Master Thesis

A Value Reuse Technique with Static Information of Assembly Language

Supervisor Professor Shinji Tomita

Department of Communications and Computer Engineering
Graduate School of Informatics
Kyoto University

Manrath Charoensuk

February 9, 2001

A Value Reuse Technique with Static Information of Assembly Language

Manrath Charoensuk

Abstract

This paper presents an analysis of the possibility of performance gain by Value Reuse in function-level, and discusses a preliminary implementation. It is well known that many programs execute at lower performance due to the redundant and frequent execution of the same instructions in order to get the same result. Based on this observation, the Value Reuse in function level tries to find the function execution with same parameters as already executed and to omit an actual execution by obtaining the result reserved by the first execution of that function with those parameters.

In order to achieve this Value Reuse in function level, we adopt the technique to modify the assembly object code before execution. We develop the special program named VR-translator to modify the object codes to reserve the function identification, parameters and results, and to check reserved data whether already executed or not. For reserving the function execution result, we define the Reuse Buffer in software data structure. In order to simplify the handling and structure of Reuse Buffer, we assume some restrictions to target function, i.e. it should be a leaf function, it should not have pointer parameter, and numbers of parameters and global variables are limited.

We evaluate this Value Reuse scheme wholly implemented by software with SPECint95 programs as workloads. Statically, 1% to 38% (average of 10functions are met our restriction and dynamically, 0.2% to 3.3% function executions succeed to reuse the results with reasonable sized Reuse Buffer. We had analysised later that if our technique can cope with the pointer arguments the static results will be up to 60% to 80%, the argument pointer is the next goal to improve our technique. However, the execution time is increased 6% to 12%, because of software Reuse Buffer look up. We envision the hardware assist, for example, hardware table look up and so on, in order to utilize the Value Reuse opportunity in future works.

静的情報を用いた値再利用の適用と評価

マンラット チャロエンスック

内容梗概

本論文では，値再利用による高速化技術を関数レベルにおいて適用するための実現方法，および，その効果について分析を行った．近年，コンパイル時には検出できないものの，プログラムの走行時に同じ命令列を繰返し実行するという冗長性が存在する場合があり，さらに性能向上の余地があることが報告されている．関数レベルの値再利用とは，繰返し実行される関数の中には，呼び出される際の引数が全く同じである場合が少なくないことを利用した高速化技術である．

本研究の最終的な目標は，一般的なプロセッサに対して値再利用を支援するハードウェア機構を追加し，コンパイラとの協調により高速化を図ることである．本論文では，従来のコンパイラが生成するアセンブリ言語ソースプログラムに対して，値再利用（Value Reuse）を行うための命令列を追加するＶＲトランスレータを開発した．簡単化のために，対象とする関数は，リーフ関数であること，引数がポインタを含まないこと，引数および大域変数の数に上限を設けることとした．値再利用は，既存のプロセッサにより実行可能な命令の組合せにより実現している．過去の実行結果を格納する再利用バッファを参照し，関数の入力データが一致した場合には，再利用バッファに登録されている出力データを用いることにより，命令の逐次実行を省略する．

現実的なエントリ数の再利用バッファを構築し，SPECINT95 ベンチマークプログラムを用いて測定した結果，前述の条件を満たす関数の静的な数は，全体の１％から３８％（平均１０％），また，プログラムを走行させた場合に，再利用バッファにヒットし，実際に再利用可能な関数呼び出しの回数は，条件を満たす関数全体の０．２％から３．３％であった．前者についてさらに分析を行った結果，引数がポインタを含むような関数も再利用できると仮定した場合，条件を満たす関数は全体の６０から８０％に達し，ポインタを考慮した値再利用技術の確立が今後の重要課題であることが明らかになった．また，後者に関しては，投機的に再利用バッファの内容を用意し，大域変数のヒット率を向上させる技術が必要であることが明らかになった．

A Value Reuse Technique with Static Information of Assembly Language

Contents

Chapter 1 Introduction

Several recent studies [1][2] have shown that there is significant result redundancy in programs, *i.e.*,many instructions perform the same computation and, hence, produce the same result over and over again. These studies have found that for several benchmarks more than 75 percent of the dynamic instructions produce the same result as before.

Also, recently, many hardware techniques have been proposed to exploit this redundancy, Value Prediction (VP) [2][3] and Instruction Reuse(IR) [4] attempt to reduce the execution time of programs by alleviating the dataflow constraint. They use the value locality or the predictable patterns in programs to determine – speculatively (Value Prediction) or non-speculatively (Instruction Reuse) – the results of instructions without actually executing them. The advantage of doing so is that instructions do not have to wait for their source instructions to execute first; they can execute sooner using the results obtained by the above techniques, thus, relaxing the dataflow constraint. A study of the differences between Value Prediction and Instruction Reuse is presented in [5].

Data value reuse is a technique that exploits the fact that many instructions or dynamic sequences of instructions(traces) are repeatly executed, and most of these repetitions have the same inputs, and thus generate the same results. Data value reuse exploits this fact by buffering previous inputs and their corresponding outputs. When an instruction/trace is encountered again and its current inputs are found in that buffer, its execution can be avoided by getting the outputs from the buffer. This reduces the functional units utilization and, more importantly, reduces the time to compute the results, and thus, shortens the lengths of critical paths of the execution. Techniques that try to reuse single instructions will be referred to as *instruction-level reuse*.

In this work, we try to reuse multiple instructions in the term of function calls unit. As we have observed many functions, having the same inputs(consequently producing the same output) when executed. This observation can be exploited to reduce the number of functions executed as follows: by buffering the previous result of the function, future instances of the same function can use the result

by establishing that the input operands in both cases are the same. we call this technique as *function-level reuse*. Data reuse can be exploited through software or hardware mechanisms, although the hardware mechanisms are better when considering the time used in looking up the reuse buffer, the main disadvantage of this mechanisms is that it requires special and complex hardware design. From the above reason, We first consider the software mechanisms, so that it can be controllable and applicable, then we use a simple hardware assist to speed up time used in looking up the reuse buffer.

In this paper, we first show the overview of Value Reuse technique and illustrate the scenarios of value reuse, why it should be better to obtain the results from the reuse buffer rather than perform the function calls again, our first approach is to reuse a value returned from a function then we find that we can extend reusing the global variables value in the function, then we describe the value reuse in function-level method and show the structure of the reuse buffer we propose. In the single instructions reuse technique, the entry used for looking up in the results buffer is a single value such as operand value, operand address or instruction address, thus the results buffer structure is not complicated. But in our approach we design the reuse buffer to keep multiple entries, the set of function address, parameters value and variables addresses. By considering that the reuse buffer should not be too large. In doing so, we limit the size of one entry, the function which has the parameter number or the global variables number more than the limited number will not be considered to reuse by our technique, and assuming that our technique cannot be used if the parameter of the function is the address or the pointer.

We then explain the methodology of our work, as described above, the function which has a pointer parameter, the number of parameters or global variables exceeds the limited number will not be considered by our technique, we show how to justify that a function is satisfied by our conditions or not. In doing so, we build the a special-purpose program to consider every function which was compiled into assembly language, the function which is satisfied with our conditions will be modified into translated source, the other functions remain in the same source. Accordingly, we get the new assembly source which was

modified by our program we called this program as *VR-translator*.

We perform several experiments to evaluate the concept of our technique. The results show that the amount of redundancy captured is less than we expected. As the results, each time the functions are called, some functions produce the different results and those results never be the same, thus we cannot use the previous outcome to bypass the executions. In addition to the above problem, we discuss some problems which make a great impact to the results. We propose a solution to address those problems in future works.

This paper is organized as follows. In the next Chapter, we explain the background of this research work, the traditional techniques for speeding up the hardware performance, the previous works related to this paper and the motivation for this work. We describe the concept of Value Reuse in function-level and a brief overview of Value Reuse method in function-level and present the scenarios to illustrate how useful our technique is and also show the scenarios in which we cannot cope with in Chapter 3. In Chapter 4 we describe a methodology for this work. In Chapter 5, we show the results and discussion and also envision the future works, Chapter 6 is devoted to concluding remarks.

Chapter 2 Background

In this Chapter, we briefly summarize the traditional techniques for speeding up the hardware performance, we show the disadvantage of these techniques. Then we describe the related works on Data Value Reuse in Chapter 2.2 and finally, we present the motivation for this works in Chapter 2.3.

2.1 Traditional Techniques for speeding up

There are many techniques proposed for speeding up the hardware performance in instruction-level.

2.1.1 Super Scalar

Super scalar is the technique that allows executing many machine instructions at the same time or in parallel. Because the physical limits to the speed single operations can be performed and many programs contain operations that can be executed in parallel, so it should be better to be executed in parallel.

The advantages of this technique are: it does not break the traditional programming model, the compiler is relatively simple, it is compatible in object instruction-level and it can use dynamic information to break control dependence but the disadvantages are: loss of data dependence knowledge during compilation and the hardware is sophisticated.

2.1.2 VLIW

VLIW(Very Long Instruction Word) is the technique that assigns many operations to be performed at the same time. The instruction size is very long (64 bits–1024 bits). The compiler investigates the executable operations which can be performed at the same time then put in the VLIW instruction. The loss of data dependence is managed when compiled

The hardware is simple although it still needs sophisticated branch prediction to achieve sufficient parallelism, but the disadvantage is there is no compatibility in the object instruction-level.

