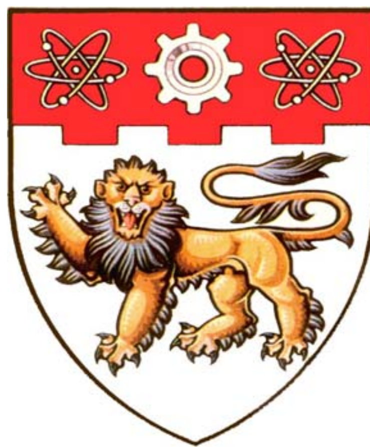
ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

POWER REDUCTION OF DATA-INTENSIVE MULTIMEDIA EMBEDDED SOFTWARE THROUGH SOURCE-LEVEL TRANSFORMATION



LI SHAN

SCHOOL OF COMPUTER ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY

2005

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Power Reduction of Data-Intensive Multimedia Embedded Software Through Source-Level Transformation

Li Shan

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in fulfillment of the requirement for the degree of
Master of Engineering

2005

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Acknowledgements

First of all, I would like to express my greatest gratitude to my school supervisor, Associate Professor Lai Ming-Kit Edmund, for his advice and support in the past years. I am very thankful to his guidance and patience. His thorough advice helps me clarify the design. He has taught me much about many different aspects of research, and has exhibited the meaning of research. In particular, this thesis would not be possible without his help and support. His inspiration will last far beyond this project.

I would like to thank my other supervisor, Mr. Mohammed Javed Absar, from STMicroelectronics Asia Pacific Pte. Ltd. He has been supervising me closely throughout this project, and has put in a lot of effort into guiding me in technical aspects. His valuable advice helps me solve the problems encountered during the progress. He has always been so supportive that makes the work enjoyable.

I would also like to thank Mrs. Sapna George, the Technological Manager of STMicroelectronics Asia Pacific Pte. Ltd, for creating a productive research environment and helping me keep on progressing.

Special thanks go to STMicroelectronics Asia Pacific Pte. Ltd. for providing me with the top-up scholarship and continuous support.

Lastly but importantly, I would like to thank the technicians, Ms. Chua Poo Hua and Ms. Oh Hwee May, in the Centre for Multimedia and Network Technology for their assistances in many aspects.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Contents

Acknowledgements . . . 1
List of Figures . . . 4
List of Tables . . . 6
Abstract . . . 8

1 Introduction **10**
1.1 Objectives . . . 12
1.2 Original Contributions . . . 13
1.3 Thesis Organization . . . 14

2 Custom Memory Management Source-Level Transformations **16**
2.1 Custom Memory Management . . . 16
2.2 High-level Transformations . . . 19
2.2.1 Loop Fusion (Merging) . . . 19
2.2.2 Loop Fission (Splitting) . . . 20
2.2.3 Loop Interchange . . . 21
2.2.4 Unroll-And-Jam . . . 21
2.2.5 Scalar Replacement . . . 23
2.3 Data Dependency . . . 23
2.3.1 Four Types Of Dependency . . . 24
2.3.2 Lexicographic Order And Transformation Legality . . . 24
2.3.3 Loop-Carried, Loop-Independent And Distance Vector . . . 26
2.3.4 Loop-Invariant Reference And Reference-Invariant Loop . . . 27

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

2.3.5 Self-Temporal Reuse And Group-Temporal Reuse 28

3 Memory Access Reduction 30

3.1 WB-AMR Speech Decoder 31

3.1.1 Principles Of WB-AMR Speech Decoder 32

3.1.2 Implementation of WB-AMR Speech Decoder 33

3.2 Profiling 35

3.2.1 Profile Of WB-AMR Decoder 36

3.2.2 Analysis 36

3.3 Inlining 40

3.4 Global Transformation 43

3.4.1 Inlining And Loop Merging 43

3.4.2 Loop Merging And Scalar Replacement 48

3.4.3 Loop Unrolling And Scalar Replacement 50

3.4.4 Other Global Transformations 52

3.5 The Profiling Results Of The Optimized WB-AMR Speech Decoder . . . 55

3.6 Summary 58

4 Memory Access Reduction For Multiple-Index Subscripted References 59

4.1 Properties Of Multiple-Index Subscripted References 61

4.2 Detection Process 64

4.3 Legality Test 66

4.4 Loop Interchange And Related Tests 67

4.5 Unroll-And-Jam 68

4.5.1 Parameters Estimation For Optimal Unrolling 69

4.5.2 Unrolling vector 73

4.6 Scalar Replacement 75

4.7 Complexity 78

4.8 Signal Processing Benchmarks 78

4.8.1 Performance 79

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

5 Conclusions **85**

5.1 Future Work 86

References **87**

A Optimizations to WB-AMR Speech Decoder **91**

B List of Author's Publications **102**

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

List of Figures

2.1 The DTSE Methodology [1] 17
2.2 Before loop fusion 20
2.3 After loop fusion 20
2.4 Before loop fission 21
2.5 After loop fission 21
2.6 Before loop interchange 21
2.7 After loop interchange 22
2.8 unroll-and-jam 22
2.9 Before scalar replacement 23
2.10 After scalar replacement 23
2.11 Loop-carried dependence 27
2.12 Loop-independent dependence 27
2.13 Loop-invariant references 28
2.14 Self-temporal reuse 28

3.1 The optimization procedure 31
3.2 Detailed block diagram of the ACELP decoder 32
3.3 Simplified program of WB-AMR speech decoder 34
3.4 Memory-access profiling process using Atomium 36
3.5 The simplified synthesis function 41
3.6 The simplified decoder function 42
3.7 The *Syn_filt_32* function 44

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

3.8 The *Deemph_32* function 44

3.9 The *HP50_12k8* function 45

3.10 After inlining and loop merging 47

3.11 Code segment for loop merging and scalar replacement 48

3.12 After loop merging 49

3.13 Forward prediction operation 50

3.14 Forward prediction operation loop nest after unrolling 51

3.15 Common operation - *Dot_product* 53

3.16 After transformation 53

3.17 Redundant copy of data array 54

3.18 After transformation 54

4.1 Examples of two kinds of references. 60

4.2 The flow of the overall process to exploit Multiple-index subscripted variable 61

4.3 General form of multiple-index subscripted reference $x[\vec{f}(\vec{I})]$ in a nest loop. 62

4.4 Examples of two kinds of self-temporal reuse. 63

4.5 Reuse within one iteration and reuse across iterations. 70

4.6 Algorithm of selecting optimal unrolling vector. 74

4.7 Scalar replacement after unroll-and-jam. 77

4.8 A for loop in *dsp_autocor* 82

4.9 *dsp_minerror* 84

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

List of Tables

3.1 List of arrays sorted by number of accesses 37
3.2 Array *old_exc* [*dec_main.c*:126] Global call 38
3.3 Array *Aq* [*dec_main.c*:129] Global call trace 39
3.4 Major function for memory accesses 39
3.5 List of arrays sorted by number of accesses 56
3.6 Results on the improvements due to reduction of multiple-index subscripted references 57
3.7 Reduction in memory accesses for different modes 57

4.1 Benchmarks from signal processing library 79
4.2 Number of loads and stores of original code 80
4.3 Results after transformations 80
4.4 Selection of unrolling vector 83
4.5 Comparison of estimation and simulation 83

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Abstract

This thesis describes techniques by which the power consumed during the execution of data-intensive application software is minimized. Power consumption has been an increasingly important factor in the cost measurement of embedded systems. For signal processing applications, data memory access accounts for a large amount of power consumption. For this reason, various methods have been proposed to reduce the data memory power usage. Among these methods, high-level compiler optimization, which has been found to be effective, has become the focus of recent research.

In our project, we aim to reduce memory access related power consumption for data-intensive applications. The means to achieve it is to reduce memory access and data transfer for this kind of software. The technique chosen is high-level source-to-source transformations. In the thesis, we present a high-level optimization procedure we have developed, which involves profiling, inlining and global transformation. These three steps serve to reduce data access of array references inside nested loops. The effectiveness of the procedure is demonstrated by applying it to an industrial application – Wideband Adaptive Multi-rate (WB-AMR) speech decoder.

During the optimization of the WB-AMR speech decoder, a large number of data accesses are generated by multiple-index subscripted references. Previous published works only deal with single-index subscripted reference and their methods are not directly applicable to multiple-index subscripted references. A new method is developed in this thesis to reduce this kind of references by improving register allocation for them. Com-

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

paring to the previous works, our method is able to extend the optimization scope for register allocation. The method developed should also be applicable to other multimedia application programs that process large amounts of data in the form of multidimensional arrays in nested loops.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Chapter 1

Introduction

Embedded systems have always been very cost-sensitive. In particular, power consumption is becoming an increasingly important contributing factor to the cost of embedded systems. One of the reasons is that many of these systems are now required to be portable. Portable devices require long battery life and light battery packs, driving the system design towards low power. Moreover, high power consumption also means more costly packaging and cooling requirements and lower reliability.

Traditionally, power reduction is achieved entirely through the use of application specific integrated circuits that incorporates low-power hardware designs. However, with the availability of high performance embedded processors, computationally demanding functions such as realtime signal processing are increasingly being realized in software. We could analyze the problem at several levels and devise strategies for power reduction that takes into account one or more of these levels [2]. Low-level techniques like voltage scaling, clock frequency scaling, clock-gating [3] and pipeline gating [4] have been known to be very useful. At the application level, source code transformation using optimizing or restructuring compilers have been gaining prominence lately [5, 6]. The advantage of optimizing compilers lies in the fact that they are not application specific. However, different classes of applications and different kinds of target platforms require slightly

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

different types of optimizations. A scientific application, which involves large amounts of computations, needs an optimizing compiler that could parallelize or vectorize the application (e.g. compilers for Cray-T4). Applications running on networked computers require optimization for minimum message passing. For embedded systems, especially those running multimedia applications, the requirement is for methods that optimize data transfers. This is the focus of this thesis.

Compiler optimizations can broadly be divided into two levels. At the lower level, the translation is more hardware-dependent, which limits the usage of the compiler to a certain class of processors. On the other hand, higher level compiler optimization techniques perform source-to-source transformations. Typically, loop transformations and data transformations are performed on the source codes, with the output in the same programming language as the input source. Considerable research has been done in the area of loop and data transformations for data locality and increased parallelism. A good example is the decade-long SUIF project [7, 6] at Stanford University. The main objective of our work, however, is in applying these high-level compiler techniques for the purpose of reducing embedded software power consumption.

Many embedded applications, both in multimedia and telecommunication applications, generally involve accessing large amounts of data. Experiments show that for both hardware and software realizations, data transfer and memory access operations consume much more power than data-path operations [8, 1, 9, 10]. Therefore, reducing data memory access will directly lead to a reduction of power consumption. Since most data memory accesses for these applications involve data arrays within (nested) loops, there is a need to find ways to minimize them. This can be achieved through transformations of the source code so that multiply accessed data can be stored in registers, minimizing redundant memory accesses. The procedure we developed, which is presented in Chapter 3, involves profiling, inlining and global transformation. Comparing to lower level

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

optimizations techniques, high-level optimization procedure on program source have the following advantages:

- From system design point of view, data flow in higher level of the design (e.g. source code) is easier to understand and the exploration space is larger. Thus it is more efficient to minimize data accesses from high-level.
- The high-level optimizations on source code is nearly independent of the target architecture. Thus it only has to be executed once during the complete design trajectory in embedded system design irrespective of the number implementation platforms that is considered. For example, the reduction of temporal reuses is inherent to the computation in an application regardless of the target platform.
- The results obtained by the platform independent optimization procedure can be further improved in platform dependent optimizations. That is done by adapting the source code to the characteristics of the underlying (platform) memory organization.

Furthermore, data dependence tests during high-level transformation guarantees the correctness and robustness of the technique.

1.1 Objectives

This project aims to develop an automated optimization method to perform source-to-source transformations that reduce the number of memory access for data-dominated applications. This project is partially supported by STMicroelectronics Asia Pacific Pte. Ltd. (STM). In the development of the method, a range of digital signal processing (DSP) applications have been selected for experimentation. Among these applications, the wideband adaptive multi-rate (WB-AMR) speech decoder is the key application STM

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

is interested in. Program codes from this application is studied in more detail and are used for illustration of the techniques concerned. However, it should be emphasized that the techniques developed are applicable to a wide range of DSP applications. The results in applying the method to other signal processing benchmarks can be found in Chapter 4.

More specifically, the objectives of the project are:

(i) To analyze how high-level compiler optimization techniques can be applied for the purpose of reducing power consumption through optimizing data memory accesses for signal processing applications.

