(1)

Exploiting Parallelism in Cyclic Pipeline Computer with an Optimizing Compiler

最適化コンパイラによる
循環パイプライン・コンピュータにおける並列化の研究

佐　藤　三　久

Exploiting Parallelism in Cyclic Pipeline Computer with an Optimizing Compiler

by

Mitsuhisa Sato

A Dissertation Submitted to
Department of Information Science,
Faculty of Science,
The University of Tokyo
in Partial Fulfillment of the Requirements for the Degree of Doctor of Science

December, 1990

ABSTRACT

In a pipelined computer, instruction dependencies involving both data and control information often limit its potential performance. A cyclic pipeline computer allows multiple instruction streams to share these pipeline stages in time to remove the data and control dependencies. These multiple instruction streams in the cyclic pipeline computer exploit more parallelism in the parallel program of scientific applications.

In this thesis, we define the basic model of a cyclic pipeline machine to examine the performance improvement of various configurations of cyclic pipeline machines compared to the same degree of pipelining of conventional architectures with single instruction stream. The simulation results indicate that pipelining within each instruction stream of a cyclic pipeline machine increases the performance to maximize the utilization of resources in a highly pipelined machine.

FLATS2 is an experimental cyclic pipeline computer with two instruction streams. FLATS2 FORTRAN compiler is an optimizing compiler for FLATS2. It enables us to exploit parallelism in parallel programs with the multiple instruction streams in FLATS2. The BL addressing of FLATS2, which integrates memory addressing and range checking, exploits the microarchitectural parallelism to reduce the execution time for array computation. The compiler generates optimized code using the BL addressing.

While the degree of pipelining in many conventional processors is usually limited on mature silicon technology, the cyclic pipeline computer provides alternative architectural solution for new technologies such as GaAs and Josephson logic device, which prefer a highly pipelined architecture.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Professor Eiichi Goto for suggestions, advice, and continuous encouragement. I am also grateful to my colleagues Norihiro Fukazawa, Shuichi Ichikawa, Paul Spee, Shu Kawakami and Wong Weng Fai at the computer architecture group of GOTO Quantum Magneto Flux Logic Project of Research Development Corporation of Japan (JRDC) for valuable help and discussion. Spee and Wong helped by carefully proofreading this thesis. I would like to thank the other members of FLATS2 project, Noriyuki Kato and Yasuo Wada at JRDC, Dr. Takashi Soma, Nobuyui Inada and Masayuki Suzuki at the Institute of Physical and Chemical Research, the members at Mitsui Engineering and Ship-building Corporation and Mitsui Zosen System Research Incorporated.

Most of this research has been supported by the Research Development Corporation of Japan.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Cyclic Pipeline Computer

Pipelining is a very appealing design technique because it offers a theoretical speedup of N when N pipeline stages are used. An operation in a pipelined machine may take several cycles to complete, but a new operation can be started on each cycle, so throughput remains high. There are, however, practical constraints that limit the possible performance increase.

Instruction dependencies involving both data and control information limit performance because they reduce the amount of the potential parallelism that is actually realized. It is well known that instruction-level parallelism is limited to a relatively small amount [Jou89].

The technology used to implement computer systems affects its architectural design. GaAs technology has recently shown rapid increases in maturity, and GaAs computer system design has already generated considerable interest. A highly pipelined architecture is extremely promising for this technology [MFH86]. A new Josephson devices, QFP (Quantum Flux Parametron) [HNM$^+$87] forces the entire system to be heavily pipelined. On mature silicon technology, the degree of pipelining in many conventional processors is usually limited. Traditional computer designs use resources inefficiently, resulting in machines whose performance is disappointing when compared to the raw speed of their components. Even in silicon VLSI technology, asynchronous system designs such as micropipelines [Sut89] and self-timed systems offer a highly pipelined system.

The pipelined memory is an attractive component for a highly pipelined system. A conventional memory chip consists of the address decoder, the memory cell array, the sense amplifier, and multiplexer. By positioning latches between them, the memory access is easily pipelined so that the cycle time rather than the access time is minimized. For data fetch, a processor can issue several memory requests concurrently as a pipelined functional unit.

The disadvantage of a pipelined memory system for a highly pipelined computer is that the increased pipeline depth places strict requirements on the optimizing compiler. For example, branch delays are longer, and early compiler efforts to replace the NOP instruction in the fill-in slots are most successful for short branch delay [MFHL87].

One way to remove the data and control dependencies in a highly pipelined system is to share these pipeline stages between different instruction streams. A *cyclic pipeline machine* [SGI89] issues instructions periodically from a fixed number of instruction streams. The cyclic pipeline machine provides identical functionality as a true multiprocessor with shared memory. In a true multiprocessors system, contention for synchronization locks and delay waiting for synchronization events can substantially increase the running time of a parallel program. Among the multiple instruction streams in a pipelined processor, however, the synchronization cost can be reduced because its pipelined memory access causes no memory access conflict.

1.2 FLATS2

FLATS2 is an experimental cyclic pipeline machine implemented using silicon ECL technology. FLATS2 serves as an example of a cyclic pipeline computer system design in this thesis. The main memory is not pipelined because the pipelined memory chips were not available in market. Ten pipeline stages are time-shared by two instruction streams [Ich90].

To get high performance per instruction stream, the instruction set is designed to allow rather complicated operations in the instruction as well as overlapped execution of instructions. For example, an instruction can take up to two memory operands, and perform operations of both the integer arithmetic function unit and the floating point function unit in one instruction.

FLATS2 provides the *BL addressing* and the *address tag* to support faster execution in each instruction stream. The BL addressing is a memory addressing mechanism which integrates memory addressing and range checking. The effective address is checked against the specified pair of base and limit addresses in registers during memory access. An address tag is a bit in a word, which indicates the capability for memory access. Combining them together, the test for terminating the loop of an array computation can be overlapped with its computational operation to reduce the execution time. We can also make use of these facilities to reduce the cost of run-time type checking in Lisp [SIG89].

Each instruction stream is called a *virtual processor*. The term "processor" means that the processor of each instruction stream has a different set of program counter, registers and processor status including the privileged mode bit. Each processor can be used to execute independent tasks under a multiprocessor operating system.

1.3 FLATS2 FORTRAN Compiler

The FLATS2 FORTRAN compiler implements most of the optimization techniques of the better traditional Fortran compilers. Code optimization includes common subexpression elimination, constant folding, code motion and strength reduction. To facilitate optimizations, the compiler converts code into *static single assignment (SSA)* form [CFR$^+$89] for machine independent optimizations. The internal code is represented in register transfer language (RTL) [DF84], which enables machine dependent optimizations such as peep hole optimization to be done in a machine independent way.

While these optimization would be essential in a commercial compiler, they are also essential to prove this thesis. Without highly optimized code, the speedup found by machine-specific features might be highly suspect, since the speed-up found by machine-specific features might be strongly biased by the code that otherwise would have been optimized away.

For a loop of array computation, the FLATS2 FORTRAN compiler automatically generates the code using the BL addressing when it finds the code to which BL addressing can be applied. With the optimizing compiler, the BL addressing reduces the execution time of scientific workloads by 10-30% in average.

For simulation of highly pipelined machines, the FLATS2 FORTRAN compiler generates the optimized code using the subset of FLATS2 instruction set for the Load/Store architecture. The Load/Store architecture allows us to simplify the model of a highly pipelined processor. The additional code scheduling phase in the code generator handles interlocks due to the data dependency to generate optimized code for different pipeline configurations.

1.4 Parallel Programming Model

The cyclic pipeline computer provides the identical functionality as the true multiprocessor with shared memory.

We use the Force [Jor87] as the parallel programming model. In the Force, many processes executes a single program. The number of processes is arbitrary, but fixed at run-time. Each processes can be synchronized by *barriers* and *critical sections*. The synchronization is specified by the parallel directives. The parallel directives are placed in a source program to make use of multiple processes provided by the cyclic pipeline computer.

The DO loop is a major source of parallelism in scientific workloads. Two types of parallel DO is provided to distribute the work of the loop whose iterations can be executed in parallel. A *self-scheduled* parallel DO allows each iteration to be assigned dynamically for processors. A *pre-scheduled* DO specifies to partition iterations at the beginning of the loop. If the work of each iteration is load-balanced, a pre-scheduled DO can execute the loop efficiently because of its low synchronization cost.

1.5 FLATS2 project

The FLATS2 project was started around 1986 by the computer architecture group of Goto Quantum Magneto Flux Logic project in Research Development Corporation of Japan (JRDC), which aims a future supercomputer with Josephson technology. The FLATS2 FORTRAN compiler is one result of that project.

The architecture of a cyclic pipeline computer was proposed suited for Josephson devices [SGI89]. FLATS2 was built as a prototype of a cyclic pipeline computer with silicon ECL technology. The basic idea of a cyclic pipeline computer and BL addressing is due to Eiichi Goto, the project leader. Shuichi Ichikawa was a main designer of FLATS2. Mitsui Engineering and Ship-building Corporation built the hardware of FLATS2.

In early stage of the project, I was charged with implementing the FLATS2 instruction-level simulator to develop the systems software. Norihiro Fukazawa worked on systems software including a assembler, a linker and a down-loader for FLATS2. Paul Spee retargeted the Gnu C compiler to FLATS2, which makes use of an address tag to detect the illegal usage of pointers. He also designed and implemented the optimization using the BL addressing for array computation. My code generation is quite different from his, but I definitely profited from his effort.

The CPX operating system, which was developed for FLATS2, was designed by Paul Spee, Norihiro Fukazawa and me. Spee did the actual implementation on FLATS2.

I worked on the FLATS2 FORTRAN compiler and parallelizing the scientific applications with the compiler. I also proposed the Lisp system for FLATS2 using the BL addressing. I did most of the experiments, collecting the program in our workloads, parallelizing them with the compiler, and coaxing them through the compiler, the simulator and FLATS2.

1.6 Scope of Study in This Thesis

We are interested in the performance improvement obtained by general cyclic pipeline computers and a real cyclic pipeline computer, FLATS2, compared to conventional architectures. The cyclic pipeline architecture is a single processor architecture, while it allows the multiple instruction streams. In this thesis, the cyclic pipeline computer is compared to other single processor architecture such as superpipelined architecture.

The FLATS2 FORTRAN compiler can generate optimized code for various configurations of architectures. With the compiler, we can parallelize scientific applications to exploit parallelism provided by a cyclic pipeline computer. The compiler also exploits the microarchitectural parallelism of the BL addressing in FLATS2. For simulation, the additional instruction pipeline scheduler of the compiler can generate optimized code of pipelined machines with single instruction stream.

In Chapter 2, we define some models of a highly pipelined machine to examine the performance improvement of a cyclic pipeline machine compared to the same degree of pipelining of a conventional machine with single instruction stream.

In Chapter 3, we describe the architecture and the programming of FLATS2. Chapter 4 gives the overview of the FLATS2 FORTRAN compiler and several algorithms used to optimize the code. In Chapter 5, the optimization for the BL addressing is presented. Chapter 6 reports and analyzes our experiments results on FLATS2 and the simulator.

Finally, Chapter 7 presents our summary and conclusions.

Chapter 2

Highly pipelined Model

In this chapter, we introduce the machine models for a highly pipelined processor. A cyclic pipeline machine is defined as one of the models. We examine the performance improvement of various configurations of cyclic pipeline machines compared to the same degree of pipelining of a conventional highly pipelined processor with single instruction stream [SIG90]. The simulation results indicate that pipelining in each instruction stream of a cyclic pipeline machine increases the performance to maximize the utilization of resources in a highly pipelined processor.

Recently, several techniques have been developed to exploit the parallelism of programs for multiprocessors [ACK87] [Pol89]. Scientific applications are often dominated by highly parallel codes, and a parallel computer would improve these application performances. The cyclic pipeline machine provides identical functionality as the true multiprocessor with shared memory. By using pipelined memory, there is no memory access conflict in a cyclic pipeline machine. As a result, the synchronization operation is very cheap. The cyclic pipeline machine can exploit such parallelism in a highly pipelined processor.

In Section 1, we define the model of the cyclic pipeline machine used as a basis for our simulations. In Section 2, the parallel programming model is described. Section 3 describes the simulation environment we used to measure the performance, and presents the detail of benchmarks and the result of our simulations. In Section 4, discussion on a cyclic pipeline machine is presented.

2.1 Models of Highly Pipelined Machines

There are several different ways to execute instructions in a highly pipelined machine. In this section, we define the basic model of a cyclic pipeline machine.

2.1.1 The base machine

We start with a typical pipeline composed of four stages: the instruction fetch (IF), the instruction decode (ID), the instruction execution (EX), and the write back (WB). For simplicity, we assume a register-register machine; this simplifies the pipeline and also makes it easier to quantify execution time. In this type of machine, instructions are classified into a small set of simple operations such as integer add/sub, logical ops, load, store, branch and floating-point ops.

An operation latency is the time (in cycle time) until the result of an instruction is available for use as an operand in a subsequent instruction. If the operation latency is one, the next instruction can use the result immediately.

To examine increases in performance due to highly pipelined structure, we define a base machine which has a non-pipelined execution pipe stage. In the base machine, instructions are issued at each machine cycle, and the latency of all operations are exactly one. Since the result of an instruction is always available for the successive instruction without delay, there are never any operation-latency interlocks in a base machine. Figure 2.1 shows a pipeline diagram for the base machine.

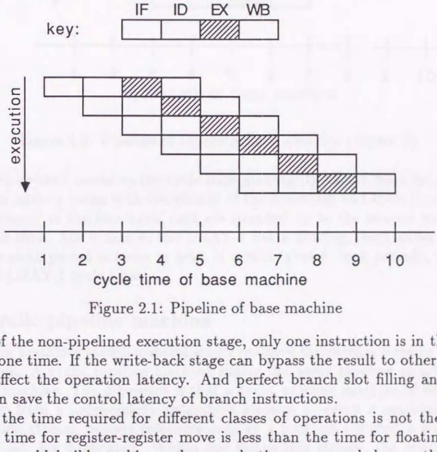


Figure 2.1: Pipeline of base machine

Because of the non-pipelined execution stage, only one instruction is in the execution stage at any one time. If the write-back stage can bypass the result to other pipe stages, it does not affect the operation latency. And perfect branch slot filling and/or branch prediction can save the control latency of branch instructions.

Actually, the time required for different classes of operations is not the same. For example, the time for register-register move is less than the time for floating-point ops. Although one could build machines whose cycle time was much longer than the time required for operation, it would be a waste of execution time and resources.

2.1.2 Superpipelined machine

Superpipelined machines exploit instruction-level parallelism by pipelining instruction execution stages [Jou89]. Each stage is divided into smaller pipeline segments. A superpipelined machine of degree m is a machine whose execution stages have parallelism of m. An existing example which has pipelined functional unit is CDC 7600.

Instructions are issued at every machine cycle, but the cycle time is $1/m$ of the base machine. The operation latency is m in its cycle time. When an operation takes a whole cycle in the base machine, given the same implementation technology it must take m cycles in the superpipelined machine. Figure 2.2 shows the execution of instructions by a superpipelined machine.

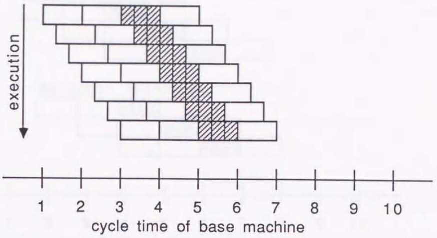


Figure 2.2: Pipeline of superpipelined machine (degree 3)

In a superpipelined machine, the cycle time granularity affects machine performance. The operation latency varies with complexity of the operation and cycle time granularity. The actual latency of the functional unit are rounded up to the nearest multiple of the machine cycle time. For example, the CRAY-1 has a floating point adder latency of 7 cycles. If the clock period is twice as long, it would takes 4 clock periods, which results in 8 cycles of CRAY-1 cycle time.

2.1.3 Cyclic pipeline machine

Cyclic pipeline machines exploit parallelism by multiple instruction streams time-sharing pipeline stages. A cyclic pipeline machine shares the same pipeline structure as a superpipelined machine. The difference is that a cyclic pipeline machine of degree n issues an instruction from n independent instruction streams at every n machine cycles. For each instruction stream, private resources such as a program counter, a status register, and registers set, are duplicated. Within one instruction stream, the operation latency is the same as that of the basic machine. Total throughput of a cyclic pipeline machine, however, is n times larger.

Figure 2.3 shows the execution of instructions by a cyclic pipeline machine.

Each instruction stream in a cyclic pipeline machine appears to be identical to a real processor in the multiprocessor. The number of processors provided by a cyclic pipeline machine is limited by the pipelining factors.

By using pipelined memory for main memory or cache, memory access can also be pipelined so that several memory requests can be outstanding at the same time. It should be noted that pipelined memory access enables each processor to share the main memory without memory access conflict. Furthermore, no extra switching network is necessary

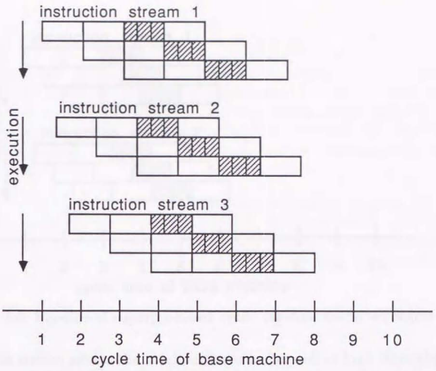


Figure 2.3: Pipeline of cyclic pipeline machine (degree 3)

between the processors and the main memory. Unlike true multiprocessors, multiple instruction streams of a cyclic pipelined machine can execute parallel programs without overhead for memory access to shared data.

FLATS2 is an experimental cyclic pipeline machine with two instruction streams.

HEP is a resource-shared pipelined machine [Kow85]. It issues an instruction in queues dynamically from the arbitrary instruction streams. It has several process execution modules (PEM) and complicated memory access mechanisms.

2.1.4 Superpipelined cyclic pipeline machine

Since two independent instructions in a instruction stream of a cyclic pipeline machine can be overlapped in the execution pipe stages, we can have a superpipelined cyclic pipeline machine. A superpipelined cyclic pipeline machine of degree *(m,n)* has a cycle time *1/mn* that of the base machine, and issues instructions periodically from n instruction streams. There are m operations of each instruction stream in progress at the same time. Figure 2.4 shows the execution of instructions by a superpipelined cyclic pipeline machine.

A cyclic pipeline machine can achieve high throughput for highly parallel code. If the applications are dominated by highly parallel code, they can be divided into several processes executed in parallel. To perform a single task with multiple processes, some parts must be executed sequentially and process synchronization is inevitable. Even if there is no overhead to access the shared resources, such as lock and shared data, delays to wait for synchronization of events can substantially increase the running time of a parallel program, and seriously degrade processor utilization. For example, if other processes await an event set by one process, the execution time of the process to reach the event point dominates the total execution time. Even in highly parallel codes, the performance for

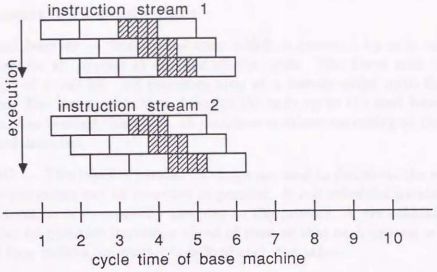


Figure 2.4: Pipeline of superpipelined cyclic pipeline machine (degree (2,2))

one instruction stream affects the total performance as well as high throughput. Pipelining within the individual instruction stream can improve the performance of instruction stream.

On the other hand, instruction-level parallelism, which can be exploited by pipelining, is limited to a small amount. Highly pipelining in a superpipelined machine can not drastically improve the performance.

Consequently, a trade-off of the degree of pipelining in each instruction stream and the number of instruction streams for a cyclic pipeline machine is required to maximize the performance on various parallel programs.

2.2 Parallel Programming Model

For a parallel program, the programmer needs tools to express the parallelism, either in the form of subroutine libraries or language extensions. Our language extension of FORTRAN is from the Force [Jor87]. The Force is based on the shared memory multiprocessor model of computation. The cyclic pipeline machine provides the identical functionality as the true multiprocessor with shared memory.

2.2.1 FORTRAN parallel directives

In the Force, multiple processes execute a single program. The number of processes is arbitrary, but fixed at run-time. The parallel constructs of our FORTRAN are:

Data allocation — All data in COMMON block are allocated globally, and can be referenced from any process. The local data is allocated in registers or stack area private for each process.

critical, end_critical — Specify the critical section which contains codes that are executed by all processes one at a time. It is often used for reduction operations such as summing into a global variable.

barrier, end_barrier — Specify the code which is executed by only one process and synchronize all process at the end of the code. The Force uses the generalized concept of a barrier. All processes stop at a barrier point until the last one has arrived. The last process then executes the code up to the **end_barrier**. Once the process has reached this point, all processes continue executing at the line following the **end_barrier**.

parallel DO — Two types of parallel DO loops are used to distribute the work of the loop whose iterations can be executed in parallel. A *self-scheduled* parallel DO specifies each iteration is dynamically assigned to the process. A *pre-scheduled* parallel DO specifies to partition iterations ahead of time so that each process will do a certain set of loop indices, no matter how long each one takes.

SYNC function — The function SYNC forces the process to wait until the synchronization data of the specified location is non-zero. This function is used to synchronize on the data between processes. Writing non-zero value to the synchronization data by other processes releases the waiting process on the data. Although this function can be implemented by the loop reading the synchronization data, this function prevents the optimization such as loop invariant code motion.

IPID function — The function IPID returns the process number of the process executing this function, starting from zero.

These directives are placed in the source program as comments starting with "*$". In compilation, these are ignored as comments when parallel directives are disabled. The statement starting with "*%" is used only for parallel execution. Figure 2.5 shows the complete program which computes an inner product.

The choice between the two types of parallel loops depends on both the characteristics of applications and the cost of synchronization. If the work is naturally load-balanced, then pre-scheduling is preferred. In a cyclic pipelined machine, self-scheduling is also efficient because of its low synchronization cost.

Note that the parallel program without the function **IPID** works on any number of processes. For such a program, the number of processes may be given dynamically at the beginning of the execution. The function **IPID** is sometimes used for parallel programming with a particular knowledge about the number of processes.

2.2.2 Implementation of Parallel Directives

No special hardware is assumed for synchronization. As an atomic operation, the *load-and-store* instruction, which swaps the value between the private register and a global lock variable, is provided. Waiting for a synchronization event is implemented by "spinning" in a software loop, repeatedly reading a synchronization variable until it become available. Since there is no memory access conflict in a cyclic pipeline computer, the *hot spot problem* does not arise.

```
c
c parallel version of inner product.
c
      PARAMETER(N = 100)
      DOUBLE PRECISION X(N),Y(N)
c global data
      COMMON /XDATA/X,Y,XY
c set sample data
*$ pre_sched_do
      DO 10 I = 1,N
         X(I) = I
         Y(I) = I + 1
 10   CONTINUE
c call parallel functions
      CALL IP(N,X,Y,XY)
*$ barrier
     WRITE(6,920) XY
 920  FORMAT('IP = ',E16.8)
      STOP
*$ end_barrier
     END
c
      SUBROUTINE IP(N,DX,DY,XY)
      DOUBLE PRECISION DX(1),DY(1),DTEMP,XY
      INTEGER N
      DOUBLE PRECISION DDTEMP
      COMMON /TDATA/DDTEMP
c
*$ barrier
      DDTEMP = 0.0D0
*$ end_barrier
      DTEMP = 0.0D0
*$ pre_sched_do
      DO 30 I = 1,N
         DTEMP = DTEMP + DX(I)*DY(I)
 30   CONTINUE
*$ critical
      DDTEMP = DDTEMP + DTEMP
*$ end_critical
*$ barrier
      XY = DDTEMP
*$ end_barrier
      RETURN
      END
```

Figure 2.5: An example of parallel program

2.3 Simulation

2.3.1 Simulation Environment

The simulation system, originally designed for FLATS2, consists of an optimizing FORTRAN compiler (including assembler, linker and debugger) and a FLATS2 instruction-level simulator. In the simulation, a subset of FLATS2 instruction set is used, including loads, stores and arithmetic instruction between register operands [1]. To specify the pipeline structure and functional unit, we classified the instructions so that instructions in the same class are likely to have identical behavior in any machine.

To investigate the effect of operation latency, we specify an operation latency for each instruction class at compile-time. The compiler includes a pipeline instruction scheduler using the algorithm in [GM86]. The scheduler reorganizes the instructions in a basic block to minimize the execution time. If an instruction requires the result of a previous instruction, the scheduler inserts NOP's or schedules the other independent instructions to avoid data dependency interlock until the result is available. Since the machine never stalls by data dependency interlock, the number of executed instructions gives the execution time in machine cycle.

Since scheduling is limited in a basic block, the instruction-level parallelism measured is limited within a basic block. Although particular compilation techniques such as trace scheduling and software pipelining can exploit the interblock instruction-level parallelism, the evaluation of these elaborate compilation techniques is beyond the scope of this thesis.

We assume that control instructions such as branch take one cycle for any pipeline structure in the simulation. The cost of control instructions depends on the instruction fetch logic. A clever instruction issue logic and good branch prediction can reduce the branch performance penalty to flush the incorrect instructions in pipeline stages. Note that if we took these costs into account, the performance would decrease in a higher degree of superpipelined machines.

2.3.2 Workloads

To investigate the performance of various superpipelined and cyclic pipeline machines on scientific workload, we chose the following programs as workloads:

Inner product computation — The inner product computation is one of the most basic computations in scientific applications. The parallel version is shown in 2.5. Each iteration of the loop is distributed to compute the partial result of the inner product. At the end of the loop, these partial result is added to the final result in critical section.

Linpack benchmark —This benchmark program solves a linear equation system, which is one of the most typical scientific computations.

[1]The register set of FLATS2 consists of 32 global registers and 32 local registers which are switched on procedure call, and 4 floating point registers. Since procedure calls are rarely used in scientific computation, the local registers do not contribute to the performance much.

The benchmark program consists of two subroutines: **dgefa** and **dgesl** (double precision). **Dgefa** factors a dense matrix by gaussian elimination into its LU components. **Dgesl** solves the equation system by factored matrix. The time needed to factor a matrix of order N is proportional to N^3, while the time for the forward elimination and back substitution only increases as N^2. It is known that for matrices with $N > 100$, LU decomposition accounts for over 90% of the execution time. In our simulation, the order of matrix is 100.

Transforming the matrix into diagonal form is accomplished by making N transformation passes over the matrix. A pass, K pass is divided into two parts: the first finds the Kth pivot element and performs a row exchange if necessary, and divides the element below the diagonal by the pivot to produce a set of *multipliers*. Then the second multiplies the part of the pivot row to the right of the diagonal times each multiplier and subtracts the product from the correspondinging part of each row to make the Kth column consist of all zeros, except for the diagonal element.

In the parallel version, each processor can perform the second part for different rows independently. The pre-schedule parallel DO loop construct is used to distribute to iteration on each row to each processor. The parallelism of this part contributes the significant performance improvement in the parallel version. At the end of this part, all processors are synchronized by a barrier. Since the cost of synchronization is very low in a cyclic pipeline machine, finding the pivot element and computing the multipliers in the first part are also distributed to each processor.

For solving the diagonalized system, each step on row for the forward elimination and back substitution must be serialized. Only computations on columns in each step are distributed. At the end of each step, all processors are synchronized by a barrier.

FEM_BAND - The finite element method using the band matrix, taken directly from Mori[Mas86]. The band matrix is solved by the modified Cholesky decomposition. The program solves the following Poisson's equation in a square region:

$$\begin{cases} -\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = -1 & \\ u(x,0) = 0, & 0 \le x \le 1 \\ u(x,1) = x, & 0 \le x \le 1 \\ \frac{\partial u}{\partial n}(0,y) = -\frac{\partial u}{\partial x}(0,y) = 0, & 0 \le y \le 1 \\ \frac{\partial u}{\partial n}(1,y) = -\frac{\partial u}{\partial x}(1,y) = 0, & 0 \le y \le 1 \end{cases}$$

where $\frac{\partial}{\partial n}$ is an element right hand side vector with respect to the boundary side of the natural boundary condition.

The program consists of two subroutines: MATGEN and SOLV. The MATGEN computes an element matrix for each element to arrange the global matrix in band form, which is solved by SOLV.

In the parallel version of MATGEN, each element matrix is computed in parallel independently. When the element matrix is added to the global matrix, the addition is done in the critical section to make sure that the other processes do not modify the matrix simultaneously. In SOLV, the modified Cholesky decomposition decomposes the positive definite matrix faster than the LU decomposition. Like the LU decomposition, transforming the matrix into diagonal form is accomplished by making N transformation passes over the matrix. In a pass of transformation, the pre-schedule parallel DO loop construct is used to distribute to iteration on column to each processor. Since the transformation on column uses the result on the previous column, we introduce the additional vector to synchronize the data on each column.

For solving the decomposed matrix, the synchronization vector is also used to distribute iterations on row to processors. Because the width of the band matrix is relatively small, the synchronization on the data is effective to get more parallelism.

FEM_ICCG — The finite element method using the matrix in list vector form, also taken from Mori [Mas86]. The matrix is solved by the ICCG (Incomplete Cholesky decomposition and Conjugate Gradient) method. The problem to be solved is the same as FEM_BAND. Like FEM_BAND, the program consists of MATGEN and SOLV. MATGEN of FEM_ICCG is similar to that of FEM_BAND except that the matrix is stored in list vector form.

In the CG method, finding the solutions and updating of the matrix are repreated until the expected precision is obtained. In the parallel version, the part to update of the matrix is parallelized because it consists many simple vector operations such as inner product. As the SOLV in FEM_BAND, the decomposing and solving of the matrix use the synchronization vector on row. Note that we can not exploit parallelism on column because the number of elements on row is at most 3.

Appendix B contains the parallel version of programs.

2.3.3 Simulation Results

We ran the workloads to measure the execution time on different configuration and degree of pipelining. The operation latencies were estimated based on CRAY-1 instruction timing. The latencies on each degree of superpipelined processors is shown in Table 2.1.

Degree of Pipelining	register move	ALU operations	load	store	FP operations	branch
1	1	1	1	1	1	1
2	1	1	2	1	2	1
3	1	1	2	1	3	1
4	1	2	3	1	4	1
6	1	2	4	1	6	1
8	2	3	6	2	8	2

Table 2.1: Latency of instructions

It is assumed that the memory system can accept a new request at each clock cycle. The effects of cache misses and page faults are ignored.

We show the simulation results in Figure 2.6, 2.7, 2.8, 2.9. The line (N,1) indicates the superpipelined processors. Note that loops in these programs are not unrolled. Loop unrolling will increase the instruction-level parallelism.

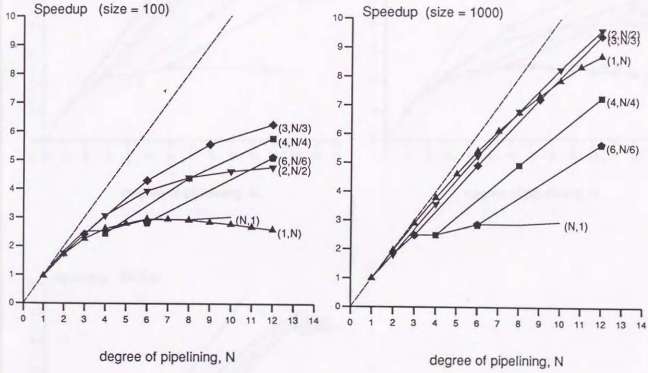
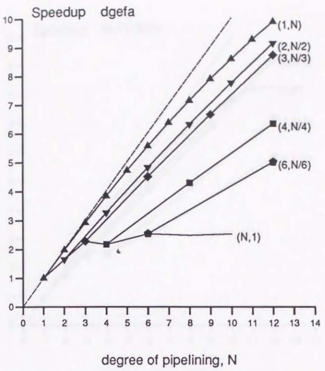


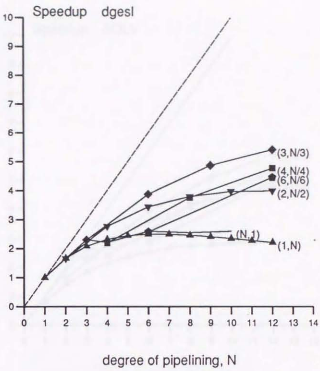
Figure 2.6: Speedup of inner product

We ran the inner product computation for 100 and 1000 in size. The execution time for the setup of the loop and adding the partial product is the same for both sizes. The performance of the size 1000 is improved more than that of the size 100 because the parallelized part is larger. Note that the performance improvement of (1,N) can not exceed the improvement of the superpipelined processor when the size is 100. The inner product computation is used in our other workloads. The cyclic pipeline machine (1,N) cannot improve the performance over the superpipelined machine for small vectors.

