An Interface between Single Assignment C and Vector Pascal

Bin Li

July 2007

A Dissertation Submitted for the Degree of Master of Philosophy

to the Faculty of Information and Mathematical Sciences,

University of Glasgow

Declaration

This dissertation is available for Library use on the understanding that it is copyright material and that no quotation from it may be published without proper acknowledgement.

I certify that all material in this dissertation which is not my own work has been identified and that no material has previously been submitted and approved for the award of a degree by this or any other University.

Abstract:

This dissertation contains an overview of the research I've been doing over in Glasgow University, which is mainly a project of developing an interface between two array programming languages, Single Assignment C and Vector Pascal, to combine them together by using the Vector Pascal code generator for Single Assignment C.

Single Assignment C provides support for multi-threading but it doesn't contain any utilization of SIMD technology, and Vector Pascal implements array operations with the help of SIMD instruction sets of modern general processors. Thus my hypothesis is that this combination will let the program enjoy higher run-time performance compared to the one which is only compiled by using Single Assignment C's compiler.

This dissertation explains the detail of designing and implementing this interface between these two languages; and the system to manipulate the three parts, i.e. the interface and the two languages' compilers together to make them work automatically. The interface is generally developed based on traversal over Syntax Tree and involves works of vectorization and loop unrolling.

Meanwhile, a benchmark testing system to validate my hypothesis is created and introduced in this dissertation too, which is accompanied with the testing results and analysis.

Acknowledgement

Completing the research in this thesis has been a tough journey in my life. Many help and support was given by people I met along the journey. By this opportunity I would like to express my greatest gratitude to all of them.

First of all, I would like to thank my supervisor, Dr W. Paul Cockshott. During the years of his supervision, he has created a friendly and helpful environment for me, where my understanding about scientific problems, particularly in designing programming languages and compilers have been greatly transformed and upgraded. His profound knowledge in various areas impressed me a lot and he is also very kind to share his knowledge with others. His sensible insights and thoughtful suggestions have been valuable in my research and can be witnessed in this thesis.

Secondly, I would thank my parents. Without their sponsoring, it is impossible for me to study in this country as an oversea student. They also provide me many supports mentally. It was them who encouraged me when I met difficulties and felt depressed in my life during the past time.

Thirdly, I would like to thank Dr. Sven-Bodo Scholz and Dr. Clemens Grelck for their kindly helps for me in using Single Assignment C and answering my questions.

Finally, I gratefully acknowledge the friends around me in Faraday Lab, especially Dr Yijun (Michael) Xiao who gave me a lot of advices for my research and living in this country.

Declaration 2
Abstract: 3
Acknowledgement 4
Chapter 1 7
Introduction 7
1.1 Objective of the Research 7
1.2 Hypothesis 8
Chapter 2 9
Background and Investigation 9
2.1 "SAC" and Functional Programming Languages 9
2.1.1 Functional Programming Languages 9
2.1.2 Single Assignment C's language Features 12
2.1.3 Multi-Threading 14
2.2 SIMD and Vector Pascal 15
2.2.1 SIMD 15
2.2.2 Data Parallel Languages 17
2.2.3 Vector Pascal 18
2.3 Vectorization 19
Chapter 3 20
Overview of the Project 20
3.1 Structure of the Whole System 21
3.2 Organizing the Files 24
3.3 Structure of the Interface 25
Chapter 4 27
Generating Optimized Vector Pascal Code 27
4.1 Porting to normal Vector Pascal Code 27
4.1.1 Structure of the First Stage 29
4.1.2 The Outline of the First Stage 30
4.1.2.1 A Single Assignment C Sample 30
4.1.2.2 Output of Single Assignment C's compiler 31
4.1.2.3 Working Progress of the First Stage of My Program 34
4.2 Vectorized the Pascal Code and Unroll the For-Loop 36
4.2.1 Why vectorization and Loop Unrolling? 36
4.2.2 Goal of Vectorization 38
4.2.3 Vectorization & Loop Unrolling 39
4.2.3.1 Data Dependencies 40
4.2.3.2 Data flow Optimization 41
4.2.3.3 Forward Substitution 43
4.2.3.4 Loop unrolling and Vectorization 45
4.3 Further Optimization 47
4.3.1 A Weird Phenomenon 47
4.3.2 Tasks in this Step of Optimization 48
4.4 Benchmark System 51
4.5 Key Notes of the Implementation 53
4.5.1 Building Syntax Tree 53
4.5.1.1 SableCC 53

4.5.1.2 “Specification File” to Activate SableCC 55
4.5.1.3 Generating the Classes by using SableCC 58
4.5.2 The first stage 59
4.5.2.1 Generating the Working Classes 59
4.5.2.2 The traversal classes 59
4.5.2.3 Other Working Classes 61
4.5.2.4 Main Class 62
4.5.3 The Second Stage 63
4.5.4 The Final Stage 63
4.5.5 The Shell Controlling File 64
Chapter 5 65
Benchmark Testing and Analyzing 65
5.1 Introduction 65
5.2 Benchmark Environment 65
5.3 Benchmark Test 69
5.3.1 Benchmark Test on Integers 70
Test 1 71
Test 2 74
Test 3 79
Test 4 83
Test 5 88
Test 6 91
Test 7 95
5.3.2 Benchmark Test on Floating Point Numbers 99
Test 8 100
Test 9 104
Test 10 106
5.4. Conclusion 109
Chapter 6 111
Conclusion and Future Works 111
6.1 Research Summary 111
6.1.1 Analyzing and Designing the Grammar 111
6.1.2 Transforming to Pascal Code 112
6.1.3 Vectorization and Loop Unrolling 112
6.1.4 Benchmark Test 112
6.1.5 Conclusion 113
6.2 Future Works 113
6.2.1 Updates and Modification 113
6.2.2 Multi-Threading 114
6.2.3 Migrating to Other Platforms 114
6.3 Generalization of the Research 115
Appendix 116
Assembly Files Generated by GCC and Vector Pascal 116
1. The assembly file generated by GCC 116
2. The assembly file generated by Vector Pascal: 117
Bibliography 120

Chapter 1

Introduction

1.1 Objective of the Research

Single Assignment C and Vector Pascal are both array programming languages, while Single Assignment C provides support for multi-threading and Vector Pascal provides the utilization of multimedia instruction sets inside of modern general purpose processors.

Almost contained in all multimedia instruction sets of the modern processors, an SIMD instruction set is very useful in many areas because that it can greatly enhance the performance of array operations. Meanwhile, since the operations on arrays are very common when dealing with scientific computing tasks, it is possible to combine these two languages' features together to achieve better runtime performance.

Therefore, the objective of my research is to develop an interface between these two languages to port array operations which are suitable for utilizing SIMD features from Single Assignment C to Vector Pascal, and compare the run-time performance with original programs which are only compiled by Single Assignment C to observe the performance enhancement.

The compilers of Single Assignment C and Vector Pascal, the interface between these two programming languages, and the benchmark system have been integrated into a system which is able to generate executable programs, perform benchmark testing, and produce result table automatically.

1.2 Hypothesis

Modern general processors normally integrate SIMD instruction set inside them, which are suitable to process vectorized code faster than traditional sequential code. Based on this point, Vector Pascal implements utilizing SIMD instructions to perform array operations towards several different processor architectures.

Therefore, when dealing with the programs output from Single Assignment C which contain sequential operations on arrays, it is possible to vectorize them and then employ Vector Pascal compiler as the code generator to compile them to achieve better runtime performance.

Chapter 2

Background and Investigation

2.1 “SAC” and Functional Programming Languages

“SAC” is the abbreviation of “Single Assignment C” [Scholz, 2003], one of the two programming languages which are combined together in this project. It is a functional programming language for generic array processing [Scholz,2003] whose compiler implements many high level sophisticated optimizations based on static single assignment form and using GCC, the GNU C/C++ compiler as its backend.

“Single assignment” is a concept which describes a programming language or representation in which one can bind a value to a variable at most once in a function. Based on this concept, a compilation technique called “Static Single Assignment Form (SSA)” has been developed. It is an intermediate representation in which each variable “has only one definition in the program text.” [Appel, 2003], i.e., by using SSA, existing variables in original intermediate representation will be split into different versions. Typically, each new version of a variable may be indicated by the original name plus a subscript. Since the new variables are numbered, it will be very easy to trace them back to the original definition points. This technique is widely used to reduce side effect in current functional programming languages, such as Haskell [Chakravarty, 2000]. It is useful for improving the results of many compiler optimizations, because it can simplify the properties of variables.

2.1.1 Functional Programming Languages

Functional programming delivers a programming paradigm that treats computation as evaluating mathematical functions. Different from imperative programming style which usually relays on changes in state, functional programming emphasizes the application of functions.

Functional programming languages have several conceptual advantages over imperative programming languages. Although they have not yet found a broad acceptance by application

programmers outside the functional community, functional languages might be quite suitable in some areas such as numerical applications involving complex operations on multi-dimensional arrays, because the dominating aspects for the choice of a programming language are caring much about execution speed, support for concurrent program execution and the potential for code reuse as well as code maintenance of existing programs [Scholz, 1994]. Meanwhile, using functional programming languages will help developers focus on algorithms instead of spending too much energy on writing correct and efficient code.

One of a common feature of functional programming languages is called "higher-order functions" which means "functions that operate over functions" [Goldberg, 1996]. Functions are higher-order when they accept functions as arguments and/or return functions as results.

Pure functional programming languages "contain no side-effect" [Hughes, 1989], and they usually enforce referential transparency which means if two expressions are equal for some sense, then one can be substituted by the other in any other expressions where it presents without affecting the result of the computation. Without side-effects, the order of evaluating the functions' values does not affect the result of the computation, which makes it easier to optimize the programs and implement parallelization. However, not all functional programming languages are pure, i.e. the impure ones allow the existence of side-effect.

In functional languages, iteration is usually accomplished in the way of tail recursion (sometimes called as tail end recursion), instead of using various kinds of loops which are used in common imperative languages. Tail recursion is a special form of recursion in which "the last operation of the function is a recursive call" [Paul, 2004], and it can save memory cost at run time compared with normal recursion.

Another way of categorizing functional programming languages is to check whether they use strict or non-strict evaluation. If arguments passed into a function are evaluated before the function call, it is a strict function evaluation; on the contrary, if unevaluated arguments are passed into the function and when will they be evaluated is determined by the caller, it is a non-strict function evaluation.

Moreover, some functional programming languages also provide support for array operations and they can be divided into two groups. The first one includes programming languages that provide

array operations similar to APL [Iverson, 1962] which supports sub-array selection, folding, rotation, transposition etc. Languages of the second group allow for more direct manipulations of arrays. Typical examples are Haskell with its array comprehension and SISAL with its “For-loops”, both of which provide iterations over pre-specified index intervals [Scholz, 1994].

SISAL (Streams and Iteration in a Single Assignment Language) is a general-purpose single assignment functional programming language with strict semantics, automatic parallelization, and efficient array handling [Raymond, 2000]. It is designed for numerical applications, but it suffers from two major deficiencies [Scholz, 1994]: firstly it does not have integrated I/O operations. Secondly, the primitive operations on arrays demands boundary checks and element selection, and the loop statement requires the specification of explicit starts, stops and strides. These factors render all programs dimension-dependent. Besides, the SISAL compiler targets the shared-memory systems, and to run SISAL on distributed systems such as cluster computers, the existing run-time systems employ a virtual shared memory concept [Scholz, 2003]. As a result, their performance will be notably depressed by the processing resources spent on the scheduling of the memory.

To overcome these problems, people from University of Kiel, University of Lübeck, University of Toronto and University of Hertfordshire cooperated and developed a functional version of C which is called Single Assignment C (SAC) with the intention of falling back on existing compiler technology to generate fairly efficient code for a large variety of platforms.

Single Assignment C is a “strict purely functional programming language” [SAC, 2006] which is designed for the use in numerical applications with the syntax very similar to that of C. In another word, we can say that Single Assignment C is a functional extension of the subset of standard C programming language. Particularly, Single Assignment C is focused on efficient support for array processing. To generate more efficient code in the compilation phases, some functional language features which are removed since they are not considered as essential elements for numerical applications, such features including “higher-order functions, polymorphism, or lazy evaluation, are not (yet) supported by SAC” [SAC, 2006].

2.1.2 Single Assignment C's language Features

As a concession to the programmers with an imperative background, Single Assignment C employs a functional subset of C [Scholz,2003] as its language kernel--- the word "functional" here indicates "a straightforward mapping of language constructs to an applied λ-calculus" [Grelck, 2001].

To avoid side-effect, Single Assignment C drops "global variables, pointers and any other kinds of data structures which rely on pointers" [Grelck, 2001]. For similar reason, the control-flow instructions such as "**break**", "**goto**" and "**continue**" are dropped as well but other structures are kept with some modifications.

In Single Assignment C's syntax, functions only have one difference compared to C, i.e. Single Assignment C supports multiple return values, and the return values are not limited to only one type while in C there can only be one return value at most. As a result, Single Assignment C's function header begins with one return type or sequences of return types separated by comas. Single Assignment C inherits C's four basic scalar types: integer numbers, single precision floating numbers, double precision floating numbers and characters, and it also "adds" a new type, "bool"--- C does not actually distinguish Boolean type from integer, but Single Assignment C does. Moreover, it supports the mechanism of "function overloading" [Cardelli, 1985], i.e. there can be several functions with same name but different parameters. The order of function definitions is irrelevant, so that one function can be used before defined [Grelck, 2001].

The function's body is normally constructed by three parts in Single Assignment C, i.e., variable declarations, assignments, and a trailing return statement. There is only one kind of local scope, the function body scope. The assignments might be a single let-assignment or compound assignment statements. The former one just binds one value to a variable while the latter one might be a conditional or one of the three loop structures in C. As functional languages usually do, an if- clause is interpreted as a functional condition with the continuation code being copied into each branch, and loops are converted to tail-end recursive functions. These works will be done automatically by the compiler during generating Single Assignment C's optimized intermediate code, and the programmers can focus on designing algorithms for solving the problems.

Single Assignment C's language kernel only supports scalar types and it is extended by taking arrays as first class objects. Arrays in Single Assignment C are represented in a way quite similar to those interpreted array languages such as APL and J, i.e. by shape vectors which provide the arrays' structure information and data vectors which show all element values in arrays. The shape vector's length denotes the array's dimension and every element of it defines the array's extent in each relative dimension. Single Assignment C also provides some built-in functions on arrays to do some basic operations, a list of these functions are excerpted from its introduction on its website [SAC, 2006] as follows:

"

- dim (**array**): This function returns **array**'s dimensionality (or rank).
- shape (**array**): This function returns **array**'s shape vector;
- sel (**index_vector, array**): It returns the elements of **array** selected by vector **index_vector;**
- reshape (**new_shape_vector, array**): This function creates a new array with all elements in the data vector of **array** but the shape is defined by **new_shape_vector;**
- modarray (**array, index_vector, value**): This function generates a new array which is same to **array** but elements selected by **index_value** will be given new values as **value**.

"

More complicated operations on arrays can be implemented by with-loop(s). "with-loop" is the Single Assignment C's specific language structure "for the definition of aggregate array operations" [Scholz, 1994] . A with-loop is basically consists of a "*generator*" and an "*operation*" [Scholz, 1994] . The *generator* defines an index vector which is used to select elements in **array**. There are three built-in functions in Single Assignment C for the with-loop, genarray (**shp, exp**), modarray (**array, index, exp**) and fold (**fold_op, neutral, exp**) [Scholz,2003].

The first one generates a new array whose shape is defined by vector "**shp**" and elements are set according to data vector **"exp"**; the second one defines an array whose shape is the result of function "shape (**array**)" and all elements are values of **exp** except those selected by vector **index**, the parameters index and exp are optional; the third function allows the specification of reduction operations, i.e., setting out with **neutral,** the value of **exp** is computed for each index vector from the specified set and subsequently folded by using the **fold_op.**

The optimized intermediate Single Assignment C code will be transformed into C code and the GCC compiler is used as the back end.

To make it clear, let's make a brief conclusion of Single Assignment C's language features. Most of them are quite similar to C, except:

- Single Assignment C functions with multiple return values are translated into C functions with a single return value and additional reference parameters;
- Tail-end recursive functions which arise from loop constructs in the original program are converted back into loops in C code; the conditionals which are lifted out into separate function definitions are re-inlined at their original locations.
- Array types, built-in functions, and operators which do not exist in C will be implemented in Single Assignment C's library.

2.1.3 Multi-Threading

Single Assignment C not only provides support for sequential execution, but also provides support for non-sequential execution on shared memory architectures by realizing the concept of scheduling and controlling multiple threads within the shared address space of a single process based on POSIX multi-threading standard PTHREADS [IEEE POSIX, 1995].

In modern operating systems, the word "thread" actually indicates the phrase: a thread for execution. Thread is a relative low level concept compared with another one, "process", which is an instance of a computer program that is being executed. Technically, it is defined as an independent stream of instructions that will be scheduled to run as such by the operating system. Though it may be slightly different from one operating system to another, threads belonging to a process are usually created inside it and share the resources associated with that process, such as memory address space and file descriptors. They will have separated runtime stacks and communicate based on accessing the memory locations shared by them.

Multiple threads can be executed in parallel on many computer systems. Traditionally, multi-threading is implemented by means of "time slicing" which means a single processor switches between different threads very fast. Obviously it is not really simultaneous but just an illusion for

users because one traditional single processor which has one executive unit can do only one thing at a time. Thanks to the improvement of the technology, multi-processors or multi-core processor architectures have become widely used in small scale computers in recent years. Thus threads are able to run on different processors or cores literally simultaneously, and the performance is also enhanced by scheduling threads on different processors or cores.

Therefore, it is significant to write multi-threading programs for better performance than sequential program, but it is not very easy for programmers to achieve this goal with manipulating threads via system call provided by the kernel of operating system directly. In order to take full advantage of the capabilities provided by threads, a standardized programming interface is required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard [IEEE POSIX, 1995]. Implementations which adhere to this standard are referred to as POSIX threads, or PThreads. Most hardware vendors now offer PThread in addition to their proprietary API's [PThreads Programming, 2006].

2.2 SIMD and Vector Pascal

2.2.1 SIMD

SIMD which stands for "Single Instruction Multiple Data" is a technique employed to achieve data level parallelism [Hennessy & Patterson, 2003]. It indicates a model of computing in which a single instruction causes the same mathematical operation to be carried out on many operands, or pairs of operands at the same time. The SIMD model was firstly popularized in large-scale supercomputers (as opposed to MIMD [Hennessy & Patterson, 2003] parallelization) and the support for smaller-scale SIMD applications now becomes widespread in personal computer area since almost all modern general microprocessors contain multimedia instruction sets implementing SIMD operations. Today this term is associated almost entirely with the smaller units.

Almost all the general processors vendors' products have already included instructions which supports the SIMD technique in their instruction sets now, though the implementations are different.

These SIMD instruction sets are under the trademarks of PowerPC's AltiVec, Intel's MMX, SSE to SSE4, AMD's 3D! NOW, SPARC's VIS, Sun's MAJC, PA-RISC's MAX, and MIPS' MDMX and MIPS-3D.

The SIMD technique in modern general processors' instruction sets can be characterized as units that support "packaged fix-length vectors" [Eichenberger, 2004]. An application that may take the advantages of SIMD is one in which the same value is being operated on a large number of data points, and it is a common operation in many multimedia applications such as image processing.

Now let us take an example to briefly illustrate the procedure of using SIMD instruction to perform operations.

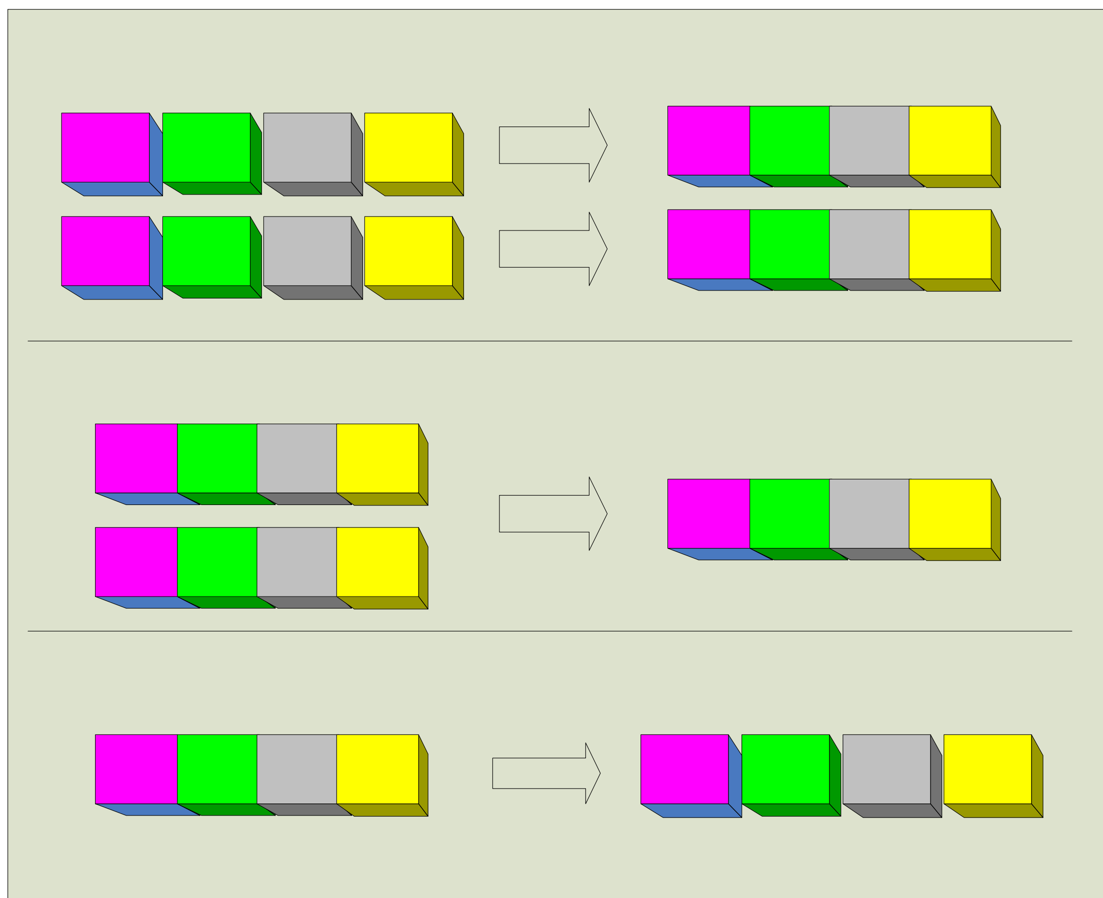


Figure 2.1 Using SIMD instruction sets to implement parallelism. In this example, we assume that the processor has 128-bit registers and is processing 32-bit integers. Each block in this figure presents a data point, and the SIMD instruction is dealing with four of them at a time.

Figure 2.1 is an intuitive sketch map, suppose that we have 128-bit general purpose registers in our processor and we are processing 32-bit integers, we can firstly use SIMD load instruction to load

four data blocks (each one is a 32-bit integer in this example) into register 1 (whose length is 128 bits) from memory, then load another four data blocks into register 2 in the same way. After that we will let the processor add them together by issuing one SIMD addition instruction, keep the result in register 3 temporarily, and finally use SIMD store instruction to store them back to memory. The processor performs operations on four data blocks at one time in each step, while in traditional way the processor has to perform one action repeatedly on each data block in each step, since it can only process one data block at a time. Therefore, using SIMD instruction set can take much less time than to process data blocks one by one as in a traditional CPU design. If we let the processor deal with 64-bit floating point numbers, the processor works almost in the same way except that it can process two numbers (load & store) or two pairs of numbers (arithmetic operations) simultaneously instead of four when dealing with 32-bit integers. Therefore, the performance will be highly enhanced by utilizing SIMD instruction sets theoretically.

2.2.2 Data Parallel Languages

For the increasing demands of high-performance scientific processing, various kinds of parallel programming concepts have been developed in the past dozens of years. Exploitation of SIMD parallelism was the one adopted to implement data parallelism. Typically this involves two approaches. One is that the operators shall be overloaded to allow array-valued expressions if we implement a data parallel language based on an existing sequential language, so that operations on whole arrays like those introduced in APL will be valid, or else we have to develop a brand new language and considering employing some new operators to perform array operations. The second one is relative to loop unrolling. In practical, since different processor families have different implement of instruction set, compilers usually generate architecture specified program to maximize the performance.

For data parallelism, APL and J [Iverson Software Inc., 1996] represented radical breaks from their contemporaries, introducing novel notations. Many concepts of them are employed in later array programming languages' design, such as:

- Operations on whole arrays
- Array slicing

- Data reorganization
- Reduction operations
- Conditional operations

2.2.3 Vector Pascal

Vector Pascal incorporates APL's approach to data parallelism to provide "an efficient and concise notation for programmers using SIMD instruction sets by borrowing concepts for expressing data parallelism." [Cockshott, 2004] There are some complications to do with the semantics of operations between arrays of different sizes and dimensions. By following Kenneth Iverson's concept [Iverson,1979], Vector Pascal implements a consistent treatment to deal with these problems and it provides machine independent data parallelism.

Another highlight of Vector Pascal is that it adopts the concept of using multimedia instruction sets of modern processors to enhance the runtime performance when compiling vectorized code. The traditional ways of utilizing multimedia extensions of the instruction sets can be mainly divided into to categories. One is using in-line assembly language, and another is using intrinsic functions embedded in a high-level programming language [Eichenberger, 2004]. Both of these solutions are error-prone and hard to manipulate for programmers. Vector Pascal implements a more convenient programming model for the developers, i.e., people just need to write vectorized code in Vector Pascal which is a high level programming language, and then the compiler will unroll the loops, package the vectors and generate low-level assembly code which utilizes SIMD instructions automatically.