(ii) To develop a methodology for applying compiler techniques automatically for data-dominated programs, in which data accesses are represented by multi-dimensional arrays inside nested loops. This involves not only the evaluation of the advantages and cost of any chosen transformation, but also the automation of the transformation under data dependency constraints.

(iii) To determine the effectiveness of the method in reducing memory accesses for data-dominated programs.

1.2 Original Contributions

In the thesis, we studied source-level optimizations that can be used to minimize data memory access for data-dominated applications. This is because we believe that power efficiency should be explored fully in the early stage of the design process, i.e. at the source level during software compilation. This concept is originally proposed by researchers at the Interuniversity Microelelctronics Centre (IMEC) in Belgium [1]. They believe that there is a need for fast and early feedback at the algorithm level without going all the way to assembly code or hardware layout. Under this context, we further implement the concept and convert it into an automatic procedure which consists of three main steps:

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

profiling, inlining and global transformation. The procedure developed in this thesis is under the concept of the DTSE methodology [11, 1], and it is platform independent. It reduces array reference accesses at the source code level. Its main advantage is that it only needs to be executed once within the complete design cycle. Profiling results show that significant improvements can been achieved.

Another main contribution of the thesis is the new systematic method that has been developed to exploit multiple-index subscripted references. In the optimization of the WB-AMR speech decoder, multiple-index subscripted references are found in filtering operations, which account for a large number of data access. Reuse occurs when a datum is accessed more than once. To exploit these reuses, one possible way is to apply scalar replacement to store the reused reference into a scalar variable. However, in applying high-level transformations to improve register allocation, previous works [12, 13, 14, 15, 16] limit the references to be single-index references. These methods are not applicable to multiple-index subscripted references, producing unsatisfactory results. Hence, a new method that is able to exploit multiple-index subscripted references is developed in this thesis. This method performs a sequence of high-level transformations to minimize temporal reuses based on an estimation model. It broadens the optimization scope compared to previous methods based on single-index subscripted references resulting in further reduction in memory access. The effectiveness of the method is demonstrated by applying it to signal processing benchmarks.

The techniques and results reported in this thesis have been published in international conferences and submitted to refereed archival journals. A complete list of them can be found in Appendix B.

1.3 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 gives background knowledge in both memory management and the foundations of high-level transformation. Memory

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

management in embedded system design forms the context of this thesis, and high-level transform is the main technique used in the thesis. Chapter 3 presents our optimization procedure with its application to the WB-AMR speech decoder. The WB-AMR speech decoder, which is used as an example in the discussion of the optimization procedure, is also briefly described. The optimization results on the decoder confirm the effectiveness of the procedure. Chapter 4 proposes a new method that is able to exploit multiple-index subscripted references. The method is analyzed on signal processing benchmarks. Finally, Chapter 5 concludes the thesis and provides some recommendations for further research.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Chapter 2

Custom Memory Management Source-Level Transformations

The focus of the thesis is on software which involves a large amount of data memory accesses. In Section 2.1 we provide a brief review of memory management for design space exploration and optimization in memory system of embedded system which forms the context of this thesis. Our approach to memory access reduction is to use source-level transformations. Section 2.2 will introduce fundamental loop transformation techniques and related data dependency problems for the understanding of the method in the rest of this thesis.

2.1 Custom Memory Management

For data-dominated applications like multimedia applications, memory related power consumption has become an additional major design issue in embedded system design. This is because experiments show that power consumption of this kind of applications is largely contributed by data transfer and memory access operations [17, 1, 18, 9]. In contrast, data-path operations consume much less power. Consequently, design space

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

exploration and optimization for both custom architectures (ASIC) and programmable processor platforms with predefined memory organizations in embedded system has become an important issue. We shall refer to them as memory management issues.

Memory related design and optimization can be realized in both hardware [19, 20, 21] and software [22, 23, 24], or the combination of the two [25, 26]. However, memory system of embedded system design is a complex function of hardware architecture and software usage. A systematic method is needed to help designers with the data storage and transfer exploration problem. IMEC was the first research group to identify the importance of a formal approach to system-level memory management because of the large costs in terms of power, area, and performance associated with data transfer and storage [11]. The data transfer and storage exploration (DTSE) methodology being developed by IMEC helps the designer to determine an optimal execution order for data transfers together with an optimal memory architecture for storing the data of the given application [1].

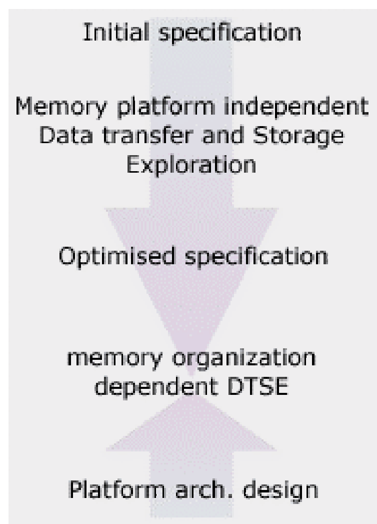


Figure 2.1: The DTSE Methodology [1]

The main idea of the DTSE methodology is shown in Figure 2.1. The methodology consists of a platform-independent stage followed by a platform-dependent one. Both stages include the systematic application of well decoupled optimization steps. The

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

first stages are almost independent of the target architecture. The advantage is that they only need to be executed once during the complete design trajectory irrespective of the number implementation platforms that is considered. The results obtained by the platform independent stage can be further improved. That is done by adapting the source code to the characteristics of the underlying (platform) memory organization. Its advantage is mainly the possibility of exploring platform specific trade-offs (e.g. speed versus power) that add crucial gains to the target cost function (e.g. power).

Our work is developed under the concept of the DTSE methodology. We aim to minimize data memory related power consumption by reducing data memory access. The most effective place to verify the data flow of an application and determine an optimal execution order for data access is at the system level specification where in practice it is usually the source code of an application. Thus we adopt the concept of platform independent stage of the DTSE methodology to optimize the initial specification (in the form of the original source code. Our specification optimization method to be presented in Chapter 3 consists of three major steps: profiling, inlining and global transformation. This procedure can be incorporated into any DTSE-like methodology for platform independent optimization. In the procedure, the main technique used is high-level compiler transformations, which are loop transforms and data transformations. Section 2.2 will introduce some fundamental transformations and the related data dependency problem.

Our approach to reduce memory access is by improving register allocation through high-level transformations. In the area of applying high-level transformations to improve register allocation, Callahan et al. [16] first introduced scalar replacement to identify potential reuse of subscripted variables. They also showed how to enhance scalar replacement by unroll-and-jam. Later on, Carr and Kennedy [12] applied scalar replacement and unroll-and-jam to improve the balance of loops. More recently, Carr and Guan [13] improved unroll-and-jam by using the reuse model proposed in [27] and its associated linear algebra framework. At the same time, they also limit their scope to uniformly

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

generated set [27], i.e. the subscript expressions of any two array references in the group are only different with a constant term, e.g. $x[i]$, $x[i+1]$ and $x[i+5]$. Both [12] and [13] left the task of selecting unrolling factors to the compiler rather than the programmer, meaning lack of automation. Sarkar [15] attempts to increase instruction level parallelism of loop bodies by unroll-and-jam. The problem of selecting unrolling factors for multiple perfectly nested loops is formulated as an optimization problem. In these previous works, array references are restricted to have single-index subscripts (each dimension of an array reference is a function of one loop index, e.g. $x[i][j]$ instead of $x[i+j][j]$). In this thesis, our work complements these previous works by considering the temporaries generated by multiple-index array variables. A new method of exploiting multiple-index subscripted variables is proposed in Chapter 4 and applied to signal processing benchmarks. The basic assumption we make is that the compiler's register allocator will place scalar variables into registers.

2.2 High-level Transformations

Most of the source-level optimizations are combinations of the transformations described in the following subsections. Knowledge in these transformations will facilitate the understanding of this thesis.

2.2.1 Loop Fusion (Merging)

Codes containing sequences of loops may be fused to increase data reuse. For the code in Figure 2.2, the *temp* array appears in both two loops. These loops can be fused as shown in Figure 2.3. This eliminates all references to *temp*. Loop fusion is especially important when the loop boundary value n is large and there are many cache misses. It is a difficult optimization for compilers to perform, because they must look across multiple loops and check that there is not too much register pressure before performing the fusion.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

For example, if fusing loops cause the compiler to spill and restore data from cache, the fusion may be detrimental to performance.

```
for (i = 0; i < n; i++)
   temp[i] = x[i] * y[i];
for (i = 0; i < n; i++)
   z[i] = w[i] + temp[i];
```

Figure 2.2: Before loop fusion

```
for(i = 0; i < n; i++)
   z[i] = w[i] + x[i] * y[i];
```

Figure 2.3: After loop fusion

2.2.2 Loop Fission (Splitting)

Loop Fission is opposite of loop fusion. There are times when loops need to be split or fissioned to help performance. Loop fission can be either platform independent or platform dependent. When loop fission is used to eliminate data dependency to facilitate other transformations, it is platform independent. Loop fission can be used to reduce cache delays resulting from cache line conflicts and to help optimizer perform more efficient memory blocking on resulting loops (platform dependent). Other reasons for performing loop fission are mainly to increase parallelism, or transform a loop into several perfect loop nests. Perfect loop nest makes other transformation tasks easier. An example of loop fission is shown in Figure 2.4 and Figure 2.5. The two array references $x[i]$ and $x[i+m]$ in Figure 2.4 are apart from each other and might cause a high cache miss ratio. To improve cache locality, the original loop is split into two loops as in Figure 2.5. In addition, loop fission has the disadvantages of increasing the number of loop nests and decreasing the temporal locality.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

```
for(i = 0; i < n; i++)
   y[i] = y[i] + x[i] + x[i+m];
```

Figure 2.4: Before loop fission

```
for(i = 0; i < n; i++)
   y[i] = y[i] + x[i];
for(i = 0; i < n; i++)
   y[i] = y[i] + x[i+m];
```

Figure 2.5: After loop fission

2.2.3 Loop Interchange

One of the easiest optimizations to perform for nested loops is to interchange the loops to improve data locality. Consider the code in Figure 2.6, the array y is accessed with loop i as the innermost loop, which is column-major. However, a row-major layout of memory space is allocated for the two-dimension array y, i.e. y is stored along the dimension indexed by loop j. This causes a high cache miss ratio. Interchanging the inner and outer loops should be applied here to improve the performance. The code after loop interchange is shown in Figure 2.7. Usually the loop having smaller dimension should be made the innermost loop.

```
for(j = 0; j < m; j++)
    for(i = 0; i < n; i++)
        y[i][j] = 0;
```

Figure 2.6: Before loop interchange

2.2.4 Unroll-And-Jam

Unroll-and-jam is a technique that refers to unrolling multiple loops and jamming them back together in ways that reduce the number of memory operations. Consider a perfect nest of two loops, i_1 and i_2, as shown in Figure 2.8(a), and we wish to unroll only the outer

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

2.2.5 Scalar Replacement

David Callahan, et al. [16] introduces a source-to-source transformation technique called scalar replacement to improve the likelihood of register allocations for subscripted variables. For the code in Figure 2.9, most compilers will not keep $A[i]$ in the inner loop in a register. This happens in spite of the fact that standard optimization techniques are able to determine that the address of the subscripted variable (e.g. $A[i]$) is invariant in the inner loop. On the other hand, if the loop is rewritten as shown in Figure 2.10 such that $A[i]$ is replaced by a scalar variable *tmp*, even the most naive compilers will allocate *tmp* to a register.

```
for (i = 1; i < N; i++)
    for (j = 1; j < M; j++)
        A[i] = A[i] + B[j];
```

Figure 2.9: Before scalar replacement

```
for (i = 1; i < N; i++)
{
    tmp = A[i];
    for (j = 1; j < M; j++)
        tmp = tmp + B[j];
    A[i] = tmp;
}
```

Figure 2.10: After scalar replacement

In this thesis, we assume that compilers with coloring-based register allocators are able to place a scalar variable into a register.

2.3 Data Dependency

When applying high-level transformations, we must take into account data dependency. Data dependency reflects the relationship of accesses to array data in terms of loop

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

iterations. Data dependency can be classified into four types. They are explained in Section 2.3.1. The choice of transformations is determined by the type of dependency present.

2.3.1 Four Types Of Dependency

A memory reference is either a read or a write. If A and B are two separate memory references, they can have the following four types of data dependency:

- B is flow dependent on A if the value written by A is read by B (read after write).
- A is anti-dependent on B if A must read the value before it is changed by B (write after read).
- A is output dependent on B if both A and B write to the same memory location (write after write).
- A is input dependent on B if both A and B read the same memory location (read after read).