However, both Super scalar and VLIW techniques require instruction-level parallelism.

2.1.3 Value Prediction

A technique that does not requires hardware parallelism is the Value Prediction(VP). VP predicts the results of instructions(or, alternatively, the inputs of other instructions) based on the previously seen results, performs computation using the predicted values, and confirms the speculation at a later point.

The disadvantage of this technique is when a value is mispredicted, instructions dependent on that value are re-executed. Since mispredictions are detected during the verification stage, the execution of these instructions is delayed by the VP-verification latency.

On the other hand, Value Reuse does not incur any misprediction penalty. We will describe the overview of VR in the next Chapter.

2.2 Related Works on Data Value Reuse

A number of techniques of not having to redo computation have been used before in several contexts.

Data value reuse can be implemented by software or hardware. Software implementation is usually known as *memoization* or *tabulation* [6]. Memoization is a code transformation technique that takes advantage of the redundant nature of computation by trading execution time for increased memory storage. The results of frequently executed sections of code(e.g. function calls, groups of statements with limited side effects) are stored in a table. Later invocations of these sections of code are preceded by a table lookup, and in case of hit, the execution of these sections of code is avoid. Memoization has been used for functional and logic programs. The outcome of a function(or a rule) is saved in a table. If the function is encountered again with the same parameters then the result from the table is used instead of re-evaluation. Memoization is also used to reduce the running time of optimizing compilers, where the same data dependence test is carried out repeatly.

A hardware implementation of data value reuse was proposed by Harbison for the *Tree Machine* [7]. The Tree Machine has a a stack-oriented ISA and the main novelty of its architecture was that the hardware assumed a number of compiler's traditional optimizations, like common subexpression elimination

and invariant removal. This is achieved by means of a hardware mechanism, *value cache*, which stores the results of dynamic sequences of code called *phrases*. For each phrase, the value cache keeps its result as well as an identifier of its input variables. For the sake of simplicity, input variables are represented by a bit vector called *a dependence set*. A dependence set is associated with each result in the value cache to indicate the variables used in computing the result; the bit positions are determined by the address of the variables. When an address is overwritten, all the results in the value cache which have the bit set for that address are invalidated. If a phrase is encountered again, recomputation is avoided by reading the result from value cache.

Another hardware implementation of data value reuse is the *result cache* proposed by Richardson [8]. The objective was to speed-up some long latency operations, like multiplications, divisions and square roots, by caching the results of recently executed operations. He introduces the notion of redundant computation, which is computation that produces the same result repeatedly because it gets the same value for its operands. In this work, the results of floating point operations are stored in the result cache. The result cache is indexed by hashing the source operand values, and for each pair of operands it contains the operation code and the corresponding result. The result cache is assessed in parallel with executing an floating point operation. If the result is found in the result cache then the operation is halted.

Result caching is further investigated by Oberman and Flynn [9], They propose the use of *division caches* and *reciprocal caches* for capturing the redundancy in the division and square root computation. The division caches are similar to Richardson's result cache, but for divisions only. The reciprocal caches hold the repiprocals of the divisors. They help convert the high latency division operation to relatively low latency multiply operation. These caches are accessed using the bits from the mantissa of the operands.

Sodani and Sohi propose the *reuse buffer* [4] which is a hardware implementation of data value reuse(or dynamic instruction instruction reuse, as it is called in that paper). The reuse buffer is indexed by the instruction address. They propose three different reuse schemes. In the first scheme, for each instruction in the reuse buffer, it holds the source operand values and the result of the last execution of this instruction. In the second scheme, instead of the source operand values, the buffer holds the source operand names (architectural register identifiers). In the third scheme, in addition to the information of the second scheme, the buffer stores the identifiers of the producer instructions of the source operands. In this scheme, dependent instructions that are fetched simultaneously can be reused by chaining their individual reuses. However, the reuse of each individual instruction is still a sequential process since it must wait until the reuse of all previous instructions has been checked.

Jourdan *el al.* propose a renaming scheme that exploits the phenomenon of instruction-level reuse in order to reduce the register pressure. The basic idea is that several dynamic instructions that produce the same result share the same physical register [10].

Another application of data value reuse has been presented in [11] . Weinberg and Nagel describe a technique that reuses high-level language pointer-expressions with the aid of compiler inserted hints. Basically, once the input operand set of an expression matches a previously executed instance of the same expression, the result is obtained from a table instead of recomputing it.

Monila, González and Tubella presented a reused scheme referred to as *Redundant Computation Buffer* [12]. The underlying concept is the removal of redundant computations, and in particular, the run-time elimination of quasi-common subexpressions and quasi-invariants.

Finally, Huang and Lilja have recently proposed a scheme to reuse basic blocks [13][14]. Basic block reuse is a particular case of trace-level reuse in which traces are limited to basic blocks. Trace-level reuse is more general and can exploit reuse in larger sequences of instructions, such as subroutines, loops, etc.

2.3 Motivation for this work

There are several differences between our work and the work mentioned above. First, most of the above techniques are more special purpose.

The value cache [7] approach is focused on an architecture which expresses

computation in the form of *parse trees*(Tree Machine). The results cache [8] and the division and reciprocal caches [9] target only floating point operations. Our approach is general propose in that it does not assume any special architecture.

Second, most of the above techniques target on instruction unit. Dynamic instruction reuse[4] captures reuse of any type of instruction(except stores). Our approach targets on function level.

Table 2.1: Comparision between Instruction Reuse and Value Reuse

Instruction Reuse	Value Reuse in function level
Instruction unit	Function unit
The probability of hit is high	The probability of hit is low
If reused, the number of instructions are not decreased	If reused, the number of instructions are highly decreased

Table 2.1 shows how different between value reuse of instruction unit and function level, In the techniques, which target on instruction unit, access their respective result buffers by a simple value. The Tree Machine [7] accesses the value cache by using operand address. The result cache [8], the division and reciprocal caches in [9] were accessed by using operand values. The dynamic instruction reuse [4] access the reuse buffer by using the instruction address. Our approach accesses the reuse buffer by using a larger entry(function address, parameters value and variable address), therefore the probability of hit in the case of instruction unit is higher than our approach. But when a function hits the reuse buffer, it decreases the number of instructions engaged, this advantage is more beneficial than the case of instructions unit.

The most significant benefit of our approach is that we perform static value reuse. In value cache [7], result cache [8], division caches and reciprocal caches [9], dynamic instruction reuse [4], they use special and complex hardware for their result buffers. But in our approach we use a simple and general hardware as we are proposing CAM for the reuse buffer in order to speed up the performance, and we also use a software technique to control the buffer. This

advantage makes our reuse buffer be controllable and applicable.

Chapter 3 Function Level Data Value Reuse

In this chapter, we summarize the overview of Value Reuse in function-level, then we illustrate the scenarios for value reuse in Chapter 3.2. We show the structure of reuse buffer and the concept of how to manage the reuse buffer in Chapter 3.3.

3.1 Overview of Value Reuse

Value Reuse(VR) exploits redundancy in programs by obtaining results of functions from their previous executions, and thereby, not executing through the functions again.

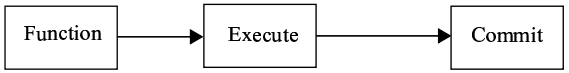

Figure 3.1: Without Value Reuse

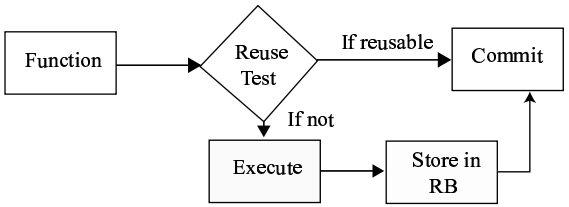

Figure 3.2: With Value Reuse

Figure 3.1 shows how the executions go without VR technique. When a function is called, it normally executes through the end of function and returns to where it was called. But with VR technique as shown in Figure 3.2, it will look up the stored results in the Reuse buffer by establishing the reuse test before executing. The reuse test validates results by establishing that the entry(set of the function address, arguments values and global variables address) is the same as those used to call this function. If it tests that this function was previously called then the stored results are read from the Reuse Buffer. Since the correct

results are known, this function is not performed, and instead it is queued for retirement. But if this function is first called or the entry is not the same as stored in reuse buffer, this function will be performed and before returning to the caller, its results will be stored in Reuse Buffer for future use.

Since VR validates results early based on inputs, it may be conservative. For example, if the inputs of a function are not ready at the time it is tested for reuse then it will not get reused; or a function that produces the same result but with different inputs will not get reused.

3.2 Scenarios for Value Reuse

In this chapter, we describe what functions which are suitable for the Value Reuse technique and also the function which are not considered by our technique.

Before developing technique to allow Value Reuse, we need to understand why this phenomenon occurs, What causes functions to be performed with the same input? Why might it be better to obtain the outcome of a function from a buffer rather than recompute it? In order to answer these questions, we look at the following scenarios.

