Multipurpose Short-Term Memory Structures

By

Yung, Chan

Supervised By :

Dr. Chi-hung, Chi

Submitted to Department of Computer Science
in partial fulfillment of the requirements for the
degree of Master of Philosophy
at the
The Chinese University of Hong Kong
June 1995

QA
76.9
M45C43
1995
ut

Abstract

There are three main problems in computer architecture that can degrade the system proformance. They are

1. Ambiguous aliasing problem which causes extra *LOAD* and *STORE* operations.

2. Too many primitive instructions.

3. High cache miss ratio.

These three problems can lead to poor system performance. In order to improve the system performance, the above three problems must be solved. The approach of eliminating the ambiguous aliasing is to use hardware support to keep coherence of the ambiguous aliased variables. The approach to the second problem is to change the operation of explicit instructions into implicit operations. And the approach to the third problem is to enable data prefetching. In this thesis, the attempt to improve the performance of the system by proposing six new memory structures based on these three approaches.

The name of these new memory structures is called **EReg** which stands for *Extended Register*. The first structure is called Basic Model which provides coherence support for all the registers such that the ambiguous aliasing problem is eliminated. The second structure is called ADM model which extends the Basic model by enabling implicit storing to further reduce the number of store operations.

The third structure is called ADS model which extends the Basic model by incorporating data prefetching. Prefetching can reduce the penalty of long memory access time by increasing the cache hit ratio. An effective hardware-based prefetching strategy is employed to prefetch the data in constant stride access patterns. The fourth structure is called ADSM model which extends the ADS model by supporting implicit storing such that the number of store operations can be further reduced.

The fifth structure is called IADSM model which extends ADSM model by enabling implicit loading such that a lot of array access *LOAD* operations can be greatly reduced.

From the simulation on the kernels of NASA7 and livermore loops, the reduction can be up to 83%.

Finally, the last structure is called IADSMC&IDLC model which extends the IADSM model by enabling implicit looping such that all the loop controlling instructions (e.g. branch instruction) can be reduced. Some of these reduced instructions are not required due to the characteristics of the new memory structure while some of them are changed into implicit instructions. If the last structure is employed, nearly 90% issued instructions are reduced.

Above all, the primary aims of the six models is to improve the system performance by

1. reducing the number of primitive instructions;

2. eliminating the ambiguous aliasing problem; and

3. reducing the cache miss ratio.

Acknowledgement

I would like to express my hearty appreciation to my supervisor, Dr. Chi Chi-Hung. I am really indebted to him for his inspiration, patience, encouragement and guidance bestowed to me throughout the project. I cannot thank anyone else more than Dr. Chi Chi-Hung.

I would like to thank Dr. Moon Yiu-Sang and Dr. Young Ho-fai, Gilbert for their helpful comments and constructive criticism in my study. Of my colleagues, Ho Chi Sum and Keith Mak was most helpful in giving suggestions to my project. Moreover, I would like to thank Kwan Chi-Wai, Kan Chi-Kwun, Yu Chung-Fai, Lei Chong-Meng, Cheung Hon-Kai and Lau Sau-ming in providing a wonderful environment for my study. Lastly, but certainly not the least, I wish to extend my gratitude to my parents for their spiritual support in the study.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In RISC architecture, all data operands used for program execution must be stored in registers. A typical instruction mix shows that 22% of instructions executed are *LOAD* instructions and 5% are *STORE* instructions. In order to reduce the memory access delay, cache is often used. Typically, a cache hit may just take 1 clock cycle, while a cache miss takes 8 - 32 clock cycles[HP90].

There are three problems inherent in the current RISC architecture. Firstly, storing data in register suffers from data aliasing problem which will be discussed later. Secondly, too many primitives instructions are executed. Thirdly, high cache miss ratio increases the overall memory access time. These three problems can cause performance degradation. In this thesis, some effective short-term memory structures are proposed to solve these problems. The name of these new memory structures is called **EReg** which stands for the *Extended Register*. In order to explain how and why these ERegs can improve the performance, it is necessary to have an overview on the current short-term memory structures (i.e. register and cache) and discuss why they cannot solve the aforesaid problems.

1.1 Cache

1.1.1 Introduction

A cache is a small, fast memory located close to the CPU. It is used to hold the most recently used instructions or data. The cache data is accessed by using memory address.

If the data item accessed by the CPU cannot be found in the cache, a cache miss occurs and the data will be retrieved from the main memory and put into the cache. Since data in the cache can be accessed in much shorter time than that in the main memory, the total memory delay during execution can be greatly reduced if a significant fraction of the references can be found in the cache.

Cache misses can be categorized into three types : compulsory, conflict and capacity. Compulsory miss is due to the first time access of the data object. Conflict miss occurs when the referenced data is replaced by another data object within the same set. Capacity miss is caused by the small cache size which is not large enough to hold the working set of the program[Hil87]. In multiprocessors environment, a data is non-cacheable if it can be read by more than one processor and its value may be changed by some processors.

1.1.2 Data Prefetching

Data prefetching can load a data item into the cache before it is actually referenced by the processor and hence, it can reduce compulsory misses. However, such techniques not only require the cache to hold the current working set, but also the future working set. It may cause cache pollution and may increase conflict and capacity misses especially when the data is prefetched too early. However, if the data is prefetched too late, the time for loading the data item into the cache may not be able to overlap with the program execution. Usually, the prefetching operations do not block the processor and the demand *LOAD* operations have a higher priority over prefetching operations.

1.2 Register

Another important short-term memory structure is register. The loading and storing of registers are done explicitly by the *LOAD* and *STORE* instructions. In general, the number of registers is much less than that of cache memory cells. Each register can be directly accessed using a short register name. Since its address space is also separated from the main memory and the number of register is small, hence the register names are typically shorter than memory addresses. Therefore, data-fetch and instruction-fetch bandwidth can be decreased. Furthermore, since access to registers does not interfere with the reference to memory, the usable bandwidth of memory can be increased.

Apart from the above discussion, there are three other reasons why register usually gives a better performance than the cache memory. Firstly, the access time of a register is faster than that of a cache entry. Secondly, access time of register is more predictable than that of the memory system including cache and main memory. Thus, better compiler optimization can be performed. Thirdly, if the program is carefully optimized, doubling the size of register file often perform better than doubling the size of cache. However, if the program is poorly optimized, doubling the size of the cache may in turn give a better performance gain than doubling the size of the register file[Sit79].

1.3 Problems and Challenges

1.3.1 Overhead of registers

Basically, there are two sources of serious overhead associated with register data. Firstly, it is often necessary to save the current machine state when there is a subroutine call or switching task. This means that explicitly saving and restoring all the programmer-visible registers are needed. When the size of the register file is getting larger, subroutine calls will become slower. Secondly, data aliasing will also cause register data to be stored and reloaded frequently. To explain this problem, let us consider the example in Figure 1.1. The value of b[i] is loaded into register in both lines L2 and L6. The reason of loading b[i] in line L6 is that if a[i] is aliased to b[i], the value of b[i] might be changed by the update of a[i] after the line L4. The compiler cannot determine if the two names a[i] and b[i] are aliased with each other. When such case happens, *STORE* and *LOAD* operations will be required. One of the main features of the ERegs is to eliminate these two problems while maintaining the benefit of registers. The next subsection 1.3.2 will discuss how to reduce the instructions of L9, L10 and L11 by using the new proposed memory models – ERegs.

Program segment in "C" language :

```
C1:    For (i=0;i<n;i+=8) {
C2:        a[i]=b[i]+c[i]
C3:        d[i]=b[i]+a[i]
           }
```

Pseudo assembly version :

```
L1:    load       c[i]
L2:    load       b[i]
L3:    compute    b[i]+c[i]
L4:    store      a[i]
L5:    load       a[i]
L6:    load       b[i]
L7:    compute    b[i]+a[i]
L8:    store      d[i]
L9:    add        i,8
L10:   compare    i,n
L11:   goto       L1  if i < n
```

Figure 1.1: Overhead of using traditional registers

1.3.2 EReg

In this thesis, discussion will be made on how ERegs are designed and used to enhance the system performance. A total of six new memory models called ERegs are proposed. The areas that ERegs try to optimize include :

1. Implicit Storing
2. Data Prefetching
3. Implicit Loading
4. Implicit Looping

Implicit Storing is emphasized on reducing the number of *STORE* instructions. The reduction is mainly done by converting explicit *STORE* operations into implicit *STORE* operations. Sometimes, the explicit *STORE* operations can be simply eliminated. 90% of the *STORE* instructions in the Livermore Loop Benchmark kernels can be reduced. In **Figure 1.1**, the *STORE* instructions of L4 and L8 can be eliminated by changing them into implicit *STORE* operations.

Data prefetching has been discussed and studied by a lot of researchers. Previous researchers proposed some prefetching designs which can achieve a high cache hit ratio. However, such designs are usually too costly to implement. In our EReg design, a very accurate data prefetching can be achieved, yet the additional hardware involved is very simple. In Figure 1.1, the traditional method of ONE BLOCK LOOK AHEAD cannot reduce any cache miss. The reason is that the next data is 64 bytes away from the current data (assume each data is 8 bytes long). However, if EReg is used, the data of a[i], b[i] and c[i] in the next iteration can be prefetched easily with the help of the compiler and very high performance can be obtained.

Implicit Loading is emphasized on reducing the number of *LOAD* instructions by changing the explicit *LOAD* operations into implicit *LOAD* operations. This area is seldom studied by previous researchers. From the simulation results, up to 90% of the

LOAD instructions in the Livermore Loop Benchmark kernels are reduced. In Figure 1.1, the instructions of L1, L2, L5 and L6 can be eliminated.

Implicit looping is to eliminate the instructions which control the operation of looping. Generally, there are three well-defined steps in the loop control:

- Increment of the index value
- Comparison bewteen the index value and the looping limit
- Conditional branch instruction

These three well-defined steps are definitely non-negligible overhead to the system performance. This area is also seldom studied by previous researchers. However, we will show that ERegs can reduce the type of loop controlling instructions in the last chapter. In the example of Figure 1.1, the instructions of L9, L10 and L11 can be eliminated.

1.4 Organization of the Thesis

In the thesis, we will go through our six new models. We will first focus on the architectural design, operational details of each model. And then an example will be used for demostration of the idea. Next, the simulation result of the kernels from the NASA7 Fortran Language Program Benchmark and Livermore Loop C Language Program Benchmark are presented and discussed. Finally, some architectural and operational variations for each model are discussed.

In Chapter 2, some previous studies on ambiguous data aliasing and data prefetching will be presented. In Chapter 3, two models, called the Basic and ADM models, will be presented. They mainly concentrate on solving the ambiguous data aliasing problem and implementing the feature of implicit storing. In general, they can replace the traditional registers effectively. In Chapter 4, two enhanced models, called the ADS and ADSM, will be presented. They are not only designed to replace the traditional registers, but also incorporated with data prefetching. In Chapter 5, the models will be further enhanced to include the feature of implicit loading and implicit looping. The models are called IADSM and IADSMC&IDLC. In Chapter 6, we will give a general discussion on the problems of implementing the models of ERegs. It includes the impact of the ERegs on the memory system and on the compiler optimization techniques. And finally, in Chapter 7, some conclusions will be made on the effectiveness of ERegs and future research direction on ERegs will be suggested.

Chapter 2

Previous Studies

2.1 Introduction

When there are more than one name to reference a program object, data aliasing problem may occur[ASU86]. Suppose there are two variable names α and β. In the compile-time analysis, if a compiler *can prove* that α and β cannot be aliases for the same memory location, then α and β will be assigned to different register. Instead, if the compiler *can prove* that α and β are always be aliases for the same memory location, then α and β are assigned to share a single register. However, if the compiler *cannot determine* whether α and β refer to the same memory location, we say that α and β are ambiguously aliased to each other.

This chapter gives a survey on the relevant previous work. Basically, there are two concerned areas. One is the ambiguous aliasing data and the second one is the data prefetching. Up to the present moment, there are seldom previous research study on the areas of implicit loading and implicit loop control. Now, let's look at the ambiguous data aliasing problem first.

A simple approach to solve this ambiguous aliasing problem is to assign these variables to different registers. However, when one of them is changed, its register value must be stored back to the memory. Moreover, further accesses to the register copy of other variables in the same aliased set will cause them to be loaded again into the register before they are accessed.

For example,

```
1.      readln(i,j);
2.      a[i] = a[j]+c;
3.      d=a[j]+c;
```

The variables a[i] and a[j] would refer to the same memory location when the input values of i and j are the same. Then, after the result of the line 2 is stored into a[i], the register copy of a[i] has to be stored back to the main memory. Furthermore, the data a[j] in line 3 has to be loaded again into the register before it can be used in the calculation. Such aliasing problem can occur in many other scenarios.

2.2 Data aliasing

There are two basic approaches to handle the data aliasing problem. One is to use software methods[CK89], [LH88] and [Mye81] to disambiguate the aliasing problem such that a better code optimization can be obtained. Nicolau[Nic89] described a technique in which dependent *STORE/LOAD* pairs can be reordered by inserting explicit address comparison and conditional branch instructions. Any incorrect execution due to wrong reordering can also be rescued by inserting some extra instructions. However, this approach might cause the overhead of a longer runtime. Another approach is to use extra hardware to maintain the consistency among the register copies of the aliased members. Since our models belong to the hardware approach, previous hardware methods will be described in more details.

The simplest method is to eliminate the register storage. However, this method has a serious drawback that it cannot obtain the benefits of using registers. Another approach is to use hardware support to maintain the consistence of accessing same program object via different variable names. The designs include tagged-register mechanism, indirection-resolution and register preload.

Sites [Sit79] generalized the structures of register and cache. He proposed an idea called "Renaming". A RENAME operator was defined such that RENAME X, Y would mean that the short name X will be used to access the long name Y until another RENAME operation involving X is encountered. This RENAME operation is very similar to the *LOAD* operation of a register, but it differs from *LOAD* operation in the way that no data movement is implied. Since one of the assumptions of the method is that the implementation must ensure that references using the short name X and the long name Y both access the same actual data, ambiguous aliasing problem will not occur.

Dietz and Chi [HC88] proposed a structure called CRegs. Basically, CRegs is a tagged register and is a superset of register and cache. Each CReg has an address field and a data field. When a CReg is updated, an fully associative search is made to determine if there are any CRegs which have the same address value as the one being updated. Any CRegs found by this association are aliases for the CReg being directly named, and the CReg hardware simply performs an associative update to these entries. Furthermore, the capacity of update associativity can be reduced by partitioning the CRegs into sets such that the compiler can put those ambiguously aliased and simultaneously live references into the same set. An implementation study of CRegs based on the MIPS-X RISC Processor was taken by Steve Nowakowski and Mattew T.O'Keefe[NT92].

Heggy and Soffa [BM90] proposed a mechanism called variable forwarding. Each data only has a single copy stored in the register file, called the leader which satisfies all the memory and register accesses to the data. Hence, there is no need to maintain the consistency between memory and register copies, or among register copies. However, the large overhead in setting the aliasing reference group and accessing method through the leader may offset the performance gain.

Chiueh in [Chi91] proposed an approach called *indirection resolution* which aims at solving the ambiguous data aliasing implicitly. The novelty of this mechanism is the integrated operation of the on-chip memory hierarchy. The on-chip storage includes an on-chip data cache and a register file. All ambiguously aliased accesses including both the register and memory references are served by the data cache. Each cache word can be accessed using memory address or cache address (i.e line number/word-in-line number). Since the cache supports two addressing modes, it is called *dual-addressed cache.*

Another important idea in this approach is the *register indirection.* During the register allocation, ambiguously aliased data will be assigned to distinct registers. However, these objects are never physically brought into the registers, Instead, the cache addresses of the ambiguously aliased data are stored into the registers. Hence, a register access is turned into a cache access using the corresponding cache address. As a result, ambiguously aliased objects can be accessed either through a memory address or through an indirect register specifier. Although there could be more than one register that holds the same cache address, only one physical copy of each ambiguously aliased object will be found in the register/cache-memory hierarchy. So there is no inconsistency problem.

William [WYMC93], described a method called *preload register update* which supports the compiler to move a memory load above a memory *STORE* when their dependence state is not certain. When a *LOAD* is moved above a *STORE* and their dependence relation is uncertain, the *LOAD* is called a preload. There is a coherence mechanism to update the preload destination register if the preload address is just the same as the *STORE* address. For each register, an address register is associated with it. If we have n general purpose registers, n address registers are added. The purpose of these address registers is to store the addresses of preloads. When a *STORE* instruction is executed, the *STORE* address is compared against all preload addresses in the address registers. When there is a match, the stored value is forwarded to the corresponding data register for updating. Since there are multiple address registers, a fully associative comparison between the *STORE* address and the preload addresses is made in order to shorten the search time. To stop the coherence update of a particular preload register, a commit instruction is introduced. It is usually inserted at the original position of the *LOAD* instruction. When the commit instruction is executed or the register is redefined by a normal instruction, the coherence update of the specified preload register is stopped.

However, the algorithm has a major limitation. Only the *LOAD* instruction can be scheduled above ambiguous *STORES*. Other instructions, such as *ADD,MOVE,*which depends on the result of the *LOAD* instruction cannot be moved above the *STORES*. In order to increase the parallelism, William based on the above idea proposed the *Memory Conflict Buffer (MCB)*. A memory *LOAD* and its dependent instructions can be moved above any number of memory *STORES* regardless of the dependence relation between the *LOADS* and the *STORES* by using the MCB. MCB supports the code reodering by detecting the situation in which the ambiguous reference pair accesses the same location and subsequently invoke a correction code sequence supplied by th compiler to restore the correctness of the program execution.

2.3 Data prefetching

2.3.1 Introduction

Due to the diverse speed gap between the processor and the memory, data prefetching can be an effective method to increase the cache hit ratio and to reduce the penalty of data access. However, since prefetched data may flush out the current working set of data in the cache, cache prefetching can cause cache pollution and can result in performance degradation. Sometimes, prefetched data may even flush out each other before they are referenced. The prefetching approaches may be based on hardware only or with software support. Data prefetching with software support has an advantage over pure hardware methods that the compiler can make an intelligent prediction on future program flow. However, their main problem is the overheads on extra instruction execution[MLG92].

2.3.2 Hardware Prefetching

The simplest Hardware Prefetching method is the ONE BLOCK LOOK AHEAD which prefetches the cache line address i+1 when the cache line address i is referenced [Smi82], [Prz90]. In general, hardware-based approaches are simple and do not have extra runtime overhead. However, they only perform will in sequential access of data and fail in all the other accesses.

As classified by [Che93] and [BC91], there are four types of access patterns – scalar, zero stride, constant stride and irregular. Most hardware-based approaches perform well in the case of constant stride access pattern.

In Chen's thesis [Che93], he proposed the Reference Prediction Table(PRT) which is used to keep track of previous reference addresses and their associated strides for *LOAD* and *STORE* instructions. The table contains four fields. The first field is a tag corresponding to the instruction address of the *LOAD/STORE*. The second field contains last(operand) reference data address for the same instruction. The third field is a stride value which is obtained from the difference between the current data access address and previous data access address for the same instruction. The fourth field is a two-bit state which indicates the past history of prefetching. If the stride value is not changed in 3 iterations, it is regarded as stable and then prefetching data in the next iteration will be performed. The performance of this design can be further improved by adding a *Lookahead Program Counter*. However, it is difficult to justify the benefit of this design since it involves too much hardware overhead for supporting prefetching. Other similar schemes are proposed in [Skl92] and [FPJ92].

2.3.3 Prefetching with Software Support

By analyzing the program statically, the compiler is able to insert data prefetching instructions into the program so that the data is in the cache before they are referenced [CKP91], [MG91] and [CMCmH91].

Porterfield[Por89] studied the prefetching of array references in the innermost loops of a program. He found that the performance of prefetches one iteration ahead is better than the simple hardware prefetching, e.g. One Block Lookahead.

Gornish et.al.[GGV90] further proposed an algorithm for prefetching an entire block instead of a single cache line. He also attempted to determine the earliest point at which a datum can be fetched in multiprocessor environment. However, dependence constraints limit the ability to pull out a particular reference from a limited number of loops.

2.3.4 Reducing Cache Pollution

Data Prefetching may increase cache pollution which in turn creates additional bus traffic. Previous researchers [KL91], [LYL87], [CBM+92] proposed an architectural support, called the prefetch/fetch buffer, to solve the cache pollution problem and to tolerate long memory access latencies. The prefetch/fetch buffer holds the possible future working set. The processor therefore operates on two separate caches. When the data in the prefetch/fetch buffer is referenced, its associated cache line is transferred from the buffer into the data cache. However, this method has two problems :

- Associativity of the prefetch/fetch buffer
- Coherence between the data cache and prefetch/fetch buffer

A similar architecture called the preload buffer was also proposed in [CBM+92]. All preload data were stored in the preload buffer. The data in the preload buffer would not be transferred to the cache when they were used.

Three small fully-associative caches called miss cache, victim cache and steam buffer were proposed by Jouppi[Jou90]. In the case of using miss cache, all the missed cache line will be forwarded to both the data cache and miss cache such that when the line is replaced from the data cache, the miss cache may still contain this line. In the case of using victim cache or steam buffer, any replaced lines from the data cache will be forwarded to victim cache or steam buffer.

Chapter 3

BASIC and ADM Models

3.1 Introduction of Basic Model

Traditional registers face the problem of *ambiguous data aliasing.* The problem is investigated in the following paragraphs. Supposing that there are two names α and β. If they are always aliased to each other, the compiler can simply assign a single register for them. On the other hand, if they are proved impossible to be aliased with each other, α and β will be assigned to different registers.

However, if the compiler cannot make sure whether they are aliased to each other, it will cause a serious problem that extra *STORE* and *LOAD* operations are required. In this case, the name α and β are said to be ambiguously aliased with each other. To explain this, let us go into the following example. There are two names α and β which are ambiguously aliased with each other. The compiler assigns different registers to hold their values. If the value of the register copy of one of them, say α, is changed, the value of the register copy of another variable (i.e β) may have been *outdated* because α and β may refer to the same location. In order to obtain the correct program execution result, the value of the register copy of β must be updated before any reference access to the register copy of β. Usually, a pair of *STORE*/*LOAD* operations is employed to update the value of β if necessary.

1. *STORE* the value of α into memory.

2. *LOAD* the value of β from memory

Hence, the ambiguous aliasing problem decreases the benefit of using registers.

Aliases can occur in a lot of ways. The followings are some of the scenarios in which ambiguous data aliasing occurs.

- There are more than one name to access a single variable in pointer operation.
- The global variable is passed into a procedure by name.
- The same variable is passed in several positions of the argument list during call-by-reference procedure call.
- The index value of array elements are determined at run time such that a[i] may be aliased to a[j].

Cache does not have ambiguous data aliasing problem. The above variables α and β can be placed safely inside cache. The reason is explained as follows. Since memory address is used to address the data in cache, if two objects in cache are aliased with each other, their memory address will be the same. Therefore, only one copy of the value will exist and there is no ambiguous data aliasing problem.

We propose our basic model of EReg in order to achieve the same benefit as traditional register and solve the problem of ambiguous data aliasing just like cache. The memory structure of the Basic model is shown in Figure 3.2. Its structure is the superset of register and cache. It has two fields – address field and data field. The address field holds the address of the data stored in data field. The main idea of the Basic model is to make the data content of the ERegs , which have same address value, coherent. To understand this, let us consider the function in Figure3.1:

If the function in Figure 3.1 is called by SUM(X,X,Z), $*$A and $*$B will refer to same object. If it is called by SUM(X,Y,Z) where X is different from Y, $*$A and $*$B will refer to different objects. Therefore, the way to call this function decide whether the names of $*$A,$*$B are aliased with each other. Since the compiler does not know whether they

```
L1: SUM(INT *A,INT *B,INT C)
L2:{
L3:     *A = *B+C
L4:     *B = *B+C
L5:}
```

Figure 3.1: Ambiguously aliasing problem

are aliased with each other, it will be harmful to place them into traditional registers as discussed previously. However, if EReg is used, *A and *B can be safely placed into two ERegs, ER1 and ER2, because the aliasing problem is solved by the EReg hardware. When the value of ER1 is changed in L4, the address field of ER1 will compare with the address field of all other ERegs in a fully associative way. If *A and *B are really aliased with each other, they will have the same address value. The matching is successful between ER1 and ER2 so that the ERegs will update the value of ER2 accordingly. In other words, the values of *A and *B are always kept in coherence with each other and without any access to the main memory. Both the number of *STORE* and *LOAD* operations can be reduced in data aliasing problem.

Similarly, if the EReg ER1 is loaded with a new data using a *LOAD* instruction, a fully associative search is also made between the address of the new data and the address fields of the EReg file and to see if there are any ERegs which have the same address values. If the matching is successful, the data value of those matched ERegs will be forward to ER1 such that the loading operation from the main memory is eliminated. Thus, in general, the average time of *LOAD* operation is shorter after using ERegs. The next section 3.2 will describe the architectural and operational details of Basic model.

3.2 Architectural and Operational Detail of Basic Model

The instructions to change the data content of EReg can be classified into three types of operations. They are *LOAD operation*, *STORE operation*, *arithmetic and logical operations.* The operational details are discussed on these three types of operations.

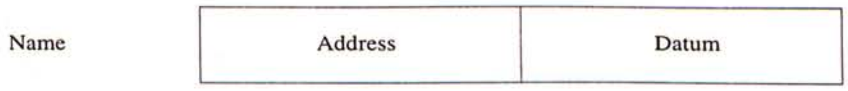

Figure 3.2: Memory structure of Basic Model

Operational Details

Case 1 : *LOAD* Instruction :

When a *LOAD* instruction for loading the content of the memory address *ADDRESS_A* into an EReg,e.g ERi, is executed:

e.g. *LOAD ADDRESS_A*, ERi

- The content of the address field of ERi is set to the address value *ADDRESS_A*.
- The address value *ADDRESS_A* is checked against the content of the address field of all ERegs in the EReg file simultaneously.

  - If a match is found between the address value *ADDRESS_A* and the content of the address field of some ERegs, the content of the datum field of these ERegs are copied into the datum field of ERi.

  - If no match is found in the EReg file, the memory content with the address *ADDRESS_A* is loaded from memory into the datum field of ERi.

Case 2 : *STORE* Instruction :

When a *STORE* instruction storing data of an EReg, e.g. ERi, to the memory address *New_address* is executed :

e.g *STORE* ERi, New_address

- The address *New_address* is checked against the content of address field of all ERegs in the EReg file,

  — If a match is found between the address *New_address* and the content of address field of some ERegs, both the memory content with address *New_address* and the content in the datum field of these ERegs are updated by the content of the datum field of ERi.

  — If no match is found in the EReg file, only the memory content with address *New_address* is updated.

Case 3 : Arithmetic and Logical Operation :

When an arithmetic or logical operation with destination EReg,e.g ERi, is executed :

- The content of the datum field of the ERi is updated as usual.
- The content of the address field of the ERi is read and compared associatively with the content of the address field of all other ERegs. If they match, the content of the datum field of these ERegs are updated by that of ERi

3.3 Discussion

3.3.1 Implicit Storing

Implicit Storing means that *STORE* operation which are performed implicitly and use address content of the EReg as storing address. It can be used to eliminate the issue time of *STORE* instructions and number of *STORE* operations. If *implicit storing* is supported by Basic model, the operation of *implicit storing* can be implemented as follows. Whenever there is an EReg to be replaced (*i.e during Loading operation*), an associative match is made between the address content of this EReg and that of all ERegs. If the matching is successful, the content of this EReg can be simply discarded and then this EReg is used to hold the new data immediately. If the matching is unsuccessful, the content of this EReg must be implicitly stored by using the content in its address field as the storing address.

However, many data are usually not modified before they are replaced. If implicit storing is really supported by Basic model, all the data must be stored implicitly when they are replaced and they do not have any aliased ERegs. A lot of unnecessary *STORE* operations are performed and thus implementing implicit storing in Basic model may not be beneficial. Hence, implicit storing is not supported by Basic model. That's also the reason to extend the Basic model to ADM model by including one more bit called modified bit which will indicate if the replaced ERegs have been modified or not.

Traditionally, there are two parameters for *STORE* instruction. One is to specify the source EReg. Another is the destination memory address. For example,

```
ST ERi, DST_ADDRESS
```

In Basic model, although the implicit storing is not supported, we may still obtain part of the benefits offered in implicit storing by including a new explicit *STORE* instruction which can provide part of the benefits offered by implicit storing.

```
STi ERi
```

This new *STORE* instruction, *STi*, uses the content of the address field of ERi as the destination address implicitly. Before we go to discuss why this new instruction can provide part of benefits offered by implicit storing, it is first necessary to discuss the benefits of implicit storing in detail. The benefits of implicit storing are :

1. eliminating the issue time of *STORE* operations;
2. eliminating some *STORE* operations;
3. reducing the demand of the number of ERegs; and
4. reducing the instructions to find the storing address of the *STORE* instruction.

Firstly, since implicit storing provides *STORE* operation implicitly, it does not involve the overhead of issuing the *STORE* instructions. Secondly, supposing that there are 3 *STORE* instructions storing the content of the variables A, B and C. If A, B and C are aliased to each other, these 3 explicit *STORE* operations can be reduced to only one implicit storing operation in case of the implicit storing being supported. Hence, the implicit storing can eliminate some *STORE* operations. Thirdly, only *LOAD* instruction can change the content of the address field of EReg. Therefore, an explicit *STORE* operation can be changed to an implicit *STORE* operation only if the value of the destination address of this explicit operation has been loaded and stayed in EReg file currently. Hence, the destination address should have been available in the address field of an EReg, say ERi, before the issue of the explicit *STORE* operation on ERi. Since we do not provide means to access the address field of EReg, this require an extra EReg to hold this destination address. If implicit storing is supported, this extra EReg is not required and hence the demand of the number of ERegs can be reduced. Fourthly, if the number of ERegs is not sufficient to hold the value of the above destination address, some instructions are required to find out the value of this destination address before the issue of the corresponding *STORE* instructions. Again, if implicit storing is supported, these instructions can be eliminated.

The new instruction,*STi*, can provide the above third and fourth benefits. Since the destination address has been stored in the address field of the EReg, say ERi, it is not necessary to use an extra EReg to hold the destination address or some instructions to find out the value of this destination address again.

3.3.2 Associative Logic

It may be very expensive to implement a fully associative Basic EReg file, the EReg file can be divided into sets, e.g, four ERegs each set. All the ambiguous aliasing objects will be placed into same set. Only those within the same set are maintained to be coherent. The simple way to divide the EReg file into sets can be handled by the method that all ERegs belong to the same set if all the bits of their ERegs' name are the same, except the last two bits. If the number of the ambiguous objects is more than the number of entries of the set, we can handle the problem by the following mechanism :

1. We can assign EReg to each ambiguous object freely before all the ERegs within the set are filled up.
2. Once all the ERegs within the set are filled up, further EReg request from same aliasing set will be honored by replacing one of the ERegs within the set.

As the basic feature of EReg, any loading will trigger an comparison between the *LOAD* address and the content of the address fields in other ERegs within the set. Any match will cause the data from the matched ERegs forward to the loading EReg rather than loading from memory. Thus, during the step two of the above mechanism, the replacement may not imply a *LOAD* operation. The reason is that if there are any aliased ERegs, the loading operation is changed to data movement operation between EReg and EReg.

3.4 Example for Basic Model

The kernel of hydro fragment from the Livermore Loop is extracted out as an example in Figure 3.3. In the example, if traditional register is used, the array elements z[k+10] & z[k+11] will be loaded in each iteration since there may be an aliasing problem between x[k] and z[k+11] or x[k] and z[k+12]. However, if BASIC model of EReg is used, we can schedule the coding as shown in Figure 3.4. Then, the total number of 4 loading instructions for the array elements of z[k+11], z[k+10], z[k+12] and z[k+11] in each iteration can be decreased to only 2 loading instructions z[k+11] and z[k+12] with one more *LOAD* instruction outside the loop of "for (k=0;k<n;k++)".