Therefore, we can write programs in Vector Pascal to process arrays directly and run the algorithm application based on array operations faster than ever, without being bothered by thinking about how to write code which is suitable for using SIMD instructions, but just simply focus on the algorithm of solving the whole problem and write code in our familiar way.

2.3 Vectorization

Vectorization is one of the most important & common technologies of advanced modern compiler optimizations. It is the process of converting a computer program from a scalar implementation, which does an operation on a pair of operands at a time, to a vectorized program where a single instruction can perform multiple operations on a pair of vector (series of adjacent values) operands. During this process, the compile needs to analyze and eliminate control and data dependences, and optimizes the program for execution on a vector processor [Baxter, 1989].

As one of the major features of modern computers, vector processing has received great attention and many researches on the topic of automatic vectorization have been done to find out some methods that may let the compilers convert scalar programs into vectorized programs without human assistance.

Automatic Vectorization refers to the automatic transformation of a series of operations performed sequentially, one operation at a time, to operations performed in parallel, several at once, in a manner suitable for processing by a vector processor, e.g. a processor contains SIMD instructions in its instruction set. The most common way is based on loop unrolling, with two steps.

- Find an innermost loop which is supposed to be vectorized
- Unroll this loop and generate vectorized code

There are several obstacles which may prevent the loop unrolling, including function calls and data dependence. Obviously, a loop with function calls inside is not able to be vectorized unless adjusting both the loop and the function manually. But it might be possible to overcome data dependences for a compiler. Data dependences present where two statement instances are accessing the same data element and at least one of the accesses is a write. The direction of data dependence is determined by "the sequential execution order of the statement instances of the program" [Tang, 1995]. The common strategy of dealing with data dependences is to parallelize statements which do not contain data dependence and enforce all the data dependences in the program to keep the original orders of the presents of those statements with data dependences to guarantee the correctness of the program.

Chapter 3

Overview of the Project

As introduced in last chapter, Single Assignment C is a high-level functional array language which has a high performance compiler called sac2c. Efficiency in program development is improved by its specification of array operations at a high level abstraction. Meanwhile, as a functional language with the single assignment feature, a variety of compiler optimizations are simplified and improved. Thus the efficiency in program execution, i.e. the runtime performance of programs both in time and memory consumption is enhanced by the compilation schemes including taking the cache hierarchy into account during the code generation and a mixture of conventional optimizations such as function inlining, constant folding, and loop invariant removal [SAC, 2006]. But Single Assignment C doesn't employ any support for SIMD which can highly improve the vectorized program's execution performance on arithmetic applications. So it will be worthwhile to use Vector Pascal as the code generator for Single Assignment C to deal with some array operations which are suitable for SIMD processing.

Now let's have a look on an example of utilizing SIMD instruction set to see the benefit brought by utilizing SIMD instruction sets. Suppose we have a microprocessor which contains 128-bit general purpose registers and our program needs to perform operations on arrays whose elements are 32-bit integers, we can let the processor operate on four integers at one time.

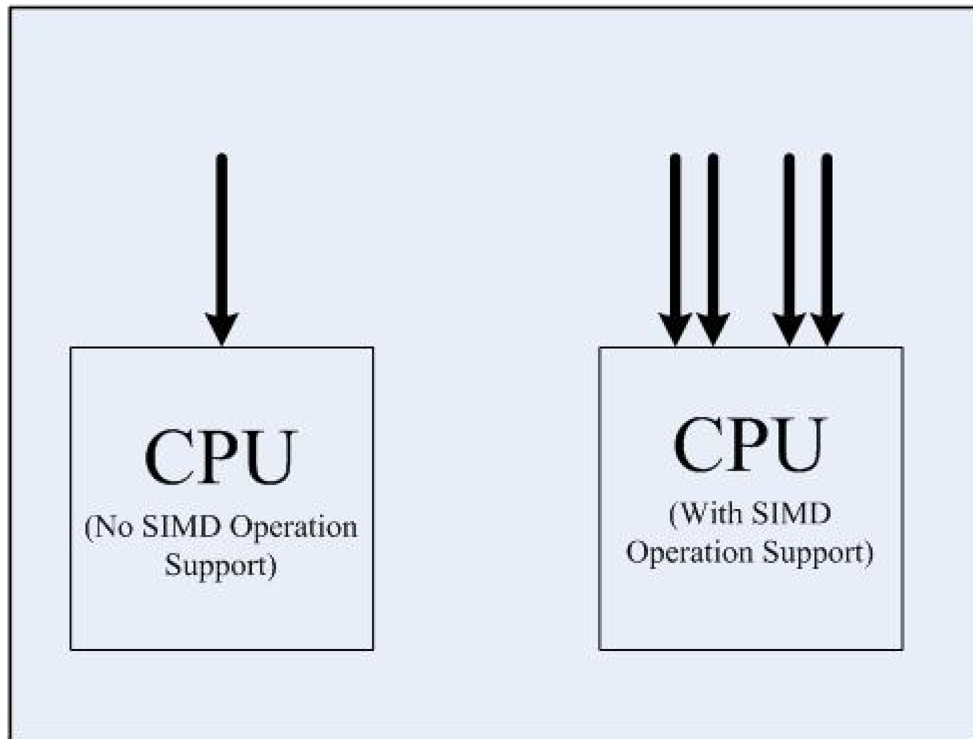


Figure 3.1 Comparison between programs without using SIMD instruction set (left) and using SIMD instruction set (right). Each arrow in this figure presents one data input stream.

As shown in Figure 3.1, the left one is a sketch map showing a processor working in traditional way (without using SIMD instruction sets) while the right one performs a program handling data based on 32-bit integers by utilizing SIMD features of the processor. By this comparison and considering Figure 2.1 in last chapter, we can initially expect that when processing 32-bit integers, the program generated under this combination could have a performance which would be theoretically about up to as four times faster as the opponent's compiled only be Single Assignment C, taking into account the SIMD instruction sets' characteristic of the processors. And the enhancement could be 200% when processing 64-bit floating point numbers. Moreover, if the techniques are developed in future, for example, the microprocessors employ bigger size registers, the enhancement can be further improved.

3.1 Structure of the Whole System

The inputs of the system of my programs are some C source code files generated by Single Assignment C; each of them contains a for-loop inside, because the Single Assignment C's compiler now can figure out the proper part of the arrays' operations which are suitable for utilizing Vector Pascal's SIMD feature during its compile-time optimizations on their Single Assignment C

intermediate code and port them into C code [Grelck, 2001]. These C files will be transformed into Pascal and vectorized by my program.

Considering given a Single Assignment C source code:

```
1   use Structures: all;
2   use SimplePrint: all;
3   int main()
4   {
5     v1 = genarray( [100], 2);
6     v2 = genarray( [100], 1);
7     for( i=0; i<10000; i++) {
8       v1 = with(iv)
9             (. <= iv <= .) : v1[iv] + v2[iv];
10            modarray(v1);
11    }
12    res = print( v1[0]);
13    return( res);
14  }
```

Program 3.1 A sample of Single Assignment C source code. This program adds array v2 to v1 for 10,000 times by using Single Assignment C's with-loop wrapped in a for-loop.

This program contains a typical with-loop of Single Assignment C from line 8 to line 10 as its innermost loop, which performs an addition operation on arrays *v1* and *v2*, and then stores the result to *v1*. The program enters the system at the top left of the figure shown below:

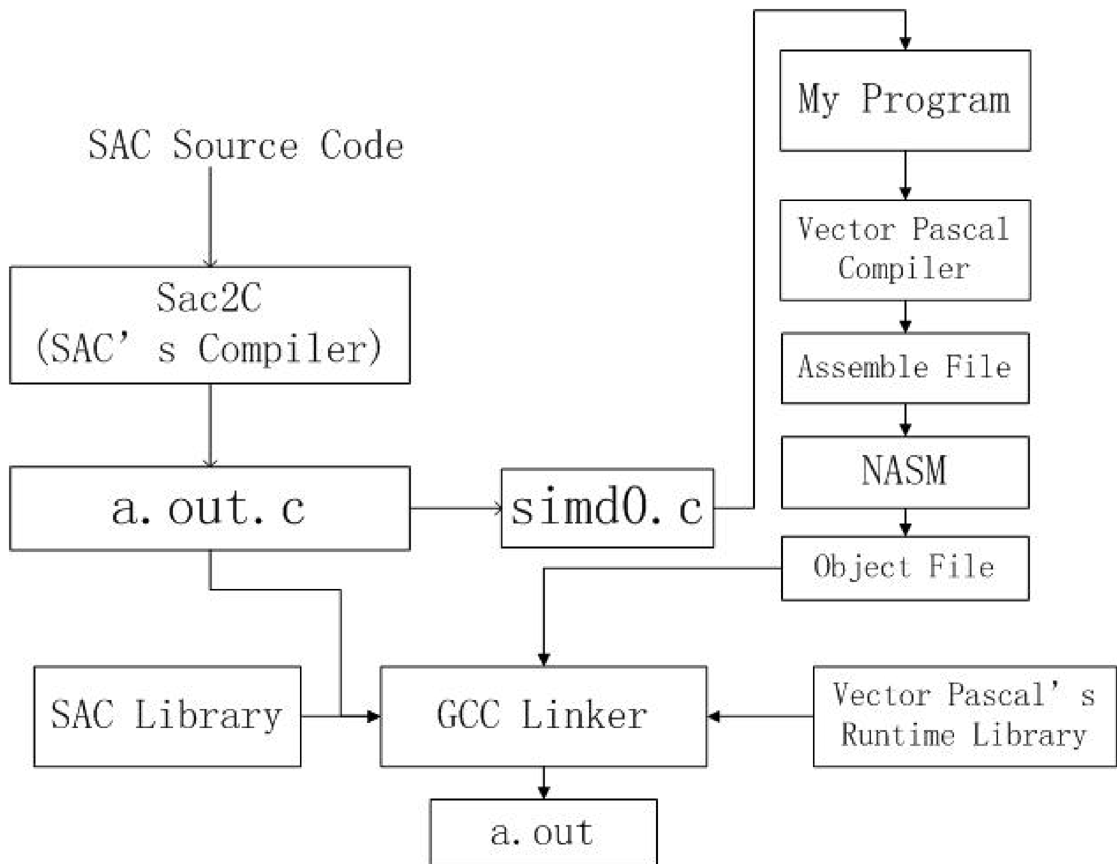


Figure 3.2 Structure of the whole system. This figure shows the whole system’s working process, which takes Single Assignment C’s source code, compiles it and using Vector Pascal’s compiler as code generator.

Figure 3.2 is the concise over view of the whole system’s structure, including the compilers of these two programming languages, my program (the interface between them) and the invoking of the GCC linker to link to generate executable file. Besides, I also created a set of script programs to integrate the components shown in Figure 3.2 together.

Because Single Assignment C’s compiler uses GCC as its backend, it will generate a C source file called “a.out.c” as the main file which will be compiled by GCC compiler. Meanwhile, it will also find out the innermost loop and port it to several C source files each with one for-loop inside. In this example, there is only one of them generated, name simd0.c as shown in figure 3.2.

These C source files will be named “simd” followed by a number designating its serial number. Since there is only one operation in this innermost loop (with-loop between line 8 and line 10 in Program 3.1), the compiler will only generate one source file, which is called “simd0.c” here (shown in the middle of figure 3.2).

“Simd0.c” will be transformed and vectorized by my program to Vector Pascal, compiled to source

code of assembly language by Vector Pascal's compiler, and then it will be translated into object file by invoking NASM assembler (it was changed to GAS since the latest update of Vector Pascal generates better optimized code for utilizing GAS than for NASM when processing floating point numbers). In the final stage, all compiled files and relative libraries files will be linked together by GCC linker to generate executable file, called a.out.

3.2 Organizing the Files

When compiling the Program 3.1, Single Assignment C's compiler generates the "simd0.c" file with content as follows:

```
1   {
2     int SAC_stop_SACp_pinl_337___nt2ot_21 = 100;
3     int SACp_pinl_337___nt2ot_21;
4     for (SACp_pinl_337___nt2ot_21 = 0; SACp_pinl_337___nt2ot_21 < 100; )
5     {
6       {
7         ;
8       }
9       ;
10      {
11        *SACp_pinl_332__emal_311__pinl_301___flat_139=
            SACp_pinl_330____v1[*SACp_pinl_342___wlidx_290__flat_14] );
12      }
13      ;
14      {
15        *SACp_pinl_332__emal_311__pinl_301___flat_139=
16          ((*SACp_pinl_332__emal_311__pinl_301___flat_139)+ (1));
17      };
18      ;
19      {
20        (SACp_pinl_333__emal_309_v1__SSA0_1[*SACp_pinl_342___wlidx_290__flat_14])=
      *SACp_pinl_332__emal_311__pinl_301___flat_139;
21      };
22      {
23        ;
24      }
25      *SACp_pinl_342___wlidx_290__flat_14=*SACp_pinl_342___wlidx_290__flat_14+1;
26        SACp_pinl_337___nt2ot_21 = SACp_pinl_337___nt2ot_21 + 1;
27    }
28  }
```

Program 3.2 Content of simd0.c. This program is generated by Single Assignment C and ported to my program as the input, it will be transformed into vectorized Vector Pascal code by my program.

We can see that there is only a for-loop in Program 3.2, and it is not encapsulated in a function. It works well in C programming language, because the file "simd0.c" will be used by "#include"

directive key word in a.out.c like this:

#include "/tmp/SAC_HCQA82/simd0.c"

In C programming language, this is equivalent to reading in the file and replacing the #include directive with the contents of the file. But it will cause a problem in my program. As mentioned in section 3.1, this "simd0.c" file will be transformed and vectorized to Vector Pascal code, and finally be compiled by Vector Pascal's compiler. Obviously, a C program can not "#include" Vector Pascal code directly. Thus I have to transform the simple use of "#include" into a function call to guarantee correctness of the program, and then encapsulate the file compiled by Vector Pascal's compiler into a library file so that the GCC linker can link it together with other files in the final step.

3.3 Structure of the Interface

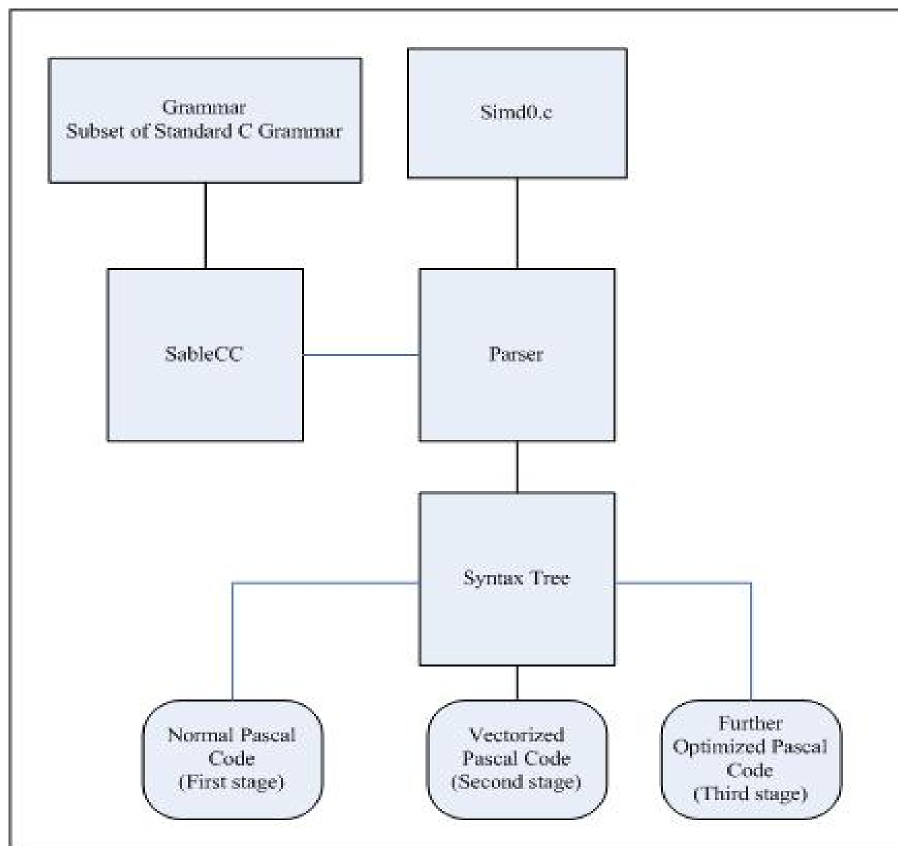


Figure 3.3 Structure of the interface between Single Assignment C and Vector Pascal. It is based on traversal over syntax trees, takes intermediate C code generated by Single Assignment C's compiler as input, and generates Vector Pascal code/ vectorized Vector Pascal code. And it will also do some other optimizations to achieve better performance in utilizing Vector Pascal's compiler as the code generator.

My program is based on traversal over the Syntax Tree built by using the grammar written by myself. Since the input file (simd0.c shown in figure 3.3) which will be processed by the parser is a C source program generated by Single Assignment C's compiler, and it only contains a for-loop, my grammar excludes those unnecessary features in C language for the goal of saving memory cost. In another word, it is a subset of standard C's grammar. Based on this grammar, a parser is built with the help of SableCC, and the parser will be used to the Syntax Tree by parsing simd0.c. More details of the grammar and building the parser will be expressed in next chapter which talks about the design and implementation of my program.

After that, the program can be considered as three different stages. The first one's goal is to translate the C code output by Single Assignment C's compiler into Pascal code for the initial test of the correctness of the system. In the second stage, my program will vectorize it and unroll the for-loop. The last stage is doing further optimization to gain better performance.

The result, or say, output, of each stage, is a series of Pascal source programs (or Vectorized Pascal source program). Since the Syntax Tree based on C grammar has already been built, I decide to keep using it instead of creating several new ones based on Pascal or Vector Pascal grammar. Therefore, the three stages are not like a stair which is totally one based on another. Instead, we may consider that the Syntax Tree based on C grammar as an intermediate layer once the normalization works have been done and the Syntax Tree has been adjusted.

Chapter 4

Generating Optimized Vector Pascal Code

As mentioned in section 3.3, my program consists of three different stages to achieve goals of translating to normal Pascal code, vectorization and further optimizations. In this chapter, I shall explain the details of implementing these three stages, and introduce the benchmark test system.

My works are mainly based on multi-pass traversal over a concrete syntax tree. I wrote the grammar to build a parser for generating the syntax tree by parsing input C files ported from Single Assignment C. The details of my work will be introduced in the last section of this chapter.

4.1 Porting to normal Vector Pascal Code

This is the first stage of my interface in which the program from Single Assignment C will be translated into normal Vector Pascal code, and then use the Vector Pascal compiler to generate assembly language source file. A brief example is as follows to illustrate the main task in this stage.

The C source code porting from Single Assignment C is like this:

```
for (i=loop_begin;i<=loop_stop;i++)
{
    Array1[i]=Array2[i]+Array3[i];
}
```

Program 4.1.1 a for-loop in C generated by Single Assignment C's compiler. Single Assignment C’s compiler generates intermediate code in C programming language. This is a for-loop relative to the innermost loop in the original Single Assignment C’s source code and will be processed by my program, i.e. the interface between Single Assignment C and Vector Pascal. Variables in this for-loop have been simplified for convenience of reading

The Vector Pascal code which will be generated by the interface developed by me:

```
for i:= loop_begin to loop_end do
begin
    Array1[i]:= Array2[i]+Array3[i];
end;
```

Program 4.1.2 the corresponding code of Program 4.1.1 in Vector Pascal. This code is generated in the first stage of my program, which is just testing the correctness of the transforming.

The code segments shown in Program 4.1.1 and Program 4.1.2 are the simplest ones to illustrate this transforming in the first stage. The real code will be more complicated in practice and with unreadable formats; we shall see a real example later.

Notice that there is a slight difference between the for-loop in C and Pascal. We can stop the iteration when the variable "i" equals to, or less than the target value, but in Pascal the iteration can only stop at the target value. Therefore, if the operator is "<=", variable "*loop_end*" in the output Pascal file should equal to "*loop_stop*" in C. Otherwise, else if that operator is "<", "*loop_end*" will be equal to "*loop_stop-1*".

This transforming will be based on the traversal over the Concrete Syntax Tree, which is generated by activating the parser to parse the C source file output from Single Assignment C, as shown in Figure 3.3. During this progress, my program will also modify the Syntax Tree to make it more convenient for following works.

4.1.1 Structure of the First Stage

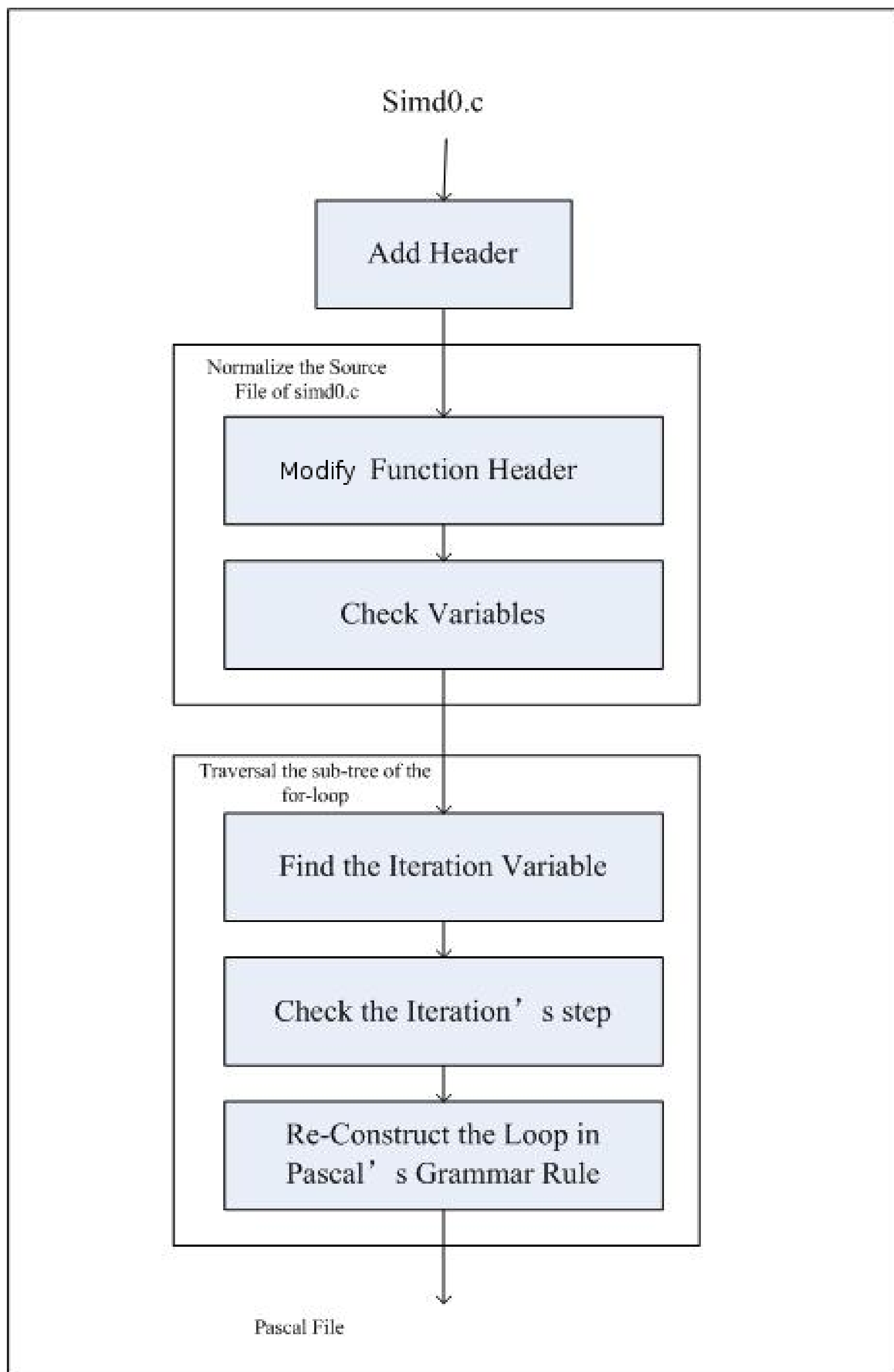


Figure 4.1 Structure of the first stage of the interface. This figure shows the works will be done in the first stage of my stage, including two parts. The first one is normalizing the input C file and the second one is transforming the C file to Vector Pascal code. All the works are based on tree traversal

Figure 4.1 is a schematic view of the first stage's structure. It can be generally divided into two parts. The first one is to normalize the input C file by modifying the headers and checking variables definition. The second one focuses on the transformation of the for-loop. Before these two parts, my program will firstly add a simplest header to the C file to satisfy the tree traversal. I will explain it in the following section with an example.

4.1.2 The Outline of the First Stage

Now let's have a look on an example which can help us to have an intuitive view to the stage, and I will explain the detail of this stage step by step with this example.

4.1.2.1 A Single Assignment C Sample

Let's take a look on the sample code in Program 4.2 now. This program was presented in section 3.1, and I copied it here for convenience of reading.

```
1   use Structures: all;
2   use SimplePrint: all;
3   int main()
4   {
5     v1 = genarray( [100], 2);
6     v2 = genarray( [100], 1);
7     for( i=0; i<10000; i++) {
8       v1 = with(iv)
9             (. <= iv <= .) : v1[iv] + v2[iv];
10            modarray(v1);
11    }
12    res = print( v1[0]);
13    return( res);
14  }
```

Program 4.2 A sample of Single Assignment C source code. This program uses a with-loop to add array v2 to v1, and wraps the with-loop by a normal for-loop to perform the operation for 10,000 times.

This program performs an addition operation on two arrays, v1 and v2. The first two lines are using the libraries of Single Assignment C. Line 5 and Line 6 respectively defines these two arrays which have 100 elements with values 2 and 1 on each position of them. This program also contains one of the typical structures of Single Assignment C's array subsystem, called “with-loop” as the innermost loop, from line 8 to line 10. It is wrapped by a for-loop which is coinciding to the for-loop’s format in C programming language. In this loop, the program adds every corresponding element in area defined by the index vector called “iv” in arrays v1 and v2, returns a modified version of v1, and then prints this result to the standard output of the operating system.