The first scenario involves calling function repeatedly in a C program. As illustrated in Figure 3.3, when the function *func* is called. In addition to performing the function call from block (b), the processor may normally execute instructions from the block (c) which is similar to the assembly source of block (b). When the function *func* is called again, the instructions from the block (c) will be re-executed again. In order to get the same output, performing the same instructions as before causes redundancy in program.

In this paper, we propose a new way of performing such the instructions as shown in block (d). In block (d), before starting execute any instructions, the processor will search in the buffer as illustrated by the function `search_bf`, function `search_bf` will test the function identity and its arguments value in the stored results, and if there are previous results stored in the buffer then it will jump to get the result from the buffer without recomputing the instructions again, but if there are no such results, it will perform the executions and store

```
main( ){
  ....
  func(a, list, size);
  ....
  func(a, list, size);
  ....
}
```

(a)

```
int func(x, list, size){
  int i;
  for(i=0;i<size;i++){
    if(x==list[i]) return i;
  }
  return -1;
}
```

(b)

```
*1 i = 0
*2 if (i>= size) jump out
*3 p = list+i
*4 val = Memory[p]
*5 if (a == val) jump found
*6 i++
*7 jump 2
*8 if (i>= size) jump out
  ....
```

(c)

```
*1 if(search_bf == TRUE) jump 92
*2 i = 0
*3 if( i>= size) jump out
  ....
  ....
*90 store results in buffer
*91 return
*92 read results from buffer
*93 return
```

(d)

Figure 3.3: Scenario for Value Reuse of called functions

the results in the buffer before returning to where it was called.

All functions in C are functions with the option that they do not have to return a value, functions that return a value are *called functions*. Our initial goal for developing the Value Reuse in function-level was to obtain a value from called functions as described above, However, when we were studying the effectiveness of the technique that we develop for the above case, we discovered that the concept was much more powerful. The other scenario where the value can be reused also arises in functions with do not return a value.

In functions which do not return a value. Because no value will be returned from this function, what value can be reuse? There are two different variables in C programs, static variables and global variables. The global variables whose value still remain after a functions has been returned unlike the static variables which first be generated when a function is called and then disappeared after a

function has been returned. Our Value Reuse technique can be developed for the global variables, thus, we consider the value reuse of the global variables in functions

The above situations are best illustrated by an example. Consider the example of Figure 3.4, the function *func* again but this time *func* is a function which does not return a value.

```
main( ){
  ....
  func( );
  ....
  func( );
  ....
}
```

(a)

```
void func( ){
  FLAG = TRUE;
  max = maxval(a, b, c);
  ....
  count = add(a, b);
  ....
}
```

(b)

```
void check_gb( ){
  if( this_gb( )=TRUE &&
      next_gb == FALSE)
    return TRUE;
  else return FALSE;
}
```

(e)

```
*1 FLAG = 1
*2 call maxval
*3 max = maxval
*4 ....
*5 call add
*6 count = add
*7 ....
*8 ....
  ....
```

(c)

```
*1  if(search_bf == FALSE) jump 4
*2  if(check_gb( ) ==FALSE) jump 4;   # FLAG
*3  if(check_gb( ) ==FALSE) jump 4;   # max
*4   execute instructions
     ....
*50 if(check_gb( )==TRUE) jump 92;   # count
*51 execute instructions
*90 store results in buffer
*91 return
*92 read results from buffer
*93 return
```

(d)

Figure 3.4: Scenario for Value Reuse of the global variables

In Figure 3.4(b) function *func* consists of three global variables, FLAG, max and count.

In the function which has the global variables, after checking that all arguments are the same as stored in the buffer, it need check whether there are other global variables in this function or not. If there are global variables exist, it need check every global variable in the program before knowing that this function and its entry was previously performed.

In Figure 3.4(d), if the test of the arguments fails, then there is no need to perform the next global variables test, it will jump to perform normal executions and store the results before returning to the caller. But in case that the arguments test is true, it will test the global variables and even if this global variable test is true, there is a test whether the next global variable exists or not inside the global variable test, these tests repeat until there is no global variable in the reuse buffer for this function. Note that if one of these test fails, there is no need to perform the next test again, it will jump to perform the normal executions.

We consider the initial value of global variable as input and the value after a function is performed as output, we consider this output to be reused by our technique. At every time this function is called, if it produces the same results of these global variables, it should be better to receive the results from the Reuse Buffer rather than performing this function calls again. So we consider the global variables as one of the input entry besides the arguments entry, If we test that every global variable has the same initial value as stored in the Reuse Buffer, then we pass the stored result to them and jump out to the end of function without performing the remains of instructions.

3.2.1 The Nested Functions

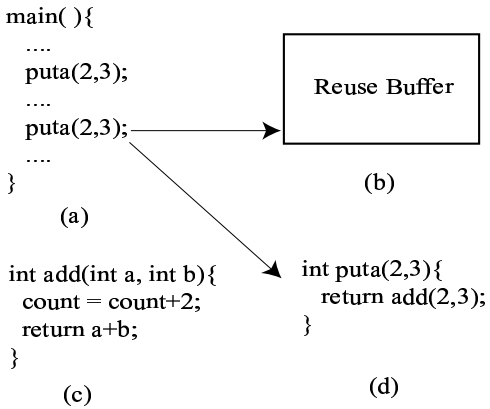

Figure 3.5: The nested functions

Consider the scenario illustrated in Figure 3.5. The function `add` has a global variable `count`, the function `puta` does not have a global variable. When the function `puta` in the main function is first called, it looks in the reuse buffer and finds no previous results, so it performs the executions and stores the results in the reuse buffer before returning to the main program. But when the `puta` is called for the second times, this time `puta` looks in the stored results and finds the result, so it uses the previous results without performing the executions. In this case, the program does not perform the execution of the global variable `count`. Such this result causes error in execution time.

To reuse such this case, it needs to store the results of function `add` in the entry of the function `puta`, thus, the results of the lower function will be stored in the entry of the upper function. In this works, we do not consider such these functions

3.2.2 The Functions which have pointer parameters

In general, the pointer refers to two value, its value and the value which it points to. In C, we clearly know what value this pointer is referring to. But in assembly source, it is difficult to determine especially when the pointer is a parameter. Because in parameter, unlike global variable, the compiler does not provide the fix memory location to keep the value of the parameter, it value is kept in a register or a frame pointer, instead. When there exists the pointer parameters in the functions, in order to prevent execution error, it needs to check what value the registers keep before performing the execution. By these problems, we do not consider the pointer parameters in this works.

3.3 Structure of Reuse Buffer

In general, there exists the functions which do not have to return the value. The functions which produce the output are *called function*. And as we clearly know that in called function, how many parameters a function has, it produces only one output as illustrated in Figure 3.6 But in the case of the global variables, if we consider a global variable as an input of the function or *global variable input*, after calling a function, the value of global variable is changed, we consider the final value of the global variable as *global variable output* and the number of

output in the case of the global variables is not only one, it depends on the number of *global variable input*. Figure 3.7 illustrates the function which does not return a value, this function produces only the output of the global variables .

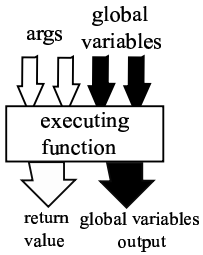

Figure 3.6: Return a Value

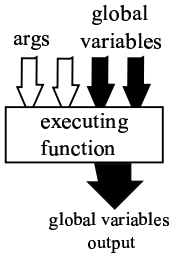

Figure 3.7: Do not return a value

To introduce reuse technique, it needs to store the entries in the reuse buffer, in the single instructions reuse technique, such as the instruction reuse(IR). The entry consists of the information of instruction and some information of operands as shown in Figure 3.8. The *address* stores memory address(the outcome of calculation). The *result, operand value1 and operand value2* store the result and the operand values of the instruction. These fields are used to identify the instruction that can be reused. The *memvalid* bit [4] indicates whether the value loaded from memory(present in the result field) is valid. Because the architecture assumes three operand instructions, the entry size is not large and complicated. Note that the third operand is equal to the result field.

address	operand1 value	operand2 value	result	mem valid

Figure 3.8: The entry of Instruction Reuse

In our approach we attempt to reuse value in function-level, not only a value returned from the function but also the global variables value in the function.

The entry of the reuse buffer must consist of the information of function, every parameter and every global variable. In general, the functions have no limit of the parameter number and no limit of the global variable number. In this work, we build the reuse buffer with an array structure and in array, the size must be limited.

Process ID	function address	P1 value	....	Pn value	variable1			.....	variable_n			result
					address	init	last		address	init	last	

Figure 3.9: The entry of the Function-level Reuse Technique

We then consider the size of one entry, one entry must have the information of process ID, the information of function, the information of parameters and the information of global variables as shown in Figure 3.9. To limit the size of the reuse buffer made by an array structure, we must limit the number of parameters and the number of global variables. Any functions which have the parameters or the global variables more than the limited number will not be stored in the reuse buffer, thus they will not be considered by our technique. By this limit, some functions which have the large number of parameters or global variables cannot be covered.

The structure of the reuse buffer consists of the group of entries as shown in Table 3.1. The process ID are the same so it can be ignored. The first field of the entry is the information of the function, the best way to refer to the function identity is its address. The next field is the information of parameters, we refer to it by its value, then followed by the output or a value return from the function. The remains of the fields are the information of the global variables. We refer to the identity of the variables by its address, the input of the variables by its initial value or the value before the function is called and finally the output of the variables by the final value or the value after the function has been returned.