Hydro fragment :

```
for ( k=0;k<n;k++)
x[k]=q+y[k]*(r*z[k+10]+t*z[k+11]);
```

Pseudo assembly version :

```
0.    STEP1
1.    LOAD       z[k+11]
2.    Compute    t*z[k+11]
3.    LOAD       z[k+10]
4.    Compute    r*z[k+10]
5.    .....      ......
6.    Compute    q + y[k]*(r*z[k+10]+t*z[+11] ) and STORE into x[k].
7.    LOAD       z[k+12]
8.    Compute    t*z[k+12]
9.    LOAD       z[k+11]
10.   Compute    r*z[k+11]
11.   .....      .........
12.   Compute q + y[k+1]*(r*z[k+11]+t*z[+12] ) and STORE into x[k+1].
13.   Increment the value of k by 2.
14.   goto       STEP1
```

Figure 3.3: Example : Hydro Fragment

3.5 Simulation Results

There are totally 15 kernels used in the simulating the effect of the Basic model of EReg. The source code of these 15 kernels are placed in Appendix A. The first 12 are Livermore Loop "C" language kernels. All kernels will be compiled by cc compiler into assembly programs in SPARC machine environment with the command " cc -O4 -S filename ".

```
3.      LOAD        z[k+10]
0.      STEP1
4.      Compute     r*z[k+10]
1.      LOAD        z[k+11]
2.      Compute     t*z[k+11]
5.      .....       ........
6.      Compute     q + y[k]*(r*z[k+10]+t*z[+11] ) and STORE into x[k].
9.      ( remove )
10.     Compute     r*z[k+11]
7.      LOAD        z[k+12] ( use same EReg as z[k+10] )
8.      Compute     t*z[k+12]
11.     .....       .........
12.     Compute     q + y[k+1]*(r*z[k+11]+t*z[+12] ) and STORE into x[k+1].
13.     Increment the value of k by 2.
14.     goto STEP1
```

Figure 3.4: Example : Modification of Hydro Fragment by Basic Model

The last three kernels are the NASA 7 "FORTRAN" language Kernels. The command to generate their assembly codes is " f77 -O4 -S filename ". All the traditional registers in assembly programs are replaced by the Basic model of EReg to see if there are any improvement. The result contains three data :

1. Total number of instructions issued
2. Number of *LOAD* instructions issued
3. Number of *STORE* instructions issued.

The result is presented in table 3.1. Since the Basic model of the EReg aims at solving ambiguously aliasing problem, an improved result for EReg can be obtained when there are any aliasing problems in the tested kernels. We found that the kernels 1, 2, 6, 7, 8, 12, 14 and 15 have ambiguously aliasing problems. The kernel 14 exhibits the greatest performance improvement. The fortran source of kernel 14 is shown in Figure3.5

Since the testing program in NASA 7 Benchmark passed the parameters IS = -1, M = 128, M1 = 128 and N = 256 into the kernel 14, we found that there are two loops, which

	Percentage of instr remained	Percentage of *LOAD* instr remained	Percentage of *STORE* instr remained
1. Hydro fragment	87.6%	66.8%	100%
2. ICCG excerpt	77.8%	80.3%	100%
3. Inner product	100%	100%	100%
4. Banded linear equations	100.4%	100%	100%
5. Tri-diagonal elimination , below diagonal	100%	100%	100%
6. General linear recurrence equations	75.9%	66.7%	0.2%
7. Equation of state fragment	a) 100% b) 85.7%	a) 100% b) 66.8%	a) 100% b) 100%
8. ADI integration	80%	63.7%	100%
9. Integrate predictors	100%	100%	100%
10. Difference predictors	100%	100%	100%
11. First Sum	100%	100%	100%
12. First Difference	75.9%	50.05%	100%
13. Mxm.f	100%	100%	100%
14. Cff2td1.f	51.4%	25%	66.7%
15. Cholesky Decomposition / Substitution	96.66%	93.42%	83.58%

Table 3.1: Results obtained from the Basic Model

```
F40        DO 130 K = 1, M
F41        CT = X(K,II) - X(K,IM)
F42        X(K,II) = X(K,II) + X(K,IM)
F43        X(K,IM) = CT * CX
F44   130CONTINUE
F52      DO 140 K=1,M
F53        CT=X(K,I)
F54        X(K,I)=X(K,II)
F55        X(K,II)=CT
F56   140CONTINUE
```

Figure 3.5: Main Source Program Statements From Kernel 14 – CFFT2D2.F From NASA 7 Kernels

contain the instructions from F41 to F43 and from F53 to F55 in Figure3.5, accounting for 99.3% in the total number of instructions. As the compiler cannot make sure if CT is aliased with X(K,IM) or X(K,II), after the value of CT is changed in F41, the value of CT is stored into memory and the data of X(K,II) and X(K,IM) are loaded again before they are used as operands in F42. Since the Basic model of EReg can solve the ambiguous data aliasing problem implicitly, the value of CT and the values of X(K,IM) and X(K,IM) do not be required to be stored and loaded again respectively if the Basic model of EReg is used to replace the traditional registers. The locations for the data CT and CX are invariant in the loop of F41-F43. If they are aliased with X(K,IM) and X(K,II), the values of CT and CX will be changed. Hence, the data of CT and CX are loaded before they are used as operands in F43. After the Basic model of EReg is used to eliminate all the operations due to aliasing problem in the loop of F41-F43, the total number of instructions is decreased to 52.9% only. After considering the loop of F53-F55, the *LOAD* operation on the data CT can also be eliminated by solving the aliasing problem between it and X(K,II). The total number of instructions of the kernel finally remained is only 51.4%.

In kernels 1,2, and 12, the Basic model of EReg cannot reduce the instructions by consider only one iteration only, but it reduces them after considering the previous or the next iterations. For example, in kernel 2 of Figure 3.6, x[k+1] refer to the same data as that referred by x[k-1] in next iteration. Hence, after solving the aliasing problem by EReg, the data of x[k-1] in next iteration does not has to be loaded so that the number of *LOAD* instructions can be reduced.

```
for (k=ipnt+1;k<ipntp;k=k+2) {
        i++;
        x[i]=x[k]-v[k]*x[k-1]-v[k+1]*x[k+1];
}
```

Figure 3.6: Main Source Program Statements From Kernel 2 - ICCG exerpt From Livermore Loop Benchmark Kernels

It seems surprised that nearly all the write instructions are eliminated in kernel 6 of Figure 3.7. The reason is that if w[i] is aliased with b[k][i], error may result if the value of w[i] is not stored immediately after each w[i]+=b[k][i]*w[(i-k)-1] performed. However, it will be possible to move the *STORE* instruction of w[i] outside loop2 if the EReg is used.

```
for (i=1;i<n;i++)
        for (k=0;k<i;k++)
                w[i]+=b[k][i]*w[(i-k)-1];
```

Figure 3.7: Main Source Program Statements From Kernel 6 – GENERAL LINEAR From Livermore Loop Benchmark Kernels

```
        temp = x[k-1];
for (j=4;j<n;j=j+5) {
        temp -= x[lw]*y[j];
        lw++;
}
        x[k-1]=y[4]*temp;
```

Figure 3.8: Main Source Program Statements From Kernel 4 – BANDED LINEAR EQUATIONS From Livermore Loop Benchmark Kernels

The kernel 4 of Figure 3.8 is a very good example to illustrate the fact that the operation of Basic model of EReg sometime may increase the number of instructions on the contrary. There is a variable TEMP can be directly loaded from x[k-1]. If the EReg is used, the EReg representing TEMP will have same address value as that of the EReg representing x[k-1]. When the value of TEMP is changed later, the change of the value of the EReg representing x[k-1] will cause an wrong result. One instruction is added to remedy this situation by loading the value of x[k-1] into another EReg, say ER1, and then, the content of ER1 is moved into the variable TEMP.

From the table 3.1, there are two results in kernel 7 of Figure 3.9. The reason is that there are two code scheduling algorithms in this kernel. The first algorithm (ie.7a)

```
for (k=0;k<n;k++) {
        x[k]= u[k]+r*(z[k]+r*y[k])+
        t*(u[k+3]+r*(u[k+2]+r*u[k+1])+
        t*(u[k+6]+r*(u[k+5]+r*u[k+4])));
}
```

Figure 3.9: Main Source Program Statements From Kernel 7 – EQUATION OF STATE FRAGMENT From Livermore Loop Benchmark Kernels

rearrange the execution order of *LOAD* instructions such that the operations of 12 *LOAD* instructions in the innermost loop are changed from memory-EReg to EReg-EReg due to the data coherence mechanism of EReg. These instructions are the loading instructions for the array elements of u[k], u[k+1], u[k+2], u[k+3], u[k+4] and u[k+5]. If the second algorithm (i.e. 7b) is employed, 6 *LOAD* instructions can be eliminated. 2 *LOAD* operations will change from memory-EReg to EReg-EReg due to data coherence property of Basic model. Since it is difficult to justify which algorithm is better, both the result are shown in the table 3.1.

```
for ( kx=1 ; kx<3 ; kx++ ){
    for ( ky=1 ; ky<n ; ky++ ) {
        du1[ky] = u1[nl1][ky+1][kx] - u1[nl1][ky-1][kx];
        du2[ky] = u2[nl1][ky+1][kx] - u2[nl1][ky-1][kx];
        du3[ky] = u3[nl1][ky+1][kx] - u3[nl1][ky-1][kx];
        u1[nl2][ky][kx]=
            u1[nl1][ky][kx]+a11*du1[ky]+a12*du2[ky]+a13*du3[ky] + sig*
            (u1[nl1][ky][kx+1]-2.0*u1[nl1][ky][kx]+u1[nl1][ky][kx-1]);
        u2[nl2][ky][kx]=
            u2[nl1][ky][kx]+a21*du1[ky]+a22*du2[ky]+a23*du3[ky] + sig*
            (u2[nl1][ky][kx+1]-2.0*u2[nl1][ky][kx]+u2[nl1][ky][kx-1]);
        u3[nl2][ky][kx]=
            u3[nl1][ky][kx]+a31*du1[ky]+a32*du2[ky]+a33*du3[ky] + sig*
            (u3[nl1][ky][kx+1]-2.0*u3[nl1][ky][kx]+u3[nl1][ky][kx-1]);
    }
}
```

Figure 3.10: Main Source Statements of Kernel 8 : ADI INTEGRATION From Livermore Loop Benchmark Kernels

The reduction of the instruction in the kernel 8 of Figure 3.10 can be achieved by the standard procedure of the basic model of EReg. The standard procedure of the basic

```
        DO 6 I = 0, NRHS
                DO 7 K = 0, N
                DO 8 L = 0, NMAT
8                       B(I,L,K) = B(I,L,K) * A(L,0,K)
        DO 7 JJ = 1, MIN (M, N-K)
                DO 7 L = 0, NMAT
7                       B(I,L,K+JJ) = B(I,L,K+JJ) - A(L,-JJ,K+JJ) *B(I,L,K)
C
        DO 6 K = N, 0, -1
                DO 9 L = 0, NMAT
9                       B(I,L,K) = B(I,L,K) * A(L,0,K)
        DO 6 JJ = 1, MIN (M, K)
                DO 6 L = 0, NMAT
6                       B(I,L,K-JJ) = B(I,L,K-JJ) - A(L,-JJ,K) * B(I,L,K)
C
        RETURN
        END
```

Figure 3.11: Main Source Statements of Kernel 15 : CHOLSKY.F From NASA7 Benchmark Kernels

model of EReg sometimes may yield different number of instructions with traditional registers for the same operation. For example, memory is frequently used as an extension of EReg file in using traditional registers. If the value of index register for loop controling is used and its value is changed during the looping, it must be stored and loaded in performing the operation of loop controling. If EReg is used, such *STORE* and *LOAD* operations are eliminated. Hence, although the kernel 15 of Figure 3.11 does not have data aliasing problem, the number of instructions decreases after applying the standard procedure of changing the traditional register by the Basic model of EReg.

3.6 Temporary Storage Problem in Basic Model

3.6.1 Introduction

The main purpose of the structure is tried to eliminate the ambiguous aliasing problem. It has an address field which can be used to identify if two names are aliased with each other or not. However, there are some difficulties encountered. The reason is that at the present moment, memory spaces are frequently used as an extension of the register file such as the case in the MXM program from NAS Kernel Benchmark which is compiled by the command " f77 -O4 -S -Sun4 filename " . From the assembly code, we found that

ld [%fp-132], %l3 ... Line 63

```
and     ld     [%fp-132],     %l5     ... Line 91
```

where l3 represents the index J+1 while l5 represent K in the program. If the traditional registers are changed by ERegs, the value of l3 and l5 will be bound to be the same due to the coherence property of ERegs. Since every EReg has an address field which contains the memory address of the entity represented by this EReg, if an EReg is used as a storage similar to the temporary register. Then, the compiler is required to find some special reserved memory addresses for this EReg to use.

Basically, there are two approaches in the variations of BASIC model to handle the temporary storage problem. The first approach is to allow traditional registers coexisting with ERegs so that the traditional registers can be used to act as temporary storage. The second approach is to adapt some compiling techniques to overcome the temporary storage problem. Several solutions are listed out as follows.

If the first approach is used, there are two solutions :

1. One more bit called Temporary bit (or T-bit) is added such that when this bit is set, all the special property including coherence property are void. From the view point of hardware support, it is simply to disconnect those ERegs, whose T-bit is on, from the circuit of coherence electrically. Whenever a temporary storage is required, one of the ERegs is selected and its t-bit is set to ON.

2. The register file contains two kinds of registers – traditional registers and ERegs. For example, R1 - R16 are traditional registers while R17 - R32 are ERegs. The feature of ERegs will be able to handle the problem of aliasing. On the other hand, the traditional registers can handle the above temporary storage problem.

If the second approach is used, the technique to handle the problem of

```
        ld     [%fp-132],     %l3     ... Line 63
and     ld     [%fp-132],     %l5     ... Line 91
```

is to use one EReg more. Then, the instructions will change to

```
ld                          [%fp-132], %l1
( setting temporary )       %l3
( setting temporary )       %l5
mov                         %l1,%l3
mov                         %l1,%l5
```

There are three solutions in this approach to set the temporary addresses to the ERegs %l3 and %l5.

1. One is to use *LOAD* instructions to perform *LOAD* operation on these ERegs from some pre-defined temporary memory address so as to load the temporary addresses into the address field of these ERegs .
2. Another is to use a new instruction, say UPDATE, which can set the content in the address field so that the compiler assign a temporary address to these ERegs.
3. The last one is that there is a instruction TEMP which is specially created to assign a temporary memory address to these ERegs automatically.

3.6.2 Discussion on the Solutions

1. T-bit

   By adding a T-bit to each EReg in Basic model, we can solve the Temporary EReg problem easily. The mechanism of this new structure can be described as follows. The function of T-bit is to indicate whether the EReg is used to act as a Temporary EReg or Data EReg. However, the T-bit must be set before the EReg is used.

   For example, T-bit = 0 represents EReg being used as a temporary EReg while T-bit = 1 represents EReg being used as a normal EReg. When T-bit of an EReg is set to 0, the coherence ability of the EReg will be eliminated such that the EReg act like a traditional register. Hence, the Temporary storage can be handled easily.

The advantage of this method is that it provide a simple means to handle the problem of temporary storage, but it has an overhead of increasing the size of each EReg by 1 bit. Since the t-bit of each EReg are preset such that temporary register with t-bit = 1 hold temporary values only and datan EReg with t-bit = 0 used to hold data operand only. Normally, the process to set this t-bit occur infrequently. If the address field of the EReg cannot be accessed and operated by arithmetic and logic instructions. The address fields will be wasted if t-bit indicate the EReg to be temporary.

However, if the address field of the EReg can be accessed and operated by arithmetic and logical instructions, then, each EReg may be organized as a set of two traditional registers with a T-bit in which one is used as the address field and the other is used as the data field. The simple way to group the traditional registers into sets is by the use of the register names. If the name for the two registers are the same except the last bit, they belong to the same set. The one with last bit equal to 1 is used to store the data while the other with last bit equal to 0 is used to store the address. The T-bits of the ERegs are group together to form an special register called TR. The value of this special register can be modified just as an ordinary register. The structure can be organized as shown in Figure 3.12.

When the *LOAD* operation is performed, the data will be loaded into the specific register which would be represented as the data field and if the t-bit of this set is OFF, the address of the data is loaded into the other register within the set which represents the address field. If the t-bit of this set is ON, each EReg will be discomposed into two traditional registers.

The process to access a register is as follows. When a program access the register with register name Ri, the machine will check the i/2th bit of TR. If the bit is OFF, then the registers R_i and R_{i+1} act as an EReg. That's means the last bit of EReg name is zero and so only even EReg names are valid. If the bit is ON, then the value of register R_i is accessed.

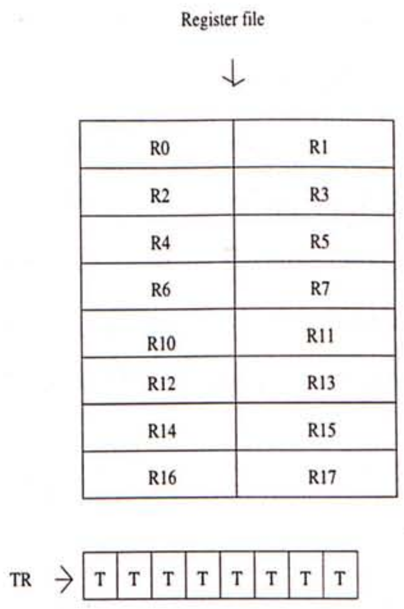

Figure 3.12: Second basic variated structure

If t-bit is OFF, the double word *LOAD* operation can be operated as follows. Supposing that the destination is ER_i. Then, the data of the lower order word would be loaded into the data register R_{i+1} while the address of the lower order word would be loaded into the address register R_i. In the meantime, the data of the higher order word and the address of the higher order word would be loaded into the data register R_{i+3} and the address register R_{i+2} respectively. The operation of quadword *LOAD* operation is similar.

If t-bit is ON, the single word *LOAD* operation just operated on the specific register only. But if the *LOAD* operation is a double word *LOAD* operation , the lower-order word is loaded into the specific register and the higher order word is loaded into the register which is just follow the specific register. It is the responsibility of programmer or compiler to ensure that they have selected the right registers to store their data. It is error to store data acrossing both *Temporary* and *Data EReg*.

2. Traditional registers coexists with ERegs.

The method of register allocation will be same as the traditional method except the case that if some variables are ambiguously aliased with each other, they will be assigned to ERegs. The arrangement of traditional register and ERegs can be described as follows. R1 - R16 are traditional registers while R17 - R32 are ERegs. When the load operation occur on, the data will be loaded into the specific register or the data field of ERegs. If the destination is EReg, the address of the data will also be loaded into the address field of ERegs. This design may be effective since the associativity mechanism of ERegs may be expensive to be implemented. Less EReg to be implemented means less money to be used.

3. Use *LOAD* instruction to assign a temporary address to ERegs

   Some space in cache/memory system are reserved for temporary register. Whenever an EReg is required to store temporary data, its address field must be loaded by a pre-defined temporary address using a load instruction. Since temporary data does not require any *STORE* operations, the value of these temporary addresses can be invalid. The advantage of using this method is that it is simplest method to implement. At the present moment, when the number of traditional registers is not enough, memory is often used as an extension of the register file. However, this method cannot handle this issue well because it requires one more EReg to act as the immediate storage so that the content of address field of the EReg used for temporary storage does not changed. Since every *LOAD* instruction would require a *LOAD* operation plus an associative update among the EReg file, this method is not efficient.

4. Use a new instruction, say UPDATE, to assign a temporary address to ERegs

   Whenever an EReg is required to store temporary data, a pre-defined temporary address is loaded into its address field by using an UPDATE instruction, "UPDATE ERi DST_ADDRESS". If this UPDATE instruction is reserved for assigning temporary address, then no associative compare and update is required to keep data coherence in EReg file because the temporary address for each EReg used to hold temporary data is unique. As previous method, the temporary data does not require any *STORE* operations such that the value of these temporary addresses can be invalid. The assignment of the temporary address in this method being handled by the compiler. The operation of this method is faster the operation of *LOAD* instruction, but it introduced one more instruction into instruction set.

5. Use a instruction " TEMP ERn " assign a temporary memory address to the register implicitly.

   Whenever an EReg is required to store temporary data, its address field must be loaded by a pre-defined temporary address using this TEMP instruction. As previous method, the temporary data does not require any *STORE* operations such that the value of these temporary addresses can be invalid.

   The advantage of this method is that it reduce the trouble for the user in setting the value of temporary address. In fact, the EReg name can be used as part of temporary address value of that EReg by using the EReg name to add with a certain memory address base, and then this address can be used as the address value of this EReg. It is a simple method to create an unique memory address for each EReg and implement this TEMP instruction. This method over the method 4 is that it does not require the compiler or assembly programmer to assign and to maintain the bookkeeping of temporary address so that the chance to produce error is reduced.

3.7 Introduction of ADM Model

Each EReg can be used to hold temporary value or data value. The data value referred to the data loaded from the memory for execution while temporary value refer to the value of indexing or intermediate result of the arithmetic and logical operation. If an EReg hold data value, and this data value is changed, such changed value are bound to be stored back to main memory later by using *STORE* instructions explicitly. Then, if there exists another bit called M-bit to indicate if each EReg has been modified or not, we can adapt a implicit storing method. The basic idea of implicit storing method is that if the M-bit of ERegs are ON (i.e. one), the datum value of these ERegs will be stored automatically once it is replaced by another datum (just like the operation of write back policy).

The ADM model is the basic model with a M-bit. It aims at reducing both the number of *STORE* instructions and operations. Normally, no *STORE* instruction is required, all the data references will first be forwarded to EReg file to see if the data exist in it. If there are several aliased ERegs to be replaced, in fact, only the value of one of them storing back to the memory has been sufficient since they have the same address and data values. Hence, only one of the M-bit of these aliased ERegs (i.e. they have same address value) is required to maintain the state of "ON".

When an EReg is replaced and its M-bit is OFF, there are two implications. Firstly, the value of the EReg hasn't been modified. Secondly, although the value of the EReg may has been modified, there are other aliased ERegs. In both these cases, no implicit storing operation is required when replacing this EReg. However, if its M-bit is ON, it means that the current data of the EReg has been modified such that the corresponding memory element should be updated before the new data is forwarded to the EReg if it doesn't have other aliased ERegs. If it has other aliased ERegs in the EReg file, we can delay the implicit storing operation by changing the M-bit of one of those ERegs to ON instead of updating the memory content before the new data is forward to the EReg. In conclusion, all *STORE* instructions can be eliminated and implicit storing operation is only required to perform when the M-bit of the EReg is ON and it doesn't have any aliased ERegs.

However, even if the implicit storing is supported, the explicit *STORE* instruction and operation may still be required. This can be explained in the following way. Supposing that there is a variable and the EReg copy of this variable has been modified. When some other devices (e.g. other processors) access the memory location of this variable, an explicit *STORE* operation must be performed such that an updated data content can be provided to these devices. This is the reason that we must support the *STORE* instruction in our ADM model.

The ADM model is a superset of Basic model such that ADM model can reduce the

LOAD/STORE instructions and operations in the case of data aliasing. The next section is concerned about the architectural and operational details of ADM model.

3.8 Architectural and Operational Detail of ADM Model

The memory structure of the ADM model is shown in Figure 3.13. It has two fields/bit – address field, data field and M-bit. The instructions to change the data content of EReg can be classified into three types of operations. They are *LOAD operation*, *STORE operation*, *arithmetic and logical operations.* The operational details are discussed on these three types of operations.

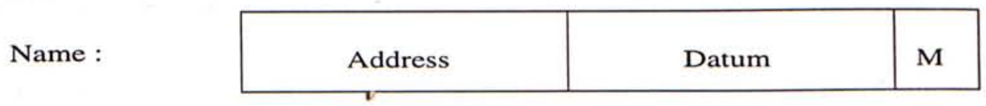

Figure 3.13: Memory structure of ADM model

Operational Details

Case 1 : *LOAD* Instruction :

When a *LOAD* instruction for loading the content of the memory address *ADDRESS_A* into an EReg, e.g. ERi, is executed:

e.g. *LOAD ADDRESS_A*, ERi

Preprocessing

Before loading, if the M-bit of ERi is ON, an comparison in a fully associative way is made to see if there is any matches between the address value of ERi and the content of the address field of all ERegs in the EReg file.

1. If any matches are found, the M-bit of one of these matched ERegs is set to ON.

2. If no match is found, the datum value of the ERi is stored back to the memory by using the address value of ERi as the memory address.

Loading

- M-bit of ERi is set to OFF
- The content of the address field of ERi is set to the address value *ADDRESS_A*.
- The address value *ADDRESS_A* is checked against the content of the address field of all ERegs in the EReg file simultaneously.

  – If a match is found between the address value *ADDRESS_A* and the content of the address field of some ERegs, the content of the datum field of these ERegs are copied into the datum field of ERi.

  – If no match is found in the EReg file, the memory content with the address *ADDRESS_A* is copied from memory into the datum field of ERi.

Case 2 : *STORE* Instruction :

When a *STORE* instruction storing data of an EReg, e.g. ERi, to the memory address "New_address" is executed :

STORE ERi, "New_address"

- The address "New_Address" is checked against the content of address field of all ERegs in the EReg file,

  – If a match is found between the address "New_Address" and the content of address field of some ERegs, both the memory content with address "New_Address" and the content in the datum field of these ERegs are updated by the content of the address field of ERi. The M bit of these ERegs are set to OFF.

  – If no match is found in the EReg file, only the content with memory address "New_Address" is updated.

- the M-bit of ERi is set to OFF.

Case 3 : Arithmetic and Logical Operation :

When an arithmetic or logical operation with destination EReg,e.g ERi, is executed :

- The content of the datum field of the ERi is updated as usual.
- The M-bit of the EReg will be set to ON
- The content of the address field of the ERi is read and compared associatively with the content of the address field of all other ERegs. If they match, the content of the datum field of these ERegs are updated by that of ERi and their M-bit are set to OFF.

3.9 Discussion

3.9.1 File Partition

As previous model, if the fully associative logic is too expensive, the EReg file can be divided into sets,e.g. four ERegs per set. All the ambiguous aliased objects must be assigned to same set. If all the ERegs have been filled up, when there is a *LOAD* operation, an EReg is chosen to be replaced in order to accommodate the new data. The operation is just as same as the normal *LOAD* operation discussed previously. The first step is to check if the M bit of this EReg is ON or OFF. If new data has aliased variables in ERegs within the set, *LOAD* operations are not required; while If the replaced EReg has aliased ERegs within the set, th implicit *STORE* operation is not required.

3.9.2 *STORE* Instruction

The reason of including *STORE* instruction in ADM model is that when there are some devices to read the memory content, an explicit *STORE* operation must exist to deal with this situation. However, there is another approach that the EReg file can be included in the memory hierarchy such that other devices can get the correct value from the EReg file when they read the memory hierarchy.

If any memory accesses will first search the EReg file to see whether the required data has been inside the EReg file, all *STORE* instructions can be completely reduced. This can be explained as follows. If the value of the destination address of the *STORE* instruction has been loaded into EReg file, the *STORE* instruction can be eliminated.

3.10 Example for ADM Model

From the simulation result of basic model, the fifth kernel – Tri-diagonal elimination, below diagonal does not obtain any improvement from the operation of basic model. However, if the ADM model is used, the implicit storing feature can improve the performance by reducing the total number of instructions 34.3%. The main "C" statements and its corresponding pseudo assembly code of the fifth kernel is shown in Figure 3.14. Since every result x[i] must be stored explicitly and each x[i-1] is loaded only once, there is no improvement after using basic model. However, if the implicit storing is supported, all the *STORE* instructions can be reduced as shown in Figure 3.15. After rearrange the coding, the line 6, line 12 and line 18 is removed while the line 3 is moved out of the loop and line 24 is changed from *STORE* to LOAD instructions. One tricky technique in the program segment of Figure 3.15 is that the EReg used to store the value of x[i+3] in line 24 is the same as that used in line 3. Hence, the content in ER1 can further be used in line 5 during the next iteration.

3.11 Simulation Results

As previous model, there are totally 15 kernels used in the simulating the effect of the ADM model of EReg. The source code of these 15 kernels are placed in Appendix A. The first 12 are Livermore Loop "C" language kernels. All kernels will be compiled by cc compiler into assembly programs in SPARC machine environment with the command

Main "C" statements :

```
for (i=1;i<n;i++)
x[i]=z[i]*( y[i]-x[i-1])
```

Pseudo Assembly Version :

```
0.     STEP1
1.     LOAD        y[i]
2.     LOAD        z[i]
3.     LOAD        x[i-1]
4.     Compute     ( y[i] - x[i-1] )
5.     Compute     z[i]*(y[i]-x[i-1])
6      STORE       x[i]
7.     LOAD        x[i]
8.     LOAD        y[i+1]
9.     Compute     ( y[i+1] - x[i] )
10.    LOAD        z[i+1]
11.    Compute     z[i+1]*(y[i+1]-x[i])
12     STORE       x[i+1]
13.    LOAD        x[i+1]
14.    LOAD        y[i+2]
15.    Compute     ( y[i+2] - x[i+1] )
16.    LOAD        z[i+2]
17.    Compute     z[i+2]*(y[i+2]-x[i+1])
18.    STORE       x[i+2]
19.    LOAD        x [i+2]
20.    LOAD        y [i+3]
21.    Compute     ( y[i+3] - x[i+2] )
22.    LOAD        z[i+3]
23.    Compute     z[i+3]*(y[i+3]-x[i+2])
24.    STORE       x[i+3]
25.    Increment   the value of i by 4.
26.    goto STEP1
```

Figure 3.14: Example : Tri-Diagonal Elimination, Below Diagonal from Livermore Kernels

```
3.    LOAD        x[i-1], ER1
0.    STEP1
1.    LOAD        y[i]
2.    LOAD        z[i]
7.    LOAD        x[i]
4.    Compute     ( y[i] - x[i-1] )
5.    Compute     z[i]*(y[i]-x[i-1])
      and use ER1 to be destination of the computation.
6.    ....        ( remove ) ... ..
8.    LOAD        y[i+1]
13    LOAD        x[i+1], ER2
9.    Compute     ( y[i+1] - x[i] )
10.   LOAD        z[i+1]
11.   Compute     z[i+1]*(y[i+1]-x[i])
      and use ER2 to be destination of the computation.
12.   ....        ( remove ) ... ..
14.   LOAD        y[i+2]
19.   LOAD        x[i+2],ER3
15.   Compute     ( y[i+2] - x[i+1] )
16.   LOAD        z[i+2]
17.   Compute     z[i+2]*(y[i+2]-x[i+1])
      and use ER2 to be destination of the computation.
18.   ....        ( remove ) ... ..
20.   LOAD        y[i+3]
24.   LOAD        x[i+3], ER1
21.   Compute     ( y[i+3] - x[i+2] )
22.   LOAD        z[i+3]
23.   Compute     z[i+3]*(y[i+3]-x[i+2])
      and use     ER1 to be destination of the computation
25.   Increment   the value of i by 4.
26.   goto STEP1
```

Figure 3.15: Example : Modification of Tri-Diagonal Elimination, Below Diagonal from Livermore Kernels by ADM model

	Percentage of instr. remained	Percentage of *LOAD* instr remained	Percentage of *STORE* instr remained
1. Hydro fragment	87.6%	66.8%	100%
2. ICCG excerpt	77.8%	80.3%	100%
3. Inner product	100%	100%	100%
4. Banded linear equations	99.4%	100%	0
5. Tri-diagonal elimination , below diagonal	65.7%	100.07%	0
6. General linear recurrence equations	75.9%	66.7%	0.2%
7. Equation of state fragment	85.7%	66.8%	100%
8. ADI integration	76%	63.7%	50.2%
9. Integrate predictors	100%	100%	100%
10. Difference predictors	100%	100%	100%
11. First Sum	77.9%	100.2%	0.1%
12. First Difference	75.9%	50.05%	100%
13. Mxm.f	93.5%	100%	0.02%
14. Cff2td1.f	38.5%	25%	0
15. Cholesky Decomposition / substitution	88.16%	93.42%	0.07%

Table 3.2: Results obtained from the ADM Model

" cc -O4 -S filename ". The last three kernels are the NASA 7 "FORTRAN" language Kernels. The command to generate their assembly codes is " f77 -O4 -S filename ". All the traditional registers in assembly programs are replaced by the ADM model of EReg to see if there are any improvement. The result contains three data :

1. Total number of instructions issued

2. Number of *LOAD* instructions issued

3. Number of *STORE* instructions issued.

The result is presented in table 3.2. The extra feature in ADM model aims at reducing the number of *STORE* instructions by introducing implicit storing. There are some situations that the implicit storing cannot or should not replace the explicit *STORE* instructions. For example, "st %f30,[%o2]"

if there are some devices waiting to access the memory content of the address [%o2], this explicit *STORE* instruction cannot be changed to implicit storing. Or if the memory content with address [%o2] hasn't been loaded into ERege file before, the only way is to load the data of [%o2] first, say " ld [%o2],%f20 ", and then change the instruction of "st %f30,[%o2] to " mov %f30,%f20 ". Since the value of %f20 is modified by the mov instruction, the value of %f20 will be written back to memory later. However, it may involve too much overhead so that it may not be beneficial to change the explicit *STORE* instruction to implicit storing. If there is an instruction to change the value in the address field of EReg, we may use it to change the content of address field of %f30 instead of executing an explicit *STORE* instruction "st %f30,[%o2]".

