the S-820, but as described later, these values are not reliable because the absolute performances of indirect accesses on the SX-2N and the SX-3R are worse.

Fujitsu VP-2600/10

One of the specific features of the VP-2600[7] is that it is equipped with 2 load/store pipes[46]. It means that the VP-2600 can execute `A(I)=1.0D0` twice faster than `A(I)=B(I)` if both of the pipes work in parallel. But it is inappropriate to evaluate the throughput of indirect store instruction because these pipes are used separately for load and store to execute `A(L(I))=1.0D0`. The compiler did not unroll the loops in our program because we had hided the vector lengths of the loops to avoid unexpected optimizations. Therefore the values labeled (B),(D) in Table 2.2 reflect the throughput of one pipeline.

As the value of (U) is a lower one compared with the other machines, the VP-2600 is considered to be of a rather poor performance of the indirect load. But contrarily, the indirect store (V) shows remarkable performance. Note that VP-2600 has no special instructions to keep the order. The values of the Y-MPs with 1 CPU are better than that of VP-2600, but the Y-MPs are too slow to compare with. The value of (V) of the Y-MP with 4 CPUs also seems better, but like VIST of the S-820 and the S-3800, the sequential order of the stores in elements of the list vector for an indirect store on Y-MP cannot be preserved. On the VP-2600, both of the absolute and relative performance of indirect stores, which preserves the order of stores, are remarkably high.

It is conspicuous that the values of diréct accesses (A), (B) are much lower than those of indirect accesses (C),(D). Therefore (W) is not considered to be suitable for the parameter of the performance for the indirect accesses for the same reason with the S-820. The alternative value (X) is neither so high nor so low when compared with the other target machines.

Note that the minimum data transfer rates of the store instructions are lower than those of the load instructions. It may be caused from the difference of the continuous access times between read and write to the memory. It should be also noticed that the absolute performance of the minimum data transfer rates are lower than those of the most of the other machines.

[7]Data Processing Center, Kyoto University.

NEC SX-2N

The SX-2N[8] is a dated machine but we show the results from the SX-2N for comparison with the SX-3R series. Since the vector unit of the SX-2 series is equipped with only one load/store pipe, it does not match the model to evaluate the benchmarking results as described in Section 2.3.

As shown in Table 2.2, the SX-2N showed quite poor performance for vector indirect accesses. As our main interest is the improvement of the performance of the SX-3R, we only summarize the result in short. The SX-2N shows only quite poor performance for both of the indirect loads and stores. It is interesting that even in a direct store, the performance is rapidly degraded by bank conflicts (see (V) in Table 2.2).

Cray Y-MP 8/4128 and Y-MP 2E/264

The memory organization of Cray Y-MP series has complicated structure (except Cray Y-MP EL) and does not match with our simple model in Figure 2.1. However, in the case of Y-MPs, the memory organization and detailed behaviors of the memory unit are fully described in manuals [7]. Figure 2.3 shows the block diagram of the memory unit of our target machines, Y-MP 8/4128[9] and Y-MP 2E/264[10]. The memory systems of these two machines are organized as the same except the capacity. The memory is divided into 4 sections, each section consists of 8 sections and each section contains 8 banks. The basis for memory performance for a CPU is the number of section because each CPU can access continuously to a section only in every 5 clock periods. Different CPUs can access to the same section simultaneously but each bank can be accessed only in every 5 clock periods. Therefore, the worst performance of memory access is the same as 1/5 of the peak performance.

On the benchmark with 4 processors, we allowed the program to do indirect vector store in parallel. This access may break the sequential

[8]This machine was settled at Computation Center, Osaka University. This machine has been replaced by the SX-3R/14 in Feb. 1993.

[9]Institute of Fluid Science, Tohoku University.

[10]Supercomputer Laboratory, Institute for Chemical Research, Kyoto University.

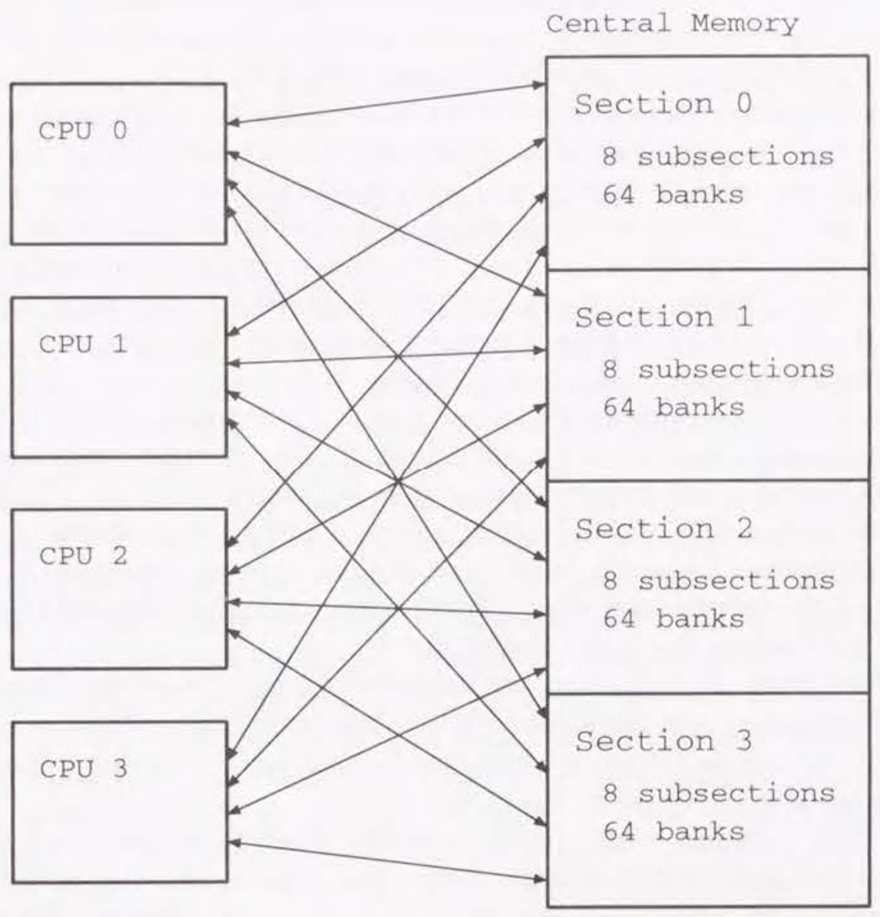


Figure 2.3: Y-MP Memory Organization[7]

order of writes, therefore the result should be treated as the same as that of VIST on the S-820 and the S-3800. When the order of stores must be preserved, the performance of the indirect store will be degraded as the same as that of 1 CPU.

Since the Y-MPs are equipped with 2 load pipes and 1 store pipe on each CPU, it is easy to evaluate our benchmarking results as described in Section 2.3. One of the most remarkable results is that the values of (C),(D),(E) and (F), on Y-MPs with both 1 CPU and 4 CPUs, indicate that the throughput of the load and store pipelines seem to be always constant on both of the direct and indirect accesses. The values of (U) also substantiate it. The values of (E) are a little worse than those of the others, but the performance of `A(I)=B(L(I))` relative to `A(I)=B(I)` on the Y-MPs are still quite good compared with the other machines. However, in the case of the results from Y-MP with 1 CPU, as the absolute performance is much lower than the other machines, the results are not so surprising. Another remarkable result is that the obtained results with 4 CPUs show that the performance of the Y-MP is quite good unless bank conflict occurs. The absolute performance of (C),(D),(E) and (F) are comparable to that of the S-820. As the peak performance of the Y-MP 8/4128 is 2 GFlops, the balance of the performance of memory access and that of the computation is much better than that of the S-820. This balance of the Y-MPs seems to be the best among the target machines.

Although the maximum performances of indirect accesses of the Y-MP are quite good, the value of (W) indicates that the performances of indirect loads and stores are degraded by bank conflicts rather seriously compared with the other machines.

Note that the Y-MP 2E/264 and the Y-MP 8/4128 are equipped with the same memory system except their capacity and their architectures are the same. Therefore the results are almost the same. But the results show that the newer model 2E has been a little improved in the performance of direct stores.

S-3800/480

The S-3800 is a successor of the S-820 and has similar features with the S-820. Each vector processor of the S-3800 is also equipped with 1 load/store and 1 load pipe. VIST instruction is also supported. Therefore the tendency of the results is similar to that of the S-820. The main difference is that the S-3800 is a multi-processor system while the S-820 is a single-processor system. The S-3800 on which we executed our benchmarking program is a system with 4 CPUs, but we had no chance to obtain the results with multi-processors.

We should comment that there was no unusual result of the performance as seen in the S-820. The performance for stride-1 is always the same as the peak performance.

As shown in Table 2.2, the peak transfer rates of memory accesses of the S-3800 are the most high among the target machines. In comparison with that of the S-820, they are improved about 2.5 times. VIST of the S-3800 is 3 times faster than that of the S-820 and the relative throughput of indirect access (U) is also improved. From the point of view of the balance between the performance of memory access and computation, S-3800 with 1 CPU is 2.7 times faster than the S-820, therefore the balance can be considered to be almost the same as that of the predecessor. This improvement is significant because the clock period of the S-3800 is only the half of that of S-820 and there is no change in the organizations of the load and store pipelines.

Another improvement of the performance is seen in the robustness to bank conflicts. The value (W) is the best among the target machines[11]. The large number 64 of S_{half} also proves the robustness.

However, the relative throughput of the indirect load is a little decelerated for load operations. The reason is that the performance of direct load (C) is improved remarkably. Another deceleration is seen in the results of direct access for vector length 2^8. The value of (A),(B) is almost as the same as that of the S-820 while the performances for vector length 2^{18} is 2 or 3 times faster than that. It means that the S-3800 has much bigger N_{half} value which also affects to the unusual value of the robustness to bank conflicts (W).

Also the minimum data transfer rates of the store instructions are lower than that of the load instructions like VP-2600. It is curious that the absolute performance of the minimum data transfer rates of store

[11]The value of (W) for SX-2N is neglectable because of its poor indirect load and store performance.

operations has gone worse compared with that of the S-820.

SX-3R/14

A fully configured SX-3R system is a multi-processor system with 4 CPUs, but the machine we could use is a SX-3R/14[12] which is the fastest machine among the single processor models.

Unlike the S-3800, the SX-3R series is equipped with fully reorganized memory system compared with the predecessors. A CPU of the SX-3R has 2 load and 1 store pipes while the SX-2N has only 1 pipe. This means that it is easy to evaluate the results as described in Section 2.3.

From the results, it is noticed that this machine shows quite good performance for direct loads and stores even for the short vector length. On the contrary, throughput for indirect access is still poor. The absolute performance of the indirect access is almost the same as that of S-820/80, but compared with the performance of the direct access, the performance of indirect read is 2.9 times lower than that of direct read and indirect write is 7.8 times lower than direct write. It is obvious that the throughput of the load and store pipelines of the SX-3R is seriously degraded in indirect access.

Since this machine shows good performance even for a short length of vector, the value of (W) is available to evaluate the robustness to bank conflicts. The value of (W) means that SX-3R can be decelerated easily by bank conflicts. Comparing the performance parameters of direct store (B) with (H), we can see that the effects of bank conflicts on this machine is quite serious when writing. It also indicates that the minimum data transfer rate of the SX-3R is not only lower than that of the other machines but also decelerated by indirect accesses as shown in the values (I),(J).

2.5 Conclusions

We have obtained fundamental data on the performance of indirect vector accesses. As mentioned in the first section, vector indirect ac-

[12]Computation Center, Osaka University.

cesses are valuable in the fields of numerical computations, and are also needed for advanced automatic vectorization. Therefore this data enables us to make good use of supercomputers and make it possible to extend their applications.

We had no chance to run the benchmarking program on neither SX-3R with 4 CPUs, Y-MP 8 with 8 CPUs, C90 nor T90. To make our research more concrete, it is indispensable to get their data and we are eager for the chance.

Table 2.1: The performance and hardware facilities of each machine

Machine	S-820/80	VP-2600/10	SX-2N	CRAY Y-MP 8-4128
Number of Processors	1	1	1	4
Theoretical Max.speed (MFlops)	3000†	5000	1142	500† × 4 processors
Load/store pipes	(4data para.) × 1 load pipe (4data para.) × 1 load/store pipe	(4data para.) × 2 load/store pipes	(8data para.) × 1 load pipe also serves as (4data para.) × 1 store pipe	2 load pipes 1 store pipe × 4 processors
clock period(ns)	4	3.2	7	6
Number of Memory Banks	128	512	512	256
Vector length per strip mining	512	2048	256	64
Compiler Version	FORT77/HAP V24-0F	FORTRAN77EX/VP V11L10	FORT77/SX Rev.041	CFT77 4.0.3.6

Machine	S-3800/480	SX-3R/14	CRAY Y-MP 2E-264
Number of Processors	4	1	2
Theoretical Max.speed (MFlops)	8000 × 4 processors	6400	500† × 2 processors
Load/store pipes	(4data para.) × 1 load pipe (4data para.) × 1 load/store pipe × 4 processors	(4data para.) × 2 load pipes (4data para.) × 1 store pipe	2 load pipes 1 store pipe × 2 processors
clock period(ns)	2	2.5	6
Number of Memory Banks	512	512	256
Vector length per strip mining	512	256	64
Compiler Version	FORT77/HAP V26-00	f77sx Rev.040	CFT77 5.0.3.0

† This value differs from the official performance in U.S. and Europe, but can be achieved and officially announced in Japan.

Table 2.2: Experimental results on each machine

Benchmarking items		Vector length	(Label)	S-820/80	VP-2600/10	SX-2N
Maximum data transfer rate (Dwords/sec.)	A(I)=B(I)	2^8	(A)	352	234	457
	A(I)=1.0D0	2^8	(B)	469	312	599
	A(I)=B(I)	2^{18}	(C)	697	1024	334
	A(I)=1.0D0	2^{18}	(D)	978	1139	438
	A(I)=B(L(I))	2^{18}	(E)	446	443	53.2
	A(L(I))=1.0D0	2^{18}	(F)	244(426†)	578	61.5
Averaged data transfer rate (Dwords/sec.)	A(I)=B(I*S)	2^8	(G)	304	191	402
	A(I*S)=1.0D0	2^8	(H)	320	233	153
	A(I)=B(L(I))	2^{18}	(I)	379	371	52.6
	A(L(I))=1.0D0	2^{18}	(J)	224(413†)	472	61.0
Minimum data transfer rate (Dwords/sec.)	A(I)=B(I*512)	2^8	(K)	29.6	20.6	10.8
	A(I*512)=1.0D0	2^8	(L)	30.3	16.6	11.1
	A(I)=B(L(I))	2^{18}	(M)	30.0	21.6	9.76
	A(L(I))=1.0D0	2^{18}	(N)	31.1(31.1†)	17.1	10.0
S_{half} (stride)	A(I)=B(I*S)	2^8	(O)	16	16	8
	A(I*S)=1.0D0	2^8	(P)	16	16	8
	A(I)=B(L(I))	2^{18}	(Q)	16	4	128
	A(L(I))=1.0D0	2^{18}	(R)	32(32†)	4	128
S_{worst} (stride)	A(I)=B(I*S)	2^8	(S)	128	256	256
	A(I*S)=1.0D0	2^8	(T)	128	256	256
Relative throughput of indirect load/store	$\frac{(E)}{(C)}$	2^{18}	(U)	0.640	0.433	0.159
	$\frac{(F)}{(D)}$	2^{18}	(V)	0.249(0.436†)	0.507	0.140
Robustness to bank conflicts	$\frac{(G)+(H)}{(A)+(B)}$	2^8	(W)	0.760	0.777	0.526
	$\frac{(I)+(J)}{(E)+(F)}$	2^{18}	(X)	0.874(0.908†)	0.826	0.990

(Label)	Y-MP8/4128 (4CPUs)	Y-MP8/4128 (1CPU)	S-3800/480	SX-3R/14	Y-MP 2E/264 (1CPU)
(A)	133	114	351	813	137
(B)	123	107	470	1133	148
(C)	152	583	1941	1259	152
(D)	152	582	1913	1592	152
(E)	121	485	1087	437	139
(F)	144	571	646(1324†)	285	143
(G)	102	114	338	706	104
(H)	116	90.1	536	577	139
(I)	88.7	326	1004	382	92.2
(J)	109	411	583(1241†)	265	108
(K)	32.0	30.1	29.0	23.8	31.5
(L)	32.0	30.7	23.9	21.7	32.7
(M)	29.4	29.6	30.2	17.7	29.2
(N)	31.2	32.9	24.4(24.4†)	17.5	31.3
(O)	16	16	64	8	16
(P)	16	32	16	8	16
(Q)	128	16	32	16	16
(R)	16	16	32(16†)	16	16
(S)	16	32	512	256	32
(T)	32	32	512	256	32
(U)	0.796	0.832	0.560	0.347	0.916
(V)	0.947	0.981	0.338(0.692†)	0.179	0.941
(W)	0.852	0.923	1.063	0.659	0.853
(X)	0.746	0.698	0.916(0.931†)	0.896	0.710

† Values in parentheses are measured using VIST instructions.