The “iv” is a keyword in Single Assignment C, which stands for index vector. As the name described, it is not a scalar variable but a vector denotes the part of the arrays which will be selected and operated in the with-loop. In program 4-2, Line 9 denotes the addition operation acts on the whole array, since the token “.” indicates the lower and upper boundary of the array. It can also be written as *“([0] <=iv<= [99]): v1 [iv] +v2 [iv];”* which is equivalent.

4.1.2.2 Output of Single Assignment C’s compiler

The code from Single Assignment C as the input will only contain a for-loop, and before go further it is necessary to explain the options being used when invoking its compiler to generate intermediate code. Single Assignment C’s compiler will be invoked by using commands as follows to generate intermediate code which will be port to my program.

sac2c -simd -O3 -dcccall <filename>.sac

Or

sac2c -simd -O3 -dnocleanup <filename>.sac

The “sac2c” is the compiler’s name of Single Assignment C. The “-simd” option here is to notify the compiler to find out the innermost loops in the Single Assignment C’s source code, i.e. the with-loop in this example. Then the compiler will check whether this innermost loop is able to be vectorized. Meanwhile, the compiler will also consider whether this innermost loop worth the operation of

vectorization to get better performance. For example, if we have an even more simple code as the one in Program 4.3, the compiler will port nothing for vectorization because the total amount of operations is too small (10*10 arithmetic operations on arrays' elements), and the cost of function call (vectorized code will be encapsulated into functions, compiled by Vector Pascal's compiler, and finally linked by the GCC linker, thus the main program will invoke function calls to use them) may counteract the benefit from vectorization.

```
1   use Structures: all;
2   use SimplePrint: all;
3   int main()
4   {
5     v1 = genarray( [10], 2);
6     v2 = genarray( [10], 1);
7     for( i=0; i<10; i++) {
8       v1 = with(iv)
9              (. <= iv <= .) : v1[iv] + v2[iv];
10            modarray(v1);
11    }
12    res = print( v1[0]);
13    return( res);
14  }
```

Program 4.3 A Single Assignment C code with pretty few operations. Since the operations which will be performed in this program are very few, Single Assignment C's compiler will not port any intermediate files for using Vector Pascal's compiler as code generator.

Otherwise, if line 12 of the program 4.2 is replaced by:

res = print (v2[0]);

The compiler will also port nothing for vectorization no matter how big the amount of operations on those arrays, because that operations in the loops are only effect on v1, and they will actually be omitted in the progress of optimization.

The "-O3" option let the compiler give out the most expensive and efficient optimizations, in Chapter 5, all benchmark test programs will be compiled with this optimization level. And the "-dcccall" or "-dnocleanup" options will let the compiler keep the files for vectorization, because they are temporary files and are stored under some sub-directories of "/tmp" in Linux Operating

System. Without any of these two options, the compiler will remove those temporary files generated by it after the compiling progress is done.

By using one of those two commands, the Single Assignment C's compiler will generate several C source files. In this example, two C source files are created. One is called "a.out.c". As introduced in Chapter 2, Single Assignment C is using GCC as the backend to generate executable file. This C file is the one which contains the main C program generated from the Single Assignment C's source code. Another one is named as "simd0.c" which is generated for vectorization.

The program presented below is the practical program inside this "simd0.c". We can see that variables' names are hard to read and remember, since their pretty long names contain so many tokens, digitals and characters.

```
1   {
2     int SAC_stop_SACp_pinl_337___nt2ot_21 = 100;
3     int SACp_pinl_337___nt2ot_21;
4     for (SACp_pinl_337___nt2ot_21 = 0; SACp_pinl_337___nt2ot_21 < 100; )
5     {
6       {
7         ;
8       }
9       ;
10      {
11        *SACp_pinl_332__emal_311__pinl_301___flat_139=
            SACp_pinl_330____v1[*SACp_pinl_342___wlidx_290__flat_14] );
12      }
13      ;
14      {
15        *SACp_pinl_332__emal_311__pinl_301___flat_139=
16          ((*SACp_pinl_332__emal_311__pinl_301___flat_139)+ (1));
17      };
18      ;
19      {
20        (SACp_pinl_333__emal_309_v1__SSA0_1[*SACp_pinl_342___wlidx_290__flat_14])=
        *SACp_pinl_332__emal_311__pinl_301___flat_139;
21      };
22      {
23        ;
24      }
25      *SACp_pinl_342___wlidx_290__flat_14=*SACp_pinl_342___wlidx_290__flat_14+1;
26          SACp_pinl_337___nt2ot_21 = SACp_pinl_337___nt2ot_21 + 1;
27    }
28  }
```

Program 4.4 the content of simd0.c. This program is a pure for-loop in C programming language, generated by Single Assignment C's compiler as part of its intermediate code, and it will be transformed to Vector Pascal code by my program.

4.1.2.3 Working Progress of the First Stage of My Program

Program 4.4 shows a for-loop in C which is the content of simd0.c, it is generated by Single Assignment C. As briefly discussed in section 3.2, this program is used by “#include” in the main program in file a.out.c, which is equivalent to copy all content to the main program. But it will be ported to Vector Pascal now, thus it has to be processed as a separate C source program. Therefore, it can not be analyzed directly by the parser in the above form without a function header. So, the first step is to add a simple header in the beginning of the file as follows, in order to make it coincident to the standard C grammar.

void foo( )

With this simple function header, the program fulfills the rules of the parser now, but this is not enough. The first part of this stage of my program has to normalize the function body and modify the header by adding parameter list to it for following reasons.

In program 4.4, there are some variables including array pointers used in the loop body as the global variables in the main program in a.out.c. Still, program 4.4 can be used correctly without any header by the compiler of Single Assignment C, because the last stage of compiling is to compile a.out.c “includes” it which will not meet any obstacle for handling these variables. But the Vector Pascal's compiler will not know the variables’ values. Thus only adding a function’s header can not maintain the correctness of the program. It is necessary to find and recognize the variables and add them to the header's parameters list, so that the main program in a.out.c will be able to pass correct addresses of those arrays to the procedures of Pascal inside the Vector Pascal’s library file. The header will finally be like this:

```
1    void foo (int *SACp_pinl_332__emal_311__pinl_301___flat_139,
2    int *SACp_pinl_342___wlidx_290__flat_14,
3    int *SACp_pinl_330____v1,
4    int *SACp_pinl_333__emal_309_v1__SSA0_1)
```

Program 4.5The function header added to Program 4.4. This function header has a parameter-list for handling the variables passed from the main program.

Meanwhile, during traversal the syntax tree and examining the variables, my program will also check the iteration variables to judge whether the program can be transformed into for-loop in Vector Pascal and be vectorized.

Then the second part of this stage which was shown in Figure 4.1 is activated. The code which will be output after translation is shown as follows:

```
1  library to_vpc_lib;
2  interface
3  TYPE INTARRAY=ARRAY[0..99] OF INTEGER;
4  procedure to_vpc ( VAR SACp_pinl_337__nt2ot_21:INTEGER; VAR SACp_pinl_342___wlidx_290__flat_14:INTEGER;
    VAR SACp_pinl_332__emal_311__pinl_301___flat_139:INTEGER;VAR SACp_pinl_333__emal_309_v1__SSA0_1:INTARRAY);
5  implementation
6  procedure to_vpc ( VAR SACp_pinl_337__nt2ot_21:INTEGER; VAR SACp_pinl_342___wlidx_290__flat_14:INTEGER;
    VAR SACp_pinl_332__emal_311__pinl_301___flat_139:INTEGER;VAR SACp_pinl_333__emal_309_v1__SSA0_1:INTARRAY);
7  VAR SAC_stop_SACp_pinl_175__nt2ot_20 :INTEGER;
8  begin
9    SAC_stop_SACp_pinl_175__nt2ot_20 :=100 ;
10   for SACp_pinl_175__nt2ot_20 :=0  to  100 -1 do
11   begin
12     SACp_pinl_332__emal_311__pinl_301___flat_139 := SACp_pinl_330____v1[SACp_pinl_342___wlidx_290__flat_14];
13     SACp_pinl_332__emal_311__pinl_301___flat_139 := SACp_pinl_332__emal_311__pinl_301___flat_139 +  1;
14     SACp_pinl_333__emal_309_v1__SSA0_1[SACp_pinl_342___wlidx_290__flat_14] :=
         SACp_pinl_332__emal_311__pinl_301___flat_139;
15     SACp_pinl_342___wlidx_290__flat_14 := SACp_pinl_342___wlidx_290__flat_14+1;
16  end;
17  end;
18  begin
19  end.
```

Program 4.6 output of the first stage. This is a for-loop in Vector Pascal, transformed from the corresponding one in C, which is shown in Program 4.4

We can see that there is still a for-loop in this program and the program is just modified to coincide with Vector Pascal's grammar. From line 12 to line 15, the operations in program 4.4 have all been reserved here. This is only normal Pascal code without any vectorization, but it is still significant because it provides some basement for the following works and it validates the correctness of the combination of Single Assignment C and Vector Pascal.

4.2 Vectorized the Pascal Code and Unroll the For-Loop

4.2.1 Why vectorization and Loop Unrolling?

In the first stage, we have already gotten the normal Pascal code which can be compiled by Vector Pascal's compiler, and the function call has been created & adjusted so that the GCC compiler can correctly invoke the function encapsulated in a Pascal library. In this stage, the function's body will be changed, i.e. the code performs operations on arrays which will be vectorized and the for-loop is unrolled into straight code.

Vectorization is the process of converting a program using scalar data which performs an operation on a pair of operands at a time, to a vectorized program in which a single instruction can perform multiple operations on a pair of vector operands which contain series of adjacent values. This technology is very useful in applications doing array operations since the elements of the arrays can be considered as adjacent values. Suppose that the processor can perform operations on four operands at a time, we may write a program performs addition between two arrays of numeric data as follows in a traditional way.

```
1 for (i = 0; i < 1024; i++)
2 {
3     C[i] = A[i]+B[i];
4 }
```

Program 4.7.1 a sequential for-loop in C. This is a for-loop performs addition operations on arrays A and B, then store the result to C. The loop will iterate for 1,024 times.

Or write a vectorized code:

```
1  for (i = 0; i < 1024; i+=4)
2 {
3     C[i:i+3] = A[i:i+3]+B[i:i+3];
4 }
```

Program 4.7.2 a vectorized for-loop. This loop is written in pseudo code similar to C, to illustrate the vectorization in this situation.

Comparing the two programs shown above, the first one is a code segment of normal C program which adds elements of arrays A and B together and assignment the result to array C. It is a sequent program and in each iteration only one pairs of the corresponding elements in the arrays A and B will be added and stored in C. The second one is a vectorized program written in pseudo code similar to C, just to show what vectorization is. "C [i: i+3]" represents the four array elements from position "i" to "i+3" in array C. Since the execution time of four operations performed by one instruction is approximately same to the time taken for one scalar instruction, the vector code saves 75% time cost on processing data and it can run at least four times faster than the original code theoretically.

The vectorization is usually implemented in a compiler approach which is based on loop-unrolling. Meanwhile, loop-unrolling also bring other benefits.

By unrolling the loop into straight code, the processor does not need to judge and predict the branch at the beginning of each time of the for-loop iteration, the comparison and the self-increment/ decrement of the loop's variable is also saved.

Another important thing beyond these reductions is that the straight code does not suffer from the potential hazard of branch prediction failure, which may cause the pipeline stalls. In modern general processors, especially those x86 architecture ones, the pipelines are very long compared with RISC architecture processors. For example, Pentium III has 10 stages pipeline while this number is increased to 20 in Pentium IV [Intel, 2001]. Even in Intel's newest multi-core processors, this length is shortened to 14 [Lowney, 2006], it is still pretty long compared to those typical RISC processors such as MIPS series which normally have less than ten stages. Thus one failure of branch prediction will waste more than ten clock cycles.

Count all these factors together, including vectorization and loop unrolling, we can expect a performance enhancement on the arrays operation part theoretically more than 400% when using modern x86 processors which provide 128-bit registers for exploiting SIMD parallelism to deal with 32-bit integers (and 200% enhancement when processing 64-bit floating point numbers). Consider that there are also some other parts of calculations & operations in the program, the enhancement may possibly be less than this theoretical value, but it still shall be notable when the task's scale is big enough, i.e. the arrays' size or loops' size is big enough.

4.2.2 Goal of Vectorization

The following code samples show this change briefly:

```
for i:=0 to 999 Do
begin
        j:=k;
        tmp:=(v1[j]);
        tmp:=tmp+v2[k];
        v3[k]:=tmp;
        k:=k+1;
end;
```

Program 4.8.1 A sequential for-loop in Pascal. This is a typical program generated in the first stage of my program, and its variables' names have been simplified.

```
begin
        v3[k..(k+999)]:=v1[k..(k+999)]+v2[k..(k+999)];
        i:=1000;
        j:=999;
        k:=1000;
end;
```

Program 4.8.2 Corresponding code after vectorization. This program is a vectorized version of Program 4.8.1. The expression of operations on arrays' elements has been vectorized to those operations on whole arrays. And some variables have been eliminated since they are only local variables which are used to pass values between statements in the scope of the loop body.

Program 4.8.1 is an example of the simplified code (variables' names are renamed for convenience of reading) generated in the first stage of my program, which is a normal Pascal for-loop. Program 4.8.2 is the goal of vectorization with loop unrolling in the second stage of my program. As mentioned in section 3.3, the second stage is not totally based on the first stage, because this task can be done based on traversal over existing C Syntax Tree, and it might be a kind of waste of the resources to generate another Pascal Syntax Tree.

Comparing Program 4.8.1 and 4.8.2, we can see that the local variable "*tmp*" is eliminated in Program 4.8.2 because it is only a temporary intermediate variable which is used to pass value. The 1000-times-cycling for-loop is unrolled to a straight operation on arrays which is acting on whole arrays at one time. What about variable *"i", "j", "k"*? They are set to the final value of them when the original for-loop is finished to guarantee the correctness of the program, because there is no evidence to judge whether their value will be used again after the original for-loop or not, based on the original standard C code ported from Single Assignment C's compiler.

Vector Pascal implements the concept of operations on arrays but it can not automatically vectorize sequential code. Therefore, a programmer shall not use loops to process these operations in practical, except those situations in which the program delivers operations which are not able to be vectorized, such as dealing with discrete elements of an array. Similarly, because Vector Pascal doesn't provide the function of automatic vectorization, simply translating those C code generated by Single Assignment C into normal Pascal will not enjoy the enhancement brought by Vector Pascal's SIMD features. As a result, my program, the interface between these two programming languages, will also try to generate vectorized code in Vector Pascal.

4.2.3 Vectorization & Loop Unrolling

Before implementing vectorization and loop unrolling, there are some questions need to be answered. What kind of for-loop can be unrolled and vectorized? How to achieve this goal? When should it be done?

Firstly, let's have a look on what may cause difficulties for vectorization and loop unrolling. As mentioned in Chapter 2, there are some obstacles which may prevent vectorization on loops, such as function call, control dependences, and data dependences. The files ported from Single Assignment C are not hand written but generated by its compiler, and they only include the innermost loops which are chosen to be vectorized. Thus there is no function calls nor control dependences inside but only data dependences may exist.

4.2.3.1 Data Dependencies

Data dependence is a kind of read-and-write confliction, arises from two statements which access the same resource and at least one of the accesses is a write. It's also known as a data hazard. There are two common types of data dependences, one is the dependences between statements in the same loop-iteration, and another is known as loop-carried dependences.

```
1. A=3;
2. B=A;
3. C=B;
```

Program 4.9 dependences between lines. The dependences in this code segment are true dependences.

The first type is like the code shown above in Program 4.9, which is also called as true dependence because that one statement must wait for a previous one's result. It is easy to eliminate this kind of dependences and unroll the loop. But the second type, loop-carried dependence is a little bit complicate since it has two different cases. Let's have a look on two sample code segment as fellows:

```
for(u=0;u<array_size;u++)
{
    A[u+1] = A[u]+C[u];
}
```

Program 4.10.1 a loop carried dependences which can not be eliminated. This is the first type of the common loop-carried dependences.

```
for(u=0;u<array_size;u++)
{
    A[u]   = A[u]+B[u];
    B[u+1] = C[u]+D[u];
}
```

Program 4.10.2 a loop carried dependences which is able to be eliminated. This is the second type of the common loop-carried dependences.

We would like to run all iterations of each loops in parallel, but in the first one the value being read in each iteration was written in the previous iteration. It is not possible to execute this recurrence in parallel and still guarantee the sequential semantics. But the second program can be vectorized because it can be sliced into two independent loops. As the C source files which contain innermost for-loops are generated by the Single Assignment C's compiler after their basic analysis, they are all able to be vectorized, i.e. they may only contain the first type of dependences.

4.2.3.2 Data flow Optimization

Before vectorization, several procedures might be done because the original source code is not perfectly optimized, and they may also cause some difficulties for the following work.

◆ Constant Propagation & Folding

Constant Propagation means to replace instances of variables whose values are known at compile time with said values. Constant folding goes one step further based on constant propagation, which replaces expressions whose values are known at compile time with their results.

```
1.  int i, a[32];
2.  int j=32;
3.  int c=3;
4.  c=c+j;
5.  for(i=0; i<j; i++)
6.  {
7.    a[i] = a[i] + c;
8.  }
```

Program 4.11.1 a segment of program before constant propagation & folding

```
1.  int i, a[32];
2.  for(i=0; i<32; i++)
3.  {
4.    a[i] = a[i] + 35;
5.  }
```

Program 4.11.2 a segment of program after constant propagation & folding

We can see that in the code segment after constant propagation & folding, in line 2, variable j is replaced by its value “32”, and in line 4, the variable c is replaced by the result of expression in original code line 4. Then statements from line 2 to 4 are removed.

- Copy Propagation

Copy Propagation is a way of high level optimization which replaces the copies of a variable with its original name and eliminates redundant copies. For example:

```
int main(int i, int j)
{
    int t,s,a[32];
    t=i*3;
    s=t;
    a[t]=a[t]+j;
    ......
}
```

Program 4.12.1 original code, before copy propagation

```
int main (int i, int j)
{
    int a[32];
    a[i*3]=a[i+3]+j;
    ......
}
```

Program 4.12.2 the result of copy propagation

Constant propagation & folding and copy propagation are useful “clean up” optimizations before following works. In some sense, they reduce the complication of processing dependences eliminating and they are very helpful method to enhance the final performance because lots of works are done in compile-time.

4.2.3.3 Forward Substitution

Now here comes the questions, how and when to do vectorization & loop unrolling? In my program's system, optimizations mentioned in 4.2.3.2 will be automatically done as a "side effect" of vectorization when using context substitution. Considering that my Syntax Tree traversal is based on depth-first algorithm, I picked forward substitution.

Forward substitution is a concept which vividly describes the operations on context here, i.e. a kind of substitution using previous text to replace current one and step over. It is similar to the one with the same name in triangular system such as Gaussian elimination. To explain this concept itself and the mechanism to implement it, let's have a look on a sample code first. Suppose we have a Single Assignment C's source code like this:

```
1   use Structures:all;
2   use SimplePrint:all;
3   int main()
4   {
5       v1=genarray([1000],1);
6       v2=genarray([1000],2);
7       v3=genarray([1000],3);
8
9       for(i=0;i<1000;i++)
10      {
11          v3=with (iv)
12          (.=<iv<=.) : v1[iv]+v2[iv];
13          modarray(v3);
14
15      }
16      res=print(v3);
17      return(res);
18  }
```

Program 4.13 a Single Assignment C's source code with addition operation between two arrays

This program performs the operations of adding two arrays of integers together and storing the result to another for one thousand times. The compiler will generate a C file named "simd0.c" relative to the addition operation inside with-loop as follows:

```
{
  for(i =0; i<1000;)
  {
        j=k;
        tmp=(v1[j]);
        tmp=tmp+v2[j];
        v3[j]=tmp;
        k=k+1;
        i=i+1;
   }
 }
```

Program 4.14 a simplified version of simd0.c generated by Single Assignment C's compiler. This for-loop in C is the intermediate code which is relative to the innermost loop between line 11 and line 13 in the original Single Assignment C's program, i.e. Program 4.13

This is a typical simplified "simd" file generated by Single Assignment C's compiler. The only difference of this sample from the original one is that all variables' names are simplified for convenience of reading, because as mentioned previously the original one is pretty long and complicated which is composed by various kinds of tokens, digitals and characters.

Suppose that "*k*" is a variable passed from other scope outside of this for-loop, then use it to replace every "*j*" after expression "*j = k;*". Next is "*tmp*", which is finally transformed into "*tmp=v1[k]+v2[k];*". The last substitution is use this expression to replace the variable "*tmp*" in "v3[j] = tmp;".

Thus after forward substitution, this for-loop turns into:

```
{
   for (i=0;i<1000;)
   {
       j=k;
       tmp=(v1[k]);
       tmp= v1[k]+v2[k];
       v3[k]=v1[k]+v2[k];
       k=k+1;
       i=i+1;
    }
 }
```

Program 4.15 code after forward substitution. By utilizing forward substitution, Program 4.14 has been simplified and transformed into this one.

Since variable "*j*", "*tmp*" are no longer necessary; they will also be eliminated in final result:

```
{
    for (i=0;i<1000;)
    {
        v3[k]=v1[k]+v2[k];
        k=k+1;
        i=i+1;
     }
 }
```

Program 4.16 code after eliminating redundant expressions. Some expressions which are expired in program 4.15 have been eliminated in this program.

After forward substitution, we have transformed the source program from what is shown in Program 4.14 to that in Program 4.16. Based on its reshaped Syntax Tree, we can do the vectorization now.

4.2.3.4 Loop unrolling and Vectorization

The first step of vectorization is to find out the arrays' index variables. Consider a code segment as follows:

```
{
 iv=iv0;
 for(i=begin_val;i<end_val;i++)
 {
   v3[iv]=v1[iv]+v2[iv];
   iv=iv+1;
   i=i+1;
  }
}
```

Program 4.17 a program which will be vectorized. The arrays' indices will be substituted in this program during vectorization.

If we look into this sample, we will see that "*iv*" is self-increasing by step of 1 in each time of iteration. In other word, it goes along with iteration variable "*i*". Actually, no matter there are how many index variables, all of them will act in same way. Otherwise, for example, if one of the index variables steps over by 2 every time of iteration, the operations on that array will not be able to be vectorized. During checking the variables and normalizing the for-loop's body, the first stage of my

program has already checked it. If my program recognizes these kinds of problem, it will turn them into another class, i.e. they will be transformed into while-loop and will not be vectorized.

So, the index variable actually equals to its initial value before the iteration begins pluses the difference between the starting value and the final value of the iteration variable. In this sample, it is: "*iv=iv0+(i-begin_val);*", which means that the elements from position "iv0" to position "(iv0 + (end_val - 1) - begin_val))" are operated inside the for-loop. The reason of using "(end_val - 1)" but not "end_val" here is that the loop will be stopped when "i" equals to "end_val" and the elements on position "end_val" of the three arrays will not be operated.

Thus my program will substitute all index variables in those expressions which represent operations on arrays by a statement as:

iv0 .. (iv0 + (end_val - 1) - begin_val)

Then set iv to its correct final value: "iv0 + (end_val - begin_val)", and also set i to "end_val" to guarantee the correctness of the whole program, because this for-loop must be an innermost loop if there are multiple level of loops and these variables may also be used outside of this loop.

Finally, by removing the for-loop's loop header, i.e. "*for (i=begin_val;i<end_val;i++)*" and translating the code into Pascal's code, vectorization and loop unrolling is finished. We have the final code here:

```
begin
   v3[iv0 .. (iv0 + (end_val - 1) - begin_val)]:= v1[iv0 .. (iv0 + (en
d_val - 1) - begin_val)]+v2[iv0 .. (iv0 + (end_val - 1) - begin_val)];
   iv:=iv0+ end_val - begin_val;
   i:=end_val;
end;
```

Program 4.18 The result of vectorization. After vectorization, the for-loop's body in Program 4.17 has been transformed and vectorized into this Vector Pascal code by my program (the second stage)

When compiled by Vector Pascal, the compiler will process v1, v2, v3 as three vectors by using SIMD instructions of the processors and the performance will be enhanced comparing to the non-vectorization version which was produced in the first stage of my program.

4.3 Further Optimization

4.3.1 A Weird Phenomenon

In the previous example mentioned in section 4.2.3.3, the program performs an addition operation on two arrays of integers and a store operation to the third one. Since most modern general processors provide 128-bits registers for implementing SIMD operations and an integer only takes 32-bits, the program's performance shall be 400% of the original one in theoretical. This sample is a very simple one and the operations on arrays which can be processed by using the help from SIMD instructions take very high percentage compared to the total number of operations. Thus we may assume that the performance should be quite near to the theoretical one. But on the contrary, it is only a slight speed-up of this example in practical.

By looking into the compiler's mechanism, we shall find that this phenomenon is reasonable. The Vector Pascal's compiler will create a new descriptor for those arrays which have indices. With the new descriptors, there will be several more operations in the library file generated by Vector Pascal's compiler, to calculate the correct address of each element in the arrays; as a result the total amount of the operations will be increased to several times bigger.

Consequently, there shall be some further enhancement if we can prevent the compiler creating new descriptors for arrays by removing the indices of them to generate programs like what is shown in Program 4.19 as follows:

```
begin
    v3 := v1 + v2;
    iv: = iv0 + end_val - begin_val;
    i:  = end_val;
end;
```

Program 4.19 vectorized Pascal code in which arrays are presented without index variables. This is the goal of the third stage of my program, to generate code which can utilize Vector Pascal's compiler's features better. When compiling code with arrays in this form, the compiler does not need to create new descriptors and this will save a lot of work spent on calculating addresses.