In fact, even if the content of destination address has been loaded, it may still not beneficial to change the explicit *STORE* instruction to implicit storing. Let's consider the following two cases shown in figures 3.16 and 3.17

```
ld                  [%i5],     %f6
ld                  [%i5+2],   %f7
                ...
                ...
fmuld               %f6,   %f8,   %f10
                ...
                ...
st                  %f10,  [%i5]
st                  %f10,  [%i5+2]
```

Figure 3.16: Case a in reducing *STORE* instruction

For the case (a), it may be beneficial to change the explicit *STORE* instruction to implicit storing instruction as shown in figure 3.18.

For the case (b), we found that a mov instruction with an implicit *STORE* is needed to replace an explicit *STORE* instruction. Thus, it may be still not beneficial to change the explicit *STORE* operation to implicit *STORE* operation as shown in figure 3.19.

```
fmuld                  %f6,   %f8,   %f10
                 ...
                 ...
ld                     [%i5],      %f6
ld                     [%i5+2],  %f7
                 ...
                 ...
st                     %f10,  [%i5]
st                     %f10,  [%i5+2]
```

Figure 3.17: Case b in reducing *STORE* instruction

```
ld                     [%i5],      %f6
ld                     [%i5+2],  %f7
                 ...
                 ...
fmuld                  %f6,   %f8,   %6
```

Figure 3.18: Result of reducing *STORE* instructions in case a

```
fmuld                  %f6,   %f8,   %f10
                 ...
                 ...
ld                     [%i5],      %f6
ld                     [%i5+2],  %f7
                 ...
                 ...
mov                    %f10,  %f6
mov                    %f10,  %f7
```

Figure 3.19: Result of reducing *STORE* instructions in case b

The result of ADM model in Figure 3.2 is discussed as follows. In kernels 1,2,7,9 and 12, the destinations of the *STORE* instructions haven't been loaded into the EReg file before the execution of *STORE* instructions. Hence, it isn't beneficial to implement implicit storing such that no improvement is obtained. In kernel 3, no *STORE* instruction cause no improvement. In kernel 10, since it may not beneficial to change the *STORE* instructions by mov instructions followed by implicit storings (case b) and hence no improvement in reducing the number of *STORE* instructions. In kernels 4,13,14 and 15, since the destinations have been loaded before, *STORE* instruction can be reduced (case a). In kernels 5 and 11, the destinations can be loaded before the *STORE* instructions by using the technique of unrolling. The *STORE* instructions then can be reduced (case a). In kernel 6, the destination is invariant of the loop if the EReg is used. Thus, the *STORE* instructions have been reduced by the coherence property of EReg. In kernel 8, some destinations can be loaded before the *STORE* instructions,(du1[ky],du2[ky] and du3[ky]). As before, the *STORE* instructions for du1[ky],du2[ky] and du3[ky] then can be reduced (case a).

3.12 Temporary storage Problem of ADM Model

3.12.1 Introduction

As described in previous model, it is very important for the idea of TEMPORARY EReg which is used to store the data that any temporary EReg should not be used to store data operand, since once the value of this temporary EReg is changed, it means that the memory content represented by this temporary EReg is changed. In order to cope with this problem of temporary EReg, the solutions discussed in the section 3.6 of page 29 can be used in ADM model. However, some modifications should be made.

3.12.2 Discussion on the Solutions

1. T-bit

Implicit Storing is disabled when the T-bit is ON. As similar to the organization in Basic model, if the each EReg may be organized as a set of two traditional registers with two bits - T-bit and M-bit. One traditional register is used as the address field and another traditional register is used as the data field. The simple way to group the traditional registers into set is by using the register names. If the name for the two registers are the same except the last bit, they belong to the same set. The one with last bit equal to 1 is used to store the data while the other with last bit equal to 0 is used to store the address. All the T-bits and M-bits are grouped together to form two special registers TR and MR respectively, the structure is shown in Figure 3.20.

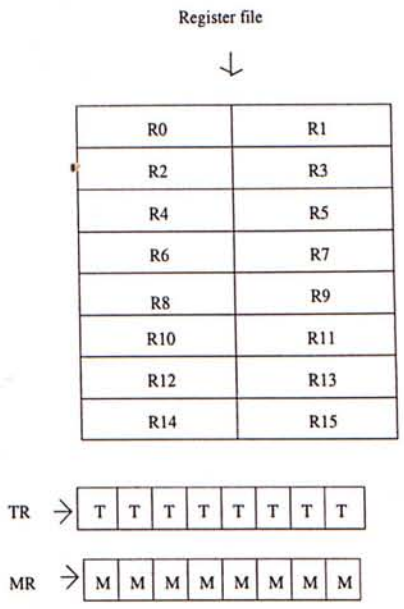

Figure 3.20: ADM second variant

2. Traditional registers coexists with ERegs :

   The operational details of this solution is the same as that in Basic model. The only difference is that the EReg here is ADM model while the EReg there is Basic model.

3. Use *LOAD* instruction to assign temporary addresses to ERegs

When the content of the EReg is changed, the M-bit of the EReg become ON. Then, the data value of the EReg will be stored back to the memory implicitly if the temporary variable represented by the EReg is replaced by another temporary variable. Hence, the temporary address assigned by *LOAD* instruction must be valid. Some spaces in cache/memory system are reserved for temporary storage. However, other aspects concerning this solution are just the same as that in Basic model.

4. Use a new instruction, say UPDATE, to assign temporary addresses to ERegs.

   This new UPDATE instruction is reserved for assigning temporary address. The temporary address is unique for each EReg. Moreover, the temporary addresses need not be valid. Then, no associative comparison and update is required. Hence, this solution is much better than the previous method 4 in ADM model. This solution does not cause implicit *STORE* operation. In other words, the M-bit of the EReg has no effect in this solution. This solution here is the same as that in Basic model.

5. Use a instruction " TEMP ERn " to assign a temporary memory address to the register implicitly.

   This solution is the same as that in Basic model. Whenever an EReg is required to store temporary data, its address field must be loaded by a pre-defined and unique temporary address using this TEMP instruction. As previous method, the temporary data does not cause any *STORE* operations such that the value of these temporary addresses can be invalid. In other words, the solution here is also the same as that in Basic model.

   The advantage of this method is that it can reduce the trouble of the user to set the value of temporary address. This method over the method 5 is that it does not require the compiler or the assembly programmer to assign and to maintain the bookkeeping of temporary address so that the chance to produce error is reduced.

Chapter 4

ADS Model and ADSM Model

4.1 Introduction of ADS Model

A cache memory is a well known mechanism used to reduce the average memory access latency. The main objective of data prefetching is to decrease the cache miss ratio so as to improve the overall performance. Usually, prefetch instruction is employed in conventional methods to identify the piece of data which may be used in the near future. According to this identification, the data is prefetched in advance before it is used. Hence, the cache is expected to hold not only the current working set, but also the future working set simultaneously.

In this chapter, we introduce a new hardware-based with software support prefetching method incorporated into the EReg. It requires a new instruction to indicate how and where to prefetch the data, but not those prefetched instructions in traditional methods. This new instruction is called SET_S which is used to set the value of the stride field of EReg. Chen[Che93] classified the data access patterns into four types - scalar, zero stride, constant stride and irregular. Since the coherence property of EReg eliminate the ambiguous data aliasing problem, the number of *LOAD* operations of scalar stride can be reduced. Our prefetching scheme is designed to generate an effective implicit data prefetching operation on the constant stride data access patterns.

The ADS model is an extended version of Basic model. The ADS model require one more field to store the stride value of the constant stride data access pattern. To illustrate the concept, we consider the following MXM – matrix multiplication in NASA 7 Benchmark Kernels in figure 4.1.

```
      DO 110 J=1,M,4
           DO 110 K=1,N
                DO 110 I=1,L
                C(I,K)=C(I,K)+A(I,J)*B(J,K)
                     +A(I,J+1)*B(J+1,K)+A(I,J+2)*(J+2,K)
                     +A(I,J+3)*B(J+3,K)
 110 CONTINUE
```

Figure 4.1: NASA MXM.f Kernel – Matrix Multiplication Problem

During the innermost loop, both the stride values of C(I,K) and A(I,J) are 4. If the block size is less than 4 bytes, reference to C(I,K) or A(I,J) causes data prefetching at every iteration (assume byte addressable machine is used). Since there are no reference access to B(J,K), no prefetching on this data is required.

4.2 Architectural and Operational Detail of ADS Model

The memory structure of the ADS model is shown in Figure 4.2. Its structure is an extended version of Basic model. It has three fields – address field, data field and stride field. The instructions to change the data content of EReg can be classified into three types of operations. They are *LOAD operation*, *STORE operation*, *arithmetic and logical operations*. The operational details are discussed on these three types of operations.

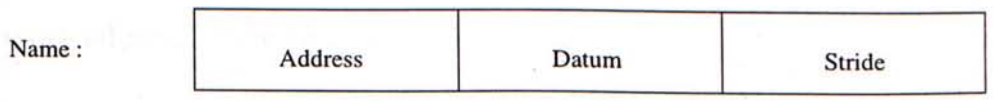

Figure 4.2: Memory structure of ADS model

Operational Details

Case 1 : *LOAD* Instruction :

When a *LOAD* instruction for loading the content of the memory address *ADDRESS_A* into an EReg, e.g. ERi, is executed:

e.g. *LOAD ADDRESS_A*, ERi

- The content of the address field of ERi is set to the address value *ADDRESS_A*.
- The address value *ADDRESS_A* is checked against the content of the address field of all ERegs in the EReg file simultaneously. – If a match is found between the address value *ADDRESS_A* and the content of the address field of some ERegs, the content of the datum field of these ERegs are copied into the datum field of ERi. – If no match is found in the EReg file, the memory content with the address *ADDRESS_A* is copied from memory into the datum field of ERi.
- If the content in STRIDE field is not zero, next data will be prefetched from main memory to cache using the prefetching address (STRIDE value + current address *ADDRESS_A*). The new created instruction :

  SET_S ERi,#no

  can set the stride value of ERi to #no. However, data prefetching is only executed if the intended data to be prefetched does not exist in the EReg file and Data Cache.

Case 2 : *STORE* Instruction :

When a *STORE* instruction storing data of an EReg, e.g. ERi, to the memory address "New_address" is executed :

STORE ERi, New_address

- The address "New_Address" is checked against the content of address field of all ERegs in the EReg file, - If a match is found between the address "New_Address" and the content of address field of some ERegs, both the content with memory address "New_Address" and the content in the datum field of these ERegs are updated by the content of the datum field of ERi. - If no match is found in the EReg file, only the memory content with address "New_Address" is updated.

Case 3 : Arithmetic and Logical Operation :

When an arithmetic or logical operation with destination EReg, e.g ERi, is executed :

- The content of the datum field of the ERi is updated as usual.
- The content of the address field of the ERi is read and compared associatively with the content of the address field of all other ERegs. If they match, the content of the datum field of these ERegs are updated by that of ERi

4.3 Discussion

4.3.1 Prefetching Priority

Every ERegs have a STRIDE field which store the offset between the prefetching address and current address. After each *LOAD* operation, the loading address will be added with the stride value of the EReg to produce an prefetching address which is used as a hint to load the data from the lower hierarchy's memory to cache memory so as to reduce cache miss ratio. Since data of the prefetched address is not in the current working set, the demand loading is more critical than the prefetching operation and hence the priority of prefetching operation is lower than that of demand *LOAD* operation. Moreover, the prefetching is only performed when the prefetched data does not exist in the cache.

4.3.2 Data Prefetching

Basically, the data prefetching mechanism is effective only if the prefetching data are the array elements with constant stride. However, if there exist a method treating the content of the address field to be zero such that the prefetching address is equal to the value in the STRIDE value, many dynamic data can be prefetched. The method is that the compiler check the data dependency firstly and then try to identify the next access address before the current *LOAD* instruction. If it is possible, the STRIDE value can be set by this next access address such that the prefetching operation can be performed after the current *LOAD* instruction. However, such dynamic data prefetching is usually poor as it cost one prefetch instruction for each prefetching process. The performance of the current method is better in reducing the cache miss ratio since it will only introduce one more SET_S instruction until the current stride value is changed. The figure 4.3 illustrates the additional operation of ADS model compared with Basic Model.

4.3.3 EReg File Splitting

As the previous models, when it is too expensive to build an fully associative EReg file, it is possible to split it into a number of sets. Only those within the same set are maintained coherent. Each set may contain four ERegs. All the ambiguous aliasing objects will be placed into same set. The replacement strategy and implicit storing issues can be just the same as that in the Basic model.

4.3.4 Compiling Procedure

There are some steps for the compiler to follow in order to make full use of the Data prefetching in ADS model :

1. Study the stride value of each *LOAD* operation in the kernels.

2. Mark stride value of all constant stride *LOAD* operation accesses, e.g array element access.

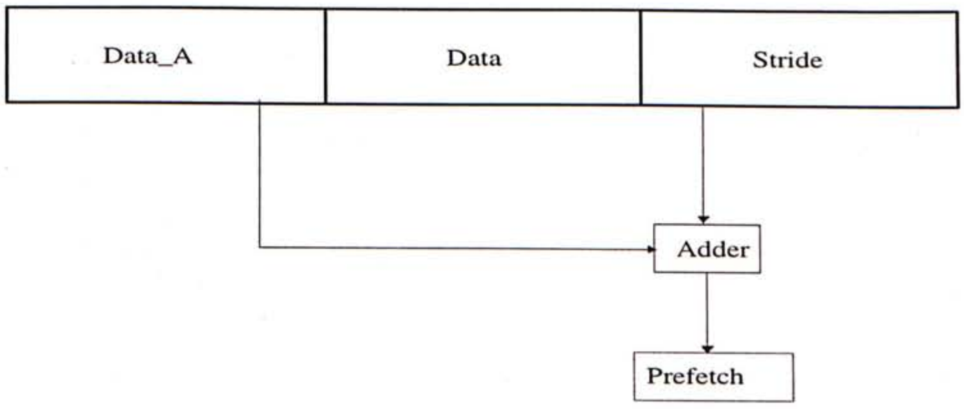

Figure 4.3: ADS Construction

3. Assign an EReg to each marked Data operand.

4. Use *SET* instructions to set the content of the stride field of these ERegs.

5. Insert these *SET* instructions into the program code before those marked *LOAD* instructions.

The next section will give an example on how to perform data prefetching by making use of the above compiling steps.

4.4 Example for ADS Model

The kernel of hydro fragment from the the Benchmark of Livermore Loop in Figure 3.3 of 23 is used to demonstrate the performance of ADS model. In the kernel of hydro fragment, z[k+10] & z[k+11] will be loaded for each k since there may be an aliasing problem between x[k] and z[k+11] or x[k] and z[k+12]. However, if ADS model of EReg is used, we can reschedule the coding as shown in Figure 4.4

Then, the total number of 4 loading instructions z[k+11], z[k+10], z[k+12] and z[k+11] in one looping will be decreased to only 2 loading instructions z[k+11] and z[k+12] with ONE instruction outside the looping "for (k=0;k<n;k++)". After the stride value is set in ER1, the prefetching operation is performed after each loading operation. Nearly all the data value can be prefetched before the operation of *LOAD* instruction. When

```
            SET_S        ER1,8 ; This value 8 represent the stride value of the array
            SET_S        ER2,8
    3.      LOAD         z[k+10]
    0.      STEP1
    4.      Compute      r*z[k+10]
    1.      LOAD         z[k+11]
    2.      Compute      t*z[k+11]
    5.      .....        ........
    6.      Compute      q + y[k]*(r*z[k+10]+t*z[+11] ) and STORE into x[k].
    9.      ( remove )
    10.     Compute      r*z[k+11]
    7.      LOAD         z[k+12] ( use same EReg as z[k+10] )
    8.      Compute      t*z[k+12]
    11.     .....        .........
    12.     Compute      q + y[k+1]*(r*z[k+11]+t*z[+12] ) and STORE into x[k+1].
    13.     Increment    the value of k by 2.
    14.     goto         STEP1
```

Figure 4.4: Example : Modification of Hydro Fragment by ADS Model

the data cache size is 8KByte, the block size is 16Byte and the data cache is a 2-way associative, the miss ratio after using this prefetching scheme is 0.95% while the miss ratio of no prefetching is 25.5%.

4.5 Simulation Results

There are totally 15 kernels used in the simulating the effect of the ADS model of EReg. The source code of these 15 kernels are placed in Appendix A. The first 12 are Livermore Loop "C" language kernels. All kernels will be compiled by cc compiler into assembly programs in SPARC machine environment with the command " cc -O4 -S filename ". The last three kernels are the NASA 7 "FORTRAN" language Kernels. The command to generate their assembly codes is " f77 -O4 -S filename ". All the traditional registers in assembly programs are replaced by the ADS model of EReg to see if there are any improvement. The result contains :

	Percentage of instr remained	Percentage of *LOAD* instr remained	Percentage of *STORE* instr remained
1. Hydro fragment	87.6%	66.8%	100%
2. ICCG excerpt	(a) 78.8% (b) 77.9%	(a) 80.3% (b) 80.3%	(a) 100% (b) 100%
3. Inner product	100.2%	100%	100%
4. Banded linear equations	104.3%	100%	100%
5. Tri-diagonal elimination , below diagonal	100.2%	100%	100%
6. General linear recurrence equations	75.9%	66.7%	0.2%
7. Equation of state fragment	a) 100.04% b) 85.8%	a) 100% b) 66.8%	a) 100% b) 100%
8. ADI integration	80%	63.7%	100%
9. Integrate predictors	100.048%	100%	100%
10. Difference predictors	100.038%	100%	100%
11. First Sum	100.18%	100%	100%
12. First Difference	76%	50.05%	100%
13. Mxm.f	100%	100%	100%
14. Cfft2d1.f	51.4%	25%	66.7%
15. Cholesky Decomposition / Substitution	96.66%	93.42%	83.58%

Table 4.1: Simulation Result for ADS Model

1. Total number of instructions issued

2. Number of *LOAD* instructions issued

3. Number of *STORE* instructions issued

4. Miss ratio with no prefetching

5. Miss ratio with prefetching provided in this model

The new *SET* instruction in ADS model is added outside the innermost loop of each kernels in the simulation. The effect of *SET* instruction on the number of instruction is very little. Hence, the number of instructions using ADS model is similar to that using Basic model as shown in the Table 4.1 and the Table 3.1 in Page 25. The reason for the reduction of instructions is the same for both the ADS model and Basic model.

The ADS model of EReg can prefetch data after the data is actually accessed. The method can be described as follows. All the kernels are compiled into assembly code. We study the stride value of each *LOAD* operation in the kernels. And then we insert the *SET* instruction to enable the function of implicit data prefetching. Normally, the assembly program codes are also required to be rewrited due to the different properties between ADS model and traditional register.

The simulation results is carried out on SPARC machine. The Shade Analyzers which from *Sun Microsystems*,Inc is used to simulate a cache. Since data prefetching only give an improvement in reducing data miss, the instruction cache were not considered in out simulation. Some policies on the simulated data cache are shown as follows:

- Least Recently Used adopt in Replacement Policy
- Write Back and Write Allocate adopt in Write Policy
- No Sub-block
- Only one level cache

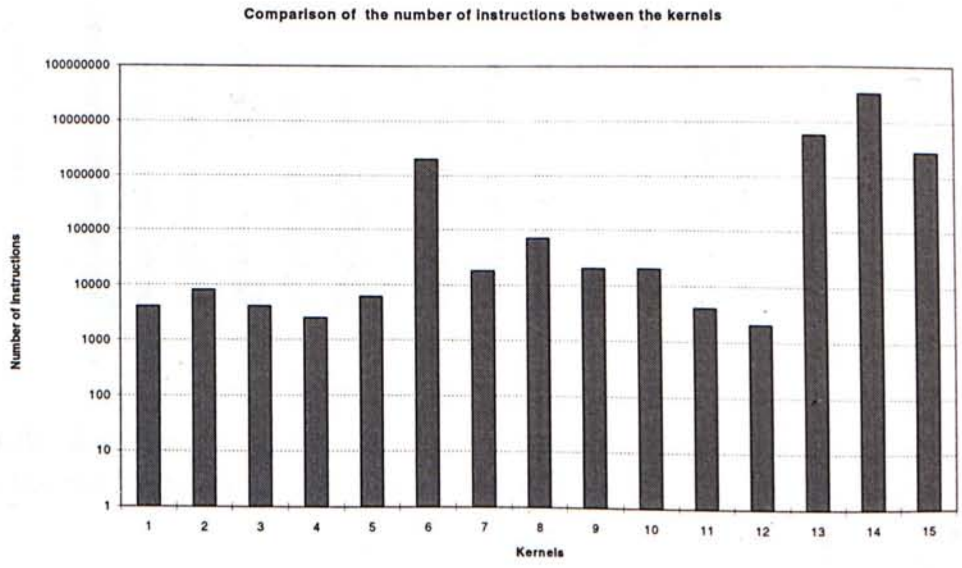

Figure 4.5: Comparison on the number of instructions between the kernels

The Figure 4.5 shows the comparison of the number of instructions between the kernels. By varying the parameters of cache size, block size and way associative, we obtain the miss ratios of no prefetching under different conditions for each kernel. Then, we modified the Shade Analyzers such that it can perform data prefetching. When the modified Shade Analyzers encountered each *LOAD* operations, the loading address are added with its corresponding stride value to produce the prefetched address. The prefetched data are moved into the data cache immediately. The data prefetching made by ADS model can be called *ERegs' prefetching*. For ERegs' prefetching, no memory delay is assumed such that all prefetching data are able to be prefetched into cache. In general, larger cache often give a better result (i.e lower miss ratio) for same amount of data.

The best performance of ERegs' prefetching is on the Kernel 8 with the cache configuration of cache size=8KB, block size=16KB and 4-way associative as shown in Figure 4.6. The miss ratio with no prefetch is 17.2% while that with ERegs' prefetching is 0.084%.

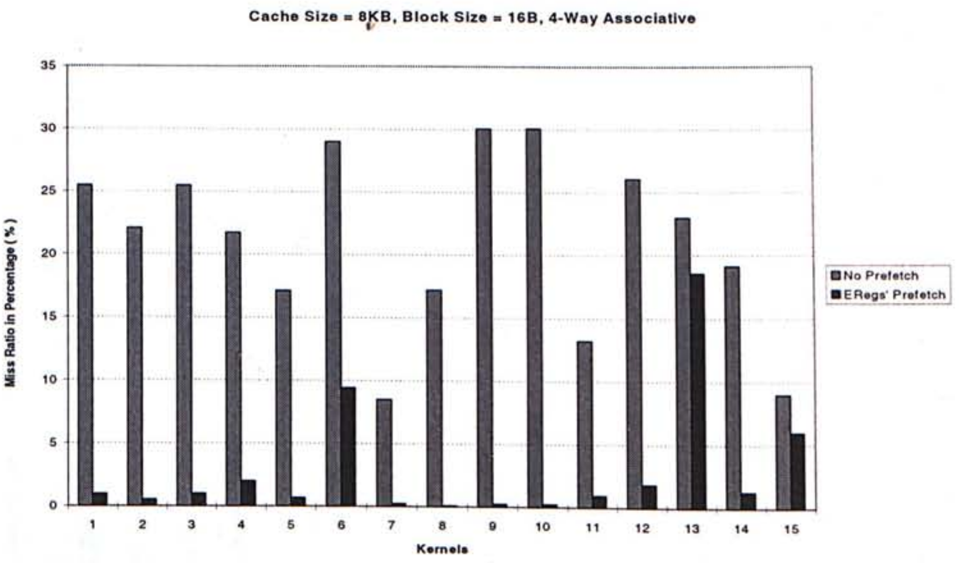

Figure 4.6: Miss Ratio comparison between the no prefetch and EReg's prefetch algorithms at the configuration : Cache Size = 8KB, Block Size = 16B, 4-Way Associative

The greatest miss ratio occur in the kernel 13 which is an matrix multiplication program from NASA7 Benchmark. At the configuration of cache size = 8KB, block size = 32B and 4-way associative as shown in Figure 4.7, the miss ratio of this kernel is 29.69%

with no prefetch and 27.48% with ERegs' prefetch. The reason for such high miss ratio can be explained as follows. There are three 2-dimensional arrays in kernel 13. They are A(256,128), B(128,64) and C(256,64). At the configuration of cache size = 8KB, block size = 32B and 4-way associative, the number of cache lines is 256 and the number of set is 64. At the innermost loop, there are five loadings on the array elements of A(I,J), A(I,J+1), A(I,J+2), A(I,J+3) and C(I,K). Since the size of each element is 8 byte, the elements of A(I,J), A(I,J+1), A(I,J+2), A(I,J+3) will always be mapped into the same set, causing a big cache pollution problem. The prefetched data of an element may flush out that of another element. And its referenced data loading from demand prefetch also may flush out that of another element. Generally, such pollution will happen at the following situation :

- There are several loading data in a loop.
- The difference between their loading addresses or prefetched addresses are multiple of number of cache lines, but the size of each set (i.e way associative) is smaller than the number of loading data.

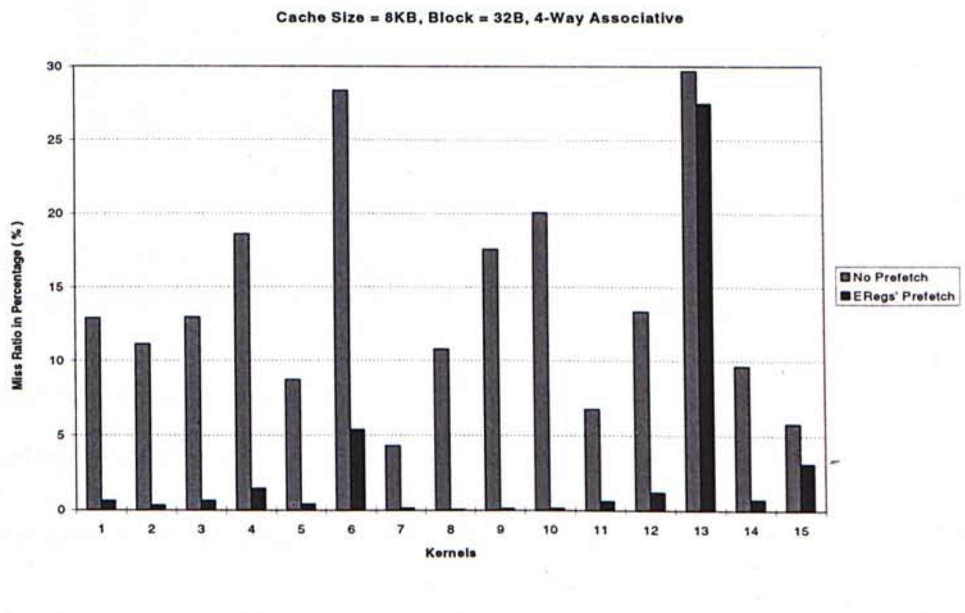

Figure 4.7: Miss Ratio comparison between the no prefetch and EReg's prefetch algorithms at the configuration : Cache Size = 8KB, Block Size = 32B, 4-Way Associative

The reasons are described as follows. Not only all the data from demand fetch will be mapped to same set, but all of prefetched data will also be mapped into same set. If the size of each set (i.e way associative) is smaller than the number of loading data. Their data will flush out with each other. In fact, even if the size of the set is equal to the number of loading data, the similar cache pollution still occur seriously. The loading and prefetching of C(I,K) in kernel 13 make the cache pollution worse. The similar pollution problem occur at the following configuration.

1. Cache size = 8KB, Block size = 16B, 2-way associative as shown in Figure 4.8.
2. Cache size = 8KB, Block size = 16B, 4-way associative as shown in Figure 4.6.

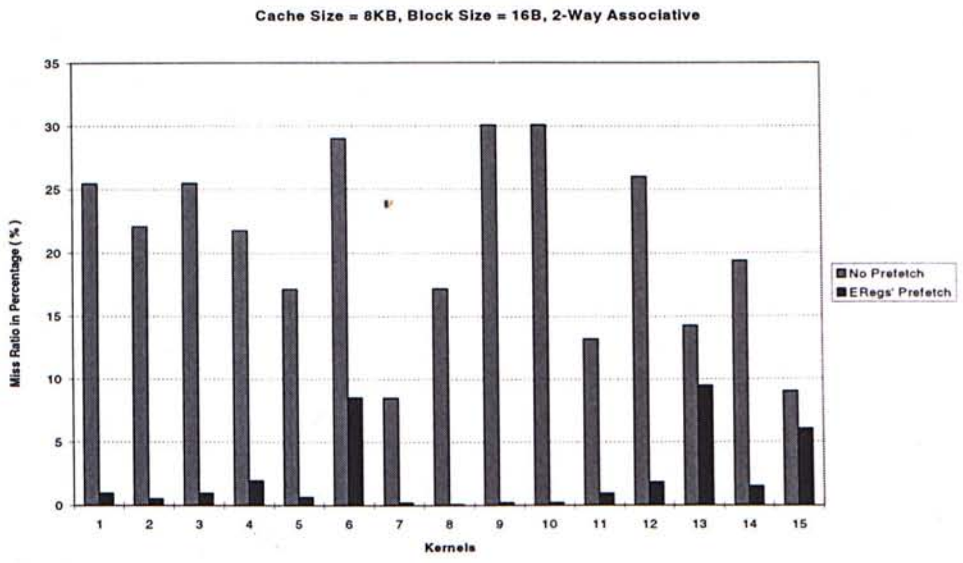

Figure 4.8: Miss Ratio comparison between the no prefetch and EReg's prefetch algorithm at the configuration : Cache Size = 8KB, Block Size = 16B, 2-Way Associative

Once the size of cache is changed from 8KB to 16KB, the above situation is changed. A much better performance is obtained at the configuration below :

1. Cache size = 16KB, Block size = 32B, 4-way associative as shown in Figure 4.9.
2. Cache size = 32KB, Block size = 32B, 4-way associative as shown in Figure 4.10.
3. Cache size = 32KB, Block size = 32B, 8-way associative as shown in Figure 4.11.