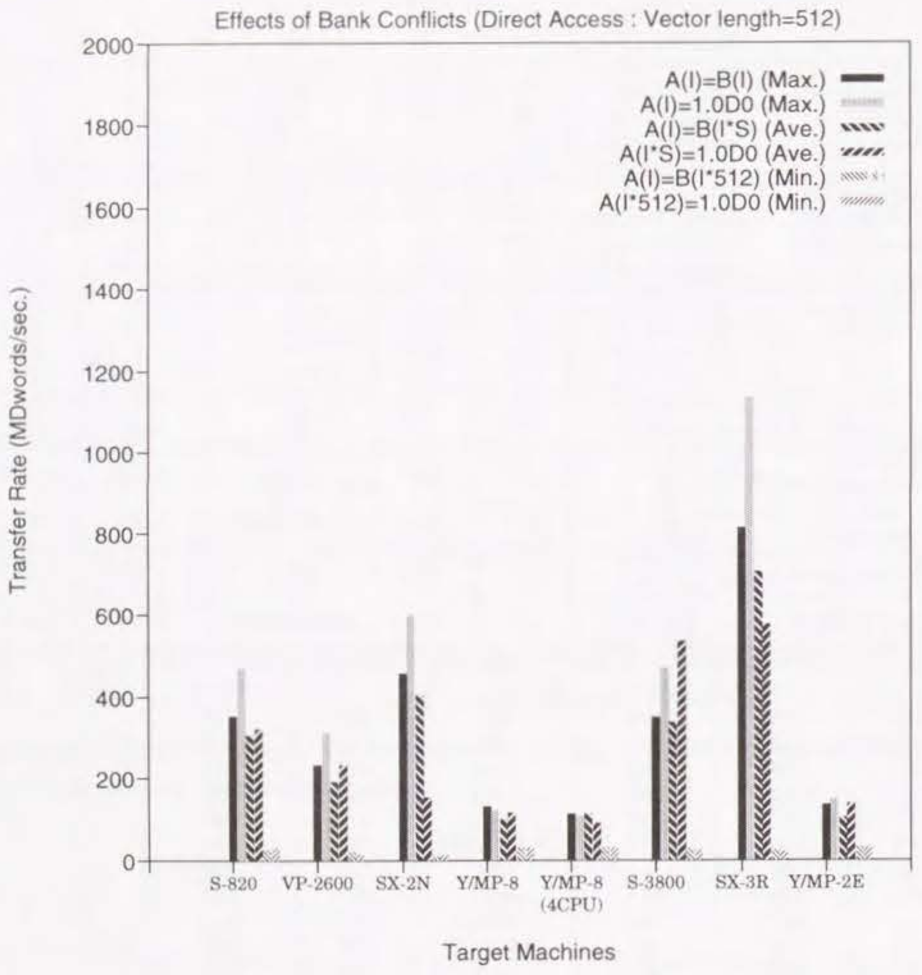


Figure 2.4: Effects of bank conflicts using direct access instructions.

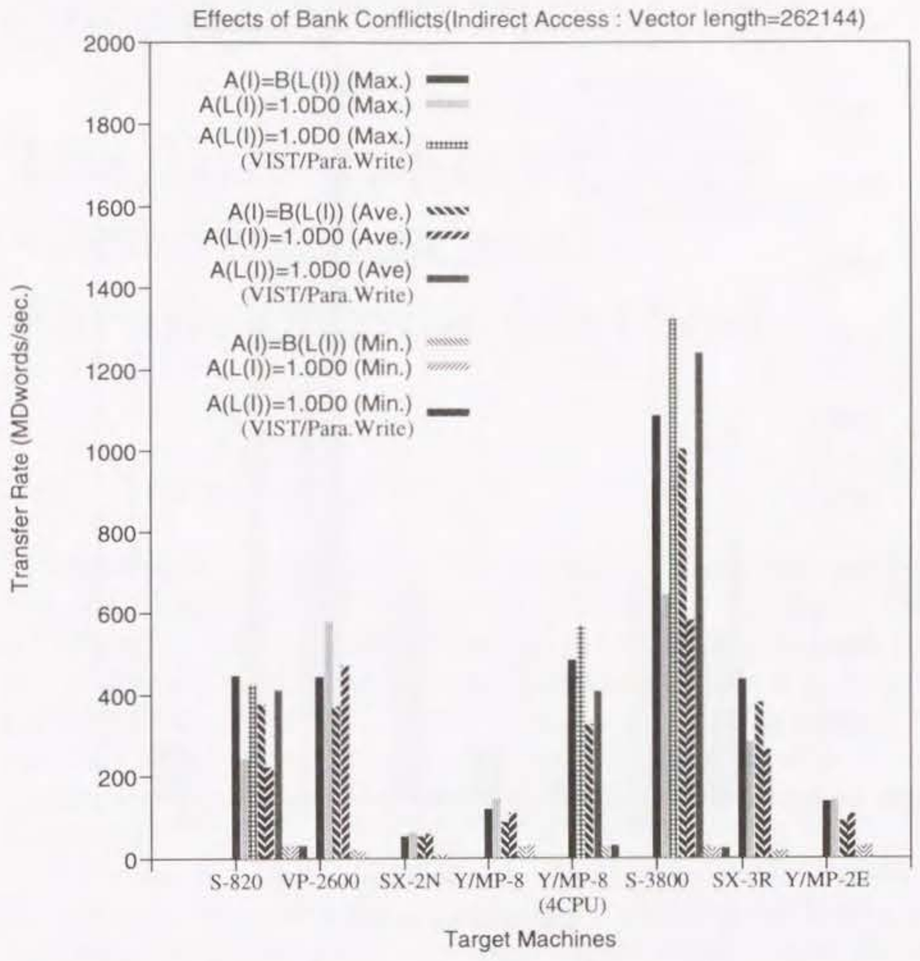


Figure 2.5: Effects of bank conflicts using indirect access instructions.

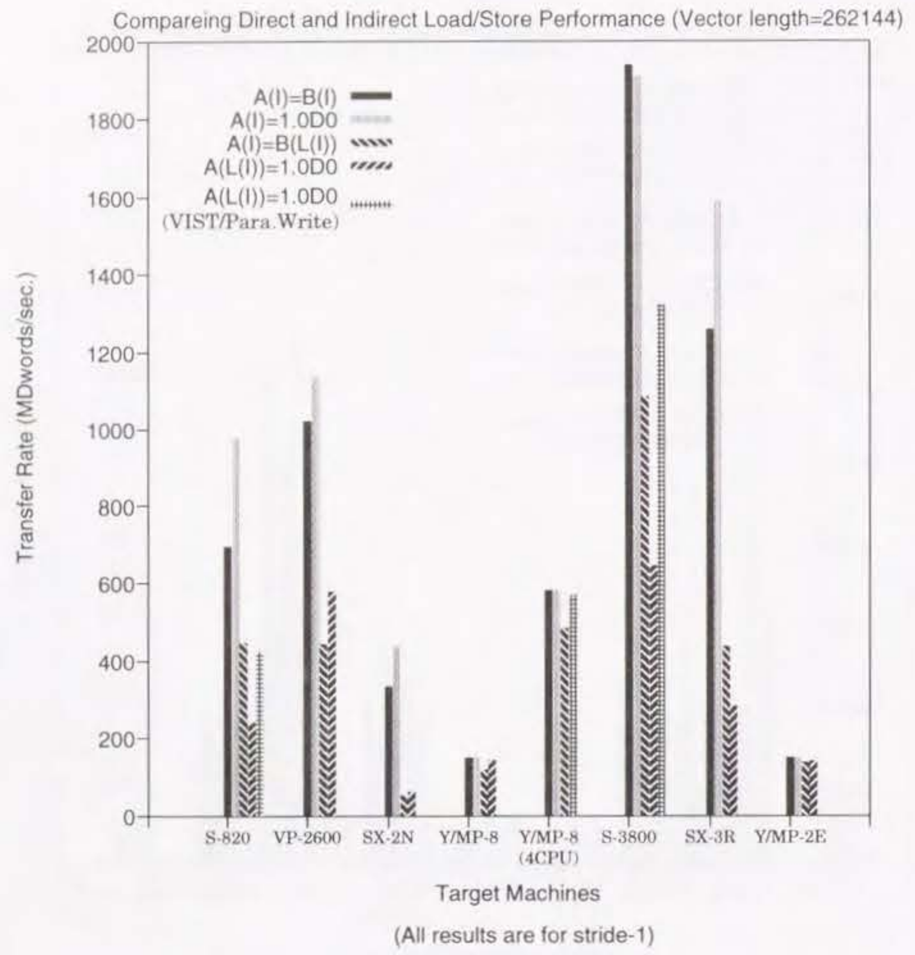


Figure 2.6: Maximum throughput using direct/indirect access instructions.

Chapter 3

The Breadth-first Vectorization and Parallelization Method

3.1 Introduction

In this chapter, we propose a new method for automatic vectorization and parallelization of recursive procedures and functions[1]. There are a number of recursive algorithms for searching, sorting, manipulation of structured data such as trees. Writing programs with recursive procedure is a natural and neat way to implement such algorithms. Accordingly automatic vectorization and parallelization of recursive procedures enable us to apply these recursive algorithms easily on supercomputers.

There are some previous studies to optimize recursive procedures. For example, tail-recursion elimination is a well-known method to convert recursive procedures to equivalent simple loops. There are also some works to eliminate redundant operations in recursive procedures written in functional languages[3][51]. These methods aim to reduce the instructions but these studies are not always applicable for automatic vectorizing/parallelizing compilers. The reason is that the accel-

[1]In this thesis, The term "procedures" include both of procedures and functions, i.e. procedures which have return value.

eration of supercomputers is obtained mainly from vector or parallel execution. It follows that the compilers have to stand on a view of detecting instructions which can be executed in parallel. From this point of view, we propose a new method named *breadth-first method* to vectorize/parallelize recursive procedures. The point is that we have to take notice to the fact that a VP is a sort of SIMD[9] processor. We have to detect parts which consist multiple data-flow from a sequential program to obtain enough performance. We introduce a concept of "dependence of environments" to analyze data-flow in recursive procedures. In our definition, an "environment" is a state of correspondences between the local variables and the stack. In other words, each environment corresponds to each invocation of the recursive procedure.

3.2 Preliminaries

3.2.1 Classification of Recursive Procedures

Here we define a classification of recursive procedures. Note that most of the example programs in this thesis are written in a Pascal-like language, but the argument is not limited to the Pascal but applicable to almost all block-structured languages which include Fortran 90 and C.

Recursive procedures are roughly defined as procedures which contain one or more statements to call itself. Strictly speaking, we have to take *indirectly* recursive procedures into consideration. Let us take a case of a procedure named **A** which calls another procedure **B**. If **B** also calls **A**, we call **A** and **B** as indirectly recursive procedures. But in this thesis we treat only directly recursive procedures for simplicity. In most cases, indirectly recursive procedures can be converted into equivalent directly recursive ones using inline expansion, as shown in Figure 3.1. But this conversion is not always possible. Figure 3.2 shows the example. This type of procedures are left to our future works.

In this thesis, we classify recursive procedures into three types according to the number of recursive-call statements in procedures. Exactly speaking, we classify them according to possible number of recursive invocations on each environment. When a recursive procedure invokes itself not more than once on each environment, we classify it

```
procedure A;
begin                         procedure A';
   S1;                        begin
   if P then B;                  S1';
   S2                            if P' then
end;                                S3';
procedure B;         ⟹             if Q' then A';
begin                               S4'
   S3;                           end;
   if Q then A;                  S2'
   S4                         end;
end;
```

Figure 3.1: Conversion from an indirectly recursive procedure to directly recursive procedure.

```
procedure A;
begin
   S1;
   if P then A;
   if Q then B;
   S2
end;
procedure B;
begin
   S3;
   if R then A;
   if S then B;
   S4
end;
```

Figure 3.2: An indirectly recursive procedure which is impossible to convert into directly recursive procedure by inline expansion.

into **Type-1**. If a recursive procedure invokes itself more than once but the number of invocation is statically limited by a constant on each environment, we classify it into **Type-2**. The other procedures are classified into **Type-3**. Examples of each procedure are shown in Figure 3.3.

Behavior of a type-1 procedure is very simple. It can be distinguished into two phases. In the first phase, recursive calls occur continuously. But once the control reaches to the end of the procedure, it never calls itself again until the control returns out from the procedure. Therefore in the second phase returns occur continuously. These two phases can be easily expressed as loops and a type-1 procedure can be easily converted into a non-recursive procedure.

Contrarily, behaviors of type-2 and 3 procedures are complicated. It is still possible to convert these procedures into non-recursive procedures, but the rewrited procedures contain complicated sequences of statements which emulate the stack explicitly.

3.2.2 Facilities of Vector Supercomputers

Vector supercomputers are classified into SIMD(Single Instruction Multiple Data stream) computers[9]. Their high performance results from the facility of multiple vector pipelines. When each element of a vector has no dependence on the other elements, the pipelines work efficiently and we can obtain high performance.

Most of recent vector supercomputers also provide special instructions called *vector-macro instructions*. Examples of these instructions are shown in Table 3.1. They are used to execute computations on which vectors have dependence between the elements. The performance of the vector-macro instructions is not so high as that of ordinal vector instructions, but it is much higher than that with scalar instructions for the same computation. As we describe later, the vector-macro instructions are very important facilities in order to vectorize recursive procedures.

```
function  power(x : real; n : integer) : real;
  begin
    if n=0 then
        power := 1
    else if odd(n) then
        power := power(x*x, n div 2)*x
    else
        power := power(x*x, n div 2)
  end;
```
(A) calculate power x^n ($n \geq 0$) (a type-1 procedure)

```
procedure  hanoi( disk, pole1, pole2 : integer );
  begin
    if disk>0 then begin
        hanoi( disk-1, pole1, 6-pole1-pole2 );
        writeln('Disk', disk, pole1, '->', pole2 );
        hanoi( disk-1, 6-pole1-pole2, pole2 )
    end
  end;
```
(B) tower of Hanoi (a type-2 procedure)

```
procedure Nqueens( i:integer );
var j:integer;
begin
  for j := 1 to N do
    if A[j] and B[i+j] and C[i-j] then begin
        X[i] := j;
        A[j] := false;  B[i+j] := false;  C[i-j] := false;
        if i<N then Nqueens(i+1) else PrintAnswer;
        A[j] := true;   B[i+j] := true;   C[i-j] := true
  end
end;
```
(C) N queens (if N is constant, classified into type-2. otherwise, type-3)

Figure 3.3: Examples of recursive procedures.

Table 3.1: Examples of the vector-macro instructions.

macro type	S-820/3800[13]	VP400/2600[10]	SX-2/3[26]
Vector Element Sum `s := s + a[i]`	VSMD VSMDM VSMDN VSM VSMM VSMN	VSMD VSM VSMM VSMS VSMSD VSMSE	VSUM
Vector Inner Product `s := s + a[i] * b[i]`	VIPD VIPDM VIPDN VIP VIPM VIPN		
First Order Iteration `a[i] := a[i-1] * B[i] + c[i]`	VITRD VITR	VRC VRCD VRCE	VIMA VIMS VIAM VISM
Vector Element Increment `a[i] := a[i-1] + c[i]`	VINCD VINC		VFIA VFIS VFIM
Find Maximum Data `s := Max(a[i])`	VMAXD VMAX	VFXD VFX VFXDX VFXX	
Find Minimum Data `s := Min(a[i])`	VMIND VMIN	VFND VFN VFNX VFNDX	

3.3 The Depth-first Method

Before describing our new method, we describe the depth-first method which was previously proposed because the weakness of this method will help to introduce our new method.

3.3.1 The Depth-first Vectorization

It is always possible to convert recursive procedures into equivalent non-recursive procedures. Since the converted procedures contain some loops, we have some opportunities to vectorize these loops. We call this approach **the depth-first vectorization method**. This method is based on the idea which was originally given by Kozuka[19] and we slightly revised and formalized the method.

Let us start our explication with a simple example. Consider a recursive procedure to obtain N-th factorial (Figure 3.4). This procedure is classified into type-1. For the latter arguments, we transform the procedure as shown in Figure 3.5. The purpose of this transformation is to get a simple statement which consists no instruction except a recursive call, in other words, a set of instructions to store the values of arguments, to call itself and to load the return value. This automatic transformation is always possible introducing some temporary variables. In this case, we removed the term of **n** from the right hand side of the assignment statement introducing a temporary variable **r**.

As mentioned in section 3.2.1, the behavior of this type of procedure can be divided into two phases. In the case of Figure 3.5, while the condition $n \neq 0$ is satisfied, the recursive calls occur repeatedly. In the first phase, the statements labeled *S1* and *S2* in the Figure 3.5 are executed. Once the condition $\mathbf{n} = 0$ is satisfied, the execution shifts to the phase two. In this turn, the statements from *S2* to *S6* are executed repeatedly.

Transforming the recursive procedure into a non-recursive procedure, we can obtain two loops as shown in Figure 3.6. The former loop can be easily converted with **while**. These two loops can be vectorized using existent methods described in [40] and [33]. Thus the recursive procedure is automatically vectorized as shown in Figure 3.7.

From this example we can easily suppose that all of type-1 recursive

```
function fact(n : integer) : integer;
begin
    if (n = 0) then
        fact := 1
    else
        fact := fact(n-1) * n
end;
```

Figure 3.4: A procedure to get N-th factorial.

```
function fact(n : integer) : integer;
var r : integer;
begin
    if not(n = 0) then  { S1 }
        r := fact(n-1); { S2 }
    if (n = 0) then     { S3 }
        fact := 1       { S4 }
    else                { S5 }
        fact := r * n   { S6 }
end
```

Figure 3.5: Converted form of the procedure *fact*.

```
function fact(n : integer) : integer;
label 1;
var stackR : array [0..MAX] of integer;        { stack }
    stackN : array [0..MAX] of integer;        { stack }
    stackFact : array [0..MAX] of integer;     { return values }
    sp, maxsp : integer;                       { stack pointer }
begin
    { Setup the stack }
    sp := 0;
    stackN[sp] := n;
    { Phase1 start }
1:
    if not(stackN[sp] = 0) then begin          { S1 }
        { S2 : call sequences }
        stackN[sp+1] := stackN[sp] - 1;
        sp := sp + 1;
        goto 1
    end;
    { Phase2 start }
    maxsp := sp;
    for sp := maxsp downto 0 do begin
        if not(stackN[sp] = 0) then            { S1 }
            { S2 : get the return value }
            stackR[sp] := stackFact[sp+1]
        if (stackN[sp] = 0) then               { S3 }
            stackFact[sp] := 1                 { S4 }
        else                                   { S5 }
            stackFact[sp] := stackR[sp] * stackN[sp] { S6 }
    end;
    { Restore form the stack }
    fact := stackFact[sp]
end
```

Figure 3.6: Non-recursive form of the procedure *fact*.

```
function fact(n : integer) : integer;
label 2;
var stackR : array [0..MAX] of integer;      { stack }
    stackN : array [0..MAX] of integer;      { stack }
    stackFact : array [0..MAX] of integer;   { return values }
    sp, maxsp : integer;                     { stack pointer }
    OUTER,INNER : integer;                   { loop counter }
begin
    sp := 0;     { Setup the stack }
    stackN[sp] := n;
    { Phase1 start }
    for OUTER := 0 to (MAX mod STRIPWIDTH) do begin
        for INNER := 0 to (STRIPWIDTH-1) do
            stackN[sp+INNER+1] := stackN[sp+INNER]-1;
                { *vectorized* }
        if (RUNOVER) then begin
            BACKTRACKING;
            goto 2
        end;
        sp := sp + STRIP_WIDTH;
    end;
    STACKOVERFLOW;
2:  { Phase2 start }
    maxsp := sp;
    for sp := maxsp downto 0 do begin
        if not(stackN[sp] = 0) then
            stackR[sp] := stackFact[sp+1] { *vectorized* }
        if (stackN[sp] = 0) then
            stackFact[sp] := 1  { *vectorized* }
        else
            stackFact[sp] := stackR[sp] * stackN[sp]
                { *vectorized* }
    end;
    fact := stackFact[sp]      { Restore form the stack }
end
```

Figure 3.7: Vectorized form of the procedure *fact*.

procedures can be treated in the same way. In fact, it is always possible to convert any type-1 procedure automatically into a non-recursive procedure with such simple two loops, and the opportunity of vectorization is found in the loops. Therefore the type-1 procedures are considered to be suitable to apply the depth-first vectorization method. This is the basic idea of this method.