4.3.2 Tasks in this Step of Optimization

To achieve the goal of this optimization, there are several other things need to be done to guarantee the correctness of the program beyond simply removing the index variables.

In the second stage, the main program in file a.out.c will invoke the function-calls by passing the first elements' addresses of the arrays; and then the scopes of the part which will be operated are designated by the index variables passed inside and the loop's iteration times. For example:

v3[iv0 .. (iv0 + (end_val - 1) - begin_val)]

No matter whether the variable "iv0" begins from zero or not, the program is guaranteed to operate on the correct elements. But without the index variables, the program will automatically operate on the whole "v3" if "iv0" does not equal to zero and "((end_val - 1) - begin_val)" does not equal to the array's size.

The solution is to check the scope of the arrays which will be operated, and let the function's caller, i.e. the main program of a.out.c file pass the correct address of those arrays with the proper modified beginning address. Yet only modifying the list of parameters is not enough.

As mentioned in previous chapter, the Vector Pascal source program has several procedures relative to the SIMD files from Single Assignment C, and will be encapsulated into a library. Thus in the Vector Pascal file, I have to declare some arrays in the procedures' parameter lists to receive those arrays passed into it. Let's take a look on a new sample now. It is only slightly different from the one shown in Program 4.13.

```
1  use Structures:all;
2  use SimplePrint:all;
3  int main()
4  {
5      v1=genarray([1000],1);
6      v2=genarray([1000],2);
7      v3=genarray([1000],3);
8
9      for(i=0;i<1000;i++)
10     {
11         v3=with (iv)
12         ([4]<iv<[16]) : v1[iv]+v2[iv];
13         modarray(v3);
14
15     }
16     res=print(v3);
17     return(res);
18 }
```

Program 4.20 a Single Assignment C source code. This program performs array operations on part of the arrays instead of on the whole arrays.

Then it will have a relative simd0.c file generated by the compiler. For convenient, I have simplified it as fellows:

```
{
 for (i=5;i<16;)
 {
   v3[k]= v1[k] + v2[k];
   i++;
   k++;
  }
}
```

Program 4.21 simplified version of the content of the simd0.c. This program contains the for-loop which is corresponding to the with-loop in Program 4.20.

The only difference between of this one from Program 4.13 is that the addition operation performs only on part of those arrays instead of which performs on whole arrays. It will finally be transformed into code like this:

```
library test;
interface
TYPE INTARRAY=ARRAY [0..10] OF INTEGER;
procedure foo(var v1,v2,v3:INTARRAY;var i,k:INTEGER);
implementation
procedure foo(var v1,v2,v3:INTARRAY;var i,k:INTEGER);
begin
        v3:=v1 + v2;
        i:=16;
        k:=k+11;
end;
begin
end.
```

Program 4.22 Vectorized code in Vector Pascal. This program shows the output of the third stage of my program, which is manipulating arrays without index variables.

We can see the two modifications in Program 4.22. One is that the "INTARRAY" is defined as an array with 11 elements instead of which has the same size of the whole arrays passed into this procedures in stage 2, because only 11elements of array v1 and v2 will be added together. Another modification is that the arrays index variables are removed.

This will be compiled by Vector Pascal and be called by a.out.c. Thus in order to pass correct address, the function call in a.out.c will be transformed from:

foo(&v1, &v2,&v3, &i, &k)

to

foo(&(v1[k]), &(v2[k]), &(v3[k]), &i, &k)

4.4 Benchmark System

Since the source program of simd0.c file is able to be transformed into what I expect in the third stage of my program, it is the time to examine the enhancement of the performance now, to see whether my hypothesis in the very beginning of this project is correct.

One of the most common ways to test a program's performance is to calculate the cost of the time by running it. For example, in Linux environment, we may use "time" command to see the result. The result is perspicuous, but not very accurate because it is only provides precision. Moreover, other program's process will also affect this result because the modern operating system is using time slicing method to implement multiple tasks.

However, another solution will be able to solve these two problems. Because all works is currently done on x86 architecture platform, I looked into some documents from Intel Corporation about this architecture, and luckily I found that all main stream general processors from both Intel and AMD are providing an instruction called RDTSC for accurate timing.

Beginning with the Pentium processor, Intel brought a time-stamp counter into their processor family. It keeps an accurate count of every cycle which "occurs on the processor." [Intel Corp, 2001]. The RDTSC instruction will return the information of this counter to the caller, which can be a kind of precious calculation of the program's performance. This mechanism, including the time-stamp counter and the RDTSC instruction, also presents in current AMD processors. Thus it is guaranteed to run the benchmark based on this mechanism on both Intel & AMD platform.

Before using it in practice, someone may argue that modern processors usually adopt out-of-order Execution. This seems to be a potential hazard because there is no guarantee about whether the RDTSC instruction executes at its original position in the assemble language file generated by the compiler, i.e. the processors may execute them in some different order rather than what we expect. Indeed this was a big problem in processors of several generations before. At that time, programmer had to tune their program very carefully with inserting CUPID instruction into the assemble language file to force the processors execute following instructions in-order. In the following sample, we can see that the CPUID instruction appears before using RDTSC instruction in the very

beginning to clear all previous tasks. And it also appears before the next time of using RDTSC instruction to force the instructions executed before it. Thus the difference between the results of this two times of using RDTSC instruction returns the clock-cycle of those instructions between them properly.

```
cpuid                    ; force all previous instructions to comple
rdtsc                    ; read time stamp counter
mov     time, eax        ; move counter into variable
fdiv                     ; floating-point divide
cpuid                    ; wait for FDIV to complete before RDTSC
rdtsc                    ; read time stamp counter
sub     eax, time        ; find the difference
```

Program 4.23 an assembly program's segment using CPUID and RDTSC instruction for precise timing

But since the advent of MMX Technology, which is also the first generation of the SIMD instructions set on x86 architecture platforms, the CPUID instruction is not necessarily for serialization. [Intel Corp, 2001]

One thing need to notice is that, due to the high frequency level of the modern processors, the result stored in the time-stamp counter is normally very big. This counter can not be cleared to zero after the computer starts up and that's why we use the difference between twice using RDTSC instructions. Therefore, to restore this very big value and to avoid the wrong result of the subtraction, we shall use 64-bit integer, because an overflow is very possible if we only use the low 32-bit of that counter's value.

The example code of using this will be given in next chapter, i.e. Chapter 5 Benchmark Testing and Anglicizing.

4.5 Key Notes of the Implementation

In this section, I will introduce some important parts on the implementation of the program, including building the syntax tree and some detail things of the three stages of my program.

4.5.1 Building Syntax Tree

Since most of the works have been done based on traversal over the syntax tree, building the syntax tree is the foundation of the whole system of my program. There are several tools which can help developer build their own defined syntax tree, and I choose SableCC to implement the parser for generating syntax tree.

4.5.1.1 SableCC

SableCC is “an object-oriented framework that generates compilers and/or interpreters” [Gagnon, 1998] which is implemented in Java programming language. It has two important features. Firstly, this framework is implemented by using java programming language, which is an object-oriented programming language and it can build strictly-typed Syntax Tree (Abstract or Concrete, due to the “specification file” which defines the actions of the parser). Secondly, SableCC generates “tree-walker classes”, i.e. a means of traversal over the syntax tree, using an extended version of the “visitor design pattern” [Gagnon, 1998] to enable the implementation of actions on the nodes of the syntax tree using inherit mechanism which is a famous highlight of object-oriented concept. With these two features, SableCC can help people develop compilers more quickly and convenient.

Features of SableCC

SableCC extends Backus-Naur Form grammar syntax, i.e. it supports the '*', '?' and '+' operators. It can automatically generate a strictly-typed syntax tree and tree-walker classes which can be inherited

by the working classes. The old version of SableCC can only generate Concrete Syntax Tree, while the newer version also supports generating Abstract Syntax Tree by inputting a proper grammar which contains the Abstract Syntax Tree production definitions and also provides way of transferring from Concrete Syntax Tree to Abstract Syntax Tree.

When this project was begun, the support of transforming from Concrete Syntax Tree to Abstract Syntax Tree was too new and there wasn't a stable version of SableCC at that time which implements this design. Therefore, I decided to build Concrete Syntax Tree.

Generating a Parser with the Help of SableCC

As a "compiler compiler", which means it is able to “compile” some files to generate a compiler, SableCC can generate either a Concrete Syntax Tree or an Abstract Syntax Tree (only supported since version 3.0) based on the aiming programming language's grammar. Grammars for SableCC sometimes also need to be modified to suit different version of the SableCC, even just dealing with the same programming language.

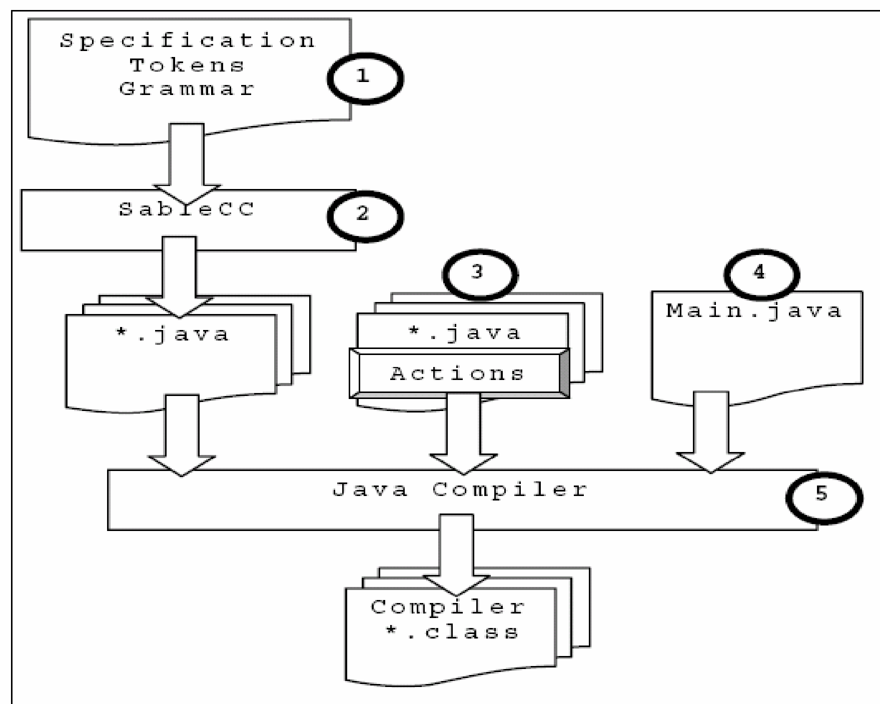


Figure 4.2 the working progress of SableCC [Gagnon, 1998]. This figure shows the five steps of utilizing SableCC and specification file which contains proper grammar to build parser.

Producing a compiler by using SableCC normally requires five steps, which are shown in Figure 4.2.The first step is to create a specification file for SableCC to let it know how to generate the Syntax Tree. This specification file must at least contain the lexical definitions and the grammar of the programming language to be compiled. Some other information, such as project's repository or definition of the rules for transforming from Concrete Syntax Tree to Abstract Syntax Tree, is also possibly added. In the second step we can launch SableCC on the specification file to generate a framework, which contains the Syntax Tree's structure and an abstract Depth-First Tree Walker.

"Abstract" here means that this Tree Walker can do nothing except the traversal over the Syntax Tree. Therefore, in the third step we have to extend it to add some practical utilities to it. Sometimes, only traversal the Syntax Tree is not enough, thus we may also generate more working classes. ("class" means a kind of encapsulation in object oriented programming language, such as Java which is used in SableCC). Some of these classes are possibly inheriting from classes generated by SableCC.

The forth step is to create a main compiler class that activates lexer, parser and working classes, i.e. a program provides the entry of the whole system. Finally, we can use a Java compiler to compile these classes in the system to generate the target executable compiler.

4.5.1.2 "Specification File" to Activate SableCC

The Structure of the Specification File and Applicability to the Project

The "specification file" of SableCC is not a file defining the features or requirements or something else of it, but one which provides everything needed to activate SableCC to generate a syntax tree and a "Tree Walker" hierarchy. The "specification file" is a text file that contains the project/author information (optional), the lexical definitions and the grammar productions of the language to be recognized by SableCC. It should also provide the directory of the destination as the root of Java package for storing generated files. The lexical definitions use regular expressions and the grammar should be written in Backus-Naur Form (BNF).

For compatible reason as mentioned in last section, I decided to use SableCC to build a Concrete Syntax Tree by writing a suitable grammar based on ISO/IEC 9899:1999 Standard [ISO, 1999]. I didn't change it to work on Abstract Syntax Tree when a stable version of SableCC which supported the AST came out because it is pretty complex to build a grammar manually based on the existing one for generating Abstract Syntax Tree, and almost all work which had been done with Concrete Syntax Tree could not be reused.

The grammar part does not include all features of the standard C because some components will not be used in the out put of Single Assignment C. In other words, the grammar is for a language which is the subset of standard C.

The specification file is divided into several parts as following:

<grammar>

[<package declaration>] [<helper declarations>]

[<states declarations>] [<token declarations>]

[<ignored tokes>] [<productions>]

The <package declaration> section is used to name the destination root, i.e. the directory where all the generated files will be placed.

In the section of <helper declarations>, a lot of "constants" are defined. As this section's name implies, the "constants" in the section are used as helpers of things in later sections. They are used for convenient to present some very common things in regular expression, such as numbers, characters, and the combinatorial logic sets of them by defining them as helper, just like what we used to do on those pre-defined things in C. The helpers are quite similar in many languages.

The section of <state declarations> does not exist in my specification file because it is useless in this case and also useless in most of the cases.

In the following part, <token declarations>, we define the terminals and tokens. The tokens obey a rule which is quite similar with the corresponding set in the Regular Expression. A token defined by a string of characters is declared between quotes, e.g. program -'program', and every declaration ends with a semicolon. Some tokens might be composed of other tokens.

The next section is <ignored tokens> in which we declare the tokens to be ignored by the parser. For example, comments, blank space, carriage return, etc. This part is almost the same in most programming languages.

Finally, there is the last section, the <productions> section in which we declare the productions of the grammar for the language. The productions are defined in BNF or EBNF (Extended Backus-Naur Form). It is quite similar to the context-free grammars only except that we do not declare a non-terminal surrounded by '<' and '>', and the '->' is replaced by '='. Furthermore, we precede each alternative of a production with a name between curly braces as SableCC's demanding. This name serves to identify a specific alternative of a production.

Productions of the grammar

This is the chief and most complex section of the specification file. It describes the language's syntax characteristics, and the syntax tree's casting hierarchy is also defined here.

This section is composed of three parts: Expressions, Statements, and Declarations. All declarations and most of the expressions in standard C grammar are included in my grammar. In the statements part, Labeled statements and Jump statements are not necessary here in my program. Besides, external definitions, pre-processing directives and function definitions / declarator are not supported because that they won't appear in the source code which need to be processed.

The following works are based on the Concrete Syntax Tree, so the Abstract Syntax section is not needed in the specification file.

4.5.1.3 Generating the Classes by using SableCC

During the first and second step of Figure 4.2, SableCC generates files by processing the specification file. The output files from SableCC are classified and stored in four packages: "analysis", "lexer", "node" and "parser". Each file contains either a class or an interface.

With the specification file, an LALR (1) parser [Aho, 1988] will be generated and packaged in the class "root.parser.Parser" successfully by running SableCC on the Java Virtual Machine, unless there is some LALR (1) conflictions which are led by some errors in the <productions> section of the specification file. If an error occurred, the error message will show the nature of the conflictions such as shift or reduction (two common errors), the look ahead token and the complete set of LR (1) items. In the class name, "root" is the directory that we defined in the section of <package declaration>. The parser generator uses token declarations and production declarations to resolve the type of terminals and not-terminals appearing on the right-hand side of the productions.

In the directory “node” we can find the nodes of the Concrete Syntax Tree packaged in java classes with proper names. Each class of production's name is prefixing with an uppercase 'P', the first letter is replaced with an uppercase of itself, and the letter leading by an underscore is replaced with an uppercase of itself. All underscores are removed at the same time. While processing the alternatives, it is quite similar with processing the productions. The only difference is just to prefix the name with an uppercase 'A' instead of 'P'.

The parser class builds the Concrete Syntax Tree while parsing by shifting the tokens received from the lexer on the parse stack. The parser creates a new instance for every reduced alternatives, pops the elements off the parse stack and attaches them to the new instance, then pushes the instance back onto the parse stack. [Gagnon, 1998]

When the specification file is created, we can use SableCC based on Java Runtime Environment to generate the parser for building Concrete Syntax Tree.

4.5.2 The first stage

Before activating the working classes to traversal the syntax tree, the simd0.c file as the input of the parser needs to be modified. Let's recall what we have discussed in section 4.1, simd0.c only contains a pure for-loop, and the parser will complain that it does not coincide with the C grammar. Thus my program will firstly add a simple function header without parameters list to it.

4.5.2.1 Generating the Working Classes

In the Concrete Syntax Tree, some nodes are just for tree mapping and helping to compose the cast hierarchy of the tree. These nodes are "redundant" in some sense, i.e. they shall be left alone during the translation from C code (generated by Single Assignment C's compiler) to Pascal code (to be compiled by Vector Pascal's compiler). As a result, I must check out which nodes in the Concrete Syntax Tree are exactly the target nodes which need to be operated on.

In order to find the appropriate nodes and sub-trees, some Tree Walkers extending the one provided by SableCC are needed to implement the traversal on the whole Concrete Syntax Tree or some sub-trees, and to trace the nodes' content depending on the changes that we want to do on them. By extending the DepthFirstAdapter class which is generated by SableCC, I wrote several walkers to implement traversals over the whole Concrete Syntax Tree and some sub-trees of it.

4.5.2.2 The traversal classes

All the translation works are based on operations on the nodes during traversal on the Concrete Syntax Tree. These operations are defined and implemented by the traversal classes working on the whole tree or sub-trees which extend the DepthFirstAdapter class. Moreover, operations are implemented by overloading the necessary methods which are inherited from the DepthFirstAdapter class and stand for their relative nodes.

Though I also need to design some new methods to implement some works, most of the translating work can be done by overloading the "out method" of the nodes while traversing the tree. The difficulty in this phrase is how to find the right nodes, because in the Concrete Syntax Tree there are a lot of castings from one production to another which form the multi-level tree structure. Though it is obviously that most of the target nodes should be in the lowest level of the tree, i.e. they are leaves, sometimes I still need to trace the depth first traversal and watch the contents of the nodes to adjudge which are the exactly nodes of the alternatives that need to be operated on because that they hide behind the complex casting relationship.

Another problem is that the sequences of the components are different in C and Pascal, so that the translation shall be based on the transform of sub-trees. To adjust some sub-trees' structures during translating, multi-passes of traversal on the whole tree or on the sub-trees are demanded.

Translating Variable Declarations

My program needs to find out the declarations and definitions of the variables in the C format file first, because the global variables should be treated as passed in as parameters so that the program needs to know them during modifying the headers, and they will be declared in the first section of the code segment (only after some universal code such as "procedure" and user defined types, if exists) in the output file. The code from Single Assignment C should be firstly transformed into a function which contains a for-loop with array operations and also some variables declaration (possibly). Thus all undeclared variables should all be considered as global variables which are passed from the caller file, a.out.c. To achieve this target, my program needs to traverse the whole tree to find them and then do the transformation.

During the traversal on the Concrete Syntax Tree, a symbol table will record all the variables which are not declared inside the function’s body. Once the traversal is done, this symbol table contains all the variables passed into this function, and then my program will adjust the format of them and output them as the parameters list of the function’s header.

Check the Iteration Expression of the For-Loop.

A for-loop in C programming language has the format as below:

for (clause-1; expression-2; expression-3)

{ statement}

In the line above, clause-1 is either a declaration of expression which denotes the variable used as the controller of the loop. The expression expression-2 is the controlling expression which indicates the stop condition of the for-loop and is evaluated before each execution of the loop body. The expression expression-3 is some operation on the variable given in clause-1.

The for-loop has several different formats. Sometimes expression-3 does not appear together with the expression-2 here but in the body of the loop, i.e. in "statement". In this case, there must be a traversal on the sub-tree of the for-loop to find it out, in order to rearrange the sequence.

4.5.2.3 Other Working Classes

By the consideration of safety and the robustness, I packaged the translations into several classes as follows:

- The transformation of for-loop
- The transformation of variables' declaration
- Iteration finding
- Up/down judgement
- Class for replacing token "Equal"

The class doing the up/down judgment is a part of for-loop transforming and it is to check the condition of the iteration. It will pass a flag to the class of for-loop transformation so that the latter one knows whether the loop is a circulation with increasing or decreasing iteration. The program will

also check the loop's step, i.e., if it is suitable to be translated into Pascal's for-loop or while-loop and return a flag to inform later part of the program.

The class for replacing the token "Equal" can be a universal class and run in the main class. But those assignments in the variables' initializations will be processed in the translation of variables, so this class is only called in the class of for-loop translating.

4.5.2.4 Main Class

The working classes have been implemented in the last section, now it is time to implement the main class. As opposed to the examples in the SableCC documents, this class here does not only simply activate the lexer and the parser but also activate all the working classes listed above. This is because some of the working classes are not traversing on the whole Concrete Syntax Tree but only on a sub-tree. It is necessary to call them in the proper places in the main class, to pass the right parameters into them and to hold the staff returned from them.

The main class itself is a depth first traversal class on the whole tree. In this pass of the traversal on the tree, classes in which the variables' declaration translated are called at the very beginning, i.e. in the "in method" of the start node. The translation of the for-loop is called in the "in method" of the node for-iteration statement. Others are called in these three classes respectively.

Now it is the time to compile the classes in J2SDK and check if it can translate the input c file into Pascal file correctly.

The system of my program shall run well now, but there are still some significant things to do after that, i.e. the normal Pascal code shall be vectorized so that the performance is very possibly being improved, not only because Vector Pascal compiler's feature of utilizing SIMD instruction set, but also because the benefits from loop unrolling.

4.5.3 The Second Stage

Forward Substitution

To implement this procedure, I designed a new symbol table to record the information about every expression in an entry.

In this symbol table, every entry has three parts. The first one is the key of it, which will record the left-hand side of the current expression. The second one is the “value domain” of the key, which will be used in substitution. This part records the right-hand side of the expression. When traversal on the sub-Syntax Tree which corresponds to the loop’s body, each time the Tree Walker meets an expression, it goes deeper into the right hand side’s sub-tree of that expression. Then the Tree Walker will look into this symbol table to search if the first knot of this sub-tree exists in the table by using this knot as the key. If it finds out relative entry, the Tree Walker will read that value domain out and use it to replace the current knot, then steps to the next knots. If the Tree Walker can not find any relative entry exists, it will send current expression to this symbol table to store it.

The last part is a “sign domain” which holds Boolean values. Once an expression is used in a substitution, for example, expression “*j=l;*” in previous sample, its sign domain will be set as true, which means the key part of this entry is actually an “intermediate variable”.

Once the Tree Walker finished the traversal, we can output all entries whose sign domains’ values are false, which means that these entries shall be kept to guarantee the correctness of the program. At the time, the Syntax Tree is transformed by removing the sub-trees relative to those “intermediate variables”.

4.5.4 The Final Stage

Removing the index variables is not a very difficult task at the moment, since it is similar to the substitution of index variables in the second stage when doing vectorization. Just simply remove all those tokens of “[” and “]” instead of replace the index variables can achieve this goal.

But modifying the function call is a little bit troublesome. Try to recall Figure 4.1, we will find that the header of the function is generated at the very beginning of the program when adding headers to the C source code in simd0.c. Thus the modification must be based on that quite early phase.

Luckily, there was a Symbol Table to record all variables, and we can achieve our target by modifying the related issues in that Symbol Table. Therefore, a traversal on the Syntax Tree to find out the beginning address was added at that phase. Then the program which adds headers and parameters list to simd0.c file will update its Symbol Table and output corrected parameters list.

4.5.5 The Shell Controlling File

We have completed implementing the main functional module so far, but they have to be used manually. To make them as one integrated program, I created several shell script files to combine the control flow together so that the whole system including calling Single Assignment C to generate the source files, program transforming, compiling by Vector Pascal, linking by the GCC linker, and running benchmark will run automatically.

All the C source files for being transformed by my program, will be stored to some temporary directories under "/tmp" in Linux system by the compiler of Single Assignment C, and the name of the target temporary directory changes every time the compiler works, and the directory shall be indicated by another output file: a.out.c which is usually in the same directory of the source file. Since the main program of a.out.c will use them in way of "#include", a search of the directories is needed.

Then the shell script will activate the second and the third stage of my program to generate separated executable programs which will be tested later in the benchmark system.

The output Pascal files from these two stages will be compiled into assembly files by the Vector Pascal compiler. Then I use NASM assembler to generate object file and use the GCC linker to link them all together. The file generated by Vector Pascal demands a runtime library of Vector Pascal which is called "rtl.c" and this file shall also be linked.

Once the executable programs generated, the benchmark will be activated to test and record the time consumption of them.

Chapter 5

Benchmark Testing and Analyzing.

5.1 Introduction

The benchmark system uses sample programs to test the performance of using the Vector Pascal code generator for Single Assignment C with two optimization steps, which are introduced in section 4.2 and 4.3. By examining the results, we will see notable enhancement gained by combining these two languages together.

Since the C file generated by Single Assignment C's compiler is only related to the with-loop in Single Assignment C source code, all tests in this benchmark system are focusing on different situations inside the with-loop and comparing the performance before & after utilizing vector Pascal's help.

5.2 Benchmark Environment

Hardware Environment 1:

Intel Centrino Duo Processor T2300

All testing programs are forced to run on one core during the test.

Hardware Environment 2:

AMD SP 64 2800

Operating System:

Ubuntu Linux 6.10 with Linux kernel 2.6.18

Terminal:

GNU Bash 3.1.17

Single Assignment C's Compiler Version:

Sac2c 1.00 beta (14997)

Optimization Options of the Compilers:

Single Assignment C:

The compiler works in optimization level 3 (-O3), which provides the most expensive and efficient optimizations.