In Figure 2.7, the speedups of a superpipelined machine are limited to about two times by the instruction-level parallelisms of these subroutine. These results match the results reported in [Jou89].

For **dgefa**, nearly linear speedup can be achieved in the cyclic pipeline machine. The reason is that the amount of codes executed in parallel is very large and delay times for waiting at a barrier is relatively small. The number of row operations required in each iteration decreases steadily, since only the columns that have not yet been diagonalized need to be manipulated. For each pass, the number of row operations done per processors is nearly balanced.





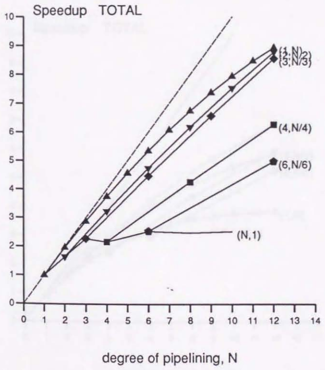


Figure 2.7: Speedup of Linpack

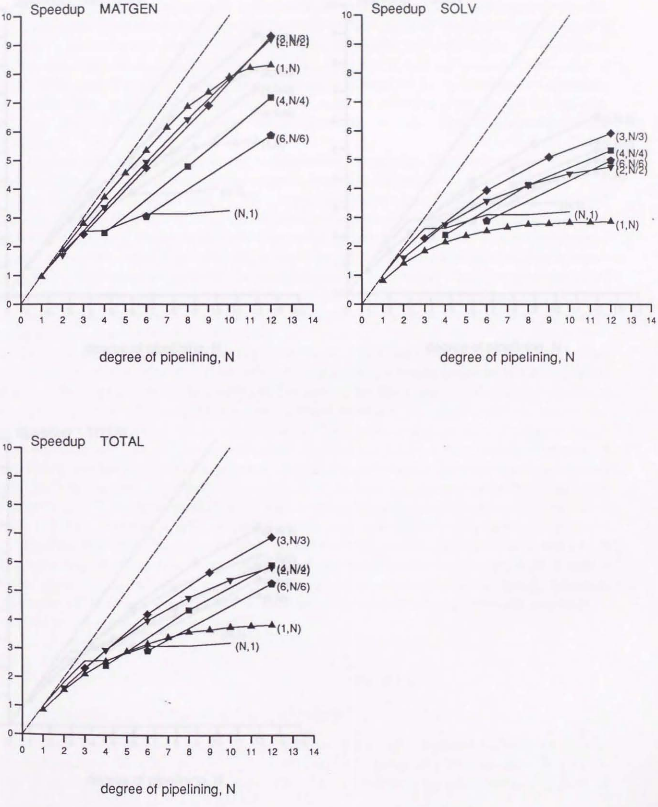
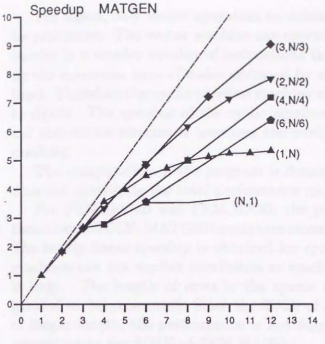
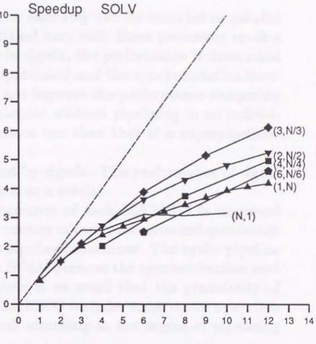


Figure 2.8: Speedup of FEM_BAND (size = 16*16)





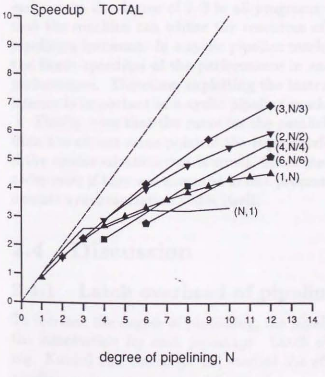


Figure 2.9: Speedup of FEM_ICCG (size = 16*16)

For **dgesl**, only vector operation on columns in each step can be executed in parallel by processors. The vector machine can execute **dgesl** very well. Since processors reach a barrier in a smaller number of instructions than in **dgefa**, the performance is dominated by the execution time of codes executed by one processor and the synchronization overhead. Therefore the cyclic pipeline machine can not improve the performance comparing to **dgefa**. The speedup of the cyclic pipeline machine without pipelining in an individual instruction streams is increases the performance less than that of a superpipelined machine.

The computation of the program is dominated by **dgefa**. The performance of **dgesl** does not contribute the total performance so much as a result.

For FEM_BAND and FEM_ICCG, the performance of each MATGEN is improved than that of SOLV. MATGEN computes element vectors in each processors independently. The nearly linear speedup is obtained for cyclic pipelined machines. The cyclic pipeline machines can not exploit parallelism so much in SOLV because the synchronization cost is large. The length of rows in the sparse matrix is so small that the granularity of parallelism become small. Since the SOLV of FEM_ICCG includes some vector operation of longer length, the performance is not degraded according to the degree of pipelining comparing to the SOLV of FEM_BAND.

For relatively less parallel programs such as **dgesl** and SOLV, the cyclic pipeline machine with the pipelining degree of 2–3 in instruction streams provides the best performance. The reason is that this configuration can utilize the resources of a highly pipelined machine the most efficiently. In a superpipelined machine, the performance increases linearly up to the degree of 2–3 in all programs. The linear speedups in performance imply that the machine can utilize the resources of pipelines without loss when the degree of pipelining increases. In a cyclic pipeline machine, pipelining of the degree which provides the linear speedups of the performance in each instruction stream can achieve the best performance. Therefore, exploiting the instruction-level parallelism given the operation latency is important in a cyclic pipeline machine as well as decreasing the latency.

Finally, note that the curve for the parallel code for a cyclic pipeline machine start off from the almost same point of the superpipelined machine of the same pipelining degree if the synchronization cost is small. Since the parallel codes still include synchronization codes even if they are executed in one processor, this difference indicates the overhead to execute synchronization codes itself.

2.4 Discussion

2.4.1 Latch overhead of pipelining

To increase the degree of pipelining, the pipeline latch registers must be inserted to keep the information for each pipestage. Latch overhead limits the performance of pipelining. Kunkel and Smith [KS86] studied the effect of latch overhead in various degrees of pipelining via simulation of CRAY-1S.

Latch propagation delay occurs in gate used to construct latches. A latch typically has a propagation delay from clock to output of at least two gate delays. It involves significant delay in pipelining system when the clock period becomes very short. To reduce the delay, we can use the latch which performs useful logic functions. The so-called Earle latch and the polarity hold latch can perform any combinational logic function as well as the latching function. One of the authors proposed a new latch design, G-series gates base on ECL [GHHK], which can perform any three-input logic function efficiently in gate level. If logic functions are performed with this latch, the propagation delay can be essentially eliminated.

The clock frequency is also limited by physical characteristics of signal propagation. Data skew is the difference between the maximum and minimum signal propagation times through combinational logic between pipeline stages, and in the latches that separate the stages. The clock period must be long enough to ensure reliable latching data.

In a synchronous system, global clock signal must be controlled to reach all latches at the same time. When designing real processors, however, there is always some uncertainty in the clock signal. For example, differences between maximum and minimum delays in clock fun-out logic and differences of propagation time in different length of wire for clock signals cause an unintentional variation in the arrival time of the clock at succeeding latches in a pipeline. This clock signal skew also increases the clock period.

Micropipelines [Sut89] using self-timed logic are very attractive for a highly pipelined system. Since no global clock is needed in self-timed logic, there is no problem with clock skew. Each pipestage is connected with the two-phase bundled data convention, and communicates with each other. Micropipelines can achieve the speed of control event signal propagation, and provide very high throughput.

2.4.2 Memory system design of cyclic pipeline machine

The pipelined memory system is a key design in a cyclic pipeline machine. Pipelining is a technique that can be employed for both instructions and data. Given sufficient jobs which can be executed independently, pipelined memory allows a cyclic pipeline machine to achieve maximum throughput even without any cache because the pipelines are fully utilized. But for parallel programs executed by many processes, the performance of individual instruction stream reduces the delay time to await a synchronization event. Since caches are used to reduce the impact of memory latency on the performance of instruction stream, they are also effective in a cyclic pipeline machine.

When a processor is highly pipelined and the clock period becomes very short, it is necessary to issue instructions as fast as possible to achieve high throughput, so a highly pipelined machine requires high memory bandwidth. When the instruction fetch can be pipelined, the cycle time of memory is more essential than the access time. While the interleaved memory and the bank parallel memory achieve maximum throughput when there is no access conflict, pipelining is a very effective method for achieving performance increases with relatively small costs. If the memory access can not be pipelined to fetch instructions, independent instruction caches for each instruction stream could provide the same function as the pipelined instruction memory. For instruction cache, even if the memory is pipelined, multiple pipelined caches for different instruction streams help the increase of throughput in a cyclic pipeline machine when the cycle time of the pipelined cache is larger than the clock period.

For data memory, a pipelined memory enables the processor to give the cache several memory requests concurrently before getting any data back. In the pipelined system, the entire system operates at maximum throughput rate of its slowest pipestages. If the cycle time of the pipelined memory was larger than the clock period, the pipestages of data fetch limit the throughput of pipelining, because the successive memory requests were blocked. One way to reduce the performance degradation due to the difference of throughput is to place queues between processor and memory. If data from cache does not arrive in the expected time due to extra memory latency such as cache miss, the entire processor must block to await the data.

Note that the disadvantage of a cyclic pipeline machine is that random memory access pattern of different instruction streams decreases locality of memory reference. More instruction streams in a cyclic pipeline machine would need a larger cache. To balance the number of instruction streams and the cache size, a trade-off is required between them.

2.4.3 New technology for a highly pipelined computer

The cyclic pipeline machine was originally proposed as an architecture suited for a new Josephson logic device DCFP. One of the distinct characteristics of Josephson logic is that each basic logic device acts as a latch. Therefore, in this technology, high pitch, shallow-logic pipelining can be used without the delay time and cost of pipeline latch registers. By using the Josephson devices for the processor and the main memory (or cache), the over all system can be naturally pipelined with the same clock time.

A highly pipelined architecture is also extremely promising for GaAs technology. The high electron mobility of GaAs transistors results in very fast electron transit times across their active regions and hence generates the potential for extremely short gate propagation delay. This characteristic of GaAs devices offers high system clock rates compared to silicon devices. A signal processor implemented with GaAs technology can often be deeply pipelined and thus can exploit high system clock rate [GNST86].

One of the most critical differences between silicon technology and GaAs technology is that the GaAs wafers have a higher density of defects. This results in a very low chip yield, indirectly limits individual chip area and transistor counts. The limitation needs the system to be divided into small components such as cache and co-processor. In GaAs technology, the ratio of off-chip memory access speed to on-chip memory access speed is larger than in silicon technology. The penalty for accessing off-chip memory forces the architects to minimize the number of off-chip access, or, alternatively, to minimize the penalties for going off-chip. The longer GaAs memory system delay does not result from the lower raw speed of the memory itself, but from longer relative propagation delays between the processor and the memory. Breaking the instruction fetch stage into multiple stages, pipelined instruction fetch can reduce these penalties.

Another major difference between silicon and GaAs is the more limited fan-in/fan-out capability of GaAs. This characteristic of GaAs prefers the simple and regular arithmetic functional unit design such as the ripple-carry adder. Pipelining of arithmetic functional units can not improve the latency, but can improve the bandwidth [HF72].

Chapter 3

A Cyclic Pipeline Computer, FLATS2

FLATS2 is an experimental cyclic pipeline machine implemented using silicon ECL technology. In this section, we describe the FLATS2 machine from programmer's point of view. Its implementation and detailed description are given in [Ich87, Ich90].

3.1 The architecture of FLATS2

3.1.1 Memory and Registers

FLATS2 provides 32-bit single virtual space, as shown in Figure 3.1.

The lower half of the space is the D-space, which stores data. The upper half is divided into the space for instructions (I-space) and the space for memory-mapped registers (GV-space). In FLATS2, every data word has a one bit tag called the *address tag*, which indicates whether a word is an address or not. The D-space is byte-addressed. Each word (32 bits) word-aligned in memory has a bit for an address tag. The load/store operations transfer the entire 33 bit word between memory and registers.

FLATS2 has 64 general purpose registers and 6 floating point registers. Each general purpose register is 33 bit long, 32 bit for data and one bit for the address tag. The general-purpose registers are divided into two groups: 32 global registers and 32 local frame registers. They are mapped into the GV spaces by the Global Frame Pointer (GFP) and the Current Local Frame Pointer(CFP) respectively. Local frame registers are switched by call/return instructions to make function calls faster. The call instruction switches the local frame by incrementing CFP and saving the previous processor status including CFP and the program counter (PC) into the current local frame. Floating-point registers are 64 bits long.

3.1.2 Instructions

The instruction of FLATS2 is 64 bits long. The basic instruction format is shown in Figure 3.2.

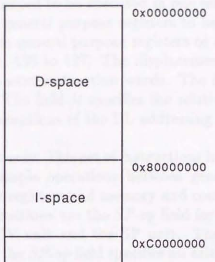


Figure 3.1: Memory space of FLATS2

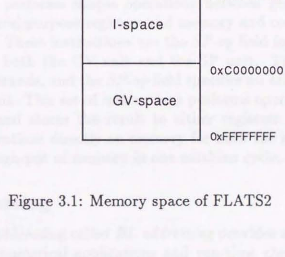


Figure 3.2: Basic instruction format of FLATS2

FLATS has two functional units: *GV unit* and *SP unit.* The GV unit performs simple integer operations between general purpose registers and load/store operations. The SP unit performs integer multiply and divide operations and floating point operations. The *GV-op* and *SP-op* fields control the GV unit and the SP unit respectively. Most instructions in FLATS2 are implemented to be executed in one machine cycle.

The *s1*, *s2*, *s3* fields specify the general purpose registers to be accessed. The *GV-op* field specifies the operations between general purpose registers or addressing modes with the short displacement *d1*, *d2* from -128 to 127. The displacements can be extended to long displacements (32 bits) in the next instruction words. The displacement field may be used as an immediate operand. The field *j1* specifies the relative location for in-word branching. It is also used for the exceptions of the BL addressing mode, to be explained later.

GV instructions use the GV unit only. This set of instructions is basically a *Load/Store* instructions set, which performs simple operations between general purpose registers, load/store between general purpose registers and memory and control operations such as branch and call/return. These instructions use the *SP-op* field for other purposes.

SP instructions use both the GV unit and the SP unit. The *GV-op* field specifies addressing modes of operands, and the *SP-op* field specifies an arithmetic operation to be performed by the SP unit. This set of instructions performs operations between registers or memory operands, and stores the result to either registers or memory. They can perform arithmetic operations directly on memory for both the read and write operands to achieve the full through-put of memory in one machine cycle.

3.1.3 BL Addressing

The FLATS2 memory addressing called *BL addressing* provides architectural support for array computations in numerical applications and run-time checking in Lisp. The BL addressing is the integration of memory addressing and range checking. The effective address is checked against base address and limit address in a register pair during memory access. It allows memory access and range checking to be performed in parallel by hardware. If the effective address is within the range between the given base and limit addresses, memory access completes successfully, otherwise an exception occurs. FLATS2's addressing modes are listed in Table 3.1.

In Table 3.1, *b* specifies the base and limit addresses as a pair, which is called a *BL pair.* A BL pair is formed by an even/odd register pair. With BL addressing, an additional label can be specified in each instruction. If the label is specified, control transfers to the target specified by the label when memory access completes successfully. When an exception occurs, the next instruction is executed to handle the exception. The BL addressing is often used at the end of a loop to terminate the loop where the label specifies the address of the first instruction of the loop. If a label is not specified, the exception of the BL addressing causes a trap.

Both base and limit of a BL pair must have address tag, as well as words used to calculate the effective address. If non-address word is used as an address entity in addressing, an exception also occurs. Only *load effective address* instruction and memory access instruction can calculate an address. Thus, the calculated effective address can

Notation	Effective Address	Side Effect
$\#xxx$	(Immediate)	
gr$n$, vr$n$	(GV register)	
P,Q,R,S,T,U	(SP register)	
b:$disp(p)$	$p + disp$	
b:>$disp(p)$	$p + disp$	$p \mathrel{+}= disp^1$
b:<$disp(p)$	p	$p \mathrel{+}= disp^2$
b@i	$b + i$	
b@>i	$b + i$	$b \mathrel{+}= i^1$
b@<i	b	$b \mathrel{+}= i^2$
b@$disp$	$b + disp$	
b@>$disp$	$b + disp$	$b \mathrel{+}= disp^1$
b@<$disp$	b	$b \mathrel{+}= disp^2$
b@ : $disp(i)$	$b + i + disp$	
b:$disp(p)i$	$p + i + disp$	

NOTE: *b* specifies odd/even register pair as BL.
In effective address and side effect,

b denotes the base register of BL.
i specifies a register as index register.
p specifies a register as address register.
disp specifies a constant as displacement.
1) pre-modify
2) post-modify

Table 3.1: BL addressing mode of FLATS2

never be outside of the given range.

Note that the BL addressing works as memory protection mechanism in FLATS2's single virtual space. At the beginning of the execution of a program, the BL pair of the entire program space is given by the operating system. During the execution, no address outside of the initial BL pair can be produced to avoid the memory access to other program spaces. We call this protection scheme *BL scheme*.

As default, the compiler use the adddressing mode which calculates the effective address of a memory operand by adding the address in a register and a displacement. For the default addressing mode, the BL addressing modes which add the pointer and a displacement is used with the BL pair of the entire program space as default. Figure 3.3 illustrates the register fields in BL addressing mode.

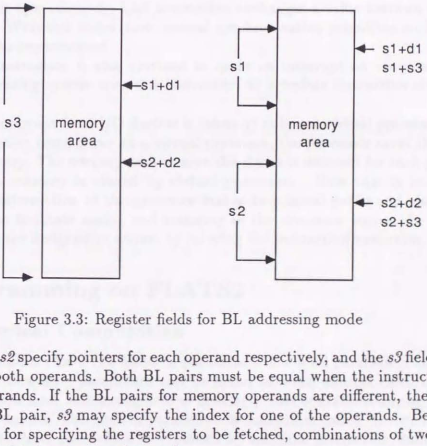


Figure 3.3: Register fields for BL addressing mode

The fields *s1*, *s2* specify pointers for each operand respectively, and the *s3* field specifies the BL pair for both operands. Both BL pairs must be equal when the instruction takes two memory operands. If the BL pairs for memory operands are different, the fields *s1*, *s2* specify each BL pair, *s3* may specify the index for one of the operands. Because the fields are limited for specifying the registers to be fetched, combinations of two memory operands are restricted and not orthogonal.

BL addressing modes can involve *post-modify* and *pre-modify* side effects. The post-modify (pre-modify) side effect is to change the base register or pointer register after (before) the effective address calculation has been done according to the address modes. For example, the post-modify index mode specifies the base address of BL pair as effective address and updates the base register incrementing it by the index register. Unlike autoincrement addressing modes in conventional processors such as VAX or 68000, the BL addressing can modify registers by any displacement in both direction. When the modified address is outside the range of the BL pair in post-modify mode, the address tag of the modified register is set to zero so it cannot be used as an address any more.

Since the address whose most significant bit is one is an address in I space or GV space, it is treated as illegal data address in the memory addressing.

3.1.4 Cyclic Pipeline Architecture of FLATS2

The cyclic pipeline architecture of FLATS2 implements two instruction streams, which share ten pipeline stages. The main memory is not pipelined. Each instruction stream is called a *virtual processor.* The term "processor" means that the processor of each instruction stream has a different set of program counter, registers and processor status including the privileged mode bit. Each processor can be used to execute independent tasks under a multiprocessor operating system.

To synchronize among instructions streams, *Load and Store (LAS)* instruction is provided as an atomic operation; the LAS instruction exchanges a value between a register and memory cell. With this instruction, several synchronization primitives such as locks and barriers can be implemented.

A privileged instruction is also provided to cause an interrupt on every instruction stream. The operating system uses this instruction to schedule instruction streams for user processes.

The external interrupt from I/O devices is taken by only one virtual processor. When interrupt or exception traps occur on a virtual processor, the processor saves the current status in the memory. The memory area to save the status is different for each processor, because the main memory is shared by virtual processors. Note that it is necessary to minimize the information of the processer status in internal paths and simplify the pipeline control to facilitate saving and restoring of the processor status. In FLATS2, most instructions are designed to restart by retrying the instruction execution.

3.2 Programming on FLATS2

3.2.1 Numerical Computation

Numerical computations have two forms of parallelism: low-level parallelism and higher-level parallelism. Low-level parallelism can be found between address calculations and floating points operations, and among several floating-points operations during computations of long expressions. This kind of parallelism is sometimes called "instruction-level" parallelism. Higher-level parallelism is exploited by multiple instruction/data streams. For example, subsequent loop iterations may be independent and can therefore proceed in parallel. Each instruction stream can then execute each iteration independently. Using the FLATS2 FORTRAN compiler, a programmer can express such parallelism to make use of multiple instruction streams provided by the cyclic pipeline architecture.

For low-level parallelism, the BL addressing exploits a limited form of parallelism between address calculations, loop control operations and floating point operations in array computations. For example, the fragment of a program shown below,

```
      DO 10 I = 1,N
```

```
10      S = S + A(I)
```

can be executed by the following code using BL addressing:

```
            ; the variable S is allocated in the register T
            movw address of A(1), gr0
            movw address of A(N), gr1
L:          add3.d.j T,gr0@<8,T,L
```

The fragment computes the sum of the array. Before entering the loop, the BL pair for the array is formed by the address at the initial element and the address at the last element. In the loop, the BL addressing performs the following operations:

1. Check the address tags of BL pair.
2. Check whether the effective address is between base and limit of the BL pair.
3. Fetch the memory operand from the location specified by the addressing mode. In this case, the operand at the base address of the BL pair.
4. Modify the base register as a side effect. If the modified address is outside of the BL pair, the address tag of the base register is cleared.
5. Branch to the first instruction of the loop if memory access completes successfully. Otherwise, execute the following instruction to terminate the loop.

At the last iteration, the address tag of the base register is cleared. The BL addressing in the subsequent iteration detects the invalid base register to terminate the loop. Note that all of above operations are performed in one instruction cycle. As a result, the sum of the array is computed by one-instruction loop. The FLATS2 FORTRAN compiler automatically optimizes such array processing code with BL addressing.

3.2.2 Run-Time Checking in Lisp

We can make use of BL addressing and address tag to reduce the cost of run-time checking in Lisp. In Lisp, an object is represented as a pointer or an immediate value. An address tag distinguishes a pointer object from an immediate data type such as fixnum. Lisp generic arithmetic instructions of FLATS2 check the address tags of operands to perform the operation according to its data type.

By allocating the same types of objects in a segment, BL addressing checks the type of a pointer object during memory access by testing which segment the pointer points into. In dynamically typed languages such as Lisp, APL, and Icon, all data objects are allocated dynamically at run-time, and their types must be checked at run-time. Objects in memory are referenced indirectly through pointers. The objects can be allocated in the heap space associated with its type. We call the heap space corresponding to each type, a *type segment*. The type of the pointer can then be checked by testing which type segment the pointer points into.

The compiler knows the expected data type and how to access the object in memory through the pointer. By using BL addressing mode with the BL pair of the type segment, the type can be checked in parallel with data access. For example in Figure 3.4, a cons cell in Lisp is represented by two words in the cons cell segment, where `BL-cons` indicates. The primitive operations *car, cdr* on a pointer p to the cons cell are performed by loading from memory with BL addressing mode respectively as follows:

*car(*p*)* := `BL-cons:(p)`

*cdr(*p*)* := `BL-cons:4(p)`

Here the size of a word is 4 bytes. If p points outside of `BL-cons`, a trap occurs.

The predicate on list data type, *listp*, is implemented simply by range checking on a pointer with the BL of cons cell segment.

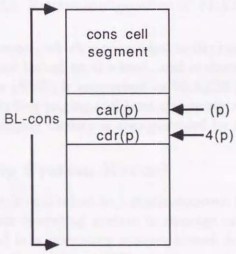


Figure 3.4: CAR/CDR operations by BL addressing

We can check data type by range checking instead of tag checking found in the tagged architectures such as FLATS[SHS+87]. This scheme are not specific to Lisp, so that it can be used more generally than other tagged architecture.

We can also make use of this scheme to check the number of arguments and multiple return values. The design of a Lisp system using BL addressing is described in Appendix A.

3.3 Programming Environment

3.3.1 FLATS2 System

The configuration of FLATS2 is illustrated in Figure 3.5.

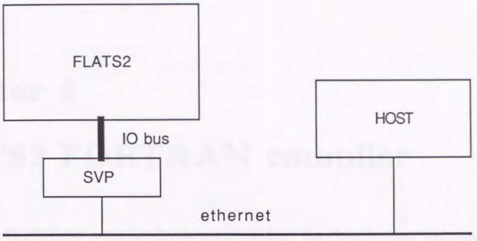


Figure 3.5: System configuration of FLATS2

FLATS2 is a back-end processor, which is connected to the host computer via Ethernet. A user program is compiled and linked on the host, and is down-loaded into FLATS2 to execute. The service processor (SVP) is connected to FLATS2 through the external bus to perform I/O operations including paging and boot strapping, requested by the FLATS2 processor. The hardware debugging facility is also provided by the SVP.

3.3.2 CPX Operating System Kernel

Since a cyclic pipeline machine is equivalent to a multiprocessor from user's point of view, FLATS2 needs a multiprocessor operating system to manage user programs.

CPX [SSFG89b, SSFG89a] is a operating system kernel designed for FLATS2. To exploit parallelism in the operating system, CPX provides an object oriented interface for its operating system functions. Currently, a simple operating system is implemented on CPX to execute down-loaded programs on FLATS2.

3.3.3 FLATS2 instruction-level simulator

We have also developed an instruction-level simulator for FLATS2. All cross development tools on host such as assembler, linker and compiler, were developed using the simulator. The FLATS2 debugger, which is similar to the UNIX debugger *adb*, provides the same user interface to both the simulator and FLATS2 machine. As well as debugging programs, the simulator can be used to measure the performance of various configurations of the FLATS2 architecture. For example, the number of instruction streams can vary as a parameter given by users. The simulator can measure the precise count of executed instructions. The results of several experiments in this thesis was measured in the simulator.

Chapter 4

FLATS2 FORTRAN compiler

The FLATS2 FORTRAN compiler implements most of the optimization techniques including common subexpression elimination, constant folding, code motion, and strength reduction. To facilitate optimizations, the compiler converts the code into Static Single Assignment (SSA) form[CFR$^+$89] to perform machine independent optimizations efficiently and simpler. The internal code is represented in register transfer language (RTL)[DF84], which enables machine dependent optimizations such as peep hole optimization to be done in a machine independent way. The compiler generates simple code from the intermediate code of the front-end, and then these codes are improved to more compact instructions of FLATS2 by an extensive peep hole optimization called *code reconstruction* through the optimization. For register allocation, we present a global register allocation algorithm, which allocates registers locally using the approximation of the global register allocation by the priority coloring algorithm. While the registers are allocated globally, the local register allocation of our algorithm can select the good instruction to reference the variable in memory. The algorithm can allocate variables in a small number of floating-point registers in FLATS2 as well as the general purpose registers. In this chapter, we describe the overview and algorithms of the FLATS2 FORTRAN compiler.

4.1 Overview

Figure 4.1 shows the organization of FLATS2 FORTRAN compiler.

The front-end does the lexical analysis, parsing, and symbol table maintenance. Like many retargetable compilers, it compiles source code into an intermediate code, which is similar to the PCC intermediate code [Joh81]. The front end of the FLATS2 compiler is similar to UNIX FORTRAN 77 compiler, except that it produces special code for BL checking in a DO loop. The BL optimization requires special handling for loop variables. Constant folding and associative-low simplifications are also done during this pass.

The next phase expands the intermediate code into register transfer language (RTL) code [DF84], a representation roughly equivalent to assemble code. For a non-optimizing compilation, assembly code is generated directly during this phase instead of RTL code. In RTL code, the instructions to be output are represented in an algebraic form that indicates what the instruction does. It has a textual form that is used for printing debugging dumps. Like Lisp lists, the textual forms uses nested parentheses to indicate the pointers in the

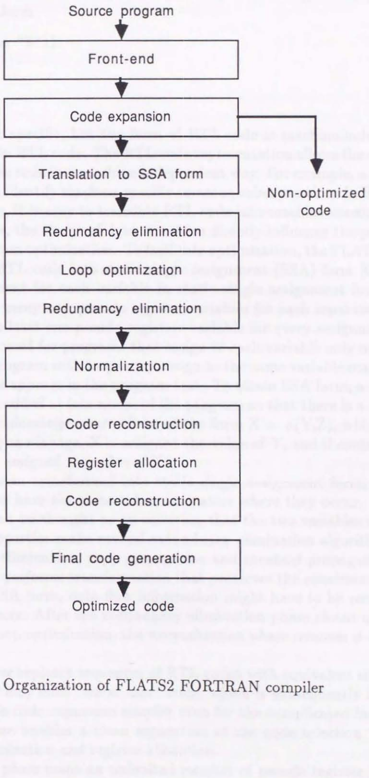


Figure 4.1: Organization of FLATS2 FORTRAN compiler

internal form. For example, if the instruction is represented by the assembly code

```
    movw @X,r0
```

which loads the register with the memory cell labeled X, then it might be represented with the RTL code in list form

```
    (set (reg 0) (name "X"))
```

or, in expression form

```
    r[0] = m[X]
```

Any RTL code is machine specific, but the form of RTL code is machine-independent. Most optimization is done in RTL code. The RTL code representation allows the optimizer to optimize machine-specific code in a machine-independent way. For example, a machine-independent algorithm can identify machine-specific common subexpressions in RTL code. After several optimizations, it is easy to translate RTL code into machine assembly code.

In optimizing compilers, the choice of data structure directly influence the power and efficiency of practical program optimization. To facilitate optimization, the FLATS2 FORTRAN compiler changes RTL code into static single assignment (SSA) form [CFR+89]. There is only one assignment for each variable in static single assignment form. This transformation introduces many new pseudo register variables for each separate variable in the original program, at least one pseudo registers variable for every assignment. The phase *single assignment* is used for programs that assign to each variable only once when running. Dynamically, a program with loops may assign to the same variable many times, even if only one assignment appears in the program text. To attain SSA form, a new type of assignment statement is added at join nodes of the program so that there is a dominating assignment. These *pseudo-assignment* will be of the form $X = \phi(Y,Z)$, which means that if control enters along one inedge, X is assigned the value of Y, and if control enters along another inedge, X is assigned the value of Z.

After a program has been transformed into static single assignment form, lexically identical expressions always have the same value, no matter where they occur. A trivial assignment in SSA form can be thought as an assertion that the two variables represent the same value. These properties make several redundancy elimination algorithms such as common subexpression elimination, copy propagation and constant propagation more simple. Loop optimization performs transformation that preserves the consistency of this representation. Without SSA form, data flow information might have to be recomputed each time code motion occurs. After the redundancy elimination phase cleans up redundancies generated by the loop optimization, the *normalization phase* removes ϕ-functions in join nodes.

The *reconstruction phase* replaces sequences of RTL codes with equivalent singletons. The code expansion phase may emit "worst case" code, which is subsequently improved by this phase. It makes the code expansion simpler even for the complicated instruction set of FLATS2. This phase enables a clean separation of the code selection from the machine-independent optimization and register allocation.

The *register allocation phase* maps an unlimited number of pseudo register variables onto a finite set of registers provided by the hardware, introducing spill codes where needed. After the register allocation, the reconstruction phase is executed again to combine spill codes with other instructions. The *final code generation phase* translates optimized RTL code into assembly code.