The procedure of the depth-first vectorization for type-1 procedures are given as follows. For simplicity, we assume a type-1 recursive procedure with no loop structure and no other procedure call except recursive ones in the body. After the needed interchange and the replacement of the statements, a recursive procedure which satisfies the assumption can be converted into a procedure like Figure 3.8. Here we call the process as "normalization". The normalization can be done with commonly-used methods for compilers. The details are described in Figure 3.9.

After the normalization, we can easily eliminate the recursive call and can get the non-recursive procedure as Figure 3.10. The algorithm for the conversion is shown in Figure 3.11 and Figure 3.12.

Thereby we have obtained two loops, and these are treated with our existent vectorization methods for infinite iterations[40] and loops with recursive reference[33]. Here we assume that the statements $\mathbf{S}_{head}$ and $\mathbf{S}_{tail}$ of Figure 3.10 contain no loops, but even with loops, we have methods to vectorize nested loops[43][12][53][34][24]. Thus we have the vectorization method which can treat all type-1 recursive procedures.

3.3.2 Problems

Although the depth-first method can be applied to all type-1 recursive procedures, it does not always work successfully. For example, it must be pointed out that in Figure 3.7 the recursive procedure is vectorized with vector-macro instructions. In general, if we take the depth-first vectorization method, the obtained loops have some dependencies between iterations and the recursive procedures must be vectorized with vector-macro instructions in most cases. The reason is that the statements to give arguments and get return values nearly always cause first-order dependence between the iterations of the loops of $\mathbf{S}_{head}$ and $\mathbf{S}_{tail}$. It is easily understood from the fact that each iteration of the loop

```
function recursive(arg : art-t) : ret-t;
begin
    $S_{head}$;
    if $P$ then ret := recursive(a);
    $S_{tail}$
end
```

Figure 3.8: Normalized type-1 procedure.

1. Make all arguments of recursive call(s) as simple variables.

```
                                           tmp_arg := a+b;
retval := recur(a+b,d);     ==>            retval := recur(tmp_arg,d);
```

2. Replace recursive calls with a simple variable if they are in longer expressions.

```
                                           retval := recur(a,b);
result := recur(a,b)+c;     ==>            result := retval+c;
```

3. Split off and the recursive calls from blocks and combine them as follows.

```
                                   if P then begin          {S_head}
                                       S1; a := a1          {S_head}
                                   end else                 {S_head}
if P then begin                      if Q then begin        {S_head}
    S1;                                  S3; a := a2        {S_head}
    r1 := recur(a1);                 end                    {S_head}
    S2                             end;                     {S_head}
end else                           if P or Q then
  if Q then begin           ==>        r := recur(a);
    S3;                            if P then begin          {S_tail}
    r2 := recur(a2);                   r1 := r; S2          {S_tail}
    S4                             end else                 {S_tail}
  end                                if Q then begin        {S_tail}
end;                                     r2 := r; S4        {S_tail}
                                     end                    {S_tail}
                                   end;                     {S_tail}
```

Note: **P and Q** is always **false** since **recur** is a type-1 procedure

Figure 3.9: Normalization of type-1 procedures.

```
function non_recursive(arg : art-t) : ret-t;
begin
    StackSetup;
    repeat
        $S'_{head}$;
    until not($P'$)
    for StackPointer := StackBottom downto 0 do
        $S'_{tail}$;
    RestoreFromStack;
end
```

Figure 3.10: After conversion from a type-1 recursive procedure to a non-recursive procedure.

corresponds to each environment and giving each argument or getting each return value mean data transfer between environments. Therefore this dependence is unavoidable. As the vector-macro instructions can process only some limited patterns of computation as shown in Table 3.1, the vectorization is not always succeeded. Even when it is succeeded, since the execution speed of vector-macro instructions is slower than the other vector instructions, recursive procedures are not much accelerated by the depth-first method compared with the other vectorizations[2].

3.4 Basic Idea

As mentioned in the previous section, arguments and return values of a recursive procedure causes dependence across environments. In the cases of the depth-first vectorization, this type of dependence severely restrict the opportunities for recursive procedures to be vectorized. To cope with this problem, the breadth-first vectorization method is designed standing on a point of view to avoid this dependence.

The basic idea is that if two or more environments have no dependence on each other, they can be executed in parallel. Let us start with a type-2 procedures which was already normalized in the form of Figure 3.13 as an example. Considering a situation that the execution

[2]When the vector length is long enough (typically more than 1000), we can use an algorithm described in [31] and can obtain better results.

Step-1 Construct explicit stack changing definitions of all local variables of the procedure to arrays of the original variable. Then all arguments of call-by-value and return values are treated as the same as local variables. Call-by-reference type arguments should be treated as arrays of pointers of the original. Definition for the stack pointer is also needed.

```
function rec( arg : integer; var varg : integer ) : real;
var a : real;
    b : array [1..10] of integer;
    ⇓
function rec( arg : integer; var varg : integer ) : real;
type varg-p = ↑integer;
var arg-stack : array [0..STACKMAX] of integer;
    argv-stack : array [0..STACKMAX] of varg-p;
    ret-stack : array [0..STACKMAX] of real;
    a-stack : array [0..STACKMAX] of real;
    b-stack : array [0..STACKMAX] of array [1..10] of integer;
    sp : integer; { Stack Pointer }
```

Step-2 Put the instructions to set up the explicit stacks at the entry of the procedure. Put also the instructions to restore return value form the explicit stack at the end of the procedure.

```
{ At the entry }
sp := 0; { Setup the Stack }
arg-stack[sp] := arg;
argv-stack[sp] := GetPointer(varg-p);
...
{ At the exit }
rec := ret-stack[sp]];
```

Figure 3.11: The algorithm to convert a type-1 procedure to a non-recursive procedure (cont.)

Step-3 Replace the instructions within S_{head} and S_{tail} to access the explicit stacks as follows. Name the converted S_{head}/S_{tail} as S'_{head}/S'_{tail}.

```
a := b[arg];   ⟹   a-stack[sp] := b-stack[sp][arg-stack[sp]];
a := varg;     ⟹   a-stack[sp] := argv-stack[sp]↑;
```

Step-4 Make statements for the recursive call and return from the call.

```
                      arg-stack[sp+1] := a1;
                      varg-stack[sp+1] := GetPointer(a2);
r := rec(a1, a2); ⟹  sp := sp+1;
                      &
                      sp := sp-1;
                      r[sp] := ret-stack[sp+1];
```

Step-5 Make loops from S'_{head}/S'_{tail} as shown in Figure 3.10

```
function rec( arg : integer; var varg : integer ) : ret-t;
begin
    StackSetup;
    repeat
        S'head;
        if P' then begin
            AssignArguments;
            StackPointer := StackPointer + 1
        end;
    until not(P')
    for StackPointer := StackBottom downto 0 do begin
        if P' then
            GetReturnValue;
        S'tail
    end;
    RestoreFromStack
end
```

Figure 3.12: The algorithm to convert a type-1 procedure to a non-recursive procedure (cont'd)

```
function recursive(arg : art-t) : ret-t;
begin
    $S_{head}$;
    if $P_1$ then $R_1$ := recursive($A_1$);
    $S_1$;
    if $P_2$ then $R_2$ := recursive($A_2$);
    $S_2$;
    ...
    $S_{n-1}$;
    if $P_n$ then $R_n$ := recursive($A_n$);
    $S_{tail}$
end
```

Figure 3.13: A normalized type-2 procedure without loops.

```
function recursive(arg : art-t) : ret-t;
begin
    $S_{head}$;
    if $P_1$ then $R_1$ := recursive($A_1$);
    if $P_2$ then $R_2$ := recursive($A_2$);
    ...
    if $P_n$ then $R_n$ := recursive($A_n$);
    $S_{tail}$
end
```

Figure 3.14: A target procedure for the breadth-first vectorization.

has just reached at the entry of the procedure and a certain environment has just created, it is evident that the environment is depended by each environment created by the recursive calls on the environment. But all environments created by the same environment may not have dependence on each other. If it is true, the statements S_1,S_2,...,S_{n-1} can be moved and combined with S_{head} or S_{tail} and the procedure can be expressed as in Figure 3.14. In this case, all of these environments can be invoked in parallel, consequently we can put these environments in vector execution. In fact, more detailed analysis is needed, but roughly, this is the basic idea of the breadth-first vectorization method.

3.5 Dependence between Environments

To consider the details, we introduce a concept of dependence between environments.

When an environment invokes another environment, we call the former environment as the "parent environment" and the latter as the "child environment". In general, a parent environment invokes a certain number of child environments. To express these relations of each environment, ordered trees are suitable. We call the tree as "environment tree".

Figure 3.15 shows an example of the environment tree. Each node of the tree corresponds to each environment and a parent-child relationship of the nodes corresponds to that of the environments. The root node corresponds to the environment when the procedure was invoked from outside. The order of environment invocation is expressed with the preorder traversal of the environment tree.

Here we define some terms for the environment tree in imitation of that of the ordered tree. When an environment is an ancestor of another environment on the tree, we define the former as an "ancestor environment" of the latter. Conversely, the latter is a "descendent environment" of the former. Similarly, we define "brothers environments" which form brothers in the environment tree. According to the custom, we draw the order of brothers from left to right in the tree. We also introduce the terms "environment depth", "root environment", "left-child environment", "right-child environment", "n-th child envi-

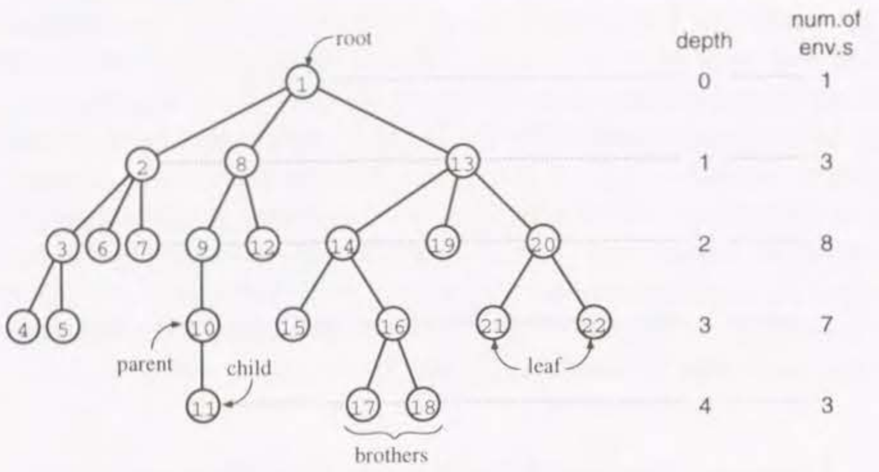


Figure 3.15: A sample of the environment tree.

ronment", "leaf environment" and so on, from the terms of trees[1].

Here we define environment dependence in imitation of that on statements[4] as follows:

In a recursive procedure P, environment E_L *directly depends* on environment E_F (denoted as $E_F \delta E_L$) if there exist a memory location M such that

1. Both E_F and E_L reference M, and at least one of those references is a write;
2. In the scalar execution of an invocation P, E_F is executed before E_L; and
3. In the same invocation of P, M is not written between the time E_F finishes and the time E_L starts.

Thus there are three types of dependence based upon the types of the two references to M. Environment E_L is

flow-dependent on E_L, if E_F writes M and the E_L reads it;
anti-dependent on E_L, if E_F reads M and the E_L writes it; and
output-dependent on E_L, if E_F writes M and the E_L reads it.

We also define the following terms:

1. Environment E_Y *indirectly depends* on E_E if there exist another environment E_P such that
   (a) E_P depends on E_E by an instance S'_E of statement S_E;
   (b) E_P also depends on E_Y by an instance S'_Y of statement S_Y; and
   (c) there exists a list of instances $S'_1, S'_2, ..., S'_n$ of statements $S_1, S_2, ..., S_n$ on E_P which satisfies $S'_E \delta S'_1 \delta S'_2 \delta ... \delta S'_n \delta S'_Y$.
2. Environment E_D *depends* on E_S if E_D directly or indirectly depends on E_S.

See Figure 3.16 and Figure 3.17 as examples.

Note that, while dependence of statements treats static program texts, dependence of environments treats relation of instances of executed statements. The shape of a environment tree comes out only after the whole execution. But we can still judge a pattern of environment dependences and use it to determine a recursive procedure can be vectorized or not.

If a pair of environments has a dependence, exchanging their order of execution causes different results. When we transform a recursive procedure, we have to give attention to the dependence of environments, and we also have to take care not to change the order. Therefore we have to analyze the statements of the recursive procedure and detect possible dependence between environments.

In recursive procedures, direct dependences arises when

1. a parent environment passes arguments to its child environment;
2. a parent environment receives return value to its child environment;
3. an environment writes to a call-by-reference argument and its parent environment reads it;

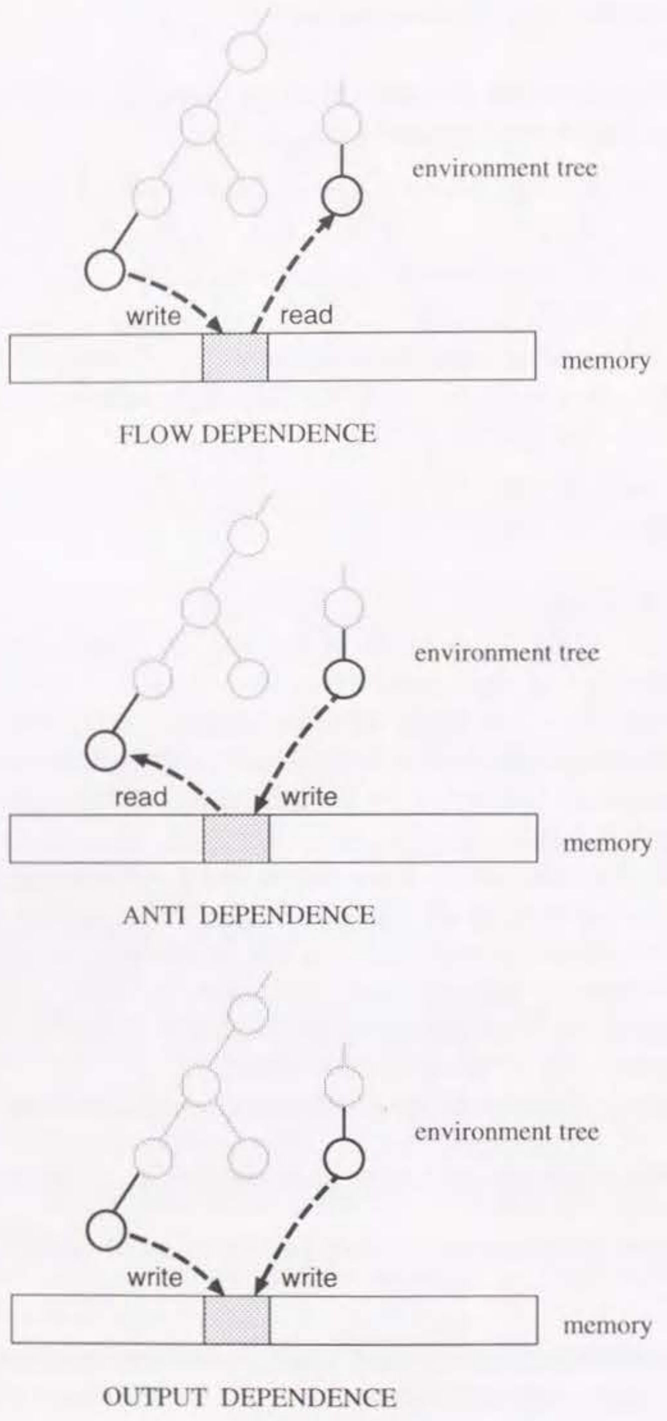


Figure 3.16: Dependence between environments

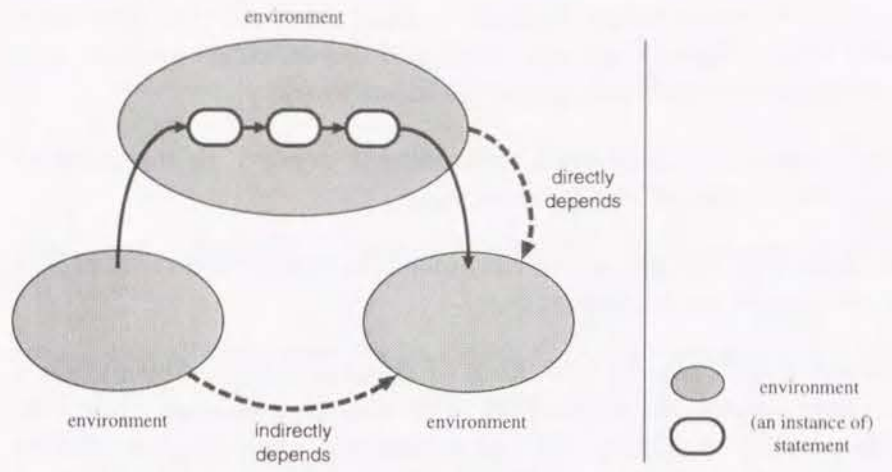


Figure 3.17: Direct and indirect dependence between environments environments

4. two environments access to the same external variable and at least one of the access is writing; and

5. an environment writes to a pointer variable which points an automatic variable of an ancestor environment, and the ancestor reads it.

The first three cases are easy to detect while the rest cases are difficult to treat. The detections are described later in Section 3.8.