For flow dependence (read after write) and input dependence (read after read), the subsequent access after the first access are reads. Hence, after the first access the datum can be placed into a register for the subsequent reads. This is done by scalar replacement.

2.3.2 Lexicographic Order And Transformation Legality

A validity problem is caused by data dependency that involves write access. No transformation is allowed to change the data dependency in the original code. The word *validity* is also referred to as *legality* in [28]. This thesis uses the word *legality* in the following discussions. To ensure legality, a concept of lexicographic order [28] is adopted in the project.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Consider an M-level loop nest, the Cartesian product of loop bounds forms an iteration space. In this M-dimensional space, the M-tuple of index vector $\vec{I} = [I_1 \ ... \ I_m \ ... \ I_M]$ can have different sets of values (e.g. Two different sets of values are denoted as $\vec{i}$ and $\vec{j}$). Hence a loop index subscripted array reference can generate different instances with different values of index vector $\vec{I}$. To determine the order of data access through the instances of a reference, the *distance vector* is computed as $\vec{d} = \vec{j} - \vec{i}$. A more formal definition of distance vector is defined in Definition 2.1. The distance vector measures the temporal distance between two data accesses in terms of the number of loop iterations. It is used to decide the legality of a transformation.

Definition 2.1. *Let M be the depth of a loop nest, and N be the dimensions of an array A. Two references $A[\vec{f}(\vec{I})]$ and $A[\vec{g}(\vec{I})]$ access data at the iterations when loop index vector $\vec{I}$ have the value of $\vec{i}$ and $\vec{j}$ respectively. The distance between the two accesses is $\vec{d} = \vec{j} - \vec{i}$. $\vec{d}$ is called distance vector.*

For a nonzero vector $\vec{v} = [v_1 \quad ... \quad v_m \quad ... \quad v_M]$, the *leading element* of the vector is its first nonzero element. If this leading element is v_l, then the positive integer l between 1 and M is called the *level* of $\vec{v}$. The level of a M-tuple zero vector is $M + 1$. A vector $\vec{v}$ is (*lexicographically*) *positive* or *negative* if its leading element is positive or negative, respectively. Thus the vectors $[2 \quad -3]$ and $[0 \quad 7 \quad -1]$ are positive, and the vectors $[-1 \quad 3]$ and $[0 \quad -8 \quad 4]$ are negative.

The sign of an integer v is denoted by $sig(v)$, where $sig(v) = 1$ if v is positive, $sig(v) = -1$ if v is negative, and $sig(0) = 0$. The sign of a vector $\vec{v} = [v_1 \quad ... \quad v_m \quad ... \quad v_M]$ is the vector of signs of its components, $sig(\vec{v}) = [sig(v_1) \quad ... \quad sig(v_m) \quad ... \quad sig(v_M)]$. A *direction vector* is a vector with each of its elements is one of the integers: 1, 0 or -1. Thus the sign of a vector is a direction vector.

With both distance vector and direction vector, the order of vectors in an iteration space can be defined.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

- $\vec{i} \prec_l \vec{j}$ iff $i_1 = j_1$, $i_2 = j_2$, ..., $i_{l-1} = j_{l-1}$ and $i_l < j_l$ for the integer l in $1 \leq l \leq M$. In other words, $\vec{i} \prec_l \vec{j}$ iff the direction vector of $\vec{d} = \vec{j} - \vec{i}$ is a positive vector of the form $[0 \quad 0 \quad ... \quad 0 \quad 1 \quad * \quad * \quad ... \quad *]$ with level l.

- $\vec{i} \preceq_l \vec{j}$ means either $\vec{i} \prec_l \vec{j}$ or $\vec{i} = \vec{j}$.

- $\vec{i} \prec \vec{j}$ iff $\vec{i} \prec_l \vec{j}$ for some l in $1 \leq l \leq M$. The lexicographic order $\prec$ is the union of all the relations $\prec_l$. It is similar for $\preceq$.

With lexicographic order defined, the data dependence between different references can be computed. The data dependence should not be broken for a legal transformation. A thorough discussion of data dependency in loop transformations can be found in [28]. The legality problem appearing in our method will be discussed in Section 4.3.

2.3.3 Loop-Carried, Loop-Independent And Distance Vector

Section 2.3.1 classifies data dependency into four types according to the order and types of access. It helps to identify temporal reuses. Another way to classify data dependency is according to the distance between multiple accesses. They are called *loop-carried dependence* and *loop-independent dependence*. A dependence is loop-carried if $\vec{d} \neq \vec{0}$, where $\vec{d}$ is the distance vector between two iterations. Otherwise, it is a loop-independent dependence.

Loop-carried dependence can be illustrated using the example in Figure 2.11. The distance between $X[i][j]$ and $X[i][j+1]$ is $[0 \quad 1]$. We say that the dependence is carried by the inner loop j. With the increment of the loop index j, the datum read by $X[i][j]$ in the current iteration is written by $X[i][j+1]$ in the previous iteration. This is a flow dependence.

In contrast to loop-carried dependence, an example of loop-independent dependence is shown in Figure 2.12. In this case, two references have same subscriptions (both are

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

```
for i=1, 10
  for j=1, 12
  {
    X[i][j+1]=...
    ...=X[i][j]
  }
```

Figure 2.11: Loop-carried dependence

$X[i][j]$). The distance between the two references $X[i][j]$ in the two statements is $[0 \quad 0]$. That means the dependence is not carried by any of the two loops. The order of the two references accessing a datum is determined by the order of the statements.

```
for i=1, 10
  for j=1, 12
  {
    X[i][j]=...
    ...=X[i][j]
  }
```

Figure 2.12: Loop-independent dependence

2.3.4 Loop-Invariant Reference And Reference-Invariant Loop

There are two important concepts used in this thesis: loop-invariant reference and reference-invariant loop. Their definitions are defined as follows.

Definition 2.2. *An array reference $A[\vec{f}(\vec{I})]$ ($A[I_1h_1+..+I_mh_m+...+I_Mh_M]$) is invariant to a loop index I_m if h_m (the corresponding coefficient of I_m) is equal to zero. That is to say, reference $A[\vec{f}(\vec{I})]$ is an I_m-invariant reference. More generally, reference $A[\vec{f}(\vec{I})]$ is a loop-invariant reference.*

On the other hand, the loop with the index I_m is invariant to the reference $A[\vec{f}(\vec{I})]$. That is to say, loop I_m is an $A[\vec{f}(\vec{I})]$-invariant loop. More generally, loop I_m is a reference-invariant loop.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

```
for i=1, 10
  for j=1, 12
    X[i]=X[i]+Y[j]
```

Figure 2.13: Loop-invariant references

Consider the example in Figure 2.13, the subscript of $X[i]$ is a function of only one loop index variable i. Thus $X[i]$ is invariant to loop j and loop j is $X[i]$-invariant. Similarly, $Y[j]$ is invariant to loop i and loop i is $Y[j]$-invariant; and loop j is $X[i]$-invariant.

2.3.5 Self-Temporal Reuse And Group-Temporal Reuse

Temporal reuse may result from *self-reuse* from a single array reference or *group-reuse* from multiple references [27]. The definitions of *self-temporal reuse* and *group-temporal reuse* are defined as follows.

Definition 2.3. *An array reference $A[\vec{f}(\vec{I})]$ generates self-temporal reuse if it accesses the same datum at different iterations $\vec{i}$ and $\vec{j}$ whenever $\vec{f}(\vec{i}) = \vec{f}(\vec{i})$.*

Definition 2.4. *Two array references $A[\vec{f}(\vec{I})]$ and $A[\vec{g}(\vec{I})]$ have group-temporal reuse if there are iterations $\vec{i}$ and $\vec{j}$ such that $\vec{f}(\vec{i}) = \vec{g}(\vec{j})$.*

```
for (i = 1; i < 10; i++)
    for (j = 1; j < 10; j++)
    ... = X[i] ...
```

Figure 2.14: Self-temporal reuse

The original definition of self-temporal reuse is proposed by M. E. Wolf and Monica S. Lam [27]. An example of self-temporal reuse from loop-invariant reference is shown in Figure 2.14. The reference $X[i]$ is invariant to the inner loop j. $X[i]$ accesses the same datum 10 times throughout the entire inner loop j. Previous works [12, 13, 14, 15, 16] on improving register allocation concentrated on loop-invariant references. This thesis

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

would consider one more situation for the self-temporal reuse. This kind of self-temporal reuse is from *multiple-index subscripted reference*. Multiple-index subscripted reference means that each dimension of an array reference can have a subscript which has more than one loop index variables. The concept of multiple-index subscripted reference will be discussed more in Section 4.1.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Chapter 3

Memory Access Reduction

This chapter presents a method to optimize the program source of an embedded software. The goal of the program source code optimization is to reduce memory access at a high level. Program source is the place where data flow is revealed clearly. Data access in a data-dominated program is dominated by access to array references inside nested loops. After a program is restructured, unnecessary multiple access to the same piece of data is eliminated while the execution order of the original computation for is preserved. Optimizations typically involve both data transformations and loop transformations.

The procedure proposed in this thesis involves three major sequential steps as shown in Figure 3.1. Profiling identifies the parts of the program where heavy data access occurs. Inlining is a technique to restructure the program such that function invocations are replaced with the body of the called function. This enables global transformation to be applied. Global transformations such as loop merging and unrolling serves to prune the data flow and minimize data access. This optimization process is almost independent of the target processor architecture. In this project, it is assumed that the target machine is a RISC processor, since RISC processors remain a popular choice in embedded and low-end market. RISC processors are load/store architectures in the sense that their instructions can process only operands present in CPU registers. Hence, minimizing the

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

number of load/store instructions (data access) by improving register allocation is the main concern in the global transformation step of the procedure, and cache is not consider in the current stage. The results obtained by this platform independent optimization procedure can be further improved by applying platform dependent optimizations. That includes adapting the source code to match the characteristics of the underlying memory organization.

Figure 3.1: The optimization procedure

In this chapter, our optimization method is described in detail. The data-intensive application chosen to study the effectiveness of our method is the WB-AMR speech decoder. Currently, the latter two steps, inlining and global transformation, in the procedure are applied to the WB-AMR speech decoder by manually tuning the program. The objective of this chapter is to investigate the effectiveness and feasibility of the procedure instead of implementing it. A formal algorithm to realize the procedure will be considered as the future work of this project. A brief overview of the WB-AMR speech decoder is given in Section 3.1. This is followed by the results obtained from applying our method to this decoder in the rest of the chapter.

3.1 WB-AMR Speech Decoder

The application that the sponsor of this project is interested in is the WB-AMR speech decoder. The source and related documents are downloaded from 3GPP's official website [29, 30]. The WB-AMR speech is developed to meet the demand for high quality speech communication with bandwidths higher than the conventional 3.4 kHz. It is based on the code-excited linear predictive (CELP) coding model with nine source rates from

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

6.60 kbit/s to 23.85 kbit/s. Speech quality exceeding G.711 PCM wireline quality can be achieved through the introduction of wider audio bandwidth (7 kHz) under medium and low error conditions. It will be applicable to existing and evolving GSM systems as well as other related systems such as those in the 3rd Generation Partnership Project (3GPP) [31]. Its channel robustness is similar to that of the existing narrowband GSM full rate (FR) and GSM enhanced full rate (EFR) speech coders when used in GSM.