The following sections describe the compiler phases in detail except the induction variable elimination, which is described in the next chapter.

4.2 Code Expansion Phase

The code expansion phase translates the expression trees of intermediate code produced by the front-end into RTL code. The code expansion process is similar to the code generation of PCC. A set of templates is given to describe the effect of the target machine instructions. The code expansion is done by matching the tree of codes and its context with the templates. If a template is found to match the tree and context, the associated instruction is generated. The tree is then rewritten, as specified by the template, to represent the effect of the generated instruction. If no template match is found, the matching routine is called recursively for a subgoal of the computation by default rewriting rules.

The template matching process performs case analysis for code selection. Some machines may have special-case instructions like increment and decrement instruction that are cheaper than adds and subtracts, and *autoincrement* addressing mode which dereferences a pointer and automatically advances it to the next item. For a "large" instruction set, we often need different patterns for different classes of operands and instructions.

Because some FLATS2 instructions are non-orthogonal in operands, a code generator would need to store and check contextual data to yield good code. For example, the FLATS2 floating-point instruction can take both of read memory operands and write memory operand for a restricted combination of addressing modes. The right choice for any particular instruction depends on which of its operands is available in a register, and which combination of these memory addressing modes can be used. Such machine-specific context checking requires much case analysis, and works against a clean separation of the instruction selection from the register allocation.

Our compiler leaves most of such case analysis to the subsequent phase. Code quality is achieved through optimization, so the templates for code expansion need not describe the full set of instructions. This makes the code expansion phase simpler, and reduces the number of templates. The templates of the FLATS2 instruction set includes an orthogonal subset with simple addressing modes.

The compiler also leaves register allocation to the optimizer. Registers are chosen from an infinite set, and register assignment is performed by the register allocation phase. The front-end and the code expansion phase allocate user-specified local variables and compiler-generated temporaries to pseudo registers. References to global variables in the source program is converted to reference to the labeled memory cell. In RTL code, we consider a pseudo register to be a variable. In FORTRAN, there are no explicit pointer variables to local variables. Only subroutine call statement must take care of aliasing and side effects on local variables passed to other subroutines. Thus, all variables can be thought to be unaliased in the subsequent optimizations.

For non-optimized compilation, register allocation is done locally in each statement, since actual instructions are generated at this phase.

4.3 Translation into Static Single Assignment Form

In this section, we describe the basic data structure used in the compiler and the translation into SSA form.

4.3.1 Build program flow graph

The low level representation upon which optimizations are performed consists a double-linked list of machine instructions. Each instruction entry in the list contains the RTL code associated with the machine instruction as well as additional information required by the optimizer and the assembly code generator.

After code expansion, the instructions in the list are partitioned into basic blocks. These comprise the basic unit for which local data flow information is calculated. The control flow graph is built with a node for each basic block and an edge for each transfer of control. We assume that all nodes are reachable from the node representing the program entry, and that each node has at most two outedges. These assumptions are not crucial, but they are convenient in several places[1]. A *join* node is a node that has two or more inedges. For simplicity, the number of inedges is reduced to one or two by inserting dummy nodes.

A *backedge* of the flow graph is any edge whose destination is an ancestor of its source in the tree defined by depth-first search [Tar70] rooted at the program entry node. (In a reducible graph, the set of backedges does not depend on the arbitrary choices made during depth-first search [HU74].) With backedges ignored, the graph becomes a DAG and may be topologically sorted. Throughout this chapter, the *topsort* order will be the reverse of the order in which subsearches terminate.

A *loop header* is any node that is the destination of a backedge. Given a loop header $\mathtt{h}$ reached by backedges from nodes $\mathtt{s}_1, ..., \mathtt{s}_k$, the corresponding *loop body* consists of all nodes $\mathtt{u}$ such that there is a path of the form $\mathtt{h} \xrightarrow{*} \mathtt{u} \xrightarrow{*} \mathtt{s}_i$ that traverses no backedges. An edge from a node in the loop body to a node not in the loop body is an *exit edge* of the loop. The destination of an exit edges is an *exit node* of the loop.

There are several ways to compute the loop body and loop exit edges for each loop without tracing path. (In a reducible graph[2], the edges that enter the loop are just the inedges of the header that are not backedges.) We first determine the nodes within the loop with header $\mathtt{h}$ by searching the graph, starting at the sources of the backedges to $\mathtt{h}$. The search proceeds backwards (from destination to source), and ignores backedges. The search is terminated at each branch that reaches $\mathtt{h}$ or previously visited node. During the search, each node visited is marked as being in the loop body for $\mathtt{h}$. Exit edges can be

[1]Except for the computed GOTO statement, a multi-way branch can be reduced by a two-way branch in FORTRAN.

[2]We assume only reducible graphs for optimization. The current version of the compiler does not optimize for nonreducible flow graphs.

determined by examing nodes in the loop body for any outedges that do not go the other nodes in the loop body.

4.3.2 The graph for DO loop

Several transformations require to move instructions "before the header". We therefore give each loop L a *landing pad* representing the entry to the loop from outside. The landing pad has only the header as successor, and all edges which formerly entered the header of L from outside L instead enter the landing pad. Edges from inside loop L to the header are not changed. Because landing pads are new nodes added for every loop header, no node will be both a landing pad and a loop header. Initially, the landing pad is empty, but transformations on L may place instructions in it.

Landing pad insertion is illustrated in Figure 4.2 for a `DO` loop.

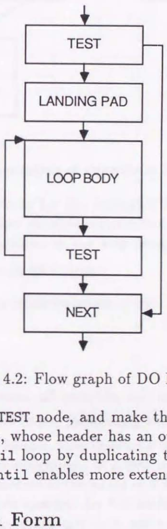


Figure 4.2: Flow graph of DO loop

The front-end duplicates the `TEST` node, and make the loop look like an `until` loop guarded by a `TEST`. A `while` loop, whose header has an outedge that is an exit from the loop, can be converted to an `until` loop by duplicating the `TEST` header node [CLZ86]. Making `DO` and `while` look like `until` enables more extensive optimizations.

4.3.3 Translate to SSA Form

In RTL code, each statement is an instruction expressed in RTL. We rename variables throughout the program to put it into *static single assignment (SSA)* form. In SSA form, each variables is assigned to exactly once in the program text. Special statements called

ϕ-functions are added at join nodes to indicate that a variable is assigned the value of one variable (if control enters along one inedge) or another variable (if control enters along another inedge).

Each mention of a variable V will be replaced by a mention of one of the new names for V. After renaming, every use for a variable V will be reached by exactly one definition of the variable. For convenience, the various new names will be denoted V_i where i is an integer. The example of renaming is illustrated in Figure 4.3.

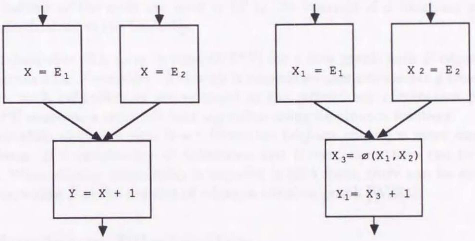


Figure 4.3: Renaming of variable in SSA form

Algorithm 4.1 computes SSA form for the reducible flow graph. This algorithm is essentially from [RWZ88], where live variable information is not used. Live variable information prevents irrelevant ϕ functions at the loop header.

Algorithm 4.1. Translation to SSA form.

Input. A flow graph with live variable information.

Output. A SSA form.

Method.

1. At the start of the program, all variables are undefined.
2. Visit the nodes in topsort order, performing the following steps for each node:
   (a) If the node is a loop header, then insert a ϕ function for each live-in variable V. The target of the assignment is a new name generated for V. The first operand of the ϕ function is the name of V that reaches the node from the landing pad. Another operand for the backedge will be filled in later.
   (b) If the node is a join node and it is not a loop header, then insert a ϕ function for each live-in variable V for which two different names reach the inedges of the node. The new assignment has the form $V_k = \phi(V_i, V_j)$, where V_i,V_j are two names of V that reach the left and right edges of the node and V_k is a new generated name for V.

(c) For each assignment of V, generate a new name for V to replace it. This name is the one that reaches the point p immediately after the assignment, and an other point q such that every path from p to q is free of assignments to the name of V.

(d) For each use of V in a statement, replace it with the name for V which reaches the statement. If the name of V is undefined, it means that an undefined value will be used.

(e) If the node is the source of a backedge, then the names that reach the bottom of the node are used to fill in the operand of ϕ functions at the destination of the backedge.

This algorithm computes SSA form in time $O(E^*V)$ for a flow graph with E edges that describes a program with V variables. Although it sometimes uses extraneous ϕ functions in loop headers, such redundancies are removed in the redundancy elimination phase. Cytron [CFR+89] proposes a more efficient algorithm using dominance frontiers.

The representation of simple data flow information (*def-use chains*) is more compact through SSA form. If a variable has D definitions and U uses, then there can be D^*U def-use chains. When similar information is encoded in SSA form, there can be at most E def-use chains, where E is the number of edges in the flow graph [WZ84].

4.4 Redundancy Elimination

In this section, we present several redundancy elimination algorithms to optimize code on SSA form. The SSA form makes conventional optimizations simpler than the conventional representation of code without SSA form. The redundancy elimination includes the following optimizations:

- Constant propagation and constant folding.
- Global common subexpression elimination.
- Copy propagation.
- Loop-invariant code motion.

4.4.1 Elimination of Trivial Assignments and Common Subexpression

Assignment statements that have trivial right-hand side (a single variable or a constant) have a special meaning in SSA form. Assignment statements whose right-hand side is a variable can be thought of as assertion that the two variables represent the same value. A trivial assignment statement, `X = c` with a constant `c`, means that the value of `X` is `c` at any use. Given a trivial assignment `X = Y`, we replace every occurrence of one variable (including those in ϕ function) with the other. This simple process works as copy propagation, while the standard copy propagation needs the data-flow computation for copy assignments.

Thanks to SSA form, identical computations in the text produce the same value. It makes the global common subexpression elimination more simple. The matching computations are really equivalent in SSA form. If statement s1 appears on every path from the entry to statement s2 in the flow graph, s1 *dominates* s2. If there are two statements X = E and Y = E where E is a common expression, we replace Y = E with Y = X when statement X = E dominates Y = E in the flow graph.

The initial list of trivial assignments to be removed includes any statements that appear originally in the program. There may also be several computations that happen to have identical right-hand sides initially. Renaming of operands in expressions may make the right-hand side become identical with the right-hand side of another computation. Elimination of common subexpression creates a trivial assignment to be removed, which is added to the worklist of trivial assignments. Removal of trivial assignment and elimination of common expression feed each other until the worklist are empty.

Renaming of the operands of ϕ functions may create a trivial assignment as follows:

- $\mathtt{X} = \phi(\mathtt{Y}, \mathtt{Y})$ is replaced by $\mathtt{X} = \mathtt{Y}$.
- $\mathtt{X} = \phi(\mathtt{X}, \mathtt{Y})$ is replaced by $\mathtt{X} = \mathtt{Y}$.

In both cases, the new assignment is added to the worklist of trivial assignments.

To facilitate these replacements for trivial assignments, we maintain a list of uses of each variable in the program. After the replacement, the use list of the replaced variable is concatenated to the use list of the other variable. A *definition statement DEF(X)* of a variable X is an assignment statement where X is defined. By the definition of SSA form, X is defined by exactly one statement. A *use list USE(X)* is a set of statements where X is used as operand.

Algorithm 4.2. Elimination of trivial assignment and common subexpression in SSA form.

Input. A flow graph in SSA form with *DEF(X)* and *USE(X)* for each variable.

Output. A revised flow graph.

Method. Removal of a trivial assignment from the worklist W is performed until W is empty, starting with the initial list of trivial assignments. The algorithm is sketched in Figure 4.4. Common subexpressions are searched in each use list, when removal of trivial assignments changes the use list. The details of eliminating common expressions are as follows:

1. Find statements which have identical computations in right-hand side within the use list, say $\mathtt{V_i} = \mathtt{E}$ where E is a common expression and i runs from 1 to n.
2. Replace each of the statement by a trivial assignment of the form $\mathtt{V_j} = \mathtt{V_i}$ if the assignment of $\mathtt{V_i}$ dominates the assignment of $\mathtt{V_j}$.
3. Put these trivial assignments on the worklist W for removal.

```
W = ∅
/* collect initial trivial assignment */
for each variable X do
  eliminate common subexpressions in USE(X)
  if DEF(X) is a trivial assignment then W = W + DEF(X)
end

while W is not empty do begin
  take a statement X = Y from W;
  for each statement S in USE(X) do
     replace X with Y;
  USE(Y) = USE(Y) + USE(X);
  USE(X) = ∅;
  delete DEF(X);
  eliminate common subexpressions in USE(Y);
  for each statement S in USE(Y) do begin
     do constant folding on S if possible
     if S is a trivial assignment then put S into W
  end
end
```

Figure 4.4: Redundancy elimination

The ϕ functions distinguish values of variables transmitted on distinct incoming control flow edges. It enables a *global value numbering* algorithm to track redundant computations access flow graph paths. The redundancy elimination by Rosen [RWZ88] eliminates global redundancies involved in a loop using the global value numbering approach. Our algorithm is rather conservative one and eliminates such redundancies in loop invariant code motion of the separate phase explained in the next subsection.

4.4.2 Loop Invariant Code Motion

Code motion is an important transformation which decreases the execution time. This transformation takes an expression that gives the same result at every iteration of a loop and places the expression in the landing pad.

A *loop invariant statement* is a statement whose operands are either constant or have all their reaching definitions outside the loop. Our algorithm starts with assuming that all variables are invariant. Since each variable has exactly one definition in SSA form, we can find loop variant variables by marking all variables defined in the loop as "variant". The variable without mark is the loop invariant variable. Once we find the statement whose operands are loop invariant, the statement is moved into the landing pad and the variable defined by the statement is unmarked.

Algorithm 4.3. Loop-invariant code motion in SSA form.

Input. A Loop L in SSA form.

Output. A revised loop.

Method.

1. Mark "variant" those variables defined within L.
2. Repeat the following step until at some repetition no code is moved to the landing pad of L:
   (a) Find loop-invariant code for each statement of the form `X = E`. If all operands of `E` are either constants or unmarked variables, `E` is a loop invariant expression [3].
   (b) If `X` is a variable, move the statement in the landing pad. After movement, the variable is unmarked.
   (c) If `X` references memory, generate a new name `Y` to replace the expression. Put a new statement `Y = E` in the landing pad.

This algorithm may increase the execution time a bit when some nodes where statements are moved do not dominates all exit nodes of L. To prevent this situation, we have to add the condition that the moved statement dominates all exit of L.

When loops are nested, loop-invariant code motion is performed from the inner loop. Since the algorithm does not use data-flow information such as use-def chains, it is not necessary to maintain data-flow information for each loop. The SSA form nicely summarizes the conditions relevant to code motion. An advanced constant propagation algorithm is proposed to delete branches to code proven unexecutable at compile-time [WZ84]. Without SSA form, data-flow information may have to be recomputed each code motion occurs.

Loop invariant information computed in this phase is also used for the loop induction variable elimination, described in the next chapter.

4.4.3 Alias Analysis in Loop

If an expression includes a memory reference, we need alias analysis to determine whether the expression is invariant in the loop. For example, in the following loop

```
      DO 10 I = 1,N
10    S = A(J) + B(I)
```

`A(J)` is a loop-invariant memory reference. The value can be referenced instead in the register loaded from the memory reference in the landing pad.

In RTL code, reference to a global variable is represented as reference to the memory cell. If the global variable has no alias in the loop, it can be assigned to a register in the landing pad to reference its value in the register within the loop.

[3]Some division statements must not be computed in the landing pad as an invariant code. For example, if a division `x/y` in a loop preceded by a test to see whether `y=0`, the division moved in a landing pad may cause a division by zero unintentionally.

Although the presence of pointers usually makes data-flow analysis more complex, the SSA form enables us to track the effect of pointer assignments across the flow graph. We make some assumptions about arithmetic operations on pointers. If p points to a memory area, addition or subtraction by an integer produces a pointer value pointing somewhere in the same memory area. Other arithmetic operations on pointers produce a value that is not pointer, and sometimes meaningless. Although the programmer can move a pointer from one array a to anther array b by adding to the pointer, such action would depend on the particular implementation to make sure that the array b follows a in storage. The optimization assumes the programmer does not depend on the particular order for data in memory. In FORTRAN, data is grouped in the memory area of a common block labeled with its block name.

If p receives the value pointing to a memory area, the SSA form makes sure that p points to the memory area at every use of p. The ϕ function implies that the variable may have both values of operands transmitted on distinct inedges. The following algorithm computes the set *REF(p)* of memory area to which the pointer p may point.

Algorithm 4.4. Alias computation in SSA form

Input. A flow graph in SSA form with *DEF(p)* for each pointer.

Output. *REF(X)* for each pointer.

Method. We use the recursive function *FINDREF(p)* of Figure 4.5, which computes *REF(p)* for each pointer p. Before each call of *FINDREF(p)*, all pointers are unmarked "visited" to terminate recursive calls.

During computation we can use $REF(p)$ for search on p to save the time if $REF(p)$ is already computed. While the algorithm computes $REF(p)$ in time $O(E * V)$ where E is the number of edges and V is the number of pointers.

An indirect assignment by p may modify the memory area of $REF(p)$. The set of modified memory area in the loop is the union of the memory area modified by all statements. If p is a loop-invariant variable and the memory area of $REF(p)$ is not modified in the loop, then the memory reference by p gives the same value at every iterations. As the code motion of loop-invariant variables, the memory reference to the memory area which is not modified in the loop can be moved in the landing pad. It sometimes enables other expressions to move in the landing pad. Since unlike variables in SSA form, the memory area can be modified by more than one assignments in the loop, the data-flow computation is needed to move an assignment to memory.

If no alias is found for a global variable within the loop, it can be assigned to a register. The reference to the global variable is replaced with a compiler-generated variable, and codes are inserted to load its value into the register in the landing pad and update the global variable at each exit node. This transformation reduces expensive memory access for the global variable in the loop. Since reference to an array element with constant index is represented as reference to the memory cell as same as the global variable, above transformation can be applied for such memory reference.

Alias analysis across procedure calls requires interprocedural information; interprocedural analysis is not performed in our compiler. If the loop contains calls, we assume that they could modify any memory area.

```
function $FINDREF(p)$
begin
   if $p$ is "visited" then return $\emptyset$ /* empty set */
   mark $p$ "visited";
   if $DEF(p)$ is a form $p = A \pm c$
      where $A$ is a label of memory and $c$ is an integer offset then
      return $\{A\}$;
   if $DEF(p)$ is a form $p = q \pm c$
      where $q$ is a pointer and $c$ is an integer offset then
      return $FINDREF(q)$
   if $DEF(p)$ is a form $p = \phi(q, r)$ then
      return $FINDREF(q) \cup FINDREF(r)$
   else
      the pointer $p$ is meaningless;
end
```

Figure 4.5: Alias computation

4.4.4 Normalization of SSA form

After optimization on SSA form, the normalization phase eliminates ϕ functions. Every computation of the form $X = \phi(Y, Z)$ is replaced by an assignment $X = Y$ on one of the entering branches, and by $X = Z$ on the other. Each assignment is placed at the end of the code.

Many variables can be merged together by using their *live range* information. If two variables have disjoint live ranges, then they can be merged into one variable. If each live range for the variables in $X = \phi(Y, Z)$ is disjoint to each other, this variables is replaced with one variable so that no additional statement is placed. An induction variable in the loop is often the case. The live range information is also used for coalescing the source and target in statement to recognize the simple increments of variables.

4.5 Code Reconstruction

While the object code optimizer is machine independent and general, peephole optimization is often machine specific and ad hoc. The RTL code avoids machine dependencies in peep optimization.

The FLATS2 instruction set is too complicated to emit optimal code by simple pattern matching in the code expansion phase. This phase replaces sequences of instructions with equivalent "larger" instructions to achieves code quality. Because most instructions are executed in the same time in FLATS2, larger instructions are executed faster than the sequence of simpler instructions. Since final instructions are reconstructed from simpler instructions, we call this phase *code reconstruction*. This phase is called before and after the register allocation phase.

4.5.1 Combining Instructions

The code reconstruction phase seeks a pair of instructions in RTL code that can be replaced with singletons. It symbolically simulates these instructions to learn their combined effect and searches for an instruction with this combined effect by machine-dependent routines, called *instruction recognizer*. If it finds one, it replaces the original instructions with the singleton. The code reconstruction phase computes the combined effect of two instructions by substituting the values assigned to cells in the first for occurrences of those cells in the second. For example, consider the following instructions;

```
r[1] = m[r[0]]
r[2] = r[2] + r[1]
```

where `r[1]` is dead at the second. Their combined effect is computed by replacing the `r[1]` in the second instructions with the value assigned to `r[1]` in the first, and then concatenating the two effect. This yields

```
r[2] = r[2] + m[r[0]]
```

If `r[1]` is live after the second instruction, its replacement cannot be performed. A conventional peep hole optimizer correct only those sequences that match a few hand-written, machine-specific patterns. The code reconstruction phase combines all related pairs in machine-independent manner.

This replacement must treat an assignment to the program counter as a special case. Since operations for these registers such as branch or push/pop on the stack are defined as operators in RTL code, the program counter and stack pointer does not appear explicitly. The condition codes can not be used explicitly.

For autoincrement addressing modes, we treat a simple increment instruction specially. For example, the following codes can be combined into an autoincrement addressing mode.

```
r[2] = r[2] + m[r[1]]
r[1] = r[1] + C
```

yields

```
r[2] = r[2] + m[r[1]]; r[1] = r[1] + C
```

where these assignments are done in parallel [4]. Such a simple increment is recognized in the normalization phase of SSA form by live range information.

Some FLATS2 instructions can perform operation directly on memory operands, and store the result in memory in some operands combinations. The code expansion phase

[4]In RTL code, the code for autoincrement addressing is expressed as `m[r[1] = r[1] + C]`.

emits code using only one memory operand to simplify code expansion process. The reconstruction phase combines a store operation with other instructions. It also combines an increment with memory addressing in other instructions to make use of autoincrement addressing mode of the BL addressing. Even though the available operand combinations of the BL addressing modes are non-orthogonal because of a fixed length of the FLATS2 instruction, the case analysis is done in machine-independent manner.

Davidson and Fraser's YC compiler [DF84] uses a similar approach on peephole optimization by using the table which is automatically generated by the machine description. We used a function instead, which is given by a compiler writer, to recognize the instruction. This phase is retargeted by supplying a machine-dependent instruction recognizer. As well as adjacent instructions in text, the code reconstruction phase moves an instruction as needed to increase the chance of combination using a machine instruction dependency graph.

4.5.2 Instruction Reorganization

Since a related pair of instructions may be separated by other instructions, instruction reorganization increases the chance of combination for such a pair. The reorganization problem has been discussed by many researchers [HC83] [TTT81], related to compile-time pipeline scheduling and compacting of microcode. Like peephole optimization, microcode optimization improves the code by compacting the vertical microoperations into horizontal microinstructions where these operations do not overlap in resource utilization. The peephole optimization, however, often deals with vertical aspects of instruction ordering, involving data dependencies between instructions. It works as a part of the compiler like pipeline scheduling. Possible combinations of peephole optimization is not dynamic, while pipeline reorganization concerns interlocks whose effect is dynamic since the context of a particular instruction determines whether or not that instruction is legal in its current position.

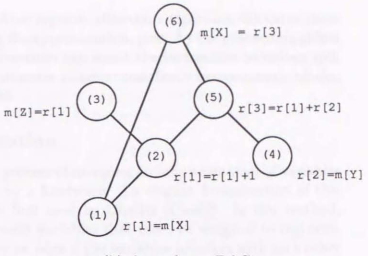
Clearly, instructions in a program cannot be reordered arbitrarily. Certain instruction must remain ahead of other instructions in the resulting code sequence for the overall effect of the program to remain unchanced. To express the constraint to rearrange instructions without compromising correctness, we construct for each basic block a directed acyclic graph (DAG) whose nodes are the instructions in the block and whose edges represent serialization dependencies between instructions. An edge from instruction i to instruction j indicates that i must be executed before j to preserve correctness. The DAG takes into account all serialization constraints, including register dependencies, memory dependencies, control transfer instructions such as calls and branches. If the variable which is live at the end of the block, we allocate a special node as the successor to represent the liveness of the variable. An example code sequence and its dependency DAG are shown in Figure 4.6.

We serialize definition vs. definitions on any particular resource such as a register to simplify selecting an order without dead lock. Our DAG is the same as that used for the reorganization for pipeline scheduling proposed by Gibbsons [GM86].

As long as the instructions in the basic block are reordered in some topological sort order of the dependency DAG, the overall effect of the block is the same as its execution

```
(1) r[1] = m[X]
(2) r[1] = r[1] + 1
(3) r[Z] = r[1]
(4) r[2] = m[Y]
(5) r[3] = r[1] + r[2]
(6) r[X] = r[3]
```

(a) code



(b) dependency DAG

Figure 4.6: Dependency DAG

in the original order. The algorithm reorders the instructions to find a related pair of instructions to be combined with dependency DAG.

Algorithm 4.5. Code reconstruction with dependency DAG.

Input. A basic block with its dependency DAG.

Output. A revised basic block.

Method. Repeat the following steps until all instructions are reordered:

1. An instruction is a *candidate* for reordering if all its immediate predecessors in the DAG have been reordered.
2. Among the candidates, select an instruction. (We can select the instruction which is executed first in the original order.)
3. Combine the selected instruction with its successor in the DAG if possible. Since the definition is dead after all its uses, the definition have to be used by only one successor. If the instruction is combined to the other instruction, delete it from the DAG. Otherwise, reorder the instruction.

Since the dependency DAG is computed in $O(N)$ where N is the number of instructions in the basic block, the algorithm reorders the instructions in $O(N)$.

4.6 Register Allocation

The global register allocation is successful in the machine where the instruction set is regular and a large number of registers are available. FLATS2 has a few floating-point registers and their usage is restricted. Our register allocation algorithm allocates these registers locally in each basic block using the approximation given by the proceeding global register allocation. The local register allocation can select the instruction to reduce spill code for the registers. Then it inserts spill codes to keep consistency between basic blocks. It can allocate a large register set as well.

4.6.1 Global Register Allocation

Register allocation can be viewed as the process of mapping an infinite number of variables into the finite set of registers provided by a hardware. An elegant formalization of this problem is the graph-coloring approach first used by Chaitin [Cha82]. In this method, the nodes in an *interference graph* represent variables that must be assigned to registers. Two nodes in the graph are connected by an edge if the variables interfere with each other in such a way that they must reside in different registers. A coloring of a graph is an assignment of a color to each node of the graph in such a manner that each two nodes connected by an edge do not have the same color. In coloring the interference graph, the number of colors used for coloring, r, is the number of registers available for use in register allocation.

The standard coloring algorithm to determine whether a graph is *r-colorable* is NP-complete. Many graphs cannot be colored because the minimum number of colors required (the *chromatic number*) is greater than the number of available registers. A practical register allocator does not look for an optimal coloring, but rather for a correct and feasible one. When a graph's chromatic number exceeds the number of hardware registers by causing some variables to reside in memory, rather than a register, the allocator decides which variables to spill to enable a coloring.

The allocator operates on RTL code, which is machine-dependent, rather than a machine independent intermediate representation. We refer to both user-defined and compiler-generated temporaries as "variables" in this section, because RTL code makes no distinction two, which are represented by pseudo-register variables.

The FLATS2 FORTRAN compiler uses a graph-coloring algorithm called *priority-based coloring*, developed by Chow [CH84]. Each variable is assigned a priority that is the estimated additional cost if the variable resides in memory rather than in a register. Variables with more than r are assigned to registers in decreasing order of priority. A variable that cannot be assigned to a register because r or more of its neighbors have been colored must be spilt and spill code introduced. A variable is spilt by dividing the set of blocks in which the variable is live into two sets and allocating separately in each.

4.6.2 Register Allocation on FLATS2

The global register allocation is successful for a large register sets. FLATS2 has 64 general purpose registers (GV registers), which consists of 32 global registers and 32 local frame registers. Four registers of the local frame registers hold a return address and procedure status informations. The register allocator uses 15 of the 32 global registers. Integer and address variables are allocated to GV registers.

Because the local frame registers are saved/restored by the CALL/RETURN instructions, a variable whose live range includes procedure calls should be assigned to a local frame register. Such variables belong to a different class of registers. The coloring algorithm can easily extend to the case of multiple register classes. The interference graph will only give interferences between variables of the overlapped classes.

The FLATS2 register set has the following problems on register allocation:

1. A BL pair must be allocated in an even/odd GV registers pair. The conventional global register allocator does not take into account allocating overlapping registers of different sizes.

2. Some instructions operate only on registers. A pointer used in addressing mode must also be in a register. To ensure that it is always possible to load the temporary to a register, the allocator needs to reserve registers for this purpose [LH86].

3. For the floating-point registers, there are several restrictions on their combination of operands. The register usage is also constrained due to hardware restriction. For example, the target of the multiplication instruction must be P register or Q register in the floating-point register set, or memory. The floating point registers are divided to different classes according to their restrictions. As a result, each register class contains one or two registers. If the class contains only one register, the variable allocated by global register allocation must be spilled out frequently at each time other instruction uses other variables in the same class.

4. The floating-point instructions can operate on memory operands directly. Since most floating-point instructions are executed in one cycle even with memory operands, variables can sometimes reside in memory. Because memory operands depend on the operand combination of each instruction, the global register allocation can not take code selection into account.

While the global register allocation can make good use of a large registers set, it often fails for a small registers set. If only one registers is available and two variables need the register, the conflict between these register should then be resolved locally. The local register allocation can select "better" instructions to reference the variable efficiently on register or memory.

4.6.3 Local Register Allocation with Global Approximation

The global register allocation can be thought as an approximation of register allocation. The global register allocation is performed by the priority coloring algorithm before registers are allocated actually by the local register allocation. For each variables, the global register allocation returns the register assigned globally at each basic block.

Our algorithm allocates registers to variables locally in the basic block according to the approximation given by the global allocation phase, and then inserts spill codes to keep consistency between basic blocks. During the local allocation process, we maintain two tables to keep track of the status of registers and variables. The register status table keeps track of which variable is currently in each register. We consult this table whenever a new register is needed. We assume that at the beginning of the block the registers contain the value of variables allocated in the global allocation phase. For each variable, we maintain the variable status table that keeps track of the location where the current value of the variable can be found at run time. The location might be a register, a stack location or a memory cell. Before register allocation, the home location is allocated for all variables to make sure that the program is executable without using the optimizer. The status includes other information needed to update the home location of the variable.

The local allocation phase assigns a register for each variable, and computes the following information used to keep consistency between basic blocks in the subsequent phase:

- *available_expected*[n] — a set of variables whose values the local allocation phase expects to be available in the registers assigned in the global allocation phase at the beginning of block n.
- *available_gen*[n] — a set of variables which reach at the end of block n and whose values are available in the registers assigned in the global allocation phase at the end of the block.
- *available_through*[n] — a set of variables whose values are available in the register assigned in the global allocation phase at the beginning of block n, and which do not reach the end of the block or whose values still reside in the register at the end of the block.
- *modified_gen*[n] — a set of defined variables in block n which reach the end of the block and whose home locations are not updated with the modified value.

Actually, since these are subsets of variables, we can use a bit vector representation for these sets, and the amount of space used will not be prohibitive.

Algorithm 4.6 Local register allocation with global approximation.

Input. A basic block and its global register allocation.

Output. A revised basic block with registers allocated locally, and *available_expected[n], available_gen[n], available_through[n] and modified_gen[n].*

Method.