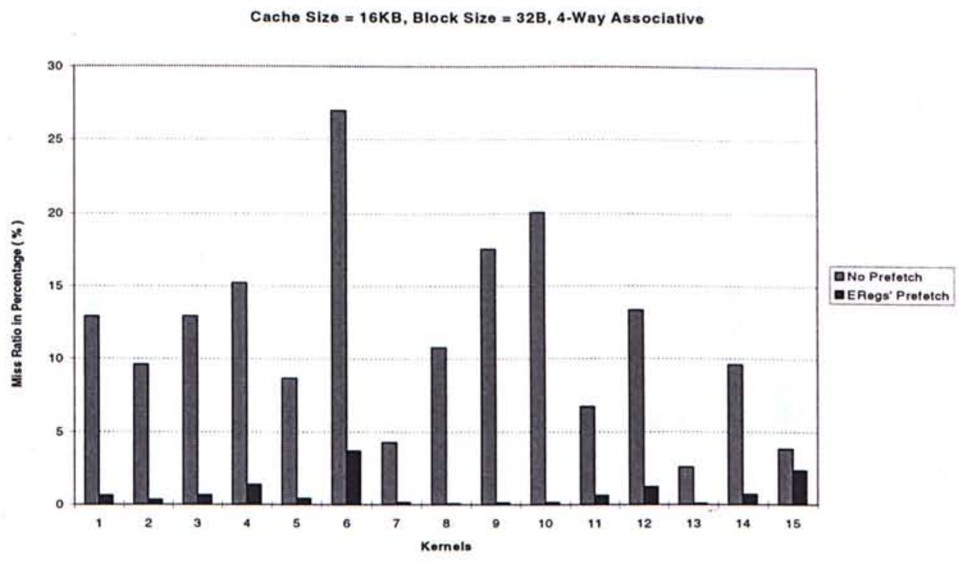

Figure 4.9: Miss Ratio comparison between the no prefetch and EReg's prefetch algorithms at the configuration : Cache Size = 16KB, Block Size = 32B, 4-Way Associative

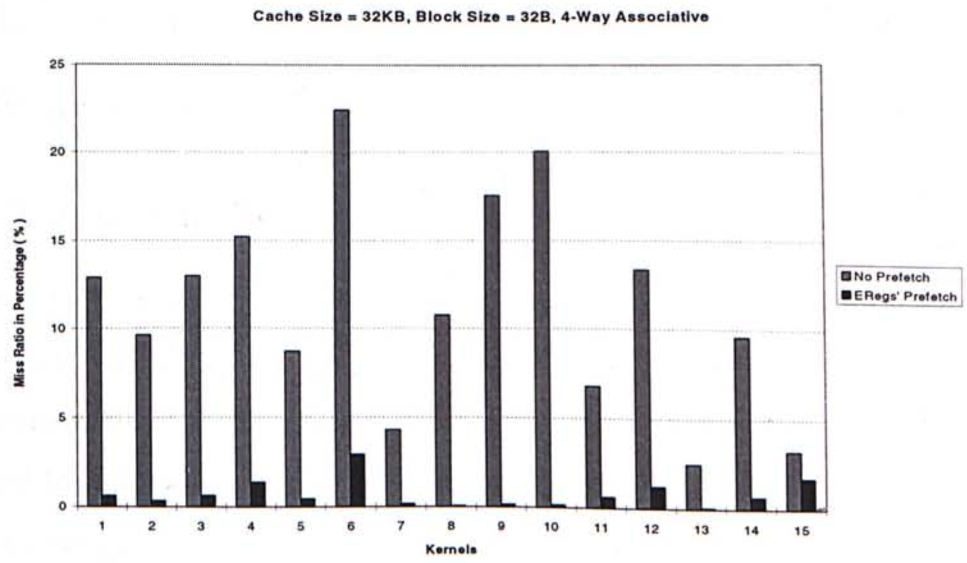

Figure 4.10: Miss Ratio comparison between the no prefetch and EReg's prefetch algorithms at the configuration : Cache Size = 32KB, Block Size = 32B, 4-Way Associative

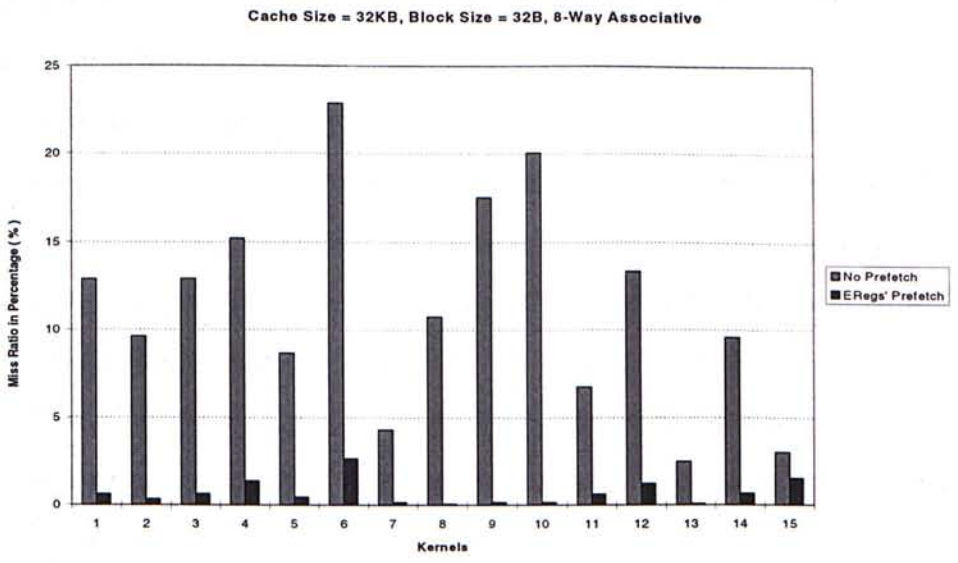

Figure 4.11: Miss Ratio comparison between the no prefetch and EReg's prefetch algorithms at the configuration : Cache Size = 32KB, Block Size = 32B, 8-Way Associative

All the miss ratio of EReg's prefetch at these configurations are below 0.08% while that of no prefetch at these configurations are below 2.6%.

4.6 Discussion on the Architectural and Operational Variations for ADS Model

4.6.1 Temporary storage Problem

As previous models, it is very important for the idea of TEMPORARY EReg which is used to store the data that any temporary EReg should not be used to store data operand, since once the value of this temporary EReg is changed, it means that the memory content represented by this temporary EReg is changed. In order to cope with the problem of temporary storage, the solutions discussed in the section 3.6 of page 29 can be used in ADS model. There are totally five solutions. However, the first solution of using T-bit requires some modification.

In the the solution of using T-bit to solve temporary storage problem, the modification are presented as follows. If the stride field and address field of the EReg can be accessed and operated by arithmetic instructions, each EReg can be organized as a set of three traditional registers with a t-bit which can be used to overcome the temporary storage problem as described in previous models. These three traditional registers are used to represents address field, data field and stride field. The simple way to group the traditional registers into set is by using the register names. Since each EReg contains three traditional registers. The number of register in register file should be multiple of three. If T-bit is OFF, 3 traditional registers are combined to be preform as an EReg in ADS model. All the T-bits of ERegs are grouped together to form an special register called TR. The structure is shown in figure 4.12.

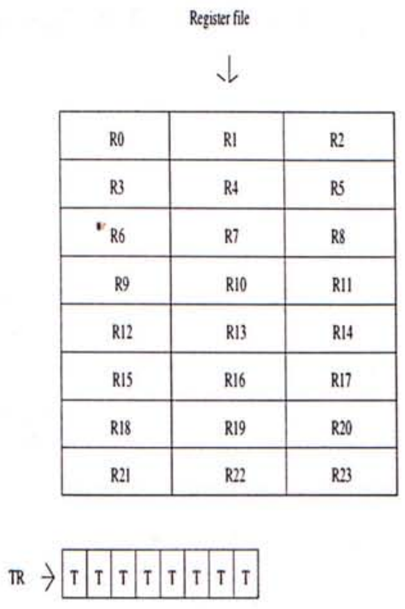

Figure 4.12: ADS second variated structure

4.6.2 Operational variation for Data Prefetching

The idea of the variation is to change the stride value dynamically instead of setting the stride value statically by using the new instruction – *SET*. Data is prefetched only when the stride value is stable. Hence, although the method to set the stride value is changed, the target of the data prefetching still aim at constant stride data access patterns. The mechanism is as follows. When a *LOAD* instruction is executed, difference between the loading address of data and the current data address is compared with the stride value stored in stride field. If they are equal, the prefetching operation is performed using the sum of the stride value and the loading address as the prefetched address for next data. If they are not equal, the stride value is updated by the difference between the loading address of data and the current data address.

This variation handle the data prefetching implicitly. Although it reduces the overhead of setting the stride value, it introduces a larger overhead in making decision on whether prefetching should be performed. Therefore, the method is not used.

4.7 Introduction of ADSM Model

Since each EReg has the address of the datum that it contains, if the value of the datum is changed, it is bound to be stored later by using a *STORE* instruction explicitly. Moreover, the same data may be stored many times during the looping. For example, there are six aliased ERegs from ER1 to ER6. If their values are modified, they will be stored explicitly before they are replaced by objects. Then, there may exist six explicit *STORE* operations. If the function of implicit Storing in ADSM model is used, the number of *STORE* operations will be reduced to one only.

From this point, the feature of implicit storing not only can change the explicit *STORE* to implicit *STORE*, but also the actual number of store operation can be reduced. Basically, the ADSM model is an extended version of ADS model. The implicit storing operation of ADSM model is similar to ADM model. The *STORE* instruction can be eliminated if the EReg file can be served as a part of memory hierarchy such that other memory references can first access the EReg file. In the following sections, a more detail description on the architectural and operational issues will be accounted.

4.8 Architectural and Operational Detail of ADSM Model

The memory structure of the ADSM model is shown in Figure 4.13. Its structure is the superset of ADS model and ADM model. It has four fields/bit — address field, data field, stride field and modified bit. The instructions to change the data content of EReg can be classified into three types of operations. They are *LOAD operation, STORE operation, arithmetic and logical operations.* The operational details are discussed on these three types of operations.

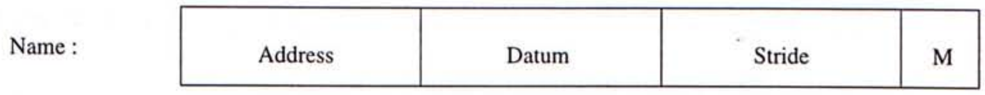

Figure 4.13: Memory structure of ADSM Model

Operational Model

Case 1 : *LOAD* Instruction :

When a *LOAD* instruction for loading the content of the memory address *ADDRESS_A* into an EReg, e.g. ERi, is executed:

e.g. *LOAD ADDRESS_A*, ERi

Preprocessing

Before loading, if the M-bit of ERi is ON, an comparison in a fully associative way is made to see if there is any matches between the address value of ERi and the content of the address field of all ERegs in the EReg file.

1. If any matches are found, the M-bit of one of these matched ERegs is set to ON.
2. If no match is found, the datum value of the ERi is stored back to the memory by using address value of ERi as the memory address.

Loading

- The M-bit of ERi is set to OFF.
- The content of the address field of ERi is set to the address value *ADDRESS_A*.
- The address value *ADDRESS_A* is checked against the content of the address field of all ERegs in the EReg file simultaneously.

  – If a match is found between the address value *ADDRESS_A* and the content of the address field of some ERegs, the content of the datum field of these ERegs are copied into the datum field of ERi.

  – If no match is found in the EReg file, the memory content with the address *ADDRESS_A* is copied from memory into the datum field of ERi.
- If the content in STRIDE field is not zero, next data will be prefetched from main memory to cache using the prefetching address (STRIDE value + current address *ADDRESS_A*). The new created instruction

SET_S ERi, #no

can set the stride value of ERi to #no. However, data prefetching is only executed if the intended data to be prefetched does not exist in the EReg file and Data Cache.

Case 2 : *STORE* Instruction :

When a *STORE* instruction storing data of an EReg, e.g. ERi, to the memory address "New_address" is executed :

STORE ERi, "New_address"

- The address "New_Address" is checked against the content of address field of all ERegs in the EReg file,

  – If a match is found between the address "New_Address" and the content of address field of some ERegs, both the content with memory address "New_Address" and the content in the datum field of these ERegs are updated by the content of the datum field of ERi. The M bit of these ERegs are set to OFF.

- If no match is found in the EReg file, only the content with memory address "New_Address" is updated.

Case 3 : Arithmetic and Logical Operation :

When an arithmetic or logical operation with destination EReg,e.g ERi, is executed :

- The content of the datum field of the ERi is updated as usual.
- The M-bit is set to ON
- The content of the address field of the ERi is read and compared associatively with the content of the address field of all other ERegs. If any matches occur, the content of the datum field of these ERegs are also updated by that of ERi and their M bits are set to OFF.

4.9 Discussion

In this model, the EReg will be equipped with one more bit. This bit will enable implicit storing such that the data in the EReg will be written back to the memory automatically. Moreover, if the EReg mechanism keep in contact with the data bus, the request of data can be forwarded from the EReg file rather than memory. Then, store instruction is not required. The figure 4.14 illustrate the additional operations compared with Basic Model.

4.10 Example for ADSM Model

From the result of basic model in section 3.4, the fifth kernel – *original elimination below diagonal* does not obtain any improvement from the operation of basic model. However, if the ADSM model is used, the implicit storing feature can improve the performance

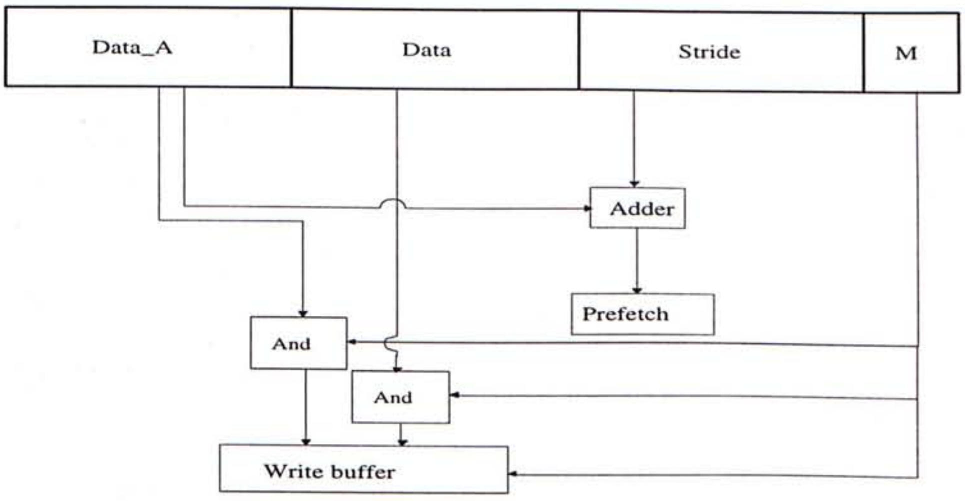

Figure 4.14: ADSM Construction

by reducing the total number of instructions 34.3% as shown in the Figure 4.15. If the feature of implicit storing (i.e ADSM) is supported, all the *STORE* instructions can be reduced as shown in the Figure 4.16.

Upon rearranging the coding, the line 6, line 12 and line 18 is removed while the line 3 is moved out of the loop and line 24 is changed from *STORE* to *LOAD* instructions. One tricky technique in above program segment is that the EReg used to store the value of x[i+3] in line 24 is the same as that used in line 3. Hence, the content in ER1 can further be used in line 5 during the next looping. Nearly all the data value can be preloaded before the operation of *LOAD* instruction.

4.11 Simulation Results

There are totally 15 kernels used in the simulating the effect of the ADSM model of EReg. The source code of these 15 kernels are placed in Appendix A. The first 12 are Livermore Loop "C" language kernels. All kernels will be compiled by cc compiler into assembly programs in SPARC machine environment with the command " cc -O4 -S filename ". The last three kernels are the NASA 7 "FORTRAN" language Kernels. The command to generate their assembly codes is " f77 -O4 -S filename ". All the traditional registers in assembly programs are replaced by the ADSM model of EReg to see if there are any

Main "C" statements :

```
for (i=1;i<n;i++)
x[i]=z[i]*( y[i]-x[i-1])
```

Pseudo Assembly Version :

```
0.     STEP1
1.     LOAD        y[i]
2.     LOAD        z[i]
3.     LOAD        x[i-1]
4.     Compute     ( y[i] - x[i-1] )
5.     Compute     z[i]*(y[i]-x[i-1])
6      STORE       x[i]
7.     LOAD        x[i]
8.     LOAD        y[i+1]
9.     Compute     ( y[i+1] - x[i] )
10.    LOAD        z[i+1]
11.    Compute     z[i+1]*(y[i+1]-x[i])
12     STORE       x[i+1]
13.    LOAD        x[i+1]
14.    LOAD        y[i+2]
15.    Compute     ( y[i+2] - x[i+1] )
16.    LOAD        z[i+2]
17.    Compute     z[i+2]*(y[i+2]-x[i+1])
18.    STORE       x[i+2]
19.    LOAD        x [i+2]
20.    LOAD        y [i+3]
21.    Compute     ( y[i+3] - x[i+2] )
22.    LOAD        z[i+3]
23.    Compute     z[i+3]*(y[i+3]-x[i+2])
24.    STORE       x[i+3]
25.    Increment   the value of i by 4.
26.    goto STEP1
```

Figure 4.15: Example : Tri-Diagonal Elimination, Below Diagonal

```
SET_S   ER1, 16 ; This value 16 represent the stride value of the array
SET_S   ER2, 8
SET_S   ER3, 8
SET_S   ER4, 16
3.      LOAD        x[i-1], ER1
0.      STEP1
1.      LOAD        y[i]
2.      LOAD        z[i]
7.      LOAD        x[i]
4.      Compute     ( y[i] - x[i-1] )
5.      Compute     z[i]*(y[i]-x[i-1])
        and use ER1 to be destination of the computation.
6.      ....        ( remove ) ... ..
8.      LOAD        y[i+1]
13      LOAD        x[i+1], ER2
9.      Compute     ( y[i+1] - x[i] )
10.     LOAD        z[i+1]
11.     Compute     z[i+1]*(y[i+1]-x[i])
        and use ER2 to be destination of the computation.
12.     ....        ( remove ) ... ..
14.     LOAD        y[i+2]
19.     LOAD        x[i+2],ER3
15.     Compute     ( y[i+2] - x[i+1] )
16.     LOAD        z[i+2]
17.     Compute     z[i+2]*(y[i+2]-x[i+1])
        and use ER2 to be destination of the computation.
18.     ....        ( remove ) ... ..
20.     LOAD        y[i+3]
24.     LOAD        x[i+3], ER1
21.     Compute     ( y[i+3] - x[i+2] )
22.     LOAD        z[i+3]
23.     Compute     z[i+3]*(y[i+3]-x[i+2])
        and use ER1 to be destination of the computation
25.     Increment the value of i by 4.
26.     goto STEP1
```

Figure 4.16: Example : Modification of Tri-Diagonal Elimination, Below Diagonal from Livermore Kernels by ADSM model

improvement. The result contains three data :

1. Total number of instructions issued
2. Number of *LOAD* instructions issued
3. Number of *STORE* instructions issued.

The result is shown in the table 4.2. ADSM model is the superset of ADS model and ADM model. The new instructions - *SET* used in data prefetching are placed outside the innermost loop while the feature of implicit storing usually can eliminate the store instructions in the innermost loop. Then, the number of instructions of ADSM model (shown in Table 4.2) in general is the same as that of ADM model (shown in Table 3.2 of Page 43. Although the feature of implicit storing can reduce the number of *STORE* instruction, it often only change the explicit *STORE* into implicit *STORE*. Hence, the result of data prefetching is similar to that of ADS model.

4.12 Discussion on the Architectural and Operational Variations for ADSM Model

4.12.1 Temporary storage Problem

As described in previous models - Basic model, ADM model and ADS model, it is very important for the idea of TEMPORARY EReg which is used to store the data that any temporary EReg should not be used to store data operand, since once the value of this temporary EReg is changed, it means that the memory content represented by this temporary EReg is changed. In order to cope with the problem of temporary storage, the solutions discussed in the section 3.12 of page 46 can be used in ADSM model. There are totally six solutions. However, the first solution of using T-bit requires some modification.

In the solution of using T-bit to solve temporary storage problem, the modification are presented as follows. If the stride field and address field of the EReg can be accessed

	Percentage of instr. remained	Percentage of *LOAD* instr remained	Percentage of *STORE* instr remained
1. Hydro fragment	87.6%	66.8%	100%
2. ICCG excerpt	(a) 78.8% (b) 77.9%	80.3%	100%
3. Inner product	100.2%	100%	100%
4. Banded linear equations	103.4%	100%	0
5. Tri-diagonal elimination , below diagonal	65.9%	100.07%	0
6. General linear recurrence equations	75.9%	66.7%	0.2%
7. Equation of state fragment	(a) 100.04% (b) 85.8%	(a) 100% (b) 66.8%	(a) 100% (b) 100%
8. ADI integration	76%	63.7%	50.2%
9. Integrate predictors	100.048%	100%	100%
10. Difference predictors	100.038%	100%	100%
11. First Sum	78.1%	100.2%	0.1%
12. First Difference	76%	50.05%	100%
13. Mxm.f	93.5%	100%	0.02%
14. Cff2td1.f	38.5%	25%	0
15. Cholesky Decomposition / substitution	88.16%	93.42%	0.07%

Table 4.2: Simulation Result for ADSM Model

and operated during arithmetic operations, each EReg may be organized as a set of three traditional registers with a T-bit and a M-bit. T-bit is used to overcome the temporary storage problem as described in ADM model while M-bit is used to perform implicit storing operation. These three traditional registers are used to represent address field, data field and stride field. The simple way to group the traditional registers into set is by using the register names. Since each EReg contains three traditional registers. The number of ERegs in EReg file should be a multiple of three. If T-bit is OFF, 3 traditional registers are combined to act as an EReg in ADSM model. All the T-bits and M-bits are grouped together to form two extra special registers. One is TR while the other is MR. The structure is shown in Figure 4.17.

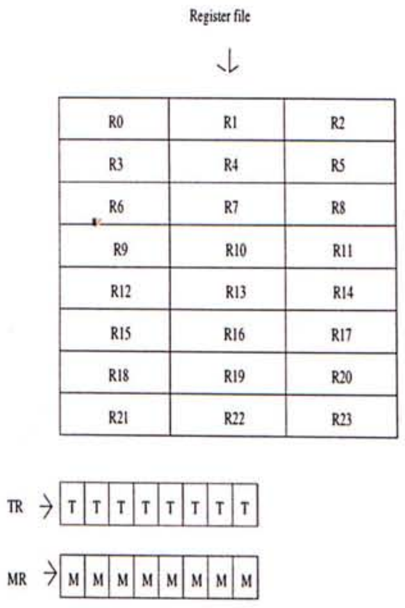

Figure 4.17: ADSM second variant

4.12.2 Operational variation for Data Prefetching

The same operational variation as ADS model can be used in this ADSM model for data prefetching. The basic idea is to change the stride value dynamically instead of setting the stride value statically by using the new instruction – *SET*. Data is prefetched only when the stride value is stable. Hence, although the method to set the stride value is changed, the target of the data prefetching still aim at constant stride data access patterns. The mechanism is as follows. When a *LOAD* instruction is executed, difference between the loading address of data and the current data address is compared with the stride value stored in stride field. If they are equal, the prefetching operation is performed using the sum of the stride value and the loading address as the prefetched address for next data. If they are not equal, the stride value is updated by the difference between the loading address of data and the current data address.

This variation handles the data prefetching implicitly. It reduces the overhead of setting the stride value, but it introduces an overhead in making decision on whether prefetching should be performed. As previous method, this method is not used.

Chapter 5

IADSM Model and IADSMC&IDLC Model

5.1 Introduction of IADSM Model

IADSM model aims at reducing the number of instruction to be issued. It is a hardware based method in which the explicit *LOAD* instruction can be changed to implicit *LOAD* instruction. From the discussions in previous models, it is obvious that the compiler can make 100% accurate prefetching address for constant stride array reference. IADSM model takes advantage of this feature and extends the ADSM model by allowing an implicit loading operation. To illustrate the concept, let us consider the the figures 5.1 and 5.2

```
200 L7700 :     ...
201             ...
202             load      ERi
203             add       ERi,ERj
204             ...
205             bne       L7700
```

Figure 5.1: Original Program listing

```
202                load        ERi
200 L7700 :        ...
201                ...
203                add         ERi,ERj
204                ...
205                bne         L7700
```

Figure 5.2: Program listing after using EReg in IADSM model

If IADSM model is used, the *LOAD* instruction can be moved out one more loop such that nearly all the number of explicit *LOAD* operations are changed to implicit *LOAD* operations for the looping. The design of ADSM model aims at improving the whole performance not by reducing the number of *LOAD* operations, but by reducing the time to issue such *LOAD* operations.

5.2 Architectural and Operational Detail of IADSM Model

The memory structure of the IADSM model is shown in Figure 5.3. Its structure is an extended version of ADSM model. It has four fields/bit – Instruction address field, data address field, data field, stride field and M-bit. The instructions to change the data content of EReg can be classified into three types of operations. They are *LOAD operation*, *STORE operation*, *arithmetic and logical operations.* The operational details are discussed on these three types of operations.

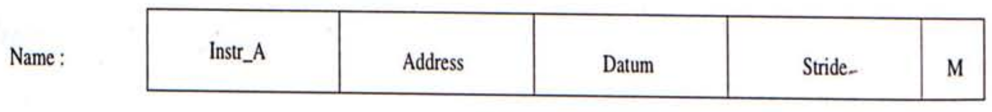

Figure 5.3: IADSM structure

Operational Details

The content of the program counter will always compare with the content of the

INST_A field of all ERegs in a fully associative way. If there is any match within ERegs, the content of the datum field of these ERegs are first stored using the address value of their address fields if their M-bits are ON. And the stride values of these ERegs are added up with their corresponding ADDRESS values and two times of ADDRESS values to produce the implicit loading addresses and prefetching addresses respectively. The INST_A field of an EReg, e.g. ERi, can be set by the instruction. However, if the value in INST_A field is zero, the operation of Implicit Loading is void.

SET_INST ERi,#address

Case 1 : *LOAD* Instruction :

When a *LOAD* instruction for loading the content of the memory address itADDRESS_A into an EReg, e.g. ERi, is executed as follows:

LOAD *ADDRESS_A, ERi*

Preprocessing

Prior to loading, if the M-bit of ERi is ON, an comparison in a fully associative way is made to see if there is any matches between the address value of ERi and the content of the address field of all ERegs in the EReg file.

1. If any matches are found, the M-bit of one of these matched ERegs is set to ON.
2. If no match is found, the datum value of the ERi is stored back to the memory by using address value of ERi as the memory address.

Loading

- The M-bit of ERi is set to OFF.
- The content of the address field of ERi is set to the address value *ADDRESS_A*.

- The address value *ADDRESS_A* is checked against the content of the ADDRESS field of all ERegs in the EReg file simultaneously.

  – If a match is found between the address value *ADDRESS_A* and the content of the address field of some ERegs, the content of the datum field of these ERegs are copied into the datum field of ERi.

  – If no match is found in the EReg file, the memory content with the address *ADDRESS_A* is copied from memory into the datum field of ERi.

- If the content in STRIDE field is not zero, next data will be prefetched from main memory to cache using the prefetching address (STRIDE value + current address *ADDRESS_A*). The new created instruction

  SET_S ERi, #no

  can set the stride value of ERi to #no.

Case 2 : *STORE* Instruction :

When a *STORE* instruction storing data of an EReg, e.g. ERi, to the memory address "New_address" is executed :

```
STORE        ERi,"New_address"
```

- The address "New_Address" is checked against the content of address field of all ERegs in the EReg file,

  – If a match is found between the address "New_Address" and the content of address field of some ERegs, both the content with memory address "New_Address" and the content in the datum field of these ERegs are updated by the content of the datum field of ERi. The M bit of these ERegs are set to OFF.

  – If no match is found in the EReg file, only the content with memory address "New_Address" is updated.

Case 3 : Arithmetic and Logical Operation :

When an arithmetic or logical operation with destination EReg, e.g. ERi, is executed :

- The content of the datum field of the ERi is updated as usual.
- M bit is ON.
- The content of the address field of the ERi is read and compared associatively with the content of the address field of all other ERegs. If they match, the M bit of these ERegs are set to OFF and the content of the datum field of these ERegs are also updated by the datum value of ERi.

5.3 Discussion

5.3.1 Implicit Loading

The implicit loading provided by IADSM model should be used carefully. When the implicit loading is not required, the INST_A field must be set to zero. The behaviour of implicit *LOAD* operation is just the same as that of a normal *LOAD* operation. That is, the implicit loading address is checked against address fields inside the EReg file such that if there is a successful matching, the target data is loaded from the matched EReg instead of the target memory location.

The limitation of this implicit loading feature is that it can support constant stride array access only. Before entering the loop, the stride fields of the ERegs, which are used to store data and perform implicit loading operation, must be set. This should be done easily by the compiler. Figure 5.4 illustrates the additional operation apart from that in basic Model.

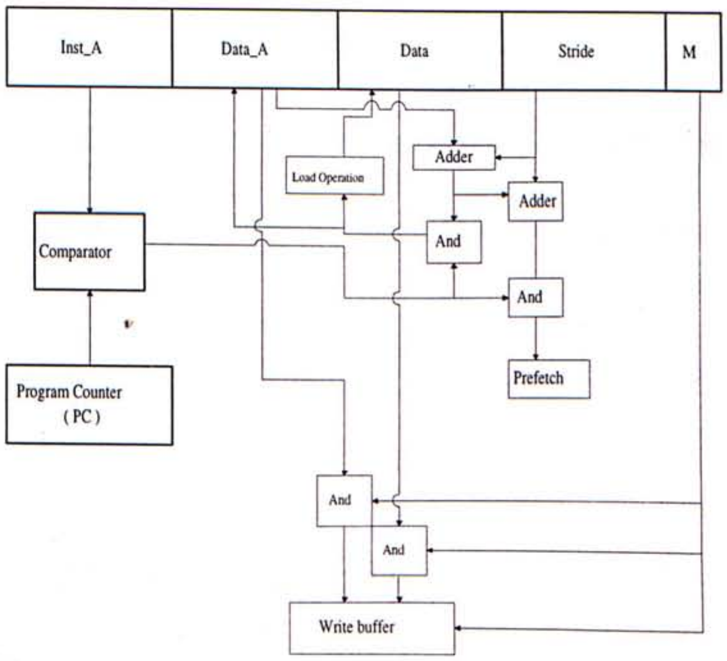

Figure 5.4: IADSM Construction

5.3.2 Compiling Procedure

In order to utilize the implicit loading feature of IADSM model, there are some steps for the compiler to follow :

1. Identify the constant stride *LOAD* instructions inside the loop and their corresponding stride values.

2. Move these *LOAD* instructions out of the loop.

3. For each EReg, set its stride field to the stride value of the corresponding constant stride *LOAD* instruction.

4. For each EReg, mark its earliest point where the next data can be loaded into it.

5. For each EReg, set the instruction address of its earliest point into its instruction field.

The earliest point for an EReg is the point following instruction where this EReg is lastly referenced during the loop.

5.4 Example for IADSM Model

The first kernel - Hydro Fragment from the livermore loop benchmark is used as an example as shown in Figure 5.5. The array elements of z[k+10] & z[k+11] are loaded for each k since there may be an aliasing problem between x[k] and z[k+11] or x[k] and z[k+12]. However, if IADSM model of EReg is used, we can reschedule the coding as shown in Figure 5.6. From the simulation result, only 56.6% of the instructions are remained.

Hydro fragment :

```
for ( k=0;k<n;k++)
x[k]=q+y[k]*(r*z[k+10]+t*z[k+11]);
```

Pseudo assembly version :

```
0.    STEP1
1.    LOAD        z[k+11]
2.    Compute     t*z[k+11]
3.    LOAD        z[k+10]
4.    Compute     r*z[k+10]
5.    .....       .......
6.    Compute     q + y[k]*(r*z[k+10]+t*z[+11] ) and STORE into x[k].
7.    LOAD        z[k+12]
8.    Compute     t*z[k+12]
9.    LOAD        z[k+11]
10.   Compute     r*z[k+11]
11.   .....       .........
12.   Compute q + y[k+1]*(r*z[k+11]+t*z[+12] ) and STORE into x[k+1].
13.   Increment the value of k by 2.
14.   goto        STEP1
```

Figure 5.5: Example : Hydro Fragment

```
3.     LOAD         z[k+10], R1
1.     LOAD         z[k+11], R2
       SET_S        R1, 8; This value 8 represent the stride value of the array
       SET_S        R2, 8
       SET_INST     R1,2; This value 2 represent the instruction address of the
                       following program segment
       SET_INST     R2,8This value 8 represent the instruction address of the
                       following program segment
0.     STEP1
4.     Compute      r*z[k+10] ( use EReg R1 )
2.     Compute      t*z[k+11] ( use EReg R2 )
5.     .....        .... ...
6.     Compute      q + y[k]*(r*z[k+10]+t*z[+11] ) and STORE into x[k].
9.     ( remove )
10.    Compute      r*z[k+11] ( use EReg R2 )
8.     Compute      t*z[k+12] ( use EReg R1 )
11.    .....        .... ...
12.    Compute      q + y[k+1]*(r*z[k+11]+t*z[+12] ) and STORE into x[k+1].
13.    Increment    the value of k by 2.
14.    goto         STEP1
```

Figure 5.6: Example : Modification of Hydro Fragment from Livermore Kernels by IADSM model

5.5 Simulation Results

There are totally 15 kernels used in the simulating the effect of the IADSM model of EReg. The source code of these 15 kernels are placed in Appendix A. The first 12 are Livermore Loop "C" language kernels. All kernels will be compiled by cc compiler into assembly programs in SPARC machine environment with the command " cc -O4 -S filename ". The last three kernels are the NASA 7 "FORTRAN" language Kernels. The command to generate their assembly codes is " f77 -O4 -S filename ". All the traditional registers in assembly programs are replaced by the IADSM model of EReg to see if there are any improvement. The result contains three data :

1. Total number of instructions issued
2. Number of *LOAD* instructions issued
3. Number of *STORE* instructions issued.

The number of instructions reduced by IADSM model are shown in Figure 5.1. Since IADSM model extends the ADSM model by changing the explicit loading operations into implicit loading operations, the following will discuss how the performance improvement can be achieved in changing from the ADSM model to IADSM model. By comparing the performance difference between IADSM model in Figure 5.1 and ADSM model in Figure 4.2, we found that the greatest performance improvement is achieved by kernel 15. The percentage of the instructions is reduced from 88.16% in ADSM model to 16.6%.IADSM model. From the report [Cha95], after the number of instruction of kernel 4 reduced by ADSM model are :

1. Total number of instructions issued : 8130288
2. Number of *LOAD* instructions issued : 2185320
3. Number of *STORE* instructions issued : 616

If the IADSM model is used, the number of instructions are :

	Percentage of instr remained	Percentage of *LOAD* instr remained	Percentage of *STORE* instr remained
1. Hydro fragment	56.6%	0.35%	100%
2. ICCG excerpt	(a) 42.1% (b) 40.5%	(a) 1.9% (b) 1.9%	(a) 100% (b) 100%
3. Inner product	42.3%	0.4%	100%
4. Banded linear equations	50.4%	1.2%	0
5. Tri-diagonal elimination , below diagonal	21.3%	0.5%	0
6. General linear recurrence equations	24.8%	0.4%	0.0001%
7. Equation of state fragment	a) 50.2% b) 50.1%	a) 0.15% b) 0.117%	a) 100% b) 100%
8. ADI integration	38.2%	2.06%	50.2%
9. Integrate predictors	52.6%	0.23%	100%
10. Difference predictors	60.5%	0.145%	100%
11. First Sum	18.8%	100%	100%
12. First Difference	48.9%	0.2%	100%
13. Mxm.f	23.9%	0.29%	2.17%
14. Cfft2d1.f	16.2%	0	0
15. Cholesky Decomposition / Substitution	16.6%	0.4%	0.066%

Table 5.1: Results for IADSM Model

1. Total number of instructions issued : 1526200
2. Number of *LOAD* instructions issued : 9320
3. Number of *STORE* instructions issued : 616

Comparing with ADSM model, IADSM model can further reduce the number of *LOAD* instructions by changing the operation of them into implicit operations. It may be surprised that the number of instructions reduced are much more than the total number of *LOAD* instructions. In fact, the reason is very simple. Each *LOAD* instruction usually require several instructions to compute the loading address and to make an EReg available to store the new data. When IADSM model change explicit *LOAD* instructions into implicit *LOAD* operations, not only the *LOAD* instructions are eliminated, but their relevant instructions are also eliminated.