Table 3.1: Reuse Buffer

func	p_1	$\cdots$	p_n	out	$variable_1$			$\cdots$	$variable_n$		
					addr	in	out		addr	in	out
$entry_1$											
$entry_2$											
$entry_3$											
$\vdots$											
$\vdots$											
$entry_n$											

3.4 Schemes for Value Reuse

In this chapter, We describe software schemes to implement Value Reuse in function-level. To reuse a function we need to determine that its outcome is going to be the same as a previous outcome, and reuse the previous outcome. The reuse scheme described in this chapter implement this determination in different ways. We store the results of a previouly-executed instruction in a Reuse Buffer. When a function is encountered, the RB is queried to see if it contains a reusable result for the function. Three issues need to be dealt with:

1. how the information in the RB is accessed
2. how we know that the accessed RB entry has reusable information
3. how the buffer is managed

The first issue is dealt with: we provide an index for searching the RB. The RB could be organized with any degree of complexity, the larger the number of entries, the more the time needed for searching the index of RB.

To deal with the second issue, we need to develop a reuse test which checks information(function, its arguments and its global variables) accessed from the RB to see if there is a reusable result.

There are two aspects to RB management: deciding which entries get placed in the buffer and maintaining the consistency of the buffer. The decision as to what to place in the buffer can range from no policy, *i.e.,* place all recently entries in this buffer, to a more judicious policy that filters out entries that are

not likely to be reused. Maintaining the consistency of information in the RB, we discuss the following issues:

1. What information is stored in the RB?

   RB entry: As we described before, An entry in RB for the value reuse in function-level technique is shown in Figure 3.9
2. How is the reuse test performed?

   Reuse Test: For testing reuse, the address of function, the value of the parameters are performed first at the beginning of the functions, then it established the reuse test of the global variables when it finds the global variables.

we will show more detail of our method in the next chapter.

Chapter 4 Static Translation of Assembly Source Codes

In this chapter we first describe the strategies of our works. And since we do reuse in the Sparc environment, we simply show the configuration of assembly source code on Sparc architecture then we show our designed structure of reuse table.

4.1 Strategies

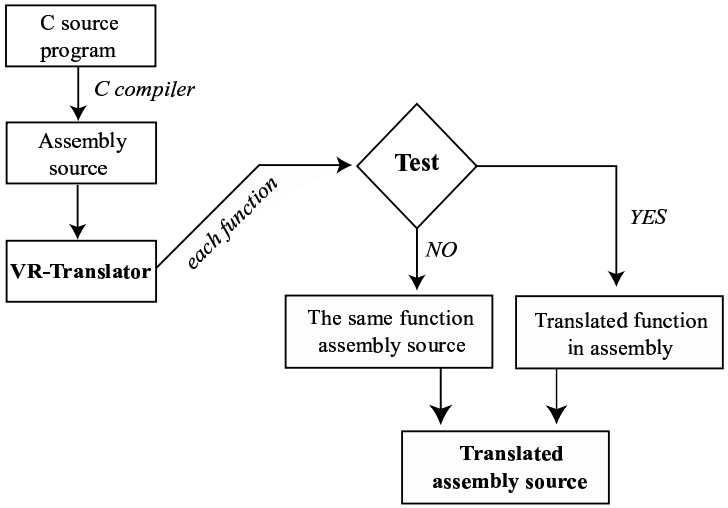

Figure 4.1: Methodology

Our work is to modify assembly source code to the new assembly source. by considering each function in the assembly source. In this work, we first assume the limit the number of parameters in the reuse table to four and the number of global variables to ten, we will describe the validity later. And as described before by some problems we ignore the functions which have pointer parameters. In doing so, we build the program by C to consider each function

in assembly source code we called this program *the VR-translator*

Figure 4.1 describes our work briefly. Firstly, the C source program or the benchmarks program is compiled to assembly source by gcc compiler with no optimization option on Sparc architecture. Althrough the non-optimized code empties the delay slot while executing and the performance is worse than the optimized code, we decided to do with this code because it is easy to judge by our translator, and our first goal is to compare the number of time between the VR method and Non-VR method. we will do it next with `-O2` optimization option to upgrade the performance. After compiled by the compiler then the assembly source code will be performed by our translator, the translator modifies the assembly source in each function unit, each function is tested with the following conditions.

1. Every argument is not an address.
2. The maximum global variable number is not larger than ten.
3. The maximum argument number is not larger than four.
4. The return value is not larger than one word.
5. The functions which do not return a value must have at least one global variable.
6. The functions are leaf functions.

The first condition is to stop doing with the pointer parameters, if the translator finds that at least one of the parameters is pointer, the test is stopped. The second, the third and the fourth condition come from the limited entry size of the reuse buffer we restricted. The fifth condition, we cannot reuse any values with those functions. The last condition is because we can cope only with the leaf functions. If any functions are satisfied with all the conditions above, they will be translated. If not, the translator will do nothing with those functions, finally, we will get the combination of modified and non-modified functions in assembly source code or *translated assembly source*.

In C, it is easy to know what a function is satisfied with the six conditions above or not, but in the assembly source code, there are different ways to know that. We will show how to justify whether a function in assembly source code is satisfied with the six conditions above or not in Chapter 4.3.

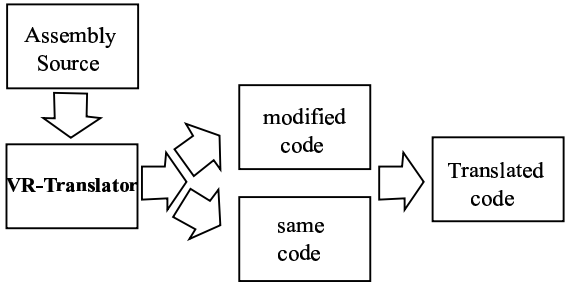

Figure 4.2: The action the translator takes with a function

The translator cannot know that a function is satisfied with all conditions and it should translate this function at the beginning of assembly source code since many conditions can be known later such as the number of the global variables. It has to count the number of the global variables until the end of function and in assembly source, the type of a returned value declares at the end of function. By this problem, Figure 4.2 illustrates the action the translator takes with a function, it generates both the modified source and also keep the same source in the temporary buffer, when it knows that this function can be translated, it writes the modified source code in the output file, otherwise it writes the same source code of this function.

4.2 Assembly source code on Sparc Architecture

The source programs are compiled to assembly source on Sparc architecure, In this chapter, we briefly show the structure of the assembly source on Sparc architecture in order to understand the concept of our works.

The assembly source code on Sparc Architecture [15][16] is divided in three different parts as shown in Figure 4.3. The first part, *prologue*, declares registers saving, the most executions are performed in the *body* part, the return and restore executions are in the *epilogue* part.

The parameters value are found in the beginning of the body part. The global variables are found everywhere in the body part like they can appear everywhere in the C source program. In C, we know what size of the value (one

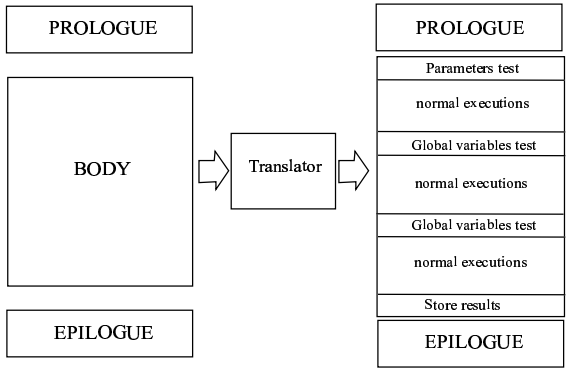

Figure 4.3: Assembly source code before and after modified by the translator

word, two words) a function returns at the beginning of the source program but in assembly source it is known at the end of body part.

The translator will insert the set of instructions in order to get the values from parameters, global variables and the return value, thus it will insert the set of instructions useful to get the parameters test at the beginning of the body part, the set of instructions after it finds the global variables and the set of instructions in order to store and read the results from the reuse buffer at the end of body part. Figure 4.3 shows the assembly source after modified by the translator, the modified assembly source are likely to be larger than the assembly source because some instructions have been inserted, however, it does not mean that the number of executions occurs in the modified assembly source always more than the previous source because when the reuse test is found to be true, it will jump to use the results from the reuse buffer and return to the caller.

When the modified assembly source is executed, the function establishes reuse test of the parameters at the beginning of the body part. The reuse test of the global variables occurs in the whole body part whenever the new global variable is found(the previous found variable will not be tested). Storing the results is at the end of the body part before returning to where it was called.

4.3 Assembly language in Sparc Architecture

In this Chapter, we show how the translator knows the information of the function, the parameters, the global variables including the value return from the functions. We also show what set of instructions should be inserted when the translator finds those values in order to keep the values.

To get the address of the function, the parameters value, the global variables address and the value return from the functions, the translator provides a global variable named *global_entry* which is an array variable.