1. If a variable v is assigned to the register r in the global allocation phase at the beginning of the basic block, set the variable status of v and the register status of r to indicate that the value of v resides in register r. And then mark v "expected" in the variable.
2. For each instruction i, perform the following actions:
   (a) For each use of v in i, find the location of v. If v is found in a memory cell and the instruction can perform direct operation on the memory operand, replace the use of v with the memory operand. Otherwise, find the register r for v by calling $REFREG(v)$ to determine the register where the

```
function $REFREG(v)$
begin
  if $v$ is found in a register $r'$ then begin
    if $v$ is marked "expected" then
      add $v$ to $available\_expected[n]$;
  $r = r'$
  end else begin
    $v$ is found in a memory cell $m$;
    $r = GETREG(v)$;
    insert the load instruction "MOV$m, r$"
      before the current instruction;
  end
  return $r$
end
```

Figure 4.7: Function REFREG

```
function $GETREG(v)$
begin
  if $v$ reside a register $r$ then return $r$;
  if $v$ is assigned to $r'$ in the global allocation phase
    then $r = r'$
  else begin
    choose an appropriate register $r''$ for $v$;
    $r = r''$
  end
  if $r$ is used by other variable $v'$ then begin
    if $v'$ is marked "modified" then
      update the memory cell for $v'$; /* spill out */
    clear the status and mark for $r$ and $'v$
  end
  update the tables for $v$ and $r$ to associte $r$ with $v$;
  return $r$
end
```

Figure 4.8: Function GETREG

```
function $DEFREG(v)$
begin
  if $v$ is found in a register $r'$ then begin
    if $v$ is marked "expected" then
      unmark $v$ "expected";
    $r = r'$
  end else begin
    /* $v$ is found in the home location */
    $r = GETREG(v)$
  end
  mark $v$ "modified";
  return $r$
end
```

Figure 4.9: Function DEFREG

instruction should be performed, and then replace the use of v with r. The function $REFREG$ calls the function $GETREG$ which returns the register to hold the value of v for the instruction. The function $GETREG$ updates the tables to keep track the current status for variables and registers.

(b) If the use of v is the last use and its value resides in the register r, update the tables to indicate that r is not used any more. If the variable is marked "expected", put it in *available_through*$[n]$

(c) For a definition of v in i, find the location of v. If v is found in a memory cell and the instruction can perform direct operation on the memory operand, replace the definition of v with the memory operand. Otherwise, find the register r for v by calling $DEFREG(v)$ and then replace the definition of v with r. The function $DEFREF$ is slightly different from $REFREG$.

3. At the end of the basic block, if the value of a variable does not reside the register assigned in the global allocation phase and the variable is marked "modified", update the corresponding home location with the register content. If the variable is dead on exit, the register for the variable is removed in Step 2(b). Then, compute *available_gen*$[n]$ and *modified_gen*$[n]$. If the value of a variable resides in the register assigned in the global allocation phase, the variable is in *available_gen*$[n]$. The variable marked "modified" is in *modified_gen[n]*. If the variable is still marked "expected", put it in *available_through*$[n]$.

If a variable is not allocated to registers in the global allocation phase, the function $GETREG$ must choose an appropriate register for the variable. We may choose the least frequently used register or the least recently used register as a candidate [Fre74]. The function $GETREG$ returns the register assigned in the global allocation phase if any. Although we may not follow the approximation by the global register allocation, the globally optimal solutions sometimes belong to the local optimal solutions.

4.6.4 Inserting Spill Code

Once the local register allocation is done for each basic block, we insert load/store codes to keep consistency between basic blocks. A *live range* of a variable is an isolated and contiguous group of nodes in the flow graph in which the variable is defined or referenced. We define an *available range* of the variable as a subset of the live range in which the value of variable resides in the register. The following sets of variables are computed to find the available range for each node:

- *available_in*[n] — a set of variables whose value is available in the register assigned in the global allocation phase at the beginning of block n.
- *available_out*[n] — a set of variables whose value is available in the register assigned in the global allocation phase at the end of block n.

These sets can be computed by the following data flow equation, which is similar to that for available expression computation:

$$\begin{aligned} &available\_out[n] \\ &\quad = (available\_in[n] \cap available\_through[n] \cap live\_out[n]) \\ &\qquad \cup available\_gen[n] \end{aligned}$$

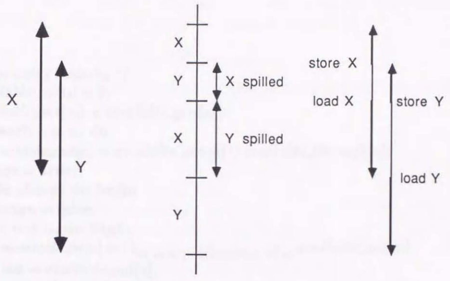
$$available\_in[n] = \bigcap_{m \text{ a predecessor of } n} available\_out[m]$$
$$\text{for } n \text{ is not an entry node}$$

$$available\_in[n] = \emptyset$$
$$\text{for } n \text{ is an entry node}$$

The live variable information, *live_out*[n], is used to remove the variable in *available_through[n]* which does not reach the end of the block.

If the value of the variable is changed in the intervening code where it resides in register, the home location of the variable must be updated with the register contents at the end of the code segments unless it is dead on exit. The live range of the variable may be split into several available ranges. Figure 4.10(a) shows a region of code in which variable X and Y are to be allocated in the register. Figure 4.10(b) and (c) show possible allocation result and its available ranges with spill codes.

If the local allocation expects the variable to be loaded at the entry of the available range, the load instruction is needed to load the value into the register. If the modified variable in register is live at the exit of the available range, the value must be spilled into the home location. If the available range of the variable covers the whole live range and the variable is used locally in the procedure, no spill code is needed because the variable is dead at every exit.



(a) live range (b) possible allocation (c) available range

Figure 4.10: Example of register allocation

The local allocation phase can compute local information only in the basic block. If the value in the register is changed in some block, its value may be spill out at the exit of the available range to which the node belongs. The following set is computed to know where the value in the register should be store into the home location:

- *modified_out*[n] — a set of variables whose values are available in the registers assigned in the global allocation phase at end of block n and its values are different from the values in the home locations.

The set *modified_out*[n] is computed by propagating *modified_gen*[n] along with the path of the available range.

Algorithm 4.7. Available values in registers

Input. A flow graph with *available_gen*[n], *available_through*[n] and *modified_gen*[n] computed by the local allocation phase for each node n, and live range information.

Output. *available_in*[n] and *modified_out*[n] for each node n.

Method. Execute the program in Figure 4.11 and Figure 4.12.

The next algorithm inserts spill codes to keep consistency between basic blocks. By the set *available_expected*[n], it checks whether the value in the register is actually needed or not. If the global allocation phase allocated the variable in different registers for two blocks, the value on the register must be reloaded to the other register.

Algorithm 4.8 Inserting spill codes at basic block bounary.

```
/* for entry node $n_0$ */
$available\_in[n_0] = \emptyset$;
$available\_out[n_0] = available\_gen[n_0]$
for each $n \neq n_0$ do
  $available\_out[n] = available\_gen[n] \cup available\_through[n]$;
$change$ = true;
while $change$ do begin
  $change$ = false;
  for $n \neq n_0$ do begin
    $available\_in[n] = \bigcap_{m \text{ is a predecessor of } n} available\_out[m]$;
    $out = available\_out[n]$;
    $available\_out[n] =$
      $(available\_in[n] \cap available\_through[n] \cap live\_out[n])$
        $\cup available\_gen[n]$;
    if $available\_out[n] \neq out$ then $change$ =true
  end
end
```

Figure 4.11: Available registers computation

```
for each $n$ do
  $modified\_out[n] = modified\_gen[n]$;
$change$ = true;
while $change$ do begin
  $change$ = false;
  for $n$ do begin
    $in = \bigcup_{m \text{ is a predecessor of } n} modified\_out[m]$;
    $out = modified\_out[n]$;
    $modified\_out[n] =$
      $(in \cap available\_in[n] \cap available\_through[n])$
        $\cup modified\_gen[n]$;
    if $modified\_out[n] \neq out$ then $change$ =true
  end
end
```

Figure 4.12: Modified values propagation

Input. A flow graph with $available\_in[n]$, and $modified\_out[n]$ for each node n. In each block, registers was allocated locally by Algorithm 4.6.

Output. A flow graph after register allocation.

Method. For each block of node n and each variable n, execute the program in Figure 4.13.

```
save = false;
load = false;
if v is in available_in[n] then begin
  if v is in available_expected[n] or
        v is in available_through[n] then begin
     find register r for v at the beginning of n;
     for each m a predecessor of n do begin
       find register r' for v at the end of m;
       if r ≠ r' then begin
         save = true;
         load = true;
       end
     end
  end
  if v is not in available_through[n] then save = true
end else begin
  save = true;
  if v is in available_expected[n] then load = true
end
if save then
  for each m a predecessor of n do
    if v is in modified_out[m] then begin
      insert the store instruction to
         update home location of v;
      remove v from modified_out[m]
    end
if load then
  insert the load instruction to load the value into the register
end
```

Figure 4.13: Inserting spill code

Chapter 5

Loop optimization with BL Addressing

In this chapter, we describe the code generation for the BL addressing, which FLATS2 provides to reduce the execution time of array computation in numerical workload by integrating memory addressing and range checking. To make use of the BL addressing, we need an extensive induction variable elimination to access an array by incrementing the pointer by a constant. Although it may generate too many temporaries for induction variables, our induction variable elimination finds optimal induction variable elimination by taking into account the cost for spilling these temporaries to memory.

5.1 Addressing mode for Numerical Computation

5.1.1 The Nature of Numerical Computations

In typical numerical computations, the absolutely predominant data structure is the array. The critical loops in numerical computation perform floating-point operations on the elements of arrays. In the majority of the cases, the array elements are accessed in regular sequence. There are a few "working locations" in the array, and their addresses change as arithmetic progressions. The step is quite often equal to one element size, the column size or some other constants. The arithmetic operation performed on integer scalar variables used as loop-counters and array-indexes is simple and corresponds to the above "regular sequence" of array accesses: increment by a constant, compare and branch.

Address computations for multi-dimensional arrays require integer multiplication. Most of the times, the optimizing compiler can replace those integer counter/indexes by actual memory pointers by induction variable elimination. For example, in the code shown below,

```
    DO 10 I = 1,N
 10 S = S + A(I)
```

an optimizing compiler will generate the code shown in Figure 5.1

```
(1)   B1:  r[p] = A - 8                /* base address of A */
(2)        r[q] = A - 8 + 8*N          /* limit address of A */
(3)   B2:  r[S] = r[S] + m[r[p]]
(4)        r[p] = r[p] + 8
(5)        if r[p] <= r[q] then goto B2
```

Figure 5.1: Optimized code of array computation

The numerical computations are usually floating-point operations. Counter/address calculations and floating-point operations are often executed in parallel, especially when the programs perform a certain computation on all elements of a vector or an array.

5.1.2 Autoincrement Addressing Mode

If an induction variable is a pointer, we call the induction variable an *induction pointer*. For the induction pointer, *Autoincrement* addressing modes can sometimes be used to increment the pointer when the pointer is allocated to a register within the loop. These addressing modes dereference a pointer and automatically advance it to the next element. For example, instruction (3) and (4) in Figure 5.1 can be executed in one instruction. In this case, the execution time for the increment is absorbed into the addressing operations.

```
(1)   B1:  r[p] = A - 8                /* base address of A */
(2)        r[q] = A - 8 + 8*N          /* limit address of A */
(3')  B2:  r[S] = r[S] + m[r[p]];
           r[p] = r[p] + 8             /* autoincrement */
(5)        if r[p] <= r[q] then goto B2
```

Figure 5.2: Optimized code with autoincrement

FLATS2 has several autoincrement addressing modes. These addressing modes may reduce the number of cycles for array computations. Some conventional machines such as VAX and 68000 also have autoincrement addressing modes.

5.1.3 BL addressing

In the BL addressing of FLATS2, the effective address is checked by comparing the address with the base address and the limit address (*BL pair*) whenever the operand in memory is accessed. It enables memory access and range checking to be performed in parallel. If the effective address is in the range between given base and limit addresses, and the memory access completes successfully, then a branch is taken to the specified target.

Otherwise, the next instruction is executed. The branch of the BL addressing can be executed without delay in the FLATS2 pipeline due to the cyclic pipeline architecture.

In FLATS2, every word has an address tag. The address tag of the register modified by post-increment addressing mode is set to zero when the modified address goes outside the given range. The value in the register can never be used as an address any more.

A base address of BL pair may also specify the effective address for memory access. In this case, the post-increment addressing mode modifies the base address of BL pair. Range checking is done against the previous BL pair. If the modified base address is outside of the range of the previous BL pair, the address tag of the base address is cleared, so that the BL pair cannot be used for memory addressing.

```
(1)    B1:   r[p].base = A - 8          /* base address of A */
(2)          r[p].limit = A - 8 + 8*N  /* limit address of A */
(3'') B2:   if address tag of r[p].base == 0 then goto next;
             r[S] = r[S] + m[r[p].base];
             t = r[p].base + 8;
             if not BL(t,r[p]) then address tag of r[p].base = 0
             r[p].base = t
             goto B2                    /* BL addressing */
```

Figure 5.3: Optimized code with BL addressing

For a loop of array computation, the range checking of the BL addressing can terminate the loop if the first instruction in the loop uses the post-increment addressing mode to access the element of an array in regular sequence. The base address of the BL pair specifies each element of the array in each iteration by using post-increment addressing mode. When the last element is accessed, the address tag of the base address is cleared. Next time the instruction is executed, range checking of the BL addressing finds the base address cleared to terminate the loop.

In Figure 5.1, the addresses of the first element and the last element of an array `A` form the BL pair for `A` in a pair of registers. By using the BL addressing mode with post-increment for memory access in instruction `(3)`, instructions `(3)`,`(4)` and `(5)` are executed in one instruction. As a result of this transformation, the loop becomes one-instruction loop, as shown in Figure 5.3. The function `BL` in the Figure returns true if the first operands is within the range of the second operand.

As mentioned before, array computation is often dominant in numerical computation. The BL addressing integrates memory addressing and range checking to reduce the overhead of iterations for many cases. The transformation by the compiler is described in later section.

5.2 Optimal Induction Variable Elimination

In this section, we describe an induction variable elimination algorithm that finds near-optimal induction variables with given addressing modes and registers. The algorithm can be parameterized to cater to different memory addressing characteristics and the number of registers among machines.

5.2.1 Induction Variable Elimination

Induction variable elimination is an important and effective loop optimization, especially for scientific application where loop is the most frequently used control structure. This optimization may replace some multiplication operations within a loop by simple additions. A multiplication is automatically generated as a part of the array accessing mechanism. The optimization known as *reduction of operator strength* or *strength reduction* is to eliminate such multiplications whenever possible. Strength reduction replaces an expensive operation by a cheaper one, such as a multiplication by an addition.

A *loop induction variable* is a variable whose value is changed within the loop by a constant amount. A *region constant* is a variable whose value is not changed within the loop. A constant in the program is also included in this class. An obvious example of an induction variable is the DO-loop variable. Some other induction variable whose value is a linear function of the DO-loop variable is the actual pointer used to access the array. Often, the only use of a DO-loop variable is in the test of loop termination. We can then get rid of the DO-loop variable by replacing its test by one on the pointer. A DO-loop variable is a *basic induction variable*, which is a variable whose only assignment within the loop is an increment of the variable. An *induction expression* is any expression which is a linear function of the other induction variable. The value of an induction variable is computed by the induction expression.

Induction variable elimination converts a nontrivial subscript expression (typically involving one or more multiplications) inside DO-loop as a temporary containing the exact address of the array element referenced on each iteration and a simple increment of this value at the end of the loop. It increases the chance to make use of array access mechanism provide by hardware such as autoincrement addressing and BL addressing.

The standard induction variable elimination algorithm does not take into account the cost and saving involved in the optimization. By cost, we refer to the cost for induction variables being incremented by constants in temporaries. By saving, we refer to the gain in execution speed due to the induction variable being computed by a simple increment instead of the induction expression. The standard algorithm is based on the assumption that keeping the value of induction variables in temporaries is cheaper than the computation of any induction expression. If a machine has an "index addressing mode", the induction expression which computes the address by adding the offset to the base of the array can be absorbed into memory addressing [1]. For autoincrement addressing mode, the increment of an induction variable may be combined into memory addressing when the induction variable is available in registers.

The transformation of induction variable elimination sometimes generates too many temporaries needed for a collection of subscript expressions, so that some of these temporaries must be spilled out into memory. If the induction expression is simple enough to compute the value as needed, the spill cost may be larger than its computation cost.

[1]In CISC machines, it is not clear that index addressing is faster than addressing with the pointer computed by the addition.

Therefore, the number of available registers also affects on the optimal induction variable elimination. Minimizing the number of temporaries need for a related collection of subscript expressions is discussed in Sites [Sit79]. It does not take into account the effect of spilling induction variables and the computation cost of induction expressions.

5.2.2 Cost and Saving Estimates

An induction variable is either a basic induction variable or a variable J for which there is an induction variable such that each time the variable is assigned in a loop L, J's value is computed by an linear function of the other induction variable. We assume that the linear function is performed by an instruction. Introducing a temporary which holds the value of the induction expression, the linear function can be replaced by a simple increment. For example, if the linear function is a multiplication, it can be done by a cheaper addition.

If the linear function is an addition, an increment, however, may not be faster than the addition on some machines. When the value of the induction expression is used only for memory addressing, some addressing mechanisms can execute the simple addition efficiently as a part of memory addressing. An increment on the temporaries for the induction variable can also done in memory addressing such as autoincrement addressing if the temporary resides in register. If the temporary can not reside in a register during loop, its value must be saved in memory at each iteration. This extra move operations between registers and memory increase the execution time for the temporary. The saving by induction variable elimination depends on how the value of the induction expression is used and where the temporary for the induction variable resides. Thus, we define the following parameters, which vary among target machines:

$COMPSAVE$ — The amount of execution time for the instruction computing an induction expression, which may be the execution time of memory addressing if the value of the induction expression is used only once for memory addressing and is computed as a part of the addressing mode.

$INCRCOST$ — The cost of an increment of the temporary for an induction expression, when the temporary resides in register. If the value of the expression expression is used only for memory addressing, a special memory addressing mechanism such as autoincrement addressing gives the cost.

$SPILLCOST$ — The cost of spilling the temporary of an induction variable to memory, which holds the value at each iteration.

The standard induction variable elimination works on each induction variable locally. For each induction variable, say i, the saving that can be achieved by the transformation can be estimated by:

$$LOCALSAVE_i = COMPSAVE_i - INCRCOST_i - SPILLCOST * w_i$$

where w_i is either 0 or 1. The parameter, w_i depends on whether the temporary for i resides in register within the loop. If $LOCALSAVE_i$ is positive, the transformation on the induction variable can reduce the execution time. The strength reduction always replaces a multiplication by an increment, because it is faster on most machines even if spill is needed.

When the total cost for a related collection of induction variables is considered, the transformation may reduce the cost even if the local saving for an induction variable is negative. Once the transformation has been performed, some induction variables will no longer be needed. Even for their uses within a conditional branch instruction, the *linear function test replacement* can eliminate all code involving such an induction variable if the test can be replaced by the test of a generated temporary. Thus, the saving for an induction variable j which is a linear function of an induction variable i, can be estimated by:

$$SAVE_j = LOCALSAVE_j + (INCRCOST_i + SPILLCOST * w_i) \times u_{ij}$$

where u_{ij} is 1 if j can be removed after the transformation, otherwise 0. Even if the linear function is an addition with the same cost as an increment, the induction variable is eligible for the transformation when it gets rid of other induction variables. The parameters u_{ij} are determined by the relation among induction variables.

The total saving for a set of induction variables is estimated by:

$$TOTALSAVE = \sum_{i \text{ an induction variable}} SAVE_i$$

The TOTALCOST depends on each of w_i and u_{ij}. The value w_i is determined by the number of registers available for the temporaries generated for induction variables.

Let $MAXSAVE(n)$ to be the maximum saving by induction variable elimination with n temporaries for induction variables. Because a simple increment is one of the fastest operations on most machines, we assume that $COMPSAVE_i$ is greater or equal than $INCRCOST_i$. The quantity $MAXSAVE(n)$ represents the maximum possible saving, assuming an infinite number of registers are available for the transformation:

$$MAXSAVE(n) = \sum_{i \text{ an induction variable}} (COMPSAVE_i - INCRCOST_i * U_i)$$

where U_i is 1 if the value of i is needed after the transformation, otherwise 0. The number of induction variables which have the value U_i of 1 is n.

Actually, some of temporaries must however be spilled when the number of temporaries are greater than one of actual registers. Thus, the actual saving after the transformation is gives as:

$$TOTALSAVE = MAXSAVE(n) - SPILLCOST \times (n - r)$$

where r is the number of available registers. To maximize the total saving, we must minimize the spill cost as well as maximing the saving. The spill cost may decrease the total saving even if the $COMPCOST$ is greater than the $INCRCOST$ for a particular induction variable. We prevent "over reduction" of induction variable elimination by maximizing the quantity $TOTALSAVE$.

5.2.3 Induction Variable DAG

In this subsection, we describe the data structure for our algorithm. One of the first subtasks of the induction variable elimination is to find induction variables and region constants. We assume that the previous optimization phase such as loop invariant code motion has determined the set of region constants. A basic induction variable of a loop L is a variable whose only assignments within L are of the form $I = I \pm C$, where C is a region constant.

A labeled DAG, called an *induction variable DAG* is used to represent a related set of induction variables. Nodes are labeled by the associated induction variables. The label on a root node is a basic induction variable. The induction expressions are also specified for interior nodes. The operands of an interior node are given by the label in the parent.

Algorithm 5.1 Constructing an induction variable DAG.

Input. A loop L in SSA form with the set of region constants.

Output. An induction variable DAG.

Method.

1. Find all basic induction variables by scanning the statement of L. We use the set of region constants here. The root node labeled with the variable is associated with each basic induction variable. In SSA form, a basic induction variable is represented with three variables in Figure 5.4. Both `v1` and `v2` are basic induction variables for the node. Repeat the next step until no induction variable is found.
2. Search for variable k with a statement within L having one of the following forms:

   $$k = j \pm b, k = b \pm j, k = j \pm i, k = j * c, k = c * j, k = j/b$$

   where b is a region constant, and i and j are induction variables, basic or otherwise. Create an interior nodes for k whose parent is associated with j. For statement $k = i \pm j$, the node has two parents for i and j. Note that due to SSA form there is no other assignments for k in the program.

Figure 5.5 shows the example of an induction variable DAG. An induction variable DAG often becomes a tree. If a node for j has the node for a basic induction variable i as its ancestor, we say that j belongs to the *family* of i. The basic induction variable i belongs to its own family.

Once the induction variable DAG is constructed, we first assign the cost and saving for each node, given by the machine. We take its usage into account to determine the cost; if the value of an induction variable is used once for memory addressing, and its computation can be done as a part of addressing, then its cost could be less than that of computational cost by an instruction. For example, the cost of increment for such variables is zero in FLATS2 because the execution time for increment of a pointer is completely absorbed into autoincrement addressing. The addition of a base pointer with an offset can also be performed by index addressing.

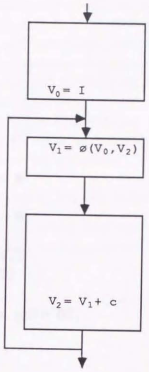


Figure 5.4: Basic induction variable in SSA form

5.2.4 Optimal Induction Variable Elimination

The problem here is to optimize the total saving for induction variables, given the number of register for temporaries of induction variables. The optimization problem can be shown to be NP-complete even when an induction variable DAG is a binary tree; this is done using *partially ordered knapsack problem* [GJ79]. To find the transformation which maximizes the quantity $TOTALSAVE$, we need to compute $MAXSAVE(n)$ for each n. Suppose that all increment cost is the same, the quantity $MAXSAVE(n)$ is computed by:

$$MAXSAVE(n) = \sum COMPSAVE_i - INCRCOST \times n$$

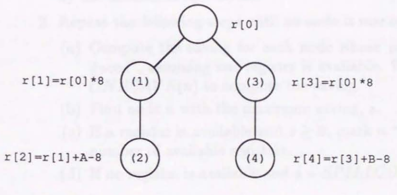
The partially ordered knapsack problem is as follows:

> *Given:* Finite set U, partial order $<$ on U, for each $u \in U$, value $v(u)$, positive integer k
>
> *Problem:* Find a subset of U, U' such that
>
> 1. the number of $u \in U'$ is k,
> 2. if $u \in U'$ and $u' < u$, then $u' \in U'$,
> 3. $\sum_{u \in U'} v(u)$ is maximal.

Because the computation of children nodes uses the result of its parent node, we can define the partial order on the nodes with the tree in the transformation. If each induction

```
(1) B2:   r[1] = r[0]*8
(2)       r[2] = r[1] + A - 8
(3)       r[3] = r[0]*8
(4)       r[4] = r[3] + B - 8
(5)       r[T] = m[r[2]]
(6)       r[T] = r[T]*m[r[4]]
(7)       r[S] = r[S] + r[T]
(8)       r[0] = r[0] + 1;
(9)       if r[0] <= N then goto B2;
```

(a) code



(b) induction variable DAG

Figure 5.5: Induction variable DAG

variable in the interior nodes is used only in operands of the expressions of its children nodes in the binary tree, one temporaries is need to remove the induction variable by the transformation. We can perform the transformation on n variables with n temporaries. The $COMPSAVE$ for each node is associated with the value in the problem. Thus, the choice of the nodes to be transformed for maximizing the total value with n temporaries is equivalent to the partial knapsack problem.

Since this optimization process will be a part of each compilation and optimal induction variable elimination is very expensive, we will concentrate on "good" solution rather than on optimal one. The basic heuristics are as follows:

Algorithm 4.2 Optimal induction variable elimination.

Input. An induction variable DAG for a loop with cost and saving for each node, and the number of registers available for temporaries to hold the values of the induction variables.

Output. A set of induction variables to be transformed by induction variable elimination.

Method.

1. Initially, root nodes are marked "reduced" because basic induction variables are to be assigned to registers. The number of available registers is decreased by the number of root nodes.
2. Repeat the following steps until no node is marked "reduced".
   (a) Compute the saving for each node whose parent nodes are marked "reduced", assuming one register is available. We use the recursive function $ONESAVE(n)$ to compute the saving.
   (b) Find node n with the maximum saving, s.
   (c) If a register is available and $s \geq 0$, mark n "reduced" and decrement the number of available registers.
   (d) If no register is available and $s - SPILLCOST \geq 0$, mark n "reduced".

Using the COMPCOST of each node, we can find the node with the minimum computation cost for test replacement. Since the loop optimization is done from the inner loop to the outer, the number of available registers is decremented by the register used in the inner loops.

5.2.5 Inserting Increment for Transformed Induction Variables

The increment for a transformed induction variable is placed before the tail of the loop because the tail is executed at every iteration. In the landing pad, we place the code to set the initial value to the temporary and compute the step incremented in each iteration. The transformation for the initial value and step of the induction variable is described in Aho [ASU86].

```
function $ONESAVE(n)$
begin
  $saving = 0$;
  for each child $m$ of $n$ do begin /* count for children */
    $s = ONE_SAVE(m)$;
    /* don't count negative saving */
    if $s >= 0$ then $saving = saving + s - SPILLCOST$;
  end
  /* count for n */
  if $n$ is needed after reduction then
    /* increment cost is needed for n */
    $saving = saving - INCRCOST_n$
  else
    /* one of children can reside in a register */
    $saving = saving + SPILLCOST$
  /* minimum saving is $COMPSAVE(n) - INCRCOST(n)$ */
  return $max(COMPSAVE(n) + saving, COMPSAVE(n) - INCRCOST(n))$
end
```

Figure 5.6: Function ONESAVE

If an induction variable is used only for memory addressing, the increment instruction for the variable may be combined with the memory addressing of other instructions in the same basic block in the instruction reconstruction phase. In our compiler, if the loop is the most inner loop and exit only from the tail, we place the increment in the same block of its last use as possible to increase the chance of combining with memory addressing. Such a loop is called a "hammock" loop. Only one increment must be placed after its use in every path form the header to the tail in the loop. Since more than one increments may be generated, code size may become larger but the execution time is not increased. If more than two increments are placed in the flow graph, the ϕ function is generated to keep the code in SSA form.

Algorithm 5.3 Finding positions of increment for induction variables.

Input. A hammock loop L in SSA form and a variable to be incremented in the loop.

Output. A set of nodes in the flow graph to place the increments and the ϕ functions.

Method. We assume the initial value is computed in v_0 in the landing pad. Variable v_1 represents the value of the induction variable at the beginning of the loop header.

1. Compute whether the variable may be incremented in each node. Perform the following step for node n in the loop in reverse topsort order:

   (a) If the variable is used in the node, mark the node "used".
   (b) If either of the successors is marked "used", mark the node "used".

2. Perform the following step for node n in the loop in topsort order:
   (a) If the node is a loop header, v_1 reached the beginning of the block.
   (b) If v_i reaches the end of the block of all predecessors, v_i reaches the beginning of the block.
   (c) If v_i reaches the end of the block of one predecessor and v_j reaches the end of the block of another predecessor, generate a new variable v_k to place $v_k = \phi(v_i, v_j)$ at the beginning of the block. If v_i is v_1, generate a new variable v_i to place the increment $v_i = v_1 + C$ at the end of the block of the predecessor. Variable v_k reaches the beginning of the block.
   (d) If the node is marked "used", the variable that reach the beginning of the block must be v_1. If the successor is marked "used", generate a new variable v_n and place an increment $v_n = v_1 + C$ at the end of the block. The variable v_n reaches the end of the block, otherwise the variable at the beginning of the block reaches the end of the block.

3. A variable, say v_l, reaches the end of the tail. Place $v_1 = \phi(v_0, v_l)$ at the beginning of the header.

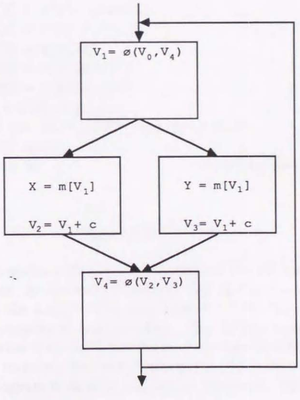


Figure 5.7: Inserting increments

Figure 5.7 shows the example of inserting increments for an induction variable. Although the algorithm may generate the redundant increment for some flow graphs, the redundancy elimination phase eliminates unused increments after the loop optimization. For most cases, the register allocation phase allocates these variables for an induction variable in the same register.

5.3 BL Code Generation

The BL addressing performs range checking during memory access. Range checking is also implemented as an individual instruction. We call *BL code* the code using range checking. In this section, we describe the code generation for instructions involving range checking.

5.3.1 DO Loop with Range Checking Instruction

We can make use of the range checking instruction to terminate a DO loop. The front-end generate the BL code optionally. Figure 5.8 shows an example of BL code.

```
(1)   B1:  r[1] = MKBL(1,N)             /* make BL pair for counter */
(2)   B2:  r[2] = r[1].base*8
(3)        r[2] = r[2] + A - 8
(4)        r[3] = r[1].base*8
(5)        r[3] = r[3] + B - 8
(6)        r[T] = m[r[2]]
(7)        r[T] = r[T]*m[r[3]]
(8)        r[S] = r[S] + r[T]
(9)        t = r[1].base + 1;
           if not BL(t,r[1]) then goto next;
           r[1].base = t;
           goto B2                 /* ACBL instruction */
```

Figure 5.8: BL code

The operation MKBL makes a BL pair for the range of the DO-loop variable in the node entering the loop header. At the end of the loop, *add and compare, branch on BL(*ACBL*)* instruction terminates the loop [2]. The base register of the BL pair holds the current value of the DO-loop variable at each iteration. The ACBL instruction increments the base register and branches if the modified value of the base is within the previous range, otherwise the control transfers the next instruction. Note that fetching the value for both increment and compare is done in parallel by hardware. The branch is taken after

[2]FORTRAN does not allow DO-loop variables to be assigned within the loop.

all operations of the instruction. As the BL code modifies the base register at every iteration, the range becomes minimal at the last iteration.

It should be noted that the BL code can be applied efficiently even if the step for the DO loop variable is specified by a variable at run-time. Without range checking, the code for the variable step must switch the compare instruction for the DO loop variable according to the sign of the step value, resulting in inefficient complicated codes. The BL code allows the same DO loop code to be generated for any step.

5.3.2 Induction Variable Elimination for BL code

Induction variable elimination for BL code is slightly different from conventional one. Like other basic induction variables, the DO-loop variable involving BL code is recognized as a basic induction variable. The first operand of MKBL operation for the DO loop variable is used to set up the initial values of induction variables in its family. For example, the first operand of the MKBL operation is used as an initial value to perform the transformation of induction variable elimination. This enables constant folding for calculation of the initial value of other induction variables if the initial value of the basic induction variable is a constant.