3.6 Breadth-first vectorization

Consider a procedure like Figure 3.14. For simplification, it is also assumed that the procedure calls no other procedure. This procedure can be classified into type-2. Assuming that the procedure has no write statements to the external variable, direct dependences are caused by only from the arguments and the result values on each environment. It follows that direct dependences exists only between an environment and its child or parent environment, and indirect dependences only arise between brothers environments. We also assume that, in this procedure, none of the arguments A_2,A_3,...,A_n and the boolean variables

P_2,P_3,...,P_n depends on any of R_1,R_2,...,R_{n-1}, then the brothers environments never depends on each other and dependences remains only in the child-parent relationship, on the other words :

- Instances of S_{head} in each environment depend on instances of S_{head} in its parent environment, and
- Instances of S_{tail} in each environment depend on instances of S_{tail} in its parent environment.

It means that when an execution of statement S_{head} finished on a certain environment, all of its child environments can start their execution in parallel. Applying this fact recursively, the procedure can be transformed and executed as shown in Figure 3.18.

Figure 3.19 shows the execution process on the environment tree. This transformation change the order of execution of environments, but with the aforementioned assumption, the order is preserved between a pair of depended environments to each other. Thus we can get the same result as that of the normal execution. And in Step-2 and Step-6 of Figure 3.18, all environments are independent each other, and they can be executed in parallel. In this case, on each parallel environment, the instructions is all the same, therefore they can be executed with vector instructions in most cases. Thus we can vectorize recursive procedure. This process is applicable to the recursive procedures which may call itself more than twice in an environment, i.e. type-2 or type-3 procedures. This is the newly-proposed vectorization method, "breadth-first vectorization method".

According to this method, we can transform the procedure of Figure 3.14 to that of Figure 3.20. The body of the procedure consists of two doubly nested loops. But the inner loop of each nested loop can be vectorized because each instance of the loop corresponds to each independent environment.

In general, when a type-2 or type-3 recursive procedure always generate an environment tree which satisfies the following condition, we call the the procedure as "suitable" for the breadth-first vectorization.

> Each environment doesn't depend on the other environments except its ancestors and descendant environments.

Step-1 Let a counter $D \leftarrow 0$. D holds the depth of the environment tree.

Step-2 For all environments on depth D of the environment tree,

1. Execute S_{head},
2. Evaluate all of P_1,P_2,...,P_n,
3. Count the number of environments on depth D+1, and
4. Pass arguments to the child environments.

Step-3 Let $D \leftarrow D+1$.

Step-4 Repeat the process from Step-2 to Step-3 until there exists no environment on depth D.

Step-5 Let $D \leftarrow D-1$.

Step-6 For all environments on depth D,

1. Get return values from their child environments,
2. Execute S_{tail} in parallel.

Step-7 Repeat the process from Step-5 to Step-6 until D equals 0. □

Figure 3.18: The behavior of a type-2 procedure vectorized by the breadth-first vectorization method.

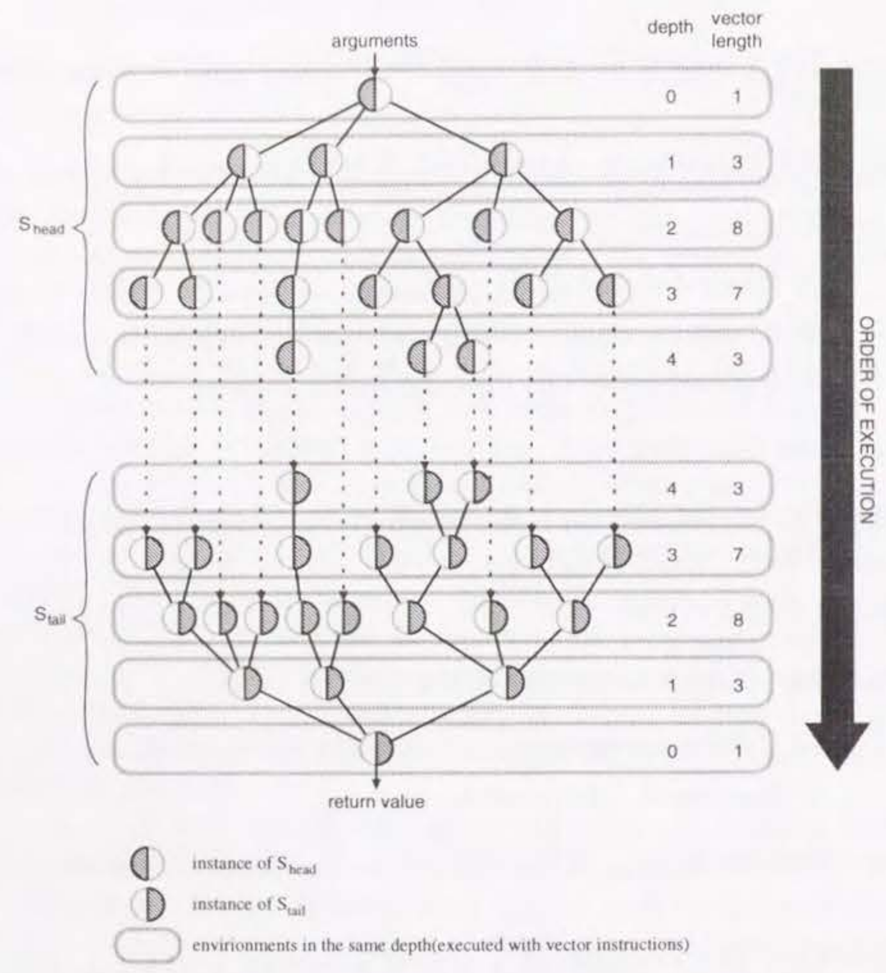


Figure 3.19: Breadth-first vector execution.

```
function recursive(arg : art-t) : ret-t;
begin
      StackSetUp;
      EnvTreeDepth := 0;
      NumberOfEnv[EnvTreeDepth] := 1;
      repeat
 {*v*}    for i:= 1 to NumberOfEnv[EnvTreeDepth] do begin
 {*v*}          $S'_{head}$;
 {*v*}          if $P'_1$ then $CreatEnvAndSetArgForA_1$;
 {*v*}          if $P'_2$ then $CreatEnvAndSetArgForA_2$;
 {*v*}          . . .
 {*v*}          if $P'_n$ then $CreatEnvAndSetArgForA_n$;
            end
            EnvTreeDepth := EnvTreeDepth + 1;
      until (NoChildEnvExists)
      MaxDepth := EnvTreeDepth - 1;
      for EnvTreeDepth := MaxDepth downto 0;
 {*v*}    for i:= 1 to NumberOfEnv[EnvTreeDepth] do begin
 {*v*}          if $P'_1$ then $GetReturnValueForR_1$;
 {*v*}          if $P'_2$ then $GetReturnValueForR_2$;
 {*v*}          . . .
 {*v*}          if $P'_n$ then $GetReturnValueForR_n$;
 {*v*}          $S'_{tail}$
            end
      end
end
```

Note : lines with {*v*} are vectorized

Figure 3.20: After the breadth-first vectorization.

The breadth-first vectorization may changes the order of execution between environments, but the order between a pair of environments with ancestors-descendant relationship never be exchanged. Hence we can apply the breadth-first vectorization method to the procedure if this condition is always guaranteed in the procedure.

3.7 Implementation Details

3.7.1 Code generation

To generate efficient vector instructions from the recursive procedure by this method, storage managements arises to the subject. In the case of the depth-first vectorization method, since the order of the execution does not differ from that of the normal execution, we have only to emulate behavior of the stack. But in the case of the breadth-first vectorization, a kind of memory management mechanism is needed to store the whole environment tree. In general, the shape of environment tree appears only after the execution, therefore we have to treat the storage dynamically.

Recent supercomputers provides **vector load expanding** and **vector store compressing**[3] instructions. Behavior of these instructions is shown in Figure 3.21. Using these instructions, we can dynamically allocate the storage with minimum fragmentation.

As an example, see procedure *sum* in Figure 3.22. Since this procedure has no assignment statements to the external variable *left, right, number*, direct dependence arises only by the argument and the result value. In this case, *lsum* does not depended by the following recursive call *sum*(*right*[*pointer*]), it causes no indirect dependence between all of the brothers environments. Then we can transform this procedure as shown in Figure 3.23 and Figure 3.24. Note that in the figure *envptr[depth]* means the index for leftmost environment on *depth* of the environment tree. After this transformation, the statements marked with {*v*} in the figure can be vectorized. Forced vectorization of the

[3]"Vector load expanding and vector store compressing" are of S-820 and S-3800. In VP-400/2600, instructions with the same functions are called "vector expand/compress". In SX-2/3, they are called "vector scatter/gather."

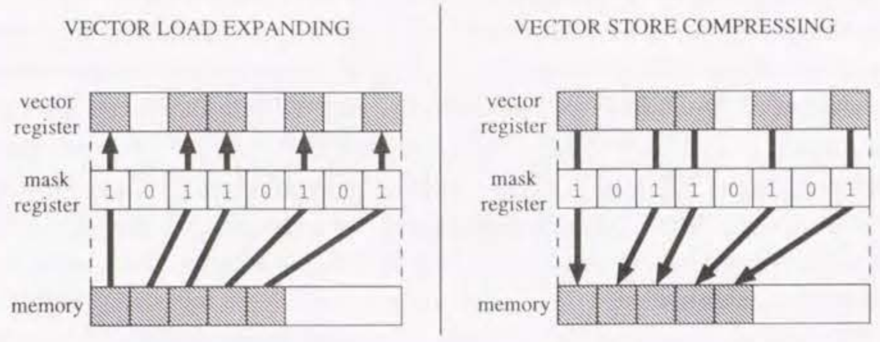


Figure 3.21: Vector expand/compress instructions.

```
{Global variable : left — pointers to the left subtrees
right — pointers to the right subtree
number — values of each node }
function sum( pointer : integer ) : integer;
    var lsum,rsum : integer;
    begin
        lsum := 0; rsum := 0;
        if left[pointer] < > 0 then
            lsum := sum(left[pointer]);
        if right[pointer] < > 0 then
            rsum := sum(right[pointer]);
        sum := lsum + rsum + number[pointer]
    end;
```

Figure 3.22: A procedure to get the sum of a binary-tree data.

inner loop causes different behavior of *eptr* of Figure 3.20, on execution compared the scalar execution. But it will not affect the result because it only affects to the order of allocation for each environment.

In general, if all of the environments never depend on the other environment except its ancestor and descendant environments, the type-2 procedure is normalized as in Figure 3.14. If a statement exists between the recursive procedure call statements and it cannot be merged into S_{head} or S_{tail}, it means that the statement depends on at least two of these recursive calls and it causes indirect dependence between brothers environments. After the normalization, we can convert the procedure with the algorithm shown in Figure 3.25 and Figure 3.26, and it is always vectorizable.

We can also apply this method to type-3 procedures. If a type-3 procedure causes no dependence to prohibit the breadth-first vectorization, the procedure can be vectorized in the same way as type-2 procedures. An example is shown by Figure 3.27 and Figure 3.28. In this case, the problem is that the loops labeled **L1**,**L2** form a nested loop and **L2** cannot be vectorized. But if we ignore dependence of *eptr* we force to vectorize the loop interchanging the loops[34][24]. After the vectorization, the behavior of value of *eptr* changes but it will not affect the whole result of the execution. Similarly, we must exchange the loops **L3** and **L4** in the figure.

In general, if type-3 procedure can be normalized into the form of Figure 3.29, it can be vectorized with the breadth-first method as the same way as the example.

```
        function sum( pointer : integer ) : integer;
        var lsum-array,rsum-array : array [1..MAXENV] of integer;
            pointer-array : array [1..MAXENV] of integer;
            ret-array : array [1..MAXENV] of integer;
            depth, maxdepth : integer; { depth on the environment tree }
            envptr : array [0..MAXDEPTH] of integer;
            i, eptr : integer;
        begin
            depth := 0;
            envptr[0] := 1; { Setup }
            envptr[1] := 2; { Setup }
            pointer-array[envptr[0]] := pointer; { Setup }
            repeat
            eptr := envptr[depth+1];
{*v*}           for i := envptr[depth] to envptr[depth+1]-1 do begin
{*v*}               lsum-array[i] := 0; rsum-array[i] := 0  { S_head }
{*v*}               if left[pointer-array[i]]<>0 then begin
{*v*}                   pointer-array[eptr] := left[pointer-array[i]];
{*v*}                   eptr := eptr+1;  allocate a new environment
{*v*}               end;
{*v*}               if right[pointer-array[i]]<>0 then begin
{*v*}                   pointer-array[eptr] := right[pointer-array[i]];
{*v*}                   eptr := eptr+1;  allocate a new environment
{*v*}               end
                end;
                depth := depth + 1;
                envptr[depth+1] := eptr;
            until( eptr = envptr[depth] ); { until no child env.s exists }
            maxdepth := depth - 1;
                        Note : lines with {*v*} are vectorized
```

Figure 3.23: Vectorized sample procedure of type-2 procedure. (cont.)

```
          for depth := maxdepth downto 0 do begin
              eptr := envptr[depth+1]; pointer to the child environments
              for i := envptr[depth] to envptr[depth+1]-1 do begin
{*v*}             if left[pointer-array[i]]<>0 then begin
{*v*}                 lsum-array[i] := ret-array[eptr];
{*v*}                 eptr := eptr + 1;
{*v*}             end
{*v*}             if right[pointer-array[i]]<>0 then begin
{*v*}                 rsum-array[i] := ret-array[eptr];
{*v*}                 eptr := eptr + 1;
{*v*}             end;
{*v*}         ret-array[i] := lsum[i] + rsum[i]
{*v*}                 + number[pointer-array[i]]; { S_tail }
              end
          end
          sum := ret-array[1];
      end;
                    Note : lines with {*v*} are vectorized
```

Figure 3.24: Vectorized sample procedure of type-2 procedure. (cont'd)

Step-1 Arrange storage for all local variables, arguments and return value on environments. We have only to rewrite each declaration of local variable with its array as the following. Also prepare an array to keep the number of environments on each depth, named *envptr*, and some working variables.

```
function rec( arg : integer; var varg : integer ) : real;
var a : real;
    b : array [1..10] of integer;
    ⇓
function rec( arg : integer; var varg : integer ) : real;
type varg-p = ↑integer;
var arg-array : array [1..MAXENV] of integer;
    argv-array : array [1..MAXENV] of varg-p;
    ret-array : array [1..MAXENV] of real;
    a-array : array [1..MAXENV] of real;
    b-array : array [1..MAXENV] of array [1..10] of integer;
    envptr : array[0..MAXDEPTH] of integer;
    eptr : integer;
```

Figure 3.25: The algorithm to convert a type-2 procedure to apply the breadth-first vectorization. (cont.)

Step-2 Put instructions to set up the root environment and to initialize some working variables at the entry of the procedure. Also put instructions to restore the return value at the exit of the procedure.

```
{ At the entry }
depth := 0; { start form root environment }
envptr[0] := 1; { set the root environment }
eptr := envptr[0];
envptr[1] := 2;
arg-array[1] := arg; { pass the argument to root }
argv-array[sp] := GetPointer(varg-p);
retadd[sp] := 0;
...
{ At the exit }
rec := ret-array[1];
```

Step-3 Replace the instructions within the S_{head} and S_{tail} to access the arrays, and name them as S'_{head} and S'_{tail}.

Step-4 Make statements for the recursive call and return from the call as the same as the conversion in Figure 3.11.

Step-5 Make loops from S'_{head} and S'_{tail} as shown in Figure 3.23.

Figure 3.26: The algorithm to convert a type-2 procedure to apply the breadth-first vectorization. (cont'd)

```
function sum2(pointer : integer) : integer;
var sum, ssum : integer;
    ptr : integer;
begin
    ptr := pointer;
    sum := 0; ssum := 0;
    repeat
        if eldestson[ptr]<>0 then
            ssum:= ssum + sum2(eldestson[ptr]);
        sum := sum + number[ptr];
        ptr := brother[ptr];
    until ptr=0;
    sum2 := sum + ssum;
end
```

(a) Original procedure

Figure 3.27: Example of the breadth-first vectorization of a normalized type-3 procedure. (cont.)

```
function sum2(pointer : integer) : integer;
var sum-array : array [1..MAXENV] of integer;
    ssum-array : array [1..MAXENV] of integer;
    ptr-array : array [1..MAXENV] of integer;
    pointer-array : array [1..MAXENV] of integer;
    ret-array : array [1..MAXENV] of integer;
    envptr : array[1..MAXDEPTH] of integer;
    childnum : array [1..MAXENV] of integer; { Number of children }
    depth, maxdepth : integer;
    eptr, i,j : integer;
begin
    depth := 0;
    envptr[0] := 1; { Setup }
    envptr[1] := 2; { Setup }
    pointer-array[envptr[0]] := pointer; { Setup }
    eptr := envptr[0]; { Points the index to newly generated }
    repeat
 {*v*}  for i := envptr[depth] to envptr[depth+1]-1 do begin
 {*v*}      sum-array[i] := 0;
 {*v*}      ssum-array[i] := 0;
        end;
 {*v*}  for i := envptr[depth] to envptr[depth+1]-1 do begin {L1}
 {*v*}      childnum[i] := 0;
            repeat                                              {L2}
 {*v*}          if eldestson[ptr-array[i]]<>0 then begin
 {*v*}              pointer-array[eptr] := eldestson[ptr-array[i]];
 {*v*}              eptr := eptr + 1;
 {*v*}              childnum[i] := childnum[i] + 1;
 {*v*}          end;
 {*v*}          sum-array[i] := sum-array[i] + number[i];
 {*v*}          ptr-array[i] := brother[ptr-array[i]]
            until ptr-array[i] = 0;
        end
        depth := depth + 1;
        envptr[depth+1] := eptr;
    until (eptr = envptr[depth] );
```

(b) After Conversion (cont.)

Figure 3.28: Example of the breadth-first vectorization of a normalized type-3 procedure. (cont'd)

```
function recursive(arg : art-t) : ret-t;
begin
    S_head;
    for i := 1 to n do
        R[n] := recursive(A[n]);
    S_tail
end
```

Note: the value of variable n is indefinite and it is settled only after the execution of S_{head} on each environment.

Figure 3.29: A normalized type-3 procedure for the breadth-first vectorization.

3.7.2 Storage Overflow

When we execute a procedure vectorized with the breadth-first vectorization, the size of storage for the execution is directly proportional to the number of nodes of the environment tree. Generally it is much larger than that of the scalar execution which is proportional to the height of the environment tree. Concretely, when a procedure calls itself not more than a times on each environment, the storage size of for the worst case $O(a^h)$ is needed while the size of $O(h)$ is needed for the of the tree. Therefore it is much important to take measures for the storage overflow in the case of the breadth-first vectorization.