3.1.1 Principles Of WB-AMR Speech Decoder

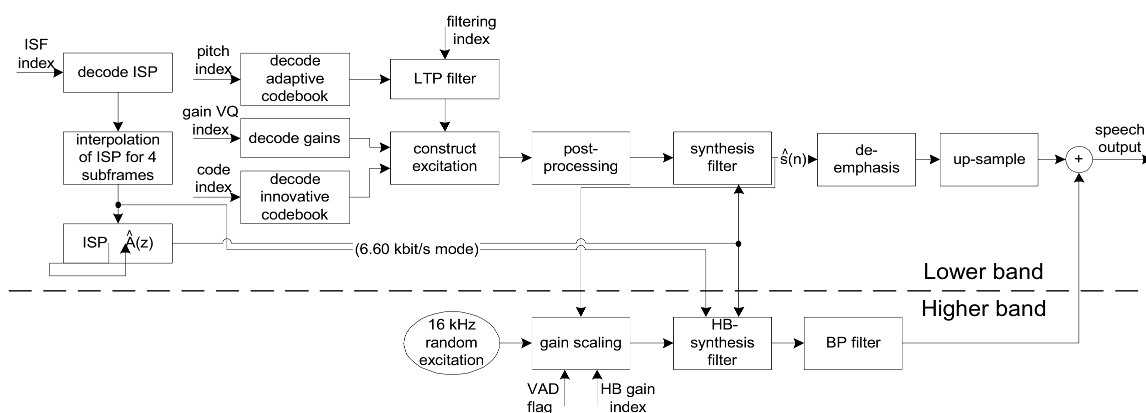


Figure 3.2: Detailed block diagram of the ACELP decoder

The signal flow of the decoder is shown in Figure 3.2. The decoder extracts the indices from the received bit stream sent by the encoder. The indices are decoded to obtain the coder parameters at each transmission frame. These parameters are the Immittance Spectral Pair (ISP) vector, four fractional pitch lags, four Long-Term Prediction (LTP) filtering parameters, four innovative code-vectors, and four sets of vector quantized pitch and innovative gains. In the 23.85 kbit/s mode, the high-band gain index is also decoded. The ISP vector is converted to LP filter coefficients and interpolated to obtain LP filters for each subframe. For each 64-sample subframe:

- The excitation is constructed by adding the adaptive and innovative code-vectors scaled by their respective gains.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

- The 12.8 kHz speech is reconstructed by filtering the excitation through the LP synthesis filter.
- The reconstructed speech is de-emphasized.

Finally, the reconstructed speech is up-sampled to 16 kHz and the high-band speech signal is added to the frequency band between 6 and 7 kHz.

3.1.2 Implementation of WB-AMR Speech Decoder

The simplified program of the WB-AMR speech decoder is shown in pseudo code in Figure 3.3. The program reads in data from the encoder frame by frame. For each frame, the Linear Prediction Coding (LPC) coefficients are computed from ISF indices. These coefficients are used in each of the 4 subframes to synthesize speech. In every subframe, the pitch index and code index are used to generate the adaptive codebook and the innovative codebook. The program then goes through a series of processes (phase dispersion, noise enhancement, pitch enhancement and post-processing) to enhance the two codebooks and gains to generate excitation signal which is passed to the *synthesis* function. The *synthesis* function generates synthesis speech and high frequency noise through LPC synthesis followed by filtering operations like de-emphasis and band selections.

From the implementation of this decoder, it can be seen that decoding is performed on a frame-by-frame basis. Within each frame, the parameters and filter memories are updated on a subframe basis. This implies that there are strong flow dependence between consecutive subframes, making any parallel processing based on frame or subframe to minimize common array accesses impossible.

However, other characteristics of the program still offer space for optimization. A series of processes are performed on a common data array in the *synthesis* function. This provides the opportunity to do loop merging among these processes. If any process is

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

```
main()
{
    //Read in data from encoder output frame by frame into 1-D array,
    //frame size depends on transmission mode
    while(array != EMPTY)
    {
         decoder(array);
    }
}

//decoder data in each frame
//and invoke synthesis function to synthesize speech
decoder(word16 * array)
{
     decode ISF index array[];
     compute ISP from ISF;
     compute LPC coefficient a[] from ISP;
     for (i_subframe = 0; i_subframe < frame_length; i_subframe)
     {
          decode pitch index from array[];
          obtain adaptive codebook by LTP filtering;
          decode code index from array[];
          generate innovative (fixed) codebook from code index;
          decode codebook gains (adaptive & fixed) from array[];
          phase dispersion (anti-sparseness) applied to fixed codebook;
          noise enhancer applied to fixed codebook gain;
          pitch enhancer applied to fixed codebook;
          construct total excitation exc[] from adaptive codebook and
             innovative codebook;
          post-processing applied to the total excitation exc[];
          synthesis(exc, a);
     }
}

synthesis(word16* exc, word16* a)
{
     //synthesis speech
     generate synthesis speech syn_speech[]
        by synthesis filtering (exc[], a[]);
     deemphasis by passing syn_speech[] through
        perceptual weighting filter;
     precaution HP filter with fc = 50Hz on syn_speech[];
     over sampling syn_speech[] from 12.8kHz to 16kHz;

     //generate high frequency noise
     generate white noise vector hf[];
     compute gain for noise;
     HP filter with fc = 400Hz on hf[];
     generate synthesis noise at 6kHz - 7kHz;
     add filtered hf[] noise to syn_speech[];
}
```

Figure 3.3: Simplified program of WB-AMR speech decoder

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

inside a function, this function can be inlined into the caller function. Another characteristic is that this program performs a lot of digital filtering which account for a substantial amount of data memory accesses. Thus optimizations can be expected. In the following subsections, we shall optimize this WB-AMR speech decoder using our optimization procedure.

3.2 Profiling

The optimization procedure is a source-to-source transformation process. In this thesis, we restrict the source to be C programs. Starting from the source code, a profile of memory access during the execution of the program is required. This profile shows the number of data array reads and writes for the whole program. This counting process is done by means of a software profiling tool such as the one in the Atomium tool suite, which is developed by IMEC to support the DTSE methodology [32]. In this tool suite, Atomium/Analysis [33] performs C source level profiling. It is used in our project. In the rest of the thesis, Atomium/Analysis will be referred to simply as Atomium.

The profiling analysis is instrumentation-based as shown in Figure 3.4. Atomium reads in a set of C program files, inserting some C preprocessor macros into the code and writing them out again as a set of C++ files. These files must then be compiled with a C++ compiler and linked with the Atomium run time library. When executed, the resulting program will generate access statistics for the arrays present in the code performing its normal functions. From the profiling results, areas of heavy data array access (referred to as bottlenecks later) and the call-path (functions that access the arrays) of these arrays can be identified. These bottlenecks are candidates for transformations in later steps.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

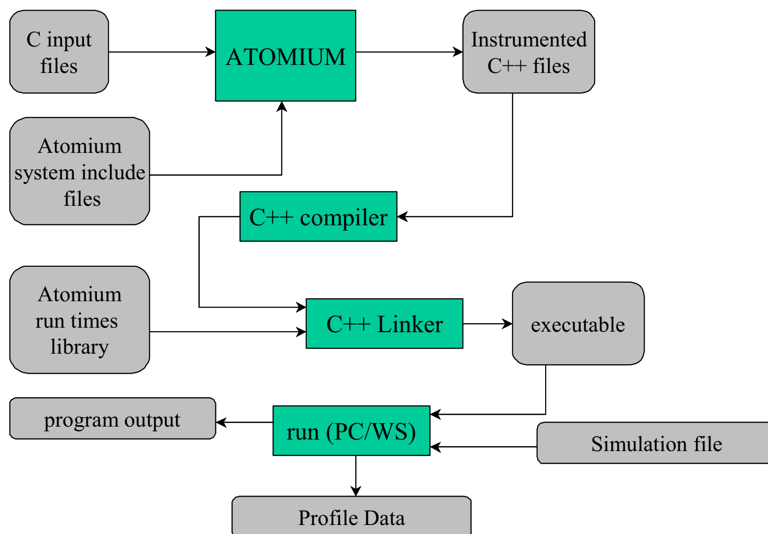


Figure 3.4: Memory-access profiling process using Atomium

3.2.1 Profile Of WB-AMR Decoder

As mentioned in Section 3.1, the WB-AMR codec consists of nine coding modes with bit-rates ranging from 6.60 kbit/s to 23.85 kbit/s. In each 20 ms speech frame, the number of bits generated varies from mode to mode. The differences in bit allocations make the encoding and decoding processes different in terms of complexity. However, the underlining algorithm remains same. Also the bottlenecks of different codec modes are essentially associated with same arrays. This can be observed by examining the Atomium profile results. Without loss of generality, only the mode 0 profile report is shown. The output of Atomium is in HTML format as shown in Table 3.1.

3.2.2 Analysis

For mode 0, the total number of access is approximately 25 million. This number is contributed by a total of 85 arrays. The first nine arrays (*old_exc*, *x*, *fir_6k_7k*, *inter4_2*, *Aq*, *signal*, *fir_up*, *y_buf*, *Ap*) involve about 16 million, or 64%, of total accesses. These nine arrays together with another nineteen arrays (*synth_hi*, *synth_lo*, *code*, *exc2*, *HF*, *st*, *synth*, *f1*, *f2*, *synth*, *excp*, *code2*, *HfIsf*, *a*, *a*, *b*, *b*, *code2*, *IsfDiff*) comprise 24.5 million,

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 3.1: List of arrays sorted by number of accesses

Total accesses:

Accesses	Writes	Reads
25021926	3140739	21881187

Accesses per array:

Accesses	Writes	Reads	Elements	Lives	Name	Location	Type
2935300	497228	2438072	505	197	old exc	[dec main.c:126]	Word16
2064560	86680	1977880	110	788	x	[hp6k.c:44]	Word16
1954271	31	1954240	31	1	fir 6k 7k	[hp6k.c:23]	Word16
1639168	128	1639040	128	1	inter4 2	[pred lt4.c:24]	Word16
1628008	13396	1614612	68	197	Aq	[dec main.c:129]	Word16
1601216	69344	1531872	88	788	signal	[decim54.c:51]	Word16
1513080	120	1512960	120	1	fir up	[decim54.c:29]	Word16
1355360	78800	1276560	100	788	y buf	[syn filt.c:31]	Word16
1278136	16548	1261588	21	788	Ap	[dec main.c:751]	Word16
983424	63040	920384	80	788	synth hi	[dec main.c:747]	Word16
932992	63040	869952	80	788	synth lo	[dec main.c:748]	Word16
843331	177001	666330	64	197	code	[dec main.c:132]	Word16
832768	296448	536320	256	197	exc2	[dec main.c:134]	Word16
756480	315200	441280	80	788	HF	[dec main.c:750]	Word16
602984	263168	339816	1	1	st	[dec main.c:43]	Decoder State
553176	151296	401880	64	788	synth	[dec main.c:749]	Word16
486984	166268	320716	11	1576	f1	[isp az.c:34]	Word32
433400	148144	285256	10	1576	f2	[isp az.c:35]	Word32
318080	191040	127040	320	1	synth	[decoder.c:29]	Word16
310016	132864	177152	64	197	excp	[dec main.c:144]	Word16
302592	183744	118848	128	788	code2	[ph disp.c:49]	Word16
271072	66192	204880	20	197	HfIsf	[dec main.c:146]	Word16
201731	3	201728	3	1	a	[hp50.c:25]	Word16
201731	3	201728	3	1	a	[hp400.c:25]	Word16
151299	3	151296	3	1	b	[hp50.c:22]	Word16
151299	3	151296	3	1	b	[hp400.c:22]	Word16
100864	50432	50432	64	197	code2	[dec main.c:133]	Word16
100578	25222	75356	14	788	IsfDiff	[isfextrp.c:29]	Word16
...	...	...	...	...	...	...	...

Sort by [accesses] [writes] [reads] [elements] [name]
Top-level report

Data collected at Tue Apr 22 14:08:31 2003. Report generated at Tue Apr 22 14:09:23 2003.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

or 98%, of total access. These arrays are declared exclusively in eleven C program files (*dec_main.c*, *hp6k.c*, *pred_lt4.c*, *decim54.c*, *syn_filt.c*, *isp_az.c*, *decoder.c*, *ph_disp.c*, *hp50.c*, *hp400.c* and *isfextrp.c*). It can be seen that memory accesses are spread over many parts of the program. The most frequently accessed array (*old_exc*) accounts for only 11.6% of total access. This implies that transformations will have to be performed in many parts of the C code in order to make the optimization effective. To locate the memory related bottlenecks of the application, every array listed in Table 3.1 has to be traced through the HTML profile.

Table 3.2: Array *old_exc* [*dec_main.c*:126] Global call

Local			Activ.	Description
Accesses	Writes	Reads		
0	0	0	1	main [decoder.c:25]
302592	50432	252160	197	+-decoder [dec_main.c:123]
1690260	51220	1639040	788	+-+-Pred_lt4 [pred_lt4.c:31]
592576	296288	296288	985	+-+-Scale_sig [scale.c:25]
249008	99288	149720	2758	+-+-Copy [util.c:42]
100864	0	100864	197	+-+-dtx_dec_activity_update [dtx.c:445]

Data collected at Tue Apr 22 14:08:31 2003. Report generated at Tue Apr 22 14:09:58 2003.

The global trace of the most frequently access array *old_exc* is listed in Table 3.2. The accesses to the array *old_exc* are distributed among the function *decoder* and its descendants. These functions are the candidates for inlining and loop transformations. Whether these functions are eventually inlined or not depends on data dependency, gain and cost. The situation is different for array *Aq*, with its trace shown in Table 3.3. It can be observed that memory access to array *Aq* happen entirely inside the functions *Syn_filt_32* and *Isp_Az*.