The test replacement on BL code generates the BL pair for the induction variable which is used to terminate the loop instead of the DO-loop variable. The limit of the BL pair is calculated with the second operand of MKBL operation of the DO-loop variable. The MKBL operation to form the BL pair placed in the landing pad. In this case, the original MKBL is removed by redundancy elimination because it is used no longer after induction variable elimination. Figure 5.9 shows the code after induction variable elimination.

```
(1')  B1:   r[2'] = MKBL(A-8,A+N*8-N)        /* make BL pair for A */
(1'')       r[3'] = MKBL(B-8,*)              /* make BL par for B */
(6)   B2:   r[T] = m[r[2'].base]
(7)         r[T] = r[T]*m[r[3'].base]
(8)         r[S] = r[S] + r[T]
(9')        r[3'].base = r[3'].base + 8
(9)         t = r[2'].base + 8;
            if not BL(t,r[2']) then goto next;
            r[2'].base = t;
            goto B2                          /* ACBL instruction */
```

Figure 5.9: BL code after induction variable elimination

If the BL pair does not test the termination of the loop, the BL pairs may omit the calculation of its limit to reduce the cost for setup before the loop; the maximum address can be used for the limit if the value of the induction variable increases at each iteration, otherwise the minimum address can be used for the decreasing induction variable. If the step for the induction variable is variable, we must calculate its proper limit for the BL pair. In Figure 5.9, the limit of the BL pair for B is the maximum address.

5.3.3 Code Generation for BL addressing

For a loop with BL code, we generate the BL addressing mode for memory operand addressed by an induction pointer. Although BL addressing modes require more registers for BL pairs than conventional addressing modes, our induction variable elimination algorithm optimizes the cost involving the limited number of registers.

As mentioned in chapter 3, the combination of addressing modes in FLATS2 is restricted because two addressing modes share register fields in the fixed-length instruction format. The front-end generates code using the default addressing mode, which calculate the effective address by adding the pointer with an offset. The BL pair for the default addressing mode is the default BL pair (the entire program space). When both memory operands access arrays, we can use the addressing modes with different BL pairs. In this combination of addressing modes, we can increment both base addresses in one instruction to optimize array computations. If either addressing mode is for an array and that is not used for range checking, the addressing mode may be converted to the default addressing mode using the base of the BL pair as a pointer.

The transformation for the BL addressing is performed as follows:

1. Make a BL pair for the induction pointer in the loop which is used only for memory addressing in the induction variable elimination phase. The base address of the BL pair is used to access the array element. The limit may be the maximum/minimum address as mentioned in the previous subsection. And convert these default addressing modes of the induction pointer to the BL addressing modes using the BL pair as base.

2. If the first instruction in the loop accesses memory with the above BL addressing mode and the BL pair can be used for the test of termination, calculate the limit of the BL pair from the limit of the original DO loop variable to replace the test with it. Otherwise, the induction variable with cheaper computation is chosen.

3. Combine the increments of induction pointers with memory addressing modes in the code reconstruction phase.

4. If the first instruction in the loop accesses the memory with the post-modify BL addressing mode for the array and the test uses the BL pair of the BL addressing mode at the first instruction to terminate the loop, move the instruction before the loop and combine the instruction with the test.

For array access, a base address of the BL pair specifies the effective address for memory access, and the base register is incremented by the post-increment BL addressing mode at each iteration. The effective address is checked by comparing the address with the given BL pair in parallel with memory access. The branch is taken to a specified target for the next iteration if the memory access completes successfully. The address tag of the modified base register is set to zero when the modified address is outside the given range. Then, the value in the register can never be used as an address any more at the last iteration. Next time the BL pair is used to access the memory, the BL addressing terminates the loop to detect no address tag of the BL pair.

In the code reconstruction phase, the first instruction is moved to both the end of the landing pad and the end of the loop as shown Figure 5.10. For the DO loop, the test before the loop ensures that the loop is executed at least once, so that the memory access of the instruction in the landing pad complete successfully.

```
(1')  B1:  r[2'] = MKBL(A-8,A+N*8-N)         /* make BL pair for A */
(1'')      r[3'] = MKBL(B-8,*)               /* make BL par for B */
(6')       r[T] = m[r[2'].base];
           t = r[2'].base + 8;
           if not BL(t,r[2']) then address tag of r[2'].base = 0
           r[2'].base = t                    /* BL addressing */
(7')  B2:  r[T] = r[T]*m[r[3'].base];
           r[3'].base = r[3'].base + 8       /* autoincrement */
(8)        r[S] = r[S] + r[T]
(6'')      if address tag of r[2].base == 0 goto next;
           r[T] = m[r[2'].base];
           t = r[2'].base + 8;
           if not BL(t,r[2'].base) then address tag of r[2'].base = 0
           r[2'].base = t
           goto B2                           /* BL addressing */
```

Figure 5.10: BL code after code reconstruction

The BL addressing reduces the overhead to control the loop. Although the execution time for the test of loop termination is relatively smaller than the time for the loop body, it is effective for basic array operations such as inner product because the body consists of a few instructions.

Chapter 6

Experiments on FLATS2

In this chapter, we report several experiments on using the FLATS2 FORTRAN compiler to evaluate FLATS2.

A cyclic pipeline architecture exploits the parallelism with multiple instruction streams which share the same pipeline in time. In chapter 2, we have demonstrated the performance improvement of a cyclic pipeline machine with the highly pipeline model. [1].

In FLATS2, the degree of pipeline in execution stages is limited to a relatively small number. No pipelined memory access is allowed in FLATS2. The performance of an individual stream is improved with microarchitectural parallelism as well as overlapped execution in an individual instruction stream.

The question addressed by this chapter is: Is a cyclic pipeline architecture still effective given a set of scientific workloads and a set of technology constrains of FLATS2?

In Section 1, we report the result of experiments on the real FLATS2 machine using some benchmark programs. In Section 2, we consider the architectural alternatives based on the FLATS2 pipeline to evaluate the FLATS2 architecture.

6.1 The performance of FLATS2

In this section, we report the performance measured on FLATS2. We chose the Livermore loops [McM84] and Linpack benchmarks [Don79] as our benchmarks. The cycle time of FLATS2 is 65 ns. The instruction is issued at every 2 cycles.

6.1.1 Speedup with BL addressing

We have examined the speedup using the BL addressing in an individual instruction stream. Table 6.1 shows the speed by BL addressing for each benchmark.

The Livermore benchmark is a collection of kernel loops which frequently appear in scientific workload. For the Livermore benchmark, the average is shown in Table 6.1. Figure 6.1 shows the speedup for each loop in various spans of iterations.

The BL addressing can be applied only when the first instruction of the loop accesses the array in regular sequence; loop 15,16 and 17 have no such a loop. The BL addressing

[1]The estimated performance is optimistic due to the lack of cache models in the simulation.

Program	-O (MFLOPS)	-OB (MFLOPS)	Speedup
Linpack	1.96	2.66	1.36
Livermore loops (average)	1.47	1.74	1.18

Key: -O — optimized without BL addressing.
-OB — optimized with BL addressing.

Table 6.1: Speedup with BL addressing

can reduce a few instructions, so the speedups become large in loop 1,3,5,10,11 and 12 where the loop bodies are small. Even if the BL addressing can be applied, the speedup is relatively small for the large body of the loop as Loop 13. In code generation, the compiler may place the first instruction which does not access the memory in regular sequence. For example, in loop 21 the compiler generates the loop invariant memory access as the first instruction. The performance would be improved if the source program is modified explicitly to make use of the BL addressing,

For the BL addressing, we need extra registers and computations to make BL pairs. Due to this overhead, the performance may be degraded in smaller span in some loops. Since available operand combinations of BL addressing modes are different from that of the conventional addressing mode in FLATS2, the optimization of BL addressing affects the code selection.

6.1.2 Speedup by two instruction streams

We have measured the speedup by two instruction streams for the Linpack benchmark to parallelize the program. The speedup of the benchmark is shown in Table 6.2. The BL addressing also increases the performance in the parallel version.

Compile option	One instruction stream (MFLOPS)	Two instruction streams (MFLOPS)	Speedup
-O	1.96	3.70	1.89
-OB	2.66	5.00	1.87

Key: -O — optimized without BL addressing.
-OB — optimized with BL addressing.

Table 6.2: Speedup of Linpack by Two instruction streams

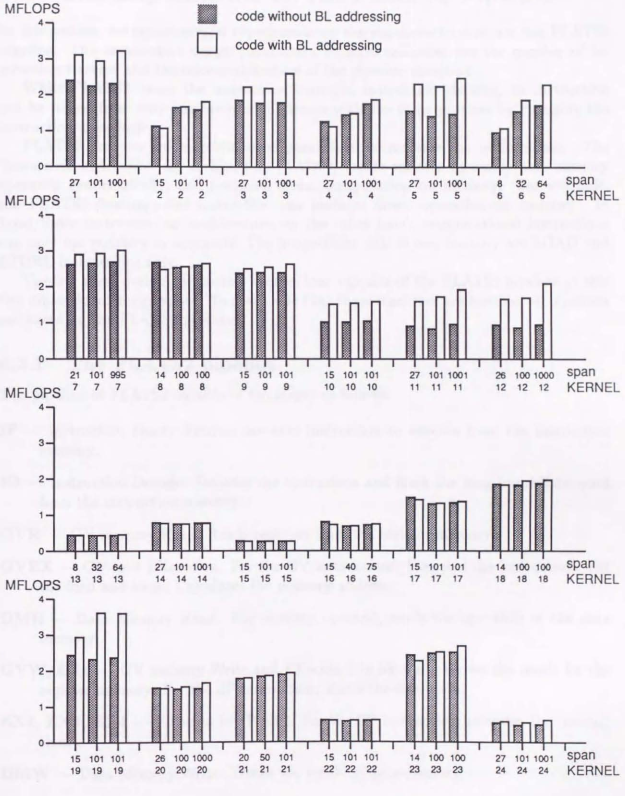


Figure 6.1: Performance of Livermore loops with BL addressing

6.2 The Experiments on the FLATS2 Pipeline

In this section, we report several experiments on the simulator to evaluate the FLATS2 pipeline. The architecture design parameters we have examined are the number of instruction streams and the microarchitecture of the pipeline structure.

While FLATS2 issues the instruction from two instruction streams, an instruction can be issued from only one instruction stream with the same pipeline by changing the instruction issue logic.

FLATS2 provides microarchitectural parallelism to increase the performance. The *Symmetric* instruction set architecture [AAD90] treats register operands and memory operands symmetrically, and performs direct computation on memory. For example, the FLATS2 floating-point instruction can perform direct operation on memory. In Load/Store instruction-set architecture, on the other hand, computational instructions can only use registers as operands. The instructions that access memory are LOAD and STORE instructions only.

Varying these design parameters creates four variants of the FLATS2 pipeline in this two dimensional design space. To make sure that these machines are feasible, all pipelines are based on the FLATS2 pipeline.

6.2.1 The FLATS2 pipeline

The pipeline of FLATS2 consists of ten stages as follows:

IF — Instruction Fetch. Fetches the next instruction to execute from the instruction memory.

ID — Instruction Decode. Decodes the instruction and fetch the long immediate word from the instruction memory.

GVR — GV memory Read. Reads registers from the register memory.

GVEX — GV unit EXecution. For the GV instructions; Executes the instruction. For the load and store; Calculates the memory address.

DMR — Data Memory Read. For memory operand, reads the operands in the data memory.

GVW/EX1 — GV memory Write and EXecute 1 in SP unit. Writes the result for the register memory. For the SP instruction; starts the execution.

EX2, EX3, EX4 — EXecute in SP unit. For the SP instruction; executes the instruction.

DMW — Data Memory Write. Writes the result in data memory.

Figure 6.2 shows the FLATS2 pipeline.

Each stage is executed in one cycle. The instruction is issued every 2 cycles from two instruction streams. As a result, the instruction of an individual instruction stream

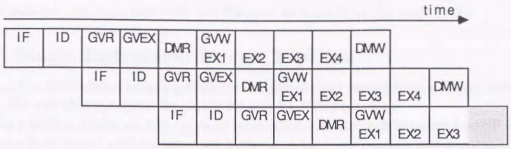


Figure 6.2: FLATS2 pipeline

is issued every 4 cycles. The SP unit executes the floating points operations and some integer operations such as MUL. The floating point registers are placed in the SP unit. The integer and address values are stored in the register memory. The GV instructions which perform the simple integer operation are executed in the GVEX stage. The GV instructions perform the computational operation only on register operands.

In each instruction stream, two computational instructions interlock when the instruction reaches the DMR stages and one of its source operands is not ready because it is the destination of a previous instruction that has not reached the DMW stage yet. If the instruction is about to enter DMR then the previous instruction is already done with EX4 and the result to be stored in DMW can be used as soon as it becomes available. The output of the EX stages can be written into *bypass registers*, which can be read in DMR. They allows the destination operand of a computational instruction to be used as the source of the next instruction. Physically, special hardware is used to detect that this memory operand is to be from the bypass registers not from the memory. The bypass registers eliminate the interlock due to data dependency. The bypass in the FLATS2 pipeline is shown in Figure 6.3.

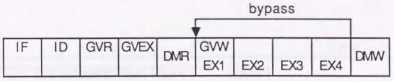


Figure 6.3: Bypass in FLATS2 pipeline

Resource conflict never occurs because each stages is executed by only one instruction at any time. Some integer computational instructions are executed in the SP unit. For example, the integer MUL instruction is executed in the SP unit, and the result must be written into the register memory, Since the next cycle is taken to write the result, these instructions always take two cycles.

In the FLATS2 pipeline, the compare-and-branch instruction can be executed without delay. Since the comparison is performed in the GVEX stages, the IF stage can select the next-PC according to the result of the comparison. The BL addressing is thought as a variant of the compare-and-branch instruction because the GVEX stage calculates the effective address and compare with the BL pair to branch at the next cycle.

6.2.2 Single Instruction Stream Pipeline

Changing the instruction issue logic allows the instruction issued from a single instruction stream. We call this machine the single stream FLATS2 machine.

In the pipeline, there are two types of interlocks: an interlock between two SP instructions (interlock delay) and an interlock between a load instruction and a computational instruction (load delay). The two SP instructions interlock when the source operand of the instruction is not ready because it is the destination of the next instruction which has not finished its execution stages yet. The other interlock occurs when a load instruction loads a register from memory and the next instruction read this register. When the next instruction use this register for address calculation, this interlock is known as an AG (Address Generation) interlock. For both interlocks, the pipe is stalled for one cycle. We assume that the operand written by the GV instruction except the load instruction is bypassed to the source operand of the next instruction because the operand is ready after the GVEX stage.

The compare-and-branch instruction is not provided for the single stream machine. The separate compare instruction sets the conditional flags and the next conditional branch instruction computes the branch-target and selects the next PC according to the flags. Since the compare instruction evaluate the condition at the GVEX stage, the branch instruction can use the flags at the ID stages. The branch instruction is executed without delay.

6.2.3 Load/Store Instruction Set

The Load/Store architecture is identical to the Symmetric architecture except for the Symmetric model's extra capability to operate directly on memory. Figure 6.4 shows the pipeline of the Load/Store machine based on the FLATS2 pipeline.

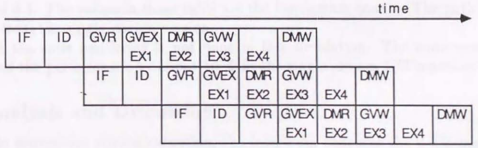


Figure 6.4: Pipeline of Load/Store machine

The set of GV instructions of FLATS2 is a Load/Store instruction set. The execution of SP unit starts after the GVR stage, because there is no memory operand for the SP instructions. As a result, the instruction is executed in 8 stages. We call this machine *L/S machine.*

Consider the interlocks in the L/S machine. As in FLATS2, the SP instructions and a load instruction may cause the interlocks when its destination operand is used as the source operand of the next instruction. Then the pipe is stalled for one cycle. In the L/S machine, we assume that the register MOV instruction between the floating point registers never cause an interlock.

We can think a version of L/S machine with two instruction streams. We call this machine *two streams L/S machine.* In this machine, there is no interlock between instructions.

For both L/S machines, the compare-and-branch instruction and BL addressing are not provided. In the pipeline, there is no resource conflict because each stage is executed by exactly one instruction at any time. As in FLATS2, the SP instruction which takes the GV register as a destination operand always takes two cycle.

6.2.4 Experiments

Since we assume that there is no hardware interlocking mechanism, the code generation phase of the compiler handles interlock due to the data dependencies. The execution time can be estimated by:

$$\text{Program execution time} = \text{path-length} \times \text{CPI} \times \text{cycle-time}$$

where path-length is the number of instructions executed, and CPI is the average number of cycles per instruction. Suppose there is no hardware interlocking or hardware data dependency resolution mechanism, the code generator inserts NOP codes to handle all the interlocks. In this case, path-length is increased by the number of NOP codes, while CPI is always one. We define the path-length as the instruction count without NOP codes. The instruction count includes the executed NOP codes to give the execution time. For multiple instruction stream machines by a cyclic pipeline architecture, the instruction count is the total number of instruction executed in all instruction streams.

In the simulation, we used the same workloads in Chapter 2. The results are shown in Table 6.3 and 6.4. The values in these table are the instruction counts. The path-length is computed from the single stream version.

Note that the code reordering is not done in this simulation. The code reordering would improve the performance by about 10 % in the single stream L/S machine.

6.2.5 Analysis and Discussion

For the single instruction stream execution, the interlocks degrades the performance by 20%-60% from the path length; the path length indicate the ideal execution time without interlock delay. Since there is no interlock in cyclic pipeline machines, the performance degradation is caused by the synchronization overhead and the time waiting the other processes.

Program	size	path length	FLATS2 single stream (ratio)	FLATS2 without BL (ratio)	FLATS2 with BL (ratio)
Inner product	100	444	748(1.68)	818(1.84)	526(1.19)
	1000	4044	7087(1.752)	6218(1.54)	3226(0.80)
Linpack	100×100				
dgefa		1308232	2051575(1.57)	1351441(1.03)	1009172(0.77)
dgesl		44165	66659(1.51)	85750(1.94)	57536(1.30)
TOTAL		135297	2118234(1.56)	1437191(1.06)	1066708(0.79)
FEM_BAND	16×16				
MATGEN		327183	4366797(1.34)	359781(1.10)	334377 (1.02)
SOLV		357022	543556(1.52)	678909(1.90)	686502(1.93)
TOTAL		684205	980353(1.43)	1048690(1.53)	1020879(1.49)
FEM_BAND	32×32				
MATGEN		1320519	1770581(1.34)	1427269(1.08)	1385929(1.05)
SOLV		3640710	5851324(1.67)	6631005(1.82)	6554342(1.80)
TOTAL		4961229	7621905(1.54)	8058274(1.62)	7940271(1.60)
FEM_ICCG	32×32				
MATGEN		2783292	3446593(1.24)	3208541(1.15)	3126960(1.60)
SOLV		5991694	7640083(1.28)	9981139(1.67)	9669397(1.61)
TOTAL		8774986	11086676(1.26)	13189680(1.50)	12796357(1.46)
FEM_BAND	16×16				
SOLV(BL)		357022	543556(1.52)	—	514931(1.42)
TOTAL		684205	980353(1.43)	—	849308(1.44)

Ratio: a ratio to the path length.

Table 6.3: Instruction counts in FLATS2 and single stream FLATS2

The performance improvement obtained by N processors is given as follows:

$$Speedup(N) = \frac{1}{1 - \alpha + \alpha/N} - Overhead(N)$$

where α is the ratio of the total execution time of parallel code. While the execution time of parallel codes is reduced by the number of processors, The synchronization cost $Overhead(N)$ decreases the speedup for the parallel execution. If the overhead is negligible small, the speedup with two instruction stream is $2/(2 - \alpha)$. For example, to obtain the performance improvement by 50%, α must be $2/3$.

For the inner product computation, the parallel execution becomes less for the smaller vector. If the size is less than 100, then the performance of two instruction streams can not exceed that of single instruction streams in both of FLATS2 and L/S machines.

The reason that the L/S machine increase the performance more than FLATS2 in the two instruction stream compared to the single stream machine is that the path length in the loop is shorter in FLATS2 while the path length to setup iterations before the loop

Program	size	path length	L/S machine single stream (ratio)	L/S machine two streams (ratio)
Inner product	100	1049	1255(1.20)	1234(1.18)
	1000	10049	12055(1.20)	10234(1.02)
Linpack	100 × 100			
dgefa		3359007	4444457(1.32)	3421009(1.02)
dgesl		106033	138236(1.30)	129244(1.22)
TOTAL		3464040	4582693(1.32)	3550253(1.03)
FEM_BAND	16 × 16			
MATGEN		477734	598662(1.25)	533629(1.12)
SOLV		747725	886378(1.19)	1075745(1.44)
TOTAL		1225459	1485040(1.21)	1609374(1.31)

Ratio: a ratio to the path length.

Table 6.4: Instruction counts in L/S machines

is still the same. We observe the same situation in Linpack and FEM_BAND. Since the optimization of the loop reduces the path length of the loop, the performance improvement for the optimized code would be less than that of the non-optimized code in parallel programming. If the simpler instruction set architecture makes the cycle time shorter than a complicated instruction set, it would obtain better performance.

The two instruction stream machines can execute the program of Linpack better than the single instruction stream machines because the routine **dgefa** involves large parallel codes. For **dgesl**, the average of vector length is less than 100 and then its performance is not improved as the inner product of the size 100.

Unfortunately, the performance of SOLV in both FEM_BAND and FEM_ICCG can not be improved by the two instruction stream machines. Since the size of the loops in the computation is small, the iterations over rows are distributed to each stream. These streams synchronize each other with the synchronization vector of rows. Because the synchronization cost is large compared to the performance improvement obtained by two instruction streams, its performance can not exceed the performance of single instruction stream machine even if the size is large. More instruction streams would increase the performance as shown in Chapter 2.

In MATGEN, each element vector is computed by each instruction streams independently and each stream seldom waits at the critical section to add it up to the global matrix. Since the execution time is dominated by SOLV for larger size, the contribution of MATGEN become smaller for the total execution time.

For the inner product and Linpack, the BL addressing eliminates the overhead to iterate the loop. Since the code to control the loop does not do effective jobs in the computation, the code can be thought as a part of sequential code. The BL addressing can reduce the execution time in parallel execution as same as in sequential execution. In FEM_BAND and FEM_ICCG, the BL addressing can not be used because the synchronization code is the first instruction in the most loops to synchronize the data. Although the BL addressing mode is used for some loop, its contribution is small. In another version of SOLV shown as SOLV(BL) in Table 6.3, the program is modified to make use of the BL addressing. Since it uses the process number, the program works only on two instruction streams, while the original SOLV is independent from the number of processors. SOLV(BL) can improve the performance over that of the single stream machines. Thanks to the two instruction stream, FLATS2 can implement the BL addressing without delay. As a result, BL addressing exploit microarchitectural parallelism in the cyclic pipeline architecture effectively.

Chapter 7

Summary and Conclusions

In this thesis, we have presented an evaluation of cyclic pipeline machines for a highly pipelined processor. Although the performance of a highly pipelined processor is limited by instruction-level parallelism in application programs in a conventional approach of single instruction stream, a cyclic pipeline machine enables multiple instruction streams to exploit more parallelism in the parallel program of scientific workload even in a single highly pipelined processor. The simulation results indicate that effective pipelining in the individual instruction stream of the cyclic pipeline machine increases the performance to maximize the utilization of resources in a highly pipelined processor.

New technologies such as GaAs and QFP prefer a highly pipelined architecture to make use of the raw speed of components efficiently. A cyclic pipeline machine provides an alternative architectural solution of a highly pipelined system. Even in silicon VLSI technology, it may be an interesting architecture, especially for asynchronous self-timed systems.

We have evaluated an experimental cyclic pipeline computer, FLATS2 with FLATS2 FORTRAN compiler. The FLATS2 FORTRAN compiler implements several optimization algorithms with static single assignment (SSA) form to generate quality code. FLATS2 has two instruction streams in its ten pipeline stages. The parallel directives of our FORTRAN enables us to exploit parallelism in parallel program with multiple instruction streams. While the performance of Linpack benchmark is 1.96 MFLOPS with one instruction stream, the performance is increased with two instruction stream to 3.70 MFLOPS.

In FLATS2, we can exploit the microarchitectural parallelism with the BL addressing, which integrates memory addressing and range checking. With the BL addressing, the test for termination of the loop of array computation can be overlapped with the computational operation. The FLATS2 FORTRAN compiler generates optimized code with BL addressing to reduce the execution time for array computation. Since array computation often dominates scientific workloads, we reduce the execution time of scientific application by 10-30%. In a parallelized program, the BL addressing reduces the execution time of the loop which is distributed to each instruction stream. The cyclic pipeline computer can implement the BL addressing to remove the interlock due to control dependency involved in the BL addressing.

We have examined the FLATS2 pipeline compared to the single instruction stream pipeline with the same structure by the simulator. For highly parallel programs such as Linpack, FLATS2 can obtain the performance improvement over the single instruction stream pipeline. But for less parallel program, the synchronization overhead between instruction streams is larger than the overhead due to interlocks in single instruction stream.

FLATS2 exploits the microarchitectural parallelism by rather complicated instruction set, which allows the memory operands to be accessed for both load and store in one instruction. It reduces the number of instructions to be executed. If we use the Load/Store instruction set in the modified FLATS2 pipeline, the performance improvement contributed by the two instruction streams is larger than that of FLATS2. The reason is that the execution time of parallel loop is large compare to the sequential code in Load/Store architecture. Nevertheless, we can obtain the better performance in FLATS2 than the Load/Store architecture because the number of instruction to be executed is less in FLATS2. If the simpler instruction set makes the cycle time shorter, it would obtain better performance.

The cyclic pipeline architecture is a single processor architecture, while it allows the multiple instruction streams. We expect further research to study the multiprocessor system with cyclic pipeline processors. As the cyclic pipeline architecture reduce the impact due to memory access latency by pipelined access, it can reduce the impact due to the network latency [Mor90]. In a cyclic pipeline computer, the number of instruction stream is fixed and they are scheduled statically. To schedule the instruction streams dynamically, we would have to place several queues to handle the activity of the instruction stream, as HEP and data flow computers.

Since there is no memory conflict among instruction streams of a cyclic pipeline computer, the synchronization operation is very cheap. Although the parallelism of programs may be expressed explicitly in the form of language extension as in our simulations, automatic parallelization compilers, including automatic program restructuring, is expected to exploit parallelism for a cyclic pipeline machine with minimum programming efforts.

Appendix A

RUN-TIME CHECKING IN LISP BY INTEGRATING MEMORY ADDRESSING AND RANGE CHECKING

This paper presents the design of a Lisp system using the BL addressing of FLATS2. The paper was published in the conference proceedings of 16th International Symposium on Computer Architecture[SIG89].

RUN-TIME CHECKING IN LISP BY INTEGRATING MEMORY ADDRESSING AND RANGE CHECKING

Mitsuhisa Sato[*], Shuichi Ichikawa[*] and Eiichi Goto[* **]

[*] Research Development Corporation of Japan (JRDC),
5-6-4 Tsukiji, Chou-ku, Tokyo 104, Japan
[**] University of Tokyo, Department of Information Science,
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

Abstract

This paper describes the BL addressing mode and the address tag in FLATS2 machine, which is a general-purpose MIMD computer now under construction. The BL addressing mode integrates memory accessing and range checking by hardware. Address tag is a bit in word, which indicates the capability for memory access. Combining them together, efficient memory protection is provided at run-time. It reduces the cost of run-time type checking in Lisp by checking the address tag and the address of a pointer against the range of the region associated to a type, in parallel with the memory access. The arithmetic instructions check the address tags of operands to support the generic arithmetic in Lisp. We can also make use of this scheme to check the number of arguments and multiple return values and to check array-bounds to support faster execution of Common Lisp program. These facilities are not specific to Lisp, so that they can be used more generally than other tagged architectures.

1. Introduction

Lisp includes a number of features that make Lisp programs difficult to execute efficiently on conventional machines. One of these features is *dynamic type checking*. Lisp associates type with values rather than identifiers. Dynamic type checking is implemented by adding to each data object a tag that encodes type information. Most of Lisp machines provide architectural support for tag manipulation, together with other works related to dynamic type checking and generic operations, to execute Lisp programs efficiently.

In this paper, we propose new architectural supports for run-time checking in Lisp, *BL addressing mode* and *address tag*, which can be used more generally than tagged architecture. BL addressing mode is integration of memory addressing and range checking. The effective address is checked against the specified pair of base and limit address in registers during memory access. An address tag is a bit in a word, which indicates whether the word is an address or not. It checks the capability of memory access.

In Lisp, an object is represented as a pointer or an immediate value. Address tag distinguishes a pointer object, from an immediate data type such as fixnum. Generic arithmetic instructions check the address tag of operand to perform the operation according to the data type. By allocating the same type of objects in a segment, BL addressing mode checks the type of a pointer object during memory access by testing which segment the pointer points into. So, we can check the data type by the range checking instead of a tag. Like tag checking of tagged architecture, the range checking overlaps the memory access.

FLATS2 is an MIMD computer by cyclic pipeline architecture(CPA[9]), which exploits these features. It is a general-purpose computer, not lisp-specific one, because BL addressing mode and address tag are general architectural supports, not lisp-specific supports.

One of FLATS2's target languages is Common Lisp [13]. Common Lisp is an "industrial strength" dialect of Lisp providing a wide variety of data types and control structures. In this paper, we explain how FLATS2 supports faster execution of Common Lisp programs with BL addressing mode and address tag.

In Section 2, we describe the basic concept of BL addressing mode and address tag. Section 3 describes the basic architecture of FLATS2, and in Section 4, we explain the design of Common Lisp using our architectural support. In Section 5, we discuss BL addressing mode and address tag (compared to other tagged architectures) and their hardware implementation of FLATS2.

2. The concept of BL addressing mode and address tag

2.1 BL addressing mode

To access the memory, addressing modes locate operands in memory. In BL addressing mode, the effective address is checked by comparing the address with base address and limit address whenever the operand in memory is accessed. It allows memory access and range checking to be performed in parallel by hardware. If the effective address is not in the range between given base and limit, either a branch to a specified location or a trap occurs.

BL addressing modes are specified in the following form:

<BL> : <addressing mode>, <label>

where **<addressing mode>** is **displacement(address)** or **index(address)**. (**Displacement** is a constant. Address and index may be a variable specified in a register.) <BL> gives the base and limit as a pair, which is called *BL pair*. <label> may be omitted and then a trap occurs instead of branching.

BL addressing mode provides memory protection on small domain at run-time. For example, BL addressing mode is used for array-bounds checking. It checks the indices to make sure that the reference is inside the given vector specified by BL pair. It is desirable to check array bounds to improve program reliability even in statically typed languages such as FORTRAN, PASCAL.

2.2 Run-time data type checking by BL-addressing mode

In dynamically typed languages such as Lisp, APL, and Icon, all data objects are allocated dynamically at run-time, and their types must be checked at run-time. Objects in memory are referenced indirectly through pointers. Each type of the objects can be allocated in the heap space associated with its type. We call a heap space corresponding to each type, *type segment*. The type of the pointer, then, can be checked by testing which type segment the pointer points into.

The compiler knows the expected data type and how to access the object in memory through the pointer. By using BL addressing mode with the BL pair of the type segment, the type can be checked in parallel with data access. For example in Figure 1, a cons cell in Lisp is represented by two words in the cons cell segment, where BL-cons indicates. The primitive operations **car**, **cdr** on a pointer **p** to the cons cell are performed by loading from memory with BL addressing mode respectively as follows:

car(p) := BL-cons : (p)
cdr(p) := BL-cons : 4(p)

Here the size of a word is 4 bytes. If **p** points outside of BL-cons, a trap occurs.

The predicate on list data type, **listp** is implemented simply by range checking on a pointer with the BL of cons cell segment.

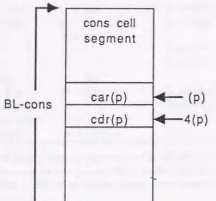


Figure 1. CAR/CDR operations by BL addressing mode

2.3 Address tag

Even though the address is the most fundamental property in computation, most of computers have no means to distinguish the address from just a data. In FLATS2, every data word on either memory or registers has one bit tag which indicates whether a word is an address or not.