Vector Pascal:

Optimization options from option 0 to option 3 are tested to find out which one has the best performance in this benchmark test.

Calculating Method:

Using RTDSC instruction on x86 platform.

```
1  inline volatile long long RDTSC(){
2      Register long long TSC asm("exa");
3      Asm volatile (".byte 15, 49 "  : : : "eax ", "edx ");
4      Return TSC;
5  }
```

Program 5.1 Using RDTSC instruction in C. This is a C function which contains utilizing assembly code to read value from TSC counter and return this value as a 64-bit integer. This function will be called at the beginning and the end of the main function in a.out.c.

Using the value returned from calling this function in the end of a.out.c minus the one in the beginning of a.out.c, we shall get a result of counting how many clock cycles the executive file a.out (generated from a.out.c) takes.

Calculating the whole program's clock cycle number may affect the performance enhancing ratio, but this effect can be neglected comparing with the plenty of operations (millions) which need to be counted.

The "Performance (ratio)" map & sheet shows the ratio of performance enhancement, which is concluded from the following formulas:

$$
\begin{aligned}
&\textit{the Ratio of the Performance Enhancement} \\
&= \frac{\textit{Speed of Vectorized Program}}{\textit{Original Program's Speed}} \\
&= \frac{\textit{Time Cost by Original Program}}{\textit{Time Cost by Vectorized Program}} \\
&= \frac{\textit{Clock Cycles Cost by Original Program}}{\textit{Clock Cycles Cost by Vectorized Program}}
\end{aligned}
$$

Equation 5.1 The Ratio of the Enhancement of the Performance. The definition of the ratio is derived from its intuitive view which is the division result based on speed, to the division which is based on clock cycles cost by the executable program.

Then the ratio is actually defined by equation:

$$
\begin{aligned}
&\textit{the Ratio of the Performance Enhancement} \\
&= \frac{\textit{Clock Cycles Cost by Original Program}}{\textit{Clock Cycles Cost by Vectorized Program}}
\end{aligned}
$$

Equation 5.2 The definition of the ratio of the performance enhancement. This equation shows how to calculate the actual ratios which are stated in section 5.3. The phrase "original program" indicates the executable program which is only generated by Single Assignment C's compiler. The phrase "vectorized program" indicates the executable program generated by using Vector Pascal compiler as the code generator for Single Assignment C, which is either the output of the second or third stages of my program.

Therefore, if we define the original executable program (generated by Single Assignment C's compiler)'s runtime performance as 1, we can have the ratio "x" which indicates that the enhancement of the program is about x times.

About the Multi-tasking Operating System:

Linux is a multi-tasking operating system based on the time-slicing method, and the time/clock cycle calculation is easily affected by its scheduling scheme. In order to gain accurate result, there are only two active processes during running the benchmark test. One is the tested program and another is the benchmark test system. All other user processes are eliminated, or tuned to sleep. Further more, the

process of benchmark test system will be suspended when the tested programs are running, since the benchmark test system is used to activate the tested programs and record the results. Based on the results of using "time" command, we can see that the costs on system processes can be neglected. Meanwhile, in each test, the program compiled by using each optimization level will be executed for ten times to avoid the exceptional result being recorded.

Format of Test Results and Data Sheets:

-opt	0	1	2	3
VPC with index	Numbers of Clock Cycles & Time Consumed	Numbers of Clock Cycles & Time Consumed	Numbers of Clock Cycles & Time Consumed	Numbers of Clock Cycles & Time Consumed
VPC without index	Numbers of Clock Cycles & Time Consumed	Numbers of Clock Cycles & Time Consumed	Numbers of Clock Cycles & Time Consumed	Numbers of Clock Cycles & Time Consumed
SAC(-O3)	Numbers of Clock Cycles & Time Consumed			

This is the data sheet which will be used in section 5.3 to show the test results. The first line of it indicates the optimization option of Vector Pascal's compiler being used to generate executable files for the tested programs. The second line records the results of programs generated in the second stage of my program i.e., vectorized programs in which arrays' indices are reserved; and the third line records the results of programs generated in the final stage of my program which are further optimized. The last line keeps the result of running the program which is only compiled by Single Assignment C's compiler. The programs generated by the first stage of my program will not be presented here because there is no optimization in that stage but only translating the code into Pascal code and it is not worthwhile to test the performance of them.

The test results in the sheet include two parts. One is the numbers of clock cycles recorded by using RDTSC instruction introduced in section 4.4 with unit clock cycles (shorten as CC in following data sheets), and another is the results of using "time" command in Linux operating system which provides a brief but less accurate view. When calculating the ratio of the performance enhancement, I choose Equation 5.2 which is based on using numbers of clock cycles instead of using results from "time" command.

5.3 Benchmark Test

This section shows the benchmark test results and the analysis of them. Figure 5.1.1 gives the overview of the first 7 benchmark tests' results (programs processing arrays of 32-bit integers). Data in this sheet are the highest enhancement ratio in each test. Figure 5.1.2 gives the overview of benchmark testing results of test 8, 9, and 10 (processing arrays of 64-bit floating point numbers).

As explained in last section, all ratio in this section will be calculated by Equation 5.2, and the performance of original executable program which is compiled by Singe Assignment C's compiler in each case will be defined as '1' when calculating the ratio.

	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7
Intel	4.38	5.12	6.66	2.84	2.62	1.96	1.78
AMD	2.00	8.37	5.04	5.71	3.77	2.05	1.86

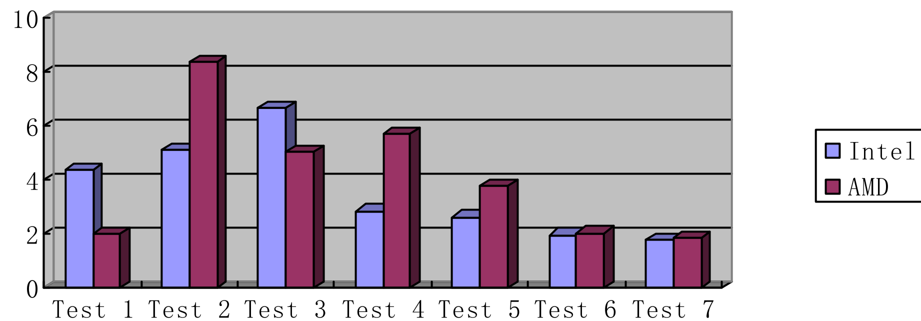


Figure 5.1.1 The overview of the results of benchmark tests on integers (ratio diagram). Since the further optimized programs generated in the third stage of my program always show better performance than the corresponding ones generated in the second stage of my program, the ratio are all calculated following the definition of Equation 5.2 by using result of clock cycles cost by original programs (only compiled by Single Assignment C's compiler) over which clock cycles cost by further programs generated in the third stage of my program.

	Test 8	Test 9	Test 10
Intel	3.163	1.982	1.422

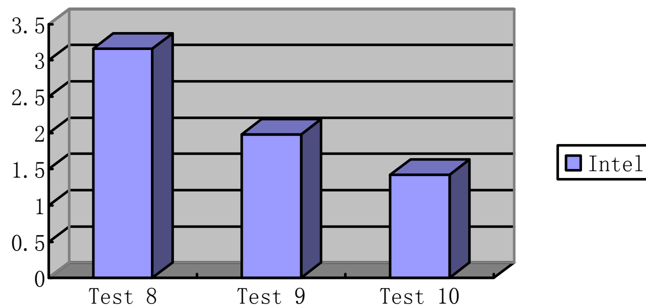


Figure 5.1.2 The overview of the results of benchmark tests on floating point numbers (ratio diagram). Because the computer in hardware environment 2 is no longer available for me, this part only contains benchmark test result in hardware environment 1 (on Intel Processor).

We can see that the enhancement ratio is great in both hardware environments（Intel & AMD）when dealing with programs which process operations on arrays of integers. And it is also notable when testing programs processing arrays of 64-bit floating point numbers (double precision)

5.3.1 Benchmark Test on Integers

In this section, the tested programs are all processing 32-bit integers. As explained in Chapter 3, we shall expect four-time enhancement, though this number may be diminished in practical because the ratio of operations which are suitable for using SIMD instruction set may be decreased. Variables in the C files generated by Single Assignment C in the following tests will be simplified for the convenience of reading.

Test 1

The first test program is one of the simplest programs, which only tests the system's performance when processing integers' storage, i.e. generate a large array with integer constants.

```
1  use Structures:all;
2  use SimplePrint:all;
3  int main()
4  {
5      v1=with(.<=iv<=.)
6      genarray([1000000],6);
7      res=print(v1);
8      return(res);
9  }
```

Program 5.2.1 Original source code in Single Assignment C Test 1. This program is used to initialize an array of integers. Array v1 contains 1,000,000 elements.

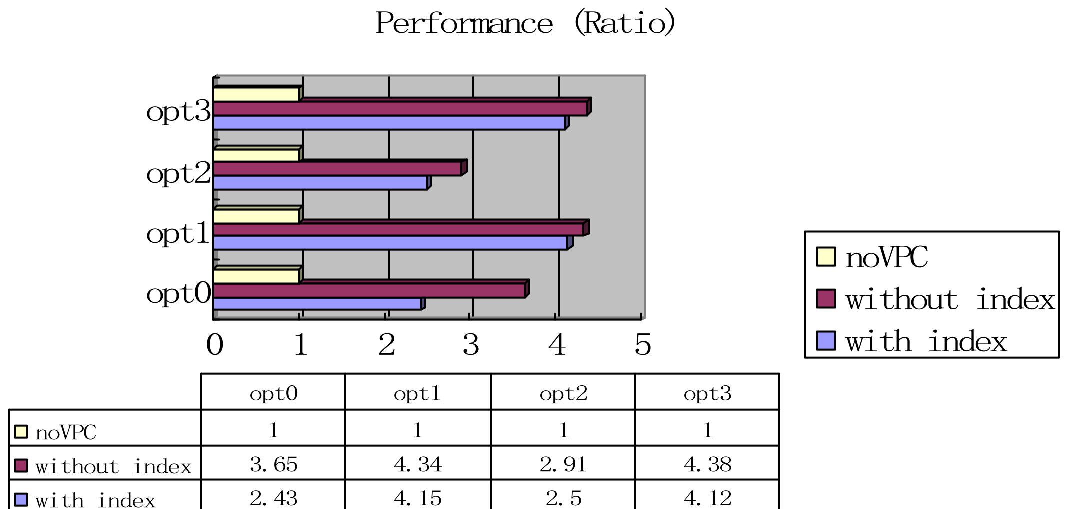
The corresponding output C file (named as simd0.c) of this program generated by Single Assignment C's compiler contains the following code:

```
1  {
2    int s = 1000000;
3    for (i = 0; i < 1000000; ){
4      v1[j] = k;
5      j = j + 1;
6      i = i + 1;
7    }
8  }
```

Program 5.2.2 The for-loop contained in simd0.c. This file is the intermediate code in C, generated by Single Assignment C's compiler. All variables' names in this program have been simplified for convenience of reading. This program is corresponding to the innermost loop of the Program 5.2.1, a with-loop which consists of line 5 and line 6, and is used to do the initializing job on the array.

Results in Hardware Environment 1 (Intel):

-opt	0	1	2	3
VPC with index	11,755,620CC real 0m0.006s user 0m0.000s Sys 0m0.004s	6,896,090CC real 0m0.006s user 0m0.004s sys 0m0.000s	11,464,060CC real 0m0.007s user 0m0.000s sys 0m0.008s	6,950,720CC real 0m0.005s user 0m0.000s sys 0m0.004s
VPC without index	7,845,760CC real 0m0.006s user 0m0.000s sys 0m0.004s	6,588,460CC real 0m0.005s user 0m0.004s sys 0m0.000s	9,847,340CC real 0m0.006s user 0m0.004s sys 0m0.004s	6,531,570CC real 0m0.005s user 0m0.000s sys 0m0.004s
SAC (-O3)	28,615,450CC real 0m0.005s user 0m0.000s sys 0m0.004s			



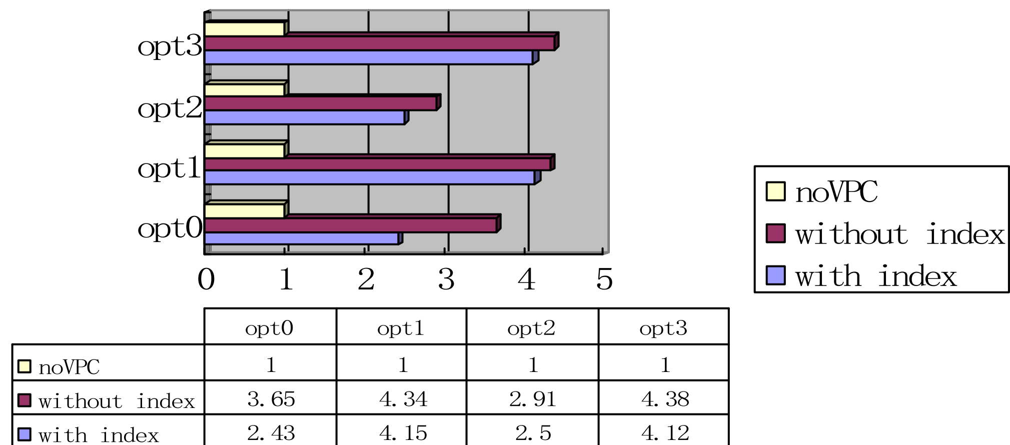
	opt0	opt1	opt2	opt3
noVPC	1	1	1	1
without index	3.65	4.34	2.91	4.38
with index	2.43	4.15	2.5	4.12

Figure 5.2.2 The results (ratio diagram) of Test 1 in hardware environment 1 (Intel). This diagram gives out the performance in each case based on the ratio defined by Equation 5.2 in hardware environment 1. The performance of original program which is only compiled by Single Assignment C's compiler is defined as '1'.

Results in Hardware Environment 2 (AMD):

-opt	0	1	2	3
VPC with index	17,024,928CC 0.00s user 0.01s system 0.01s elapsed	13,530,550 CC 0.00s user 0.00s system 0.00s elapsed	17,064,308 CC 0.01s user 0.00s system 0.02s elapsed	13,240,538 CC 0.01s user 0.00s system 0.01s elapsed
VPC without index	14,116,096 CC 0.00s user 0.00s system 0.00s elapsed	12,912,789 CC 0.00s user 0.01s system 0.01s elapsed	13,981,173 CC 0.00s user 0.00s system 0.01s elapsed	12,734,759 CC 0.00s user 0.00s system 0.01s elapsed
SAC (-O3)	25,499,114 CC 0.01s user, 0.00s system, 0.01s elapsed			

Performance (Ratio)



	opt0	opt1	opt2	opt3
noVPC	1	1	1	1
without index	3.65	4.34	2.91	4.38
with index	2.43	4.15	2.5	4.12

Figure 5.2.2 The results (ratio diagram) of Test 1 in hardware environment 2 (AMD). This diagram gives out the performance in each case based on the ratio defined by Equation 5.2 in hardware environment 2. The performance of original program which is only compiled by Single Assignment C's compiler is defined as '1'.

Analysis:

This sample program is a very simple one which only contains an array's initialization. In this case, all operations relative to the array will be included in the part of the code which will be translated and vectorized into Vector Pascal code, then compiled by Vector Pascal compiler. So it is actually the best program which reflects the enhancement and the enhancement ratio of the performance.

From the above two sheets we can see that the best performance is gained by using Vector Pascal compiler with its optimization option "–opt3", in which the highest enhancement ratio is about 4.4 times at most (4.3811) on Intel processor (in hardware environment 1).

Test 2

This program adds two arrays of integers together, and prints the first elements of the array v1 which holds the result of the addition operation.

```
1   use Structures:all;
2   use SimplePrint:all;
3   int main()
4   {
5    v1=genarray([100],2);
6    v2=genarray([100],1);
7    for(i=0;i<1000000;i++)
8    {
9       v1=with(iv)
10      (.<=iv<=.) : v1[iv]+v2[iv];
11      modarray(v1);
12   }
13   res=print(v1[0]);
14   return(res);
15  }
```

Program 5.3.1 Original source code in Single Assignment C for Test 2. This program is used to initialize arrays and perform addition operation.

The corresponding output C files of this program generated by Single Assignment C's compiler are as follows:

```
1  {
2    int s = 100;
3    for (i = 0; i < 100; ){
4      v1[j] = k;
5      j = j + 1;
6      i = i + 1;
7    }
8  }
```

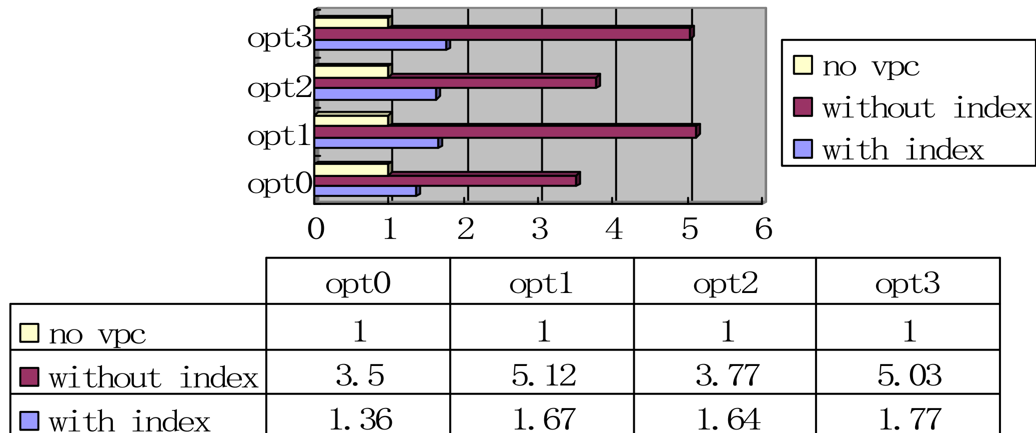
Program 5.3.2 The for-loop in simd0.c. This program assignments initial values to each position in v1.

```
1  {
2    int s = 100;
3    for (i = 0; i < 100; ){
4      iv = j;
5      tmp = ( SACp_pinl_368____v1[iv] );
6      tmp = ((tmp)+ (1));
7      SACp_pinl_365__emal_334_v1__SSA0_1[j] = tmp;
8      j = j + 1;
9      i = i + 1;
10   }
11 }
```

Program 5.3.3 The for-loop in simd1. This program is corresponding to the innermost loop presents between line 9 and line 11 in Program 5.3.1, which adds two arrays together. In this program, we can see that it copies the value from each original position of v1 (array SACp_pinl_368___v1 is the actual place storing v1's elements) to a temporary intermediate variable "tmp" in line 5, adds the value of the corresponding elements in v2 (which is a constant of integer 1 here) to that temporary variable in line 6. Then in line 7, it stores the result of addition operation into a new version of v1 which is "SACp_pinl_365__emal_334_v1__SSA0_1" , because Static Singe Assignment Form has been employed in this intermediate represent level. The reason of using constant '1' value in line 6 to present v2 instead of a variable is that this with-loop is actually operates on array v1, and array v2 holds constants but not variables as its elements, thus the compiler optimizes the code during generating this segment of intermediate code.

Results in Hardware Environment 1 (Intel):

-opt	0	1	2	3
VPC with index	1,359,164,150 CC real 0m0.887s user 0m0.828s sys 0m0.000s	1,106,725,400 CC real 0m0.694s user 0m0.692s sys 0m0.000s	1,131,901,850 CC real 0m0.759s user 0m0.724s sys 0m0.000s	1,045,992,430 CC real 0m0.704s user 0m0.696s sys 0m0.004s
VPC without index	528,571,330 CC real 0m0.376s user 0m0.372s sys 0m0.004s	362,108,910 CC real 0m0.238s user 0m0.220s sys 0m0.000s	491,175,930 CC real 0m0.324s user 0m0.324s sys 0m0.000s	368,370,390 CC real 0m0.259s user 0m0.256s sys 0m0.004s
SAC (-O3)	1,852,392,430 CC real 0m1.212s user 0m1.176s sys 0m0.000s			

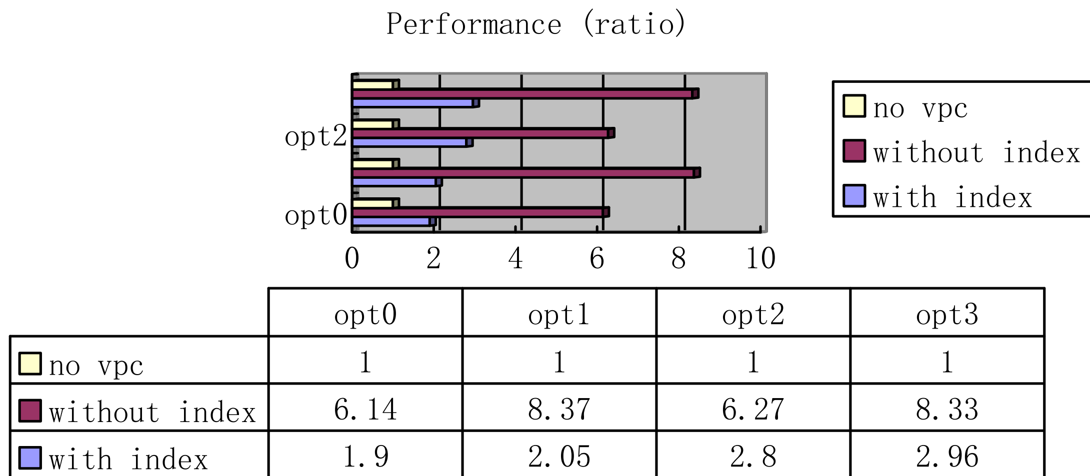


	opt0	opt1	opt2	opt3
no vpc	1	1	1	1
without index	3.5	5.12	3.77	5.03
with index	1.36	1.67	1.64	1.77

Figure 5.3.1 The results (ratio diagram) of Test 2 in hardware environment 1 (Intel). This diagram gives out the performance in each case based on the ratio defined by Equation 5.2 in hardware environment 1. The performance of original program which is only compiled by Single Assignment C's compiler is defined as '1'.

Results in Hardware Environment 2 (AMD):

-opt	0	1	2	3
VPC with index	1,464,272,722 CC 1.03s user 0.00s system 1.35s elapsed	1,352,306,525 CC 1.01s user 0.00s system 1.03s elapsed	992,075,458 CC 0.70s user 0.01s system 1.00s elapsed	937,031,924 CC 0.64s user 0.00s system 1.01s elapsed
VPC without index	452,354,892 CC 0.32s user 0.00s system 0.45s elapsed	331,615,170 CC 0.23s user 0.00s system 0.29s elapsed	442,406,423 CC 0.32s user 0.00s system 0.40s elapsed	333,405,240 CC 0.23s user 0.00s system 0.34s elapsed
SAC (-O3)	2,775,482,985 CC 1.94s user, 0.01s system, 2.14s elapsed			



	opt0	opt1	opt2	opt3
no vpc	1	1	1	1
without index	6.14	8.37	6.27	8.33
with index	1.9	2.05	2.8	2.96

Figure 5.3.2 The results (ratio diagram) of Test 2 in hardware environment 2 (AMD). This diagram gives out the performance in each case based on the ratio defined by Equation 5.2 in hardware environment 2. The performance of original program which is only compiled by Single Assignment C's compiler is defined as '1'.

Analysis:

In this program, the vectorized parts are composed of the addition operation and the assignment operations in Program 5.3.2 and 5.3.3. The arrays' sizes are not very big (100 elements per array), but the loop's size is notable (1,000,000 iterations). The executable programs generated by using Vector Pascal compiler with its optimization option "–opt1" and "–opt3" get almost the same results. The highest enhancement ratio is gained on AMD processor with "–opt1" which is 8.37.

This result is a little bit astonishing at a first glance. Since there are only 128-bit registers inside the processors' kernel and we are processing 32-bit integers, the highest reasonable ratio of the performance shall be up to 4. But some other things may affect it. To find out the reasons, let's have a look on a simple example. Consider a for-loop in C as follows (which is a simplified version of the input "simd1.c" files):

```
1  int foo(int *v1, int *v2)
2  {
3      int i=0;
4      for(i=0;i<1000;i++)
5      {
6          v2[i]=v1[i]+1;
7      }
8      return *v2;
9  }
```

Figure 5.3.3 A for-loop wrapped in a function in C. This program is a C function which contains a for-loop inside. The loop performs operations of adding a constant to the first one thousand elements of array "v1" and storing them to array v2's corresponding positions. This program will be compiled by using the GCC compiler to generate assembly code.

All intermediate C files will be compiled by using the GCC compiler except those will be vectorized by my program and be compiled by Vector Pascal. Thus we just need to compare this for-loop in C with its vectorized version in Vector Pascal. The corresponding vectorized Vector Pascal program is as fellows:

```
1  library test;
2  interface
3  TYPE INTARRAY=ARRAY [0..999] of INTEGER;
4  procedure foo(var v1,v2 :INTARRAY);
5  implementation
6  procedure foo(var v1,v2 :INTARRAY);
7  begin
8      v2:= v1+1;
9  end;
10 begin
11 end.
```

Figure 5.3.4 The vectorized Vector Pascal version of program in Figure 5.3.3. As my program does in its third stage, this one is the vectorized code transformed from the C program in Figure 5.3.3. It will be compiled by Vector Pascal compiler to generate assemble code.

The vectorized program contains a vectorized version of the array operation, between line 7 and line 9 the program adds the constant to array "v1" and stores the result to array "v2". If we check the assembly file generated by them, we can calculate the numbers of load and store instructions and get result as follows:

	In the assembly file generated by Vector Pascal	In the assembly file generated by GCC
Number of Total Instructions	2900	13000
Number of Total Load Instructions	2000	6000
Number of Total Store Instructions	500	2000
Number of Iteration times	100	1000

The assembly files are attached in the appendix of this dissertation. From the sheet above, we can see that the number of total instructions in Vector Pascal version is only 2900/13000 ≈ 22.3% of that in C version, and that the number of total load instructions in Vector Pascal version is about 1/3 while the number of total store instructions is only 25% of the C version. Moreover, the iteration times are also reduced in Vector Pascal version to 10% of the C version, this also diminish the hazard of fail in branch prediction and operand pre-fetching which may cause the stall of the pipeline inside the processors. Thus it is not odd to get the results shown in Figure 5.3.2.