The Main Fortran Statements of kernel 15 is shown in Figure 3.11 of page 3.11. The above statements account for over 97% of the total number of instructions of the kernel 15 - Cholsky.f from NASA Benchmark. All the data B(I,L,K), B(I,L,K+JJ) and B(I,L,K-JJ) have been loaded before they are stored into memory. Therefore, if the implicit storing is applied, nearly all the *STORE* instructions are reduced. Therefore, ADSM can reduce the total number of instructions to 88.16%. From the report [Cha95], every computation instruction requires association with several instructions. These instructions include

1. compute the address of operands
2. load the operands.

In the kernel 15, these instructions is about 10 times for one computation. If IADSM model is applied, all the *LOAD* and their relevant instructions in the above statements are eliminated such that only those computation instructions remained in the program. Hence, the total number of instructions can be decreased to 16.6%.

Although all the *LOAD* instructions and their relevant instructions in all kernels are eliminated very effectively by the the IADSM model, the performance of the kernels are not the same. It is because the performance result usually depend on those innermost loops of the kernels. When there are some instructions which cannot be eliminated by IADSM model, a diversify result is obtained. For example, the *STORE* instructions in kernel 10 should not be changed into implicit storing as discussed in ADM model. Then, the number of instructions remained in IADSM model not only include the computation instructions, but also the *STORE* instructions. This is also the reason of why the smallest performance improvement is achieved in the kernel 10.

5.6 Temporary Storage Problem of IADSM Model

As described in previous models – Basic model, ADM model, ADS model and ADSM model, it is very important for the idea of TEMPORARY EReg which is used to store the data that any temporary EReg should not be used to store data operand, since once the value of this temporary EReg is changed, it means that the memory content represented by this temporary EReg is changed. In order to cope with the problem of temporary storage, the solutions discussed in the section 3.12 of page 46 can be used in this IADSM model. There are totally six solutions. However, the first solution of using T-bit requires some modification. If the Instruction field, stride field and address field of the EReg can be access and operated by arithmetic instructions, each EReg may be organized as a set of four traditional registers plus a T-bit and a M-bit. The T-bit is used to overcome the temporary storage problem as described in previous model while the M-bit which is used to enable implicit storing. These four traditional registers are used to represents instruction field, address field, data field and stride field. The simple way to group the traditional registers into set is by using the register names. For example, 8 bits is used to address registers. The first six bits are used to select the set (i.e. EReg) while the last two bits are used to define the fields within the EReg. Since each EReg contains four traditional registers, the number of registers in register file should be a multiple of four plus two. If T-bit is OFF, four traditional registers are combined to be preform as an EReg in IADSM model. All the T-bits and M-bits are grouped together to form two special registers called TR and MR respectively. The structure is shown in figure 5.7.

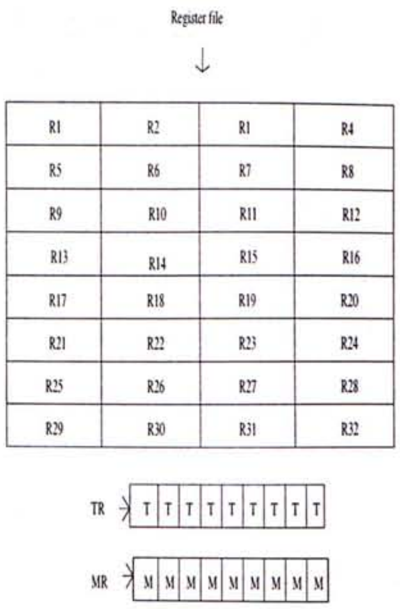

Figure 5.7: IADSM second variated structure

5.7 Introduction of IADSMC&IDLC Model

From the observation of the first twelve kernels from livermore loop kernels, we realized that nearly one fourth of instructions is used to control the looping. Generally, it includes the following operations :

1. increment the index value.
2. compare the index value with the limit
3. conditional branch depending on the result of the comparison.

In this model, we try to incorporate the above operations into implicit operations in our EReg design. IADSMC&IDLC model is an extension of IADSM model. By reorganizing the interpretation, IADSMC&IDLC model can reduce such operations by a lot. When the number of instructions within the loop is small, the impact of reducing such kind of instructions can be quite impressive.

5.8 Architectural and Operational Detail of IADSMC & IDLC Model

The model described in the section contains two models. The value of the C-bit determine which model the interpretation of the instruction should base on. The architectural and operational detail of model (A) is just the same as that of IADSM Model. The detail architectural and operational model of model (B) described in next section. It has four fields. The first field contains the instruction address where the branch should be taken. The second field contains the destination address of the branching while the third field contains the number of branching.

Name:	Instr_A	Address	Datum	Stride	M	C

Figure 5.8: IADSMC&IDLC model A structure

Name :	Instr_A	Dest_A	LOOP	C

Figure 5.9: IADSMC&IDLC model B structure

Operational Details

If the conversion bit C is ZERO, the operations will base on the model (A); Otherwise, the operations will base on the model (B). The operational details and behaviors of model (A) is just the same as that of IADSM model. These two models can be set by the instructions SET_A and SET_B. SET_A can set C bit to ZERO while SET_B will set C bit to ONE.

In both models, a fully associative match is always taken between the program counter and INST_A field. If any match occurs and C bit is ONE and the value in LOOP field is non-zero, the value in Dest_A field will be moved into program counter and the number #no in Loop field will be decreased by 1. The content of the fields in sub-model (B) can be set by a new instruction.

SET_L Rn, #addr_1, #addr_2, # no

where #addr_1 is stored into Inst_A field.

#addr_2 is stored into Dest_A field.

#no is stored into Loop field.

All other instructions including *LOAD* and *STORE* are not allowed to be operated on the model B. Otherwise, error will be result. There is a new instruction "div" to be introduced. The function of this instruction is to divide the first parameter by the second parameter and then return the quotient and the remainder to the third and fourth parameter respectively.

For example : " div 14,4,%l9,%l10 "

This instruction will cause the EReg %l9 containing the value of 3 and the EReg %l10 containing the value of 2. This instruction is very important in doing loop unrolling. After we have placed the number of looping into the first parameter and the size of unrolling into the second parameter, the third parameter will represent the number of looping for the part with unrolling and the fourth parameter will represent the number of looping for the remaining part without unrolling.

5.9 Discussion

5.9.1 Additional Operations

The model is mainly to reduce the number of instructions which is used to control the looping operations by implicit control. The figures 5.10 and 5.11 illustrate the additional operation compared with basic Model.

and

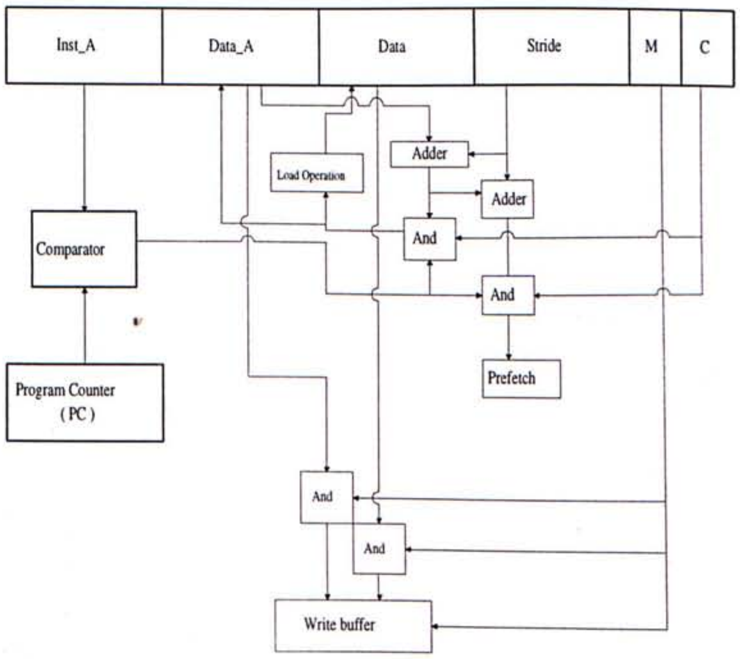

Figure 5.10: IADSMC Construction

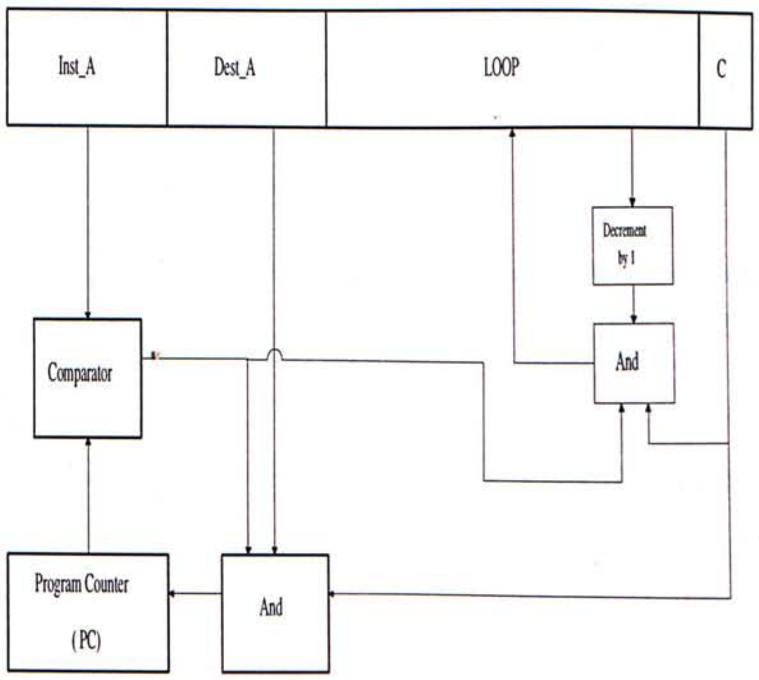

Figure 5.11: IDLC Construction

5.9.2 Compiling Procedure

The followings are some steps for compilers to make use of the feature of implicit looping :

1. Identify all the index values which are used to control the looping.
2. Calculate the number of times of looping from those index values.
3. Select and set an EReg to format B,i.e IDLC format..
4. Set the loop field of this EReg to the number of times of looping.
5. Set the Instruction field of this EReg to the instruction address which just follow the last instruction in the looping to be executed.
6. Set the Destination field of this EReg to the instruction address which is just above the first instruction in the looping to be executed.

Basically, the implicit looping aim at the loopings which have well defined loop limits. But if there is a case that the loop limit is defined at run time, it is still possible to handle it by setting the value of loop field to a very large number and then fix it later when the ultimate loop limit is available.

5.10 Example for IADSMC&IDLC Model

The kernel of hydro fragment from the the Benchmark of Livermore Loop in Figure 3.3 of 23 is used to demonstrate the performance of IADSMC&IDLC model. In the kernel of hydro fragment, z[k+10] & z[k+11] will be loaded for each k since there may be an aliasing problem between x[k] and z[k+11] or x[k] and z[k+12]. However, if IADSMC&IDLC model of EReg is used, we can reschedule the coding as shown in Figure 5.12.

```
3.    LOAD        z[k+10], ER1
1.    LOAD        z[k+11], ER2

      SET_S       ER1, 8 ; This value 8 represent the stride value of the array
      SET_S       ER2, 8
      SET_INST    ER1,2     ; This value 2 represent the instruction address of
                              the following program segment
      SET_INST    ER2, 8    ; This value 8 represent the instruction address of
                              the following program segment
      SET_B       ER3
      SET_L       ER3,14,4,n
4.    Compute     r*z[k+10] ( use EReg ER1 )
2.    Compute     t*z[k+11] ( use EReg ER2 )
5.    .....       .... ...
6.    Compute     q + y[k]*(r*z[k+10]+t*z[+11] ) and STORE into x[k].
9.    ( remove )
10.   Compute     r*z[k+11] ( use EReg ER2 )
8.    Compute     t*z[k+12] ( use EReg ER1 )
11.   .....       .... ...
12.   Compute     q + y[k+1]*(r*z[k+11]+t*z[+12] ) and STORE into x[k+1].
13.   Increment   the value of k by 2.
14.   .....       ( others )
```

Figure 5.12: Example : Modification of Hydro Fragment by IADSMC&IDLC Model

5.11 Simulation Results

There are totally 15 kernels used in the simulating the effect of the IADSMC&IDLC model of EReg. The source code of these 15 kernels are placed in Appendix A. The first 12 are Livermore Loop "C" language kernels. All kernels will be compiled by cc compiler into assembly programs in SPARC machine environment with the command " cc -O4 -S filename ". The last three kernels are the NASA 7 "FORTRAN" language Kernels. The command to generate their assembly codes is " f77 -O4 -S filename ". All the traditional registers in assembly programs are replaced by the IADSMC&IDLC model of EReg to see if there are any improvement. The result contains three data :

1. Total number of instructions issued

	Percentage of instr remained	Percentage of *LOAD* instr remained	Percentage of *STORE* instr remained
1. Hydro fragment	47.2%	0.349%	100%
2. ICCG excerpt	31.6%	1.93%	100%
3. Inner product	28.5%	0.4%	100%
4. Banded linear equations	41.6%	1.22%	0
5. Tri-diagonal elimination , below diagonal	14.4%	0.5%	0
6. General linear recurrence equations	12.6%	0.4%	0
7. Equation of state fragment	45.4%	0.117%	100%
8. ADI integration	35%	0.242%	50.2%
9. Integrate predictors	47.8%	0.23%	100%
10. Difference predictors	56.7%	0.145%	100%
11. First Sum	9.9%	0.6%	0.1%
12. First Difference	39.9%	0.4%	100%
13. Mxm.f	14.3%	0.29%	2.17%
14. Cff2td1.f	13.7%	0	0
15. Cholesky Decomposition / Substitution	9.87%	0.4%	0.066%

Table 5.2: Results for IADSMC&IDLC Model

2. Number of *LOAD* instructions issued

3. Number of *STORE* instructions issued.

The IADSMC&IDLC model provide another interpretation of the EReg. IADSMC&IDLC model improve the IADSM model by reducing the loop controling instructions. The result is illustrated in the Table 5.2. By using the compiler procedure described in subsection 5.9.2, the looping control instructions can be easily reduced. A new created instruction – *div* can make the computation of the value of loop field easier.

In the result of the kernels, the kernel 3 – Inner Product from Livermore Loop Benchmark obtain the greatest performance improvement in changing from IADSM to IADSMC&IDLC. Its main source statements in "C" program is shown in Figure 5.13

```
for ( k=0;k<n;k++)
        q+=z[k]*x[k];
```

Figure 5.13: Main statements in Kernel Three : INNER PRODUCT from Livermore Loop

All the *LOAD* and relevant instructions of z[k] and x[k] are reduced by the IADSM model. If the IADSMC&IDLC model is applied, the looping control instructions are also eliminated. These looping control instructions include :

1. Increment of the value k
2. Compare the value of k with n
3. Branch back to the beginning of the loop if the value of k is smaller than n

In the experiment carried out in the report [Cha95], only two kinds of computation instructions exits in the looping. One is the multiplication instruction of z[k] and x[k]. Another is the addition instruction of q and (z[k]*x[k]). The performance improvement is from 42.3% to 28.5%. The best performance is achieved in the kernel 15. Only 9.87% of the number of instructions are remained. Since every kernel requires the loop controling instructions to control the looping, all the kernels can obtain benefit from this model.

5.12 Temporary Storage Problem of IADSMC&IDLC Model

As described in previous five models - Basic model, ADM model, ADS model, ADSM model and IADSM model, it is very important for the idea of TEMPORARY EReg which is used to store the data that any temporary EReg should not be used to store data operand, since once the value of this temporary EReg is changed, it means that the memory content represented by this temporary EReg is changed. In order to cope with the problem of temporary storage, the solutions discussed in the section 3.12 of page 46 can be used in this IADSMC&IDLC model. There are totally six solutions. However, the first solution of using T-bit requires some modification. If the Instruction field, stride field and address field of the EReg can be access and operated by arithmetic instructions, each EReg may be organized as a set of four traditional registers with T-bit and M-bit. The T-bit is used to overcome the temporary storage problem as described in previous models while the M-bit is used to enable implicit storing. There is a simple way to group the traditional registers into set is by using the register names. For example, 8 bits is used to address EReg. The first six bits are used to select the set (i.e. EReg) while the last two bits are used to define the fields within the EReg. When the C-bit is zero, these four traditional registers represent instruction field, data address field, data field and stride field. When the C-bit is one, these four traditional registers represent instruction field, destination field, and loop field only. The loop field may be represented by two registers or one register with one register left to be useless. Since each EReg contains four traditional registers, the number of registers in register file should be a multiple of four plus two. If T-bit is OFF, four traditional registers are combined to be preform as an EReg in IADSMC&IDLC model. All the T-bits and M-bits can be grouped together to form two special registers called TR and MR respectively. The structure is shown in Figure 5.14.

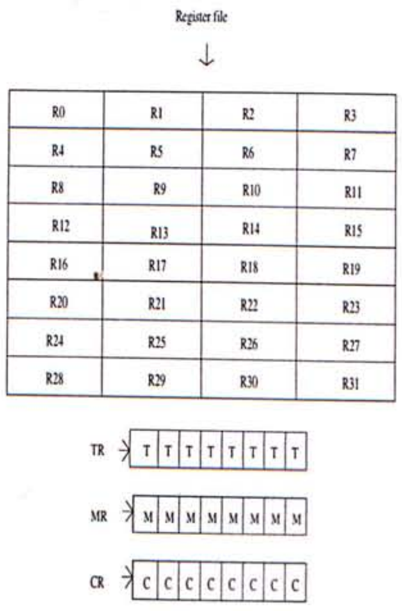

Figure 5.14: IADSMC&IDLC second variated structure

Chapter 6

Compiler and Memory System Support for EReg

6.1 Impact on Compiler

6.1.1 Register Usage

EReg will change some compiling styles of the present compiler. For example, some memory spaces are used as an extension of the register file. If there are two or more temporary registers loading value from same memory location, they are considered to be aliased with each other. Their value will be bound to be the same by the coherence property of EReg and hence, error will result.

```
        ld      [%fp-132],%l3 in Line 63
and     ld      [%fp-132],%l5 in Line 91
```

Figure 6.1: %l3 and %l5 represent different index values

The assembly instrucions in Figure 6.1 are obtained from NASA7 MXM program which is compiled by the command "f77 -O4 -S -Sun4 filename" in sparc machine. The traditional registers %l3 and %l5 represent different index values. If the traditional registers are changed by ERegs, the EReg values of %l3 and %l5 will be bound to be the same.

```
143.  ld     [%o4+4],%f25
144.  ld     [%o4],%f24
167.  ld     [%o2+4],%f5
168.  ld     [%o2],%f4
169.  fmuld  %f4,%f24,%f4
```

Figure 6.2: Problem of using DATA EReg to store temporary value

Another programming style which is dangerous for using ERegs is shown in Figure 6.2. Since once the value of an EReg is changed, it means that the memory content represented by this EReg is changed. If traditional registers are replaced by ERegs, data loaded from the memory locations [%o4+4] and [%o4] may be incorrect before the EReg %f4 is replaced by a new variable.

Thus, there are two logical types in using ERegs. One is DATA EReg and the other is TEMPORARY EReg. Data EReg is used to store data in *LOAD* instruction while the TEMPORARY EReg is used to store the result of other instructions (e.g. ADD, SLL, INC etc). Although both of them may be physically the same, they must be pre-assigned logically before use. That is, once ER1 is use'd as DATA EReg, ER1 should not be used as TEMPORARY EReg to store the result except the case that the destination address is the memory address represented by ER1. In general, the assembly program generated from the present C compiler must be rewritten if EReg is used.

6.1.2 Effect of Unrolling

The unrolling technique is more important for the program of using EReg than for that of using traditional registers. This technique can enhance the performance of Implicit Loading and DATA Prefetching especially in the case of using IADSMC&IDLC model. The unrolling effect of kernel 11 – First Sum from livermore benchmark is considered. There are two parts in this kernel as shown in Figure 6.3. The first part is with unrolling size of 4 while the second part is without unrolling. In IASDMC&IDLC model, it is easy for registers to be Implicit Loading in part (a) even after reducing the index and branch instructions and if part (b) is further reduced to as shown in Figure 6.4. It will be difficult to load the next data into %f24,%f25 and %f26,%f27 before the next instruction S97 is executed.

```
        for (k=1;k¡n;k++)
        x[k]=x[k-1]+y[k];

    After the program is assembled,

Part ( a )

        SET_B  %l11 ; set the instruction for implicit loading and
        SET_L  %l11, S87,S48,%l9 ; implicit branch
  S48   faddd  %f0,%f2,%f6 ; x[k] = x[k-1] + y[k]
  S65   faddd  %f6,%f10,%f12 ; x[k+1] = x[(k+1)-1] + y[(k+1)]
  S72   faddd  %f12,%f16,%f18 ; x[k+2] = x[(k+2)-1] + y[(k+2)]
  S79   faddd  %f18,%f22,%f0 ; x[k+3] = x[(k+3)-1] + y[(k+3)]

Part ( b )

  mS90  mL77013:
  S97   faddd  %f24,%f26,%f4 ; x[k] = x[k-1] + y[k]
  S99   cmp    %i5,%i2
  S103  bcs    mL77013
  S98   inc    8,%i5
```

Figure 6.3: Main "C" Statements in Kernel 11 – First Sum From Livermore Loop Benchmark Kernels

```
        SET_B  %l11 ; set the instruction for implicit loading and
        SET_L  %l11, S100,S97,%l10 ; implicit branch
  S97   faddd  %f24,%f26,%f4 ; x[k] = x[k-1] + y[k]
  S104  ......  ......... ; Other unrelated instructions
```

Figure 6.4: Modification of part two in Kernel 11 by IADSMC&IDLC model

6.1.3 Code Scheduling Algorithm

```
for (k=0;k¡n;k++) {
        x[k] = u[k]+r*(z[k]+r*y[k]) +
                t*( u[k+3] + r*( u[k+2] + r*u[k+1] ) +
                t*( u[k+6] + r*( u[k+5] + r*u[k+4] )));
}
```

Figure 6.5: Example used for different code scheduling

Sometimes, there are two or more code scheduling. In the example of Figure 6.5, one schedule is just to rearrange the *LOAD* instructions such that although there are no improvement in the number of instructions issued, 12 *LOAD* operations within the looping have changed from memory-EReg to EReg-EReg due to the coherence property of EReg. These instructions are the loading instructions for the array elements of u[k], u[k+1], u[k+2], u[k+3], u[k+4] and u[k+5]. While the other code scheduling algorithm is to group the array elements u[k] & u[k+1] , u[k+2]& u[k+3], u[k+4]&u[k+5] in pairs, and then if the techniques illustrated in 3.2 are used, 6 *LOAD* instructions can be reduced. Since the content of array element u[k+5] is the same as that of previous array element u[k+6], 2 *LOAD* operations can be changed from memory-EReg to EReg-EReg due to the coherence property of EReg. However, it is still difficult to ascertain that which code scheduling algorithm must be better because the actual performance also depend on other hardware supports.

6.2 Impact on Memory System

6.2.1 Memory Bottleneck

Since the ERegs reduce a great number of instructions to be issued especially in IADSM model and IADSMC&IDLC model, the memory bandwidth become the bottleneck. The method of solving this problem is to increase the On-Chip memory bandwidth. As quite a many of *LOAD* instructions operate on sequential data, it will be very beneficial to be able to load quadword, octaword data into a set of sequential ERegs.

6.2.2 Size of EReg Files

Since the Implicit Loading & Data Prefetching will restrict one EReg to one *LOAD* instruction only within the looping, the SET_S and SET_INST instructions can be moved just outside of the innermost looping such that the number of ERegs can still be maintained at a reasonable size. Usually, the number of ERegs is similar to that of traditional registers. About 32 ERegs will be sufficient in each model.

Chapter 7

Conclusions

7.1 Summary

This thesis introduces six new memory structures. The name of these structures is called EReg which stands for "Extended Register". They improve the system performance by 3 approaches. The first approach is to eliminate the ambiguous data aliasing. The second approach is to reduce the number of primitive instrucions by implementing implicit storing, implicit loading and implicit looping. The final approach is to implement data prefetching mechanism so as to reduce the cache miss ratio. These strutures are Basic model, ADM model, ADS model, ADSM model, IADSM model and IADSMC&IDLC model.

ERegs can be managed as efficient as traditional registers. Since a small number of ERegs are sufficient, the cost of building an EReg file should not be very expensive. The operations for the features of data prefetching, implicit loading, implicit storing and implicit looping should be able to be carried out in parallel with other instructions provided that the bandwidth is large and the access time of the memory is fast. This is especially important for the IADSMC&IDLC model which is the superset of all other models. The memory structures shown in Figures 5.10 and 5.11 show all the relationships between the fields of the ERegs completely. Since the features supported by each EReg can be performed independently, each EReg can be built by several separated components. In hardware implementation, these components can be located in different locations or areas within the processor chip. If the bandwidth and the speed of memory are improved in the future, the performance of ERegs can be enhanced significantly.

7.2 Future Research

If a conventional register is replaced by one of the ERegs, a new horizon will be unveiled in the field of compiler optimization technology. For future work, we would like to further pursue the following issues :

- Coherence between the multiprocessors

  The coherence can be maintained in a more easy and effective way because once the value of shared data is changed in one processor, a signal can be generated to update the ERegs with same address in the other processor which cannot be achieved in using the traditional registers. Thus, a detail study on the application of ERegs in multiprocessors may be carried out in the future.

- Incorporation of the factor of virtual memory into ERegs

  ERegs can be investigated in the environment of using virtual memory. Actually, some ideas of ERegs can also be applied on the design of virtual cache. Taking an example, there are two tags – virtual address tag and physical address tag in each cache line. An associative comparison can be carried out to make all aliased virtual address coherent by bounding them to be the same if they have same physical address value. Supposing that the way to select the set number is the same for both virtual address and physical address, an associative match can be made just within the same set when the value of any cache line is changed and hence the complexity of associativity can be greatly reduced.

- Building a compiler and a EReg machine simulator

  Due to the time contraint, the assembly process of using ERegs is done manually at the present moment. If a compiler can be built to support the automatic code generation in the future, a more detail study and evaluation of ERegs can be carried out.

Bibliography

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

[BC91] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings, 1991 International Conference on Supercomputing*, pages 176–186, 1991.

[BM90] B.Heggy and M.L.Soffa. Architectural support for register allocation in the presence of aliasing. In *Proceeding of Supercomputing'90*, pages 730–739, 1990.

[CBM+92] William Y. Chen, Roger A. Bringmann, Scott A. Mahlke, Richard E. Hank, and James E. Sicolo. An Efficient Architecture for Loop Based Data Preloading. In *Conference Proceedings, The 25th International Symposium on Microarchitecture*, pages 92–101, December 1992.

[Cha95] Y. Chan. Impact of EReg on the Performance of "C" and "Fortran" Programs. Technical report, Computer Science Division, The Chinese University of Hong Kong, May 1995.

[Che93] Tien-Fu Chen. *Data Prefetching for High-Performance Processors.* PhD thesis, Department of Computer Science and Engineering, University of Washington, July 1993.

[Chi91] Tzi-Cker Chiueh. An Integrated Memory Management Scheme For Dynamic Alias Resolution. pages 682–691, 1991.

[CK89] K. D. Cooper and K. Kennedy. Fast Interprocedural Alias Analysis. In *Conference Record of Sixteenth ACM Symposium on Principles of Programming Languages*, pages 49–59, 1989.

[CKP91] D. Callahan, K. Kennedy, and A. Porterfield. Sofware prefetching. In *Proceedings of Fourth International Conference on Architectural Support for Programming languages and Operating Systems*, pages 40–52, 1991.

[CMCmH91] William Y. Chen, Scott A. Mahlke, Pohua P. Chang, and Wenmei Hwu. Data Access Microarchitectures for Superscalar Processors. In *Conference Proceedings, The 24th International Symposium on Microarchitecture*, pages 69–73, 1991.

[FPJ92] John W.C. Fu, Janak H. Patel, and Bob L. Janssens. Stride directed prefetching in scalar processors. In *Conference Proceedings, The 25th International Symposium on Microarchitecture*, pages 102–110, 1992.

[GGV90] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed Data Prefetching in Multiprocessors with Memory Hierarchies. In *Proceedings, 1990 International Conference on Supercomputing*, pages 354–368, 1990.

[HC88] H.Dietz and C.H.Chi. Cregs: A new kind of memory for referencing arrays and pointers. In *Proceedings of supercomputing '88*, pages 360–367, 1988.

[Hil87] Mark D. Hill. Aspects of Cache Memory and Instruction Buffer Performance. Technical report, Computer Science Division,University of California at Berkeley, November 1987. Technical Report UCB/CSD 87/381.

[HP90] J. Hennessy and D. Patterson. *Computer Architecture : A Quantitative Approach.* Morgan Kaufmann, 1990.

[Jou90] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Conference Proceedings, The 17th International Symposium on Computer Architecture*, pages 364–373, May 1990.

[KL91] Alexander C. Klaiber and Henry M. Levy. An Architecture for Software-controlled Data Prefetching. In *Conference Proceedings, The 18th International Conference on Computer Architecture*, pages 43–53, 1991.

[LH88] J. R. Larus and P. N. Hilfinger. Detecting Conflicts Between Structure Accesses. In *Conference on Programming Language Design and Implementation*, pages 621–634, 1988.

[LYL87] R. L. Lee, P. C. Yew, and D. H. Lawrie. Data Prefetching in shard memory multiprocessors. In *Proceedings of 16th International Conference on Parallel Processing*, pages 28–31, 1987.

[MG91] T. C. Mowry and A. Gupta. Tolerating latency through software-controlled data prefetching. *Journal of Parallel and Distributed Computing*, pages 87–106, 1991.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings, ASPLOS IV*, pages 62–73, 1992.

[Mye81] E. W. Myers. A Precise Inter-procedural Dataflow Algorithm. In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, pages 219–230, January 1981.

[Nic89] A. Nicolau. Run-time disambiguation: coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, pages 663–678, 1989.