Only *load effective address* instruction and memory access instructions can calculate an address, so that an address is defined in well-defined way. Both of base and limit of a BL pair must have address tags, as well as *address* in <addressing mode>. If a non-address word is used as an address entity in BL addressing mode, the exception occurs. The calculated effective address can never be outside of the range between given base and limit addresses.

At beginning of the execution of a program, the BL pair of the entire user's program space are given by the operating system. During the execution, no address outside of initial BL pair can be produced. So BL addressing mode and address tag work as memory protection mechanism on the single virtual space of FLATS2. We call this protection scheme *BL scheme*. Combining address tag and BL address checking can be thought as a kind of implementation of *capabilities* [16].

3. Basic architecture of FLATS2

In this section, we describe the basic architecture of FLATS2 briefly. The cyclic pipeline architecture of FLATS2 implements two instruction/data streams (virtual processors†) in one hardware. A more detail description is given in [7].

3.1 Memory and registers

FLATS2 provides the 32-bit single virtual space. The lower half of the address spaces is D space, which is used to store data. The upper half is divided into the space for instructions (I space) and the space for register-mapped memory (GV space). FLATS2 has 64 general purpose registers and four special purpose floating point registers. Each general purpose registers contain 33 bit word, 32 bits data and one bit for address tag. The general registers are divided into two groups: 32 global register and 32 local frame register. They are mapped on GV space by Global Frame Pointer (GFP) and Local Frame Pointer (CFP). Local frame registers are saved/restored by CALL/RETURN instruction to make function call faster. CALL instruction changes the local frame by incrementing CFP and saves the old processor status including CPF and the program counter (PC) in the new local frame. Floating point registers contain 64 bits floating point.

D space is byte-addressed. Each word (32 bit) word-aligned on memory has another bit for an address tag. The load/store operations transfer the entire 33 bit word between memory and registers.

3.2 Instructions

Most of instructions in FLATS2 are executed in one instruction cycle. Its basic formats is 64 bits in size, as shown in Figure 2. The fields *s1, s2, s3* specify the general purpose registers to be fetched. *GV-op* field specifies the operation between the general registers, and the addressing modes with the short displacement *d1, d2* form -128 to 127. The displacement can be extended to the long displacement (32 bit) in the following instruction word. The displacement field may be used as a immediate operand. *SP-op* field controls the

† This term *virtual processor* should not be confused with a virtual processor of the memory management in the operating system. In this paper, a virtual processor is a instruction/data stream in MIMD.

integer and floating points arithmetic unit, called *SP unit*.

The instructions without *SP-op* field perform address calculation, byte and short load/store, integer arithmetic operation, and a few of lisp specific operation. This set of instructions are basically load/store instruction set. These instructions use *SP-op* field for other purpose.

SP unit instructions (SP instruction) perform the floating and integer arithmetic operation between the general registers, floating point registers and memory operands, and store the result to either memory or registers. We can achieve full through-put of memory by the operation between memory operands in one instruction cycle.

The field *j1* specifies the relative jump location for in-word branching. It controls the exception of BL-addressing mode, as explained later.

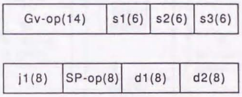


Figure 2. Basic instruction format of FLATS2

3.3 Variants of BL addressing mode

In addition to the basic BL addressing mode described in previous section, FLATS2 has some variants of BL addressing mode. FLATS2's addressing modes are listed in Table 1. The general purpose registers are used as either BL pair, index or address register. A BL pair is formed by an even/odd registers pair. The effective address is checked against BL pair in register, in any BL addressing modes.

Notation	Effective Address	Side Effect
#xxx	(Immediate)	
gr*n*, vr*n*, S*, P*	(Register)	
b:disp(p)	$p + disp$	
b:>disp(p)	$p + disp$	$p \mathrel{+}= disp^{1}$
b:<disp(p)	p	$p \mathrel{+}= disp^{2}$
b@i	$b + i$	
b@>i	$b + i$	$b \mathrel{+}= i^{1}$
b@<i	b	$b \mathrel{+}= i^{2}$
b@disp	$b + disp$	
b@>disp	$b + disp$	$b \mathrel{+}= disp^{1}$
b@<disp	b	$b \mathrel{+}= disp^{2}$
b@<disp(i)	$b + i + disp$	
b:disp(p)i	$p + i + disp$	

NOTE: *b* specifies odd/even register pair as BL. In effective address and side effect, *b* denotes the base register of BL. *i* specifies a register as index register. *p* specifies a register as address register. *disp* specifies a constant as displacement.

*) S,P are floating point registers. 1) pre-modify 2) post-modify

Table 1. BL addressing mode of FLATS2

BL-addressing modes can involve side effects, which are called *post-modify* and *pre-modify*. The side effect of post-modify (pre-modify) changes the base register or pointer register after (before) the effective address calculation according to the addressing modes. For example, the combination of index mode and post-modify specifies the base address of BL pair as effective address and update the base register of BL pair incrementing by the index register. These side effects are used as post/pre-increment/decrement addressing mode, in conventional processors such as VAX or 68000, and each register can be modified by any displacement in both direction. When the modified address is outside of the range of BL pair, the address tag of the modified register is set to 0 and then cannot be used as an address any more.

The exception handling in BL-addressing mode is different from the definition described in previous section. Without specifying the jump target, the exception of BL addressing mode causes a trap. When the jump field is specified in BL addressing mode, this jump is executed if the memory access is completed successfully, otherwise the next instruction is executed. The following instructions handle the exception.

Since the pointer whose most significant bit is 1 points into the I space or GV space, it is a pointer to function code or local frame. Such a pointer is treated specially in BL addressing mode.

4. Design of Lisp on FLATS2

In this section, we describe how FLATS2 supports faster execution of Common Lisp program by using BL addressing mode and address tag.

4.1 Data formats and structures of Lisp

The formats and structures of data types are designed with BL addressing mode and address tag in mind, because the implementation of type checking depends on the data representation. Lisp objects are 33 bits long, 32 bits word and an address tag. A type is either an immediate type or a pointer type. An immediate type object has its address tag cleared. The type, fixnum is only an immediate type. The value of fixnum is stored in 32 bit word.

All data types except for fixnum are referenced through pointer objects. The word of a pointer object specifies a memory location where the data associated with the pointer object are stored: cons cells, symbols, ratio and floating point are represented in this way. These fixed length objects are allocated in *type segment* associated with a pointer type. The data of fixed-length objects are accessed through a pointer object by BL-addressing mode with constant offset and BL of its type segment.

For variable length object such as vector, array and string, the fixed-length header, which contains the descriptor of the variable length data, is stored in its type segment. The descriptor is a BL pair of variable length data in *data heap*. The BL pair of a vector is loaded from its header to access the elements of vector. BL addressing mode checks the accesses on the vector without overhead. Run time checking for vector access is rather natural usage for BL addressing mode.

4.2 Operations on list elements

Lists are one of the most frequently used data structure in Lisp. The basic operations required are **cons** to add an element to a list, and **car** and **cdr** to access the first element and tail of a list. Lists are sequences represented in the form of linked cells, called *cons cell*. **Car** and **cdr** are performed by load operation on a cons cell with BL addressing mode.

Car and **cdr** are defined in Common Lisp not only for a pointer to cons cell but also for a pointer to nil. In Common Lisp as in most Lisp dialects, the symbol **nil** is used to represent the empty list and the false value for boolean test. Nil is a special object which belongs to both list and symbol. Although it is a symbol like any other symbols, it appears to be treated as a variable when evaluated. The data structure of a symbol object consists of several words, including value cell, the function definition cell, property list, print name, package cell of a symbol and other attributes. The symbol structure is designed so that its value cell and function cell can be accessed by **car** and **cdr** operation as if it is a cons cell. The cons cell type segment adjoins the symbol type segment. The symbol **nil** is located on the intersection of both type segments. The BL pair used in **car** and **cdr** operation includes the **nil** symbol as shown in Figure 3. In case of **nil** symbol, its value cell and function definition cell are also **nil**.†

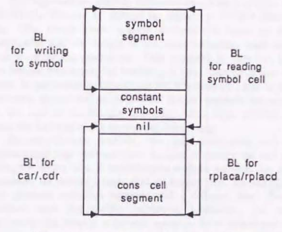


Figure 3. Cons cell segment and symbol segment

4.3 Constant symbols and read-only protection

Common lisp allows symbols to name a constant value by **defconstant** declaration. Once a name has been declared by **defconstant** as a constant, any further assignment to or binding of that named symbol is an error. In the interpreter, such binding can be detected by checking the attribute of symbols indicating the named constant with run-time overhead. The compiler can replace references to the name of a constant by its value of the constant in source level, and may also choose to issue warnings about bindings of the lexical variable of the same name.

However, even in compiled code, it is necessary to check the bindings to the system-supplied constant such as **nil** and **t**. Such read-only attributes of these objects can be implemented by selecting BL pair used in store operations. The shallow bindings to special variables modify the value cell of a symbol using BL pair of symbol type segment excluding read-only constants.

Rplaca and **rplacd** operations modify the list structures. Unlike **car** and **cdr**, these use BL excluding **nil** symbols to avoid modifying the value of **nil**.

† The value of nil symbol is defined as nil itself. This prevents nil symbol from being defined as a function name. Note that there is no description in [13] whether the function named "nil" is permitted to be defined or not. We may layout the internal structure as to use other cell instead of the function cell.

4.4 Explicit type operations

Type checking is required in checking operations specified at the source level, such as the function **atom**. The predicate for testing an individual primitive type is implemented by checking a object pointer against the BL pair of the associated type segment.

The data types defined in Common Lisp are arranged into a hierarchy defined by the subset relationship. For example, a type number includes integer, ratio, floating-point numbers and complex as its subtypes. The layout of type segments in memory is determined as to make it easy to test supertypes of basic data types (Figure 4). The type segments of subtypes are placed in the range of supertype. We can test the supertype by checking against the BL of the supertype. For example, the predicate function **numberp** tests the pointer object against BL pair including all subtype segment of number type.

We can implement the intersectional type of two types. Nil is an example of intersectional types of cons and symbol. The predicates **endp** and **consp** test against the BL excluding **nil** while the predicate **listp** uses the BL including **nil** as car and cdr.

The range checking for the pointer object is done by *load effective address* instruction with the pointer mode of BL addressing modes. This instruction branches by in-word jump field without delay.

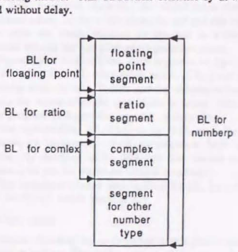


Figure 4. Layout of type segments for number

4.5 Generic arithmetic operations and Fixnums

Lisp generic functions must determine the types of their operands at run-time in order to perform the appropriate action.

For arithmetic operations, the most frequent type of operands is fixnum, which is the only type represented by an immediate type. To avoid testing the types of operands explicitly, the hardware performs an address tag check in parallel with arithmetic operations. FLATS2 has two kinds of arithmetic operation; *Lisp arithmetic* instructions checks address tags of both operands. Another kind of arithmetic instructions ignore address tags to perform the arithmetic operation on both value of data and produce non address result.

If both operands in Lisp arithmetic have no address tag, then arithmetic operation is completed just as conventional arithmetic operations. If either operand has an address tag, the instruction forces a trap to software. The software dispatches on the types of each operand to determine if both are numbers, coerces one of them if they have different types, and then perform the generic arithmetic operations by proper sequence of machine instructions. When the overflow condition is detected, a trap also occurs to extend the result to *bignum*.

There are two equality predicates used frequently in Common Lisp: EQ testing for the identical objects and EQL testing for identical objects or equality for numbers of the same type. The EQ test is performed simply by comparing the objects including address tags. The EQL comparison is more complicated because the pointer objects of numbers which have the same value may be located in different location. As Lisp generic arithmetic, Lisp compare instruction tests and traps to software if either operand has an address tag. If both operands are non-address words, then comparison is completed as conventional compare instruction.

This approach on generic arithmetic operation is similar to SPUR[14]. We use an address tag instead of SPUR's *fixnum* tag. This integer-biased generic arithmetic is based on the assumption that the integer is the most commonly used data type in arithmetic operations. This approach is very fast for the integer data types, but handling of other data types can be slow. Its performance depends on the treatment of a trap in the operating system and on how often integer operands are used. In the case of non-integer operands, rather large amount of time can be required for recovering from a trap.

In compile-time analysis, the type declaration and a sophisticated type inference can be used to reduce the cost of using the wrong bias. If compile-time analysis indicates that the operands are probably other than of integer type, the compiler can generate code that operates with a different bias. Even without such lisp-specific arithmetic instruction, we can implement the generic arithmetic operation by 4 instructions in software; 2 instructions for testing address tags of both operands, 1 for adding, 1 for checking overflow. This software approach is also biased to integer type, but we can avoid the overhead by recovering from a trap.

4.6 Function call

The stack is provided in D space for passing arguments, returning values, shallow binding of special variables and spilling registers. The temporary variable whose lifetime is known to end when a function returns are allocated in the local frame registers or on the stack. Because of lexical scope rule, the local variables which can be referenced by other functions must be allocated either on stack or in closure type segment. Switching the local frame registers by CALL/RETURN instructions avoids saving and restoring the temporaries of the calling function to make the function call faster.

The global registers are used to pass the arguments and return values. Common Lisp allows the variable number of arguments to be passed to functions, and provides a variety of specification on parameters. Called functions must check the number of arguments to take each argument according to the parameter specifier.

The calling function pushes the arguments onto the stack, and then uses the global register to pass the BL pair indicating high and low address of the arguments on stack instead of the number of arguments. The BL pair can be calculated from the stack pointer and the number of arguments in one instruction, called *make BL pair (mkbl)*. In order to setup the parameter variables, the arguments can be accessed by BL addressing mode in parallel with checking the number of arguments. The exceptions may cause a trap for an error, or move the default value to the parameter.

Common Lisp also allows the function to return the multiple values. Like passing argument, the called function pushes all the returned values on the stack and then returns the BL pair indicating high and low address of the return values. When control returns to the caller, it takes the return value through the returned BL pair to convert the values to the form required by the caller. Checking the number of the return value can be done during accessing the return value on the stack.

4.7 Global registers and system constants

FLATS2 has 32 global registers. The most frequently referenced system values are kept in global registers. These system values are the constant nil value, the stack pointer, and the BL pair for the entire own memory space. The BL pairs for checking the most common type such as cons cell are also kept in the global registers. We must choose carefully which BL pair of type segments should be kept in the global register.

BL pairs of other type segments are placed in memory. They are loaded into local frame registers or free global registers whenever needed. The compile-time analysis keeps such BL pairs in registers in the compiled function.

4.8 Combining predicate and access to object

In Lisp programs, accessing an object often follows the predicates testing its type. For example, **car** and **cdr** often take place after the check whether its operand is a list by the predicate **listp** or the complementary predicate **atom**.

The compiler can combine such sequence of type checking and accessing into BL addressing mode. The jump specifies the action after the data access, and the successive instructions execute the action after the type check is failed. The example of the program in **Appendix** uses this optimization.

This optimization is effective to implement the generic operation on *sequence*, which encompasses both lists and vectors. By checking on list data type first, we can do generic operations on the list sequence without overhead.

This technique is used also in FLATS1 [5], by using *error jump* facility of car/cdr instruction.

4.9 Array access

Bounds checking on array access is straightforward for BL addressing modes. The indices are checked against the bounds by index mode with BL pair of the array. When an array is accessed in loop, the compiler keeps the BL pair of its array in register as possible.

If an array has a fill pointer, it may be represented as an pointer instead of an index. Its elements are accessed through the BL pair of the array's base address and the fill pointer. If the access failed, the fill pointer is modified by a new address and then is accessed again. The element inside of the fill pointer can be accessed without overhead of checking against a fill pointer explicitly.

4.10 Floating point

FLATS2 supports single precisions (32 bits) and doule precisions (64 bits) of floating points numbers, which are basically compatible to the DEC floating points format [15]. In FLATS2, no immediate floating-point data types are used such as the single precision data type in Symbolics 3600 [8], because an immediate object represents only the fixnum data type. All floating point objects are represented through the pointer.

Most floating arithmetics are performed in one instruction cycle between memory operand and the floating point registers by SP instruction. Temporary results of floating points arithmetic can be kept either in the register or on the stack as two fixnum objects. For floating-point operands, the integer-biased generic arithmetic operation first have to dispatch on the types of their operands and then use the pointers to load the floating point numbers from memory to register. If either of the operands is known to be floating-point types at compile-time by the type declaration, the floating arithmetic instruction can access the operand on memory in parallel with type checking against the BL pair of floating point segment. When the operand is the pointer object of the floating type, we don't have to loose time for checking data types. If type check failed, the object is converted to the floating point data type and then arithmetic operation is done again.

4.11 Data allocation and storage management

Allocating storage for an object in a type segment is accomplished simply by incrementing the free pointer associated with that segment. For example, cons operation which allocates a cons cell, is accomplished by incrementing the free pointer associated with cons cell segment. When a free pointer for any space is incremented, a check must be made to see if the free space runs out. If so, the garbage collector is invoked.

The free area in a type segment is also managed as the BL pair of the free pointer and the limit pointer, which points the end of the free area. For *cons* operation, the special instruction is provided to store two words to the memory at the base of BL to increment the base pointer as side effect, and move the old base pointer to the target register as a newly created cons object. All these operations are executed in one instruction, called *allocate space (allocs)*. To allocate other types of objects, the side effect of BL addressing mode of store operation increments the base of BL pair pointing the free area. In either case, when the new base address runs over its limit address, the address tag of the base of BL pair is cleared to notify that the free space runs out. Next attempt for allocating the object will fail and cause a trap to reclaim memory. We minimize the time for the simple object allocations such as cons cell, floating-point by expanding it inline with a few instructions.

Storage is reclaimed with a stop-and-copy garbage collector by the following reasons:

— Copying improves paging performance because it compacts objects.

— If free space runs out, it is necessary to relocate the object in the segment into the other bigger segment in order to enlarge the size of free space. Copying makes it easy to implement the relocation of segments.

— We have no room for keeping extra bits in the pointer object for incremental GC.

Garbage collector needs to distinguish pointers from data. Address tags allow the pointers to be recognized by their tags.

As described in section 4.1, the data of variable-length data type is allocated in data heap. All free objects in data heap are managed by the free list.

5. Discussion

In this section, we discuss the feature of BL addressing mode compared to tagged architecture, and the hardware implementation of the BL addressing mode in FLATS2. It is difficult to evaluate the performance of Lisp on a specific architecture. Its performance depends on many factors including compiler quality and the operating system environment as well as the architectural support.

5.1 Full tagged architecture

Run-time type checking is implemented by adding a tag to each data object to encode the type of that object in either software or hardware. Most of Lisp machines such as Symbolics 3600 [8], TI Explorer [1], SPUR [14] and FLATS1 [8], adopt a tag architecture to execute Lisp programs efficiently. These machines provides a few of bits for the tag to represent the data types. We call such class of tag architecture *full tagged architecture*. The Lisp machines with tagged architecture support tag checking operations in parallel with other operations.

In addition to run-time checking, the tagged architecture offers several advantages. In most of Lisp machines except for SPUR, high-level instructions corresponding to Lisp primitive functions are micro-coded in firmware. Because micro-programming allows more parallelism, it could be faster than that of the software implementation on conventional machines. For example in TI Explorer VLSI processor [1], a tag dispatch table in chip is used by micro code to support generic arithmetic operations. If the test for the most common integer case failed, the micro-code dispatches on the type of operands. Further, language-specific instruction set designed by micro-program provides the high-level instructions easy to be compiled into, and compact codes, which increase code density. However, micro-code requires the additional hardware to control the sequence of micro-instructions. It would add on cycle time and decrease the total performance.

SPUR is a RISC processor, which incorporates tagged architecture into RISC. It has a few lisp-specific instructions for tag read/write, list access, integer-biased generic arithmetic. However, SPUR does not allow parallel checking on memory access other than list access, so tags except for list are checked by software.

FLATS2 provides only one bit tag indicating the capability for accessing the memory. The hardware related to the address tag is hard-wired. A valid address is created only from the effective address calculation by BL addressing modes. It enables the memory protection on small domains with BL addressing modes.

5.2 Run-time checking in full tagged architecture and BL addressing mode

Both of full tag checking and BL addressing modes provide dynamic type checking facility on the pointer object during address calculation.

In tag checking, the expected tag could be specified in a register, as an immediate, or in the opcode. The hardware testing is limited to a simple tag check, and is sufficient for list accessing. But for vector operation, array bounds checking would still have to be done in software or firmware.

In BL addressing mode of FLATS2, the BL pairs for type segments must be specified in registers. BLs for the type segments may change by relocation at run-time. But if a BL pair cannot be kept in the register, it takes more instructions to load it from memory. Some of the BL pairs which are used frequently, can be kept in global registers. The compile-time analysis can keep other BL pairs in registers during register allocation phase. The pair of registers occupy double registers, so BL addressing mode requires the large number of registers in order to make BL addressing mode work effectively.

As well as type checking at run-time, it can be effective in a number of situations, as described in the previous section. BL addressing mode provides more flexible checking mechanism at run-time than tag checking does.

The type checking on pointer objects is implemented by the address range checking in BL addressing mode. Type checking by BL-addressing mode requires objects of the same data type to be allocated in contiguous area on memory. Even in full tagged architecture, some Lisp systems allocates each objects in this way to make the storage management easier, so that it is not serious restriction on implementing Lisp system.

It is sometimes useful to dispatch on the type of the object like in a case statment. Such type checking is expensive in BL addressing mode. In full tagged architecture, it is easy to recognize data type from the object pointer simply by extracting the tag, because the tag in a object describes its data type. But in our approach, it is necessary to check the pointer address against BL pairs of all type-segments sequentially.† Sequential type checking can detect the most common type faster, but the detection on other data type can be slower. Fortunately, most of type checking in Lisp source program are performed by the predicate function of an individual data type. Dispathing on the data type is rarely used in compiled code.

5.3 The implementation of BL-addressing mode in FLATS2 CPC architecture

Steenkiste and Hennessy [11,12] studied on the cost of tag checking in respect of software and hardware support in RISC architecture. According to their results, run-time tag checking for primitive Lisp operations in software costs about 25 % of total execution time on average in Gabriel's benchmark programs[3]. By overlapping this tag checking and other operations in hardware, we can reduce this cost to obtain substantial speed up of execution.

Rather complicated instructions for type checking must be faster than a sequence of simpler instruction if they are to give performance improvement. The proposed hardware schemes must be evaluated not only for instruction count or convenience of software, but also for potential negative factor on the processor's cycle time.

In the case of FLATS2, BL addressing modes always take a BL pair as register operand, and jump field as an immediate, so rather long instruction is required. It may decrease code density and increase paging activity. Additional hardware required for BL addressing mode consists of two comparators for range checking and some logic for address tags. To read BL pair from register, an additional read port is required in register file. The BL range checking on effective address overlaps load/store operation form/to memory. If such hardware is added to the critical path of execution, it would be negative impact on the cycle time.

FLATS2 has ten pipeline stages, and each stages are executed in 50 ns. Each instruction takes two successive slot in pipeline and the entire pipeline is shared by two instruction/data stream. Consequently, each instruction of a single virtual processor is executed in 200 ns. In the case of a conventional pipelined processor, branch dependences cause delays by a conditional branch such as the exceptional condition of BL addressing mode. Because each pipeline stages are shared by two virtual processor in FLATS2, a detection of branch targets can be determined by the next instruction fetch. In cyclic pipeline architecture, the total throughput can be achieved, though each program can take fixed of the total (a half of FLATS2).

Without cyclic pipeline architecture, BL addressing mode would be implemented with two instructions; The first instruction checks the pointers against the specified BL pair and jumps according to the result of the range checking, and the second instruction accesses the memory. With a "squashed delayed branch" in MIPS-X[2], these two instructions can be overlapped. The branch condition is calculated while the next instruction is fetched, and the effect of both instructions is canceled if the branch does occur.

The similar range checking instruction is proposed in [6] to support the range check of data values.

6. Conclusion

In this paper, we described the architectural support of the integration of memory addressing and range checking in FLATS2, and how it supports the efficient execution of Common Lisp program. It provides a variety of checking required at run-time, such as type checking, generic arithmetic, array-bounds checking, and the checking on the number the argument and multiple return value of functions. We can reduce the cost of run-time checking by BL addressing mode and address tag as by tagged architectures.

FLATS2 provides the large number of registers to check many types efficiently, because BL addressing mode requires the BL pair in registers to be checked. The compiler can optimize the register allocation for BL pairs to make use of the BL addressing modes.

In cyclic pipeline architecture of FLATS2, we implement BL addressing modes in one instruction cycle. In conventional computers, it would be implemented with two instructions.

As discussed in [4], incorporating tag architecture into a general-purpose computer might impose high-level language features that are essentially at odds with the computational model of statically typed languages. Our architectural support, however, is so primitive mechanism that it can be used also for array-bounds checking in FORTRAN, and the detection of illegal pointer usage in C [10].

FLATS2 is currently under construction. Its Lisp system is being developed in the instruction level simulator.

ACKNOWLEDGEMENTS

We are grateful to Norihiro Fukazawa, Paul Spee and the members of FLATS2 project for helpful discussions.

† If each type segment were allocated to make higher bits of the address indicate the type like the tag, we could do the same operation as the full tagged architecture.

REFERENCES

[1] Bosshart,P., Hewes,C., Chang, M., and Chau, K. "A553K-Transistor LISP Processor Chip". *Digest 1987 International Solid-State Circuits Conference*, IEEE, New York, February 1987.

[2] Chow,P.,and Horowitz,M. "Architectural Tradeoffs in the Design of MIPS-X". *Proceedings of the 14th Annual International Symposium on Computer Architecture*, ACM, June, 1987.

[3] Gabriel,R.P. *Computer Systems Series*. Volume: *Performance and evaluation of LISP systems*. The MIT Press, 1985.

[4] Gehringer, E.F. and Keedy, J.L. "Tagged Architecture: How Compelling Are its Advantage?". *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ACM, June, 1985.

[5] Goto, E., Soma, T., Inada, N., Ida, T., Idesawa, M., Hiraki,. K., Suzuki, M., Shimizu, K., and Philipov, B. "Design of a Lisp Machine - FLATS2". *Conference Proceedings of 1982 ACM Sym. LISP and Functional Programming*, PittsPurgh, August 1982.

[6] Hill, D.D. "A hardware mechanism for supporting range checks". *Computer Architecture News*(ACM/SIGARCH), vol. 9, no. 4, pp. 15-21, June 1881.

[7] Ichikawa, S. "A study on the Cyclic Pipeline Computer: FLATS2". *MS Thesis*, February 1987, University of Tokyo.

[8] Moon,D.A. "Architecture of the Symbolics 3600". *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ACM, June 1985.

[9] Shimizu, K., Goto, E. and Ichikawa, S. "CPC(Cyclic Pipeline Computer) - An Architecture Suited for Josephson Pipelined-Memory Machines". *Proceedings of 4th Riken Symposium on Josephson Electronics*, Wako-shi, March 1987 (to appear in IEEE Transactions on Computers).

[10] Spee, P. "Dynamic Type and Range Checking in C using a Tagged Architecture". Research and Development Corp. of Japan, 1988.

[11] Steenkiste, P. and Hennessy, J. "LISP on a Reduce-Instruction-set-Processor". *Proceedings of the 1986 Conference on LISP and Functional Programming*, ACM, Boston, August 1986.

[12] Steenkiste, P. and Hennessy, J. "Tags and Type Checking in LISP: Hardware and Software Approaches". *Proceedings of 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, October 1987.

[13] Steele, Jr. G. *Common Lisp - The language*. Digital Equipment Corporation, 1984.

[14] Taylor, S.T., Hilfinger, P.N., Larus, J.R., Patterson, D.A. and Zorn, B.G. "Evaluation of the SPUR Lisp Architecture". *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ACM, June 1986.

[15] *VAX Architecture Handbook*, Digital Equipment Corp., 1981.

[16] Wilkes, M.V. "Hardware Support for Memory Protection: Capability Implementations". *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, Ca., March 1982.

APPENDIX. An example of list append function

```
; source program of list append
;
;(defun append (x y)
;   (cond ((endp x) y)
;         (t (cons (car x) (append (cdr x) y)))))
;
; gr0-gr32 : global register
; vr0-vr32 : local frame register
; gr0/gr1  : BL for passing argument
;                and multiple return values
; gr10/gr11: BL for cons cell segment.
; gr12/gr13: BL for free area of cons cell segment.
; gr30/gr31: BL for entire space.

_Lappend:
    movw sp,fp              ; no local variable
                            ; on the stack
    movw gr10@0,vr1         ; move x to vr1.
    movw gr10@4,vr0         ; move y to vr0.
    movw.j gr10:(vr1),vr2,L1
                            ; load car(x) to vr2.
                            ; jump L1 if success.
    jmp L3                  ; if failed,
                            ; return y (vr0)
L1:
    movw gr10:4(vr1),vr3
                            ; load cdr(x) to vr3
    movw vr0,>-4(sp)        ; push y
    movw vr3,>-4(sp)        ; push cdr(x)
    mkbl (sp),4(sp),gr0     ; make BL to pass
                            ; the argument
                            ; base = sp,
                            ; limit = sp+4
    call _Lappend           ; call function
    lea  8(sp),sp           ; reset stack
    movw gr0@0,vr0          ; get return to vr0
L2:
    alloc.j gr12,vr2,vr0,#8,vr0,L3
            ; vr0 = cons(vr2,vr0)
            ; v2 and vr0 are stored to double word
            ; specified by the base of gr12,
            ; and gr12 is incremented by 8.
            ; if base > limit,
            ; then address tag of gr12 is cleared.
            ; and go to L3.
    trap #GC_TRAP           ; already address tag
                            ; of gr12 is cleared
                            ; call GC
    jmp  L2                 ; again do cons

L3:
    movw vr0,>-4(sp)        ; push return value
    mkbl (sp),(sp),gr0      ; make BL for
                            ; return value on gr0
    movw fp,sp              ; reset stack
    ret                     ; return
```

Appendix B

Benchmark programs in Parallel FORTRAN

This appendix contains parallel versions of benchmark programs in FORTRAN.