Other arrays show similar patterns of access. An analysis of the profile reveals the memory access hot spots in the code. These functions listed in Table 3.4 are respon-

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 3.3: Array Aq [$dec_main.c$:129] Global call trace

Local			Activ.	Description
Accesses	Writes	Reads		
0	0	0	1	main [decoder.c:25]
0	0	0	197	+--decoder [dec_main.c:123]
0	0	0	788	+--+--synthesis [dec_main.c:742]
1614612	0	1614612	788	+--+--+--Syn_filt_32 [syn_filt.c:71]
0	0	0	197	+--+--Int_isp [int_lpc.c:30]
13396	13396	0	788	+--+--+--Isp_Az [isp_az.c:31]

Data collected at Tue Apr 22 14:08:31 2003. Report generated at Tue Apr 22 14:09:25 2003.

Table 3.4: Major function for memory accesses

Function	**Calling Function**	**Local Access**	**Total Access (%)**
decoder	Main	1076490	4.3
Filt_6k_7k	synthesis	4097600	16.4
Syn_filt_32	synthesis	3430164	13.7
Pred_lt4	Decoder	3329300	13.3
Interpol	Up_samp	3025920	12.1
Syn_filt	synthesis	2774548	11.1
Copy	Many	987040	3.9
Scale_sig	decoder | synthesis	794304	3.2
Dot_product12	D_gain2 | synthesis |voice_factor	529536	2.1
Get_isp_pol_16kHz	Isp_Az	518504	2.1
HP50_12k8	synthesis	463344	1.9
HP400_12k8	Synthesis	463344	1.9

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

sible for most of the memory accesses in the code. They account for 86% of the total memory access. Most of these functions have only one direct caller, the function *synthesis*. Furthermore, both *synthesis* and the *Pred_lt4* are directly called by *decoder*. The optimization of the application is done within this scope.

3.3 Inlining

When writing programs, the concept of functions provides clarity of programming. The mapping from the system to the program through functions is easier than that of a one-function program. However, from the viewpoint of optimization, the drawback of functions is that they may act as brick walls between sections of code. For example, redundant access to a common array in both the caller (a function that initiate a function call) and the callee (a function invoked by another function) can be eliminated when the code of the callee is in the caller. Inlining is the technique for removing these brick walls.

The main purpose of inlining a function is to enlarge the exploration space for optimization. Inlining makes it possible to reduce access to a common array in different functions. It provides the opportunity to merge related operations. Inlining also provides scheduling freedom so that parallel processing can be synchronized to produce a lower cycle budge. This is especially useful for real-time signal processing applications. Apart from the advantage of facilitating the optimization of the called function in the context of call-site, inlining also reduce function calls. The main drawback of inlining is that it increases the code size. It has been shown that maximizing function calls under code size constraint is NP-hard [34]. In addition, inlining may increase register pressure too.

With our method, only the bottleneck functions are evaluated for inlining. A call-based inlining scheme is adopted. The decision to inline is made independently at each call-site (where a function call is raised). With the bottlenecks identified from the profile, the call-sites at these bottlenecks are candidates for inlining. The reason for inlining is

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

that it facilitates global transformation. For example, inlining allows us to merge loops that exist in two different functions.

For the WB-AMR decoder, the selection of functions to inline is done by inspecting the source code and the Atomium profile. Not all of the functions in Table 3.4 are suitable for inlining. Most of the inlinings happen inside the *synthesis* function and the *decoder* function. The *decoder* function decodes the input stream, followed by the *synthesis* function that synthesizes the output speech. These two main tasks are partitioned into many smaller tasks in different functions. These functions are invoked by *decoder* and *synthesis*. Inlining the related functions into *decoder* and *synthesis* helps to merge the related operations in the decoder algorithm.

```
static void synthesis(…)
{
    …
    Syn_filt_32(…);
    …
    Deemph_32(…);
    …
    HP50_12k8(…);
    …
    Dot_product12(…);
    …
    HP400_12k8(…);
    L_tmp = 1L;
      for (i = 0; i < L_SUBFR; i++)
            L_tmp = L_mac(L_tmp, synth[i], synth[i]);
    …
}
```

Figure 3.5: The simplified synthesis function

The selected functions for inlining in the *synthesis* function are shown in Figure 3.5. The *synthesis* function takes the excitation signal from the decoder functions and computes the synthesis speech. The first four functions (*Syn_filt_32, Deemph_32, HP50_12k8, Dot_product12*) in the *synthesis* are inlined, since they have same dimensions of *for* loop, and access the same arrays. After inlining, the *for* loops in these functions can be merged together and redundant memory access can be reduced. The function *HP400_12k8* is inlined so that it can be merged with the *for* loop following its function call in synthesis.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

```
void decoder(…)
{
    for (i_subfr = 0; i_subfr < L_FRAME; i_subfr += L_SUBFR)
    {
        …
        Copy(&exc[i_subfr], exc2, L_SUBFR);
        Scale_sig(exc2, L_SUBFR, -3);
        …
        Copy(&exc[i_subfr], exc2, L_SUBFR);
                    for (i = 0; i < L_SUBFR; i++)
        {
                                                …
            exc[i + i_subfr] = …;
        }
        …
        for (i = 0; i < L_SUBFR; i++)
        {
            tmp = abs_s(exc[i + i_subfr]);
            …
        }
        …
        synthesis(…)
        …
    }
}
```

Figure 3.6: The simplified decoder function

The *decoder* function in Figure 3.6 decodes the parameters from the input data stream and computes the excitation signal which is then passed to the *synthesis* function. The *decoder* function operates on subframes. Although the *synthesis* function is in the *i_subfr* loop, it is not suitable for inlining. This is because the synthesis is only performed after all the decoding related processes are done. That is, the synthesis can not be merged with one or two decoding related processes. In the *decoder* function, the functions inlined are *Copy* and *Scale*. They are inlined for loop merging since they access the same array *exc*. The second *Copy* function is inlined to merge with the two *for* loop immediately following it.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

3.4 Global Transformation

Actual memory access reduction is effected through global transformation. Memory access is reduced if the number of the load and the store instructions is reduced. This can be achieved by increasing the exploitation of array data reuse in space (spatial reuse) and data reuse in time (temporal reuse). Using multiple data points of a cache line before the line is replaced with some other line is an example of spatial reuse. Temporal reuse refers to multiple usage of the same data that are very close in time.

Reuse is something inherent in the computation and does not depend on the particular way the loops are written [27]. Loop transformation aims to obtain a better data locality and increase the exploitation of reuse. For example, an array data item is first written to and then followed by a read some time later after a few iterations. There is reuse of this array data. If the write and the read are too far apart in time then there is poor locality. However, if a loop transformation can be performed to bring the read and the write closer (better locality), then the data can be kept in a processor register. In this thesis, the optimization procedure is a platform independent process. Hardware supports such as cache are not considered. Thus, space locality and cache policy are left for platform dependence optimizations in further research. Thus we only consider exploitation of temporal reuse.

For the WB-AMR speech decoder, different types of loop transformations are applied to exploit reuse and reduce data access in the program. In the following subsections, the transformations and how they can be combined in different ways to reduce memory accesses are explained.

3.4.1 Inlining And Loop Merging

In the *synthesis* function, the first four functions (*Syn_filt_32, Deemph_32, HP50_12k8, Dot_product12*) are inlined. The original code of these functions are shown in Figure 3.7, Figure 3.8 and Figure 3.9 respectively.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

```
void Syn_filt_32(…)
{
   for (i = 0; i < L_SUBFR; i++)
   {
     L_tmp = 0;
     for (j = 1; j <= m; j++)
     L_tmp = L_msu(L_tmp, sig_lo[i - j], a[j]);
     L_tmp = L_shr(L_tmp, 16 - 4);
     L_tmp = L_mac(L_tmp, exc[i], a0);
     for (j = 1; j <= m; j++)
     L_tmp = L_msu(L_tmp, sig_hi[i - j], a[j]);
     L_tmp = L_shl(L_tmp, 3);
     sig_hi[i] = extract_h(L_tmp);
     L_tmp = L_shr(L_tmp, 4);
     sig_lo[i] = extract_l(L_msu(L_tmp, sig_hi[i], 2048));
   }
}
```

Figure 3.7: The *Syn_filt_32* function

```
void Deemph_32(…)
{
  L_tmp = L_deposit_h(sig_hi[0]);
  L_tmp = L_mac(L_tmp, sig_lo[0], 8);
  L_tmp = L_shl(L_tmp, 3);
  L_tmp = L_mac(L_tmp, *mem, fac);
  L_tmp = L_shl(L_tmp, 1);
  synth[0] = round(L_tmp);

  for (i = 1; i < L_SUBFR; i++)
  {
     L_tmp = L_deposit_h(sig_hi[i]);
     L_tmp = L_mac(L_tmp, sig_lo[i], 8);
     L_tmp = L_shl(L_tmp, 3);

     L_tmp = L_mac(L_tmp, synth[i - 1], fac);
     L_tmp = L_shl(L_tmp, 1);
     synth[i] = round(L_tmp);
  }
}
```

Figure 3.8: The *Deemph_32* function

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

```
void HP50_12k8(…)
{
    for (i = 0; i < lg; i++)
    {
        x2 = x1;
        x1 = x0;
        x0 = synth[i];
        L_tmp = 16384L;
        L_tmp = L_mac(L_tmp, y1_lo, a50[1]);
        L_tmp = L_mac(L_tmp, y2_lo, a50[2]);
        L_tmp = L_shr(L_tmp, 15);
        L_tmp = L_mac(L_tmp, y1_hi, a50[1]);
        L_tmp = L_mac(L_tmp, y2_hi, a50[2]);
        L_tmp = L_mac(L_tmp, x0, b50[0]);
        L_tmp = L_mac(L_tmp, x1, b50[1]);
        L_tmp = L_mac(L_tmp, x2, b50[2]);

        L_tmp = L_shl(L_tmp, 2);

        y2_hi = y1_hi;
        y2_lo = y1_lo;
        L_Extract(L_tmp, &y1_hi, &y1_lo);

        L_tmp = L_shl(L_tmp, 1);
        synth[i] = round(L_tmp);
    }
}
```

Figure 3.9: The *HP50_12k8* function

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

In Figure 3.7, the function *Syn_filt_32* modifies both arrays *sig_hi* and *sig_lo*. These two arrays are then passed to the function *Deemp_32* in Figure 3.8. Similarly, the functions *Deemp_32* and *HP50_12k8* access the same array *synth*. This indicates that the multiple accesses to these arrays can be eliminated by storing the array data into temporary variables by applying inlining and loop merging.

The dimensions of the loops that can be merged should be the same and aligned. The loop in *Deemp_32* starts from 1 instead of 0, while the one in *Syn_filt_32* starts from 0. This problem can easily be solved by introducing a temporary variable to eliminate the difference in computing $synth[0]$ and $synth[i]$ $(i >= 0)$. The new code after inlining and loop merging is shown in Figure 3.10. Comparing the merged loop with the original code, the number of array references drops from sixteen (only the arrays in the loop counted) to eight. Apart from this gain, the computation is also reduced after transformation. In function *Deemph32*, *L_tmp* is recomputed from $sig_hi[i]$ and $sig_lo[i]$, indicated by the shaded lines in *Deemph32*. This re-computation is replaced by one line (shaded in Figure 3.10) after loop merging. Note that the coefficient arrays $a50[0..2]$, $b50[0..2]$ in Figure 3.10 can be replaced by 6 constants to achieve another improvement. One drawback of this inlining and loop merging process is the introduction of additional variables. There are 24 new variables in *synthesis*.