Test 3

This program has the same amount of the total operations as Program 5.3.1, but this one has bigger arrays' sizes while Program 5.3.1 adopts bigger loop size.

```
1
2  use Structures:all;
3  use SimplePrint:all;
4  int main()
5  {
6   v1=genarray([100000],2);
7   v2=genarray([100000],1);
8   for(i=0;i<1000;i++)
9   {
10      v1=with(iv)
11      (.<=iv<=.) : v1[iv]+v2[iv];
12      modarray(v1);
13   }
14   res=print(v1[0]);
15   return(res);
16 }
```

Program 5.4.1 Original source code in Single Assignment C for Test 3. This program adds two arrays, v1 and v2 together and prints the first element of v1 to the standard output.

The corresponding output C files of this program generated by Single Assignment C's compiler as follows:

```
1  {
2    int s = 100000;
3    for (i = 0; i < 100000; ){
4      v1[j] = SACp_emal_342__pinl_156__flat_60;
5      j = j + 1;
6      i = i + 1;
7    }
8  }
```

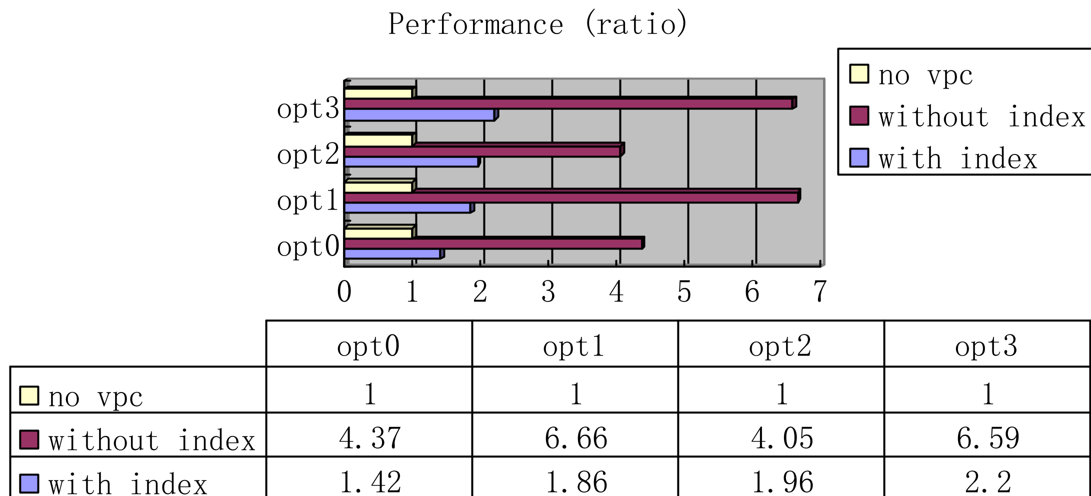
Program 5.4.2 The for-loop in simd0.c. This program is for initializing array v1.

```
1  {
2    int s = 100000;
3    for (i = 0; i < 100000; ){
4      iv = j;
5      tmp = ( SACp_pinl_368____v1[iv] );
6      tmp = ((tmp)+ (1));
7      SACp_pinl_365__emal_334_v1__SSA0_1[j] = tmp;
8      j = j + 1;
9      i = i + 1;
10   }
11 }
```

Program 5.4.3 The for-loop in simd1.c. This program is similar to the simd1.c file in Test 2, it adds the value of array v2 to each element of v1. The difference is that the size of v1 in this program is bigger than the one in Program 5.3.3, therefore the size of this for-loop in C language is bigger.

Results in Hardware Environment 1 (Intel):

-opt	0	1	2	3
VPC with index	1,245,237,260 CC real 0m0.756s user 0m0.748s sys 0m0.004s	955,171,810 CC real 0m0.656s user 0m0.652s sys 0m0.000s	906,337,470 CC real 0m0.600s user 0m0.592s sys 0m0.000s	806,052,330 CC real 0m0.541s user 0m0.540s sys 0m0.004s
VPC without index	406,466,820 CC real 0m0.285s user 0m0.276s sys 0m0.000s	266,434,080 CC real 0m0.233s user 0m0.228s sys 0m0.004s	438,162,210 CC real 0m0.299s user 0m0.292s sys 0m0.004s	269,347,530 CC real 0m0.197s user 0m0.196s sys 0m0.000s
SAC (-O3)	1,774,243,460 CC real 0m1.186s user 0m1.132s sys 0m0.000s			



	opt0	opt1	opt2	opt3
no vpc	1	1	1	1
without index	4.37	6.66	4.05	6.59
with index	1.42	1.86	1.96	2.2

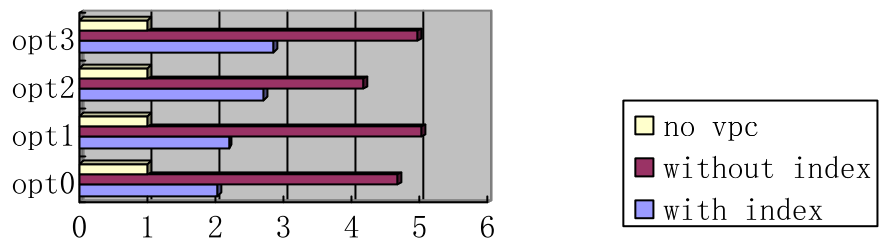
Figure 5.4.1 the results (ratio diagram) of Test 3 in hardware environment 1 (Intel). This diagram gives out the performance in each case based on the ratio defined by Equation 5.2 in hardware environment 1. The performance of original program which is only compiled by Single Assignment C's compiler is defined as '1'.

Results in Hardware Environment 2 (AMD):

-opt	0	1	2	3

VPC with index	1,417,301,021 CC 1.00s user 0.01s system 1.14s elapsed	1,313,746,136 CC 0.92s user 0.00s system 1.05s elapsed	1,062,701,657 CC 0.77s user 0.00s system 1.07s elapsed	1,009,509,952 CC 0.73s user 0.00s system 0.85s elapsed
VPC without index	615,850,294 CC 0.47s user 0.00s system 0.56s elapsed	570,590,779 CC 0.46s user 0.00s system 0.59s elapsed	688,716,523 CC 0.45s user 0.00s system 0.66s elapsed	579,121,546 CC 0.41 s user 0.00s system 0.54s elapsed
SAC (-O3)	2,877,127,608 CC 2.02s user, 0.01s system, 2.82s elapsed			

Performance (ratio)



	opt0	opt1	opt2	opt3
no vpc	1	1	1	1
without index	4.67	5.04	4.18	4.97
with index	2.03	2.19	2.71	2.85

Figure 5.4.2 The results (ratio diagram) of Test 3 in hardware environment 2 (AMD). This diagram gives out the performance in each case based on the ratio defined by Equation 5.2 in hardware environment 2. The performance of original program which is only compiled by Single Assignment C's compiler is defined as '1'.

Analysis:

The original Single Assignment C's program in this test is similar to the one in Test 2; the differences are that the loops' and arrays' sizes in this program have been adjusted, so that the arrays' sizes become notable. The executable programs generated by using Vector Pascal compiler with optimization option "–opt1", "-opt3" enjoy the best performance enhancement. The highest ratio of the enhancement is about 6.67 on Intel processor. Besides, if we compare the result of this test with the one from Test 2, we can initially conclude that when the numbers of total calculating operations are the same, program with bigger arrays' sizes gains fewer enhancements than that with bigger loop size.

Test 4

This program tests some more complicate operations, including generating a two dimensional array.

```
1  use Structures:all;
2  use SimplePrint:all;
3  int main()
4  {
5      v1=genarray([1000],1);
6          v2=genarray([1000],2);
7          v3=genarray([1000],3);
8
9          for(i=0;i<100000;i++)
10         {
11                 v1=with (iv)
12                 (.<iv<.) : v1[iv]+v2[iv]*v3[iv];
13                 modarray(v1);
14
15         }
16     v4=with(.<iv<.)
17     genarray([10],v1,v1);
18     res=print(v4);
19     return(res);
20 }
```

Program 5.5.1 Original source code in Single Assignment C for Test 4. This program multiplies each elements in corresponding position of arrays v2 and v3, adds the result to array v1, then generate a two dimensional array v4.

The corresponding output C files of this program generated by Single Assignment C's compiler as follows:

```
1  {
2    int s = 1000;
3    for (i = 0; i < 1000; ){
4      v1[j] = SACp_emal_368__pinl_141__flat_60;
5      j = j + 1;
6      i = i + 1;
7    }
8  }
```

Program 5.5.2 for-loop in simd0.c. This program is for initializing v1.

```
{
  int s = 1000;
  for (i = 0; i < 1000; ){
    iv = j;
    tmp = ( SACp_pinl_395____v1[iv] );
    tmp = ((tmp)+ (6));
    SACp_pinl_392__emal_355_v1__SSA0_1[j] = tmp;
    j = j + 1;
    i = i + 1;
  }
}
```

Program 5.5.3 The for-loop in simd1.c. This program adds the result of the multiplication of each elements in corresponding position of arrays v2 and v3 to array v1 (presented by SACp_pinl_395____v1 in line 5), then store the result to a new version of v1 which is SACp_pinl_392__emal_355_v1__SSA0_1 in line 7.Since arrays v2 and v3 contain constants, the compiler calculates the result of multiplication in compile time, and uses the result as a constant in this program, which we can see the integer '6' in line 6

```
{
  int s= 1000;
  for (i = 0; i < 1000; ){
    tmp = i;
    tmp = ( SACl_v1__SSA0_1[tmp] );
    SACp_emal_359_v4[j] = tmp;
    j = j + 1;
    i = i + 1;
  }
}
```

Program 5.5.4 The for-loop in simd2.c This program is used to generate the two dimensional array v4. It assigns values to the target positions in array SACp_emal_359_v4, which is actually where elements of v4 are stored.

```
1  {
2   int s = 1000;
3   for (i = 0; i < 1000; ){
4     tmp = i;
5     tmp = ( SAC1_v1__SSA0_1[tmp] );
6     SACp_emal_359_v4[j] = tmp;
7     j = j + 1;
8     i = i + 1;
9   }
10 }
```

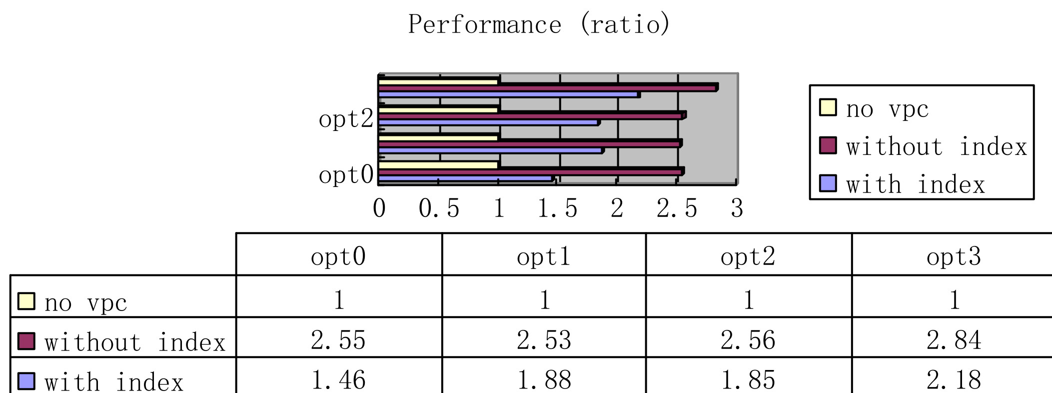
Program 5.5.5 The for-loop in simd3.c. This program is used to generate the two dimensional array v4. It assigns values to the target positions indicated of v4. The content of this program is exactly the same as the one in Program 5.5.4 and Program 5.5.6, but they will be called in different places of the main program inside "a.out.c". By inserting output expressions into the main program, we can see that the index variables of array SACp_emal_359_v4 are different, which means that these programs write data to different positions of the array.

```
1  {
2   int s = 1000;
3   for (i = 0; i < 1000; ){
4     tmp= i;
5     tmp= ( SAC1_v1__SSA0_1[tmp] );
6     SACp_emal_359_v4[j] = tmp;
7     j = j + 1;
8     i = i + 1;
9   }
10 }
```

Program 5.5.6 The for-loop in simd4.c. This program plays the same role as Program 5.5.5, and for the same reason, its content is same with Program 5.5.4 and Program 5.5.5.

Results in Hardware Environment 1 (Intel):

-opt	0	1	2	3
VPC with index	1,268,474,970 CC real 0m0.765s user 0m0.756s sys 0m0.000s	981,782,960 CC real 0m0.613s user 0m0.612s sys 0m0.000s	999,333,880 CC real 0m0.642s user 0m0.608s sys 0m0.000s	846,188,870 CC real 0m0.583s user 0m0.516s sys 0m0.024s
VPC without index	725,373,430 CC real 0m0.474s user 0m0.476s sys 0m0.000s	728,964,070 CC real 0m0.444s user 0m0.444s sys 0m0.000s	721,709,720 CC real 0m0.453s user 0m0.452s sys 0m0.000s	650,150,610 CC real 0m0.454s user 0m0.416s sys 0m0.000s
SAC (-O3)	1,846,279,140 CC real 0m1.199s user 0m1.192s sys 0m0.000s			

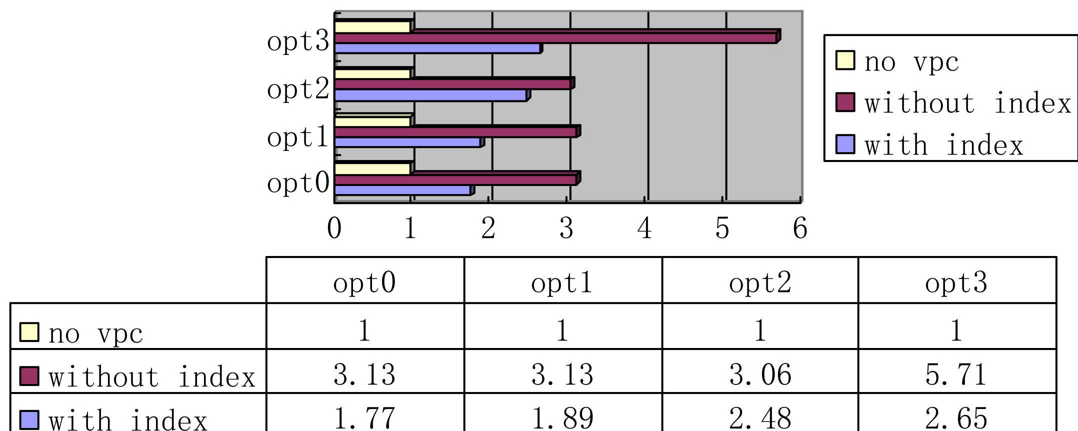


	opt0	opt1	opt2	opt3
no vpc	1	1	1	1
without index	2.55	2.53	2.56	2.84
with index	1.46	1.88	1.85	2.18

Figure 5.5.1 The results (ratio diagram) of Test 4 in hardware environment 1 (Intel). This diagram gives out the performance in each case based on the ratio defined by Equation 5.2 in hardware environment 1. The performance of original program which is only compiled by Single Assignment C's compiler is defined as '1'.

Results in Hardware Environment 2 (AMD):

-opt	0	1	2	3
VPC with index	1,541,640,990 CC real 0m1.326s user 0m1.092s sys 0m0.004s	1,440,778,375 CC real 0m1.038s user 0m1.032s sys 0m0.004s	1,100,067,900 CC real 0m0.795s user 0m0.784s sys 0m0.008s	1,030,035,700 CC real 0m0.716s user 0m0.704s sys 0m0.000s
VPC without index	874,111,045 CC real 0m0.627s user 0m0.624s sys 0m0.000s	874,010,707 CC real 0m0.631s user 0m0.628s sys 0m0.000s	892,414,261 CC real 0m0.660s user 0m0.656s sys 0m0.000s	478,495,329 CC real 0m0.349s user 0m0.340s sys 0m0.004s
SAC (-O3)	2,732,381,711 CC real 0m1.958s user 0m1.952s sys 0m0.004s			



	opt0	opt1	opt2	opt3
no vpc	1	1	1	1
without index	3.13	3.13	3.06	5.71
with index	1.77	1.89	2.48	2.65

Figure 5.5.2 The results (ratio diagram) of Test 4 in hardware environment 2 (AMD). This diagram gives out the performance in each case based on the ratio defined by Equation 5.2 in hardware environment 2. The performance of original program which is only compiled by Single Assignment C's compiler is defined as '1'.

Analysis:

On Intel processor, the max enhancement ratio is about 2.84, while on AMD processor it is about 5.7. The vectorized program without arrays' indices with "–opt3" of Vector Pascal's optimization option runs much faster than others, and this enhancement is pretty stable in this case. But in other cases, the enhancement is lowered by the mix of the two dimensional array's generation and other

calculating operations.

Test 5

This program is similar to Program 5.5.1 except that this program adopts a built- in operation on arrays, i.e. rotation of the arrays, shown as line 15 in Program 5.6.1.

```
1   use Structures:all;
2   use SimplePrint:all;
3   int main()
4   {
5       v1=genarray([1000000],2);
6       v2=genarray([1000000],1);
7       v3=genarray([1000000],3);
8
9       for(i=0;i<100;i++)
10      {
11          v1=with(iv)
12          (.<iv<.) : v1[iv]+v2[iv]*v3[iv];
13          modarray(v1);
14      }
15      v3=rotate(0,1,v1);
16
17      res=print(v3[99]);
18      return(res);
19  }
```

Program 5.6.1 Original source code in Single Assignment C for Test 5. The difference of this program from Program 5.5.1 is in line 15, which is rotation array v3, instead of generating a two dimensional array.

The corresponding output C files of this program generated by Single Assignment C's compiler are as follows:

```
1  {
2    int s = 1000000;
3    for (i = 0; i < 1000000; ){
4      SACp_emal_883_v1[j] = SACp_emal_885__pinl_491__flat_60;
5      j = j + 1;
6      i = i + 1;
7    }
8  }
```

Program 5.6.2 The for-loop in simd0.c This program is used to initialize array v1, array SACp_emal_883_v1 is

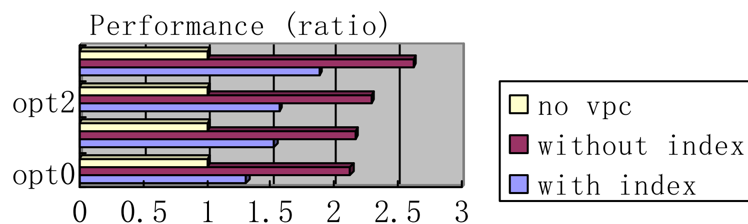
where values of elements in v1 are stored.

```
1  {
2    int s = 999999;
3    for (i = 1; i < 999999; ){
4      iv = j;
5      tmp = ( SACp_pinl_915____v1[iv] );
6      tmp = ((tmp)+ (3));
7      SACp_pinl_910__emal_866_v1__SSA0_1[j] = tmp;
8      j = j + 1;
9      i = i + 1;
10   }
11 }
```

Program 5.6.3 The for-loop in simd1.c. This program is similar to Program 5.5.3, and for the same reason, i.e., arrays v2 and v3 contain constants, the compiler calculates the result of multiplication in compile time, and use the result as a constant in this program, showing the integer '3' in line 6.

Results in Hardware Environment 1 (Intel):

-opt	0	1	2	3
VPC with index	1,4079,555,40 CC real 0m0.866s user 0m0.856s sys 0m0.012s	1,204,107,470 CC real 0m0.754s user 0m0.728s sys 0m0.004s	1,169,556,800 CC real 0m0.583s user 0m0.568s sys 0m0.012s	974,926,870 CC real 0m0.649s user 0m0.640s sys 0m0.004s
VPC without index	863,283,500 CC real 0m0.552s user 0m0.532s sys 0m0.000s	848365090 CC real 0m0.529s user 0m0.520s sys 0m0.008s	804,685,560 CC real 0m0.461s user 0m0.440s sys 0m0.008s	702,892,710 CC real 0m0.463s user 0m0.452s sys 0m0.008s
SAC (-O3)	1,840,203,190 CC real 0m1.234s user 0m1.184s sys 0m0.012s			



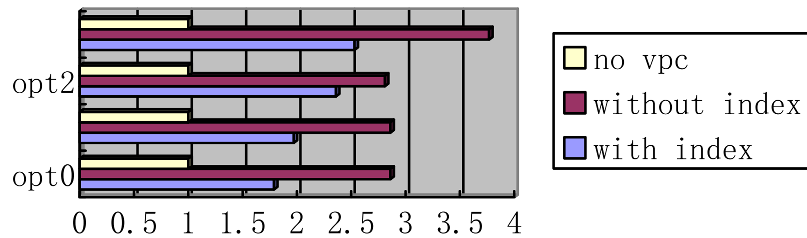
	opt0	opt1	opt2	opt3
no vpc	1	1	1	1
without index	2.13	2.17	2.29	2.62
with index	1.31	1.53	1.57	1.89

Figure 5.6.1 The results (ratio diagram) of Test 5 in hardware environment 1 (Intel). This diagram gives out the

performance in each case based on the ratio defined by Equation 5.2 in hardware environment 1. The performance of original program which is only compiled by Single Assignment C's compiler is defined as '1'.

Results in Hardware Environment 2 (AMD):

-opt	0	1	2	3
VPC with index	1,691,160,857 CC 1.18 s user 0.02s system 1.53s elapsed	1,533,468,469 CC 1.08 s user 0.01s system 1.21s elapsed	1,281,472,110 CC 0.90 s user 0.01s system 0.95s elapsed	1,195,245,284 CC 0.85s user 0.00s system 1.18s elapsed
VPC without index	1,058,230,030 CC 0.74 s user 0.01s system 0.82s elapsed	1,057,368,102 CC 0.74s user 0.01s system 0.82s elapsed	1,075,452,133 CC 0.75 s user 0.01s system 0.85s elapsed	800,543,842 CC 0.56 s user 0.00s system 0.67s elapsed
SAC (-O3)	3,022,052,178 CC 2.15s user, 0.02s system, 2.36s elapsed			



	opt0	opt1	opt2	opt3
no vpc	1	1	1	1
without index	2.86	2.86	2.81	3.77
with index	1.79	1.97	2.36	2.53

Figure 5.6.2 The results (ratio diagram) of Test 5 in hardware environment 2 (AMD). This diagram gives out the performance in each case based on the ratio defined by Equation 5.2 in hardware environment 2. The performance of original program which is only compiled by Single Assignment C's compiler is defined as '1'.

Analysis:

Initially, I was expecting that the Single Assignment C's compiler will generate some "simd" files for the rotation operation, and the program may benefit from vectorizing them. But after invoking Single Assignment C's compiler, I found that it only generated files for the assignment and addition operations.

Though this test didn't gain any information about performance enhancement for some new operations, it shows that the enhancement has been decreased because the ratio of the vectorized

parts is not as large as in first three tests. In this test, the biggest enhancement ratio on Intel processor is about 2.62, and on AMD it is about 3.77.

Test 6

This program tests another form of With-Loop of Single Assignment C, i.e., it performs different operations on different selected parts of the arrays.

```
1  use Structures:all;
2  use SimplePrint:all;
3  int main()
4  {
5      v1=genarray([10000],5);
6      v2=genarray([10000],5);
7      for(i=0;i<10000;i++)
8      {
9          v1=with(iv)
10         (.<=iv<=[5000]):v1[iv]+v2[iv];
11         ([5001]<=iv<=[9000]):v1[iv]-v2[iv];
12         ([9001]<=iv<=.):v1[iv]+v2[iv];
13         modarray(v1);
14     }
15     res=print(v1[9]);
16     return(res);
17 }
```

Program 5.7.1 Original source code in Single Assignment C for Test 6. Different from previous Tests, there are some selection operations in the with-loop inside this program. This will affect the vectorization, i.e., my program has to calculate the correct address for each vectorized programs.

The corresponding output C files of this program generated by Single Assignment C's compiler are as follows:

```
{
 int s = 10000;
 for (i = 0; i < 10000; ){
   SACp_emal_502_v1[j] = SACp_emal_504__pinl_179__flat_60;
   j = j + 1;
   i = i + 1;
 }
}
```

Program 5.7.2 The for-loop in simd0.c. This program is for initializing array v1; array SACp_emal_502_v1 is where actually stores the initial values of the elements..

```
{
 int s = 5001;
 for (i = 0; i < 5001; ){
   iv = j;
   tmp = ( SACp_pinl_544____v1[iv] );
   tmp = ((tmp)+ (5));
   SACp_pinl_535__emal_488_v1__SSA0_1[j] = tmp;
   j = j + 1;
   i = i + 1;
 }
}
```

Program 5.7.3 The for-loop in simd1.c. This program is related to the first segment of the selection in the original program's with-loop, i.e., it operates on elements from position 0 to 5,000 of the arrays.

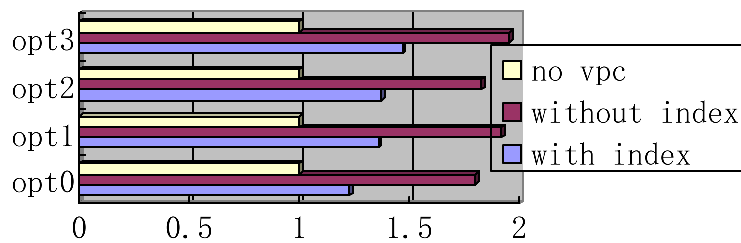
```
{
 int s= 10000;
 for (i = 9001; i < 10000; ){
   tmp = ( SACp_pinl_544____v1[iv] );
   tmp = ((tmp)+ (5));
   SACp_pinl_535__emal_488_v1__SSA0_1[j] = tmp;
   j = j + 1;
   i = i + 1;
 }
}
```

Program 5.7.4 The for-loop in simd2.c. This program is related to the first segment of the selection in the original program's with-loop, i.e., it operates on elements from position 9,001 to 9,999 of the arrays.