Linpack from [Don79]:

```
c
c parallel version of Linpack
c
      double precision aa(200,200),a(201,200),b(200),x(200)
      double precision time(8,6),cray,ops,total,norma,normx
      double precision resid,residn,eps,epslon
      integer ipvt(200)
c global
      common /adata/aa,a,b,x
      common /idata/ipvt
      common /tdata/t1,t2
c
      lda = 201
      ldaa = 200
c
      n = 100
c
*$ barrier
      call matgen(a,lda,n,b,norma)

      t1 = COUNT()
      write(6,910) t1
 910  format('dgefa+:',e16.8)
      t1 = COUNT()
*$ end_barrier

      call dgefaP(a,lda,n,ipvt,info)

*$ barrier
      t2 = COUNT() - t1
      write(6,920) t2
 920  format('dgefa-:',e16.8)

      write(6,950) (ipvt(i), i =1,10)
 950  format('pvt:',10I5)
      t1 = COUNT()
      write(6,930) t1
 930  format('dgesl+:',e16.8)
      t1 = COUNT()
*$ end_barrier
c
      call dgeslP(a,lda,n,ipvt,b,0)
c
*$ barrier
      t2 = COUNT() - t1
      write(6,940) t2
 940  format('dgesl-:',e16.8)

c
c print result
c

*$ end_barrier
      end

      subroutine matgen(a,lda,n,b,norma)
      double precision a(lda,1),b(1),norma
c
      init = 1325
      norma = 0.0
      do 30 j = 1,n
         do 20 i = 1,n
            init = mod(3125*init,65536)
            a(i,j) = (init - 32768.0)/16384.0
            norma = dmax1(dabs(a(i,j)), norma)
   20    continue
   30 continue
      do 35 i = 1,n
          b(i) = 0.0
   35 continue
      do 50 j = 1,n
         do 40 i = 1,n
            b(i) = b(i) + a(i,j)
   40    continue
   50 continue
      return
      end
c
      subroutine dgefaP(a,lda,n,ipvt,info)
      integer lda,n,ipvt(1),info
      double precision a(lda,1)
c
c     dgefa factors a double precision matrix by gaussian elimination.
c
      double precision t
      integer idamaxP,j,k,kp1,l,nm1
c
c     gaussian elimination with partial pivoting
c
      info = 0
      nm1 = n - 1
      if (nm1 .lt. 1) go to 70
      do 60 k = 1, nm1
         kp1 = k + 1
c
c        find l = pivot index
c
         l = idamaxP(n-k+1,a(k,k),1) + k - 1
         ipvt(k) = l
c
c        zero pivot implies this column already triangularized
c
         if (a(l,k) .eq. 0.0d0) go to 40
*$ barrier
c
c           interchange if necessary
c
            if (l .eq. k) go to 10
               t = a(l,k)
               a(l,k) = a(k,k)
               a(k,k) = t
   10       continue
```

```
c
c          compute multipliers
c
*$    end_barrier
           t = -1.0d0/a(k,k)
           call dscalP(n-k,t,a(k+1,k),1)
c
c          row elimination with column indexing
c
*$ pre_sched_do
           do 30 j = kp1, n
              t = a(l,j)
              if (l .eq. k) go to 20
                 a(l,j) = a(k,j)
                 a(k,j) = t
   20         continue
              call daxpy(n-k,t,a(k+1,k),1,a(k+1,j),1)
   30      continue
*  pre_sched_do_end
        go to 50
   40   continue
           info = k
   50   continue
   60 continue
   70 continue
      ipvt(n) = n
      if (a(n,n) .eq. 0.0d0) info = n
      return
      end
c
      subroutine dgeslP(a,lda,n,ipvt,b,job)
      integer lda,n,ipvt(1),job
      double precision a(lda,1),b(1)
c
c     dgesl solves the double precision system
c     a * x = b  or  trans(a) * x = b
c     using the factors computed by dgeco or dgefa.
c
      double precision ddot,t
      integer k,kb,l,nm1
c
      nm1 = n - 1
      if (job .ne. 0) go to 50
c
c        job = 0 , solve  a * x = b
c        first solve  l*y = b
c
         if (nm1 .lt. 1) go to 30
         do 20 k = 1, nm1
            l = ipvt(k)
            t = b(l)
*$ barrier
            if (l .eq. k) go to 10
               b(l) = b(k)
               b(k) = t
   10       continue
```

```
*$ end_barrier
            call daxpyP(n-k,t,a(k+1,k),1,b(k+1),1)
*$ barrier
*$ end_barrier
   20     continue
   30     continue
c
c         now solve  u*x = y
c
          do 40 kb = 1, n
            k = n + 1 - kb
*$ barrier
            b(k) = b(k)/a(k,k)
*$ end_barrier
            t = -b(k)
            call daxpyP(k-1,t,a(1,k),1,b(1),1)
   40     continue
        go to 100
   50   continue
c
c         job = nonzero, solve  trans(a) * x = b
c         first solve  trans(u)*y = b
c
  100 continue
      return
      end
c
c sequential version
c
      subroutine daxpy(n,da,dx,incx,dy,incy)
c
c     constant times a vector plus a vector.
c
      double precision dx(1),dy(1),da
      integer i,incx,incy,ix,iy,m,mp1,n
c
      if(n.le.0)return
      if (da .eq. 0.0d0) return
      if(incx.eq.1.and.incy.eq.1)go to 20
c
c        code for unequal increments or equal increments
c          not equal to 1
c
      return
c
c        code for both increments equal to 1
c
   20 continue
      do 30 i = 1,n
        dy(i) = dy(i) + da*dx(i)
   30 continue
      return
      end
c
      subroutine daxpyP(n,da,dx,incx,dy,incy)
c
```

```
c     constant times a vector plus a vector.
c
      double precision dx(1),dy(1),da
      integer i,incx,incy,ix,iy,m,mp1,n
c
      if(n.le.0)return
      if (da .eq. 0.0d0) return
      if(incx.eq.1.and.incy.eq.1)go to 20
c
c        code for unequal increments or equal increments
c          not equal to 1
c
      return
c
c        code for both increments equal to 1
c
   20 continue
*$ pre_sched_do
      do 30 i = 1,n
        dy(i) = dy(i) + da*dx(i)
   30 continue
      return
      end
c
      subroutine  dscalP(n,da,dx,incx)
c
c     scales a vector by a constant.
c
      double precision da,dx(1)
      integer i,incx,m,mp1,n,nincx
c
      if(n.le.0)return
      if(incx.eq.1) go to 20
c
c        code for increment not equal to 1
c
      return
c
c        code for increment equal to 1
c
 20   continue
*$ pre_sched_do
      do 30 i = 1,n
         dx(i) = da*dx(i)
 30   continue
      return
      end
c
c parallel version, using cascade sum
c
      integer function idamaxP(n,dx,incx)
c
c     finds the index of element having max. dabsolute value.
c
      double precision dx(1),dmax
      integer i,incx,ix,n
      integer idamax
      common ddmax,idmax
c
      idamaxP = 0
      if( n .lt. 1 ) return
      idamaxP = 1
      if(n.eq.1) return
      if(incx.eq.1)go to 20
c
c        code for increment not equal to 1
c
      return
c
c        code for increment equal to 1
c
 20   continue
*$ barrier
      idmax = 1
      ddmax = dabs(dx(1))
*$ end_barrier
      dmax = ddmax
*$ pre_sched_do
      do 30 i = 2,n
         if(dabs(dx(i)).le.dmax) go to 30
         idamax = i
         dmax = dabs(dx(i))
   30 continue
*$ critical
      if(dmax .le. ddmax) goto 40
      idmax = idamax
      ddmax = dmax
 40   continue
*$ end_critical
c wait all
*$ barrier
*$ end_barrier
      idamaxP = idmax
      return
      end
```

FEM_BAND from [Mas86]:

```
*
* parallel version for BAND
*
C       PROGRAM EXFEM
c     N is the number of points.(NX+1)*(NY+1)
c     NE is the number of element.NX*NY*2
c     NB1 is the number of points on dirichlet bounary.(NX+1)*2
c     NB2 is the nubmer of points on natural bounary. NY*2
c     MB1 is the width of band matrix. (NX + 3)
c     NX=16,NY=16 case
      PARAMETER (NX=16, NY=16)
      PARAMETER (N=289, NE=512, MB1 = 19)
      PARAMETER (NB2=32, NB1=34)
      PARAMETER (NPX=2, NPY=2)
*     FEM DATA (global)
      DOUBLE PRECISION X(N),Y(N),P(N),Q(N),F(N)
      INTEGER KNC(3,NE),KS2(2,NB2)
      INTEGER IB(NB1)
      DOUBLE PRECISION BV(NB1)
      DOUBLE PRECISION ALPHA(2,NB2),BETA(2,NB2)
      DOUBLE PRECISION B(N),U(N)
      common /fdata/X,Y,P,Q,F,KNC,KS2,IB,BV,ALPHA,BETA,B,U
      common /idata/IERR
* SOLVER DATA
      DOUBLE PRECISION AR(MB1,N)
      common /adata/AR
      INTEGER ISYNC(N)
      common /sync/ISYNC
      DOUBLE PRECISION TM,COUNT
      common /time/TM
*
*$ barrier
c
      TM = COUNT()
      write(6,1000) TM
 1000 format('FEM(BAND,parallel)+',E16.8)
c
      CALL FMDATA(NX,NY,N,NE,X,Y,KNC,P,Q,F,NB1,IB,BV,NB2,KS2,ALPHA,BETA)
*$ end_barrier
      CALL FEM(N,NE,X,Y,KNC,P,Q,F,NB1,IB,BV,NB2,KS2,ALPHA,BETA,
     &      U,IERR,MB1,AR,B,ISYNC)
*$ barrier
c
      TM = COUNT()
      write(6,1001) TM
 1001 format('FEM-:',E16.8)
c
      IF(IERR .GT. 0) THEN
         WRITE(*,2011) IERR
 2011    FORMAT('THE AREA OF THE ',I5,'-TH ELEMENT IS ZERO OR NEGATIVE')
         STOP
      ENDIF
      WRITE(*,2001)
 2001 FORMAT('--- FEM ---'/' SOLUTION U(I)')
c     print data
      DO 10 I = NY+1,1,-NPY
```

```
          JS = (NX+1)*(I-1)+1
          JE = JS + NX
          WRITE(*,2002) (U(J),J=JS,JE,NPY)
 2002     FORMAT(1X,9F7.3)
 10    CONTINUE
*$ end_barrier
      END

* sequential
      SUBROUTINE FMDATA(NX,NY,N,NE,X,Y,KNC,P,Q,F,
     $      NB1,IB,BV,NB2,KS2,ALPHA,BETA)
      DOUBLE PRECISION X(N),Y(N),P(N),Q(N),F(N)
      INTEGER KNC(3,NE),KS2(2,NB2)
      INTEGER IB(NB1)
      DOUBLE PRECISION BV(NB1)
      DOUBLE PRECISION ALPHA(2,NB2),BETA(2,NB2)
*
*     setup element
*
      HX = 1.0D0/NX
      HY = 1.0D0/NY
      DO 10 JY = 0,NY
         DO 20 IX = 0,NX
            IXY = (NX+1)*JY+IX+1
            X(IXY) = HX*IX
            Y(IXY) = HY*JY
            P(IXY) = 1.0D0
            Q(IXY) = 0.0D0
            F(IXY) = -1.0D0
 20      CONTINUE
 10   CONTINUE
*
*     element data
*
      DO 30 JY = 1,NY
         KE1 = 2*NX*(JY-1)
         K1 = (NX+1)*(JY-1)+1
         DO 40 IX = 1,NX
            KEI = KE1+IX
            KI = K1+IX-1
            KNC(1,KEI) = KI
            KNC(2,KEI) = KI+1
            KNC(3,KEI) = KI+NX+2
            KEI = KEI+NX
            KNC(1,KEI) = KI
            KNC(2,KEI) = KI+NX+2
            KNC(3,KEI) = KI+NX+1
 40      CONTINUE
 30   CONTINUE
*
*     bounary data
*
      DO 50 L = 1,NY
         KS2(1,L) = (NX+1)*L
         KS2(2,L) = KS2(1,L)+NX+1
         KS2(1,L+NY) = (NX+1)*(L-1)+1
         KS2(2,L+NY) = KS2(1,L+NY)+NX+1
         ALPHA(1,L) = 0.0
         ALPHA(2,L) = 0.0
         BETA(1,L) = 0.0
         BETA(2,L) = 0.0
         ALPHA(1,L+NY) = 0.0
         ALPHA(2,L+NY) = 0.0
         BETA(1,L+NY) = 0.0
         BETA(2,L+NY) = 0.0
 50   CONTINUE
*
*     dirichlet bounary, 0 if y=0 , x if y=1
*
      DO 60 L = 1,NX+1
         IB(L) = L
         IB(L+NX+1) = (NX+1)*NY+L
         BV(L) = 0.0D0
         BV(L+NX+1) = (1.0D0/NX)*(L-1)
 60   CONTINUE
      RETURN
      END
*
* finite element method with band matrix solver (paralell)
*
      SUBROUTINE FEM(N,NE,X,Y,KNC,P,Q,F,NB1,IB,BV,NB2,KS2,ALPHA,BETA,
     &     U,IERR,MB1,AR,B,ISYNC)
      DOUBLE PRECISION X(N),Y(N),P(N),Q(N),F(N)
      INTEGER KNC(3,NE),KS2(2,NB2)
      INTEGER IB(NB1)
      DOUBLE PRECISION BV(NB1)
      DOUBLE PRECISION ALPHA(2,NB2),BETA(2,NB2)
      DOUBLE PRECISION B(N),U(N)
c
      DOUBLE PRECISION AR(MB1,N)
      INTEGER ISYNC(N)
      DOUBLE PRECISION TM,COUNT
      common /time/TM
*$barrier
c
      TM = COUNT()
      write(6,1000) TM
 1000 format('MATGEN+:',E16.8)
      TM = COUNT()
c
*$ end_barrier
      CALL FORM(N,NE,X,Y,KNC,P,Q,F,IERR,MB1,AR,B)
      CALL BOUND2(N,X,Y,NB2,KS2,ALPHA,BETA,MB1,AR,B)
      CALL BOUND1(N,NB1,IB,BV,MB1,AR,B)
*$ barrier
c
      TM = COUNT() - TM
      write(6,1001) TM
 1001 format('MATGEN-:',E16.8)
c
```

```
*$ end_barrier
      IF(IERR .GT. 0) RETURN
*$ barrier
      TM = COUNT()
      write(6,1002) TM
 1002 format('SOLV+:',E16.8)
      TM = COUNT()
*$ end_barrier
      CALL CHDCMP(MB1,N,MB1,AR,U,ISYNC)
      CALL CHSOLV(MB1,N,MB1,AR,B,U,ISYNC)
*$ barrier
c
      TM = COUNT() - TM
      write(6,1003) TM
 1003 format('SOLV-:',E16.8)
c
*$ end_barrier
      RETURN
      END
*
*     arrenge upper right part AU and diagonal AD of matrix A
*     in compact form for ICCG method and arrange right hand side vector B
*
      SUBROUTINE FORM(N,NE,X,Y,KNC,P,Q,F,IERR,MB1,AR,B)
      DOUBLE PRECISION X(N),Y(N),P(N),Q(N),F(N)
      INTEGER KNC(3,NE)
      DOUBLE PRECISION AR(MB1,N),B(N)
c local data
      DOUBLE PRECISION EA(3,3),EF(3)

*$ barrier
      IERR = 0
*$ end_barrier
*
*     clear AR
*
*$ pre_sched_do
      DO 530 I = 1,N
         B(I) = 0.0D0
         DO 540 J = 1,MB1
            AR(J,I) = 0.0D0
 540     CONTINUE
 530  CONTINUE
*
*$ pre_sched_do
      DO 10 L = 1,NE
*
*     compute element matix EA and element vector EF
*
         CALL ELMMAT(L,NE,N,X,Y,KNC,P,Q,F,EA,EF,IER)
*$ critical
         IF(IER .GT. 0) IERR = IER
*$ end_critical
         IF(IERR .GT. 0) RETURN
*
         DO 20 I = 1,3
```

```
          IROW = KNC(I,L)
*$ critical
          B(IROW) = B(IROW)+EF(I)
*$ end_critical
          DO 30 J = 1,3
            JCOL = KNC(J,L)
            IF(IROW .GE. JCOL) THEN
              JCOL = MB1 + JCOL - IROW
*$ critical
              AR(JCOL,IROW) = AR(JCOL,IROW) + EA(I,J)
*$ end_critical
            ENDIF
 30       CONTINUE
 20     CONTINUE
 10   CONTINUE
      RETURN
      END
*
* compute element matrix EA and element vector
*
      SUBROUTINE ELMMAT(L,NE,N,X,Y,KNC,P,Q,F,EA,EF,IERR)
      DOUBLE PRECISION X(N),Y(N),P(N),Q(N),F(N)
      INTEGER KNC(3,NE)
      DOUBLE PRECISION EA(3,3),EF(3),DXY(2,3)
      DOUBLE PRECISION S,S60,S15,S12
      I = KNC(1,L)
      J = KNC(2,L)
      K = KNC(3,L)
      DXY(1,1) = X(K) - X(J)
      DXY(1,2) = X(I) - X(K)
      DXY(1,3) = X(J) - X(I)
      DXY(2,1) = Y(J) - Y(K)
      DXY(2,2) = Y(K) - Y(I)
      DXY(2,3) = Y(I) - Y(J)
*
      S = 0.5D0*(DXY(1,3)*DXY(2,2)-DXY(1,2)*DXY(2,3))
*
      IF(S .LE. 0.0D0) THEN
        IERR = L
        RETURN
      ENDIF
*
      S60 = S/60
      QM = 2*S60*(Q(I)+Q(J)+Q(K))
      S15 = 4*S60
*
*     diagonal element
*
      EA(1,1) = QM+S15*Q(I)
      EA(2,2) = QM+S15*Q(J)
      EA(3,3) = QM+S15*Q(K)
      PM = (P(I)+P(J)+P(K))/(12*S)
*
*     non diagonal element
*
```

```
      DO 10 II = 1,3
         DO 20 JJ = 1,3
            IF(II .NE. JJ) EA(II,JJ) = QM - S60*Q(KNC(6-II-JJ,L))
            DO 530 KK = 1,2
               EA(II,JJ) = EA(II,JJ)+PM*DXY(KK,II)*DXY(KK,JJ)
 530        CONTINUE
 20      CONTINUE
 10   CONTINUE
      S12 = 5*S60
      EF(1) = S12*(2*F(I)+F(J)+F(K))
      EF(2) = S12*(F(I)+2*F(J)+F(K))
      EF(3) = S12*(F(I)+F(J)+2*F(K))
      RETURN
      END

*
* arrange natural boundary condition
*
      SUBROUTINE BOUND2(N,X,Y,NB2,KS2,ALPHA,BETA,MB1,AR,B)
      DOUBLE PRECISION X(N),Y(N)
      INTEGER KS2(2,NB2)
      DOUBLE PRECISION ALPHA(2,NB2),BETA(2,NB2)
      DOUBLE PRECISION EAB2(2,2),EFB2(2)
      DOUBLE PRECISION AR(MB1,N),B(N)

      IF(NB2 .LE. 0) RETURN
*$ pre_sched_do
      DO 10 L = 1, NB2
*     compute element matrix and element right hand side vector
         CALL SIDE2(L,NB2,N,X,Y,KS2,ALPHA,BETA,EAB2,EFB2)
         DO 20 I = 1,2
            IROW = KS2(I,L)
*$ critical
            B(IROW) = B(IROW) - EFB2(I)
*$ end_critical
            DO 30 J = 1,2
               JCOL = KS2(J,L)
               IF(IROW .GE. JCOL) THEN
                  JCOL = MB1 + JCOL - IROW
*$ critical
                  AR(JCOL,IROW) = AR(JCOL,IROW) + EAB2(I,J)
*$ end_critical
               ENDIF
 30         CONTINUE
 20      CONTINUE
 10   CONTINUE
      RETURN
      END

      SUBROUTINE SIDE2(L,NB2,N,X,Y,KS2,ALPHA,BETA,EAB2,EFB2)
      DOUBLE PRECISION X(N),Y(N)
      INTEGER KS2(2,NB2)
      DOUBLE PRECISION ALPHA(2,NB2),BETA(2,NB2)
      DOUBLE PRECISION EAB2(2,2),EFB2(2)
      DOUBLE PRECISION SL,SL12,SL6
      I = KS2(1,L)
      J = KS2(2,L)
      IF(I .LE. 0 .OR. J .LE. 0) RETURN
      SL = SQRT((X(I)-X(J))**2+(Y(I)-Y(J))**2)
      SL12 = SL/12
      EAB2(1,1) = SL12*(3*ALPHA(1,L)+ALPHA(2,L))
      EAB2(1,2) = SL12*(ALPHA(1,L)+ALPHA(2,L))
      EAB2(2,1) = EAB2(1,2)
      EAB2(2,2) = SL12*(ALPHA(1,L)+3*ALPHA(2,L))
      SL6 = 2*SL12
      EFB2(1) = SL6*(2*BETA(1,L)+BETA(2,L))
      EFB2(2) = SL6*(BETA(1,L)+2*BETA(2,L))
      RETURN
      END
*
*     arragne dirichlet boundary condition
*
      SUBROUTINE BOUND1(N,NB1,IB,BV,MB1,AR,B)
      INTEGER IB(NB1)
      DOUBLE PRECISION BV(NB1)
      DOUBLE PRECISION  AR(MB1,N),B(N)

      IF(NB1 .LE. 0) RETURN
      DO 10 L = 1,NB1
         K = IB(L)
*$ pre_sched_do
         DO 20 IROW = MAX(1,K-MB1+1), K-1
            JCOL = MB1 + IROW - K
            B(IROW) = B(IROW) - AR(JCOL,K)*BV(L)
            AR(JCOL,K) = 0.0
 20      CONTINUE
c
*$ barrier
         B(K) = BV(L)
         AR(MB1,K) = 1.0D0
*$ end_barrier
*$ pre_sched_do
         DO 30 IROW = K+1,MIN(K+MB1-1,N)
            JCOL = MB1 + K - IROW
            B(IROW) = B(IROW)-AR(JCOL,IROW)*BV(L)
            AR(JCOL,IROW) = 0.0
 30      CONTINUE
 10   CONTINUE
      RETURN
      END
*
*     Modified Cholesy decomposition of symmetric band matrix
*
      SUBROUTINE CHDCMP(MDIM,N,M1,AR,W,ISYNC)
      DOUBLE PRECISION AR(MDIM,N),W(N)
      DOUBLE PRECISION T,TT
      INTEGER ISYNC(N)
      common /global/TT
*
      M = M1-1
      DO 10 K=1,N
```

```
        I0 = MAX(1,K-M)
c clear sync array
*$ pre_sched_do
*%      DO 21 I = I0,K-1
*%      ISYNC(I) = 0
*%21  CONTINUE
*$ barrier
*$ end_barrier
*$ pre_sched_do
        DO 20 I = I0,K-1
          T = AR(M1+I-K,K)
          DO 530 J = I0,I-1
*%      CALL SYNC(ISYNC(J))
            T = T - W(J)*AR(M1+J-I,I)
 530      CONTINUE
          AR(M1+I-K,K) = T*AR(M1,I)
          W(I) = T
c sync
*%      ISYNC(I) = 1
 20     CONTINUE
*$ barrier
          TT = AR(M1,K)
*$ end_barrier
          T = 0
*$ pre_sched_do
          DO 540 I = I0,K-1
             T = T - W(I)*AR(M1+I-K,K)
 540      CONTINUE
*$ critical
          TT = TT + T
*$ end_critical
*$ barrier
          AR(M1,K) = 1.0D0/TT
*$ end_barrier
 10   CONTINUE
      RETURN
      END

      SUBROUTINE CHSOLV(MDIM,N,M1,AR,B,X,ISYNC)
      DOUBLE PRECISION AR(MDIM,N),B(N),X(N)
      INTEGER ISYNC(N)
      DOUBLE PRECISION T
c
*$ pre_sched_do
*%      DO 100 K = 1,N
*%      ISYNC(K) = 0
*%100 CONTINUE
*$ barrier
*$ end_barrier
      M = M1-1
*$ pre_sched_do
      DO 10 K = 1,N
        T = B(K)
        DO 520 I = MAX(1,K-M),K-1
*%    CALL SYNC(ISYNC(I))
          T = T - AR(M1+I-K,K)*X(I)
 520      CONTINUE
          X(K) = T
*%     ISYNC(K) = 1
 10    CONTINUE
*
*$ barrier
*$ end_barrier
*$ pre_sched_do
*%     DO 101 K = 1,N
*%     ISYNC(K) = 0
*%101 CONTINUE
*$ barrier
*$ end_barrier
*$ pre_sched_do
       DO 30 K = N,1,-1
          T = X(K)*AR(M1,K)
          DO 540 I = K+1,MIN(N,K+M)
*%     CALL SYNC(ISYNC(I))
             T = T - AR(M1+K-I,I)*X(I)
 540      CONTINUE
          X(K) = T
*%     ISYNC(K) = 1
 30    CONTINUE
       RETURN
       END
```