This process is also applied to the rest of the inlined code such as the *HP400_12k8* function called by *synthesis* and *Copy* and *Scale_sig* called by *decoder*. The transformed code and the original code are listed in the appendix A.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

```
for (i = 0; i < L_SUBFR; i++)
{
  L_tmp = 0;
  L_tmp_hi = 0;
  for (j = 1; j <= M; j++)
  {
      Aqtemp = Aq[j];
      L_tmp = L_msu(L_tmp, sig_lo[i - j], Aqtemp);
      L_tmp_hi = L_msu(L_tmp_hi, sig_hi[i - j], Aqtemp);
  }
  L_tmp = L_shr(L_tmp, 16 - 4);
  L_tmp=L_add(L_tmp, L_tmp_hi);
  exc_temp = exc[i];
  L_tmp = L_mac(L_tmp, exc_temp, a0);
  exc_L_tmp = L_deposit_h(exc_temp);
  exc_L_tmp = L_shl(exc_L_tmp, -3);
  exc_temp = round(exc_L_tmp);
  exc[i] = exc_temp;
  L_sum = L_mac(L_sum, exc_temp, exc_temp);
  L_tmp = L_shl(L_tmp, 3);
  sig_hi_temp = extract_h(L_tmp);
  sig_hi[i] = sig_hi_temp;
  L_tmp = L_shr(L_tmp, 4);
  sig_lo[i] = extract_l(L_msu(L_tmp, sig_hi_temp, 2048));
  L_tmp = L_shl(L_tmp, 7);
  L_tmp = L_mac(L_tmp, synth_temp, fac);
  L_tmp = L_shl(L_tmp, 1);
  synth_temp = round(L_tmp);

  //inlined: HP50_12k8(synth, L_SUBFR, st->mem_sig_out);
  x2 = x1;
  x1 = x0;
  x0 = synth_temp;
  L_tmp = 16384L;
  L_tmp = L_mac(L_tmp, y1_lo, a50[1]);
  L_tmp = L_mac(L_tmp, y2_lo, a50[2]);
  L_tmp = L_shr(L_tmp, 15);
  L_tmp = L_mac(L_tmp, y1_hi, a50[1]);
  L_tmp = L_mac(L_tmp, y2_hi, a50[2]);
  L_tmp = L_mac(L_tmp, x0, b50[0]);
  L_tmp = L_mac(L_tmp, x1, b50[1]);
  L_tmp = L_mac(L_tmp, x2, b50[2]);
  L_tmp = L_shl(L_tmp, 2);
  y2_hi = y1_hi;
  y2_lo = y1_lo;
  L_Extract(L_tmp, &y1_hi, &y1_lo);
  L_tmp = L_shl(L_tmp, 1);
  synth[i] = round(L_tmp)
}
```

Figure 3.10: After inlining and loop merging

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

3.4.2 Loop Merging And Scalar Replacement

In order to maximally exploit the optimization opportunities, transformations are also applied in different combinations to other parts of the code besides the inlined functions. In the *decoder* function, there is a segment of code shown in Figure 3.11. The first loop have redundant accesses to the array *code*. Each of the array references from *code*[2] to *code*[$L_SUBFR - 2$] is accessed three times. In the second loop, redundant access happen to array *exc2*. Furthermore, array reference *code*2[i] is accessed in both loops. In total, there are nine references in these two loops.

```
/* code2[i] = code[i] - code[i+1]*tmp - code[i-1]*tmp */
     L_tmp = L_deposit_h(code[0]);
     L_tmp = L_msu(L_tmp, code[1], tmp);
     code2[0] = round(L_tmp);
     for (i = 1; i < L_SUBFR - 1; i++)
     {
         L_tmp = L_deposit_h(code[i]);
         L_tmp = L_msu(L_tmp, code[i + 1], tmp);
         L_tmp = L_msu(L_tmp, code[i - 1], tmp);
         code2[i] = round(L_tmp);
     }
     L_tmp = L_deposit_h(code[L_SUBFR - 1]);
     L_tmp = L_msu(L_tmp, code[L_SUBFR - 2], tmp);
     code2[L_SUBFR - 1] = round(L_tmp);

     /* build excitation */
     gain_code = round(L_shl(L_gain_code, Q_new));
     for (i = 0; i < L_SUBFR; i++)
     {
         exc2 [i] = code2[i]*gain_code + exc2[i]*gain_pit;
         L_tmp = L_mult(code2[i], gain_code);
         L_tmp = L_shl(L_tmp, 5);
         L_tmp = L_mac(L_tmp, exc2[i], gain_pit);
         L_tmp = L_shl(L_tmp, 1);
         exc2[i] = round(L_tmp);
     }
```

Figure 3.11: Code segment for loop merging and scalar replacement

The two loops in Figure 3.11 are merged to reduce common access to *code*2[i] as shown in Figure 3.12. This is done by aligning the second loop's boundaries with those of the first. Redundant access to array *code* is reduced by introducing scalar variables

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

(*exc_temp0*, *exc_temp1* and *exc_temp2*) in a round robin fashion. References to array *exc2* are also reduced to only one write and one read through scalar replacement. The array references within the merged loop are reduced to three, which is much less than the original number of references insides the loops in Figure 3.11. This technique introduces four new variables. Similarly this process is applied to the other loops of the code wherever possible (see Appendix A).

```
exc_temp1 = code[0];
L_tmp = L_deposit_h(exc_temp1);
L_tmp = L_msu(L_tmp, code[1], tmp);
L_exc2 = L_mult(round(L_tmp), gain_code);
L_exc2 = L_shl(L_exc2, 5);
L_exc2 = L_mac(L_exc2, exc2[0], gain_pit);
L_exc2 = L_shl(L_exc2, 1);
exc2[0] = round(L_exc2);
exc_temp2 = code[1];

for (i = 1; i < L_SUBFR - 1; i++)
{
  exc_temp0 = exc_temp1;
  exc_temp1 = exc_temp2;
  exc_temp2 = code[i+1];
  L_tmp = L_deposit_h(exc_temp1);
  L_tmp = L_msu(L_tmp, exc_temp2, tmp);
  L_tmp = L_msu(L_tmp, exc_temp0, tmp);
  L_exc2 = L_mult(round(L_tmp), gain_code);
  L_exc2 = L_shl(L_exc2, 5);
  L_exc2 = L_mac(L_exc2, exc2[i], gain_pit);
  L_exc2 = L_shl(L_exc2, 1);
  exc2[i] = round(L_exc2);
}
L_tmp = L_deposit_h(exc_temp2);
L_tmp = L_msu(L_tmp, exc_temp1, tmp);
L_exc2 = L_mult(round(L_tmp), gain_code);
L_exc2 = L_shl(L_exc2, 5);
L_exc2 = L_mac(L_exc2, exc2[L_SUBFR - 1], gain_pit);
L_exc2 = L_shl(L_exc2, 1);
exc2[L_SUBFR - 1] = round(L_exc2);
```

Figure 3.12: After loop merging

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

3.4.3 Loop Unrolling And Scalar Replacement

Filtering operations are very common in signal processing applications. The filtering operation usually takes the form in (3.4.1).

$$y[n] = \sum_{i=0}^{D-1} h[i]y[n-i] = h[0]y[n]+h[1]y[n-1]+...+h[D]y[n-D+1], \quad n = 1, ..., F \quad (3.4.1)$$

This is a forward prediction operation. The filter order is D. F is the frame length. $y[n]$ is the signal, which is computed from the previous D samples. $h[i]$ is the prediction parameter. The C code for this filter operation is shown in Figure 3.13. In the code in Figure 3.13, the array *old_y* and the array *new_y* have different offsets and different lengths. The *old_y*'s length equals the length of the *new_y* plus the last $D-1$ samples in the previous frame. To view the redundant accesses, the double-loop in the original code is unrolled by two unrolling factors, denoted by $u1$ and $u2$. This is shown in Figure 3.14.

```
//old_y starts from the last D-1
//elements of the previous frame.
//new_y starts from the new frame.
for ( i=0; i < F; i++)
{
    sum = 0;
    for (j=0; j < D; j++)
S:          sum = sum + old_y[i+j] * h[j];
    new_y[i] = sum;
}
```

Figure 3.13: Forward prediction operation

The statement S of the original code in Figure 3.13 is unrolled to $u1 \times u2$ statements as shown in Figure 3.14. The unrolled code reveals redundant access to the array *old_y*, represented by those references having same subscript functions ($old_y[i+(j+1)]$ and $old_y[(i+1)+j]$). The references accessing array *old_y* has two loop indices i and j in their subscripts (e.g. $old_y[i+j]$). We shall call this kind of references multiple-index subscripted references which has been introduced in Section 2.3.5 and will be further defined in Section 4.1. The combinations of different values of i and j generates temporal

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

```
c1 = F - F % u1;
c2 = D - D % u2;
for(i=0; i < c1; i += u1)
{
    sum1 = 0;
    .....
    sum(u1-1) = 0;
    for(j=0; j < c2; j +=u2)
    {
        sum1 = sum1 + old_y[i+j] * h[j];
        sum1 = sum1 + old_y[i+(j+1)]*h[j+1];
        ....
        sum1 = sum1 + old_y[i+j+u2-1] * h[j+u2-1];
        sum2 = sum2 + old_y[(i+1)+j] * h[j];
        ....
        sum2 = sum2 +old_y[(i+1)+j+u2-1] * h[j+u2-1];
        ....
        sum(u1-1) = sum(u1-1) + old_y[i+j+u1-1] * h[j];
        ....
        sum(u1-1) = sum(u1 - 1) + old_y[i+j+u1+u2-2] * h[j+u2-1];
    }
    for(j=c2; j < D; j++)
    {
        sum1 = sum1 + old_y[i + j] * h[j]
        ....
        sum(u1-1) = sum(u1-1) + old_y[i + j+u1-1] * h[j]
     }
     new_y[i] = sum1;
     ...
     new_y[i+u1-1] = sum(u1-1);
}
for(i=c1; i < F; i++)
{
      sum1 = 0;
      for(j=0; j < filter_order; j++)
            sum1 = sum1 + old_y[i + j] * h[j];
      new_y[i]=sum1;
 }
```

Figure 3.14: Forward prediction operation loop nest after unrolling

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

reuse to array *old_y* (e.g. $x[1]$ are accessed twice by both $x[0+1]$ and $x[1+0]$). This kind of redundant access can be reduced by unroll-and-jam followed by scalar replacement. The choice of $u1$ and $u2$ for unroll-and-jam depends on the number of available registers and constraints on the unrolled code size. A new systematic method has been developed in this thesis to solve this problem and will be elaborated in Chapter 4. This new method is important for applications such as the WB-AMR speech decoder. For example, the functions *Filt_6k_7k* and *Pred_lt4* (see Appendix A) are filtering operations. These two functions together account for 28.5% of total memory accesses. Hence, it is very useful to have a method to reduce memory access for this type of frequently used operations.

3.4.4 Other Global Transformations

Some parts of the code have been revised such that redundancy in computations and in memory access are eliminated. Although this kind of rewriting is neither inlining nor loop transformations, it crosses the boundaries of functions. It should be referred to as global transformation. The following paragraphs will explain two situations of this kind of global transformation.

The *decoder* function calls two functions, *D_gain2* and *voice_factor*, one after another. Inside both functions, the dot product of array *code* is computed. This redundancy can be illustrated by the pseudo code in Figure 3.15. Neither function is candidate for inlining. Without inlining, redundant computations and access to *code* cannot be eliminated. To achieve this goal, the code is revised to that as shown in Figure 3.16. Here, the *Dot_product* function is moved to the *Decoder* function and executed only once.

There is another kind of redundancy in the WB-AMR decoder code which appears more frequently than the one described above. It is illustrated by the pseudo code in Figure 3.17. The *Oversamp_16k* function interpolates the signal from a sampling rate of 12.8 kHz to 16 kHz. The two *Copy* functions combine the samples from previous subframe and the samples from the current subframe into a third array. Copying the

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

```
Decoder()
{
    Word16 code[L_SUBFR];
    …
    D_gain2(code,…);
    …
    Voice_factor(code,…);
    …
}

D_gain2(Word16 *code, …)
{
     …
     L_tmp= Dot_product(code);
     …
}

voice_factor(Word16 *code, …)
{
     …
     L_tmp= Dot_product(code);
     …
}
```

Figure 3.15: Common operation - *Dot_product*

```
Decoder()
{
    Word16 code[L_SUBFR];
    Word32 L_tmp;
    …
    L_tmp = Dot_product(code);
    D_gain2(L_tmp, …);
    …
    Voice_factor(code, …);
    …
}

D_gain2(Word32 Dot_prod, …)
{
     …
     L_tmp= Dot_prod;
     …
}

voice_factor(Word32 Dot_prod, …)
{
     …
     L_tmp= Dot_prod;
     …
}
```

Figure 3.16: After transformation

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

samples of the current subframe to the new array is regarded as redundant, since the data are already available. In order to remove this redundancy, the code is rewritten to that shown in Figure 3.18. The new code declares a longer array at the start, allocating more space than necessary to store samples in the current subframe. In this way, samples from the previous subframe can be directly copied into the current subframe. After rewriting the code, only one invocation of the *Copy* function is necessary.

```
Void synthesis(…)
{
    Word16 synth[L_SUBFR];

    Oversamp_16k(synth,…);
}

Oversamp_16k(Word16 sig12k8[], …)
{
    Word16 signal[L_SUBFR + (2 * NB_COEF_UP)];

    // copy oversampling filter memory[24] to signal
    Copy(mem, signal, 2 * NB_COEF_UP);
    //copy sig12k8 to signal
    Copy(sig12k8, signal + (2 * NB_COEF_UP), lg);
    …
}
```

Figure 3.17: Redundant copy of data array

```
Void synthesis(…)
{
     Word16 *synth;
     Word16 synth12k8[L_SUBFR + (2 * NB_COEF_UP)];

     Synth = synth12k8 + 2*NB_COEF_UP;
     Oversamp_16k(synth12k8, …);
}

Oversamp_16k(Word16 sig12k8[], …)
{
    // copy oversampling filter memory[24] to signal
          Copy(mem, sig12k8, 2 * NB_COEF_UP);
          …
}
```

Figure 3.18: After transformation

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

3.5 The Profiling Results Of The Optimized WB-AMR Speech Decoder

This section presents the profiling results of the optimized WB-AMR speech decoder. Table 3.5 shows the Atomium profile for the optimized WB-AMR decoder running in mode 0. This is the program obtained after inlining and global transformation of the original program. The number of memory accesses has been reduced from 25.0 million to 18.0 million approximately. This is a 28% reduction. The first ten arrays (sorted by the number of accesses to each array) in Table 3.5 have smaller numbers of accesses than their original numbers of accesses in Table 3.1 except two new arrays, HF_buf and $synth12k8$. For example, the access related to the array Aq is dropped to about half of the original figure. The new two arrays, HF_buf and $synth12k8$, are created during the optimization and account for 19.7% of the total access.