Results in Hardware Environment 1 (Intel):

-opt	0	1	2	3
VPC with index	1,440,054,580 CC real 0m0.942s user 0m0.940s sys 0m0.004s	1,300,131,780 CC real 0m0.795s user 0m0.792s sys 0m0.000s	1,296,700,520 CC real 0m0.832s user 0m0.832s sys 0m0.000s	1,201,885,360 CC real 0m0.756s user 0m0.752s sys 0m0.008s
VPC without index	982,681,140 CC real 0m0.594s user 0m0.592s sys 0m0.004s	921,123,920 CC real 0m0.571s user 0m0.572s sys 0m0.000s	969,797,570 CC real 0m0.601s user 0m0.600s sys 0m0.000s	905,670,260 CC real 0m0.563s user 0m0.564s sys 0m0.000s
SAC (-O3)	1,771,800,100 CC real 0m1.105s user 0m1.108s sys 0m0.000s			

Performance (ratio)



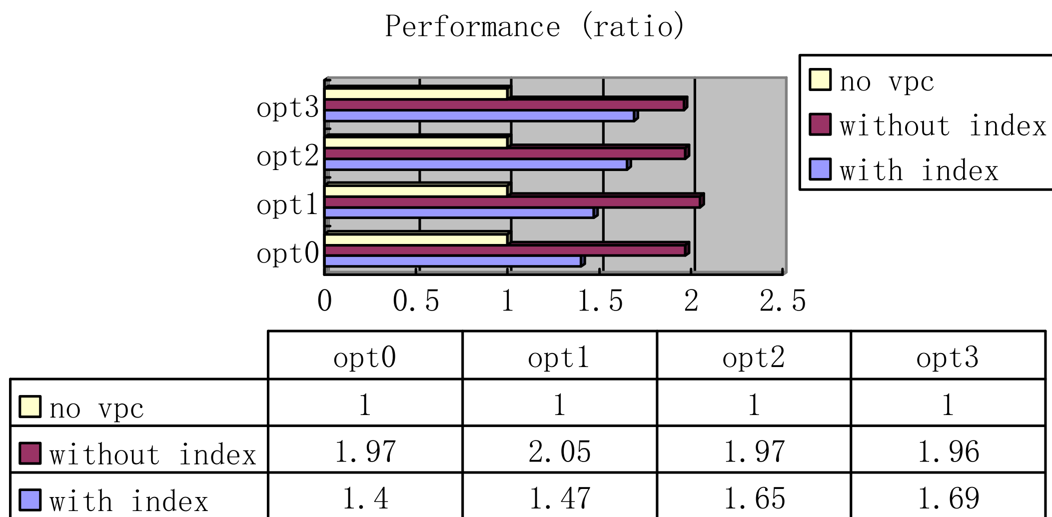
	opt0	opt1	opt2	opt3
no vpc	1	1	1	1
without index	1.8	1.92	1.83	1.96
with index	1.23	1.36	1.37	1.47

Figure 5.7.1 The results (ratio diagram) of Test 6 in hardware environment 1 (Intel). This diagram gives out the performance in each case based on the ratio defined by Equation 5.2 in hardware environment 1. The performance of

original program which is only compiled by Single Assignment C's compiler is defined as '1'.

Results in Hardware Environment 2 (AMD):

-opt	0	1	2	3
VPC with index	1,970,930,246 CC 1.39s user 0.00s system 1.39s elapsed	1,881,962,423 CC 1.34s user 0.00s system 1.37s elapsed	1,676,098,422 CC 1.19s user 0.00s system 1.41s elapsed	1,632,253,810 CC 1.16s user 0.00s system 1.34s elapsed
VPC without index	1,405,581,983 CC 1.02s user 0.00s system 1.31s elapsed	1,346,640,828 CC 0.95s user 0.00s system 1.13s elapsed	1,400,694,611 CC 1.00s user 0.00s system 1.02s elapsed	1,408,732,312 CC 0.97s user 0.00s system 1.13s elapsed
SAC (-O3)	2,762,401,336 CC 1.97s user, 0.00s system, 2.26s elapsed			



	opt0	opt1	opt2	opt3
no vpc	1	1	1	1
without index	1.97	2.05	1.97	1.96
with index	1.4	1.47	1.65	1.69

Figure 5.7.2 The results (ratio diagram) of Test 6 in hardware environment 2 (AMD). This diagram gives out the performance in each case based on the ratio defined by Equation 5.2 in hardware environment 2. The performance of original program which is only compiled by Single Assignment C's compiler is defined as '1'.

Analysis:

This test and the next one are mainly testing the correctness of the different types of with-loop, and also can be considered as an additional examination for the conclusion in test 3, i.e. when the numbers of total calculating operations are the same, the program with bigger arrays' sizes gains fewer enhancements than that with bigger loop size.

In this test, the Single Assignment C's compiler didn't generate "simd" files for all segments of the selection operation in the original program. This is because this version of Single Assignment C's compiler in the test does not consider that with-loop bodies can be vectorized if they contained any selection operation since selection operation may bring in some hazard for vectorization.

The biggest enhancement ratio can be gained by using "–opt1" option of Vector Pascal's compiler. On Intel processor it is 1.96, andon AMD is 2.05.

Test 7

Accompanied by Program 5.7, this program composes another comparison of the effect from loop size and arrays' sizes when the amount of total operations is the same.

```
1  use Structures:all;
2  use SimplePrint:all;
3  int main()
4  {
5      v1=genarray([1000000],5);
6      v2=genarray([1000000],5);
7      for(i=0;i<100;i++)
8
9      {    v1=with(iv)
10     (.<=iv<=[500000]):v1[iv]+v2[iv];
11     ([500001]<=iv<=[900000]):v1[iv]-v2[iv];
12     ([900001]<=iv<=.):v1[iv]+v2[iv];
13         modarray(v1);
14     }
15     res=print(v1[9]);
16     return(res);
17 }
```

Program 5.8.1 Original Single Assignment Code for Test 7. This program is similar to Program 5.7.1, the differences between are the loops' size and the arrays' size.

The corresponding output C files of this program generated by Single Assignment C's compiler are as follows:

```
1  {
2    int s = 1000000;
3    for (i = 0; i < 1000000; ){
4      SACp_emal_502_v1[j] = SACp_emal_504__pinl_179__flat_60;
5      j = j + 1;
6      i = i + 1;
7    }
8  }
```

Program 5.8.2 The for-loop in simd0.c This program is for initializing array v1, array SACp_emal_502_v1 is where actually stores the initial values of the elements..

```
1  {
2   int s = 500001;
3   for (i = 0; i < 500001; ){
4     iv = j;
5     tmp = ( SACp_pinl_544____v1[iv] );
6     tmp = ((tmp)+ (5));
7     SACp_pinl_535__emal_488_v1__SSA0_1[j] = tmp;
8     j = j + 1;
9     i = i + 1;
10  }
11 }
```

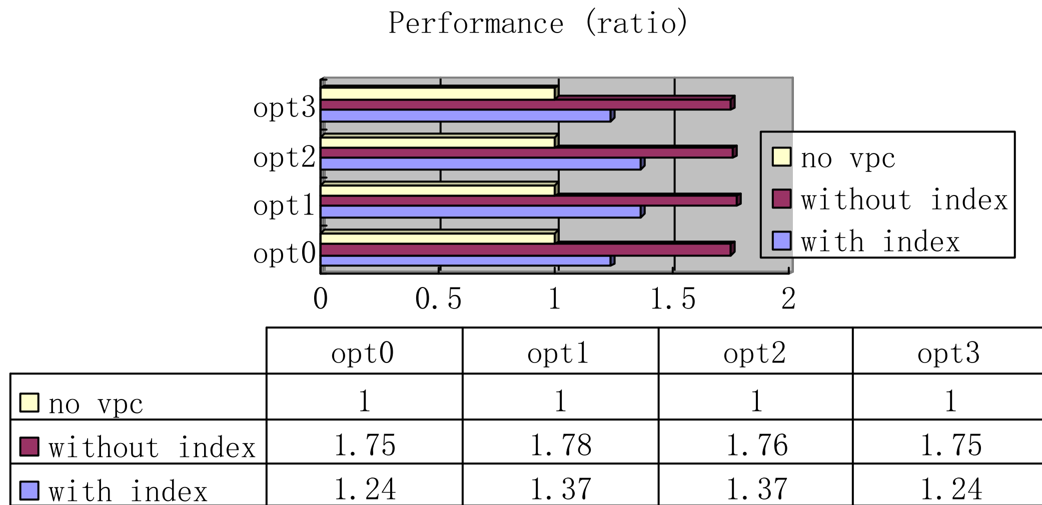
Program 5.8.3 The for-loop in simd1.c This program is related to the first segment of the selection in the original program's with-loop, i.e., it operates on elements from position 0 to 500,000 in the arrays

```
1  {
2   int s = 1000000;
3   for (i = 900001; i < 1000000; ){
4     iv = j;
5     tmp = ( SACp_pinl_544____v1[iv] );
6     tmp = ((tmp)+ (5));
7     SACp_pinl_535__emal_488_v1__SSA0_1[j] = tmp;
8     j = j + 1;
9     i = i + 1;
10  }
11 }
```

Program 5.8.4 The for-loop in simd2.c This program is related to the first segment of the selection in the original program's with-loop, i.e., it operates on elements from position 900,001 to 999,999 in the arrays

Results in Hardware Environment 1:

-opt	0	1	2	3
VPC with index	1,472,222,620 CC real 0m0.952s user 0m0.940s sys 0m0.012	1,325,376,090 CC real m0.802s user 0m0.800s sys 0m0.000s	1,332,333,480 CC real 0m0.831s user 0m0.824s sys 0m0.004s	1,468,904,760 CC real 0m0.918s user 0m0.916s sys 0m0.004s
VPC without index	1,038,516,560 CC real 0m0.641s user 0m0.636s sys 0m0.004s	1,024,017,800 CC real 0m0.632s user 0m0.628s sys 0m0.004s	1,031,908,160 CC real 0m0.637s user 0m0.632s sys 0m0.008s	1,040,199,910 CC real 0m0.639s user 0m0.640s sys 0m0.00
SAC (-O3)	1,818,766,640 CC real 0m1.195s user 0m1.188s sys 0m0.000s			

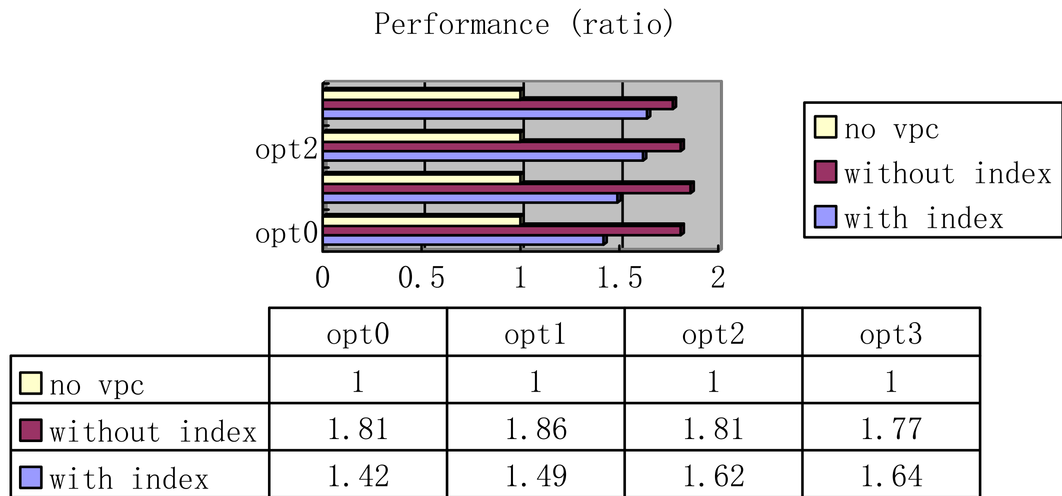


	opt0	opt1	opt2	opt3
no vpc	1	1	1	1
without index	1.75	1.78	1.76	1.75
with index	1.24	1.37	1.37	1.24

Figure 5.8.1 The results (ratio diagram) of Test 7 in hardware environment 1 (Intel). This diagram gives out the performance in each case based on the ratio defined by Equation 5.2 in hardware environment 1. The performance of original program which is only compiled by Single Assignment C's compiler is defined as '1'.

Results in Hardware Environment 2:

-opt	0	1	2	3
VPC with index	2,147,470,176 CC 1.53s user 0.00s system 1.78s elapsed	2,037,839,707 CC 1.47s user 0.01s system 1.94s elapsed	1,881,277,585 CC 1.36s user 0.02s system 1.46s elapsed	1,848,369,703 CC 1.31s user 0.01s system 1.76s elapsed
VPC without index	1,676,940,066 CC 1.20s user 0.00s system 1.57s elapsed	1,635,705,870 CC 1.19s user 0.02s system 1.40s elapsed	1,675,494,990 CC 1.18s user 0.02s system 1.51s elapsed	1,704,561,013 CC 1.14s user 0.01s ystem 1.34s elapsed
SAC (-O3)	3,038,940,634 CC 2.05s user, 0.02s system, 2.42s elapsed			



	opt0	opt1	opt2	opt3
no vpc	1	1	1	1
without index	1.81	1.86	1.81	1.77
with index	1.42	1.49	1.62	1.64

Figure 5.8.2 The results (ratio diagram) of Test 7 in hardware environment 2 (AMD). This diagram gives out the performance in each case based on the ratio defined by Equation 5.2 in hardware environment 2. The performance of original program which is only compiled by Single Assignment C's compiler is defined as '1'.

Analysis:

The biggest enhancement ratio can be gained by using "–opt1" option of Vector Pascal's compiler. On Intel processor it is 1.78, and on AMD is 1.86, both of them are fewer than the results in Test 6 in which arrays have smaller sizes.

5.3.2 Benchmark Test on Floating Point Numbers

We have already seen the speed up in processing 32-bit integers so far; now let's have a look on programs dealing with floating point numbers. In Single Assignment C, the default floating point number has a length of 64-bit, i.e. it is double precision floating point number. Since most of the current x86 processors produced by Intel and AMD provide 128-bit registers, we can let the processors deal with two double precision floating point numbers simultaneously instead of four of integers on those platforms based on the support from Vector Pascal's compiler. In another word, we would expect that the programs processed by my program run twice faster than those only compiled

by using Single Assignment C's compiler.

Following Tests, i.e., Test 8 to Test 10 only performs on hardware environment 1, i.e. on Intel Core Duo processor, because another machine which borrowed from other people is no longer available for me.

Since the Vector Pascal's code generator for GAS (GNU assembler) performs better than the one for NASM when dealing with floating point numbers and gives similar performance when dealing with integers, I changed the back-end assembler to GAS instead of NASM. (We will see the different performances in Test 8).

Test 8

This program is the corresponding floating point version of Test 2. We can make a comparison between their performances.

```
1   use Structures:all;
2   use SimplePrint:all;
3   int main()
4   {
5         v1=genarray([100],2.0);
6      v2=genarray([100],1.0);
7      for(i=0;i<1000000;i++)
8      {
9                v1=with(iv)
10           (.<=iv<=.)  : v1[iv]+v2[iv];
11           modarray(v1);
12        }
13     res=print(v1[0]);
14     return(res);
15  }
```

Program 5.9.1 Original source code in Single Assignment C for Test 8. This one is almost the same to Program 5.3.1, the difference is that in this program, the arrays holds floating point numbers instead of integers.

```
1  {
2    int s = 100;
3    for (i = 0; i < 100; ){
4      SACp_emal_274_v1[j] = SACp_emal_276__pinl_124__flat_48;
5      j = j + 1;
6      i = i + 1;
7    }
8  }
```

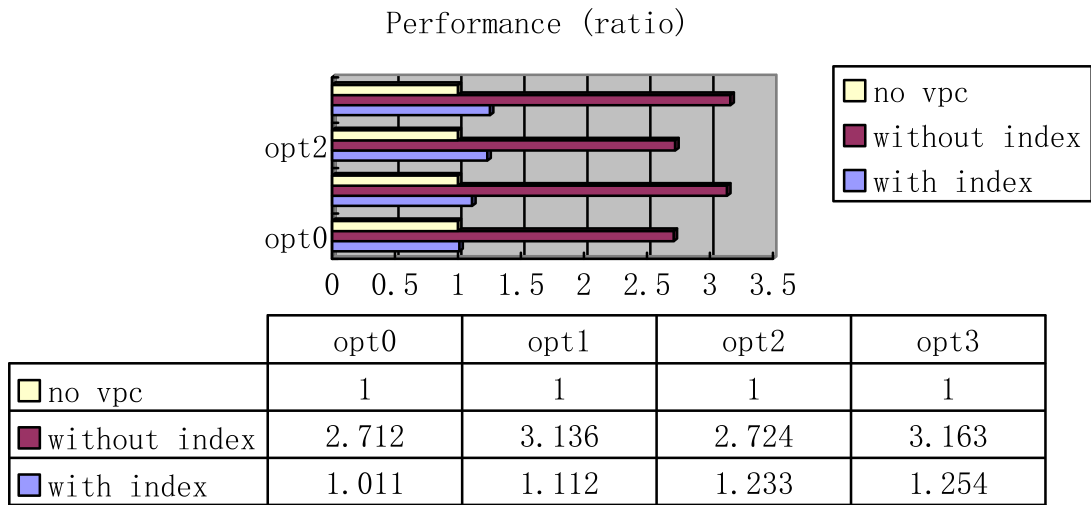
Program 5.9.2 The for-loop in simd0.c This program initializes array v1 by using floating point numbers, the variable SACp_emal_276_pinl__124_flat_48 holds double precision floating point value passed from the main function in a.out.c which uses this "simd0.c" file by "#include" directive. Array SACp_emal_274_v1 is where stores the initial values of the elements of v1.

```
1   {
2     int s = 100;
3     for (i = 0; i < 100; ){
4       iv = j;
5       tmp = ( SACp_pinl_302____v1[iv] );
6       tmp = ((tmp)+ (1.0));
7       SACp_pinl_299__emal_270_v1__SSA0_1[j] = tmp;
8       j = j + 1;
9       i = i + 1;
10    }
11  }
```

Program 5.9.3 The for-loop in simd1.c This program is for adding a floating point number to each elements of v1 which is actually stored in array SACp_pinl_302____v1, and stores the result to a new version of v1 which is SACp_pinl_299__emal_270_v1__SSA0_1 in line 7.

Benchmark Testing Result using GAS:

-opt	0	1	2	3
VPC with index	1,691,447,400 CC real 0m1.059s user 0m1.060s sys 0m0.000s	1,538,109,120 CC real 0m0.963s user 0m0.960s sys 0m0.000s	1,386,932,000 CC real 0m0.841s user 0m0.840s sys 0m0.000s	1,363,343,500 CC real 0m0.841s user 0m0.836s sys 0m0.000s
VPC without index	630,439,700 CC real 0m0.397s user 0m0.396s sys 0m0.000s	545,232,330 CC real 0m0.320s user 0m0.320s sys 0m0.000s	627,824,170 CC real m0.373s user 0m0.372s sys 0m0.000s	540,579,370 CC real 0m0.357s user 0m0.352s sys 0m0.000s
SAC (-O3)	1,709,880,010 CC real 0m1.076s user 0m1.072s sys 0m0.000s			

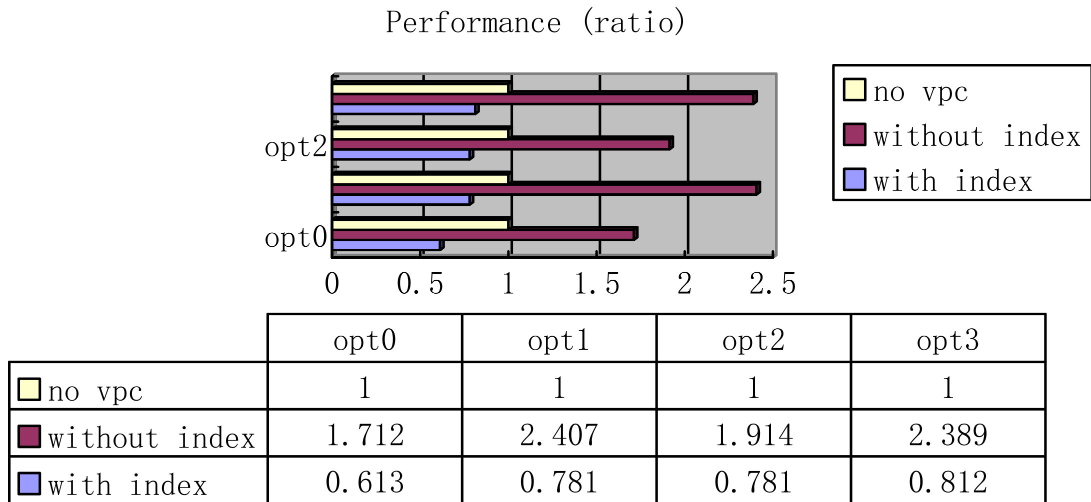


	opt0	opt1	opt2	opt3
no vpc	1	1	1	1
without index	2.712	3.136	2.724	3.163
with index	1.011	1.112	1.233	1.254

Figure 5.9.1 The results (ratio diagram) of Test 8 using GAS. This diagram gives out the performance in each case when using GAS, based on the ratio defined by Equation 5.2. The performance of original program which is only compiled by Single Assignment C's compiler is defined as '1'.

Benchmark Testing Result using NASM:

-opt	0	1	2	3
VPC with index	2,760,002,740 CC real 0m1.707s user 0m1.696s sys 0m0.004s	2,166,847,400 CC real 0m1.282s user 0m1.284s sys 0m0.000s	2,166,443,880 CC real 0m1.362s user 0m1.320s sys 0m0.000s	2,083,185,830 CC real m1.261s user 0m1.248s sys 0m0.004s
VPC without index	987,675,190 CC real 0m0.613s user 0m0.608s sys 0m0.000s	702,604,960 CC real 0m0.460s user 0m0.460s sys 0m0.000s	883,654,170 CC real 0m0.553s user 0m0.552s sys 0m0.004s	707,866,800 CC real 0m0.459s user 0m0.436s sys 0m0.008s
SAC (-O3)	1,691,230,610 CC real 0m1.056s user 0m1.044s sys 0m0.000s			



	opt0	opt1	opt2	opt3
no vpc	1	1	1	1
without index	1.712	2.407	1.914	2.389
with index	0.613	0.781	0.781	0.812

Figure 5.9.2 The results (ratio diagram) of Test 8, using NASM. This diagram gives out the performance in each case when using NASM, based on the ratio defined by Equation 5.2. The performance of original program which is only compiled by Single Assignment C's compiler is defined as '1'.

Analysis:

From these two sheets, we can see that those test programs processed by using GAS run much faster than those processed by using NASM since Vector Pascal's compiler generates more efficient code for GAS. Meanwhile, in Figure 5.9.2, we can see that the results in row of "VPC with out index" are even slower than those in row of "SAC", because that the code generated for NASM by Vector Pascal compiler is not as efficient as it for GAS, and other aspects like function call will also decrease the performance.

By comparing figure 5.3.1 and 5.9.1, we will see that when processing 32-bit integers, the program gains better performance than dealing with floating point numbers by using SIMD instruction set as we assumed.

The biggest enhancement ratio is about 3.163 with optimization option "-opt3" in Vector Pascal.

Test 9

This program is the corresponding floating point version of Test 5. We can make a comparison between their performances.

```
1  use Structures:all;
2  use SimplePrint:all;
3  int main()
4  {
5          v1=genarray([1000000],2.0);
6          v2=genarray([1000000],1.0);
7          v3=genarray([1000000],3.0);
8
9          for(i=0;i<100;i++)
10         {
11                 v1=with (iv)
12                 (.<iv<.) : v1[iv]+v2[iv]*v3[iv];
13                 modarray(v1);
14
15         }
16
17         v3=rotate(0,1,v1);
18         res=print(v3[99]);
19         return(res);
20 }
```

Program 5.10.1 Original Source Code in Single Assignment C for Test 9. This program is the corresponding floating point version of Test 5 (Program 5.6.1), all arrays holds floating point numbers.