But this problem can be easily avoided. The solution is that when a shortage of storage occurs, the procedure calls itself recursively and get the return value. Figure 3.30 shows the concept and Figure 3.31 shows the shape of an environment tree created in this case. If we limit the number of environments to a^m on each invocation of the procedure, whole needed storage size is bounded by $O(a^m * (h/m))$ for execution of an environment tree of height h.

3.8 Environment Dependence Analysis

As described in the former sections, to apply the breadth-first vectorization method, we have to guarantee statically that the dependence between environments never exists except descendant-ancestor relationship. Thus dependence analysis for environments is indispensable for automatic vectorization with this method. Here we argue on the methods to analyze the dependence statically from the program.

3.8.1 Without writing external variables

If the procedure has no assignment statements to the external variables, the direct dependence between environments exists only between pairs of parent and child environments. Then analyzing the body of the procedure, we can detect indirect dependence between brothers environments. Indirect dependences occur only when a result of a recursive procedure call affects to the other recursive procedure call. Therefore we have only to analyze relations between these procedure calls, i.e.

```
function recursive(arg : art-t) : ret-t;
begin
      StackSetUp;
      EnvTreeDepth := 0;
      NumberOfEnv[EnvTreeDepth] := 1;
      repeat
 {*v*}   for i:= 1 to NumberOfEnv[EnvTreeDepth] do begin
 {*v*}         $S'_{head}$;
 {*v*}         if $P'_1$ then $CreatEnvAndSetArgForA_1$;
 {*v*}         if $P'_2$ then $CreatEnvAndSetArgForA_2$;
 {*v*}         ...
 {*v*}         if $P'_n$ then $CreatEnvAndSetArgForA_n$;
         end
         EnvTreeDepth := EnvTreeDepth + 1;
      until (NoChildEnvExists or StorageShortage)
      if StorageShortage then begin
         for i := 1 to NumberOfEnv[EnvTreeDeptth] do begin
               $S'_{head}$;
               if $P'_1$ then $R'_1$ := recursive($A'_1$);
               if $P'_2$ then $R'_2$ := recursive($A'_2$);
               ...
               if $P'_n$ then $R'_n$ := recursive($A'_n$);
               $S'_{tail}$
         end;
         MaxDepth := EnvTreeDepth-2
      else
         MaxDepth := EnvTreeDepth-1
      end;
      for EnvTreeDepth := MaxDepth downto 0;
 {*v*}   for i:= 1 to NumberOfEnv[EnvTreeDepth] do begin
 {*v*}         if $P'_1$ then $GetReturnValueForR_1$;
 {*v*}         if $P'_2$ then $GetReturnValueForR_2$;
 {*v*}         ...
 {*v*}         if $P'_n$ then $GetReturnValueForR_n$;
 {*v*}         $S'_{tail}$
         end
      end
end
```

Figure 3.30: A measure for storage overflow.

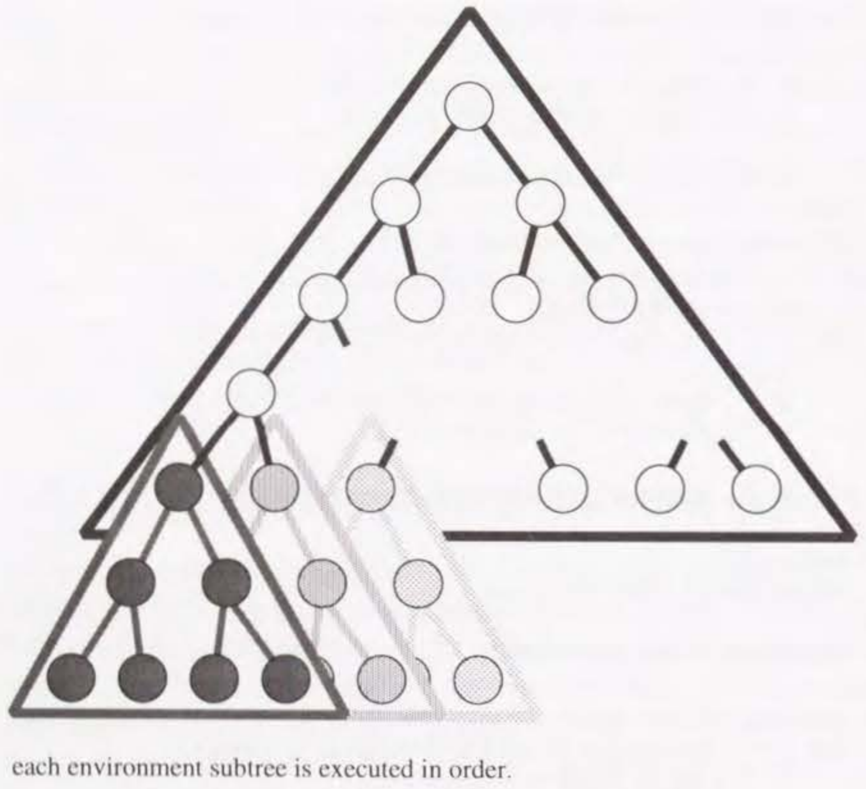


Figure 3.31: Shape of an environment tree when executed with a measure for storage overflow.

relations between each return value and arguments. Note that call-by-reference arguments also regarded as return value. This analysis of the procedure body can be done with well-known techniques such as in [2]. In the case of type-2 procedures as shown in Figure 3.14, these recursive calls are not in any loop, therefore the analysis is easier. In this case, if none of A_2,A_3,...,A_n and the boolean variables P_2,P_3,...,P_n depends on any of R_1,R_2,...,R_{n-1}, brothers environments never depend each other. As simple examples, we show two mathematical function in Figure 3.32.

In the case of type-3 procedures as in Figure 3.29, the recursive call is in a loop, and we have to treat dependence between the iterations of the loop and we should import the analysis for vectorization such as in [23].

In both cases, if we can guarantee that none of the return value affects the other recursive procedure call, we can apply the breadth-first vectorization method to the procedure.

3.8.2 External simple variables and pointer variables

If a recursive procedure contains some statements which write to external variables, we have to consider dependence via the variables. If the variable is a simple variable (neither an array nor an pointer), most of environments write to the same variable and we cannot change the order of execution of environments in most cases. we give up analyzing in this case.

If the external variable is a pointer, we have to determine if the variable points the same memory location on different environments. Generally this analysis is considered to be hard, but Matsumoto and Han[21] showed that in Pascal we can detect a tree or a linked list constructed with pointers statically form the program. This analysis is quite useful for the breadth-first vectorization which is often useful for the procedures to manipulate tree structures.

```
{ $fib_0 = fib_1 = 0,\ fib_n = fib_{n-1} + fib_{n-2}$ }
function fib(n : integer) : integer;
begin
    if n <= 1 then
        fib := 1
    else
        fib := fib(n-1)+fib(n-2)
end;
```

Fibonacci number (can be vectorized with the breadth-first method)

```
{ $A(0,y) = y+1,\ A(x,0) = A(x-1,1),\ A(x,y) = A(x-1, A(x,y-1))$
}
function ack(x, y : integer) : integer;
begin
    if (x=0) then
        ack := y+1
    else if (y=0) then
        ack := ack(x-1,1)
    else
        ack := ack(x-1,ack(x,y-1))
end
```

Ackermann function (cannot be vectorized with the breadth-first method)

Figure 3.32: Recursive procedures without writing external variables.

3.8.3 External array variables

When the external variable is an array, dependence analysis caused by the array is equivalent to analysis of subscripts of the array. When two or more instances of expression of the array subscripts take the same value, it means that the array element expressed by the value is referenced twice or more. If at least one of the references is an assignment, it means that there exists dependence caused by the array. In the case of **DO**-loops in FORTRAN, there are many studies for dependence analysis of array-elements, such as that by Banerjee[4].

But these studies are not directly applicable to the analysis for dependence between environments. Here we show the difficulty through showing an example. Assume that there is a loop as follows.

```
for i := 1 to 10 do
    A[2*i+1] := ··· ;
```

In this case, we never worry about the existence of dependence caused by the array A in the loop because the expression $2*i+1$ never take the same value in an execution of the loop. On the other hand, in the case of a recursive procedure as the following, the assignment may cause dependences, because the formal parameter i may take the same value more than once in a sequence of recursive invocations.

```
procedure F(i:integer);
begin
    A[2*i+1] := ···;
    if (i>1) then begin
        F(i-1);
        F(i-2)
    end
end;
```

Generally, it is hard to distinguish the existence of dependence caused by global array variables in recursive procedures because of the complexity of the behavior of values of the formal parameters. But in actual programs, since programmers cannot write down so complicated program such as behavior of the arguments is too hard to be understood, we believe that in not a few cases arrays in recursive procedures are used in the following form.

Recursive procedures are often useful to write a program for a divide-and-conquer algorithm[1]. In these procedures the following pattern of usage of arrays is often seen.

1. On each environment, for a certain given range of index [LB,UB] of array A, do some works.
2. Divide the range [LB,UB] into n sections i.e.
   $[\mathrm{LB}_1,\mathrm{UB}_1], [\mathrm{LB}_2,\mathrm{UB}_2], \ldots, [\mathrm{LB}_n,\mathrm{UB}_n]$
   where $\mathrm{LB} \le \mathrm{LB}_1,\ \mathrm{UB}_1 < \mathrm{LB}_2, \ldots, \mathrm{UB}_{n-1} < \mathrm{LB}_n,\ \mathrm{UB}_n \le \mathrm{UB}$.
3. Give each section to each child environment.
4. Do the above recursively.

Procedures for *quicksort*, *mergesort* are good examples of such procedures. Note that such procedures are suitable for the breadth-first vectorization because dependences exist only on ancestor-descendant relationship in the environment tree.

Generally, when a type-2 recursive procedure has at least one statement to write to an array, with the following algorithm we can judge if the procedure is suitable for the breadth-first vectorization or not. (see also Figure 3.33 and Figure 3.34.)

Step-1 Check dependences via arguments and return values between brothers environments, and if they do not exist, normalize the procedure as Figure 3.14.

Step-2 For each array used in the procedure, detect lower and upper bounds of indexes in reference. They are often expressed as functions of arguments as follows,

$f_{Arl}(\vec{a_f})$	Lower bound in reference of the index of array **A**
$f_{Aru}(\vec{a_f})$	Upper bound in reference of the index of array **A**
$f_{Aal}(\vec{a_f})$	Lower bound in assignment of the index of array **A**
$f_{Aau}(\vec{a_f})$	Upper bound in assignment of the index of array **A**

```
procedure msort(left, right: integer); { $\vec{a_f}$ = (left,right) }
var i, j, middle, k: integer;
begin
    if (left < right) then begin      { *1 }
        middle := (left+right) div 2; { *2 }
        msort(left,middle); msort(middle+1,right); {  *3 }
        i := left; j:=middle + 1;
        for k := left to right do begin
            if (i > middle)
               and (j <= right{ **}) then begin
                tmp[k] := A[j]; { *4 }
                j := j+1
            end else if (j > right)
                and (i <= middle{ **}) then begin
                tmp[k] := A[i]; { *5 }
                i := i+1
            end else if (A[i]<=A[j]) then begin
                tmp[k] := A[i]; { *6 }
                i := i+1
            else begin
                tmp[k] := A[j]; { *7 }
                j := j+1
            end
        end;
        for k := left to right do
            A[k] := tmp[k]        { *8 }
    end
end
```

Figure 3.33: Example for divide-and-conquer detection. (cont.)

- From lines labeled by {***1**} and {***2**}, we can get

$$left < middle < middle + 1 < right.$$

- From lines labeled by {***3**}, $\vec{a_1} = (left, middle)$, we can get

$$\vec{a_2} = (middle + 1, right).$$

- From lines labeled by {***4**},{***5**},{***6**} and {***7**}, we can get

$$f_{Arl}(left, right) = left, f_{Aru}(left, right) = right.$$

- From lines labeled by {***8**}, we can get

$$f_{Aal}(left, right) = left, f_{Aau}(left, right) = right.$$

- From the above, we can get

$$\begin{array}{lcl} [f_{Arl}(\vec{a_1}), f_{Aru}(\vec{a_1})] & = & [right, middle] \\ [f_{Arl}(\vec{a_2}), f_{Aru}(\vec{a_2})] & = & [middle+1, left] \\ \text{thus} & & \\ [f_{Arl}(\vec{a_1}), f_{Aru}(\vec{a_1})] \cap [f_{Arl}(\vec{a_2}), f_{Aru}(\vec{a_2})] & = & \phi \\ [f_{Arl}(\vec{a_1}), f_{Aru}(\vec{a_1})] \in [f_{Arl}(\vec{a_f}), f_{Aru}(\vec{a_f})] & & \\ [f_{Arl}(\vec{a_2}), f_{Aru}(\vec{a_2})] \in [f_{Arl}(\vec{a_f}), f_{Aru}(\vec{a_f})] & & \end{array}$$

and

$$f_{Arl}(\vec{a_f}), \leq f_{Aal}(\vec{a_f}) \leq f_{Aau}(\vec{a_f}) \leq f_{Aru}(\vec{a_f}).$$

Figure 3.34: Example for divide-and-conquer detection. (cont'd)

where $\vec{a_f}$ denotes formal arguments expressed as a vector. Then on each environment this procedure refers array **A** with a range of indexes $[f_{Arl}(\vec{a_f}), f_{Aru}(\vec{a_f})]$ and assigns the array with a range of indexes $[f_{Aal}(\vec{a_f}), f_{Aau}(\vec{a_f})]$.

Step-3 For all actual arguments $\vec{a_1}, \vec{a_2}, \ldots, \vec{a_n}$, get

$$[f_{Arl}(\vec{a_1}), f_{Aru}(\vec{a_1})], [f_{Arl}(\vec{a_2}), f_{Aru}(\vec{a_2})], \ldots, [f_{Arl}(\vec{a_n}), f_{Aru}(\vec{a_n})]$$

Step-4 Check the following conditions.

$$f_{Arl}(\vec{a_f}) \leq f_{Aal}(\vec{a_f}) \leq f_{Aau}(\vec{a_f}) \leq f_{Aru}(\vec{a_f})$$
$$[f_{Arl}(\vec{a_i}), f_{Aru}(\vec{a_i})] \cap [f_{Arl}(\vec{a_j}), f_{Aru}(\vec{a_j})] = \phi \text{ for all } i, j, i \neq j$$
$$[f_{Arl}(\vec{a_i}), f_{Aru}(\vec{a_i})] \in [f_{Arl}(\vec{a_f}), f_{Aru}(\vec{a_f})] = \phi \text{ for all } i$$

The second and third conditions guarantee that if two environment do not have ancestor-descendant relationship, they do not refer the same range of the index. Then with the first inequality, we can guarantee that each environment never refer any values which is assigned by the other environments except its ancestors or descendents. Hence if the above conditions are always satisfied for all arrays used in the procedure, we can judge that the procedure is suitable for the breadth-first vectorization. □

In Step4, to evaluate these inequalities, we have to analyze data and control flow of the procedure, and gather all possible informations. It is theoretically possible but not easy to do this automatically by a compiler in general cases. In the case of Figure 3.33, the compiler have to know the fact that the inequality $left \leq (left + right)/2 \leq right$ is always satisfied. Furthermore, conditions labeled by {**} are not necessary, but without these artificial conditions, it is not easy to detect the range of values of i and j automatically. This type of analysis still remains as our future work.

3.8.4 Additional Remarks

It is obvious that if the target procedure contains some statements to call the other procedures which have side effects, the latter procedures cause dependence and the target procedure is not suitable for the breadth-first vectorization.

3.9 The Breadth-first Parallelization

The breadth-first vectorization method stands on a strategy to detect the parallelism between environments, consequently it is also available for the other parallel architectures. Here we argue an implementation for PVP machines.

It is obvious that when a loop has no loop-carried dependence and is vectorizable, the loop can be converted into a doubly nested loop where the outer loop can be parallelized and where the inner loop can be vectorized. In the case of a recursive procedure vectorized with breadth-first parallel execution, most part of the vectorizable loops can be executed similarly, but there is a small problem in memory management. As mentioned, we used "vector load expanding" and "vector store compressing" instructions to avoid fragmentation of the storage, but these instructions cannot be executed in parallel.

To solve this problem, we propose an alternative way as shown in Figure 3.35. Here we use instructions called "**vector element sum**"[4] to count the number of the child environments. This instruction can be executed in parallel, and after that, vector load expanding and store compressing instructions can be executed in parallel. The vector sum operation is a rather fast operation within the vector macro operations, and therefore the overhead of this instruction is considered to be small. However, each sum of child environments has to be broadcasted to all processors and the overhead of this communication is not not to be ignored on some architectures.

As another solution, we can also consider to divide the environment tree into subtrees. Since these subtrees are considered to be independent, we can assign each subtree to each processor. In this case, memory managements are done separately on each processor. The main benefit of this method is that each processor do not have to do any synchronization until the whole execution of the subtree is done. On occasions, by some user-directives and so on, if we can know that the shape of the environment is balanced, the processors work effectively in parallel. But on the contrary, the numbers of nodes of each subtree are

[4]This is a term of S-820/3800. In VP-400/2600 and SX-2/3, the instruction is called "vector sum".