[NT92] Steve Nowakowski and Matthew T.O'Keefe. A CRegs Implementation Study Based on the MIPS-X RISC Processor. In *IEEE 1992 International Conference on Computer Design*, pages 558–563, 1992.

[Por89] Alan K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Department of Computer Science, Rice University, May 1989.

[Prz90] S. Przybylski. The performance impact of block sizes and fetch strategies. In *Proceedings of the 17th Annual Intl. Symp. on Computer Architecture*, pages 160–169, 1990.

[Sit79] Richard Sites. How to use 1000 registers. In *Caltech conference on VLSI*, pages 360–367, 1979.

[Skl92] I. Sklenar. Prefetch Unit for Vector Operations on Scalar Computers. In *Computer Architecture News 20*, volume 1, pages 31–37, 1992.

[Smi82] Alan Jay Smith. Cache Memories. *Computing Survey 14*, 3:473–530, 1982.

[WYMC93] JR. William Yu-Mei Chen. *Data Preload For Superscalar and VLIW Processors.* PhD thesis, Unversity of Illinois at Urbana-Champaign, 1993.

Appendix A

Source code of the Kernels

```
#include <stdio.h>
#include <stdlib.h>

main ()
{
                int       n;
                double    q,r,t;
                double    x[1500],y[1500],z[1500];
                void      loop1();

        q=0;n=1000;r=1;t=1;
        loop1(n,q,r,t,x,y,z);
}

void        loop1(n,q,r,t,x,y,z)
int         n;
double      q,r,t;
double      x[],y[],z[];
{
                long      k;

                for ( k=0;k<n;k++)
                    x[k]=q+y[k]*(r*z[k+10]+t*z[k+11]);
}
```

Figure A.1: Kernel One : HYDRO FRAGMENT from Livermore Loop

```
#include <stdio.h>
#include <stdlib.h>

main ()
{
            double  x[1500],v[1500];
            long    n;
            void    loop2();

            n=1000;
            loop2(n,x,v);
}

void        loop2(n,x,v)
long        n;
double      x[],v[];
{
            long    i,ii,ipntp,ipnt,k;

            ii = n;
            ipntp = 0;
            do {
                    ipnt=ipntp;
                    ipntp+=ii;
                    ii/=2;
                    i=ipntp - 1;
            for (k=ipnt+1;k<ipntp;k=k+2) {
                    i++;
                    x[i]=x[k]-v[k]*x[k-1]-v[k+1]*x[k+1];
            }
            } while(ii¿0);
}
```

Figure A.2: Kernel Two : ICCG EXCERPT from Livermore Loop

```
#include <stdio.h>
#include <stdlib.h>

main ()
{
            long     n;
            double   x[1500],z[1500];
            double   q;
            double   loop3();

            n=1000;
            q=loop3(n,x,z);
}

double      loop3(n,x,z)
long        n;
double      x[],z[];
{

            double   q;
            long     k;

            for ( k=0;k<n;k++) {
                     q+=z[k]*x[k];
            }
            return(q);
}
```

Figure A.3: Kernel Three : INNER PRODUCT from Livermore Loop

```
#include<stdio.h>
#include<stdlib.h>

main ()
{
                long      n;
                double    x[1500],y[1500];
                void      loop4();

                n = 1000;
                loop4(n,x,y);
}

void        loop4(n,x,y)
long        n;
double      x[],y[];
{
                long      m,k,lw,j;
                double    temp;

                m=(1001-7)/2;
                for ( k=6;k<1001;k=k+m) {
                          lw = k-6;
                          temp = x[k-1];
                for (j=4;j<n;j=j+5) {
                          temp -= x[lw]*y[j];
                          lw++;
                }
                          x[k-1]=y[4]*temp;
                }
}
```

Figure A.4: Kernel Four : BANDED LINEAR EQUATIONS from Livermore Loop

```
#include <stdio.h>
#include <stdlib.h>

main ()
{
                long    n;
                double  x[1500],y[1500],z[1500];
                void    loop5();



¿   ¿   n=1000;
                loop5(n,x,y,z);
}

void        loop5(n,x,y,z)
long        n;
double      x[],y[],z[];
{
                long    i;

                for (i=1;i<n;i++)
                        x[i]=z[i]*(y[i]-x[i-1]);
}
```

Figure A.5: Kernel Five : TRI-DIAGONAL ELIMINATION, BELOW DIAGONAL from Livermore Loop

```
#include <stdio.h>
#include <stdlib.h>

main ()
{
                long    n;
                double  w[1500],b[1500][1500];
                void    loop6();

                n=1000;
                loop6(n,w,b);
}

void        loop6(n,w,b)
long        n;
double      w[],b[][1500];
{
                long    i,k;

                for (i=1;i<n;i++)
                        for (k=0;k<i;k++)
                                w[i]+=b[k][i]*w[(i-k)-1];
}
```

Figure A.6: Kernel Six : GENERAL LINEAR from Livermore Loop

```
#include <stdio.h>
#include <stdlib.h>

main ()
{
                long     n;
                double   r,t;
                double   x[1500],u[1500],z[1500],y[1500];
                void     loop7();

                n=1000;r=1;t=1;
                loop7(n,r,t,x,u,z,y);

}

void        loop7(n,r,t,x,u,z,y)
long        n;
double      r,t;
double      x[],u[],z[],y[];
{
                long     k;

                for (k=0;k<n;k++) {
                         x[k]= u[k]+r*(z[k]+r*y[k])+
                         t*(u[k+3]+r*(u[k+2]+r*u[k+1])+
                         t*(u[k+6]+r*(u[k+5]+r*u[k+4])));
                }
}
```

Figure A.7: Kernel Seven : EQUATION OF STATE FRAGMENT from Livermore Loop

```
#include <stdio.h>
#include <stdlib.h>

main ()
{
                long      n;
                double   u1[2][1500][5],u2[2][1500][5],u3[2][1500][5];
                double   a11,a12,a13,a21,a22,a23,a31,a32,a33,sig;
                double   du1[1500],du2[1500],du3[1500];
                void      loop8();

                n = 1000;a11=1,a12=1,a13=1,a21=1,a22=1,a23=1,a31=1,a32=1,a33=1,sig=1;
                loop8(n,u1,u2,u3,a11,a12,a13,a21,a22,a23,a31,a32,a33,sig,du1,du2,du3);
}


void         loop8(n,u1,u2,u3,a11,a12,a13,a21,a22,a23,a31,a32,a33,sig,du1,du2,du3)
long         n;
double       u1[][1500][5],u2[][1500][5],u3[][1500][5];
double       a11,a12,a13,a21,a22,a23,a31,a32,a33,sig;
double       du1[],du2[],du3[];
{
                long      nl1,nl2,kx,ky;

                nl1 = 0;
                nl2 = 1;
                for ( kx=1 ; kx<3 ; kx++ ){
                     for ( ky=1 ; ky<n ; ky++ ) {
                          du1[ky] = u1[nl1][ky+1][kx] - u1[nl1][ky-1][kx];
                          du2[ky] = u2[nl1][ky+1][kx] - u2[nl1][ky-1][kx];
                          du3[ky] = u3[nl1][ky+1][kx] - u3[nl1][ky-1][kx];
                          u1[nl2][ky][kx]=
                               u1[nl1][ky][kx]+a11*du1[ky]+a12*du2[ky]+a13*du3[ky] + sig*
                               (u1[nl1][ky][kx+1]-2.0*u1[nl1][ky][kx]+u1[nl1][ky][kx-1]);
                          u2[nl2][ky][kx]=
                               u2[nl1][ky][kx]+a21*du1[ky]+a22*du2[ky]+a23*du3[ky] + sig*
                               (u2[nl1][ky][kx+1]-2.0*u2[nl1][ky][kx]+u2[nl1][ky][kx-1]);
                          u3[nl2][ky][kx]=
                               u3[nl1][ky][kx]+a31*du1[ky]+a32*du2[ky]+a33*du3[ky] + sig*
                               (u3[nl1][ky][kx+1]-2.0*u3[nl1][ky][kx]+u3[nl1][ky][kx-1]);
                     }
                }
}
```

Figure A.8: Kernel EIGHT : ADI INTEGRATION from Livermore Loop

```
#include <stdio.h>
#include <stdlib.h>

main()
{
                long    n;
                double  dm22,dm23,dm24,dm25,dm26,dm27,dm28,c0;
                double  px[1500][25];
                void    loop9();

                n=1000;dm22=1;dm23=1;dm24=1;dm25=1;dm26=1;dm27=1;dm28=1;c0=1;
                loop9(n,dm22,dm23,dm24,dm25,dm26,dm27,dm28,c0,px);
}


void        loop9(n,dm22,dm23,dm24,dm25,dm26,dm27,dm28,c0,px)
long        n;
double      dm22,dm23,dm24,dm25,dm26,dm27,dm28,c0;
double      px[][25];
{
                long    i;

                for ( i=0 ; i<n ; i++ ) {
                    px[i][0] = dm28*px[i][12] + dm27*px[i][11] + dm26*px[i][10] +
                    dm25*px[i][ 9] + dm24*px[i][ 8] + dm23*px[i][ 7] +
                    dm22*px[i][ 6] + c0*( px[i][ 4] + px[i][ 5]) + px[i][ 2];
                }
}
```

Figure A.9: Kernel Nine : INTEGRATE PREDICTORS from Livermore Loop

```
#include <stdio.h>
#include <stdlib.h>

main()
{
                long    i,n;
                double  ar,br,cr,cx[1500][25],px[1500][25];
                void    loop10();

                n=1000; ar=1;br=1;cr=1;
                loop10(n,ar,br,cr,cx,px);
}

void    loop10(n,ar,br,cr,cx,px)
long    n;
double  ar,br,cr,cx[][25],px[][25];
{
                long i;

                for ( i=0 ; i<n ; i++ ) {
                        ar          = cx[i][ 4];
                        br          = ar - px[i][ 4];
                        px[i][ 4]   = ar;
                        cr          = br - px[i][ 5];
                        px[i][ 5]   = br;
                        ar          = cr - px[i][ 6];
                        px[i][ 6]   = cr;
                        br          = ar - px[i][ 7];
                        px[i][ 7]   = ar;
                        cr          = br - px[i][ 8];
                        px[i][ 8]   = br;
                        ar          = cr - px[i][ 9];
                        px[i][ 9]   = cr;
                        br          = ar - px[i][10];
                        px[i][10]   = ar;
                        cr          = br - px[i][11];
                        px[i][11]   = br;
                        px[i][13]   = cr - px[i][12];
                        px[i][12]   = cr;
                }
}
```

Figure A.10: Kernel Ten : DIFFERENCE PREDICTORS from Livermore Loop

```
#include <stdio.h>
#include <stdlib.h>

main()
{
                long    k,n;
                double  x[1500],y[1500];
                void    loop11();

                n=1000;
                loop11(n,x,y);
}


void        loop11(n,x,y)
long        n;
double      x[],y[];
{
                long k;

                x[0]=y[0];

                for (k=1;k<n;k++)
                        x[k]=x[k-1]+y[k];
}
```

Figure A.11: Kernel Eleven : FIRST SUM from Livermore Loop

```
#include <stdio.h>
#include <stdlib.h>

main()
{

                long     n;
                double   x[1500],y[1500];
                void     loop12();

                n=1000;
                loop12(n,x,y);
}

void        loop12(n,x,y)
long        n;
double      x[],y[];
{
                long     k;

                for (k=0;k<n;k++)
                         x[k]=y[k+1]-y[k];
}
```

Figure A.12: Kernel Twelve : FIRST DIFFERENCE from Livermore Loop

```
C
        SUBROUTINE MXM (A, B, C, L, M, N)
        IMPLICIT REAL*8 (A-H,O-Z)
        DIMENSION A(L,M), B(M,N), C(L,N)
C
C       4-WAY UNROLLED MATRIX MULTIPLY ROUTINE FOR VECTOR COMPUTERS.
C       M MUST BE A MULTIPLE OF 4. CONTIGUOUS DATA ASSUMED.
C       D H BAILEY 11/15/84
C
        DO 100 K = 1, N
                DO 100 I = 1, L
        C(I,K) = 0.D0
100     CONTINUE
        DO 110 J = 1, M, 4
                DO 110 K = 1, N
                        DO 110 I = 1, L
                        C(I,K) = C(I,K) + A(I,J) * B(J,K)
                                + A(I,J+1) * B(J+1,K) + A(I,J+2) * B(J+2,K)
                                + A(I,J+3) * B(J+3,K)
110     CONTINUE
C
        RETURN
        END
```

Figure A.13: Kernel Thirteen : MXM.F from NASA7 Benchmark

```
C
      SUBROUTINE CFFT2D2 (IS, M, M1, N, X, W, IP)
      IMPLICIT REAL*8(A-H,O-Z)
C
C     PERFORMS COMPLEX RADIX 2 FFTS ON THE SECOND DIMENSION OF THE 2-D ARRAY X
C     D H BAILEY 11/15/84
C
      COMPLEX*16 X(M1,N), W(N), CT, CX
      INTEGER IP(2,N)
      DATA PI/3.141592653589793D0/
C
C     IF IS = 0 THEN INITIALIZE ONLY
C
      N2 = N / 2
      IF (IS .EQ. 0) THEN
      DO 100 I = 1, N2
            T = 2.D0 * PI * (I-1) / N
            W(I) = DCMPLX (COS (T), SIN (T))
100   CONTINUE
      RETURN
      ENDIF
C
C     PEFORM FORWARD OR BACKWARD FFTS ACCORDING TO IS = 1 OR -1
C
      DO 110 I = 1, N
            IP(1,I) = I
110   CONTINUE
      L = 1
      I1 = 1
C
120   I2 = 3 - I1
      DO 130 J = L, N2, L
            CX = W(J-L+1)
      IF (IS .LT. 0) CX = CONJG (CX)
      DO 130 I = J-L+1, J
            II = IP(I1,I)
            IP(I2,I+J-L) = II
            IM = IP(I1,I+N2)
            IP(I2,I+J) = IM
            DO 130 K = 1, M
                  CT = X(K,II) - X(K,IM)
                  X(K,II) = X(K,II) + X(K,IM)
                  X(K,IM) = CT * CX
130   CONTINUE
      L = 2 * L
      I1 = I2
      IF (L .LE. N2) GOTO 120
C
      DO 150 I = 1, N
         II = IP(I1,I)
         IF (II .GT. I) THEN
      DO 140 K = 1, M
            CT = X(K,I)
            X(K,I) = X(K,II)
            X(K,II) = CT
140   CONTINUE
      ENDIF
150   CONTINUE
C
      RETURN
      END
```

Figure A.14: Kernel Fourteen : CFFT2D2.F from NASA7 Benchmark

```
C
        SUBROUTINE CHOLSKY (IDA, NMAT, M, N, A, NRHS, IDB, B)
        IMPLICIT REAL*8(A-H,O-Z)
C
C       CHOLESKY DECOMPOSITION/SUBSTITUTION SUBROUTINE.
C
C       11/28/84 D H BAILEY MODIFIED FOR NAS KERNEL TEST
C
        REAL*8 A(0:IDA, -M:0, 0:N), B(0:NRHS, 0:IDB, 0:N),
        *EPSS(0:256)
        DATA EPS/1D-13/
C
C       CHOLESKY DECOMPOSITION
C
        DO 1 J = 0, N
                I0 = MAX ( -M, -J )
C
C       OFF DIAGONAL ELEMENTS
C
        DO 2 I = I0, -1
            DO 3 JJ = I0 - I, -1
                DO 3 L = 0, NMAT
3                       A(L,I,J) = A(L,I,J) - A(L,JJ,I+J) * A(L,I+JJ,J)
        DO 2 L = 0, NMAT
2               A(L,I,J) = A(L,I,J) * A(L,0,I+J)
C
C STORE INVERSE OF DIAGONAL ELEMENTS
C
        DO 4 L = 0, NMAT
4               EPSS(L) = EPS * A(L,0,J)
        DO 5 JJ = I0, -1
                DO 5 L = 0, NMAT
5                       A(L,0,J) = A(L,0,J) - A(L,JJ,J) ** 2
        DO 1 L = 0, NMAT
1               A(L,0,J) = 1. / SQRT ( ABS (EPSS(L) + A(L,0,J)) )
C
C       SOLUTION
C
        DO 6 I = 0, NRHS
                DO 7 K = 0, N
                DO 8 L = 0, NMAT
8                       B(I,L,K) = B(I,L,K) * A(L,0,K)
        DO 7 JJ = 1, MIN (M, N-K)
                DO 7 L = 0, NMAT
7                       B(I,L,K+JJ) = B(I,L,K+JJ) - A(L,-JJ,K+JJ) * B(I,L,K)
C
        DO 6 K = N, 0, -1
                DO 9 L = 0, NMAT
9                       B(I,L,K) = B(I,L,K) * A(L,0,K)
        DO 6 JJ = 1, MIN (M, K)
                DO 6 L = 0, NMAT
6                       B(I,L,K-JJ) = B(I,L,K-JJ) - A(L,-JJ,K) * B(I,L,K)
C
        RETURN
        END
```

Figure A.15: Kernel Fifteen : CHOLSKY.F from NASA7 Benchmark

Appendix B

Program Analysis

B.1 Program analysed by Basic Model

1. Kernel One – Hydro fragment

   The main C statements are

   ```
   for ( k=0;k<n;k++)
   x[k]=q+y[k]*(r*z[k+10]+t*z[k+11]);
   ```

   This kernel is used as an example in Basic Model. Since the variable of z[k+11] is equal to the element z[k+10] in next iteration, only the element of z[k+11] is required to be loaded in each iteration after ERegs replaced the traditional registers and solved aliasing problem.

2. Kernel Two – ICCG excerpt (Incomplete Cholesky Conjugate Gradient)
   The main C statements are

   ```
   i = ipntp - 1
   for ( k=ipnt+1;k<ipntp;k=k+2) {
           i++;
           x[i]=x[k]-v[k]*x[k-1]-v[k+1]*x[k+1]
   }
   ```

   The above program segment are translated into two assembly program code segments. One is with an unrolling size 2 while another is without unrolling. By using

EReg, the aliasing problem is solved. Only one of the the element of x[k-1] or x[k+1] is required to be loaded such that the number of load instructions can be reduced from 20 to 16.

3. Kernel Three – Inner product

   The main "C" statements are

```
for ( k=0;k<n;k++)
q+=z[k]+x[k];
```

   Since there are no aliasing problem, Basic model cannot improve the performance.

4. Kernel Four – Banded linear equations

   The main "C" statements are

```
temp=x[k-1];
for (j=4;j<n;j=j+5) {
        temp-=x[lw]*y[j];
        lw++;
}
x[k-1]=y[4]*temp;
```

   There are no aliasing problem. No benefit can be achieved in Basic model. Moreover, since the variable TEMP is directly loaded from the element x[k-1], the value element x[k-1] will change with the variable TEMP and hence error will occur. In order to cope with this problem, EReg cannot directly replace the traditional register. An extra EReg is used to hold the value from x[k-1] first and then copied its value to the EReg copy of TEMP. Therefore, the total number of instructions in basic model is increased slightly compared with the original number of using traditional registers.

5. Kernel Five – Tri-diagonal Elimination, Below Diagonal

   The main "C" statements are :

```
for (i=1;i<n;i++)
x[i]=z[i]*( y[i]-x[i-1])
```

Since every element of x[i] is stored and loaded explicitly once only, there is no improvement after using basic model.

6. Kernel Six – General Linear recurrence equations

   The main "C" statements are :

```
for (i=1;i<n;i++)                    ... loop1
        for (k=0;k<i;k++)            ... loop2
        w[i]+=b[k][i]*w[(i-k)-1];
```

   The value of w[i] is continued to be updated and stored within loop2 if traditional registers are used. The reason is that if the value of w[i] is not stored immediately after each " w[i]+=b[k][i]*w[(i-k)-1] " performed, error may result if other processes or instructions read data from the memory location of w[i] at that time. However, it will be possible to move the store instruction of w[i] outside loop2 if EReg is used. It is because every loading operations can check if the data have been already existed in EReg file. If yes, the data will be read from the EReg file directly; Otherwise, the date will be read from the specified memory location.

7. Kernel Seven – Equation of state fragment

   The main "C" statements are :

```
for (k=0;k<n;k++) {
x[k] =  u[k]+r*(z[k]+r*y[k]) +
        t*( u[k+3] + r*( u[k+2] + r*u[k+1] ) +
        t*( u[k+6] + r*( u[k+5] + r*u[k+4] )));
}
```

   There are two code scheduling algorithms. One is just to rearrange the load instructions such that although there are no improvement in the number of instructions issued, 12 load operations within the looping have changed from memory-EReg to EReg-EReg due to the basic property of EReg. These instructions are the loading instructions for the elements u[k], u[k+1], u[k+2], u[k+3], u[k+4] and u[k+5].

While another code scheduling algorithm is to reduce 6 load instructions, but only 2 load operations will change from memory-EReg to EReg-EReg due to the coherence property of EReg. Thus, it is very difficult to say which code scheduling algorithm must be better.

8. Kernel Eight – ADi integration

   The main "C" statements are

```
C27  for ( kx=1 ; kx<3 ; kx++ ){
C28          for ( ky=1 ; ky<n ; ky++ ) {
C29          du1[ky] = u1[nl1][ky+1][kx] - u1[nl1][ky-1][kx];
C30          du2[ky] = u2[nl1][ky+1][kx] - u2[nl1][ky-1][kx];
C31          du3[ky] = u3[nl1][ky+1][kx] - u3[nl1][ky-1][kx];
C32          u1[nl2][ky][kx]=
C33              u1[nl1][ky][kx]+a11*du1[ky]+a12*du2[ky]+a13*du3[ky] + sig*
C34              (u1[nl1][ky][kx+1]-2.0*u1[nl1][ky][kx]+u1[nl1][ky][kx-1]);
C35          u2[nl2][ky][kx]=
C36              u2[nl1][ky][kx]+a21*du1[ky]+a22*du2[ky]+a23*du3[ky] + sig*
C37              (u2[nl1][ky][kx+1]-2.0*u2[nl1][ky][kx]+u2[nl1][ky][kx-1]);
C38          u3[nl2][ky][kx]=
C39              u3[nl1][ky][kx]+a31*du1[ky]+a32*du2[ky]+a33*du3[ky] + sig*
C40              (u3[nl1][ky][kx+1]-2.0*u3[nl1][ky][kx]+u3[nl1][ky][kx-1]);
C41          }
C42  }
```

   The load instructions for du1[ky], du2[ky] and du3[ky] can be reduced from 3 (i.e C33,C36,C39) to 1 (i.e C33 only) if EReg is used. The aliasing problem can be reduced. Then, there are three pairs :

   - u1[nl1][ky-1][kx] & u[nl1][ky][kx+1] ;
   - u2[nl1][ky-1][kx] & u2[nl1][ky][kx+1]; and
   - u3[nl1][ky-1][kx] & u3[nl1][ky][kx+1].

   Only one elment for each pair is required to be loaded.

9. Kernel Nine – Integrate predictors

   The main "C" statements are

```
for ( i=0 ; i<n ; i++ ) {
        px[i][0] = dm28*px[i][12] + dm27*px[i][11] + dm26*px[i][10] +
                   dm25*px[i][ 9] + dm24*px[i][ 8] + dm23*px[i][ 7] +
                   dm22*px[i][ 6] + c0*( px[i][ 4] + px[i][ 5]) + px[i][ 2];
}
```

Since there are no aliasing problems within the whole program segment, No improvement can be obtained aftering using ERegs.

10. Kernel Ten – Difference predictors

    The main "C" statements are

```
for ( i=0 ; i<n ; i++ ) {
        ar = cx[i][ 4];
        br = ar - px[i][ 4];
        px[i][ 4] = ar;
        cr = br - px[i][ 5];
        px[i][ 5] = br;
        ar = cr - px[i][ 6];
        px[i][ 6] = cr;
        br = ar - px[i][ 7];
        px[i][ 7] = ar;
        cr = br - px[i][ 8];
        px[i][ 8] = br;
        ar = cr - px[i][ 9];
        px[i][ 9] = cr;
        br = ar - px[i][10];
        px[i][10] = ar;
        cr = br - px[i][11];
        px[i][11] = br;
        px[i][13] = cr - px[i][12];
        px[i][12] = cr;
}
```

    There are no aliasing problem within the aboved program segment.

11. Kernel Eleven – First sum

    The main "C" statements are

```
for (k=1;k<n;k++)
    x[k]=x[k-1]+y[k];
```

There are no aliasing problem within the above program segment.

12. Kernel Twelve – First difference

    The main "C" statements are

```
for (k=0;k<n;k++)
    x[k]=y[k+1]-y[k];
```

    Since the value of y[k+1] is equal to the value of y[k] in next looping, the number of load operations is reduced from two to one for each k after using EReg. The percentage of load instructions is nearly reduced by half.

13. Kernel Thirteen – Mxm.f

    The main "Fortran" statements are

```
DO 110 J=1,M,4
   Do 110 K=1,N
      Do 110 I=1,L
         C(I,K) = C( I,K) + A(I,J) + B(J,K)
                  + A(I,J+1)*B(J+1,K)+A(I,J+2)*B(J+2,K)
                  + A(I,J+3)*B(J+3,K)
```

    If the A,B and C are completely different (i.e No overlapping), no aliasing problem occurs and hence no benefit can be obtained from using ERegs.

14. Kernel Fourteen – Cff2ttd1.f

    The main "Fortran" statements are

```
120I2 = 3 - I1
   DO 130 J = L, N2, L
      CX = W(J-L+1)
      IF (IS .LT. 0) CX = CONJG (CX)
         DO 130 I = J-L+1, J
         II = IP(I1,I)
         IP(I2,I+J-L) = II
         IM = IP(I1,I+N2)
         IP(I2,I+J) = IM
```

```
F40          DO 130 K = 1, M
F41          CT = X(K,II) - X(K,IM)
F42          X(K,II) = X(K,II) + X(K,IM)
F43          X(K,IM) = CT * CX
F44   130CONTINUE
F45     L = 2 * L
F46     I1 = I2
F47     IF (L .LE. N2) GOTO 120
```

Since the compiler cannot determine if there are some aliasing problem among the variables, if EReg is used, the coding can be rescheduled such that the loading of X(K,II) and X(K,IM) in the F42 can be eliminated. Moreover, the explicit store of CT in F41, the loading of CT and CX in F43 can be eliminated.

15. Kernel fifteen – Cholesky.f

    The main "Fortran" statements are

```
F39   DO 6 I = 0, NRHS
F40   DO 7 K = 0, N
F41      DO 8 L = 0, NMAT
F428        B(I,L,K) = B(I,L,K) * A(L,0,K)
F43         DO 7 JJ = 1, MIN (M, N-K)
F44            DO 7 L = 0, NMAT
F457              B(I,L,K+JJ) = B(I,L,K+JJ) - A(L,-JJ,K+JJ) * B(I,L,K)
F46C
F47   DO 6 K = N, 0, -1
F48      DO 9 L = 0, NMAT
F499        B(I,L,K) = B(I,L,K) * A(L,0,K)
F50   DO 6 JJ = 1, MIN (M, K)
F51      DO 6 L = 0, NMAT
F526        B(I,L,K-JJ) = B(I,L,K-JJ) - A(L,-JJ,K) * B(I,L,K)
```

There are four parts.

(a) F41 - F42 with 740952

(b) F44 - F45 with 4020016

(c) F48 - F49 with 740952

(d) F51 - F52 with 3710784

There are no aliasing problem in these four parts, but when the techniques used to reschedule the coding in part two, some dummy instructions disappear. For detail, please refer to the report [Cha95].

B.2 Program analysed by ADM Model

1. Kernel One – Hydro fragment

   The main C statements are

   ```
   for ( k=0;k<n;k++)
   x[k]=q+y[k]*(r*z[k+10]+t*z[k+11]);
   ```

   The elements z[k+10] & z[k+11] will be loaded for each k since there may be an aliasing problem between x[k] and z[k+11] or x[k] and z[k+12]. However, if EReg is used, we can reschedule the coding such that the total number of 4 loading instructions z[k+11], z[k+10], z[k+12] and z[k+11] in one looping will be decreased to only 2 loading instructions z[k+11] and z[k+12] with ONE instruction outside the looping "for (k=0;k<n;k++)". The STORE instructions can not be reduced. Therefore, the performance of ADM model will be same as Basic model.

2. Kernel Two – ICCG excerpt (Incomplete Cholesky Conjugate Gradient)

   The main C statements are

   ```
   i = ipntp - 1
   for ( k=ipnt+1;k<ipntp;k=k+2) {
           i++;
           x[i]=x[k]-v[k]*x[k-1]-v[k+1]*x[k+1]
   }
   ```

   The above program segment are translated into an assembly program code segment with an unrolling size 2 followed by another assembly program code segment without unrolling. By using EReg, the aliasing problem between x[k-1] and x[k+1] can be solved so that the number of load instructions can be reduced from 20 load instructions to 16 instructions. x[i] will not be aliased to x[k-1], x[k] and x[k+1]. Therefore, no benefit can be obtained in performing implicit store.

3. Kernel Three – Inner product

   The main "C" statements are

   ```
   for ( k=0;k<n;k++)
   q+=z[k]+x[k];
   ```

   Since there are no aliasing problem, the features in the ADM model cannot improve the performance

4. Kernel Four – Banded linear equations

   The main "C" statements are

   ```
   temp=x[k-1];
   for (j=4;j<n;j=j+5) {
           temp-=x[lw]*y[j];
           lw++;
   }
   x[k-1]=y[4]*temp;
   ```

   There are no aliasing problem. No benefit can be achieved in Basic model. Moreover, there is an overhead that the variable TEMP can be directly loaded from x[k-1]. Therefore, the total number of instructions in basic model is increased slightly compared with original one using traditional registers. However, since the data of x[k-1] has been loaded, the explicit store "x[k-1]=y[4]*temp" can be changed to implicit store and hence the total number of instructions and total number of store instructions are reduced in using ADM model.

5. Kernel Five – Tri-diagonal Elimination, Below Diagonal

   The main "C" statements are :

```
for (i=1;i<n;i++)
x[i]=z[i]*( y[i]-x[i-1])
```

Since every result x[i] must be stored explicitly and each x[i-1] is loaded only once, there is no improvement after using basic model. However, all the store instructions can be reduced if the implicit store is supported, i.e ADM model.

6. Kernel Six – General Linear recurrence equations

   The main "C" statements are :

```
for (i=1;i<n;i++)                    ... loop1
        for (k=0;k<i;k++)            ... loop2
        w[i]+=b[k][i]*w[(i-k)-1];
```

   The value of w[i] is continued to be updated and stored within loop2 if traditional registers are used. The reason is that if the value of w[i] is not stored immediately after each " w[i]+=b[k][i]*w[(i-k)-1] " performed, error may result if other processes or instructions read data from the memory location of w[i] at that time. However, it will be possible to move the store instruction of w[i] outside loop2 if EReg is used. It is because every loading operations will check if the data have been already existed in EReg file. If yes, the data will be read from the EReg file directly; Otherwise, the date will be read from the specified memory location. If implicit store is supported (i.e ADM model), nearly all explicit stores (storing the value of w[i]) can be changed to implicit stores.

7. Kernel Seven – Equation of state fragment

   The main "C" statements are :

```
for (k=0;k<n;k++) {
x[k] =  u[k]+r*(z[k]+r*y[k]) +
        t*( u[k+3] + r*( u[k+2] + r*u[k+1] ) +
        t*( u[k+6] + r*( u[k+5] + r*u[k+4] )));
}
```

There are two code scheduling algorithms. One is just to rearrange the load instructions such that although there are no improvement in the number of instructions issued, 12 load operations within the looping have changed from memory-register to register-register due to the basic property of EReg. These instructions are the loading instructions for u[k], u[k+1], u[k+2], u[k+3], u[k+4] and u[k+5]. While the other code scheduling algorithm is to reduce 6 load instructions, but only 2 load operations will change from memory-register to register-register due to the basic property of EReg. Thus, it is very difficult to say which code scheduling algorithm must be better. Since the value of x[k] is not loaded, the explicit store of x[k] cannot be changed to implicit store. Thus, the extra feature of ADM model is useless in this program.