FEM_ICCG from [Mas86]:

```
*
* parallel verion for ICCG
*
c       PROGRAM EXFEM
c      N is the number of points.(NX+1)*(NY+1)
c      NE is the number of element.NX*NY*2
c      NB1 is the number of points on dirichlet bounary.(NX+1)*2
c      NB2 is the nubmer of points on natural bounary. NY*2
c      NX=16,NY=16 case
       PARAMETER (NX=16, NY=16)
       PARAMETER (N=289, NE=512, NR = 3)
       PARAMETER (NB2=32, NB1=34)
       PARAMETER (NPX=2, NPY=2)
*      FEM DATA
       DOUBLE PRECISION X(N),Y(N),P(N),Q(N),F(N)
       INTEGER KNC(3,NE),KS2(2,NB2)
       INTEGER IB(NB1)
       DOUBLE PRECISION BV(NB1)
       DOUBLE PRECISION ALPHA(2,NB2),BETA(2,NB2)
       DOUBLE PRECISION B(N),U(N)
       common /fdata/X,Y,P,Q,F,KNC,KS2,IB,BV,ALPHA,BETA,B,U,IERR
* ICCG DATA
       INTEGER NZU(N),NZL(N),IU(NR,N),IL(NR,N)
       DOUBLE PRECISION AU(NR,N),AL(NR,N),AD(N)
       DOUBLE PRECISION UU(NR,N),UL(NR,n),UD(N)
*
       DOUBLE PRECISION R(N),PA(N),RU(N),AP(N)
       DOUBLE PRECISION EPS
       common /adata/NZU,NZL,IU,IL,AU,AL,AD,UU,UL,UD,R,PA,RU,AP,EPS
       INTEGER ISYNC(N)
       common /sync/ISYNC
       DOUBLE PRECISION COUNT,TM
       common /time/TM
*
*$ barrier
c
       TM = COUNT()
       write(6,1000) TM
 1000 format('FEM(ICCG,parallel)+:',E16.8)
c
       EPS = 1.0E-5
       IERR = 0
       CALL FMDATA(NX,NY,N,NE,X,Y,KNC,P,Q,F,NB1,IB,BV,NB2,KS2,ALPHA,BETA)
*$ end_barrier
       CALL FEM(N,NE,X,Y,KNC,P,Q,F,NB1,IB,BV,NB2,KS2,ALPHA,BETA,
      &      EPS,U,IERR,NR,NZU,NZL,IU,IL,AU,AL,AD,B,
      &      UU,UL,UD,R,PA,RU,AP,ISYNC)
*$ barrier
c
       TM = COUNT()
       write(6,1001) TM
 1001 format('FEM(ICCG,parallel)-:',E16.8)
c
       IF(IERR .GT. 0) THEN
          WRITE(*,2011) IERR
 2011     FORMAT('THE AREA OF THE ',I5,'-TH ELEMENT IS ZERO OR NEGATIVE')
```

```
          STOP
       ENDIF
       WRITE(*,2001)
 2001  FORMAT('--- FEM ---'/' SOLUTION U(I)')
c      print data
       DO 10 I = NY+1,1,-NPY
          JS = (NX+1)*(I-1)+1
          JE = JS + NX
          WRITE(*,2002) (U(J),J=JS,JE,NPY)
 2002     FORMAT(1X,9F7.3)
 10    CONTINUE
*$ end_barrier
       END

* sequential
       SUBROUTINE FMDATA(NX,NY,N,NE,X,Y,KNC,P,Q,F,
     $     NB1,IB,BV,NB2,KS2,ALPHA,BETA)
       DOUBLE PRECISION X(N),Y(N),P(N),Q(N),F(N)
       INTEGER KNC(3,NE),KS2(2,NB2)
       INTEGER IB(NB1)
       DOUBLE PRECISION BV(NB1)
       DOUBLE PRECISION ALPHA(2,NB2),BETA(2,NB2)
*
*      setup element
*
       HX = 1.0/NX
       HY = 1.0/NY
       DO 10 JY = 0,NY
          DO 20 IX = 0,NX
             IXY = (NX+1)*JY+IX+1
             X(IXY) = HX*IX
             Y(IXY) = HY*JY
             P(IXY) = 1.0
             Q(IXY) = 0.0
             F(IXY) = -1.0
 20       CONTINUE
 10    CONTINUE
*
*      element data
*
       DO 30 JY = 1,NY
          KE1 = 2*NX*(JY-1)
          K1 = (NX+1)*(JY-1)+1
          DO 40 IX = 1,NX
             KEI = KE1+IX
             KI = K1+IX-1
             KNC(1,KEI) = KI
             KNC(2,KEI) = KI+1
             KNC(3,KEI) = KI+NX+2
             KEI = KEI+NX
             KNC(1,KEI) = KI
             KNC(2,KEI) = KI+NX+2
             KNC(3,KEI) = KI+NX+1
 40       CONTINUE
 30    CONTINUE
*
```

```
*     bounary data
*
      DO 50 L = 1,NY
         KS2(1,L) = (NX+1)*L
         KS2(2,L) = KS2(1,L)+NX+1
         KS2(1,L+NY) = (NX+1)*(L-1)+1
         KS2(2,L+NY) = KS2(1,L+NY)+NX+1
         ALPHA(1,L) = 0.0
         ALPHA(2,L) = 0.0
         BETA(1,L) = 0.0
         BETA(2,L) = 0.0
         ALPHA(1,L+NY) = 0.0
         ALPHA(2,L+NY) = 0.0
         BETA(1,L+NY) = 0.0
         BETA(2,L+NY) = 0.0
 50   CONTINUE
*
*     dirichlet bounary, 0 if y=0 , x if y=1
*
      DO 60 L = 1,NX+1
         IB(L) = L
         IB(L+NX+1) = (NX+1)*NY+L
         BV(L) = 0.0
         BV(L+NX+1) = (1.0/NX)*(L-1)
 60   CONTINUE
      RETURN
      END
*
* finite element method
*
      SUBROUTINE FEM(N,NE,X,Y,KNC,P,Q,F,NB1,IB,BV,NB2,KS2,ALPHA,BETA,
     &     EPS,U,IERR,NR,NZU,NZL,IU,IL,AU,AL,AD,B,
     &     UU,UL,UD,R,PA,RU,AP,ISYNC)

      DOUBLE PRECISION X(N),Y(N),P(N),Q(N),F(N)
      INTEGER KNC(3,NE),KS2(2,NB2)
      INTEGER IB(NB1)
      DOUBLE PRECISION BV(NB1)
      DOUBLE PRECISION ALPHA(2,NB2),BETA(2,NB2)
      DOUBLE PRECISION B(N),U(N)

      INTEGER NZU(N),NZL(N),IU(NR,N),IL(NR,N)
      DOUBLE PRECISION AU(NR,N),AL(NR,N),AD(N)
      DOUBLE PRECISION UU(NR,N),UL(NR,N),UD(N)
      DOUBLE PRECISION R(N),PA(N),RU(N),AP(N)
      DOUBLE PRECISION EPS
      INTEGER ISYNC(N)
      DOUBLE PRECISION COUNT,TM
      common /time/TM
*$ barrier
c
      TM = COUNT()
      write(6,1000) TM
 1000 format('MATGEN+:',E16.8)
      TM = COUNT()
c
```

```
*$ end_barrier
      CALL FORM(N,NE,X,Y,KNC,P,Q,F,IERR,NR,NZU,IU,AU,AD,B)
      IF(IERR .GT. 0) RETURN
      CALL BOUND2(N,X,Y,NB2,KS2,ALPHA,BETA,NR,NZU,IU,AU,AD,B)
      CALL BOUND1(N,NB1,IB,BV,NR,NZU,IU,AU,AD,B)
*$ barrier
c
      TM = COUNT() - TM
      write(6,1001) TM
 1001 format('MATGEN-:',E16.8)
c
      TM = COUNT()
      write(6,1002) TM
 1002 format('SOLV+:',E16.8)
      TM = COUNT()
c
*$ end_barrier
      CALL ICCG(N,NR,NZU,NZL,IU,IL,AU,AL,AD,B,
     &     EPS,U,UU,UL,UD,R,PA,RU,AP,ISYNC)
*$ barrier
c
      TM = COUNT() - TM
      write(6,1003) TM
 1003 format('SOLV-:',E16.8)
c
*$ end_barrier
      RETURN
      END

*
*     arrenge upper right part AU and diagonal AD of matrix A
*     in compact form for ICCG method and arrange right hand side vector B
*
      SUBROUTINE FORM(N,NE,X,Y,KNC,P,Q,F,IERR,NR,NZU,IU,AU,AD,B)
      DOUBLE PRECISION X(N),Y(N),P(N),Q(N),F(N)
      INTEGER KNC(3,NE)
      INTEGER NZU(N),IU(NR,N)
      DOUBLE PRECISION AU(NR,N),AD(N),B(N)
      DOUBLE PRECISION EA(3,3),EF(3)

*$ barrier
      IERR = 0
*$ end_barrier
*
*     clear nzu,ad,b,iu,au
*
*$ pre_sched_do
      DO 530 I = 1,N
        NZU(I) = 0
        AD(I) = 0.0
        B(I) = 0.0
        DO 540 J = 1,NR
          IU(J,I) = 0
          AU(J,I) = 0.0
 540    CONTINUE
 530  CONTINUE
```

```
*
*$ pre_sched_do
      DO 10 L = 1,NE
*
*     compute element matix EA and element vector EF
*
         CALL ELMMAT(L,NE,N,X,Y,KNC,P,Q,F,EA,EF,IERR)
         IF(IERR .GT. 0) RETURN
*
         DO 20 I = 1,3
            IROW = KNC(I,L)
*$ critical
            B(IROW) = B(IROW)+EF(I)
*$ end_critical
            DO 30 J = 1,3
               JCOL = KNC(J,L)
               IF(IROW .LT. JCOL) THEN
                  CALL GLBMAT(IROW,JCOL,
     &                  N,NR,NZU,IU,AU,EA(I,J))
               ELSEIF (IROW .EQ. JCOL) THEN
*$ critical
                  AD(IROW) = AD(IROW)+EA(I,J)
*$ end_critical
               ENDIF
 30         CONTINUE
 20      CONTINUE
 10   CONTINUE
      RETURN
      END
*
* compute element matrix EA and element vector
*
      SUBROUTINE ELMMAT(L,NE,N,X,Y,KNC,P,Q,F,EA,EF,IERR)
      DOUBLE PRECISION X(N),Y(N),P(N),Q(N),F(N)
      INTEGER KNC(3,NE)
      DOUBLE PRECISION EA(3,3),EF(3),DXY(2,3)
      DOUBLE PRECISION S,S60,S15,S12

      I = KNC(1,L)
      J = KNC(2,L)
      K = KNC(3,L)
      DXY(1,1) = X(K) - X(J)
      DXY(1,2) = X(I) - X(K)
      DXY(1,3) = X(J) - X(I)
      DXY(2,1) = Y(J) - Y(K)
      DXY(2,2) = Y(K) - Y(I)
      DXY(2,3) = Y(I) - Y(J)
*
      S = 0.5*(DXY(1,3)*DXY(2,2)-DXY(1,2)*DXY(2,3))
*
      IF(S .LE. 0.0) THEN
         IERR = L
         RETURN
      ENDIF
*
      S60 = S/60
      QM = 2*S60*(Q(I)+Q(J)+Q(K))
      S15 = 4*S60
*
*     diagonal element
*
      EA(1,1) = QM+S15*Q(I)
      EA(2,2) = QM+S15*Q(J)
      EA(3,3) = QM+S15*Q(K)
      PM = (P(I)+P(J)+P(K))/(12*S)
*
*     non diagonal element
*
      DO 10 II = 1,3
         DO 20 JJ = 1,3
            IF(II .NE. JJ) EA(II,JJ) = QM - S60*Q(KNC(6-II-JJ,L))
            DO 530 KK = 1,2
               EA(II,JJ) = EA(II,JJ)+PM*DXY(KK,II)*DXY(KK,JJ)
 530        CONTINUE
 20      CONTINUE
 10   CONTINUE
      S12 = 5*S60
      EF(1) = S12*(2*F(I)+F(J)+F(K))
      EF(2) = S12*(F(I)+2*F(J)+F(K))
      EF(3) = S12*(F(I)+F(J)+2*F(K))
      RETURN
      END

*
*     arrange off diagnal elements of global matrix
*
      SUBROUTINE GLBMAT(IROW,JCOL,N,NR,NZU,IU,AU,EAV)
      INTEGER NZU(N),IU(NR,N)
      DOUBLE PRECISION AU(NR,N)
      DOUBLE PRECISION EAV

*$ critical
      DO 40 NU = 1,NZU(IROW)
         IF(JCOL .LT. IU(NU,IROW)) THEN
            DO 50 NV = NZU(IROW),NU,-1
               IU(NV+1,IROW) = IU(NV,IROW)
               AU(NV+1,IROW) = AU(NV,IROW)
 50         CONTINUE
            NZU(IROW) = NZU(IROW)+1
            IU(NU,IROW) = JCOL
            AU(NU,IROW) = EAV
            GOTO 60
         ELSE IF(JCOL .EQ. IU(NU,IROW)) THEN
            AU(NU,IROW) = AU(NU,IROW)+EAV
            GOTO 60
         ENDIF
 40   CONTINUE
      NZU(IROW) = NZU(IROW)+1
      IU(NZU(IROW),IROW) = JCOL
      AU(NZU(IROW),IROW) = EAV
 60   CONTINUE
*$ end_critical
```

```
      RETURN
      END
*
* arrange natural boundary condition
*
      SUBROUTINE BOUND2(N,X,Y,NB2,KS2,ALPHA,BETA,NR,NZU,IU,AU,AD,B)
      DOUBLE PRECISION X(N),Y(N)
      INTEGER KS2(2,NB2)
      DOUBLE PRECISION ALPHA(2,NB2),BETA(2,NB2)
      DOUBLE PRECISION EAB2(2,2),EFB2(2)
      INTEGER NZU(N),IU(NR,N)
      DOUBLE PRECISION AU(NR,N),AD(N),B(N)
      IF(NB2 .LE. 0) RETURN
*$ self_sched_do
      DO 10 L = 1, NB2
*     compute element matrix and element right hand side vector
         CALL SIDE2(L,NB2,N,X,Y,KS2,ALPHA,BETA,EAB2,EFB2)
         DO 20 I = 1,2
            IROW = KS2(I,L)
*$ critical
            B(IROW) = B(IROW) - EFB2(I)
*$ end_critical
            DO 30 J = 1,2
               JCOL = KS2(J,L)
               IF(IROW .LT. JCOL) THEN
                  DO 540 NU = 1,NZU(IROW)
                     IF(JCOL .EQ. IU(NU,IROW)) THEN
*$ critical
                        AU(NU,IROW) = AU(NU,IROW) + EAB2(I,J)
*$ end_critical
                        GOTO 41
                     ENDIF
 540              CONTINUE
 41               CONTINUE
               ELSE IF(IROW .EQ. JCOL) THEN
*$ critical
                  AD(IROW) = AD(IROW)+EAB2(I,J)
*$ end_critical
               ENDIF
 30         CONTINUE
 20      CONTINUE
 10   CONTINUE
      RETURN
      END

      SUBROUTINE SIDE2(L,NB2,N,X,Y,KS2,ALPHA,BETA,EAB2,EFB2)
      DOUBLE PRECISION X(N),Y(N)
      INTEGER KS2(2,NB2)
      DOUBLE PRECISION ALPHA(2,NB2),BETA(2,NB2)
      DOUBLE PRECISION EAB2(2,2),EFB2(2)
      DOUBLE PRECISION SL,SL12,SL6

      I = KS2(1,L)
      J = KS2(2,L)
      IF(I .LE. 0 .OR. J .LE. 0) RETURN
      SL = SQRT((X(I)-X(J))**2+(Y(I)-Y(J))**2)
      SL12 = SL/12
      EAB2(1,1) = SL12*(3*ALPHA(1,L)+ALPHA(2,L))
      EAB2(1,2) = SL12*(ALPHA(1,L)+ALPHA(2,L))
      EAB2(2,1) = EAB2(1,2)
      EAB2(2,2) = SL12*(ALPHA(1,L)+3*ALPHA(2,L))
      SL6 = 2*SL12
      EFB2(1) = SL6*(2*BETA(1,L)+BETA(2,L))
      EFB2(2) = SL6*(BETA(1,L)+2*BETA(2,L))
      RETURN
      END
*
*     arragne dirichlet boundary condition
*
      SUBROUTINE BOUND1(N,NB1,IB,BV,NR,NZU,IU,AU,AD,B)
      INTEGER IB(NB1)
      DOUBLE PRECISION BV(NB1)
      INTEGER NZU(N),IU(NR,N)
      DOUBLE PRECISION  AU(NR,N),AD(N),B(N)

      IF(NB1 .LE. 0) RETURN
      DO 10 L = 1,NB1
      K = IB(L)
*$ pre_sched_do
        DO 20 I = 1, K-1
          NZUV = NZU(I)
          DO 30 NU = 1, NZUV
            IF(IU(NU,I) .EQ. K) THEN
              B(I) = B(I) - AU(NU,I)*BV(L)
*     shift element
              DO 40 MU = NU,NZU(I)-1
                AU(MU,I) = AU(MU+1,I)
                IU(MU,I) = IU(MU+1,I)
40            CONTINUE
              AU(NZU(I),I) = 0.0
              IU(NZU(I),I) = 0
              NZU(I) = NZU(I)-1
              GOTO 20
            ENDIF
 30       CONTINUE
 20     CONTINUE

*$ barrier
        B(K) = BV(L)
        AD(K) = 1.0
*$ end_barrier
*$ pre_sched_do
        DO 550 NU = 1,NZU(K)
          B(IU(NU,K)) = B(IU(NU,K))-AU(NU,K)*BV(L)
 550    CONTINUE
        DO 560 NU = 1, NZU(K)
          AU(NU,K) = 0.0
          IU(NU,K) = 0
 560    CONTINUE
        NZU(K) = 0
 10   CONTINUE
      RETURN
      END
c
c
      SUBROUTINE ICCG(N,NR,NZU,NZL,IU,IL,AU,AL,AD,B,EPS,X,
     &     UU,UL,UD,R,P,RU,AP,ISYNC)
      INTEGER NZU(N),NZL(N),IU(NR,N),IL(NR,N)
      DOUBLE PRECISION AU(NR,N),AL(NR,N),AD(N)
      DOUBLE PRECISION UU(NR,N),UL(NR,N),UD(N)
      DOUBLE PRECISION B(N),X(N)
      DOUBLE PRECISION R(N),P(N),RU(N),AP(N)
      DOUBLE PRECISION EPS
      INTEGER ISYNC(N)

      CALL CRCONV(N,NR,NZU,NZL,IU,IL,AU,AL)
      CALL ICDCMP(N,NR,NZL,IL,AL,AD,UL,UD,ISYNC)
      CALL CRCONV(N,NR,NZL,NZU,IL,IU,UL,UU)
      CALL CG(N,NR,NZU,NZL,IU,IL,AU,AL,AD,B,
     $     EPS,X,UU,UL,UD,R,P,RU,AP,ISYNC)
      RETURN
      END
c
      SUBROUTINE CRCONV(N,NR,NZ1,NZ2,I1,I2,A1,A2)
      INTEGER NZ1(N),NZ2(N),I1(NR,N),I2(NR,N)
      DOUBLE PRECISION A1(NR,N),A2(NR,N)
*$ pre_sched_do
      DO 540 I = 1,N
         NZ2(I) = 0
         DO 550 J = 1,NR
            I2(J,I) = 0
            A2(J,I) = 0.0
 550     CONTINUE
 540  CONTINUE
c
*$ barrier
*$ end_barrier
      DO 10 I = 1,N
*$ pre_sched_do
         DO 520 NU= 1, NZ1(I)
            NZ2(I1(NU,I)) = NZ2(I1(NU,I))+1
            I2(NZ2(I1(NU,I)),I1(NU,I)) = I
            A2(NZ2(I1(NU,I)),I1(NU,I)) = A1(NU,I)
 520     CONTINUE
*$ barrier
*$ end_barrier
 10   CONTINUE
      RETURN
      END

      SUBROUTINE ICDCMP(N,NR,NZL,IL,AL,AD,UL,UD,ISYNC)
      INTEGER NZL(N),IL(NR,N)
      DOUBLE PRECISION AL(NR,N),AD(N),UL(NR,N),UD(N)
      DOUBLE PRECISION T
      INTEGER ISYNC(N)
      EPS = 1.0E-6
c clear sync vector
*$ pre_sched_do
*%    DO 101 K = 1,N
*%    ISYNC(K) = 0
*%101 CONTINUE
*$ barrier
*$ end_barrier
*$ self_sched_do
      DO 10 K = 1,N
         DO 20 MU = 1,NZL(K)
            I = IL(MU,K)
            MUI = 1
            MUK = 1
            UL(MU,K) = AL(MU,K)
 1          CONTINUE
            IF(IL(MUI,I) .GT. IL(MUK,K)) THEN
               MUK = MUK+1
            ELSE IF(IL(MUI,I) .LT. IL(MUK,K)) THEN
               MUI = MUI+1
            ELSE
c sync
*%    CALL SYNC(ISYNC(IL(MUK,K)))
*%    CALL SYNC(ISYNC(I))
               UL(MU,K) = UL(MU,K) - UD(IL(MUK,K))*UL(MUI,I)*UL(MUK,K)
               MUI = MUI+1
               MUK = MUK+1
            ENDIF
            IF((MUI .LE. NZL(I)) .AND. (MUK .LE. NZL(K))) GOTO 1
 20      CONTINUE
         T = AD(K)
         DO 530 MU = 1,NZL(K)
            T = T - UD(IL(MU,K))*UL(MU,K)**2
 530     CONTINUE
         IF(ABS(T) .LE. EPS) T = EPS
         UD(K) = 1/T
c sync
*%    ISYNC(K) = 1
 10   CONTINUE
      RETURN
      END

      SUBROUTINE CG(N,NR,NZU,NZL,IU,IL,AU,AL,AD,B,
     $     EPS,X,UU,UL,UD,R,P,RU,AP,ISYNC)
      INTEGER NZU(N),NZL(N),IU(NR,N),IL(NR,N)
      DOUBLE PRECISION AU(NR,N),AL(NR,N),AD(N)
      DOUBLE PRECISION UU(NR,N),UL(NR,N),UD(N)
      DOUBLE PRECISION B(N),X(N)
      DOUBLEPRECISION R(N),P(N),RU(N),AP(N)
      DOUBLE PRECISION RUR0,RUR1,PAP,RES2
      DOUBLE PRECISION BETA,ALPHA,EPS,EPS2,B2
      common /g1/RUR0,RUR1,PAP,RES2,ALPHA,BETA,B2
      INTEGER ISYNC(N)
      DOUBLE PRECISION T

      KEND = N+10
      EPS2 = EPS**2
      CALL PRODMV(N,NR,NZU,NZL,IU,IL,AU,AL,AD,X,AP)
*$ pre_sched_do
      DO 510 I = 1,N
         R(I) = B(I) - AP(I)
 510  CONTINUE
      CALL ICSOLV(N,NR,NZU,NZL,IU,IL,UU,UL,UD,R,RU,ISYNC)
*$ barrier
      RUR0 = 0.0D0
*$ end_barrier
      T = 0.0D0
*$ pre_sched_do
      DO 520 I = 1,N
         P(I) = RU(I)
c         RUR0 = RUR0+R(I)*RU(I)
         T = T+R(I)*RU(I)
 520  CONTINUE
*$ critical
      RUR0 = RUR0+T
*$ end_critical
*$ barrier
      B2 = 0.0D0
*$ end_barrier
      T = 0.0D0
*$ pre_sched_do
      DO 530 I = 1,N
c        B2 = B2+B(I)**2
         T = T + B(I)**2
 530  CONTINUE
*$ critical
      B2 = B2 + T
*$ end_critical
      DO 10 K = 1, KEND
         CALL PRODMV(N,NR,NZU,NZL,IU,IL,AU,AL,AD,P,AP)
*$ barrier
         PAP = 0.0D0
*$ end_barrier
         T = 0.0D0
*$ pre_sched_do
         DO 540 I = 1,N
c           PAP = PAP + P(I)*AP(I)
            T = T + P(I)*AP(I)
 540     CONTINUE
*$ critical
         PAP = PAP + T
*$ end_critical
*$ barrier
         ALPHA = RUR0/PAP
         RES2 = 0.0D0
*$ end_barrier
         T = 0.0D0
*$ pre_sched_do
         DO 550 I = 1,N
            X(I) = X(I)+ALPHA*P(I)
            R(I) = R(I)-ALPHA*AP(I)
c           RES2 = RES2+R(I)**2
```

```
            T = T+R(I)**2
 550     CONTINUE
*$ critical
         RES2 = RES2 + T
*$ end_critical
*$ barrier
*$ end_barrier
         IF(RES2/B2 .LE. EPS2) THEN
            RETURN
         ENDIF
         CALL ICSOLV(N,NR,NZU,NZL,IU,IL,UU,UL,UD,R,RU,ISYNC)
*$ barrier
         RUR1 = 0.0D0
*$ end_barrier
         T = 0.0D0
*$ pre_sched_do
         DO 560 I = 1,N
c           RUR1 = RUR1 + R(I)*RU(I)
            T = T + R(I)*RU(I)
 560     CONTINUE
*$ critical
         RUR1 = RUR1 + T
*$ end_critical
*$ barrier
         BETA = RUR1/RUR0
         RUR0 = RUR1
*$ end_barrier
*$ pre_sched_do
         DO 570 I = 1,N
            P(I) = RU(I)+BETA*P(I)
 570     CONTINUE
 10   CONTINUE
      RETURN
      END

      SUBROUTINE PRODMV(N,NR,NZU,NZL,IU,IL,AU,AL,AD,X,AP)
      INTEGER NZU(N),NZL(N),IU(NR,N),IL(NR,N)
      DOUBLE PRECISION AU(NR,N),AL(NR,N),AD(N)
      DOUBLE PRECISION X(N),AP(N)
      DOUBLE PRECISION APV
*$ barrier
*$ end_barrier
*$ self_sched_do
      DO 10 I = 1,N
         APV = 0.0D0
         DO 520 MU = 1,NZL(I)
            APV = APV+AL(MU,I)*X(IL(MU,I))
 520     CONTINUE
         APV = APV+AD(I)*X(I)
         DO 530 NU = 1,NZU(I)
            APV = APV+AU(NU,I)*X(IU(NU,I))
 530     CONTINUE
         AP(I) = APV
 10   CONTINUE
*$ barrier
*$ end_barrier
```

```
      RETURN
      END

      SUBROUTINE ICSOLV(N,NR,NZU,NZL,IU,IL,UU,UL,UD,B,X,ISYNC)
      INTEGER NZU(N),NZL(N),IU(NR,N),IL(NR,N)
      DOUBLE PRECISION UU(NR,N),UL(NR,N),UD(N),B(N),X(N)
      DOUBLE PRECISION T
      INTEGER ISYNC(N)

c clear sync vector
*$ pre_sched_do
*%    DO 101 K = 1,N
*%    ISYNC(K) = 0
*%101 CONTINUE
*$ barrier
*$ end_barrier
*$ self_sched_do
      DO 10 K = 1,N
         T = B(K)
         DO 520 MU = 1,NZL(K)
            J = IL(MU,K)
*%    CALL SYNC(ISYNC(J))
            T = T - UL(MU,K)*X(J)
c             T = T - UL(MU,K)*X(IL(MU,K))
 520     CONTINUE
         X(K) = UD(K)*T
*%    ISYNC(K) = 1
 10   CONTINUE
c
*$ barrier
*$ end_barrier
*$ pre_sched_do
*%    DO 102 K = 1,N
*%    ISYNC(K) = 0
*%102 CONTINUE
*$ barrier
*$ end_barrier
*$ self_sched_do
      DO 30 K = N,1,-1
         T = 0.0D0
         DO 540 NU = 1,NZU(K)
            J = IU(NU,K)
*%    CALL SYNC(ISYNC(J))
            T = T + UU(NU,K)*X(J)
c             T = T + UU(NU,K)*X(IU(NU,K))
 540     CONTINUE
         X(K) = X(K) - UD(K)*T
*%    ISYNC(K) = 1
 30   CONTINUE
      RETURN
      END
```

The routines using BL addressing in SOLV of FEM_BAND:

```
c
c FEM_BAND(BL), using the process number
c
*
*     Modified Cholesy decomposition of symmetric band matrix
*
      SUBROUTINE CHDCMP(MDIM,N,M1,AR,W,ISYNC)
      DOUBLE PRECISION AR(MDIM,N),W(N)
      DOUBLE PRECISION T,TT
c
      INTEGER ISYNC(N),ID,ID0,NPROC
      common /nproc/NPROC
      common /global/TT
*
*%    ID = IPID() + 1
*%    ID0 = ID + 1
*%    IF(ID .EQ. 2) ID0 = 1
c
      M = M1-1
      DO 10 K=1,N
        I0 = MAX(1,K-M)
*%    ISYNC(ID) = 0
*     proc 1 can execute first.
*$ barrier
*%    ISYNC(1) = 1
*$ end_barrier

*$ pre_sched_do
        DO 20 I = I0,K-1
          T = AR(M1+I-K,K)
          DO 530 J = I0,I-2
            T = T - W(J)*AR(M1+J-I,I)
 530      CONTINUE
c for the last element I-1
          IF(I0 .GE. I) GOTO 900
*%    CALL SYNC(ISYNC(ID))
          T = T - W(I-1)*AR(M1-1,I)
 900      CONTINUE
c
          AR(M1+I-K,K) = T*AR(M1,I)
          W(I) = T
c sync
*%    ISYNC(ID) = 0
*%    ISYNC(ID0) = 1
 20     CONTINUE
c
*$ barrier
          TT = AR(M1,K)
*$ end_barrier
          T = 0
*$ pre_sched_do
          DO 540 I = I0,K-1
            T = T - W(I)*AR(M1+I-K,K)
 540      CONTINUE
```

```
*$ critical
         TT = TT + T
*$ end_critical
*$ barrier
         AR(M1,K) = 1.0D0/TT
*$ end_barrier
 10   CONTINUE
      RETURN
      END

      SUBROUTINE CHSOLV(MDIM,N,M1,AR,B,X,ISYNC)
      DOUBLE PRECISION AR(MDIM,N),B(N),X(N)
      DOUBLE PRECISION T
c
      INTEGER ISYNC(N),ID,ID0,NPROC
      common /nproc/NPROC
c
*
*%    ID = IPID() + 1
*%    ID0 = ID + 1
*%    IF(ID .EQ. 2) ID0 = 1
c
      M = M1-1
*%    ISYNC(ID) = 0
*     proc 1 can execute first.
*$ barrier
*%    ISYNC(1) = 1
*$ end_barrier
*$ pre_sched_do
      DO 10 K = 1,N
         T = B(K)
         J = MAX(1,K-M)
         DO 520 I = J,K-2
            T = T - AR(M1+I-K,K)*X(I)
 520     CONTINUE
c for the last element k-1
         IF(J .GE. K) GOTO 900
*%    CALL SYNC(ISYNC(ID))
         T = T - AR(M1-1,K)*X(K-1)
 900     CONTINUE
         X(K) = T
c sync
*%    ISYNC(ID) = 0
*%    ISYNC(ID0) = 1
 10   CONTINUE
*
*%    ISYNC(ID) = 0
*     proc 1 can execute first.
*$ barrier
*%    ISYNC(1) = 1
*$ end_barrier
*$ pre_sched_do
      DO 30 K = N,1,-1
         T = X(K)*AR(M1,K)
         J = MIN(N,K+M)
         DO 540 I = K+2,J
```

```
              T = T - AR(M1+K-I,I)*X(I)
 540      CONTINUE
c for the last element K + 1
          IF(K .GE. J) GOTO 901
*%      CALL SYNC(ISYNC(ID))
          T = T - AR(M1-1,K+1)*X(K+1)
 901      CONTINUE
          X(K) = T
*%      ISYNC(ID) = 0
*%      ISYNC(ID0) = 1
 30     CONTINUE
        RETURN
        END
```

Bibliography

[AAD90] D. Alpert, A. Averbuch, and O. Danieli. Performance Comparison of Load/Store and Symmetric Instruction Set Architecture. In *Proceedings of 17th International Symposium on Computer Architecture*, pages 172–181, 1990.

[ACK87] R. Allen, D. Callahan, and K. Kennedy. Automatic Decomposition of Scientific Programs for Parallel Execution. In *Proc. 14th ACM Symp. Principles Programming Languages*, pages 63–76. ACM, 1987.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Weslay Publishing Co., 1986.

[CFR$^+$89] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of 15th POPL*, pages 25–35, 1989.

[CH84] F. Chow and J. Hennessy. Register allocation by Priority-base Coloring. In *Proceedings of the ACM SIGPLAN '84 Symposium on Computer Construction*, pages 222–232, 1984.

[Cha82] G.J. Chaitin. Register Allocation ans Spilling via Graph Coloring. *ACM SIGPLAN Notice*, 17(6), June 1982.

[CLZ86] R. Cytron, L. Lowry, and K. Zadeck. Code motion of control structure in high-level languages. In *Proceedings of 13th POPL*, pages 70–85, January 1986.

[DF84] J.W Davidson and C.W. Fraser. Code selection through object code optimization. *Trans. on Prog. Lang. and Sys.*, 6(4):505–526, Oct. 1984.

[Don79] J.J. Dongarra. Linpack user's guide. Technical report, the Society for Industrial and Applied Mathematics, 1979.

[Fre74] R.A. Freiburghouse. Register Allocation Via Usage Counts. *CACM*, 17(11):638–642, November 1974.

[GHHK] E. Goto, W. Hioe, N. Homma, and R. Kamikawai. G-Series Gate - 1 GHz clock Silicon CML/ECL. to be prepared.

[GJ79] M.R. Garey and D.S. Johnson. *COMPUTER AND INTRACTABILITY, A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[GM86] P.B. Gibbons and S.S. Muchnick. Efficient Instruction Scheduling for a Pipelined Architecture. In *Proceedings of the SIGPLAN 1986 Conference on Compiler Construction*, pages 11–16, 1986.

[GNST86] B.K Gilbert, B.A. Naused, D.J. Schwab, and R.L. Thomposon. Signal Processors Based Upon GaAs ICs: The Need for a Wholistic Design Approach. *IEEE Computer*, 19(10):29–43, Oct 1986.

[HC83] J.L. Hennessy and T.R. Cross. Postpass Code Optimization of Pipeline Constraints. *ACM Trans. on Prog. Lang. and Sys.*, 5(3):422–448, July 1983.

[HF72] T.G. Hallin and M.J. Flynn. Pipelining of Arithmetic Functions. *IEEE Trans. Computers*, C-21(8):880–886, Aug 1972.

[HNM+87] Y. Harada, H. Nakane, N. Miyamoto, U. Kawabe, E. Goto, and T. soma. Basic operations of the quantum flux parametron. *IEEE Trans. Magn.*, pages 3801–3807, Sep 1987.

[HU74] M.S. Hecht and J.D. Ullman. Characterization of reducible flow graphs. *J. ACM*, 21(3):367–375, July 1974.

[Ich87] Shuichi Ichikawa. "A Study on A Cyclic Pipeline Computer: FLATS2". Master's thesis, The University of Tokyo, 1987.

[Ich90] Shuichi Ichikawa. *"A Study on A Cyclic Pipeline Computer: FLATS2"*. PhD thesis, The University of Tokyo, 1990. submitted.

[Joh81] S. C. Johnson. A Tour Through the Portable C Compiler. *Unix technical document*, 1981.

[Jor87] H.F. Jordan. The Force. In L.H. Jamieson, D.B. Gannon, and R.J. Douglass, editors, *The Characteristics of Parallel Algorithms.* MIT press, 1987.

[Jou89] Norman P. Jouppi. The Nonuniform Distribution of Instruction-level and Machine Parallelism and Its Effect on Performance. *IEEE Transactions on Computers*, 38(12):1645–1658, Dec. 1989.

[Kow85] J.S. Kowalik, editor. *Parallel MIMD Computation: HEP Supercomputer and Its Application.* MIT Press, 1985.

[KS86] R. S. Kunkel and E. Smith. Optimal pipelining in supercomputers. In *Proceedings of the 13th Annual International Symposium on Computer Architecture.* IEEE, June 1986.

[LH86] J.R. Larus and P.N Hilfinger. Register Allocation in the SPUR Lisp Compiler. In *Proceedings of the ACM SIGPLAN '86 Symposium on Computer Construction*, pages 255–263, 1986.

[Mas86] Mori Masatake. *Numerical Computation Programming.* Iwanami-shoten, 1986. in Japanese.

[McM84] F.H. McMahon. L.L.N.L. FORTRAN KERNELS: MFLOPS. Technical report, Lawrence Livermore National Lab., 1984.

[MFH86] V. Milutinovic, D. Fura, and W. Helbig. An Introduction to GaAs Microprocessor Architecture for VLSI. *Computer*, 19(3):30–42, March 1986.

[MFHL87] V. Milutinovic, D. Fura, W. Helbig, and J. Linn. Architecture/Compiler Synergism in GaAs Computer System. *Computer*, 20(5):72–93, May 1987.

[Mor90] I. Morishita. A New Pipelined MIMD Processor for Large Scale Parallel Machines with Multistage Interconnection Networks. *Trans. of Info. Proc. Soc. of Japan (in Japanese)*, 31(4):523–531, April 1990.

[Pol89] C. D. Polychronopoulos. Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design. *IEEE Transactions on Computers*, 37(8):991–1004, Aug. 1989.

[RWZ88] B.K. Rosen, M.N. Wegman, and F.K Zadeck. Global Value Numbers and Redundant Computation. In *Proceedings of 15th POPL*, pages 12–27, 1988.

[SGI89] K. Shimizu, E. Goto, and S. Ichikawa. CPC(Cyclic Pipeline Computer) - An Architecture Suited for Josephson Pipelined-Memory Machines. *IEEE Transactions on Computers*, 38(6), June 1989.

[SHS$^+$87] M. Suzuki, K. Hiraki, K. Shimizu, M. Sato, and N. Inada. FLATS Architecture and its Evaluation. In *Conference Proceedings of TENCON 87*, pages 1039–1043, August 1987.

[SIG89] Mitsuhisa Sato, Shuichi Ichikawa, and Eiichi Goto. Run-time Checking in Lisp by Integrating Memory Addressing and Range Checking. In *Proceedings of 16th International Symposium on Computer Architecture*, pages 290–297, 1989.

[SIG90] Mitsuhisa Sato, Shuichi Ichikawa, and Eiichi Goto. Multiple Instruction Streams in a Highly Pipelined Processor. In *Proceedings of 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 182–189, Dec. 1990.

[Sit79] R.L. Sites. The Compilation of Loop Induction Expressions. *ACM Trans. on Prog. Lang. and Sys.*, 1(1):50–57, July 1979.

[SSFG89a] Mitsuhisa Sato, Paul Spee, Norihiro Fukazawa, and Eiichi Goto. "CPX: Exploiting concurrency on the CPC". In *Proceedings of 6th Riken Symposium on Josephson Electronics*, pages 7–14, 1989.

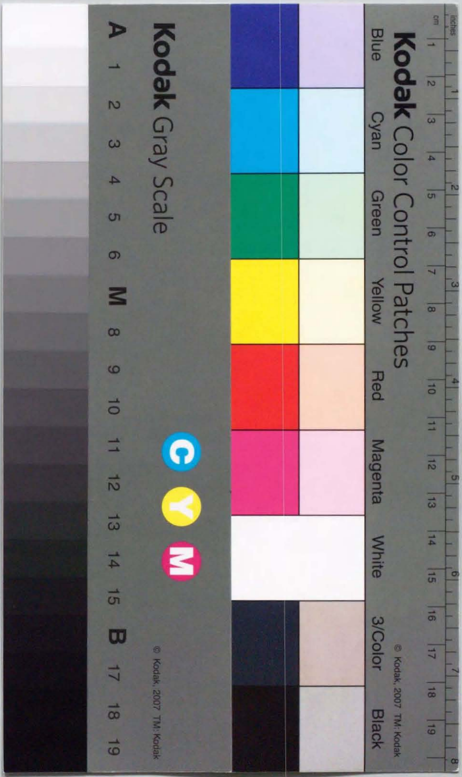
[SSFG89b] Paul Spee, Mitsuhisa Sato, Norihiro Fukazawa, and Eiichi Goto. "CPX - An operating System Kernel for CPC". In *Proceedings of 6th Riken Symposium on Josephson Electronics*, pages 15–25, 1989.

[Sut89] Ivan E. Sutherland. MICROPIPELINE. *CACM*, 32(6):720–738, June 1989. the lecture of Truing Award.

[Tar70] R.E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Computing*, 1(6):146–160, 1970.

[TTT81] M. Tokoro, E. Tamura, and T. Takizuka. Optimization of microprograms. *IEEE Trans. on Computers*, C-30(7):491–504, July 1981.

[WZ84] M.N. Wegman and F.K. Zadeck. Constant Propagation with Conditional Branches. In *Proceedings of 12th POPL*, pages 291–299, 1984.


inches
cm
Kodak Color Control Patches
Blue
Cyan
Green
Yellow
Red
Magenta
White
3/Color
Black
© Kodak, 2007 TM: Kodak
Kodak Gray Scale
C
Y
M
© Kodak, 2007 TM: Kodak
A 1 2 3 4 5 6 M 8 9 10 11 12 13 14 15 B 17 18 19