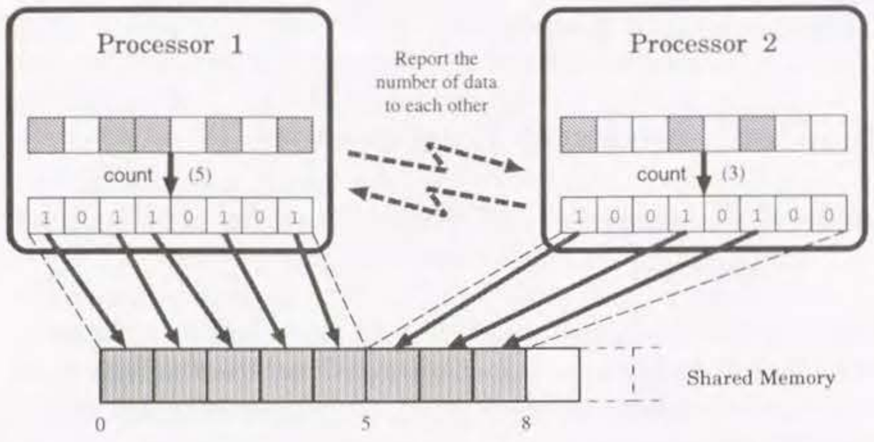


Figure 3.35: A technique to execute vector load expanding and vector store compression in parallel.

```
function Nqueens( i:integer; X,A,B,C : table; var ANS : table-array ) :
integer;
var j,k:integer;
    AA,BB,CC,XX : table;
begin
    k := 0;
    for j := 1 to N do
        if A[j] and B[i+j] and C[i-j] then begin
            AA := A; BB := B; CC := C; XX := X;
            XX[i] := j;
            AA[j] := false;  BB[i+j] := false;  CC[i-j] := false;
            if i<N then
                k := k+Nqueens(i+1,XX,AA,BB,CC,ANS)
            else begin
                k := k+1;
                ANS[k] := XX[i] { Save the answer }
            end
        end
    end;
    Nqueens := k;
end;
```

Figure 3.36: N-queen for the breadth-first vectorization

not always balanced therefore some kind of job-scheduling technique is required. It is left as our future work.

3.10 Performance Evaluation

3.10.1 Opportunities of vectorization and parallelization

As described in the former sections, the target procedure of the breadth-first vectorization must be a type-2 or type-3 procedure. Moreover, there must not be any dependence between environments except ancestor-descendant relationship.

For example, most procedures based on divide-and-conquer algorithms often have no dependence to prevent the breadth-first vectorization or parallelization. We have a conjecture that if a procedure is classified into type-2 or type-3 procedures and if the actual arguments do not depend on any of return values of the recursive calls and any values of elements of external arrays, we can vectorize the procedure with the breadth-first method. It is not easy to implement automatic detection of the dependence via external arrays. For example, procedures to solve **N-queen** problem are natively written as in Figure 3.3, and it is hard to analyze the dependence. But if it is written as in Figure 3.36, compliers can easily detect independency and users will enjoy its high acceleration.

3.10.2 Execution speed

As described in the previous chapter, execution speed in vector supercomputers owes much to the vector length and the vectorization rate. The breadth-first vectorization can vectorize almost all of the statements in the procedure body if it is applicable, and we can get enough vectorization rate.

The vector length is just the same as the number of environments at the same depth of the tree. Accordingly it depends on the shape of environment tree. In the worst case, if the environment tree has only one environment on each depth, then the converted procedure would

be executed with the vector length of only 1 and this might cause not acceleration but deceleration. But in most cases, at an enough deep level of the environment tree, the number of parallel environments is enough large to expect acceleration of ten times or more.

When amount of operation on each environment is homogeneous and the shape of the environment tree has enough width, the breadth-first vectorization works successfully. In the case of executing procedures for divide-and-conquer algorithms, amount of the operations on shallower environments may larger than that of deeper environments on the tree. The number of shallower environments is generally less than that of deeper environments and it causes shorter vector length. In this cases, instructions on shallower environments are executed many times in shorter vector length. It means that we cannot expect much acceleration. But the execution speed is estimated to be faster than that of scalar execution because of the high vectorization ratio.

3.11 Related Works

Here we compare this study with related works.

There are some works to optimize recursive procedures. The tail recursion elimination is well known and widely used to optimize recursive procedures of which recursive calls arise only at the end of the procedure body. Because this method is easy to implement and always useful to eliminate unnecessary codes, it is actually used in various existent compilers, in particular LISP compilers[30]. The depth-first vectorization method includes the tail recursion elimination because a procedure with a tail recursion is considered to be in a special case of type-1 procedures, which have no S_{tail} instructions in Figure 3.8. We also argued the opportunities of vectorization while they are leave out of consideration in the tail recursion eliminations.

Optimizing recursive procedures are mainly studied on functional programming languages. There are some theoretical works to transform recursive procedures preserving their semantics and to obtain the optimal procedure. For example, Vuillemin[51] showed that assuming a simple LISP-like language we can eliminate redundant operations in recursive procedures automatically. For example, procedure *fib* in Figure 3.32 contains many redundant operations because the same value of the argument arises many times within an environment tree. Vuillemin proposed the way to transform the procedures to remove this redundancy. We have not treated such optimization in this thesis. But Vuillemin's work is considered to be hard to apply to the conventional programming languages such as Pascal.

According to a survey of existent vectorizing C compilers on Convex and Cray by Smith[38], neither of them seems to treat recursive procedures. The vectorization methods of this thesis are considered to be applicable to C.

Some studies have shown the efficient implementation of divide-and-conquer algorithms on vector or parallel processors. In particular, Kanada[18] showed a 'schema' to solve searching problems on vector supercomputers efficiently. Our work make it easy to write a program based on the schema. Gürsoy and Kalé[11] proposed to introduce a new programming structure to directly express the divide-and-conquer algorithms into C. But the compiler never vectorize or parallelize automatically.

3.12 Experimental Results

We are now implementing the depth-first vectorization and the breadth-first vectorization and parallelization on our compiler V-Pascal version 3. We have confirmed that the transformation of intermediate codes is done but we have not obtained the object code. We summarize the results of simulations with FORTRAN codes as the experimental results, and the results obtained by our previous version of compiler, V-Pascal Ver.2.

The depth-first method owes the acceleration to vector-macro instructions. In particular, the vector instruction to operate **First Order Iteration** in Figure 3.1 often appears in both of S_{head} and S_{tail} in Figure 3.8. The execution speed of this instruction is important to estimate the acceleration which will be obtained by the depth-first vectorization method. Table 3.2 shows the benchmarking results of the **VITRD** instruction, i.e. the instruction to operate First Order Iteration on S-820/80[19]. The results show that VITRD executes more

Table 3.2: Benchmarking results of VITRD instruction on S-820/80

```
      A(I) = 1
      DO 10 I = 1, N
        A(I+1) = 2 * A(I) + 1
   10 CONTINUE
```

length of the loop	16	32	64	128	256	512	1024	2048
time for scalar execution(S)(μ sec.)	2.32	4.50	8.85	17.6	35.0	69.8	141	282
time for vector execution(V)(μ sec.)	1.48	2.11	3.25	5.53	10.2	19.4	20.7	39.9
VPU time in $V\mu$ sec.	1.02	1.60	2.75	5.05	9.66	18.9	20.0	39.2
S/V	1.58	2.14	2.72	3.18	3.43	3.60	6.81	7.07

† VPU stands for *Vector Processing Unit*.

Table 3.3: Execution time of the procedure SUM.

```
function SUM(K: integer )
  begin
    if K = 0 then SUM := 0
             else SUM := SUM(K-1) * K
  end;
```

depth of recursion	16	32	64	128	256	512	1024	2048
time for scalar execution(S)(μ sec.)	4.72	8.70	16.7	32.6	64.4	129	257	514
time for vector execution(V)(μ sec.)	3.90	6.07	10.7	19.4	36.7	72.1	142	282
VPU time in V(μ sec.)	0.94	1.18	1.69	2.72	4.76	9.07	17.4	34.2
S/V	1.21	1.43	1.56	1.68	1.75	1.79	1.81	1.82
time for scalar execution on V-Pascal Ver.2 (μ sec.)	32.8	46.9	74.0	130	241	463	914	2120

than 5 times faster than the scalar instructions if the vector length is enough long.

Table 3.3 shows an example of type-1 procedures and its execution time on S-820/80. This procedure is vectorized by the depth-first method. The execution time with vector codes is obtained by simulations with FORTRAN codes. The execution time for scalar is done with V-Pascal version 2. It is estimated that after the implementation of optimization modules, V-Pascal version 3 generates almost the same code as that of the simulation. We also show the execution time of the procedure transformed by the depth-first method but forced to be executed with scalar instructions. The results show that the procedure is accelerated by the depth-first method even if the procedure is not vectorized. If the procedure is vectorized, we obtain much greater acceleration. The execution speed with V-Pascal version 2 is very slow because of the overheads of recursive calls. It follows that we can get large acceleration only by the transformation by the depth-first method.

Table 3.4 shows the results from execution of the procedure *lsum* in Figure 3.23 on S-820/80. Results after vectorization are obtained by FORTRAN simulations. When we use the procedure, the shape of the environment tree is the same as that of the tree constructed with the arrays *left* and *right*. We gave the following types of trees to the procedure on benchmarking.

Complete binary trees It is ideal for the acceleration because the number of environments of each depth is maximum. This number directly corresponds to the vector length when executed by the breadth-first vectorization.

Linear lists It is the worst case because the number of environment on the same depth is always 1.

Fibonacci trees[28] A sort of AVL tree[29]. This is considered to be an average case.

The results shows the improvement of the execution speed up to 50 times when the lengths of the vectors are enough long. Even if the lengths of the vectors are only one, the execution speed is not less than the half of that of the execution by scalar. This results is considered to be very good and the breadth-first vectorization method is worth implementing nevertheless its difficulty.

Table 3.5 shows the results from execution of the same procedure on S-3800/480[5]. Results after vectorization are obtained by our previous version of compieler, V-Pascal version 2. Compared with the results from FORTRAN simulation, the acceleration is not good as expected. One of the reasons is considered to be of the weakness of optimization on V-Pascal version 2 because it lacks the facility to obtain optimal scheduling of the vector pipelines. We are now implementing this

[5]Computer Centre, University of Tokyo.

Table 3.4: Execution time of the procedure in Figure 3.23 (Fortran simulation).

(a) The results when the environment tree is a complete binary tree

height of the environment tree	4	6	8	10	12	14	16
total number of environments	15	63	255	1023	4095	16383	65535
time for recursive execution(R)(μ sec.)	55	138	508	2029	8076	52775	255523
maximum vector length	8	32	128	512	2048	8192	32768
time for scalar execution(S)(μ sec.)	52	122	389	1615	7001	35912	220144
time for vector execution(V)(μ sec.)	55	72	91	123	242	1007	5081
VPU time in V(μ sec.)	16	26	37	61	172	926	4961
R/V	1.00	1.92	5.58	16.50	33.37	52.41	50.29

(b) The results when the environment tree has only one environment on each depth.

height of the environment tree	4	6	8	10	12	14	16
total number of environments	4	6	8	10	12	14	16
time for recursive execution(R)(μ sec.)	38	43	50	57	64	74	95
maximum vector length	1	1	1	1	1	1	1
time for scalar execution(S)(μ sec.)	39	45	51	59	74	104	137
time for vector execution(V)(μ sec.)	56	72	86	103	117	131	151
VPU time in V(μ sec.)	16	23	33	41	50	58	66
R/V	0.68	0.60	0.58	0.55	0.55	0.56	0.63

(c) The execution time when the form of the environment tree is the same as that of the procedure *fib* in Figure 3.32.

height of the environment tree	4	6	8	10	12	14	16
total number of environments	9	25	67	177	465	1219	3193
time for recursive execution(R)(μ sec.)	46	75	162	385	974	3990	13307
maximum vector length	4	8	22	52	128	326	772
time for vector execution(V)(μ sec.)	54	71	92	108	134	177	272
VPU time in V(μ sec.)	4	14	25	35	48	76	159
R/V	0.85	1.06	1.76	3.56	7.27	22.54	48.92

Benchmarked on S-820/80, Feb.1990

method on V-Pascal version 3 and still working for the improvement of the performence.

3.13 Conclusion

We have proposed a new method to vectorize and parallelize recursive procedures. We have shown that introducing a concept of dependence of environments, we can convert the procedures and make each set of independent environments to be executed in parallel. We named this method *the breadth-first method*. The breadth-first vectorization is applicable for recursive procedures which calls itself many times on each environment. This is also applicable for parallel architectures such as PVP machines. According to the experimental results, we can expect up to 50 times' acceleration by this method.

Our future work is as follows.

- It is required to gather highly detailed information from the procedure body for more strict or exact analysis of environments. Methods to detect lower and upper bounds of simple variables are especially required.
- Implementation of the breadth-first method soon with the above analysis.
- The breadth-first method is considered to be applicable not only for VP/PVP machines but also for MPP machines.
- These two methods are considered to be also applicable to other languages such as C and Fortran 90, which must be the next standard language for Supercomputing. We will propose definite methods to apply these methods for this language.

Table 3.5: Execution time of the procedure in Figure 3.23 (V-Pascal version 2).

total number of environments	31	63	127	255	511	1023	2047	4095	8191
time for scalar execution(S)	8.0	36.0	78.0	167	334	669	1343	2687	5371
time for vector execution(V)	59	69	83	105	126	162	231	344	592
(VPU time in V)	(29)	(36)	(45)	(56)	(73)	(102)	(157)	(270)	(505)
Speedup (S/V)	0.14	0.52	0.94	1.59	2.65	4.13	5.81	7.81	9.08

(a) The results when the environment tree is a complete binary tree.

total number of environments	32	64	128	256	512	1024	2048	4096	8192
time for scalar execution(S)	19	40	78	183	319	637	1394	3599	8299
time for vector execution(V)	377	756	1528	3068	6147	12007	23777	47189	93809
(VPU time in V)	(190)	(363)	(732)	(1470)	(2947)	(5618)	(10920)	(21540)	(43960)
Deceleration (V/S)	19.83	18.90	19.59	16.77	19.27	18.85	17.06	13.11	11.30

(b) The results when the environment tree has only one environment on each depth.

All of execution times are in micro-seconds. Benchmarked on S-3800/480, Apr.1993.

Chapter 4

An Automatic Vectorizing/Parallelizing Pascal Complier V-Pascal Version3

4.1 Introduction

With the arrival of Fortran90, we can now utilize more versatile control and data structures than FORTRAN77 — for example, while loops, recursive calls and data structures constructed with pointers — in programs for numerical computations. Since these structures have been considered to be fairly helpful for writability and readability of programs, most of the application programs written in Fortran90 will contain these control/data structures in near future. Therefore automatic vectorization and parallelization techniques to deal with these structures have become a very important issue for supercompilers.

We have already proposed various vectorizing and parallelizing techniques for these versatile structures and some workbench is needed to verify the efficiency of the compilation methods. For this goal, we started to develop an automatic vectorizing and parallelizing compiler named "**V-Pascal**". We are now implementing many of our compilation techniques and realizing advanced vectorization and parallelization methods on the compiler.

This chapter describes the organization and main features of V-Pascal. Some of the implemented compilation techniques we have already proposed are also stated.

4.2 Overview

As you can see in the compiler's name, the target language chosen is Pascal. Pascal is a typical block-structured language which is widely spread for general purpose. It is equipped with more versatile control/data structures than FORTRAN77, which has been the only language for which supercomputer manufacturers have provided useful vectorizing/parallelizing compilers. To extend the horizon of vector supercomputer usage, we were interested in vectorizing/parallelizing programs that manipulate data structures other than arrays. Even in the presence of control structures such as while-loops and recursive calls, the programs should be vectorized/parallelized. The language Pascal is eligible for our purpose. Pascal has been used not only for educational purposes in classrooms but also for real-world problems, for example, the early TEXprocessor by D.E. Knuth, the vectorizing Fortran compiler construction by an American supercomputer manufacturer, and the implementation of an operating system for some Japanese minicomputers. Although the target language is Pascal, the various basic techniques that have been and will be developed for our compiler will also be useful for advanced vectorizing/parallelizing FORTRAN77 and Fortran90 compilers.

Toward "fully automatic" vectorization/parallelization for sequential programs, basically we have added no language extensions to the syntax of Pascal to express any parallelism. Therefore, the target programs for V-Pascal are exactly written in traditional sequential manner. But in some cases parallelism in programs can hardly be detected by the compilers. To cope with this problem, we introduced compiler directive facilities which are to use comment lines to indicate data-dependence information in programs. This style of solution are often seen in FORTRAN77 compilers provided by manufactures and these directive facilities can be regarded as language extensions in a sense.

But users can still write most part of programs in sequential manner and it will greatly helps to utilize vector and parallel supercomputers.

There have been four versions of V-Pascal : Versions 1.0, 1.5, 2.0, and 3. The first three will be called "early" versions. The early versions mainly aimed at the study of advanced vectorization techniques, while version 3, although dovetailing with the early versions in many aspects, is specifically meant for parallelization of Pascal programs for Japanese vector multiprocessors such as Hitachi's S-3800 and highly parallel machines like Fujitsu AP1000.

This chapter gives only about the most recent version, V-Pascal version 3. The details of the early versions were described in [43] and [45].

4.3 Organization of the Compiler

Figure 4.1 gives the organization of V-Pascal version 3 which is now being constructed. It consists of three phases and the second phase can be also broken into small analyzing and optimizing modules.

The first phase, Phase 1, parses the source program and performs syntactic and semantic analysis. After the process, the program will be converted into intermediate code which is independent of the target language. Phase 1 is operational only for Pascal at the moment, but we can support the other languages by constructing Phase 1 for them.

The second phase, Phase 2, is the optimization phase which receives sequential intermediate code from Phase 1 and outputs optimized, vectorized and parallelized intermediate code. This phase consists of a number of small optimization modules which are the implementations of vectorization/parallelization methods we have proposed. All of the modules has an uniform input/output interface; they work as filter programs to receive and output intermediate code which has the unified format. This allows to reconfigure the compiler and we can easily add new optimizing facilities to it. To support analysis performed on each optimizing module, basic analyzing modules to carry out alias analysis, data-flow analysis, control-flow analysis and data dependence analysis between array elements are also included in this phase.