The functions in C, when it is compiled to assembly source, it is generally called subroutine. In the subroutine, after a `save` instruction [15, pp. 184–188]then, we insert this instruction.

```
call    .+8
```

The address of this function is saved in `%o7`. The Sparc architecture provides six registers, `%i0 - %i5` for saving the first six arguments. The functions which have parameter(s), when they are compiled, the beginning of their assembly source code is shown as follows:.

```
st      %i0, [%fp+68]  !first argument
st      %i1, [%fp+72]  !second argument
st      %i2, [%fp+76]  !third argument
st      %i3, [%fp+80]  !fourth argument
st      %i4, [%fp+84]  !exceed the limit number
```

At the beginning of the subroutine source, when the translator finds register `%i0 - %i5` with the pattern like above, it recognizes how many parameters this function has. Then in order to get the information or the value of the parameter, the translator add some instructions after those instructions. When the register `%i4` is found(the fifth parameter), it means this function has the parameter number more than we limited then the translator recognizes that this function cannot be considered by our technique.

After finishing the parameters search, the translator inserts the set of instruction in order to get the results from the function and its parameters as follows:

```
save    %sp, -120, %sp
```



```
call    .+8             ! to get the address of function
st      %i0, [%fp+68]
st      %i1, [%fp+72]
sethi   %hi(global_entry), %o1
or      %o1, %lo(global_entry), %o1
st      %o7, [%o0]      ! address of function -> global_entry[0]
st      %i0, [%o0+4]    ! value of 1st arg    -> global_entry[1]
st      %o1, [%o0+8]    ! value of 2st arg    -> global_entry[2]
call    search_bf, 0    ! establish the parameter tests
```

By the above, `search_bf` uses the value in `global_entry` to compare with the results in the reuse buffer. Note that `search_bf` is written by C and it is linked when compiled.

The Sparc refers to the global variable by `sethi` instructions.

```
sethi   %hi(max), %o1
or      %o1, %lo(max), %o1
```

When the translator finds the pattern like above, it recognizes that there is a global variable then it inserts some instructions after those instructions in order to get the value and establish the reuse test.

The Sparc refers to a return value by the following instruction:

```
mov     %o0, %i0
```

In this case, the translator knows that the return value is not larger than one word so it can restore this result in the buffer.

In assembly source, the compiler does not provide the fix memory location for the parameters or the static variables unlike the global variables. It keeps the value of the parameters in the frame pointers. In order to perform the executions, the processor loads the value from the frame pointers to the registers. When the processor loads the value from the registers again, the translator knows that this parameter value is a pointer value.

```
ld      [%fp+68], %o0
ld      [%o0], %o1      ! pointer value
ldub    [%o0], %o1      ! pointer to string
```

The translator will flag on the the registers when the load instructions per-

forms with the frame pointer or registers `i0-i6`. If the load instruction interacts with these registers again, then it can be known that the value in the register is pointer, then it stop translating this function.

```
ld      [%fp+68], %o0   !  keep %o0 (pointer?)
...
ld      [%o0], %o1      !
...
ldub    [%o0], %o1      ! pointer to string
```

4.4 Management of Reuse Buffer

We provide one word for each element of the entry, Table 4.1 shows the structure of Reuse Table, the * sign means there is a data in this block, and the - sign means there is no data in this block, In this table , f_1 has nine global variables, f_2 and f_3 have 8 global variables. In the case of f_1 it wastes three words of storing data block and six words in f_2 and f_3, thus it wastes three words for each global variables which is not full in the Reuse Table. This is a dissipation.

Table 4.1: Management of Reuse Buffer

F	p	$variable_1$			$variable_2$			...	$variable_9$			$variable_{10}$		
		add	in	out	add	in	out		add	in	out	add	in	out
f_1	*	*	*	*	*	*	*	*	*	*	*	-	-	-
f_2	*	*	*	*	*	*	*	*	-	-	-	-	-	-
f_3	*	*	*	*	*	*	*	*	-	-	-	-	-	-
⋮														
⋮														
f_n														

We then consider the number of entries in the reuse buffer. In this works, we provide 1024 entries for the overall functions, and divide 1024 entries into blocks of functions, each block has maximum 16 entries, and we store each function in one block, thus the reuse buffer can store maximum 64 functions, but when the

new function encountered, it can be stored in the reuse buffer by replacing the old used function block.

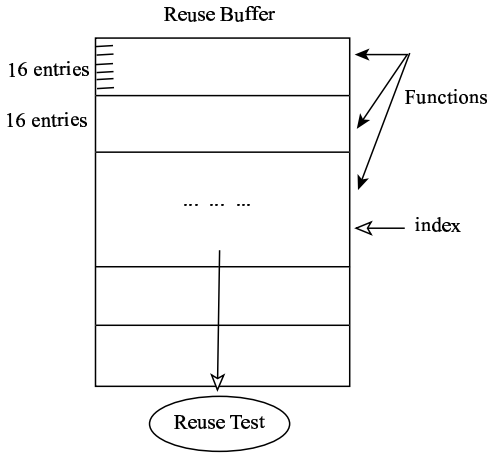

Figure 4.4: Entry of reuse buffer

Figures 4.4 shows the entries of reuse buffer, we provide a software index for searching an entry. The decision as to place the different functions in the buffer is first in first out(FIFO). First of all, when a function is requested to be stored in the buffer, it is stored in the first block. Then the next function is encountered and if its address field is the same as the previous function in the buffer, it will be stored in the same block of that function. But if its address is not found in the buffer before or this function was first called, it will be stored in the new block of the reuse buffer. When all blocks of reuse buffer are occupied, the new function encountered will replace the first-in block of the buffer.

The decision as to place the same function with different fields of parameters or different fields of global variables in the buffer is to sort from the small number to the large number. A function is stored in the first entry of a block when the next function or next entry is requested to be stored in the same block, the decision comes as to compare the value of the first parameter of the

two functions, if the value of the second entry is smaller than the first entry, the second entry will replace the first entry and the first entry will shift to the next entry. By doing this algorithm, it is useful when establishing the reuse test in the buffer, thus it stops searching an entry when the next entry value is less than the current entry the index pointed to. When all the entries are occupied, the new entry will insert into the reuse buffer and shift the largest entry out of the buffer.

In this works, we do not explore the policy that filters out the entries that are not likely to be reused. By these decisions, the time used in searching the function blocks is referred to $O(n)$ but in the entries of the same function is referred to $O(\frac{n}{2})$.

Chapter 5 Results and Discussions

In this chapter, we first describe the characteristics of the benchmarks programs used in the experiments, we perform several experiments to evaluate the concept of Value Reuse in function-level. How much static number of functions can be registered in the reuse buffer. How much dynamic number of hit functions. The amount of redundancy captured by our approach and finally we compare the performance between our technique and non-VR technique.

5.1 Benchmarks

We used the four programs from the SPECINT95 benchmarks suite for our studies. Since the integer can be reused better than floating point, so we think INT95 can evaluate the concept of our studies better than FP95. The benchmarks program are *compress, go, perl, li*, they were compiled using GNU gcc (version 2.95.2) on the Ultra 30 machine.

The characteristics of the four benchmarks are described as follows:

compress The small program, many functions are not big, many functions have a few number of parameters and global variables.

go The big program, most functions have the array variables, most functions are big and many functions have many number of the global variables and many number of arguments.

perl The middle program, most functions consist of two number of arguments but most arguments are pointers to structure variables.

li Like perl, many function consist of pointer argument and most functions have small number of arguments.

5.2 Static Number of Functions Registered

Because our technique cannot cover all the functions in C, we first calculate the percentage of the functions, which can be covered by our technique, in source program. Note that these results are not the functions which are executed dynamically. As we described before, the translator establishes the reuse test by the following conditions:

1. The parameter is not an address.
2. The parameters number does not exceed four.
3. The global variables number does not exceed ten.
4. A value return from function is not larger than one word.
5. The function which do not return a return value must have at least one global variable.
6. The function are leaf functions.

Figure 5.1 shows the static percentage of the functions which can be captured by our translator, we evaluate these results from

$$Static\ percentage = \frac{Number\ of\ registered\ functions \times 100}{Number\ of\ all\ functions\ in\ the\ benchmark\ program}$$

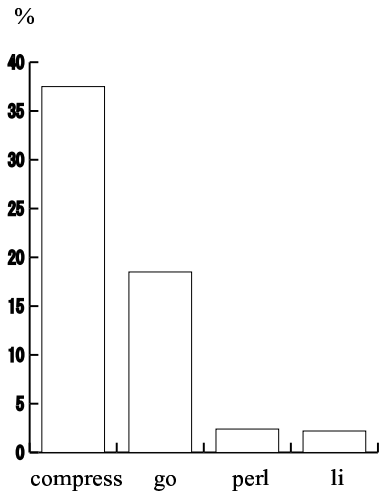

Figure 5.1: Static percentage of functions which can be translated

As the results, they are not high in *compress* and *go* and they are extremely low in *perl* and *li*(less than 5 percent). In order to do reuse an address parameter correctly, we need to perform a special check, which we do not do it in this work, thus the translator stop doing any functions if it finds that they contain

an address parameter. Unfortunately, most functions in *perl* and *li* have at least one address parameter.

Since the reuse buffer must be a limited size and its entry should not be too large. The limited number of global variables is set to ten and parameter number is set to four. Many functions in *go* have the number of global variables more than we limited, this makes the percentage of these results low.