8. Kernel Eight – ADi integration

   The main "C" statements are

```
C27  for ( kx=1 ; kx<3 ; kx++ ){
C28          for ( ky=1 ; ky<n ; ky++ ) {
C29          du1[ky] = u1[nl1][ky+1][kx] - u1[nl1][ky-1][kx];
C30          du2[ky] = u2[nl1][ky+1][kx] - u2[nl1][ky-1][kx];
C31          du3[ky] = u3[nl1][ky+1][kx] - u3[nl1][ky-1][kx];
C32          u1[nl2][ky][kx]=
C33               u1[nl1][ky][kx]+a11*du1[ky]+a12*du2[ky]+a13*du3[ky] + sig*
C34               (u1[nl1][ky][kx+1]-2.0*u1[nl1][ky][kx]+u1[nl1][ky][kx-1]);
C35          u2[nl2][ky][kx]=
C36               u2[nl1][ky][kx]+a21*du1[ky]+a22*du2[ky]+a23*du3[ky] + sig*
C37               (u2[nl1][ky][kx+1]-2.0*u2[nl1][ky][kx]+u2[nl1][ky][kx-1]);
C38          u3[nl2][ky][kx]=
C39               u3[nl1][ky][kx]+a31*du1[ky]+a32*du2[ky]+a33*du3[ky] + sig*
C40               (u3[nl1][ky][kx+1]-2.0*u3[nl1][ky][kx]+u3[nl1][ky][kx-1]);
C41          }
C42  }
```

   The load instructions for du1[ky], du2[ky] and du3[ky] can be reduced from 3 (i.e C33,C36,C39) to 1 (i.e C33 only) if EReg is used. Moreover, the aliasing problem can be reduced between u1[nl1][ky-1][kx] & u[nl1][ky][kx+1], u2[nl1][ky-1][kx] & u2[nl1][ky][kx+1] and u3[nl1][ky-1][kx] & u3[nl1][ky][kx+1]. If implicit Store is supported by the EReg (i.e ADM model), the store instructions for du1[ky] (C29), du2[ky] (C30) and du3[ky] (C31) can be removed by first loading du1[ky], du2[ky] and du3[ky].

9. Kernel Nine – Integrate predictors

   The main "C" statements are

```
C23  for ( i=0 ; i<n ; i++ ) {
C24          px[i][0] = dm28*px[i][12] + dm27*px[i][11] + dm26*px[i][10] +
C25                        dm25*px[i][ 9] + dm24*px[i][ 8] + dm23*px[i][ 7] +
C26                        dm22*px[i][ 6] + c0*( px[i][ 4] + px[i][ 5]) + px[i][ 2];
C27  }
```

   Since there are no aliasing problems within the whole program segment, No improvement can be obtained after using ADM model.

10. Kernel Ten – Difference predictors

    The main "C" statements are

```
C22  for ( i=0 ; i<n ; i++ ) {
C23          ar = cx[i][ 4];
C24          br = ar - px[i][ 4];
C25          px[i][ 4] = ar;
C26          cr = br - px[i][ 5];
C27          px[i][ 5] = br;
C28          ar = cr - px[i][ 6];
C29          px[i][ 6] = cr;
C30          br = ar - px[i][ 7];
C31          px[i][ 7] = ar;
C32          cr = br - px[i][ 8];
C33          px[i][ 8] = br;
C34          ar = cr - px[i][ 9];
C35          px[i][ 9] = cr;
C36          br = ar - px[i][10];
C37          px[i][10] = ar;
C38          cr = br - px[i][11];
C39          px[i][11] = br;
C40          px[i][13] = cr - px[i][12];
C41          px[i][12] = cr;
C42}
```

Since there are no aliasing problem within the above program segment. No improvement can be obtained after using ADM model.

11. Kernel Eleven – First sum

    The main "C" statements are

```
C24  for (k=1;k<n;k++)
C25       x[k]=x[k-1]+y[k];
```

    Since the value of x[k] has been loaded before, all explicit stores of x[k] can be changed to implicit stores if ADM model is adopt.

12. Kernel Twelve – First difference

    The main "C" statements are

```
C21  for (k=0;k<n;k++)
C22       x[k]=y[k+1]-y[k];
```

    Since the value of y[k+1] is equal to the value of y[k] in next looping, the number of load operations is reduced from two to one for each k after using EReg. The percentage of load instructions is nearly reduced by half. Since the value of x[k] hasn't been loaded before, all explicit stores of x[k] cannot be changed to implicit stores even if ADM model is adopt.

13. Kernel Thirteen – Mxm.f

    The main "Fortran" statements are

```
DO 110 J=1,M,4
   Do 110 K=1,N
      Do 110 I=1,L
         C(I,K) = C( I,K) + A(I,J) + B(J,K)
                  + A(I,J+1)*B(J+1,K)+A(I,J+2)*B(J+2,K)
                  + A(I,J+3)*B(J+3,K)
```

If the A,B and C are completely different (i.e No overlapping), no aliasing problem occur and hence no benefit obtained from using ERegs. However, if ADM model is used, the total of 1618247 store instructions will be reduced to the number of 35111 instructions only. This is due to the implicit store can be performed on the store of C(I,K) because C(I,K) has been read before the store operation.

14. Kernel Fourteen – Cff2ttd1.f

    The main "Fortran" statements are

```
F31  120I2 = 3 - I1
F32      DO 130 J = L, N2, L
F33         CX = W(J-L+1)
F34         IF (IS .LT. 0) CX = CONJG (CX)
F35            DO 130 I = J-L+1, J
F36            II = IP(I1,I)
F37            IP(I2,I+J-L) = II
F38            IM = IP(I1,I+N2)
F39            IP(I2,I+J) = IM
F40            DO 130 K = 1, M
F41            CT = X(K,II) - X(K,IM)
F42            X(K,II) = X(K,II) + X(K,IM)
F43            X(K,IM) = CT * CX
F44  130CONTINUE
F45      L = 2 * L
F46      I1 = I2
F47      IF (L .LE. N2) GOTO 120
```

    Since the compiler may be worry that there are some aliasing problem among the variables, if EReg is used, the coding can be rescheduled such that the loading of X(K,II) and X(K,IM) in the F42 can be eliminated. Moreover, the explicit store of CT in F41 and the loading of CT and CX in F43 can be eliminated. If implicit store is supported (i.e using ADM model), all the remaining explicit store in F42-F43 can be changed to implicit store.

15. Kernel fifteen – Cholesky.f

    The main "Fortran" statements are

```
F39  DO 6 I = 0, NRHS
F40  DO 7 K = 0, N
F41     DO 8 L = 0, NMAT
F428       B(I,L,K) = B(I,L,K) * A(L,0,K)
F43        DO 7 JJ = 1, MIN (M, N-K)
F44           DO 7 L = 0, NMAT
F457             B(I,L,K+JJ) = B(I,L,K+JJ) - A(L,-JJ,K+JJ) * B(I,L,K)
F46C
F47  DO 6 K = N, 0, -1
F48     DO 9 L = 0, NMAT
F499       B(I,L,K) = B(I,L,K) * A(L,0,K)
F50  DO 6 JJ = 1, MIN (M, K)
F51     DO 6 L = 0, NMAT
F526       B(I,L,K-JJ) = B(I,L,K-JJ) - A(L,-JJ,K) * B(I,L,K)
```

There are four parts.

(a) F41 - F42 with 740952

(b) F44 - F45 with 4020016

(c) F48 - F49 with 740952

(d) F51 - F52 with 3710784

There are no aliasing problem in these four parts, but when the techniques used to reschedule the coding in part two, some dummy instructions disappear. Moreover, if ADM model is used, all the remaining explicit store in these four parts can be changed to implicit store since the destinations of these store have been loaded before.

B.3 Program analysed by ADS Model

1. Kernel One – Hydro fragment

   The main C statements are

```
for ( k=0;k<n;k++)
x[k]=q+y[k]*(r*z[k+10]+t*z[k+1]);
```

All the data can be prefetched before the operation of load instruction except the first few elements which is used to set the base address.

2. Kernel Two – ICCG excerpt (Incomplete Cholesky Conjugate Gradient)

   The main C statements are

```
i = ipntp - 1
for ( k=ipnt+1;k<ipntp;k=k+2) {
        i++;
        x[i]=x[k]-v[k]*x[k-1]-v[k+1]*x[k+1]
}
```

   The above program segment are translated into an assembly program code segment with an unrolling size 2 followed by another assembly program code segment without unrolling. By using EReg, the aliasing problem between x[k-1] and x[k+1] can be solved so that the number of load instructions can be reduced from 20 load instructions to 16 instructions. Moreover, all the data can be prefetched before the operation of load instruction except the first few elements which is used to set the base address.

3. Kernel Three – Inner product
   The main "C" statements are

```
for ( k=0;k<n;k++)
q+=z[k]+x[k];
```

   Since there are no aliasing problem, the coherence features cannot improve the performance, but all the data can be prefetched before the operation of load instruction except the first few elements which is used to set the base address

4. Kernel Four – Banded linear equations

   The main "C" statements are

```
temp=x[k-1];
for (j=4;j<n;j=j+5) {
        temp-=x[lw]*y[j];
        lw++;
}
x[k-1]=y[4]*temp;
```

There are no aliasing problem. Moreover, there is an overhead that the variable TEMP can be directly loaded from x[k-1]. Therefore, the total number of instructions in ADS model is increased slightly compared with original one using traditional registers. However, all the data can be prefetched before the operation of load instruction except the first few elements which is used to set the base address.

5. Kernel Five – Tri-diagonal Elimination, Below Diagonal

   The main "C" statements are :

```
for (i=1;i<n;i++)
x[i]=z[i]*( y[i]-x[i-1])
```

6. Kernel Six – General Linear recurrence equations

   The main "C" statements are :

```
for (i=1;i<n;i++)                    ... loop1
        for (k=0;k<i;k++)            ... loop2
        w[i]+=b[k][i]*w[(i-k)-1];
```

The value of w[i] is continued to be updated and stored within loop2 if traditional registers are used. The reason is that if the value of w[i] is not stored immediately after each " w[i]+=b[k][i]*w[(i-k)-1] " performed, error may result if other processes or instructions read data from the memory location of w[i] at that time. However, it will be possible to move the store instruction of w[i] outside loop2 if EReg is used. It is because every loading operations will check if the data have been already existed in EReg file. If yes, the data will be read from the EReg file directly; Otherwise, the date will be read from the specified memory location. Moreover, all the data can be prefetched before the operation of load instruction except the first few elements which is used to set the base address.

7. Kernel Seven – Equation of state fragment

   The main "C" statements are :

```
for (k=0;k<n;k++) {
x[k] =  u[k]+r*(z[k]+r*y[k]) +
        t*( u[k+3] + r*( u[k+2] + r*u[k+1] ) +
        t*( u[k+6] + r*( u[k+5] + r*u[k+4] )));
}
```

   There are two code scheduling algorithms. One is just to rearrange the load instructions such that although there are no improvement in the number of instructions issued, 12 load operations within the looping have changed from memory-register to register-register due to the basic property of EReg. These instructions are the loading instructions for u[k], u[k+1], u[k+2], u[k+3], u[k+4] and u[k+5]. While the other code scheduling algorithm is to reduce 6 load instructions, but only 2 load operations will change from memory-register to register-register due to the basic property of EReg. Thus, it is very difficult to say which code scheduling algorithm must be better. Moreover, all the data can be prefetched before the operation of load instruction except the first few elements which is used to set the base address.

8. Kernel Eight – ADi integration

   The main "C" statements are

```
for ( kx=1 ; kx<3 ; kx++ ){
        for ( ky=1 ; ky<n ; ky++ ) {
        du1[ky] = u1[nl1][ky+1][kx] - u1[nl1][ky-1][kx];
        du2[ky] = u2[nl1][ky+1][kx] - u2[nl1][ky-1][kx];
        du3[ky] = u3[nl1][ky+1][kx] - u3[nl1][ky-1][kx];
        u1[nl2][ky][kx]=
              u1[nl1][ky][kx]+a11*du1[ky]+a12*du2[ky]+a13*du3[ky] + sig*
              (u1[nl1][ky][kx+1]-2.0*u1[nl1][ky][kx]+u1[nl1][ky][kx-1]);
        u2[nl2][ky][kx]=
              u2[nl1][ky][kx]+a21*du1[ky]+a22*du2[ky]+a23*du3[ky] + sig*
              (u2[nl1][ky][kx+1]-2.0*u2[nl1][ky][kx]+u2[nl1][ky][kx-1]);
        u3[nl2][ky][kx]=
              u3[nl1][ky][kx]+a31*du1[ky]+a32*du2[ky]+a33*du3[ky] + sig*
C40                  (u3[nl1][ky][kx+1]-2.0*u3[nl1][ky][kx]+u3[nl1][ky][kx-1]);
C41           }
C42  }
```

The load instructions for du1[ky], du2[ky] and du3[ky] can be reduced from 3 (i.e C33,C36,C39) to 1 (i.e C33 only) if EReg is used. Moreover, the aliasing problem can be reduced between u1[nl1][ky-1][kx] & u[nl1][ky][kx+1], u2[nl1][ky-1][kx] & u2[nl1][ky][kx+1] and u3[nl1][ky-1][kx] & u3[nl1][ky][kx+1]. Moreover, all the data can be prefetched before the operation of load instruction except the first few elements which is used to set the base address.

9. Kernel Nine – Integrate predictors

   The main "C" statements are

```
C23  for ( i=0 ; i<n ; i++ ) {
C24           px[i][0] = dm28*px[i][12] + dm27*px[i][11] + dm26*px[i][10] +
C25                      dm25*px[i][ 9] + dm24*px[i][ 8] + dm23*px[i][ 7] +
C26                      dm22*px[i][ 6] + c0*( px[i][ 4] + px[i][ 5]) + px[i][ 2];
C27  }
```

   Although there are no aliasing problems within the whole program segment, all the data can be prefetched before the operation of load instruction except the first few elements which is used to set the base address.

10. Kernel Ten – Difference predictors

   The main "C" statements are

```
C22  for ( i=0 ; i<n ; i++ ) {
C23           ar = cx[i][ 4];
C24           br = ar - px[i][ 4];
C25           px[i][ 4] = ar;
C26           cr = br - px[i][ 5];
C27           px[i][ 5] = br;
C28           ar = cr - px[i][ 6];
C29           px[i][ 6] = cr;
C30           br = ar - px[i][ 7];
C31          px[i][ 7] = ar;
C32          cr = br - px[i][ 8];
C33          px[i][ 8] = br;
C34          ar = cr - px[i][ 9];
C35          px[i][ 9] = cr;
C36          br = ar - px[i][10];
C37          px[i][10] = ar;
C38          cr = br - px[i][11];
C39          px[i][11] = br;
C40          px[i][13] = cr - px[i][12];
C41          px[i][12] = cr;
C42}
```

There are no aliasing problem within the aboved program segment. Moreover, all the data can be prefetched before the operation of load instruction except the first few elements which is used to set the base address.

11. Kernel Eleven – First sum

    The main "C" statements are

```
C24  for (k=1;k<n;k++)
C25      x[k]=x[k-1]+y[k];
```

    There are no aliasing problem within the aboved program segment. Moreover, all the data can be prefetched before the operation of load instruction except the first few elements which is used to set the base address.

12. Kernel Twelve – First difference

    The main "C" statements are

```
C21  for (k=0;k<n;k++)
C22      x[k]=y[k+1]-y[k];
```

    Since the value of y[k+1] is equal to the value of y[k] in next looping, the number of load operations is reduced from two to one for each k after using EReg. The percentage of load instructions is nearly reduced by half. Moreover, all the data can be prefetched before the operation of load instruction except the first few elements which is used to set the base address.

13. Kernel Thirteen – Mxm.f

The main "Fortran" statements are

```
DO 110 J=1,M,4
   Do 110 K=1,N
      Do 110 I=1,L
         C(I,K) = C( I,K) + A(I,J) + B(J,K)
                  + A(I,J+1)*B(J+1,K)+A(I,J+2)*B(J+2,K)
                  + A(I,J+3)*B(J+3,K)
```

If the A,B and C are completely different (i.e No overlapping), no aliasing problem occur and hence no benefit obtained from using ERegs. Moreover, all the data can be prefetched before the operation of load instruction except the first few elements which is used to set the base address.

14. Kernel Fourteen – Cff2ttd1.f

The main "Fortran" statements are

```
F31   120I2 = 3 - I1
F32      DO 130 J = L, N2, L
F33         CX = W(J-L+1)
F34         IF (IS .LT. 0) CX = CONJG (CX)
F35            DO 130 I = J-L+1, J
F36            II = IP(I1,I)
F37            IP(I2,I+J-L) = II
F38            IM = IP(I1,I+N2)
F39            IP(I2,I+J) = IM
F40            DO 130 K = 1, M
F41            CT = X(K,II) - X(K,IM)
F42            X(K,II) = X(K,II) + X(K,IM)
F43            X(K,IM) = CT * CX
F44   130CONTINUE
F45      L = 2 * L
F46      I1 = I2
F47      IF (L .LE. N2) GOTO 120
```

Since the compiler may be worry that there are some aliasing problem among the variables, if EReg is used, the coding can be rescheduled such that the loading of

X(K,II) and X(K,IM) in the F42 can be eliminated. Moreover, the explicit store of CT in F41, the loading of CT and CX in F43 can be eliminated and some of the data can be prefetched before the operation of load instruction.

15. Kernel fifteen – Cholesky.f

The main "Fortran" statements are

```
F39   DO 6 I = 0, NRHS
F40   DO 7 K = 0, N
F41       DO 8 L = 0, NMAT
F428          B(I,L,K) = B(I,L,K) * A(L,0,K)
F43           DO 7 JJ = 1, MIN (M, N-K)
F44               DO 7 L = 0, NMAT
F457                  B(I,L,K+JJ) = B(I,L,K+JJ) - A(L,-JJ,K+JJ) * B(I,L,K)
F46C
F47   DO 6 K = N, 0, -1
F48       DO 9 L = 0, NMAT
F499          B(I,L,K) = B(I,L,K) * A(L,0,K)
F50   DO 6 JJ = 1, MIN (M, K)
F51       DO 6 L = 0, NMAT
F526          B(I,L,K-JJ) = B(I,L,K-JJ) - A(L,-JJ,K) * B(I,L,K)
```

There are four parts.

(a) F41 - F42 with 740952

(b) F44 - F45 with 4020016

(c) F48 - F49 with 740952

(d) F51 - F52 with 3710784

There are no aliasing problem in these four parts, but when the techniques used to schedule the coding in part two, some dummy instructions disappear. Most of the data can be prefetched before the operation of load instruction.

B.4 Program analysed by ADSM Model

1. Kernel One – Hydro fragment

   The main C statements are

   ```
   for ( k=0;k<n;k++)
   x[k]=q+y[k]*(r*z[k+10]+t*z[k+11]);
   ```

   z[k+10] & z[k+11] will be loaded for each k since there may be an aliasing problem between x[k] and z[k+11] or x[k] and z[k+12]. However, if EReg is used, we can schedule the coding such that the total number of 4 loading instructions z[k+11], z[k+10], z[k+12] and z[k+11] in one looping will be decreased to only 2 loading instructions z[k+11] and z[k+12] with ONE instruction outside the looping "for (k=0;k<n;k++)". Most of the data can be prefetched before the operation of load instruction except the first few elements which is used to set the base address. However, the STORE instructions can not be reduced. Therefore, the performance of ADSM model will be same as ADS model.

2. Kernel Two – ICCG excerpt (Incomplete Cholesky Conjugate Gradient)
   The main C statements are

   ```
   i = ipntp - 1
   for ( k=ipnt+1;k<ipntp;k=k+2) {
           i++;
           x[i]=x[k]-v[k]*x[k-1]-v[k+1]*x[k+1]
   }
   ```

   The above program segment are translated into an assembly program code segment with an unrolling size 2 followed by another assembly program code segment without unrolling. By using EReg, the aliasing problem between x[k-1] and x[k+1] can be solved so that the number of load instructions can be reduced from 20 load instructions to 16 instructions. x[i] will not be aliased to x[k-1], x[k] and x[k+1]. Therefore, no benefit can be obtained in performing implicit store, but all the data can be prefetched before the operation of load instruction except the first few elements which is used to set the base address

3. Kernel Three – Inner product

   The main "C" statements are

   ```
   for ( k=0;k<n;k++)
   q+=z[k]+x[k];
   ```

   Since there are no aliasing problem, the basic features in the ADSM model cannot improve the performance, but all the data can be prefetched before the operation of load instruction except the first few elements which is used to set the base address

4. Kernel Four – Banded linear equations

   The main "C" statements are

   ```
   temp=x[k-1];
   for (j=4;j<n;j=j+5) {
           temp-=x[lw]*y[j];
           lw++;
   }
   x[k-1]=y[4]*temp;
   ```

   There are no aliasing problem. Moreover, there is an overhead that the variable TEMP can be directly loaded from x[k-1]. Therefore, the total number of instructions in ADS model is increased slightly compared with original one using traditional registers. However, since the data of x[k-1] has been loaded, the explicit store "x[k-1]=y[4]*temp" can be changed to implicit store and hence the total number of instructions and total number of store instructions are reduced in using ADSM model. Nearly all the data can be prefetched before the operation of load instruction except the first few elements which is used to set the base address

5. Kernel Five – Tri-diagonal Elimination, Below Diagonal

   The main "C" statements are :

```
for (i=1;i<n;i++)
x[i]=z[i]*( y[i]-x[i-1])
```

All the store instructions can be reduced if the implicit store is supported, i.e ADM model and nearly all the data can be prefetched before the operation of load instruction except the first few elements which is used to set the base address

6. Kernel Six – General Linear recurrence equations

   The main "C" statements are :

```
for (i=1;i<n;i++)                    ... loop1
        for (k=0;k<i;k++)            ... loop2
        w[i]+=b[k][i]*w[(i-k)-1];
```

   The value of w[i] is continued to be updated and stored within loop2 if traditional registers are used. The reason is that if the value of w[i] is not stored immediately after each " w[i]+=b[k][i]*w[(i-k)-1] " performed, error may result if other processes or instructions read data from the memory location of w[i] at that time. However, it will be possible to move the store instruction of w[i] outside loop2 if EReg is used. It is because every loading operations will check if the data have been already existed in EReg file. If yes, the data will be read from the EReg file directly; Otherwise, the date will be read from the specified memory location. If implicit store is supported (i.e ADM model), nearly all explicit stores (storing the value of w[i]) can be changed to implicit stores. Moreover, all the data can be prefetched before the operation of load instruction except the first few elements which is used to set the base address

7. Kernel Seven – Equation of state fragment

   The main "C" statements are :

```
for (k=0;k<n;k++) {
x[k] =  u[k]+r*(z[k]+r*y[k]) +
        t*( u[k+3] + r*( u[k+2] + r*u[k+1] ) +
        t*( u[k+6] + r*( u[k+5] + r*u[k+4] )));
}
```

There are two code scheduling algorithms. One is just to rearrange the load instructions such that although there are no improvement in the number of instructions issued, 12 load operations within the looping have changed from memory-register to register-register due to the basic property of EReg. These instructions are the loading instructions for u[k], u[k+1], u[k+2], u[k+3], u[k+4] and u[k+5]. While the other code scheduling algorithm is to reduce 6 load instructions, but only 2 load operations will change from memory-register to register-register due to the basic property of EReg. Thus, it is very difficult to say which code scheduling algorithm must be better. Since the value of x[k] is not loaded, the explicit store of x[k] cannot be changed to implicit store. Thus, the extra feature of ADSM model is useless in this program. The data prefetching feature enable the data to be prefetched before the operation of load instruction except the first few elements which is used to set the base address.

8. Kernel Eight – ADi integration

   The main "C" statements are

```
C27  for ( kx=1 ; kx<3 ; kx++ ){
C28          for ( ky=1 ; ky<n ; ky++ ) {
C29          du1[ky] = u1[nl1][ky+1][kx] - u1[nl1][ky-1][kx];
C30          du2[ky] = u2[nl1][ky+1][kx] - u2[nl1][ky-1][kx];
C31          du3[ky] = u3[nl1][ky+1][kx] - u3[nl1][ky-1][kx];
C32          u1[nl2][ky][kx]=
C33               u1[nl1][ky][kx]+a11*du1[ky]+a12*du2[ky]+a13*du3[ky] + sig*
C34               (u1[nl1][ky][kx+1]-2.0*u1[nl1][ky][kx]+u1[nl1][ky][kx-1]);
C35          u2[nl2][ky][kx]=
C36               u2[nl1][ky][kx]+a21*du1[ky]+a22*du2[ky]+a23*du3[ky] + sig*
C37               (u2[nl1][ky][kx+1]-2.0*u2[nl1][ky][kx]+u2[nl1][ky][kx-1]);
C38          u3[nl2][ky][kx]=
C39               u3[nl1][ky][kx]+a31*du1[ky]+a32*du2[ky]+a33*du3[ky] + sig*
C40               (u3[nl1][ky][kx+1]-2.0*u3[nl1][ky][kx]+u3[nl1][ky][kx-1]);
C41          }
C42  }
```

The load instructions for du1[ky], du2[ky] and du3[ky] can be reduced from 3 (i.e C33,C36,C39) to 1 (i.e C33 only) if EReg is used. Moreover, the aliasing problem can be reduced between u1[nl1][ky-1][kx] & u[nl1][ky][kx+1], u2[nl1][ky-1][kx] & u2[nl1][ky][kx+1] and u3[nl1][ky-1][kx] & u3[nl1][ky][kx+1]. If implicit Store is supported by the EReg (i.e ADM model), the store instructions for du1[ky] (C29), du2[ky] (C30) and du3[ky] (C31) can be removed by first loading du1[ky], du2[ky] and du3[ky]. Moreover, the data prefetching feature enable the data to be prefetched before the operation of load instruction except the first few elements which is used to set the base address

9. Kernel Nine – Integrate predictors

   The main "C" statements are

```
C23  for ( i=0 ; i<n ; i++ ) {
C24          px[i][0] = dm28*px[i][12] + dm27*px[i][11] + dm26*px[i][10] +
C25                     dm25*px[i][ 9] + dm24*px[i][ 8] + dm23*px[i][ 7] +
C26                     dm22*px[i][ 6] + c0*( px[i][ 4] + px[i][ 5]) + px[i][ 2];
C27  }
```

   There are no aliasing problems, but the data prefetching feature enable the data to be prefetched before the operation of load instruction except the first few elements which is used to set the base address

10. Kernel Ten – Difference predictors

   The main "C" statements are

```
C22  for ( i=0 ; i<n ; i++ ) {
C23          ar = cx[i][ 4];
C24          br = ar - px[i][ 4];
C25          px[i][ 4] = ar;
C26          cr = br - px[i][ 5];
C27          px[i][ 5] = br;
C28          ar = cr - px[i][ 6];
C29          px[i][ 6] = cr;
C30          br = ar - px[i][ 7];
C31          px[i][ 7] = ar;
C32          cr = br - px[i][ 8];
C33          px[i][ 8] = br;
```

```
        ar = cr - px[i][ 9];
        px[i][ 9] = cr;
        br = ar - px[i][10];
        px[i][10] = ar;
        cr = br - px[i][11];
        px[i][11] = br;
        px[i][13] = cr - px[i][12];
        px[i][12] = cr;
}
```

There are no aliasing problem within the above program segment, but the data prefetching feature enable the data to be prefetched before the operation of load instruction except the first few elements which is used to set the base address

11. Kernel Eleven – First sum

    The main "C" statements are

    ```
    for (k=1;k<n;k++)
        x[k]=x[k-1]+y[k];
    ```

    Not only the data prefetching feature enable the data to be prefetched before the operation of load instruction, but all explicit stores of x[k] can be changed to implicit stores if the value of x[k] has been loaded before and the ADSM model is adopt.

12. Kernel Twelve – First difference

    The main "C" statements are

    ```
    for (k=0;k<n;k++)
        x[k]=y[k+1]-y[k];
    ```

    Since the value of y[k+1] is equal to the value of y[k] in next looping, the number of load operations is reduced from two to one for each k after using EReg. The percentage of load instructions is nearly reduced by half. Moreover, the data prefetching feature enable the data to be prefetched before the operation of load instruction. Since the value of x[k] hasn't been loaded before, all explicit stores of x[k] cannot be changed to implicit stores even if ADSM model is adopt.

13. Kernel Thirteen – Mxm.f

The main "Fortran" statements are

```
DO 110 J=1,M,4
   Do 110 K=1,N
      Do 110 I=1,L
         C(I,K) = C( I,K) + A(I,J) + B(J,K)
                  + A(I,J+1)*B(J+1,K)+A(I,J+2)*B(J+2,K)
                  + A(I,J+3)*B(J+3,K)
```

If the A,B and C are completely different (i.e No overlapping), no aliasing problem occur and hence no benefit obtained from using ERegs. However, if ADM model is used, the total of 1618247 store instructions will be reduced to the number of 35111 instructions only. This is due to the implicit store can be performed on the store of C(I,K) because C(I,K) has been read before the store operation. the data prefetching feature enable the data to be prefetched before the operation of load instruction except the first few elements which is used to set the base address,

14. Kernel Fourteen – Cff2ttd1.f

The main "Fortran" statements are

```
F31   120I2 = 3 - I1
F32      DO 130 J = L, N2, L
F33         CX = W(J-L+1)
F34         IF (IS .LT. 0) CX = CONJG (CX)
F35            DO 130 I = J-L+1, J
F36            II = IP(I1,I)
F37            IP(I2,I+J-L) = II
F38            IM = IP(I1,I+N2)
F39            IP(I2,I+J) = IM
F40            DO 130 K = 1, M
F41            CT = X(K,II) - X(K,IM)
F42            X(K,II) = X(K,II) + X(K,IM)
F43            X(K,IM) = CT * CX
F44   130CONTINUE
F45      L = 2 * L
F46      I1 = I2
F47      IF (L .LE. N2) GOTO 120
```

Since the compiler may be worry that there are some aliasing problem among the variables, if EReg is used, the coding can be rescheduled such that the loading of X(K,II) and X(K,IM) in the F42 can be eliminated. The data prefetching feature enable the data to be prefetched before the operation of load instruction except the first few elements which is used to set the base address. Moreover, the explicit store of CT in F41 and the loading of CT and CX in F43 can be eliminated. If implicit store is supported (i.e using ADSM model), all the remaining explicit stōre in F42-F43 can be changed to implicit store.

15. Kernel fifteen – Cholesky.f

    The main "Fortran" statements are

```
F39   DO 6 I = 0, NRHS
F40   DO 7 K = 0, N
F41      DO 8 L = 0, NMAT
F428        B(I,L,K) = B(I,L,K) * A(L,0,K)
F43         DO 7 JJ = 1, MIN (M, N-K)
F44            DO 7 L = 0, NMAT
F457              B(I,L,K+JJ) = B(I,L,K+JJ) - A(L,-JJ,K+JJ) * B(I,L,K)
F46C
F47   DO 6 K = N, 0, -1
F48      DO 9 L = 0, NMAT
F499        B(I,L,K) = B(I,L,K) * A(L,0,K)
F50   DO 6 JJ = 1, MIN (M, K)
F51      DO 6 L = 0, NMAT
F526        B(I,L,K-JJ) = B(I,L,K-JJ) - A(L,-JJ,K) * B(I,L,K)
```

    There are four parts.

    (a) F41 - F42 with 740952

    (b) F44 - F45 with 4020016

    (c) F48 - F49 with 740952

    (d) F51 - F52 with 3710784

There are no aliasing problem in these four parts, but when the techniques used to reschedule the coding in part two, some dummy instructions disappear. The data prefetching feature enable the data to be prefetched before the operation of load instruction except the first few elements which is used to set the base address. Moreover, if ADSM model is used, all the remaining explicit store in these four parts can be changed to implicit store since the destinations of these store have been loaded before.

B.5 Program analysed by IADSM Model

1. Kernel One - Hydro fragment
   The main C statements are

   ```
   for ( k=0;k¡n;k++)
   x[k]=q+y[k]*(r*z[k+10]+t*z[k+1]);
   ```

   The assembly version of this program contain two parts. The first part is unrolled by 2. The second part doesn't has any unrolling. Before entering the loop, the stride values of the ERegs ,which is used to store data and perform implicitly loading operation, are set by the compiler. Moreover, the data y[0],y[1], z[10] and z[11] must be explicitly loaded in the first part must be explicity loaded to initialize the starting address of the data such that the remaining data in both first and second parts can be prefetched. By setting the instruction location to the Inst_A fields of these ERegs, the implicitly loading will be performed when the content of program counter match with this instruction location.