The last phase, Phase 3, converts machine-independent intermedi-

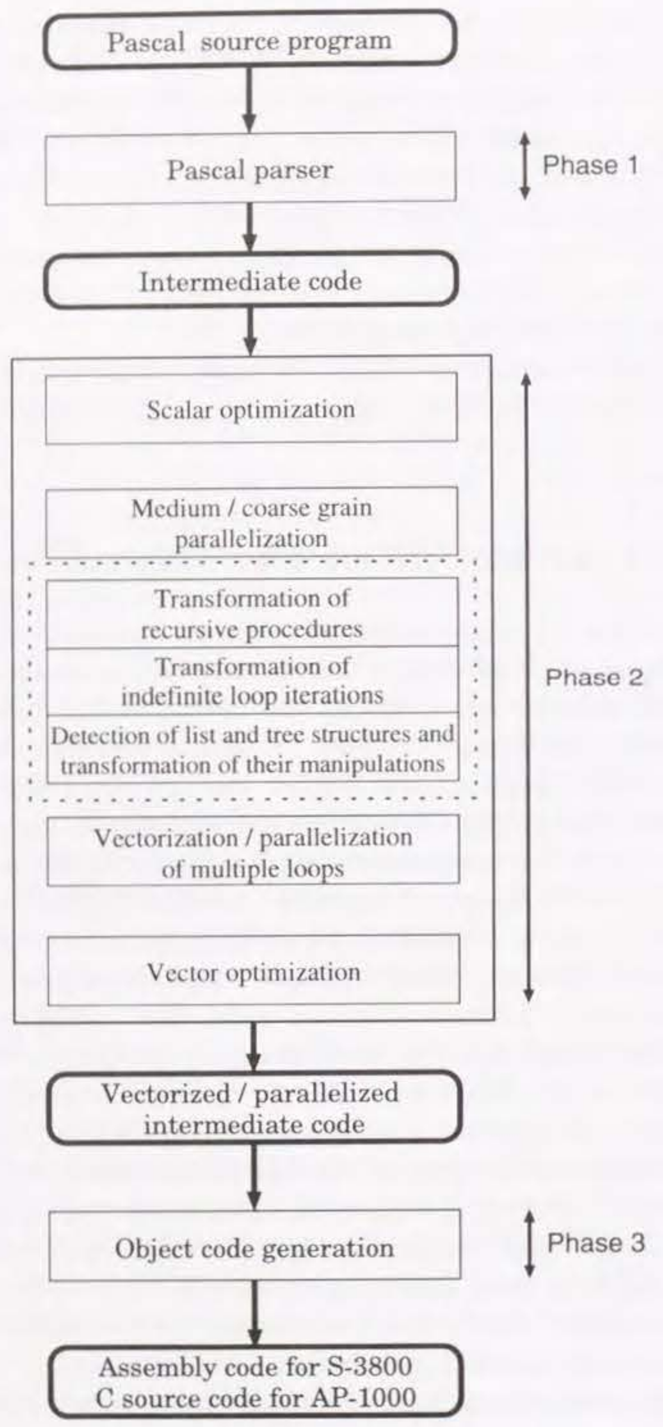

Pascal source program
Pascal parser
Phase 1
Intermediate code
Scalar optimization
Medium / coarse grain parallelization
Transformation of recursive procedures
Transformation of indefinite loop iterations
Detection of list and tree structures and transformation of their manipulations
Vectorization / parallelization of multiple loops
Vector optimization
Phase 2
Vectorized / parallelized intermediate code
Object code generation
Phase 3
Assembly code for S-3800
C source code for AP-1000

Figure 4.1: The organization of V-Pascal Version 3

ate code to machine-dependent execution code. Machine-dependent optimization is also performed in this phase. Now two versions of Phase 3 are operational, one of which generates vectorized/parallelized assembly code for S-3800 and the other yields parallelized C source code for AP1000.

As described in above, intermediate code is used as a unified input/output data format over all phases and modules in Phase 2. The design goal of intermediate code is as follows.

- Scalar instructions are simple instructions having load-store RISC-style architecture with unlimited number of registers. Vector instructions are also simple instructions of a typical Cray-style vector processor which has unlimited number of vector registers with infinite length. The vector-macro operations are also defined to cover all of vector-macro instruction sets of supercomputers which are provided by domestic manufacturers.

- Parallel execution primitives are defined in general terms for tightly coupled vector multiprocessors. Differences between target machines are absorbed by run-time libraries.

- Control structures are carefully designed. In addition to **for**-loops, **forever**-loops with exit instructions from the middle of the loops are defined to express **while**-loops, **repeat-until** loops and loops implicitly constructed by **if-goto** statements. Although these structures can also be expressed implicitly by **goto**-statement of intermediate code, we added these control structures to direct which part should be vectorized/parallelized in the intermediate code. Special branch nodes are defined so that multiple alternatively optimized codes for the same instruction sequence can be retained until the choice can be made later in Phase 3. At the moment, these nodes are implemented as normal conditional-branch nodes with special conditions which denote that either of the following paths should be eliminated in Phase 3.

V-Pascal Ver.3 is implemented in C++ while the early versions are written in Pascal. The main reason of this changeover is that we found that the object-oriented approach is fairly helpful for us to handle quite complicated data structures used in compilers. In particular, many graph-shaped data structures are used in our compiler and we had to write a sequence of pointer manipulation statements to handle them in Pascal, but using classes of C++, we can neatly express the sequence in a simple manner. The other reasons to use C++ are that we wanted to use various utilities to help the development, such as compiler-compilers (lex and yacc) and useful source-code-level debugger (gdb) which cannot be found for Pascal.

4.4 Facilities of Analysis

Each of the modules in Phase 2 calls various common analyzing modules which help optimization, vectorization and parallelization. This section describes the analyzing facilities with which V-Pascal version 3 is equipped.

4.4.1 Alias Analysis

When two different variables refer to the same location of the memory, each of them is called an alias of the other and they are called an alias pair. In cases alias pairs may exist, an assignment of a variable may change value of the other variable and it causes misjudgments in data-flow analysis. Therefore, for precise data-flow analysis, we have to determine all possible alias pairs. In other words, precise alias analysis is indispensable before data-flow analysis.

In Pascal, there are two cases which cause aliases. One of them is the case caused by call-by-reference parameters. When a function or procedure has two or more **var** arguments and actual parameters for them are the same, these parameters are an alias pair, as shown in Figure 4.2 for example. There are many methods to analyze this kind of aliases, and Cooper[6] proposed an algorithm for precise analysis of this problem. We implemented the algorithm on our compiler.

The other case, in which an alias occurs in Pascal, is by pointer references. When two of pointer variables have the same content as shown in Figure 4.3, they point the same location of the memory and that causes aliases. Since the values of these pointer variables cannot

```
procedure F(var a,b : integer);
   begin
      ...
   end;
begin
   ...
   F(c,c); { (a,b) is an alias pair }
   ...
end;
```

Figure 4.2: An example of alias pairs caused by call-by-reference parameters

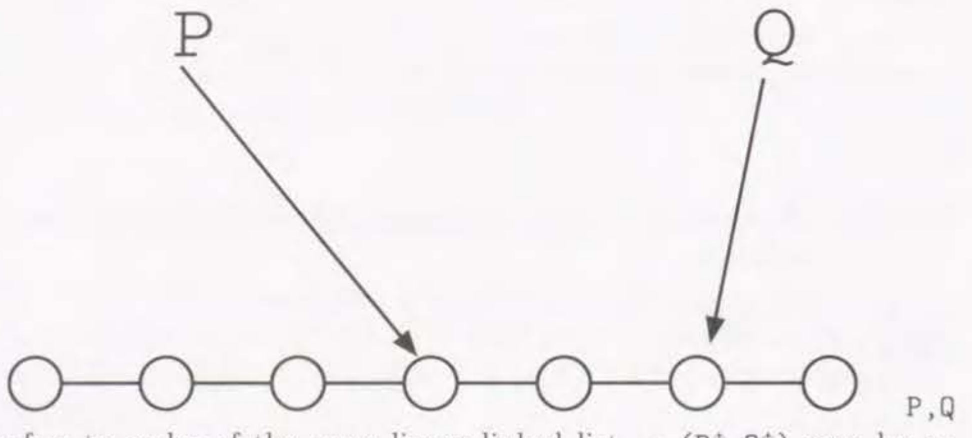
```
var p,q : ↑integer;
begin
   ...
   new(p);
   q := p; { (p↑,q↑) is an alias pair }
   ...
end;
```

Figure 4.3: An example of alias pairs caused by pointer references

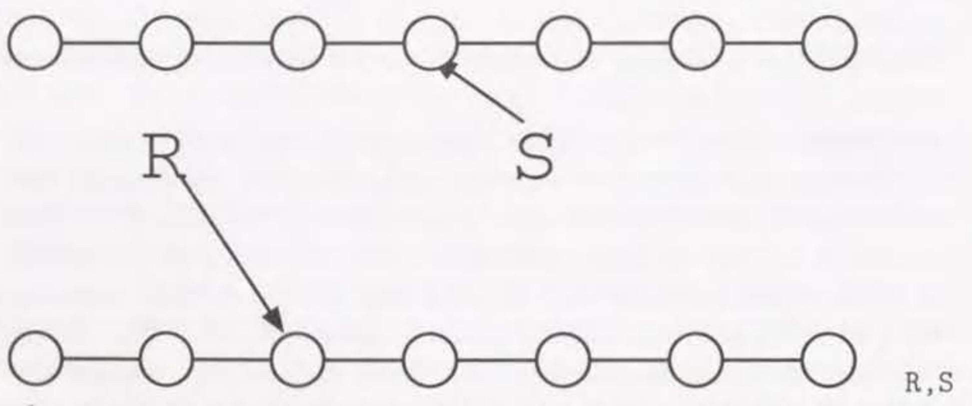
be estimated in most cases, there have been few methods to overcome this problem and almost all existing compilers give up analysis and assume that all possible pairs of such variables may be aliases. We have proposed a method to detect data structures such as linear linked lists and trees at compile time and through this process we can eliminate some possibilities of existence of pointer-aliases precisely. The details of the algorithm are described in [21]. After the analysis, we can also detect pairs of pointer variables which always refer to different data structure (for example, as R and S in Figure 4.4), and which cannot be an alias pair.

4.4.2 Control/Data-flow Analysis

Control and data-flow analysis in V-Pascal version 3 is performed by a traditional method, but the result is more precise than others because



refers to nodes of the same linear linked list → (P↑,Q↑) may be an alias pair.



refers to nodes of different linear linked lists. → (R↑,S↑) cannot be an alias pair.

Figure 4.4: An example of the detection of aliases of pointer variables

of the existence of advanced alias analysis facilities.

The results are represented in **D-matrix**[43]. It takes a hierarchical structure so that dependences between procedures/functions, those between loops plus basic blocks, and those between intermediate code statements are represented by the D-matrix. In other words, the parallelism of various granularities are designated by the same data structure. An example is shown in Figure 4.5.

4.4.3 Value-Range Estimation of Integer Variables

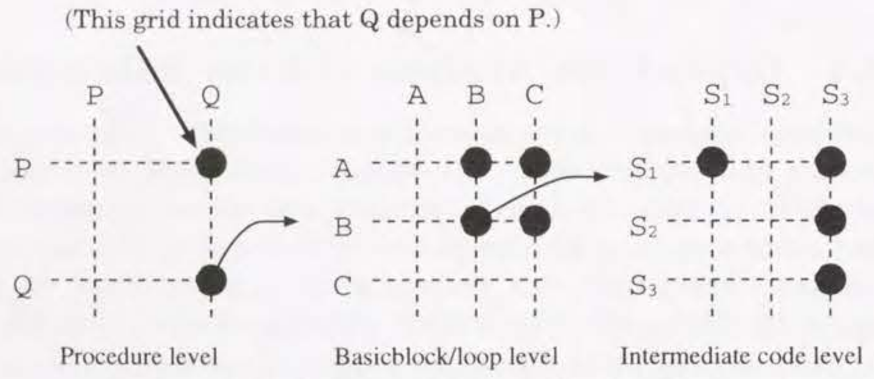
It is a well-known problem that dependence analysis of arrays is obstructed with the existence of an array subscript expression which contains some variables whose values cannot be determined statically. Therefore it is important to estimate the value range of an array subscript expression at each step of execution to raise the compiler's power for dependence analysis for vectorization and parallelization. This information acquisition is done with data-flow analysis, and the relevant information is appended to integer variables and temporaries of the intermediate-code statements. The detail will be described by a paper in preparation.

4.4.4 Dependence Analysis of Array Subscripts

Dependence analysis of array subscripts is essential for automatic vectorization and parallelization. There have been many proposals of methods to perform this analysis precisely and efficiently; one of the most famous method is the Omega test by W.Pugh[36]. We also have an original approach for this analysis which is implemented on our compiler[23]. Our method can analyze nested loops accurately, and in most cases, in polynomial time of the number of variables. When loop bounds and array subscripts are linear expressions of the surrounding loop control variables, our method is exact, i.e., the power of analysis is as same as that of Omega-test. When loops have symbolics, our method tries to find the range of the symbolics that may cause dependences preventing vectorization or parallelization, so compilers can produce conditions to judge the possibility of vector or parallel execution at the execution time. We also proposed a method to analyze

```
procedure P;
    procedure Q;
    begin
        A;
        for ... do begin { = block B}
            S₁;
            S₂;
            S₃;
        end;
        C;
    end;
begin
    Q;
end;
```

(a) A sample program.



(b) A sample D-matrix for (a)

Figure 4.5: An example of D-Matrix

loops with loop bounds and array subscripts which contain non-linear expressions.

The outline of our method is as follows. The dependence analysis of arrays is equivalent to the problem of obtaining integer solutions of simultaneous Diophantine equations and inequalities. We can easily check the existence of real solutions by integer programming which takes only the polynomial time of the number of the variables, and after that we obtain a convex hull which expresses a space of the real solution and may contain integer solutions. Then we perform exhaustive search of an integer solution around each apex of the convex. In the worst cases, this process takes the exponential time in the number of variables. But we think that it is not a disadvantage of our method. The reason is shown using Figure 4.6. This is a typical solution space that both integer solutions and real solutions exist. As shown in the figure, in most cases when a real solution exists, integer solutions also exist, and at least one of them is expected to be at near an apex of the convex (P in the figure), or the apex is an integer solution itself (Q). These solutions will be detected in short time of which the order is about the polynomial time of the number of the apexes. Moreover, even when no integer solutions exist while the real solution exists, which is a rare case, we can also expect that the convex is small enough to check all of possible integer solutions quickly.

4.5 Facilities of Vectorization/Parallelization

As mentioned before, Phase 2 of V-Pascal version 3 consists of many small optimizing modules which realize our proposed parallelizing/vectorizing methods. This section describes some of the methods we proposed and implemented.

4.5.1 Vectorization/Parallelization of Multiply-nested Loops

Most of the compilers offered by supercomputer manufacturers can only vectorize inner-most loops of multiply-nested loops, and parallelize the outer loops. But we have already proposed a method to

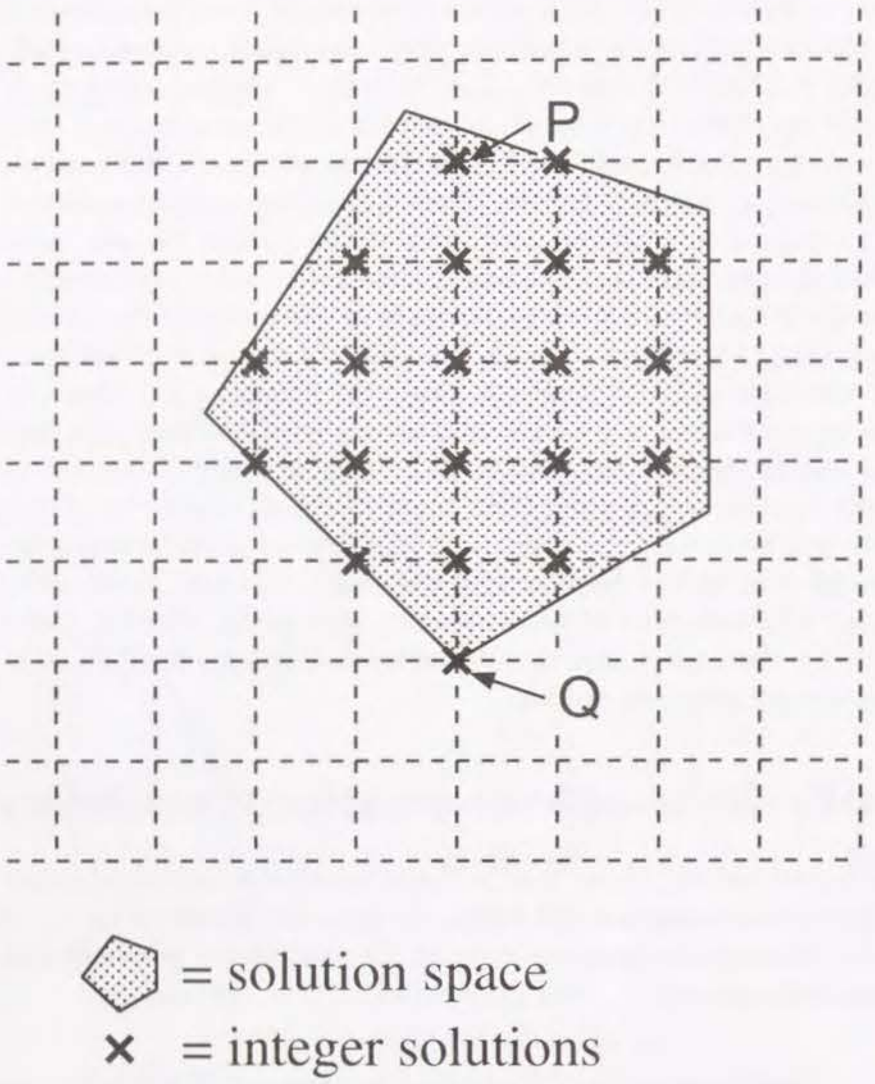


Figure 4.6: An example of solution space

```
          for i:=1 to 5
            for j:=1 to i
              ..a[i,j]..

                 ⇓reduce
    vi[] :=
{1,2,2,3,3,3,4,4,4,4,5,5,5,5,5};
    vj[] :=
{1,1,2,1,2,3,1,2,3,4,1,2,3,4,5};
    for k:=1 to 15
       ..a[ vi[k], vj[k] ]..
```

Figure 4.7: Reduction of a multiply-nested loop into a single loop

reduce multiply-nested **for**-loops into equivalent single loops which are then vectorized by extensive use of vector indirect addressing [43]. An example is shown in Figure 4.7. As we have shown in [47], recent high-speed vector supercomputers tend to have large N_{half}[14] values, therefore the reduction is efficient to make vectors longer and obtain higher performance.

4.5.2 Vectorization of While Loops

While traditional FORTRAN 77 has only `DO`-loops to express loop structures, Pascal has not only **for-** loops but also **while** and **repeat-until** loops. Since we cannot determine the number of iteration for **while** and **repeat-until** loops (therefore call them *indefinite loops*) even at the entrance of the loop in execution — in other words, we cannot determine the vector length even in execution time — we have to make some tricks to vectorize these loops. Wolfe proposed two approaches to cope with this problem : one is called loop distribution and the other is strip-mining [52]. We have proposed a hybrid method of them to treat these loops more neatly[39] and obtained high speedups for more generalized cases. Wolfe also discussed how to vectorize nested loops of which the inner loop is an indefinite loop and the outer loops is a **for**-loop, by interchanging these loops and vectorizing only the **for** loop, but this method is not efficient when the vector lengths are too short. We also proposed a method to cope with this problem interchanging these loops dynamically to keep the vector length long enough [24].