Moreover, we set each field in the reuse buffer to one word, A value which exceeds this limit cannot be stored in the reuse buffer, the functions which return a *double* value are also ignored by the translator. The last factor which makes these results low are that there are many functions which have the functions inside or nested functions, because we still cannot cope these functions.

5.3 Dynamic Number of Hit Functions

Figure 5.2 shows the dynamic percentage of the functions which can be reused, these results are different from the Figure 5.1 that we consider only the functions which can be translated. The results of these functions are stored in the reuse buffer, the entries stored in the reuse buffer can be replaced by the new entry. We evaluate the overall number of the entries stored in the reuse buffer and the number of hit functions then we calculate the percentage of hit.

Thus we evaluate these results from

$$Dynamic\ percentage = \frac{Number\ of\ hit\ functions \times 100}{Number\ of\ overall\ functions\ stored\ in\ the\ reuse\ table}$$

As the results, although the percentage of static function captured by the translator shown in Figure 5.1 is obviously different among the different benchmarks, the dynamic percentage of functions which can be reused is not different. Surprisingly, the percentage of translated functions in *go* is more than *perl* and *li* but the percentage of function reused in *go* is less than the two benchmarks.

Why does this phenomenon occur? In order to understand this problem, we show an example below.

```
int random(int a )
{
```

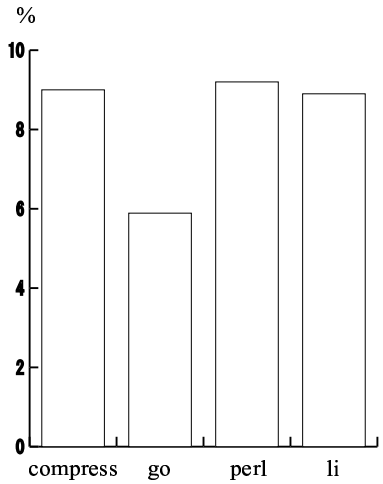

Figure 5.2: Dynamic percentage of hit functions

```
  int ran;
  ...
  line = line + 1;
  ran = 100*a /line
  ...
  return ran;
}
```

Consider the above function, line is a global variable and each time this function is called, line value is increased by 1. The results of this function is shown in Table 5.1. Note that we focus on only this variable value.

In order to satisfy the condition of the reuse test in the global variables, the address of the variable and the initial value or the value before calling the function must be the same. As the results show that every time this function is called, it will never produce the same result, or its initial value never be the same. thus the reuse test of the global variable of this function fails every time, and this problem cannot be addressed by our VR technique.

Therefore, storing many entries in the reuse buffer can not guarantee that

Table 5.1: A function which cannot be reused

func	...	global variables		
		addr	init	last
62030	...	23493	10	11
62630	...	23493	11	12
62630	...	23493	12	13
⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮
62630	...	23493	20	21

we will get the best efficiency, because some programs can not be reused and storing them in Reuse Buffer will make a great impact of wasted-space in buffer and time used in searching an entry.

5.4 Hit Percentage by the Number of Variables

By the previous results, we know that if there exists such an global variable in a function, this function cannot be hit. Then we calculate the hit percentage by the number of the global variables.

We evaluate these results from the dynamic number of hit function separated by the number of global variables

$$results\#n = \frac{Number\ of\ hit\ functions\ of\ function\ which\ have\ \#n\ variables}{Overall\ dynamic\ number\ of\ hit\ functions}$$

Figure 5.3 shows the results of the functions which have the global number of zero–one, two, three, and more than three. We do not separate the results of zero because there are a few functions which have no global variable in the benchmarks program. As the results, the functions which have a few global variables are often hit and when the number of the global variables increases the percentage of hit is increasingly low. This is because in the functions which have many global variables number, it is difficult that all the fields of the reuse entry in the reuse buffer will be the same(even if one of the fields is different, this entry cannot be reused) and also the probability that one of the global

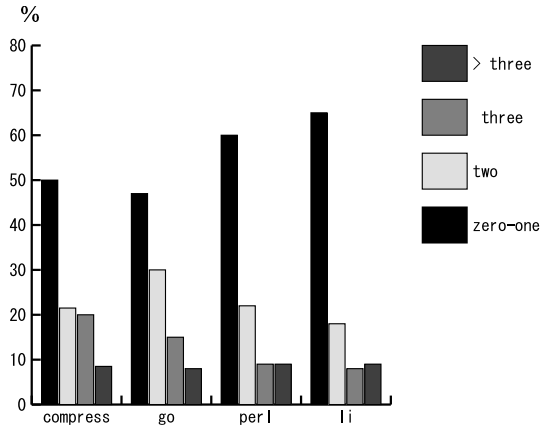

Figure 5.3: Dynamic percentage of hit functions

variables produces the different result, in each time the function is called, is high as we discussed in the Chapter 5.4.

5.5 Dynamic Number of Reusable Functions Calls

We then evaluate the amount of dynamic hit functions per the overall dynamic function calls of the benchmark program, since we do not evaluate the number of overall dynamic function calls. But by multiplying the results shown in Figure 5.1 with the results shown in Figure 5.2. We get the dynamic hit percentage of the static registered functions, this result gives the approximate value of dynamic hit functions per the dynamic overall function calls. However these results are the approximate value not the accurate value, and we call this as the amount of redundancy captured.

Thus we estimate these results from

$$results = Static\ \#\ of\ functions\ registered \times Dynamic\ \#\ of\ hit\ functions$$

Figure 5.4 shows the amount of redundancy captured by our technique. As the results show that the redundancy captured are not high as we expected. This is because both the number of functions registered and the number of hit

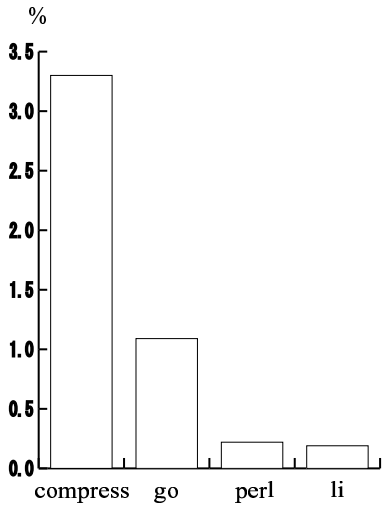

Figure 5.4: The amount of redundancy captured

functions are not high.

Note that there are significant different number of instructions in each function, some functions may have hundreds of instruction but some functions may perform thousands of instructions, these results are not the redundancy per overall instructions. Moreover there are also different in the percentage of hit, some functions are often hit but some functions are never hit. These results show the approximate amount of redundancy by assuming that every function is hit by average.

5.6 Speedups

We then perform the test of the execution time between non-VR and VR technique. In doing so, we compile the source of some benchmarks with no-optimization option then executing the assembly source compared with the translated assembly source obtained by our translator. We base the execution time of non-VR to 100 and evaluate the execution time of VR technique as follows:

$$Execution\ time\ of\ VR\ technique = \frac{T_{VR}}{T_{non-VR}} \times 100$$

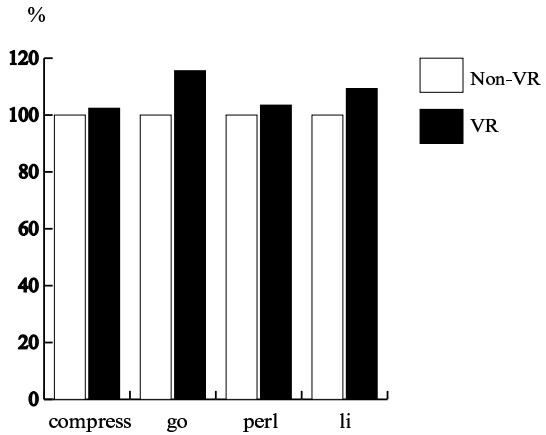

Figure 5.5: Execution time

Figure 5.5 shows the time compared with and without VR technique. The results are not different in *compress, perl and li* but it is obviously different in *go*. As the results, the execution time in our technique is more than non-VR technique.

Our goal is to reduce the execution time of program by proposing the VR technique but as the results it seems that we are proposing a technique to degrade the performance. This is because there are many other factors that contribute to overall performance, we will discuss them next.

The VR technique, some instructions will be inserted to the assembly source and these instructions have to be performed too, although our technique can bypass performing some instructions, the inserted instructions also make an impact in the performance. We will describe the situations when a function hit and does not hit.

In reusing the functions, the execution time which should be considered is the time that the program losses in establishing the reuse test. In dynamic

function reuse, the reuse test establishes only one time as shown in Figure 5.6. At the beginning of the body, thus the execution time in reuse test is not much when compared to the case that the function hits the reuse buffer, so it can bypass the execution, But we will show next that in static function reuse, the reuse tests occur more than one time and the more the reuse test establishes, the more the time losses in executing.

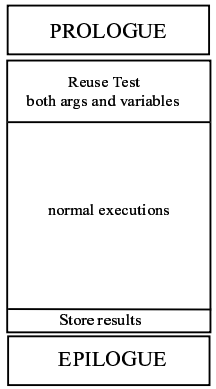

Figure 5.6: The reuse test establishes at the dynamic instruction

5.6.1 The situation when a function is not hit

Figure 5.7 shows the situation when a function does not hit the reuse buffer, the reuse test establishes the parameters test and the result found that there is no previous result for this entry. The reuse test fails, so the function must perform the executions until the end of function and execute the instructions inserted by the translator in order to keep the value of parameters and the global variables for storing them in the reuse buffer at the end of function.