Table 3.4 listed the major functions that generats heavy memory access under mode 0. After optimization, the profiling results of these functions are shown in Table 3.6. It can be seen that significant reduction has been achieved and all the functions are optimized except *Interpol*. The functions *Syn_filt_32*, *Syn_filt*, *HP50_12k8* and *HP400_12k8* are inlined into the *synthesis* function. Even then, the profiling results show that the number of accumulated access to the *synthesis* function still drops from 17181266 to 13029294, a 24.2% reduction.

The optimization results for all 9 modes of the decoder are listed in Table 3.7. The results show that the percentage reduction in memory access is between 27.5% and 31.6%. The reductions of mode 0 and mode 8 are different from those of other 6 modes. This is because their decoding and synthesis processes are a bit different from the other 6 modes.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 3.5: List of arrays sorted by number of accesses

Total accesses:

Accesses	Writes	Reads
18140790	2606315	15534475

Accesses per array:

Accesses	Writes	Reads	Elements	Lives	Name	Location	Type
2073228	497228	1576000	505	197	old_exc	[dec_main.c:126]	Word16
1922720	330960	1591760	100	788	HF_buf	[dec_main.c:768]	Word16
1651648	69344	1582304	88	788	synth12k8	[dec_main.c:767]	Word16
1513080	120	1512960	120	1	fir_up	[decim54.c:28]	Word16
1278136	16548	1261588	21	788	Ap	[dec_main.c:770]	Word16
1245040	86680	1158360	110	788	x	[hp6k.c:44]	Word16
882560	63040	819520	80	788	synth_hi	[dec_main.c:764]	Word16
882560	63040	819520	80	788	synth_lo	[dec_main.c:765]	Word16
821096	13396	807700	68	197	Aq	[dec_main.c:129]	Word16
724991	31	724960	31	1	fir_6k_7k	[hp6k.c:23]	Word16
612440	263168	349272	1	1	st	[dec_main.c:43]	Decoder_State
580608	246016	334592	256	197	exc2	[dec_main.c:133]	Word16
580096	128	579968	128	1	inter4_2	[pred_lt4.c:24]	Word16
493459	126569	366890	64	197	code	[dec_main.c:132]	Word16
359328	108744	250584	11	1576	f1	[isp_az.c:34]	Word32
330960	103228	227732	10	1576	f2	[isp_az.c:35]	Word32
318080	191040	127040	320	1	synth	[decoder.c:29]	Word16
302592	183744	118848	128	788	code2	[ph_disp.c:49]	Word16
221440	88576	132864	64	197	excp	[dec_main.c:143]	Word16
201731	3	201728	3	1	a50	[hp50.c:25]	const Word16
201731	3	201728	3	1	a	[hp400.c:25]	const Word16
151299	3	151296	3	1	b50	[hp50.c:22]	const Word16
151299	3	151296	3	1	b	[hp400.c:22]	const Word16
131596	50432	81164	20	197	Hflsf	[dec_main.c:145]	Word16
80090	25222	54868	14	788	IsfDiff	[isfextrp.c:29]	Word16

Sort by [accesses] [writes] [reads] [elements] [name]
Top-level report
Data collected at Mon May 26 09:43:32 2003.Report generated at Mon May 26 09:45:52 2003.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 3.6: Results on the improvements due to reduction of multiple-index subscripted references

Function	**Local Access**		
	Original	**Optimized**	**Reduced %**
decoder	1076490	866094	19.5
Filt_6k_7k	4097600	2048800	50.0
Syn_filt_32	3430164	inlined	-
Pred_lt4	3329300	1608308	51.7
Interpol	3025920	3025920	0
Syn_filt	2774548	inlined	-
Copy	987040	488704	50.5
Scale_sig	794304	592576	25.4
Dot_product12	529536	201728	61.9
HP50_12k8	463344	inlined	-
HP400_12k8	463344	inlined	-

Table 3.7: Reduction in memory accesses for different modes

Mode	**Orig. Accesses**	**Opt. Accesses**	**Reduction (%)**
0	25021926	18140790	27.5
1	23330696	16763844	28.1
2	22436274	16066076	28.4
3	22465554	16089769	28.4
4	22493862	16113250	28.4
5	22536148	16155028	28.3
6	22561164	16177250	28.3
7	22617641	16229155	28.2
8	26815942	18339204	31.6

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

3.6 Summary

In this chapter, a source level optimization procedure has been presented. The procedure reduces memory access for data-intensive applications through three major steps: profiling, inlining and global transformation. An industrial application, the WB-AMR speech decoder, is used as an example to explain the three-step procedure. Experimental results on the WB-AMR speech decoder confirm that significant improvements can be achieved for data-intensive programs using our method.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Chapter 4

Memory Access Reduction For Multiple-Index Subscripted References

The WB-AMR speech decoder described in Section 3.1 has many filtering operations that involve arrays having multiple-index subscripted references. Multiple-index subscripted reference is a kind of array reference with at least one dimension that involves more than one loop index (e.g. an array reference $x[i + j]$ has two loop indices i and j in its subscript of the only dimension). It is formally defined in Section 4.1. On the other hand, an array reference with each of its dimensions involving only one loop index is referred to as single-index subscripted reference (e.g. $x[i][j]$).

Existing techniques that handle single-index subscripted reference might not be able to optimize memory access involving multiple-index subscripted references. This can be illustrated through the example in Figure 4.1(a). In this example, there is a two-level loop nest. The single-index subscripted reference $x[j]$ is invariant to loop i. To exploit self-temporal reuse from $x[j]$, loop i can be interchanged with loop j, making it the innermost loop as shown in Figure 4.1(b), provided that there is no legality problem

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

arising from data dependency. After this loop interchange, the reference $x[j]$ can be replaced by a scalar as shown in Figure 4.1(c). If the array reference inside the loop nest is a multiple-index subscripted reference such as $x[i+j]$ in Figure 4.1(d), then loop interchange does not help. This is because the subscript $[i+j]$ still involves both loop indices even after loop interchange and therefore cannot simply be replaced by a scalar.

```
for (i = 1; i < 10; i++)
  for (j = 1; j < 10; j++)
    ... = x[j] ...
```

(a)

```
for (j = 1; j < 10; j++)
  for (i = 1; i < 10; i++)
    ... = x[j] ...
```

(b)

```
for (j = 1; j < 10; j++)
{
  tmp = x[j];
  for (i = 1; i < 10; i++)
    ... = tmp ...
}
```

(c)

```
for (i = 1; i < 10; i++)
  for (j = 1; j < 10; j++)
    ... = x[i+j] ...
```

(d)

Figure 4.1: Examples of two kinds of references.

From this simple example, it can be seen that multiple-index subscripted references need to be handled differently. Transformations that are designed suitable for single-index subscripted references are not applicable to this kind of references.

Carr and Kennedy [12] considered variant and invariant references with only one loop induction variable in each subscript position. Carr and Guan [13] limited the scope to not only within single-index subscripted reference, but also to uniformly generated sets. Their methods are designed primarily for single-index subscripted references. Their studies have not considered the situations for multiple-index subscripted references.

A new method that can handle multiple-index subscripted references is described in this chapter. The method attempts to find multiple-index subscripted variables that are exploitable. Conceptually, the exploitation method is simple. It tries to minimize memory accesses by replacing the frequently used multiple-index subscripted variables by temporary scalar variables. However, a transformation chosen in one situation may

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

not be suitable in another situation. Therefore, a more complex process that selects the best transformation to apply is required. The flow of the overall process is shown in Figure 4.2. Given a C program, the first step is to detect any multiple-index subscripted reference that is exploitable. Successful detection leads to three transformations that are performed one after another. These transformations are loop interchange, unroll-and-jam and scalar replacement. Legality test is always performed at the beginning of any transformation. The output of the process is an optimized C program. The process steps are discussed in detail in the following subsections.

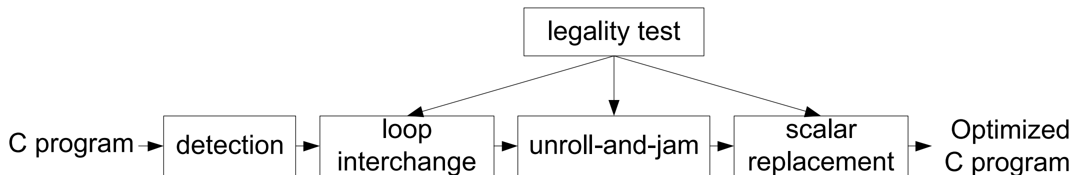


Figure 4.2: The flow of the overall process to exploit Multiple-index subscripted variable

4.1 Properties Of Multiple-Index Subscripted References

A subscripted reference is an array reference inside a loop nest. An array reference can have one or more dimensions, each denoted by a subscript. For example, the array reference $x[i][j+1]$ has two dimensions denoted by subscripts i and $j+1$ respectively. Multiple-index subscripted references refer to those array references with at least one of its dimensions involving more than one loop indices. For example, $x[i+j][j]$ has two loop indices i and j in the subscript of its first dimension.

We define an array reference $x[\vec{f}(\vec{I})]$ to be a general form of multiple-index subscripted reference inside a M-level loop nest as shown in Figure 4.3. The reference has N dimensions. In the nth dimension, a subscript expression $f_n(\vec{I})$ $(1 \leq n \leq N)$ has more than one

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

non-zero coefficients $h_{mn}(1 \leq m \leq M$ and $1 \leq n \leq N)$. That is the subscript value of the nth dimension of the array is determined by more than one loop index. When this is the case, the array reference $x[\vec{f}(\vec{I})]$ is said to be a multiple-index subscripted reference. The subscripts of N dimensions can be expressed as a matrix H which is called an access matrix. The access matrix maps the loop nest's iteration space Z^M to the array accessing space Z^N (Z denotes the entire set of natural numbers). $\vec{c}$ contains the constant terms of all subscripts.

```
for (I1 = b1; I1 < (b1'+1); I1 += t1)
  for (I2 = b2; I2 < (b2'+1); I2 += t2)
     ...
    for (IM = bM; IM < (bM'+1); IM += tM)
       ...x[ f(I) ]...
```

$$\vec{f}(\vec{I}) = \vec{I}H + \vec{c} = [I_1 \quad I_2 \quad \dots \quad I_M] \begin{bmatrix} h_{11} & h_{12} & \dots & h_{1N} \\ h_{21} & h_{22} & \dots & h_{2N} \\ \dots & \dots & \dots & \dots \\ h_{M1} & h_{M2} & \dots & h_{MN} \end{bmatrix} + [c_1 \quad c_2 \quad \dots \quad c_N]$$

or

$$\vec{f}(\vec{I}) = [f_1(\vec{I}) \quad f_2(\vec{I}) \quad \dots \quad f_N(\vec{I})]$$
$$= [I_1h_{11} + I_2h_{21} + \dots + I_Mh_{M1} + c_1 \quad I_1h_{12} + I_2h_{22} + \dots + I_Mh_{M2} + c_2 \quad \dots \quad I_1h_{1N} + I_2h_{2N} + \dots + I_Mh_{MN} + c_N]$$