4.5.3 Vectorization and Parallelization of Recursive Procedures

As shown in the previous chapter, we have proposed a method to vectorize and parallelize recursive procedures, named *breadth-first method.* We are working to implement this method with another method, *depth-first method* mentioned also in the previous chapter.

4.6 Conclusion

An overview was given of the current status of the V-Pascal version 3 compiler. In the V-Pascal Version 3 compiler, special techniques have been developed even for Algol-like features, so that they may also be useful for the full-fledged vectorizing/parallelizing compilers for the languages Fortran 90 and C. Although the initial target machine has been a tightly coupled vector multiprocessor like Hitachi's S-3800, a slight reconfiguration of compiler components allows parallelization for distributed-memory parallel computers. To demonstrate this, we have provided a compiler called **V-Pascal/DM** [50] for Fujitsu AP1000, to which the compiler output is fed in the form of parallelized C source code. V-Pascal/DM performs data-parallel execution based on an SPMD model.

Chapter 5

Conclusion

In this thesis, we have discussed the construction of an automatic vectorizing and parallelizing compiler for a block-structured language Pascal to utilize VP and PVP supercomputers for non-numerical applications.

In Chapter 2, the performance of load/store instructions on each VP/PVP supercomputer was evaluated. In particular, the performance of vector indirect load/store instructions was precisely discussed. Since the performance of indirect access dominates the performance not only of numerical applications but of non-numerical applications which treat pointers, the performance evaluation of indirect load/store instructions is very important to construct a compiler for a general-purpose language. We have developed a benchmarking program for this purpose and obtained interesting results. According to the results, some machines such as S-820/3800 showed good performance in indirect access even when the list vector indicates irregular access of the memory. These machines are considered to be effective for running non-numerical applications written in block-structured languages.

In Chapter 3, we proposed a new method to vectorize and parallelize recursive procedures. Since recursive procedures are often seen in programs written in block-structured languages, this control should be also regarded as targets for automatic vectorization and parallelization. But there have been few researches for this subject. We have proposed the breadth-first method and have shown that some recursive procedures will be greatly accelerated by this method. We have also pointed out the difficulty to distinguish the recursive procedures which can be applied the breadth-first method, and discussed some ways to cope with this problem.

In Chapter 4, we presented the overview of our automatic vectorizing and parallelizing compiler V-Pascal version 3. n the V-Pascal Version 3 compiler, various analyzing, vectorizing and parallelizing techniques have been developed even for Algol-like features, so that they may also be useful for the full-fledged vectorizing/parallelizing compilers for the languages Fortran 90 and C.

Future works and open questions of this study is as follows:

- With the recent advance of device technologies, the execution speed of processors has been rapidly progressed. This progress is said to last for the next ten years or more. Contrarily, the performance of memory devices will not be improved so rapidly. As the result, the performance of processors will become not to match with that of main memory in near future. The author believes that the vector load/store architecture will be an answer to cope with this problem. If this conjecture is true, many techniques obtained throughout this research will become useful. Whether this conjecture is true or not is remained as an open question.
- We have limited the application of the breadth-first method to certain patterns of recursive procedures. Our future work is to relax this restriction as weak as possible.
- To accomplish the implementation of the compiler is still left as a future work.

Bibliography

[1] Aho, A.V., Hopcroft, J.E. and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[2] Aho, A.V., Sethi, R. and Ullman, J.D., *Compilers—Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[3] Backus, J., *From Functional Semantics to Program Transformation and Optimization*, Proc. of the International Joint Conference on Theory and Practice of Software Developement, Springer-Verlag, pp.60–91, 1985.

[4] Banerjee, U., *Dependence Analysis for Supercomputing*, Kluwer Academic Publisher, 1988.

[5] Convex Corp., World Wide Web document `http://www.convex.com`, Jul 1995.

[6] Cooper, K.D., *Analyzing Aliases of Reference Formal Parameters*, Conf. Rec. Twelfth ACM Symposium on Principles of Programming Languages(Jan.), pp. 281-290. 1985.

[7] Cray Research Inc., *Cray Y-MP Manual : CF77/Standard C Features and Optimization Rev.D.*

[8] Cray Research Inc., World Wide Web document `http://www.cray.com`, Jul 1995.

[9] Flynn, M.J., *Some computer organizations and their effectiveness*, IEEE Trans. Comput., C-21, pp.948-960, 1972.

[10] Fujitsu Ltd., *FACOM VP series Manual of the Hardware Facilities (the 4th edition)*, 1987. (in Japanese)

[11] Gursoy, A. and Kalé, L.V., *High Level Support for Divide-and-Conquer Parallelism*, Proceedings Supercomputing '91, ACM/IEEE, pp.283–292, Nov. 1991.

[12] Hayama, S., *Automatic Vectorization of Nested Loops with Complex Control Structures*, Master thesis, Dept. of Information Science, Kyoto University, 1987. (in Japanese)

[13] Hitachi, Ltd., *HITAC S-820 Processing Unit Manual*, 1988. (in Japanese)

[14] Hockney, R. W. and Jesshope, C. R., *Parallel Computers 2, Adam Hilger*, 1988.

[15] Hockney, R.W. and Jesshope, C.R., *Parallel Computers 2*, IOP Publishing Ltd., 1988.

[16] IBM Corp., *SP2 system architecture*, IBM Systems Journal, Vol.34, No.2 pp.152-184, 1995.

[17] Intel Corp., World Wide Web document `http://www.intel.com`, Jul 1995.

[18] Kanada, Y., Keiji, K., and Masahiro, S., *A Vector-Processor-Oriented Schema for for Solving Searching Problems: Parallel Backtracking Schema*, Trans. IPS Japan, Vol.29, No.10, pp.985–994, 1988. (in Japanese)

[19] Kozuka, H., *Automatic Vectorization of Recursive Procedure in Language Pascal*, Master thesis, Dept. of Information Science, Kyoto University, 1989. (in Japanese)

[20] Kunieda, Y., *Study on An Automatic Vectorizing Pascal Compiler*, Doctoral Thesis, Kyoto University, Nov.1990. (in Japanese)

[21] Matsumoto, A., Han D.S., and Tsuda, T., *Alias Analysis of Pointers in Pascal and Fortran 90, Part I. Dependence Analysis between Pointer References*, in press for Acta Informatica, 1995.

[22] Matsumoto, H. et al., *Hardware of SX-3 Series*, NEC Technical Journal, Vol.45, No.2, pp.15–27, 1992. (in Japanese)

[23] Mizunuma, I., Uehara, T., Okabe, Y., Kunieda, Y., and Tsuda, T., *Data-Dependence Analysis of Nested Loops Containing Symbolics and Nonlinear Expressions,* Proc. of the 9th National Convention of Japan Society for Software Science and Technology (Fujisawa, Sept. 1992), pp. 485-488, 1992. (in Japanese)

[24] Murai, H., Suehiro, K., Okabe, Y., Kunieda, K. and Tsuda, T., *Vectorizing while loops by loop interchange*, Proc. of the 11th National Convention of Japan Society for Software Science and Technology (Osaka, Mar. 1994), pp.65–68, 1994 (in Japanese).

[25] Murayama, H., Fukuta, K., and Yamada, M., *Hitachi S-3000 Series Supercomputer, Its Operationg Systems and Hardware*, THE HITACHI HYORON, Vol.75, No.5 pp.15–20, 1993. (in Japanese)

[26] NEC Corp., *NEC Supercompter SX-1/SX-2 Manual of the Computational Processor Facilities*, 1985. (in Japanese)

[27] NEC Corp., World Wide Web document `http://www.nec.co.jp`, Jul 1995.

[28] Nagao, , M., et.al., *Iwanami Information Science Dictionary*, Iwanami Shoten, pp.631, 1990. (in Japanese)

[29] Nagao, M., et.al., *Iwanami Information Science Dictionary*, Iwanami Shoten, pp.55, 1990. (in Japanese)

[30] Nagao, M., et.al., *Iwanami Information Science Dictionary*, Iwanami Shoten, pp.727, 1990. (in Japanese)

[31] Nakamura, M. and Tsuda, T., *A Fast Algorithm for First Order Recurrences on Vector Supercomputers*, Transactios of Information Processing Society of Japan, Vol.36, No.3, pp. 669–680, 1995 (in Japanese).

[32] Nakamura, M. and Tsuda, T., *Methods for Idiom Recognition by Automatic Vectorizing Compilers*, Trans. of Information Processing Society of Japan, Vol.32, No.4. pp.491–503, 1991. (in Japanese).

[33] Nakamura, M., *Automatic Vectorization of Operations Which Have Recursive References*, Master thesis, Dept. of Information Science, Kyoto University, 1991. (in Japanese)

[34] Nakata, H., *Automatic Vectorization of Complicated Nested Loops with Conditional Branches*, Master thesis, Dept. of Information Science, Kyoto University, 1989. (in Japanese)

[35] Nikkei Bussiness Publications, Inc., *Supercomputing '94 report*, NIKKEI-ELECTRONICS, No.624, pp.19-20, 1994.

[36] Pugh, W., *The Omega Test : a fast and practical integer programming algorithm for dependence analysis*, Proceedings of Supercomputing '91, pp.4–13, 1991.

[37] Shimasaki, M. *Supercomputers and Programming*, Kyoritsu Shuppan, 1989. (in Japanese)

[38] Smith, L.L., *Vectorizing C Compilers : How good are they?*, Proceedings Supercomputing '91, ACM/IEEE, pp.544–533, Nov. 1991.

[39] Suehiro, K. and Tsuda, T., *Automatic Vectorization / Parallelization of WHILE Loops*, Proc. 45th Annual Convention IPS Japan (Tokushima, Nov. 1992), pp. 5-51 & 52, 1992 (in Japanese).

[40] Suehiro, K., *Automatic vectorization of while loops*, Master thesis, Dept. of Information Science, Kyoto University, 1990. (in Japanese)

[41] Thinking Machine Corp., World Wide Web document `http://www.think.com`, Jul 1995.

[42] Tsuda, T. and Kunieda, Y., *Mechanical vectorization of multiply nested DO loops by vector indirecte addressing*, Aspects of Computation on Asynchronous Parallel Processors, Proc. IFIP WG2.5 Working Conf. (Stanford CA, Aug. 1988), Elsevier Science Publisher, North-Holland, pp.101–110, 1989.

[43] Tsuda, T. and Kunieda,Y., *V-Pascal : An Automatic Vectorizing Compiler for Pascal with No Language Extensions*, The Journal of Supercomputing(Kluwer Academic Publishers), vol.4, pp.251-275, 1990.

[44] Tsuda, T. *Numerical Computation Programming*, Iwanami Shoten, 1988. (in Japanese)

[45] Tsuda,T., *Design and Implementation of a Vectorizing Compiler for the Block-Structured Language Pascal*, Supercomputer 46 (VIII-6): 12-21, 1991.

[46] Uchida, N. et al., *System Overview of FUJITSU VP2000 Series*, FUJITSU Science and Technical Journal, Vol.27, No.2, pp.149–157, 1991.

[47] Uehara, T. and Tsuda, T., *Benchmarking Vector Indirect Load/Store Instructions*, Workshop on Benchmarking and Performance Evalutation in High Performance Computing (Tokyo, Japan), pp.16–25, 1993.

[48] Uehara, T. and Tsuda, T., *Benchmarking Vector Indirect Load/Store Instructions*, Supercomputer, ASFRA, The Netherlands, Vol.8, No.48, pp.57-74, 1991.

[49] Uehara, T., *Automatic Vectorization of Recursive Procedures*, Report of graduation research work, Dept. of Information Science, Kyoto University, 1990. (in Japanese)

[50] Umeda, K., Uehara, T. and Tsuda, T., *An Automatic Parallelizing Compiler for Distributed Memory Parallel Computer V-Pascal/DM*, Proc. 48th Annual Convention IPS Japan (Kashiwa, March 1994), paper 5G-1, 1994. (in Japanese)

[51] Vuillemin,J. : *Correct and Optimal Implementations of Recursion in a Simple Programming Language*, Journal of Computer and System Sciences, Vol.9 pp.332–354, 1974.

[52] Wolfe, M., *Optimizing Supercompilers for Supercomputers*, MIT Press, 1989.

[53] Yabuuchi, K., *Mechanical Vectorization of Nested Loops Based on Data Reference Relations*, Master thesis, Dept. of Information Science, Kyoto University, 1987. (in Japanese)

[54] Zima, H., *Supercomilers for Parallel and Vector Computers*, Addison Wesley, 1991.

Acknowledgements

The author would like to express my sincere gratitude to the supervisor, Professor Takao Tsuda for giving me the oppotunity of this study, and also for his continuous guidance and encouragements during all the time of this study.

The author also would like to appreciate to Dr. Yoshitoshi Kunieda and Dr. Yasuo Okabe for their appropriate suggestions, advice and helpful discussions. Thanks are also due to Professor. Eiji Okubo of Ritsumeikan University for his encouradgements and comments.

The author is thankful to all the members of Tsuda laboratory for their comments and supports throughout this research. This work would never been accomplished without the related studies throughout our V-Pascal project.

Again, I would like to express my gtatitude to all of the supporters in my mother language.

本研究の機会を与えて頂き、研究の遂行に際し終始丁寧なご指導ご鞭撻を賜りました恩師、京都大学工学研究科教授 津田孝夫先生に深甚なる謝意を表します。先生のご指導なしには本日の研究者としての私はありませんでした。

本研究に際しましてさまざまなご助言を賜り、熱心にご討論頂きました京都大学工学研究科助教授 國枝義敏先生、京都大学大型計算機センター助教授 岡部寿男先生に心より感謝いたします。先生方には研究の細部に渡るまでご討論頂き、また特に國枝先生には幅優先法の実装作業におきまして絶大なるご助力を頂きました。

また、本研究の遂行にあたりましてご助言を頂きました立命館大学理工学部教授 大久保英嗣先生に深く感謝致します。

本研究は、津田研究室で過去に開発された V-Pascal コンパイラにおける研究を基に発展させたものです。過去の研究に従事された津田研究室の諸先輩に感謝と敬意を表します。

本研究における、V-Pascal バージョン 3 コンパイラの開発の初期段階におきましては、酒井淳嗣氏、梅田憲氏に絶大なるご助力を頂きました。ここで特に名前を挙げて感謝の意を表します。また、開発の後期には影本英樹氏、村井均氏、山本政行氏、韓東洙氏にご尽力頂きましたので、ここに感謝致します。開発過程におきましては以下の多くの方々のご助力を頂きましたので、ここに名前を挙げて感謝致します (敬称略)。

柘植宗俊、新田哲也、藤田健治、岡山藤治、中川真也、三吉郁夫、
花木義孝、松本太、上田康裕、野本政和、和田洋征、小山洋一、
中嶋廣二、吉田力

また、研究会などでご討論下さいました京都大学工学部津田研究室の他の卒業生、在学生の諸氏に感謝致します。

V-Pascal の開発に必要な技術情報を開示下さいました、株式会社日立製作所の関係各位に対しここに厚く御礼申し上げます。

最後に、研究生活を側面から補助してくださいました元技官 中谷いつ子氏、事務補助に協力してくれた妹・美根に感謝します。また、私が研究生活に入るきっかけをくれた父・明、母・洋子、さまざまなサポートをしてくれる友人 松村嘉男君をはじめ、他の多くの友人たちに感謝します。

List of Publications by the Author

Major Publications

1. Uehara, T. and Tsuda, T., *Benchmarking Vector Indirect Load/Store Instructions,* Proc. of the International Symposium on Supercomputing, Kyushu University Press, pp.134–143, 1991; also in Supercomputer, ASFRA (The Netherlands), Vol.8, No.48, pp.57–74, 1991.

2. Uehara, T. and Tsuda, T., *A Breadth-First Method for Automatic Vectorization and Parallelization of Recursive Procedures,* Proc. of Joint Symposium on Parallel Processing 1993 (Tokyo, Japan), pp.135–142, 1993. (in Japanese)

3. Uehara, T. and Tsuda, T., *Benchmarking Vector Indirect Load/Store Instructions,* Proc. of Workshop on Benchmarking and Performance Evalutation in High Performance Computing (Tokyo, Japan), pp.16–25, 1993.

4. Tsuda, T., Kunieda, Y. and Uehara, T., *The Vectorizing/Parallelizing Compiler V-Pascal,* Parallel Language and Compiler Research in Japan (Edited by Lubomir F.Bic, Alexandru Nicolau and Mitsuhisa Sato), Kluwer Academic Publishers, Chapter 12, pp.303 – 312, 1995.

5. Uehara, T., Kunieda, Y. and Tsuda, T., *An Automatic Vectorizing/Parallelizing Pascal Compiler V-Pascal Ver.3,* Proc. of Inter-

national Symposium on Parallel and Distributed Supercomputing (Fukuoka, Japan), pp. 206-213, 1995.

Technical Reports

1. Uehara, T., Kunieda, Y. and Tsuda, T., *An Automatic Vectorizing and Parallelizing Compiler V-Pascal Ver.3*, Tech. Rep. of IEICE CPSY94–90, Vol.94, No.384, pp.49–55, 1994. (in Japanese); also Tech. Rep. of IPSJ 94–OS–67, Vol.94, No.106, pp.145–152, 1994. (in Japanese)

Convention Records

1. Mizunuma, I., Uehara, T., Okabe, Y., Kunieda, Y., and Tsuda, T., *Data-Dependence Analysis of Nested Loops Containing Symbolics and Nonlinear Expressions*, Proc. of the 9th National Convention of Japan Society for Software Science and Technology (Fujisawa, Japan), pp. 485-488, 1992. (in Japanese)

2. Okayama, T., Uehara, T. and Tsuda, T., *Extentions of the Breadth-First Method for Vectorization of Recursive Procedures*, Proc. of the 48th Annual Convention of IPSJ, 6G–3, 1994. (in Japanese)

3. Umeda, K., Uehara, T. and Tsuda, T., *V–Pascal/DM, an Automatic Parallelizing Compiler for Distributed Memory Machines*, Proc. of the 48th Annual Convention of IPSJ, 5G–1, 1994. (in Japanese)