Note that when the reuse test fails, it does not need to establish the next reuse test again. Such in this case, when the parameters test fails, it does not need to perform the global variables test again. In this situation, VR technique does not bypass the executions or decrease the number of executions, so the performance of VR technique is worse than non-VR technique.

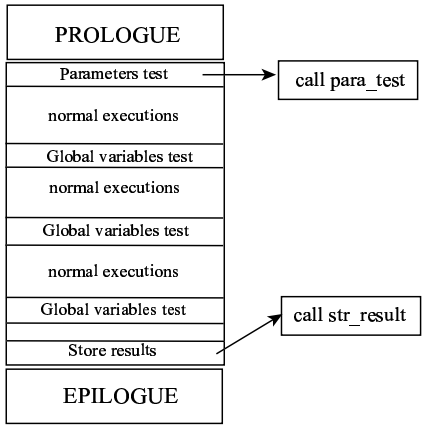

Figure 5.7: When the entry does not hit the reuse buffer

In the parameters test, the reuse test calls the subroutine which performs the function identity and all parameters test, the address of this function and the value of the parameters are first saved in the temporary buffer, the parameters test checks if all fields of this entry are the same as their previous results or not, If they are the same as the previous results, it will look in the fields of the global variables in the buffer, if there is no global variable, this test is finished and it jumps out to use their previous from the reuse buffer, if there are still the global variables in the buffer, it is necessary to perform the global variables check next.

This test concerns with many instructions and it depends on the complexity of the reuse buffer and the algorithm used in searching the entry. Next is the global variables test, at every time a global variable is first found in the function, the reuse test establishes test with this variable address and its initial value, unlike parameters test which tests all the parameters at the same time, this test performs each global variable and if this variable is found in the second time there is no need to perform the test again but its result will be saved in the temporary buffer in case that this function needed to be stored in the reuse buffer.

5.6.2 The situation when a function is hit

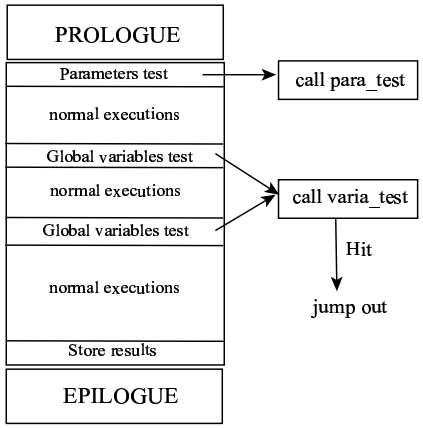

Figure 5.8: When the entry hits the reuse buffer at the middle of function

Figure 5.8 shows the situation when a function hits the reuse buffer. First of all, the reuse test establishes the parameters test at the beginning of the function, and those parameters are found in the reuse buffer but this function has the global variables so it cannot be judged at this point that it is hit. Then the reuse test establishes the global variables test in the next step, at the first global variable test, it is found that there are the other variables in the reuse buffer for this function too, so it has to test until the last variable, when the last variable was tested and it is hit the previous results, the execution jumps out to read the results from the buffer and store the results into the value return from a function and every global variables. The remains of instructions do not need to be performed. In this situation, the reuse test finishes the test at the middle of the function, the instructions which were omitted are more than the instructions were inserted to the functions, therefore, VR technique decreases the number of executions and the performance of VR technique is better than non-VR technique.

The situation when a function hits the reuse buffer is not always guaranteed

that the performance is improved as we describe the next situation.

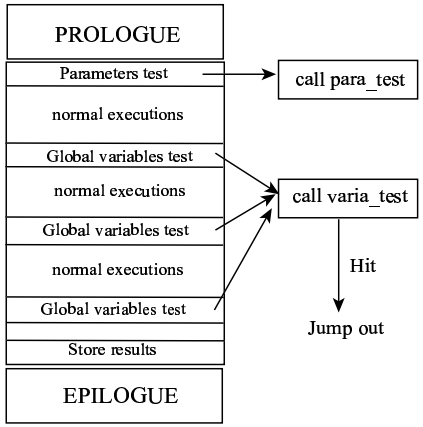

Figure 5.9: When the entry hits the reuse buffer at the end of function

In C, the global variables can appear everywhere in the program even at the end of function. Figures 5.9 shows the situation when a function hits the reuse buffer at the end of function. In this case, the reuse test establishes the global variables test at the end of function before it is hit, so the number of executions which can be bypassed are not much compared to the number of instuction inserted to the test, therefore, in this situation VR technique are likely to increase the number of executions and the performance of VR technique is worse than the non-VR technique.

5.7 Future Works

In this chapter, we envision future work proceeding on several different. First of all, we enlarge the restricted number of parameters and number of global variables. then we enable the address parameters in order to increase the static percentage of functions translated.

5.7.1 To enlarge more parameters number and variables number

At first, we limit the parameter number to four and the global variables number to ten and evaluate the results, but the results shown in Figure 5.1 is not impressive. In order to increase the static percentage of the functions translated by the translator, we enlarge the number of parameters to six and the number of global variables to twenty and evaluate the results

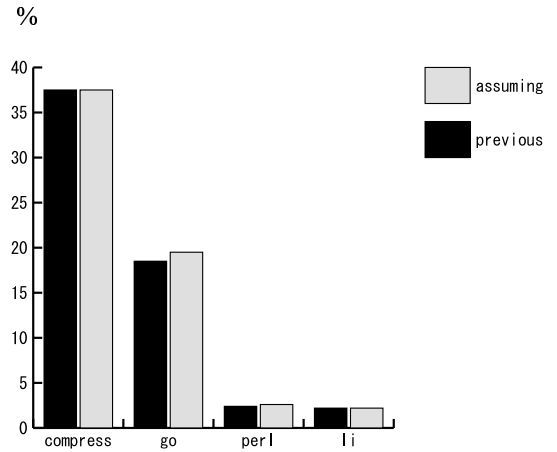

Figure 5.10: The percentage of functions captured by the translator when parameter = 6, global variables = 20

Figure 5.10 shows the results when we enlarge the restricted number but the results is not obviously different with the previous results. The results are almost the same as the previous results in *compress, perl, and li* This is because in the three benchmarks, most functions have not many parameters already and even the number of the global variables are enlarged, the number of the static percentage is limited by the nested functions. This behavior is also in *go*, although its result is quite different from the previous result.

5.7.2 To enable the functions which have address parameters

As the results shown in Figure 5.1, the main factor to decrease the percentage number of the functions which can be captured by the translator is that we cannot cope with the functions which have address parameters, in this results we show the percentage of the functions which will be captured by our translator if we can cope with the address parameters. To evaluate these results we modify the translator program to enable address parameters, we do not provide a special check with address parameter, we cannot evaluate the dynamic percentage of hit functions because executing these modified sources cause execution error.

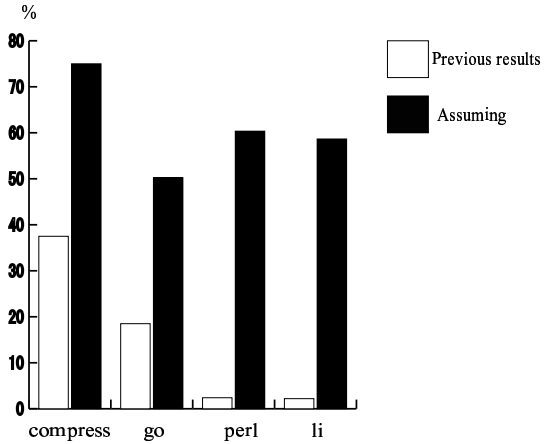

Figure 5.11: The percentage of functions captured by the translator if we enable the pointer address

Figure 5.11 shows the percentage of translated functions which should be obtained if we enable the address parameters. The results are high in *compress, perl* and *li* but there are not high in *go*, this is because in *go* many functions have global variables more than we restricted.

5.7.3 To address the functions which cannot be reused

As we described in the previous chapter, the function which cannot be reused makes a great impact in the performance. Even if we can cope with the nested

functions, but this problem still remains and makes an impact in the performace. Because the variable with has this problem in the leaf function will make an effect to the up function. A global variable whose value is changed every time when the function is called cannot be reused by our technique. But some variables value was predictably changed.

Table 5.2: A function which cannot be reused

func	...	global variables		
		addr	init	last
62030	...	23493	10	16
62630	...	23493	16	20
62630	...	23493	20	24
⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮
62630	...	23493	24	28

Table 5.2 shows the result of the global variable whose value is predictably changed. Value Prediction technique uses the previous results and unlike the Value Reuse technique, it predicts the result by using the value locality or the predictable pattern. From the table, in this case the previous results show that this variable value is changed by four every time, so it can predict the next result by adding 4 to the value when this function is called next time. However, if the value is mispredicted, instructions dependent on that value are re-executed.