The corresponding output C files of this program generated by Single Assignment C's compiler are as follows:

```
1  {
2   int s = 1000000;
3   for (i = 0; i < 1000000; ){
4     SACp_emal_861_v1[j] = SACp_emal_863__pinl_490__flat_48;
5     j = j + 1;
6     i = i + 1;
7   }
8  }
9
```

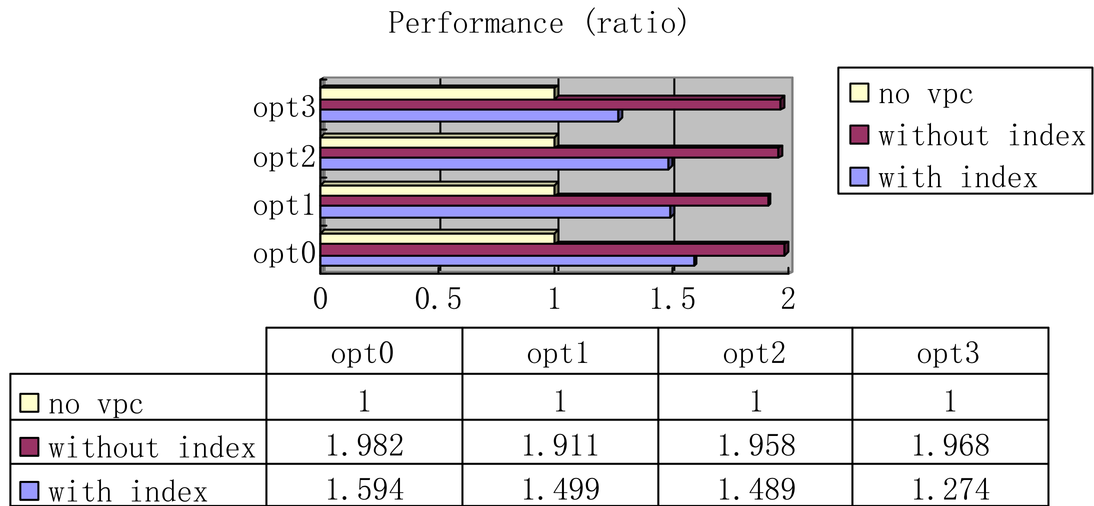
Program 5.10.2 The for-loop in simd0.c. This program initializes array v1 by using variable

SACp_emal_863__pinl_490__flat_48 which holds value of floating point number passed from the main program (a.out.c). Array SACp_emal_861_v1 is where actually stores initial values of v1's elements.

```
1  {
2    int s = 1000000;
3    for (i = 0; i < 1000000; ){
4      iv = j;
5      tmp = ( SACp_pinl_889____v1[iv] );
6      tmp = ((tmp)+ (3.0));
7      SACp_pinl_886__emal_846_v1__SSA0_1[j] = tmp;
8      j = j + 1;
9      i = i + 1;
10   }
11 }
```

Program 5.10.3 The for-loop in simd1.c. This program adds floating point numbers to elements in array v1, which is actually stored in array SACp_pinl_889____v1, and stores the result to a new version of v1 which is SACp_pinl_886__emal_864_v1__SSA0_1 in line 7.

-opt	0	1	2	3
VPC with index	1,537,278,140 CC real 0m0.956s user 0m0.932s sys 0m0.012s	1,314,935,240 CC real m0.794s user 0m0.776s sys 0m0.016s	1,306,790,170 CC real m0.824s user 0m0.780s sys 0m0.032s	1,228,589,980 CC real 0m0.761s user 0m0.744s sys 0m0.016s
VPC without index	994,937,080 CC real 0m0.604s user 0m0.592s sys 0m0.012s	1,000,160,160 CC real 0m0.626s user 0m0.612s sys 0m0.012s	1,024,853,550 CC real 0m0.658s user 0m0.648s sys 0m0.012s	987,990,210 CC real 0m0.619s user 0m0.592s sys 0m0.024s
SAC (-O3)	1,958,325,970 CC real 0m1.211s user 0m1.180s sys 0m0.016s			



	opt0	opt1	opt2	opt3
no vpc	1	1	1	1
without index	1.982	1.911	1.958	1.968
with index	1.594	1.499	1.489	1.274

Figure 5.10 The results (ratio diagram) of Test 9. This diagram gives out the performance in each case based on the ratio defined by Equation 5.2. The performance of original program which is only compiled by Single Assignment C's compiler is defined as '1'.

Analysis:

From the sheet above we can see that there are notable speed-up there, but less than the corresponding one which processing integers (Test 5). Meanwhile, similar toTest 5, since the ratio of the vectorized parts of the program in this test has been decreased; the enhancement is also less than in Test 8. The biggest enhancement ratio is 1.968 with optimization option "-opt3" in Vector Pascal.

Test 10

This program is the corresponding floating point version of Test 7.

```
1   use Structures:all;
2   use SimplePrint:all;
3   int main()
4   {
5
6           v1=genarray([1000000],2.0);
7           v2=genarray([1000000],1.0);
8           for(i=0;i<100;i++)
9           {
10                  v1=with (iv)
11                  (.<=iv<=[500000]) : v1[iv]+v2[iv];
12                  ([500001]<=iv<=[900000]):v1[iv]-v2[iv];
13                  ([900001]<=iv<=.):v1[iv]+v2[iv];
14                  modarray(v1);
15
16          }
17
18
19          res=print(v1[9]);
20          return(res);
21  }
```

Program 5.11.1 Original source code in Single Assignment C for Test 10. This one is the corresponding floating point version of Test 7, all arrays holds floating point numbers here.

The corresponding output C files of this program generated by Single Assignment C's compiler are as follows:

```
1   {
2    int s = 1000000;
3    for (i = 0; i < 1000000; ){
4      SACp_emal_505_v1[j] = SACp_emal_507__pinl_182__flat_48;
5      j = j + 1;
6      i = i + 1;
7    }
8   }
```

Program 5.11.2 The for-loop in simd0.c. This program is for initializing v1 by using variable SACp_emal_507__pinl_182__flat_48 which accepts floating point values from the main program (a.out.c). Array SACp_emal_505_v1 is where actually stores initial values of v1's elements.

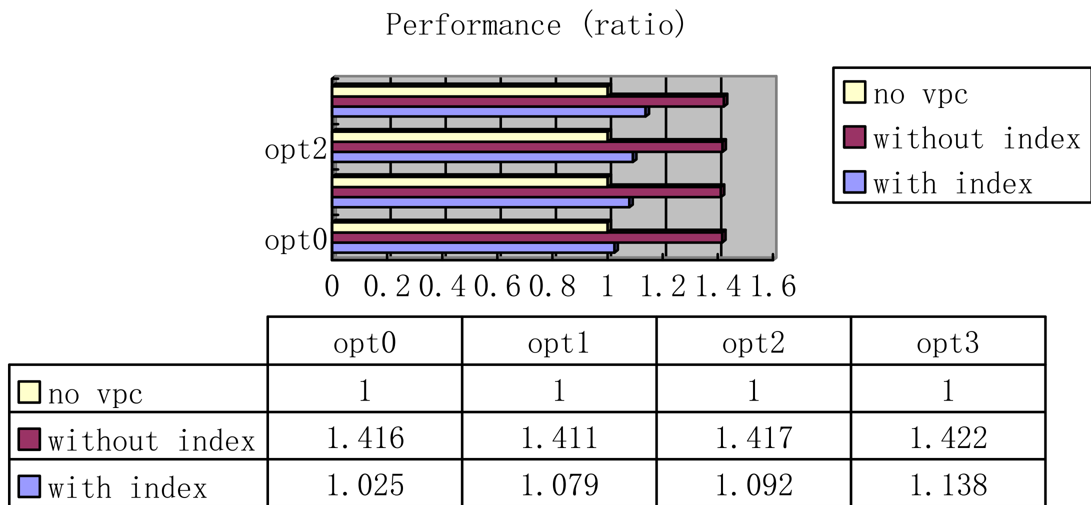
```
1   {
2     int s = 500001;
3     for (i = 0; i < 500001; ){
4       iv = j;
5       tmp = ( SACp_pinl_547____v1[iv] );
6       tmp = ((tmp)+ (1.0));
7       SACp_pinl_538__emal_491_v1__SSA0_1[j] = tmp;
8       j = j + 1;
9       i = i + 1;
10    }
11  }
```

Program 5.11.3 for-loop in simd1.c. This program adds floating point value to elements in array v1's first 500,000 positions, which is actually stored in array SACp_pinl_547____v1, and stores the result to a new version of v1 which is SACp_pinl_538__emal_491_v1__SSA0_1 in line 7.

```
1   {
2     int s = 1000000;
3     for (i = 900001; i < 1000000; ){
4       iv = j;
5       tmp = ( SACp_pinl_547____v1[iv] );
6       tmp = ((tmp)+ (1.0));
7       SACp_pinl_538__emal_491_v1__SSA0_1[j] = tmp;
8       j = j + 1;
9       i = i + 1;
10    }
11  }
```

Program 5.11.4 The for-loop in simd2.c. This program is adding floating point value to elements in positions from 900,001 to 999,999 in array v1, which is actually stored in array SACp_pinl_547____v1, and stores the result to a new version of v1 which is SACp_pinl_538__emal_270_v1__491_1 in line 7.

-opt	0	1	2	3
VPC with index	1,771,343,050 CC real 0m1.074s user 0m1.068s sys 0m0.004s	1,683,709,520 CC real 0m1.059s user 0m1.048s sys 0m0.004s	1,662,536,180 CC real 0m1.057s user 0m1.008s sys 0m0.024s	1,595,916,440 CC real 0m0.975s user 0m0.964s sys 0m0.012s
VPC without index	1,283,084,730 CC real 0m0.789s user 0m0.784s sys 0m0.000s	1,287,061,650 CC real 0m0.847s user 0m0.828s sys 0m0.008s	1,282,242,890 CC real 0m0.787s user 0m0.780s sys 0m0.008s	1,277,165,630 CC real 0m0.775s user 0m0.768s sys 0m0.008s
SAC (-O3)	1,816,316,280 CC real 0m1.162s user 0m1.148s sys 0m0.012s			



	opt0	opt1	opt2	opt3
no vpc	1	1	1	1
without index	1.416	1.411	1.417	1.422
with index	1.025	1.079	1.092	1.138

Figure 5.11 the results (ratio diagram) of Test 10. This diagram gives out the performance in each case based on the ratio defined by Equation 5.2. The performance of original program which is only compiled by Single Assignment C's compiler is defined as '1'.

Analysis:

We can see that the biggest enhancement ratio is 1.422 with optimization option "-opt3" in Vector Pascal.

5.4. Conclusion

The benchmark results illustrate that when programs processing integers and floating point numbers, we can normally get notable speed up from vectorization over using GCC to generate machine code without any SIMD utilization. And further optimized code generated by the third stage of my program can gain more enhancements.

The improvement also depends on the tested program, i.e., the performance will be better if there are fewer operations which do not act on the arrays. Besides, when the total number of operations on elements is fixed, code with bigger loop size and smaller array size beats the one with smaller loop size and bigger array size.

Consequently, we can say that the benchmark test results generally prove the hypothesis in the beginning, and in some cases the enhancements are even bigger than what we initially expected (the details of the reasons have been explained in analyses of these cases).

Chapter 6

Conclusion and Future Works

This chapter concludes the research presented in the dissertation. The first part gives a review on the research, which is combining two array languages together. The second part discusses possible future work that may further expand the works I have done and extend the applied area.

6.1 Research Summary

The research in this thesis is basically relative to the front-end design & implementation of the compiler, which includes analyzing and designing grammar to build syntax tree, transforming the source code from C to Pascal, vectorization & loop-unrolling and some other optimizations.

6.1.1 Analyzing and Designing the Grammar

Since there are no documents providing specifications to line out possible activities of the files generated by Single Assignment C's compiler, my work generally based on the assumption of that it contains almost all actions might exist in a pure for-loop of C programming language, except those actions may prevent vectorization, including function call, "label" and "jump" statements, because as mentioned in Chapter 3, Single Assignment C can find out proper innermost for-loops for vectorization and out put them in C format (Single Assignment C employs GCC as its backend). Moreover, the definition of function is also ruled out because only innermost loops will be processed.

As I discussed in both Chapter 4, my grammar is a subset of standard C grammar, but it may still be a superset of practical grammar of the C files from Single Assignment C's compiler which will be processed by my program. This will not cause any problem, but it may bring some redundant factors into the program. Though these factors will not affect runtime performance, it is still a good thing to reduce the system's size and the time spent in compile-time.

6.1.2 Transforming to Pascal Code

This part of work is the foundation of the whole system. Though the following work such as vectorization and loop unrolling were not based on the syntax tree of Pascal, this part of the work provides basic support for tree traversal and working classes. It is a tedious job to find out the exact nodes which need to be processed during traversal on the syntax tree because a concrete syntax tree normally has too much information of the input source program. As I explained in chapter 4, I didn't use SableCC to build an abstract syntax tree. There is a lot of work that needs to be done for updating all working classes based on concrete syntax tree to those based on an abstract syntax tree, which almost amount to a rewrite of the whole program and a fundamental change in its architecture.

6.1.3 Vectorization and Loop Unrolling

My program employs forward substitution to implements the data flow optimizations including constant propagation & folding and copy propagation, and also to eliminate data dependences which may exist in the files from Single Assignment C's compiler.

Unlike loop unrolling technique used commonly in other compilers, my program transforms the loops entirely to the straight vector codes as part of the vectorization work, instead of loops with bigger iteration sizes. Meanwhile, it also updates the operations on the arrays' elements to operations on the whole arrays during pursuing the goal of vectorization.

As explained in chapter 4, these optimizations didn't provide great enhancement due to the internal mechanism of Vector Pascal's compiler. That's why I develope the third stage of my program, to achieve better performance by removing the arrays indices.

6.1.4 Benchmark Test

Since the interface between Single Assignment C and Vector Pascal has been built, I designed a series programs to manipulate my program and the two languages, and do benchmark test on them to valid my hypothesis in the beginning of my research.

In this part, my programs employ two concepts of time measuring. One is using "time" command to

make an intuitive view of the enhancement and to examine the effect of tuning the threads in the system. Another is implemented by using RDTSC instruction on x86 platforms for accurate timing. From the ten tests' results, including tests of operations on arrays of integers and floating points, we can see that compared with original Single Assignment C's program, notable enhancement ratios of runtime performance have been gained by vectorizing operations on arrays and employing Vector Pascal's compiler as the code generator for them, especially in using further optimized Vector Pascal program. In some cases in which almost all operations contained in the tested program are acting on arrays, the enhancement ratios are even higher than the value I expected. The reasons of it have been explained in related analyses in Chapter 5.

6.1.5 Conclusion

After about 15 months work, the project of building an interface between Single Assignment C and Vector Pascal is finished. By the benchmark test, we can see that the hypothesis in the beginning is generally validated, i.e., the program' performance has been greatly enhanced by using the Vector Pascal code generator for Single Assignment C, and in some sense the performance may be even better than what I expected in some case. These facts prove that the combination of these two programming languages is significant and useful not only theoretically but also in practice. Meanwhile, if the techniques have further developed such as implementing bigger registers in processors and Vector Pascal's compiler provides architecture specified optimization for them, the enhancement will be also further improved automatically without any work on current programs.

6.2 Future Works

6.2.1 Updates and Modification

Though the benchmark test results seem to be perfect and it is reasonable to think that on every array operation we can enjoy some enhancement of performance, there is still one little "disharmony noise". That is, the Single Assignment C's compiler is still "pre-version" software which means that it is still a beta version and the final one has not yet been published. Therefore some bugs are possibly hiding inside it and sometimes the "-simd" option will crash the compiler. And the behavior of this compiler is also various between different beta versions. That means, once the developers of

Single Assignment C publish the final release version, i.e. Single Assignment C 1.0, some modifications in the system of my program might be necessary.

6.2.2 Multi-Threading

Moreover, Single Assignment C also provides support for thread level parallelism with the help of POSIX Threads. In recent two years, most main stream manufactures of processors are focusing on producing multi-core processors. This kind of architecture almost dominates the current market and will be the future trend in the next several years. Thus multi-threading becomes more and more important because the threads are no longer only run in turn on same processing units sharing limited resources. Though this thesis doesn't include thread level parallelism by using Single Assignment C and Vector Pascal, it would be worthwhile to try to extend the research to this area in future work, i.e., if we have a big scientific computing problem with many arithmetic tasks, we may be able to use Single Assignment C to implement the main algorithm and slice it into many smaller tasks, let the operating system map them on different threads to run on different cores to gain thread level parallelism. Meanwhile, we can use Vector Pascal to deal with each small task with the help from SIMD features to gain better performance.

6.2.3 Migrating to Other Platforms

As the developers of Single Assignment C claims, it will also support other platform including:

- Solaris / Sparc
- Solaris / x86
- Linux / x86
- DEC Alpha
- Mac OS X (ppc)

Meanwhile, Vector Pascal is developed in using Java programming language, so it should be able to run on all platforms supported by Java Virtual Machine. Though it currently only provides architecture specified optimizations on x86 and PowerPC, Vector Pascal will also be port to the famous machine "Play Station 3" produced by Sony Corp which is using IBM "Cell" processor.

Since it has been proved by the benchmark testing result that the combination of these two array programming languages will let programmers enjoy better performance, it would be advisable to migrate the whole system to every platform supported by these two programming languages. Although I also used Java to implement the interface between them, the shell script controlling system is not universal in different Operating System. Thus the programs in this system need to be reorganized based on different practical situations due to what Operating System they are ported to.

6.3 Generalization of the Research

The core component of the research is the vectorization, i.e. generating vectorized Pascal code for Vector Pascal to utilize its highlight features of using SIMD instruction sets to perform array operations. By combining the two array languages, Single Assignment C and Vector Pascal together to deal with problems demanding array operations, we can gain better performance than using them separately， as data sheets and diagrams are shown in chapter 5. Thus it is reasonable to suppose that we can employ them in many situations relative to array calculations, such as multi-media problems and scientific arithmetic problems, especially on modern processors which implement design concepts of and SIMD instruction set.

Appendix

Assembly Files Generated by GCC and Vector Pascal

1. The assembly file generated by GCC

```
	.file	"1.c"
	.text
.globl foo
	.type	foo, @function
foo:
	pushl	%ebp
	movl	%esp, %ebp
	subl	$16, %esp
	movl	$0, -4(%ebp)
	movl	$0, -4(%ebp)
	jmp	.L2
.L3:
	movl	-4(%ebp), %eax
	sall	$2, %eax
	movl	%eax, %edx
	addl	12(%ebp), %edx
	movl	-4(%ebp), %eax
	sall	$2, %eax
	addl	8(%ebp), %eax
	movl	(%eax), %eax
	addl	$1, %eax
	movl	%eax, (%edx)
	addl	$1, -4(%ebp)
.L2:
	cmpl	$999, -4(%ebp)
	jle	.L3
	movl	12(%ebp), %eax
	movl	(%eax), %eax
	leave
	ret
	.size	foo, .-foo
	.ident	"GCC: (GNU) 4.1.2 (Ubuntu 4.1.2-0ubuntu4)"
	.section	.note.GNU-stack,"",@progbits
```

2. The assembly file generated by Vector Pascal:

```
%include "/home/lee/MyProgram/vpc/macros.asm"
 times 256db ' ';0
      GLOBAL testlabel113fd6e27a11b;0
      GLOBAL testlabel113fd6e27a017;0
section .text;0
; procedure generated by code generator class ilcg.tree.PentiumCG;0
      GLOBAL test_foo;0
label113fd6e2793b:;0
test_foo:;0
; entering a procedure at lexical level 1;0
 enter spacefortest_fool0-4*1,1;0
      push ebx;0
      push esi;0
      push edi;0
; #8;0
; #substitute in (ref int32)mem(+(^(ebp),-8)):ref int32;0
; # use register eax;0
 xor eax,eax;0
 label113fd6e29bc2d:;1
cmp DWORD eax, 999;1
 jg nearlabel113fd6e29bc2f;1
 mov     ebx,DWORD[ebp+12];1
 mov     edi,DWORD[ebp+8];1
 movq    MM4, [edi+ eax* 4];1
 paddd   MM4, [testlabel113fd6e27a11b];1
 movq    [ebx+ eax* 4],MM4;1
 mov     ebx,DWORD[ebp+12];1
 mov     edi,DWORD[ebp+8];1
 movq    MM4, [edi+ eax* 4+8];1
 paddd   MM4, [testlabel113fd6e27a11b];1
 movq    [ebx+ eax* 4+8],MM4;1
 mov     ebx,DWORD[ebp+12];1
 mov     edi,DWORD[ebp+8];1
 movq    MM4, [edi+ eax* 4+16];1
 paddd   MM4, [testlabel113fd6e27a11b];1
 movq    [ebx+ eax* 4+16],MM4;1
 mov     ebx,DWORD[ebp+12];1
 mov     edi,DWORD[ebp+8];1
 movq    MM4, [edi+ eax* 4+24];1
 paddd   MM4, [testlabel113fd6e27a11b];1
 movq    [ebx+ eax* 4+24],MM4;1
 mov     ebx,DWORD[ebp+12];1
 mov     edi,DWORD[ebp+8];1
```

```
  movq     MM4, [edi+ eax* 4+32];1
  paddd    MM4, [testlabel113fd6e27a11b];1
  movq     [ebx+ eax* 4+32],MM4;1
  lea eax,[eax+10];1
  jmplabel113fd6e29bc2d;1
  label113fd6e29bc2f:;1
; #9;1
  label113fd6e279315:;1
spacefortest_fool0 equ 28;1
test_fool0exit:;1
       pop edi;1
       pop esi;1
       pop ebx;1
leave;1
  EMMS
ret 0;4
section .text;0
; procedure generated by code generator class ilcg.tree.PentiumCG;0
label113fd6e288f1f:;0
;        unit$test;0
; entering a unit at lexical level 0;0
unit$test:;0
  enter0,0;0
  call unit$system;4
cmp byte[unit$testready],1;0
  jnz unit$testinit;0
  jmp unit$testl1exit;0
unit$testinit:mov byte[unit$testready],1;0
  label113fd6e2793d:;0
spaceforunit$testl1 equ 0;0
unit$testl1exit:;0
       pop edi;0
       pop esi;0
       pop ebx;0
leave;0
  ret 0;4
       SECTION .bss;0
       alignb 16;0
resb spaceforunit$testl1;0
       alignb 16;0
  label113fd6e253c3:;0
       SECTION .data;0
unit$testready dd 0;0
  testlabel113fd6e27a11b:;0
  dd1;0
  testlabel113fd6e27a017:;0
```

```
  dd1;0
%include "/home/lee/MyProgram/vpc/systemPentium.asm"
section .data
```

Bibliography

1. [Aho, 1988] Alfred.V.Aho, Ravi Sethi, and J.D.Ullman. *Compilers—Principles, Techniques, and Tools*, Addison Wesley Publishing Company, 1988
2. [Appel, 2003] A.W.Appel & J.Palsberg. *Modern Compiler Implementation in Java, Second Edition,* Cambridge University Press, 2003
3. [Baxter, 1989] William Baxter and Henry R. Bauer, III. "The program dependence graph and vectorization" *Annual Symposium on Principles of Programming Languages; Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1989, NY
4. [Cardelli, 1985] L.Cardelli and P.Wegner. "On Understanding Types, Data Abstraction, and Polymorphism", *Computing Surveys*, Vol. n. 4, November 1985, pp 471-522
5. [Chakravarty, 2000], Manuel M. T. Chakravarty, "The Haskell Ports Library", 15 July 2000 http://www.cse.unsw.edu.au/~chak/haskell/ports/
6. [Cockshott, 2004] W.P.Cockshott. *SIMD Programming Manual for Linux and Windows,* Springer Press, 2004
7. [Eichenberger, 2004]Alexandre E. Eichenberger, Peng Wu Kevin O'Brien. "Vectorization for SIMD architectures with alignment constraints", *ACM SIGPLAN Conference on ProgrammingLanguage Design and Implementation*, Vol. 39, Issue 6, pp 82-93,May 2004
8. [Gagnon, 1998]Etienne Gagnon. Master Thesis *SableCC, an Object-Oriented Compiler Framework*. March 1998
9. [Goldberg,1996]Benjamin Goldberg, Functional Programming Languages. 1996 ACM Computing Surveys, Vol. 28, No. 1, March 1996
10. [Grelck, 2001]Clemens Grelck. Doctor Dissertation, *Implicit Shared memory Multiprocessor Support for the Functional Programming Languages SAC-Single Assignment C*, January 2001
11. [Hennessy & Patterson, 2003] John L. Hennessy & David A.Patterson, *Computer Architecture: a Auantitative Approach, Third Edition*, Elsevier Science Pte Ltd. 2003
12. [Hughes, 1989] John Hughes, "Why Functional Programming Matters", *The Computer Journal*, Vol 32, pp 98-107, April 1989
13. [IEEE POSIX, 1995] IEEE. *IEEE POSIX 1003.1c Standard* 1995, IEEE, NY
14. [Intel, 1997], Intel Corp. *Using the RDTSC Instruction for Performance Monitoring*,

http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM 1 .HTM, 1997

15. [Intel, 2001] Intel Corp. *Desktop Performance and Optimization for Intel® Pentium® 4 Processor*, Feb 2001, http://www.intel.com/design/pentium4/papers/249438.htm
16. [ISO, 1999] ISO. "Programming languages – C". *INTERNATIONAL STANDARD © ISO / IEC 9899:1999* http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=29237
17. [Iverson, 1962] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons,Inc., 1962
18. [Iverson, 1979] Kenneth E. Iverson. "Notation as A tool of Thought", *1979 ACM Award Lecture*, 1979
19. [Iverson Software Inc., 1996] Iverson Software Inc. "About J", 1996 http://www.cs.trinity.edu/Other_Attractions/j.html/
20. [Lowney, 2006] Geoff Lowney, "Software for multi-core processors" Lecture at Tsinghua University, Nov 2006,
21. [Paul,2004] Paul E. Black, "tail recursion", *NIST Digital Library*. http://www.nist.gov/dads/HTML/tailrecursn.html, December 2004
22. [PThreads Programming, 2006] "POSIX Threads Programming", *Lawrence Livermore National Laboratory*, 2006. https://computing.llnl.gov/tutorials/pthreads/
23. [Raymond, 2000] D.J.Raymond. SISAL: "A Safe and Efficient Language for Numerical Calculations", *Linux Journal*, Vol. 2000, Issue 80es, November 2000
24. [Scholz, 1994] Sven-Bodo Scholz. "Single Assignment C- Functional Programming Using Imperative Style", *In Proceedings of the 6th International Workshop on Implementation of Functional Languages (IFL'94)*,Norwich, England, UK, University of East Anglia, 1994.
25. [Scholz, 2003] Sven-Bodo Scholz. "Single Assignment C -- Efficient Support for High-level Array Operations in a Functional Setting", *Journal of Functional Programming*, Vol.13,2003, pp. 1005-1059
26. [SAC,2006] "About SAC". May 2006 http://sac-home.org/index.php?p=.%2F11_About%2F11_About_SAC.
27. [Tang,1995] Peiyi Tang, "Vectorization beyond Data Dependences", *Proceedings of the 9th international conference on Supercomputing*, 1995, Barcelona, Spain