2. Kernel Two - ICCG excerpt (Incomplete Cholesky Conjugate Gradient)
   The main C statements are

   ```
   i = ipntp - 1
   for ( k=ipnt+1;k¡ipntp;k=k+2) {
           i++;
           x[i]=x[k]-v[k]*x[k-1]-v[k+1]*x[k+1]
   }
   ```

The above program segment are translated into an assembly program code segment with an unrolling size of 2 followed by another assembly program code segment without unrolling. By using EReg, the aliasing problem between x[k-1] and x[k+1] can be solved so that the number of load instructions can be reduced from 20 load instructions to 16 instructions. Moreover, all the data can be prefetched and implicitly loaded except the first few elements which is used to set the base address. These elements include x[ipnt+1],x[ipnt],v[ipnt+1] and v[ipnt+2].

3. Kernel Three – Inner product

   The main "C" statements are

```
for ( k=0;k¡n;k++)
q+=z[k]+x[k];
```

   Since there are no aliasing problem, the coherence features cannot improve the performance, but all the data can be prefetched and implicitly loaded except z[0] and x[0] which is used to set the base address.

4. Kernel Four – Banded linear equations

   The main "C" statements are

```
temp=x[k-1];
for (j=4;j¡n;j=j+5) {
        temp-=x[lw]*y[j];
        lw++;
}
x[k-1]=y[4]*temp;
```

   There are no aliasing problem. Moreover, there is an overhead that the variable TEMP can be directly loaded from x[k-1]. Therefore, the total number of instructions in ADS model is increased slightly compared with original one using traditional registers. However, all the data in the innermost loop can be prefetched and implicitlly loaded except x[k-6] and y[4] which are used to set the base address.

5. Kernel Five – Tri-diagonal Elimination, Below Diagonal

   The main "C" statements are :

   ```
   for (i=1;i¡n;i++)
   x[i]=z[i]*( y[i]-x[i-1])
   ```

   Since every result x[i] must be stored explicitly and each x[i-1] is loaded only once, there is no aliasing problem to be reduced. However, all the data can be prefetched and implicitlly loaded except the first few elements which is used to set the base address.

6. Kernel Six – General Linear recurrence equations

   The main "C" statements are :

   ```
   for (i=1;i¡n;i++)                    ... loop1
           for (k=0;k¡i;k++)            ... loop2
           w[i]+=b[k][i]*w[(i-k)-1];
   ```

   The value of w[i] is continued to be updated and stored within loop2 if traditional registers are used. The reason is that if the value of w[i] is not stored immediately after each " w[i]+=b[k][i]*w[(i-k)-1] " performed, error may result if other processes or instructions read data from the memory location of w[i] at that time. However, it will be possible to move the store instruction of w[i] outside loop2 if EReg is used. It is because every loading operations will check if the data have been already existed in EReg file. If yes, the data will be read from the EReg file directly; Otherwise, the date will be read from the specified memory location. Moreover, all the data can be prefetched before the operation of load instruction except the first few elements which is used to set the base address.

7. Kernel Seven – Equation of state fragment

   The main "C" statements are :

```
for (k=0;k¡n;k++) {
x[k] =  u[k]+r*(z[k]+r*y[k]) +
        t*( u[k+3] + r*( u[k+2] + r*u[k+1] ) +
        t*( u[k+6] + r*( u[k+5] + r*u[k+4] )));
}
```

There are two code scheduling algorithms. One is just to rearrange the load instructions such that although there are no improvement in the number of instructions issued, 12 load operations within the looping have changed from memory-register to register-register due to the basic property of EReg. These instructions are the loading instructions for u[k], u[k+1], u[k+2], u[k+3], u[k+4] and u[k+5]. While the other code scheduling algorithm is to reduce 6 load instructions, but only 2 load operations will change from memory-register to register-register due to the basic property of EReg. Thus, it is very difficult to say which code scheduling algorithm must be better. Moreover, all the data can be prefetched and implicitlly loaded except the first few elements which is used to set the base address.

8. Kernel Eight – ADi integration

The main "C" statements are

```
C27 for ( kx=1 ; kx¡3 ; kx++ ){
C28         for ( ky=1 ; ky¡n ; ky++ ) {
C29         du1[ky] = u1[nl1][ky+1][kx] - u1[nl1][ky-1][kx];
C30         du2[ky] = u2[nl1][ky+1][kx] - u2[nl1][ky-1][kx];
C31         du3[ky] = u3[nl1][ky+1][kx] - u3[nl1][ky-1][kx];
C32         u1[nl2][ky][kx]=
C33             u1[nl1][ky][kx]+a11*du1[ky]+a12*du2[ky]+a13*du3[ky] + sig*
C34             (u1[nl1][ky][kx+1]-2.0*u1[nl1][ky][kx]+u1[nl1][ky][kx-1]);
C35         u2[nl2][ky][kx]=
C36             u2[nl1][ky][kx]+a21*du1[ky]+a22*du2[ky]+a23*du3[ky] + sig*
C37             (u2[nl1][ky][kx+1]-2.0*u2[nl1][ky][kx]+u2[nl1][ky][kx-1]);
C38         u3[nl2][ky][kx]=
C39             u3[nl1][ky][kx]+a31*du1[ky]+a32*du2[ky]+a33*du3[ky] + sig*
C40             (u3[nl1][ky][kx+1]-2.0*u3[nl1][ky][kx]+u3[nl1][ky][kx-1]);
C41         }
C42 }
```

The load instructions for du1[ky], du2[ky] and du3[ky] can be reduced from 3 (i.e C33,C36,C39) to 1 (i.e C33 only) if EReg is used. Moreover, the aliasing problem can be reduced between u1[nl1][ky-1][kx] & u[nl1][ky][kx+1], u2[nl1][ky-1][kx] & u2[nl1][ky][kx+1] and u3[nl1][ky-1][kx] & u3[nl1][ky][kx+1]. Moreover, all the data can be prefetched and implicitly loaded except the first few elements which is used to set the base address.

9. Kernel Nine – Integrate predictors

   The main "C" statements are

```
C23  for ( i=0 ; i¡n ; i++ ) {
C24          px[i][0] = dm28*px[i][12] + dm27*px[i][11] + dm26*px[i][10] +
C25                          dm25*px[i][ 9] + dm24*px[i][ 8] + dm23*px[i][ 7] +
C26                          dm22*px[i][ 6] + c0*( px[i][ 4] + px[i][ 5]) + px[i][ 2];
C27  }
```

   Although there are no aliasing problems within the whole program segment, all the data can be prefetched and implicitly loaded except the first few elements which is used to set the base address.

10. Kernel Ten – Difference predictors

    The main "C" statements are

```
C22  for ( i=0 ; i¡n ; i++ ) {
C23          ar = cx[i][ 4];
C24          br = ar - px[i][ 4];
C25          px[i][ 4] = ar;
C26          cr = br - px[i][ 5];
C27          px[i][ 5] = br;
C28          ar = cr - px[i][ 6];
C29          px[i][ 6] = cr;
C30          br = ar - px[i][ 7];
C31          px[i][ 7] = ar;
C32          cr = br - px[i][ 8];
C33          px[i][ 8] = br;
C34          ar = cr - px[i][ 9];
C35          px[i][ 9] = cr;
```

```
        br = ar - px[i][10];
        px[i][10] = ar;
        cr = br - px[i][11];
        px[i][11] = br;
        px[i][13] = cr - px[i][12];
        px[i][12] = cr;
}
```

There are no aliasing problem within the aboved program segment. Moreover, all the data can be prefetched and implicitly loaded except the first few elements which is used to set the base address.

11. Kernel Eleven – First sum

    The main "C" statements are

    ```
    for (k=1;k¡n;k++)
        x[k]=x[k-1]+y[k];
    ```

    There are no aliasing problem within the aboved program segment. Moreover, all the data can be prefetched and implicitly loaded except the first few elements which is used to set the base address.

12. Kernel Twelve – First difference

    The main "C" statements are

    ```
    for (k=0;k¡n;k++)
        x[k]=y[k+1]-y[k];
    ```

    Since the value of y[k+1] is equal to the value of y[k] in next looping, the number of load operations is reduced from two to one for each k after using EReg. The percentage of load instructions is nearly reduced by half. Moreover, all the data can be and implicitly loaded except the first few elements which is used to set the base address.

13. Kernel Thirteen – Mxm.f

The main "Fortran" statements are

```
DO 110 J=1,M,4
   Do 110 K=1,N
      Do 110 I=1,L
         C(I,K) = C( I,K) + A(I,J) + B(J,K)
                  + A(I,J+1)*B(J+1,K)+A(I,J+2)*B(J+2,K)
                  + A(I,J+3)*B(J+3,K)
```

If the A,B and C are completely different (i.e No overlapping), no aliasing problem occur and hence no benefit obtained from using ERegs. Moreover, all the data can be prefetched and implicitly loaded except the first few elements which is used to set the base address.

14. Kernel Fourteen – Cff2ttd1.f

The main "Fortran" statements are

```
F31   120I2 = 3 - I1
F32      DO 130 J = L, N2, L
F33         CX = W(J-L+1)
F34         IF (IS .LT. 0) CX = CONJG (CX)
F35            DO 130 I = J-L+1, J
F36            II = IP(I1,I)
F37            IP(I2,I+J-L) = II
F38            IM = IP(I1,I+N2)
F39            IP(I2,I+J) = IM
F40            DO 130 K = 1, M
F41            CT = X(K,II) - X(K,IM)
F42            X(K,II) = X(K,II) + X(K,IM)
F43            X(K,IM) = CT * CX
F44   130CONTINUE
F45      L = 2 * L
F46      I1 = I2
F47      IF (L .LE. N2) GOTO 120
```

Since the compiler may be worry that there are some aliasing problem among the variables, if EReg is used, the coding can be rescheduled such that the loading of X(K,II) and X(K,IM) in the F42 can be eliminated. Moreover, the explicit store of

CT in F41, the loading of CT and CX in F43 can be eliminated and some of the data can be prefetched and implicitly loaded.

15. Kernel fifteen – Cholesky.f

    The main "Fortran" statements are

```
F39   DO 6 I = 0, NRHS
F40   DO 7 K = 0, N
F41      DO 8 L = 0, NMAT
F428        B(I,L,K) = B(I,L,K) * A(L,0,K)
F43         DO 7 JJ = 1, MIN (M, N-K)
F44            DO 7 L = 0, NMAT
F457              B(I,L,K+JJ) = B(I,L,K+JJ) - A(L,-JJ,K+JJ) * B(I,L,K)
F46C
F47   DO 6 K = N, 0, -1
F48      DO 9 L = 0, NMAT
F499        B(I,L,K) = B(I,L,K) * A(L,0,K)
F50   DO 6 JJ = 1, MIN (M, K)
F51      DO 6 L = 0, NMAT
F526        B(I,L,K-JJ) = B(I,L,K-JJ) - A(L,-JJ,K) * B(I,L,K)
```

    There are four parts.

    (a) F41 - F42 with 740952

    (b) F44 - F45 with 4020016

    (c) F48 - F49 with 740952

    (d) F51 - F52 with 3710784

    There are no aliasing problem in these four parts, but when the techniques used to reschedule the coding in part two, some dummy instructions disappear. Most of the data can be prefetched and implicitly loaded.

B.6 Program analysed by IADSMC&IDLC Model

Since the method of reduce the instructions for the looping by the model is very standard, No individual discussion as previous chapter will be carried out.

1. Kernel One – Hydro fragment
   The main C statements are

   ```
   for ( k=0;k¡n;k++)
   x[k]=q+y[k]*(r*z[k+10]+t*z[k+1]);
   ```

   The assembly version of this program contain two parts. The first part is unrolled by 2. The second part doesn't has any unrolling. Before entering the loop, the stride values of the ERegs ,which is used to store data and perform implicitly loading operation, are set by the compiler. Moreover, the data y[0],y[1], z[10] and z[11] must be explicitly loaded in the first part must be explicity loaded to initialize the starting address of the data such that the remaining data in both first and second parts can be prefetched. By setting the instruction location to the Inst_A fields of these ERegs, the implicitly loading will be performed when the content of program counter match with this instruction location. Moreover, there are several instructions to be reduced:

   (a) Instructions which initialize the index k.

   (b) The explicit branch instructions which occur when the value of k is smaller than the upper limit n.

2. Kernel Two – ICCG excerpt (Incomplete Cholesky Conjugate Gradient)
   The main C statements are

   ```
   i = ipntp - 1
   for ( k=ipnt+1;k¡ipntp;k=k+2) {
           i++;
           x[i]=x[k]-v[k]*x[k-1]-v[k+1]*x[k+1]
   }
   ```

   The above program segment are translated into an assembly program code segment with an unrolling size of 2 followed by another assembly program code segment without unrolling. By using EReg, the aliasing problem between x[k-1] and x[k+1] can be solved so that the number of load instructions can be reduced from 20 load instructions to 16 instructions. Moreover, all the data can be prefetched and implicitly loaded except the first few elements which is used to set the base address. These elements include x[ipnt+1],x[ipnt],v[ipnt+1] and v[ipnt+2]. Moreover, there are several instructions to be reduced:

(a) Instructions which initialize the index k.

(b) The explicit branch instructions which occur when the value of k is smaller than the upper limit ipntp.

3. Kernel Three – Inner product

   The main "C" statements are

```
for ( k=0;k¡n;k++)
q+=z[k]+x[k];
```

   Since there are no aliasing problem, the coherence features cannot improve the performance, but all the data can be prefetched and implicitly loaded except z[0] and x[0] which is used to set the base address. Moreover, there are several instructions to be reduced:

   (a) Instructions which initialize the index k.

   (b) The explicit branch instructions which occur when the value of k is smaller than the upper limit n.

4. Kernel Four – Banded linear equations

   The main "C" statements are

```
temp=x[k-1];
for (j=4;j¡n;j=j+5) {
        temp-=x[lw]*y[j];
        lw++;
}
x[k-1]=y[4]*temp;
```

There are no aliasing problem. Moreover, there is an overhead that the variable TEMP can be directly loaded from x[k-1]. Therefore, the total number of instructions in ADS model is increased slightly compared with original one using traditional registers. However, all the data in the innermost loop can be prefetched and implicitlly loaded except x[k-6] and y[4] which are used to set the base address. Moreover, there are several instructions to be reduced:

(a) Instructions which initialize the index k.

(b) The explicit branch instructions which occur when the value of k is smaller than the upper limit n

5. Kernel Five – Tri-diagonal Elimination, Below Diagonal

The main "C" statements are :

```
for (i=1;i¡n;i++)
x[i]=z[i]*( y[i]-x[i-1])
```

Since every result x[i] must be stored explicitly and each x[i-1] is loaded only once, there is no aliasing problem to be reduced. However, all the data can be prefetched and implicitlly loaded except the first few elements which is used to set the base address. Moreover, there are several instructions to be reduced:

(a) Instructions which initialize the index i.

(b) The explicit branch instructions which occur when the value of i is smaller than the upper limit n

6. Kernel Six – General Linear recurrence equations

The main "C" statements are :

```
for (i=1;i¡n;i++)                    ... loop1
          for (k=0;k¡i;k++)          ... loop2
          w[i]+=b[k][i]*w[(i-k)-1];
```

The value of w[i] is continued to be updated and stored within loop2 if traditional registers are used. The reason is that if the value of w[i] is not stored immediately after each " w[i]+=b[k][i]*w[(i-k)-1] " performed, error may result if other processes or instructions read data from the memory location of w[i] at that time. However, it will be possible to move the store instruction of w[i] outside loop2 if EReg is used. It is because every loading operations will check if the data have been already existed in EReg file. If yes, the data will be read from the EReg file directly; Otherwise, the date will be read from the specified memory location. Moreover, all the data can be prefetched before the operation of load instruction except the first few elements which is used to set the base address. Moreover, there are several instructions to be reduced:

(a) Instructions which initialize the index i.

(b) The explicit branch instructions which occur when the value of i is smaller than the upper limit n

7. Kernel Seven – Equation of state fragment

   The main "C" statements are :

```
for (k=0;k¡n;k++) {
x[k] =  u[k]+r*(z[k]+r*y[k]) +
        t*( u[k+3] + r*( u[k+2] + r*u[k+1] ) +
        t*( u[k+6] + r*( u[k+5] + r*u[k+4] )));
}
```

   There are two code scheduling algorithms. One is just to rearrange the load instructions such that although there are no improvement in the number of instructions issued, 12 load operations within the looping have changed from memory-register to register-register due to the basic property of EReg. These instructions are the loading instructions for u[k], u[k+1], u[k+2], u[k+3], u[k+4] and u[k+5]. While the other code scheduling algorithm is to reduce 6 load instructions, but only 2 load operations will change from memory-register to register-register due to the basic property of EReg. Thus, it is very difficult to say which code scheduling algorithm must be better. Moreover, all the data can be prefetched and implicitlly loaded except the first few elements which is used to set the base address. Moreover, there are several instructions to be reduced:

(a) Instructions which initialize the index k.

(b) The explicit branch instructions which occur when the value of k is smaller than the upper limit n

8. Kernel Eight – ADi integration

The main "C" statements are

```
C27  for ( kx=1 ; kx¡3 ; kx++ ){
C28          for ( ky=1 ; ky¡n ; ky++ ) {
C29          du1[ky] = u1[nl1][ky+1][kx] - u1[nl1][ky-1][kx];
C30          du2[ky] = u2[nl1][ky+1][kx] - u2[nl1][ky-1][kx];
C31          du3[ky] = u3[nl1][ky+1][kx] - u3[nl1][ky-1][kx];
C32          u1[nl2][ky][kx]=
C33               u1[nl1][ky][kx]+a11*du1[ky]+a12*du2[ky]+a13*du3[ky] + sig*
C34               (u1[nl1][ky][kx+1]-2.0*u1[nl1][ky][kx]+u1[nl1][ky][kx-1]);
C35          u2[nl2][ky][kx]=
C36               u2[nl1][ky][kx]+a21*du1[ky]+a22*du2[ky]+a23*du3[ky] + sig*
C37               (u2[nl1][ky][kx+1]-2.0*u2[nl1][ky][kx]+u2[nl1][ky][kx-1]);
C38          u3[nl2][ky][kx]=
C39               u3[nl1][ky][kx]+a31*du1[ky]+a32*du2[ky]+a33*du3[ky] + sig*
C40               (u3[nl1][ky][kx+1]-2.0*u3[nl1][ky][kx]+u3[nl1][ky][kx-1]);
C41          }
C42  }
```

The load instructions for du1[ky], du2[ky] and du3[ky] can be reduced from 3 (i.e C33,C36,C39) to 1 (i.e C33 only) if EReg is used. Moreover, the aliasing problem can be reduced between u1[nl1][ky-1][kx] & u[nl1][ky][kx+1], u2[nl1][ky-1][kx] & u2[nl1][ky][kx+1] and u3[nl1][ky-1][kx] & u3[nl1][ky][kx+1]. Moreover, all the data can be prefetched and implicitly loaded except the first few elements which is used to set the base address. Moreover, there are several instructions to be reduced:

(a) Instructions which initialize the indices kx and ky.

(b) The explicit branch instructions which occur when the values of kx and ky are smaller than the upper limit of 3 and n respectively.

9. Kernel Nine – Integrate predictors

The main "C" statements are

```
for ( i=0 ; i¡n ; i++ ) {
        px[i][0] = dm28*px[i][12] + dm27*px[i][11] + dm26*px[i][10] +
                   dm25*px[i][ 9] + dm24*px[i][ 8] + dm23*px[i][ 7] +
                   dm22*px[i][ 6] + c0*( px[i][ 4] + px[i][ 5]) + px[i][ 2];
}
```

Although there are no aliasing problems within the whole program segment, all the data can be prefetched and implicitly loaded except the first few elements which is used to set the base address. Moreover, there are several instructions to be reduced:

(a) Instructions which initialize the index i.

(b) The explicit branch instructions which occur when the value i is smaller than the upper limit n.

10. Kernel Ten – Difference predictors

The main "C" statements are

```
for ( i=0 ; i¡n ; i++ ) {
        ar = cx[i][ 4];
        br = ar - px[i][ 4];
        px[i][ 4] = ar;
        cr = br - px[i][ 5];
        px[i][ 5] = br;
        ar = cr - px[i][ 6];
        px[i][ 6] = cr;
        br = ar - px[i][ 7];
        px[i][ 7] = ar;
        cr = br - px[i][ 8];
        px[i][ 8] = br;
        ar = cr - px[i][ 9];
        px[i][ 9] = cr;
        br = ar - px[i][10];
        px[i][10] = ar;
        cr = br - px[i][11];
        px[i][11] = br;
        px[i][13] = cr - px[i][12];
        px[i][12] = cr;
}
```

There are no aliasing problem within the aboved program segment. Moreover, all the data can be prefetched and implicitly loaded except the first few elements which is used to set the base address. Moreover, there are several instructions to be reduced:

(a) Instructions which initialize the index i.

(b) The explicit branch instructions which occur when the value i is smaller than the upper limit n.

11. Kernel Eleven – First sum

The main "C" statements are

```
for (k=1;k¡n;k++)
    x[k]=x[k-1]+y[k];
```

There are no aliasing problem within the aboved program segment. Moreover, all the data can be prefetched and implicitly loaded except the first few elements which is used to set the base address. Moreover, there are several instructions to be reduced:

(a) Instructions which initialize the index k.

(b) The explicit branch instructions which occur when the value k is smaller than the upper limit n.

12. Kernel Twelve – First difference

The main "C" statements are

```
for (k=0;k¡n;k++)
    x[k]=y[k+1]-y[k];
```

Since the value of y[k+1] is equal to the value of y[k] in next looping, the number of load operations is reduced from two to one for each k after using EReg. The percentage of load instructions is nearly reduced by half. Moreover, all the data can be and implicitly loaded except the first few elements which is used to set the base address. Moreover, there are several instructions to be reduced:

(a) Instructions which initialize the index k.

(b) The explicit branch instructions which occur when the value k is smaller than the upper limit n.

13. Kernel Thirteen - Mxm.f

The main "Fortran" statements are

```
DO 110 J=1,M,4
   Do 110 K=1,N
      Do 110 I=1,L
         C(I,K) = C( I,K) + A(I,J) + B(J,K)
                  + A(I,J+1)*B(J+1,K)+A(I,J+2)*B(J+2,K)
                  + A(I,J+3)*B(J+3,K)
```

If the A,B and C are completely different (i.e No overlapping), no aliasing problem occur and hence no benefit obtained from using ERegs. Moreover, all the data can be prefetched and implicitly loaded except the first few elements which is used to set the base address. Moreover, there are several instructions to be reduced:

(a) Instructions which initialize the index I,J and K.

(b) The explicit branch instructions which occur when the values of I,J and K are smaller than the upper limit of L,M and N respectively.

14. Kernel Fourteen - Cff2ttd1.f

The main "Fortran" statements are

```
F31   120I2 = 3 - I1
F32      DO 130 J = L, N2, L
```

```
F33         CX = W(J-L+1)
F34         IF (IS .LT. 0) CX = CONJG (CX)
F35           DO 130 I = J-L+1, J
F36           II = IP(I1,I)
F37           IP(I2,I+J-L) = II
F38           IM = IP(I1,I+N2)
F39           IP(I2,I+J) = IM
F40           DO 130 K = 1, M
F41           CT = X(K,II) - X(K,IM)
F42           X(K,II) = X(K,II) + X(K,IM)
F43           X(K,IM) = CT * CX
F44     130CONTINUE
F45       L = 2 * L
F46       I1 = I2
F47       IF (L .LE. N2) GOTO 120
```

Since the compiler may be worry that there are some aliasing problem among the variables, if EReg is used, the coding can be rescheduled such that the loading of X(K,II) and X(K,IM) in the F42 can be eliminated. Moreover, the explicit store of CT in F41, the loading of CT and CX in F43 can be eliminated and some of the data can be prefetched and implicitly loaded. Moreover, there are several instructions to be reduced:

(a) Instructions which initialize the index I,J and K.

(b) The explicit branch instructions which occur when the values of I,J and K are smaller than the upper limit of J-L+1,N_2 and M respectively.

15. Kernel fifteen – Cholesky.f

The main "Fortran" statements are

```
F39   DO 6 I = 0, NRHS
F40   DO 7 K = 0, N
F41     DO 8 L = 0, NMAT
F428      B(I,L,K) = B(I,L,K) * A(L,0,K)
F43       DO 7 JJ = 1, MIN (M, N-K)
F44         DO 7 L = 0, NMAT
F457          B(I,L,K+JJ) = B(I,L,K+JJ) - A(L,-JJ,K+JJ) * B(I,L,K)
F46C
```

```
F47   DO 6 K = N, 0, -1
F48      DO 9 L = 0, NMAT
F499        B(I,L,K) = B(I,L,K) * A(L,0,K)
F50   DO 6 JJ = 1, MIN (M, K)
F51      DO 6 L = 0, NMAT
F526        B(I,L,K-JJ) = B(I,L,K-JJ) - A(L,-JJ,K) * B(I,L,K)
```

There are four parts.

(a) F41 - F42 with 740952

(b) F44 - F45 with 4020016

(c) F48 - F49 with 740952

(d) F51 - F52 with 3710784

There are no aliasing problem in these four parts, but when the techniques used to reschedule the coding in part two, some dummy instructions disappear. Most of the data can be prefetched and implicitly loaded. Moreover, there are several instructions to be reduced:

(a) Instructions which initialize the index I,J and K.

(b) The explicit branch instructions which occur when the values of I,J and K are smaller than their corresponding upper limits in different parts respectively.

Appendix C

Cache Simulation on Prefetching of ADS model

Kernel	Total no of *LOADs*	No Prefetch	ERegs' Prefetch
1. Hydro fragment	4075	25.49693252%	0.957055215%
2. ICCG excerpt	8040	22.08955224%	0.534825871%
3. Inner product	4060	25.51724138%	0.985221675%
4. Banded linear equations	2472	21.76375405%	1.982200647%
5. Tri-diagonal elimination , below diagonal	6054	17.1291708%	0.67723819%
6. General linear recurrence equations	2000056	28.9855384%	8.543610779%
7. Equation of state fragment	18069	8.517350158%	0.215839283%
8. ADI integration	70063	17.18738849%	0.085637212%
9. Integrate predictors	20086	30.05078164%	0.209100866&
10. Difference predictors	20069	30.07623698%	0.209277991%
11. First Sum	4058	13.20847708%	0.936421883%
12. First Difference	2062	26.04267701%	1.842870999%
13. Mxm.f	5807711	14.24728262%	9.491691305%
14. Cff2td1.f	32761398	19.39192583%	1.520298371%
15. Cholesky Decomposition / substitution	2868556	9.003101212%	6.033976677%

Table C.1: Cache Miss at Cache Size = 8KB, Block Size = 16B, 2-Way Associative

Kernel	Total no of *LOADs*	No Prefetch	ERegs' Prefetch
1. Hydro fragment	4075	25.49693252%	0.957055215%
2. ICCG excerpt	8040	22.08955224%	0.534825871%
3. Inner product	4060	25.51724138%	0.985221675%
4. Banded linear equations	2472	21.72330097%	1.982200647%
5. Tri-diagonal elimination , below diagonal	6054	17.1291708%	0.67723819%
6. General linear recurrence equations	2000056	28.9855384%	9.423086154%
7. Equation of state fragment	18069	8.517350158%	0.215839283%
8. ADI integration	70063	17.18596121%	0.084209925%
9. Integrate predictors	20086	30.05078164%	0.209100866%
10. Difference predictors	20069	30.07623698%	0.209277991%
11. First Sum	4058	13.20847708%	0.936421883%
12. First Difference	2062	26.04267701%	1.842870999%
13. Mxm.f	5807711	22.99702929%	18.58856269%
14. Cff2td1.f	32761398	19.1491523%	1.278889259%
15. Cholesky Decomposition / substitution	2868556	8.974794287%	6.003403803%

Table C.2: Cache Miss at Cache Size = 8KB, Block Size = 16B, 4-Way Associative

Kernel	Total no of *LOADs*	No Prefetch	ERegs' Prefetch
1. Hydro fragment	4075	12.88343558%	0.613496933%
2. ICCG excerpt	8040	11.1318408%	0.323383085%
3. Inner product	4060	12.95566502%	0.640394089%
4. Banded linear equations	2472	18.56796117%	1.45631068%
5. Tri-diagonal elimination , below diagonal	6054	8.704988437%	0.445986125%
6. General linear recurrence equations	2000056	28.3356566%	5.384949221%
7. Equation of state fragment	18069	4.294648293%	0.149427196%
8. ADI integration	70063	10.75032471%	0.058518762%
9. Integrate predictors	20086	17.5445584%	0.134421985%
10. Difference predictors	20069	20.05082465%	0.139518661%
11. First Sum	4058	6.776737309%	0.616067028%
12. First Difference	2062	13.33656644%	1.212415131%
13. Mxm.f	5807711	29.69586469%	27.4754374%
14. Cff2td1.f	32761398	9.644310661%	0.721046764%
15. Cholesky Decomposition / substitution	2868556	5.811530261%	3.138164289%

Table C.3: Cache Miss at Cache Size = 8KB, Block Size = 32B, 4-Way Associative

Kernel	Total no of *LOADs*	No Prefetch	ERegs' Prefetch
1. Hydro fragment	4075	12.88343558%	0.613496933%
2. ICCG excerpt	8040	9.614427861%	0.323383085%
3. Inner product	4060	12.90640394%	0.640394089%
4. Banded linear equations	2472	15.21035599%	1.375404531%
5. Tri-diagonal elimination , below diagonal	6054	8.671952428%	0.445986125%
6. General linear recurrence equations	2000056	26.9678949%	3.690646662%
7. Equation of state fragment	18069	4.300182633%	0.149427196%
8. ADI integration	70063	10.74889742%	0.058518762%
9. Integrate predictors	20086	17.5445584%	0.134421985%
10. Difference predictors	20069	20.05082465%	0.139518661%
11. First Sum	4058	6.752094628%	0.616067028%
12. First Difference	2062	13.33656644%	1.212415131%
13. Mxm.f	5807711	2.579622161%	0.077035514%
14. Cff2td1.f	32761398	9.611500706%	0.683832235%
15. Cholesky Decomposition / substitution	2868556	3.830568411%	2.340341273%

Table C.4: Cache Miss at Cache Size = 16KB, Block Size = 32B, 4-Way Associative

Kernel	Total no of *LOADs*	No Prefetch	ERegs' Prefetch
1. Hydro fragment	4075	12.88343558%	0.613496933%
2. ICCG excerpt	8040	9.614427861%	0.323383085%
3. Inner product	4060	12.95566502%	0.640394089%
4. Banded linear equations	2472	15.21035599%	1.375404531%
5. Tri-diagonal elimination , below diagonal	6054	8.704988437%	0.445986125%
6. General linear recurrence equations	2000056	22.41762231%	2.905768638%
7. Equation of state fragment	18069	4.294648293%	0.149427196%
8. ADI integration	70063	10.75032471%	0.058518762%
9. Integrate predictors	20086	17.5445584%	0.134421985%
10. Difference predictors	20069	20.05082465%	0.139518661%
11. First Sum	4058	6.776737309%	0.616067028%
12. First Difference	2062	13.33656644%	1.212415131%
13. Mxm.f	5807711	2.472454294%	0.076243463%
14. Cff2td1.f	32761398	9.599434676%	0.669892048%
15. Cholesky Decomposition / substitution	2868556	3.227024329%	1.737006354%

Table C.5: Cache Miss at Cache Size = 32KB, Block Size = 32B, 4-Way Associative

Kernel	Total no of *LOADs*	No Prefetch	ERegs' Prefetch
1. Hydro fragment	4075	12.88343558%	0.613496933%
2. ICCG excerpt	8040	9.614427861%	0.323383085%
3. Inner product	4060	12.90640394%	0.640394089%
4. Banded linear equations	2472	15.21035599%	1.375404531%
5. Tri-diagonal elimination , below diagonal	6054	8.671952428%	0.445986125%
6. General linear recurrence equations	2000056	22.87700944%	2.641576036%
7. Equation of state fragment	18069	4.300182633%	0.143892855%
8. ADI integration	70063	10.74889742%	0.057091475%
9. Integrate predictors	20086	17.5445584%	0.139400578%
10. Difference predictors	20069	20.05082465%	0.139518661%
11. First Sum	4058	6.752094628%	0.616067028%
12. First Difference	2062	13.33656644%	1.212415131%
13. Mxm.f	5807711	2.472454294%	0.076243463%
14. Cff2td1.f	32761398	9.584417002%	0.654828588%
15. Cholesky Decomposition / substitution	2868556	3.009005228%	1.518987254%

Table C.6: Cache Miss at Cache Size = 32KB, Block Size = 32B, 8-Way Associative