5.7.4 Speedup Time in Reuse Buffer

As discussed in the previous chapter, searching an entry in the reuse buffer or establishing the reuse test makes an impact on execution time and it depends on the complexity of the reuse buffer and the algorithms used. To reduce the execution time in searching an entry in the reuse buffer, there are many methods to do it, redesigning the reuse buffer to make it less complex, using a better algorithms in searching the entry such as quick sort, merge sort. However, in software structure it cannot reduce the execution time impressively.

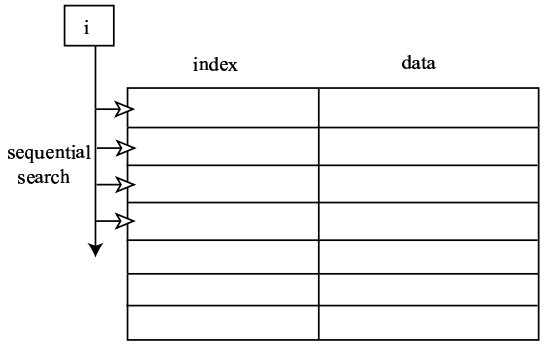

Figure 5.12: Reuse Buffer built by RAM

Figure 5.12 shows the structure of the reuse buffer built by RAM. To search an entry in RAM, it looks up the index and compare this index by each block of the RAM. As the algorithms used in our technique, searching the function blocks is from the first block to the last block thus it requires $O(n)$. But we arrange the data from the small to large, so it improves the algorithms in searching the entries of the same function to $O(\frac{n}{2})$. The question comes as why don't we use the latter algorithms with the former algorithms? or why don't we use a better algorithms to reduce the execution time. We know that even if we use the best algorithms such as quick sort, heap sort or merge sort, the timed can be reduced to $O(\log n)$ and in software structure, the time cannot be reduced more than $O(\log n)$.

By the problem that looking up the entry in RAM follows by sequential search, we propose the hardware technique or general propose CAM structure to be the reuse buffer. Figure 5.13 show how searching an entry occurs in CAM. Unlike RAM, to search an entry in CAM, it looks up the index and compare this index to every block at the same time and the result will be filtered to one output, thus searching an entry finished by only one time or $O(1)$ or in parallel. This advantage makes a great impact on the performance. This is the part of our goal at the beginning of this works. We try to compare time between our technique, with CAM, and the non-VR technique. Because of the limit of time,

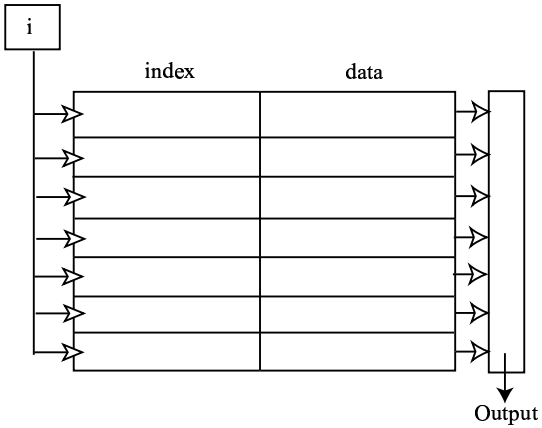

Figure 5.13: Reuse Buffer built by CAM

we will left this in the future works.

Chapter 6 Conclusion

In this paper, we introduced and studied the concept of Value Reuse in function-level. Observations suggest that in a program execution, many functions are executed repeatedly with the same parameters, generating the same results, this causes the redundancy in program. Many techniques have been proposed to reduce such the redundancy, including the Instruction Reuse technique, it uses the idea of storing the previous results of the instructions in the results buffer, if those instructions are encountered with the same operands again, it uses the previous outcome without performing the execution again.

We have proposed Value Reuse in function-level which is similar to the Instruction Reuse technique, but our technique is to reduce the instructions engaged in the term of function, some functions have hundreds or thousands of instructions and if those functions are reused, it can bypass performing many instructions unlike the Instruction Reuse which can bypass one single instruction per reused. In doing so, we built the translator program to modify the assembly source code, the translator inserts some instructions in the assembly source in order to get the values from functions, check and store these values in the reuse buffer. After the assembly sources are translated, when these new sources are being executed, the inserted instructions in the functions will look in the reuse buffer and if there are previous results stored in the reuse buffer, it does not need to perform executions through the end of function, it uses the previous results stored in buffer then return to where it was called.

Our technique is static and the merit of the static technique is that it does not require a special or complex hardware to be the Reuse Buffer. We built the Reuse Buffer in software structure, Our results showed that the percentage of translated function is not impressive because we cannot cope the functions which have address parameters and some functions have many global variables more than we limited and also some functions are nested, our techniques cannot reuse these functions, the next results showed that the percentage of reused hit is low, this is because some functions produce the different results every time they are called, this problem cannot be addressed by our technique. The results also

showed that the amount of redundancy captured by our technique is low due to the low percentage of hit and number of functions captured by the translator. Moreover our technique degraded the performance in execution time, this is because the translator inserted some instructions to establish the reuse test. The more reuse test occurs, the more instructions have to be performed, those instructions make a great impact in execution time

Finally, we discussed some methods to upgrade the performance in the future works, at first to increase the number of functions captured by the translator we will enable the functions which have the pointer parameters, then to increase the percentage of hit, we proposed the method to address the problem of the functions which cannot be reused by using Value Prediction technique which can predict the next results by the previous results. The last attempt is to reduce the execution time while establishing the reuse test, we know that it is difficult to reduce the time used in searching an entry in software structure even we use a good algorithm, we have proposed the Reuse Buffer in a simple hardware structure like general purpose CAM structure in order to reduce the time used in looking up the reuse buffer.

Acknowledgments

I would like to express my sincere appreciation to continuous support from my advisors, Professor Shinji Tomita and Associate Professor Yasuhiko Nakashima, through the trials and tribulations of this thesis. Furthermore, I would like to thank them for their encouragement and invaluable comments throughout a preparation this thesis. This thesis would not be possible without their guidance and inspiration.

I am deeply grateful to Associate Professor Toshiaki Kitamura who has helped me giving valuable advice for this research. I am also deeply thankful to every teacher who taught me many kinds of knowledge, which helped me completing this research.

My thanks go to Associate Professor Shin-ichiro Mori, Associate Masahiro Goshima, and the members of Computer Architecture and Parallel Processing Laboratory in the Graduate School of Informatics for their valuable advice, continuing encouragement, and deep friendship.

In addition, I would like to express my appreciation to Japanese Government (Ministry of Education, Science and Culture) for a scholarship during my academic years at Kyoto University.

Finally I am deeply grateful to my family who always give me endless love and willpower since I came to Japan.

References

[1] A. Sodani and G. S. Sohi: An Empirical Analysis of Instruction Repetition, *Proc. of 8th International Conference on Architectural Support for Programming Languages and Operating Systems* (1998).

[2] M. H. Lipasti, C. B. Wilkerson and J. P. Shen: Value Locality and Load Value Prediction, *Proc. of ASPLOS VII*, pp. 138–147 (1996).

[3] M. H. Lipasti and J. P. Shen: Exceeding the Dataflow Limit Via Value Prediction, *Proc. of 29th International Symposium on Microarchitecture*, pp. 226–237 (1996).

[4] A. Sodani and G. S. Sohi: Dynamic Instruction Reuse, *Proc. of 24th Annual International Symposium on Computer Architecture*, pp. 194–205 (1997).

[5] A. Sodani and G. S. Sohi: Understanding the Differences Between Value Prediction and Instruction Reuse, *Proc. of 31st Annual International Symposium on Microarchitecture* (1998).

[6] H. Abelson and G. J. Sussman: *Structure and Interpretation of Computer Programs*, McGraw Hill, New York (1985).

[7] S. H. Harbison: An Architectural Alternative to Optimizing Compilers, *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, pp. 57–65 (1982).

[8] S. E. Richardson: Caching Function Results: Faster Arithmetic by Avoiding Unneccessary Computation, Technical Report SMLI TR-92-1, Sun Microsystems Laboratories (1992).

[9] S. F. Oberman and M. J. Flynn: On Division and Reciprocal Caches, Technical Report CSL-TR-95-666, Stanford University (1995).

[10] S. Jourdan, R. Ronen, M. Kekernam, B. Shormar and A. Yoaz: A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification, *Proc. of 31st Annual International Symposium on Microarchitecture* (1998).

[11] N. Weinberg and D. Nagle: Dynamic Elimination of Pointer-Expressions, *Proc. of International Conference on Parallel Architectures and Compila-*

tion Techniques, pp. 142–147 (1998).

[12] C. Molina, A. González and J. Tubella: Dynamic Removal of Redundant Computations, *Proc. of the ACM International Conference on Supercomputing*, Rhodes(Greece) (1999).

[13] J. Huang and D. Lilja: Extending Value Reuse to Basic Blocks with Compiler Support, *IEEE Transactions on Computers*, Vol. 49, pp. 331–347 (2000).

[14] J. Huang and D. Lilja: Exploiting Basic Block Value Locality with Block Reuse, *Proc. of 5th. International Symposium on High-Performance Computer Architecture* (1999).

[15] Richard P. Paul: *Sparc Architecture, Assembly Language Programing, and C*, Prentice Hall, New Jersey (1994).

[16] David L. Weaver and Tom Germond: *The Sparc Architecture Manual*, Prentice Hall, New Jersey (1994).