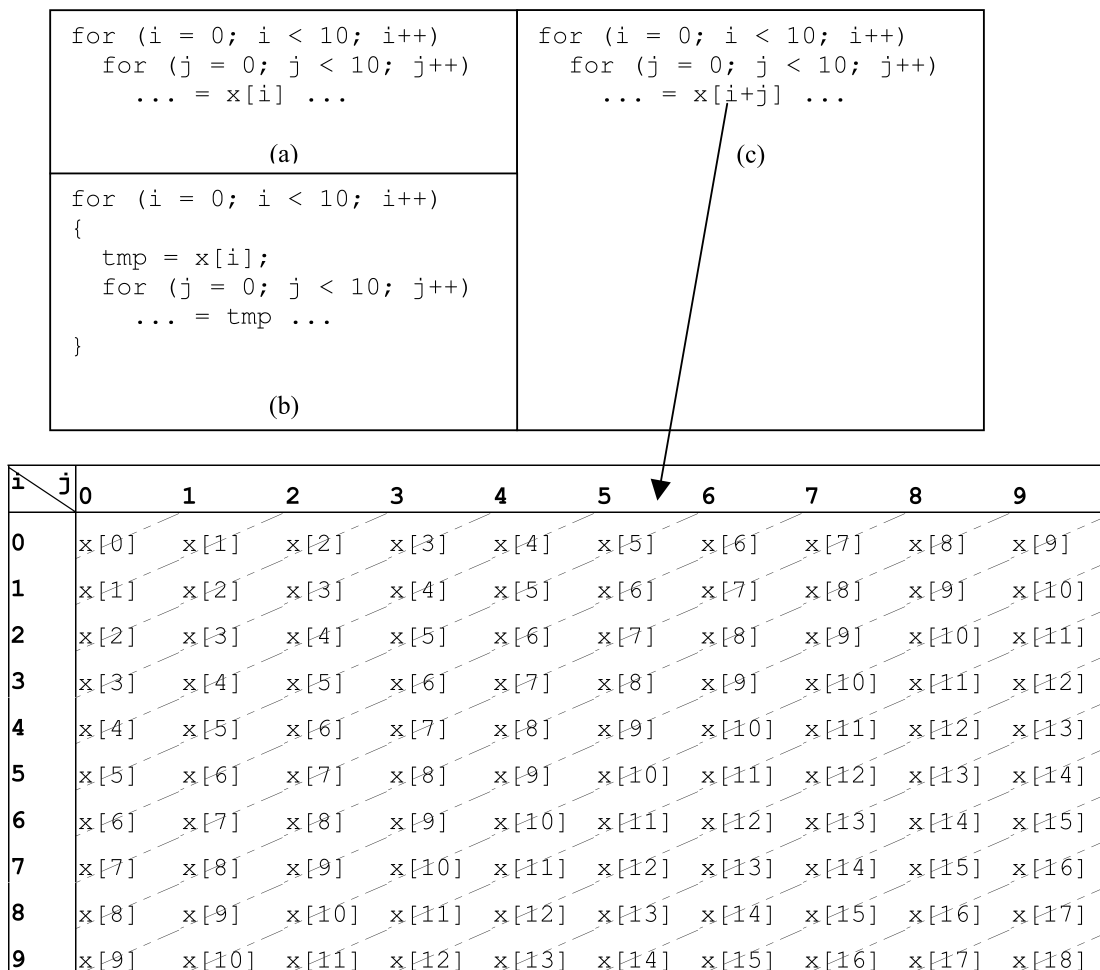
Figure 4.3: General form of multiple-index subscripted reference $x[\vec{f}(\vec{I})]$ in a nest loop.

Temporal reuse may be applied to a multiple-index subscripted reference. These references can be exploited to improve register allocation. Whether a multiple-index subscripted variable can be exploited in this way is determined by both its subscript expressions and the boundary values of the loop nest. This is explained in more detail in Section 4.2.

Temporal reuses generated by a multiple-index subscripted variable are referred to self-temporal reuse. However, this is different from the self-temporal reuse generated by loop-invariant references. In [12, 13, 14, 15, 16], the explorations of self-temporal reuse were on loop-invariant references. Whereas, the self-temporal reuse here results

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

from multiple loop indices. The self-temporal reuse described in this thesis broadens the exploration scope of self-temporal reuse.



i \ j	0	1	2	3	4	5	6	7	8	9
0	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]	x[8]	x[9]
1	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]	x[8]	x[9]	x[10]
2	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]	x[8]	x[9]	x[10]	x[11]
3	x[3]	x[4]	x[5]	x[6]	x[7]	x[8]	x[9]	x[10]	x[11]	x[12]
4	x[4]	x[5]	x[6]	x[7]	x[8]	x[9]	x[10]	x[11]	x[12]	x[13]
5	x[5]	x[6]	x[7]	x[8]	x[9]	x[10]	x[11]	x[12]	x[13]	x[14]
6	x[6]	x[7]	x[8]	x[9]	x[10]	x[11]	x[12]	x[13]	x[14]	x[15]
7	x[7]	x[8]	x[9]	x[10]	x[11]	x[12]	x[13]	x[14]	x[15]	x[16]
8	x[8]	x[9]	x[10]	x[11]	x[12]	x[13]	x[14]	x[15]	x[16]	x[17]
9	x[9]	x[10]	x[11]	x[12]	x[13]	x[14]	x[15]	x[16]	x[17]	x[18]

Figure 4.4: Examples of two kinds of self-temporal reuse.

To illustrate the differences between the two kinds of self-temporal reuse, consider the codes shown in Figure 4.4. Both reference $x[i]$ in Figure 4.4(a) and reference $x[i+j]$ in Figure 4.4(c) generate temporal reuse. However, the ways they generate temporal reuses are different in two aspects. First, the temporal reuse in Figure 4.4(a) is generated by the invariant loop index j, whereas, the temporal reuses in Figure 4.4(c) is generated by both indices i and j as shown in Figure 4.4(d). The temporal reuse generated by $x[i]$ in Figure 4.4(a) can be exploited by introducing a new scalar *tmp* as shown in Figure 4.4(b).

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Second, for every instance of an array reference, the number of access is the same for loop-invariant reference $x[i]$ but varies for multiple-index subscripted variable $x[i+j]$. For example, $x[i]$ has 10 instances from $x[0]$ to $x[9]$. Every instance is accessed 10 times, which is the number of iterations in loop j. In contrast, the number of access to every instance of $x[i+j]$ varies from 1 (e.g. $x[0]$) to 10 (e.g. $x[9]$). This example shows that multiple-index subscripted variables generate self-temporal reuse in a different way from loop-invariant references. In the more general case like that in Figure 4.3, the situation will be more complicated. This requires a systematic approach to detect and exploit multiple-index subscripted variables for temporal reuse.

4.2 Detection Process

Given a program, the compiler should identify not only the multiple-index subscripted references but also their ability to generate temporal reuse. This process step starts from the array reference's subscript expressions, which can be expressed as a matrix called the access matrix, H (defined in Figure 4.3).

For a loop-invariant reference, the access matrix H contains one or more rows of zeros. This indicates that some loop indices do not appear in any subscript expression of the reference. This evidence of temporal reuse is easy to detect. In contrast, the situation involving a multiple-index subscripted variable is so simple. An array reference of the general form $x[\vec{f}(\vec{I})]$ has self-temporal reuse when any two of its instances, $x[\vec{f}(\vec{i})]$ and $x[\vec{f}(\vec{j})]$, access the same datum in different iterations $\vec{i}$ and $\vec{j}$. This can be expressed mathematically as follows.

$$
\begin{aligned}
&\because \vec{f}(\vec{i}) = \vec{f}(\vec{j}) \text{ and } \vec{i} \neq \vec{j} \\
&\quad \vec{i}H + \vec{c} = \vec{j}H + \vec{c} \\
&\quad (\vec{i} - \vec{j})H = \vec{0} \\
&\quad \text{Let } \vec{d} = \vec{i} - \vec{j}
\end{aligned}
$$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

$$\therefore \vec{d}H = \vec{0} \tag{4.2.1}$$

The vector $\vec{d}$ in Equation 4.2.1 is the distance vector which counts the number of iterations between $\vec{i}$ and $\vec{j}$. (The definition of distance vector was introduced in Section 2.3.2.) The reference $x[\vec{f}(\vec{I})]$ has self-temporal reuse if there exists at least one non-zero solution for $\vec{d}$ in Equation 4.2.1. Thus the problem of detecting exploitable multiple-index subscripted reference is equivalent to solving for the distance vector. Before solving Equation 4.2.1, we should determine wether any solution exists for $\vec{d}$. This can be done by comparing the rank of access matrix H, denoted as $rank(H)$, with the depth of the loop nest M. Since a multiple-index subscripted reference can also be a loop-invariant reference, in general, $rank(H)$ should be compared with the depth of the loop nest M less the number of invariant loops. Assuming the number of invariant loops to be p, the comparison yields the following possible cases:

- $rank(H) = M - p$: no temporal reuse is generated by the multiple indices, $\vec{d} = \vec{0}$.
- $rank(H) > M - p$: this situation should not exist in a correctly written program. There is no reuse in this case, and $\vec{d}$ has no solution.
- $rank(H) < M - p$: the reference can generate temporal reuses with its loop indices. $\vec{d}$ has one or more solutions. Note that these temporal reuses are not necessarily exploitable. These reuses should be exploitable for real embedded codes.

$rank(H)$ can be computed using Echelon reduction [28, 35]. If $rank(H) < M - p$, the distance vector should be computed. Comparing this distance vector with the loop boundaries ($\vec{b}$ and $\vec{b'}$), we can determine whether the reference generates reuses in the loop nest. If temporal reuses exist, high-level transformations can then be applied to expose the reuse.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

4.3 Legality Test

When temporal reuses are found, high-level transformations are to be performed. Thus, the legality issue of high-level transformations arises. A legality problem is caused by the loop-carried dependence that involves write access. No transformation is allowed to violate data dependency in the original code. Traditionally, distance vector and direction vector are used to represent data dependency in loop transformations [28, 36, 37] as discussed in Section 2.3.2. They are proven to be very effective. More specifically, in order to ensure the legality of transformations in our method, three legality rules listed below should be observed (the following discussions are based on the knowledge in both transformations from Section 2.2 and data dependency from Section 2.3).

(i) A necessary condition for a legal transformation is that the polarities of all the distance vectors in a loop nest remain unchanged. That is, a lexicographically positive (negative) distance vector should remain positive (negative). This rule guarantees no violation of data dependency. For example, the legality of loop interchange can be determined by comparing the polarities of the direction vectors before and after the transformation.

(ii) For unroll-and-jam, unrolling is always legal while jamming might change the data dependency [12]. There is a range of values to unroll-and-jam each individual loop. Basically, the lower limit of the range is the minimal incremental step size of the loop, meaning no unrolling is allowed. The upper limit of the range is the number of iterations of the loop. It sets the maximal incremental step size of the loop. However, data dependency in a loop nest might restrict the range to somewhere between these limits. For example, for a flow, output or anti-dependence carried at level 2 (level is the position of the first non-zero element) with a distance vector $[0 \quad 3 \quad -1]$, the second loop cannot be unrolled by any integer value greater than

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

or equal to 3. Otherwise, the dependence -1 carried at the inner most loop will be exposed and prevents loop fusion (jamming). This is because -1 has a negative sign and change the polarity of the original data dependency.

(iii) That is to say, we can unroll-and-jam a loop to any amount not great than the iterations of the loop if it can be interchanged with the innermost loop without violating any data dependency [15]. Otherwise, rule (ii) above applies.

Note that the first rule is the most important one during any transformation. It is also the basis of the other two rules.

4.4 Loop Interchange And Related Tests

Loop interchange is a loop transformation that permutes the loops inside a loop nest [38, 28]. It is performed before unroll-and-jam. Loop interchange itself is not helpful in revealing reuse from multiple loop indices. It is performed for loop-invariant references instead of multiple-index subscripted references. Loop-invariant reference is exploitable only when its invariant loop is the innermost loop. Although this method is to exploit multiple-index subscripted variable, the occurrence of loop-invariant reference makes loop interchange necessary.

To determine whether loop interchange is necessary, the followings procedure is performed. First, identify all the reference-invariant loops that can legally be shifted to the innermost loop. Then compute the amount of exploitable reuses for each of these loops. The reference-invariant loop that generates maximum number of reuses will be chosen to be the innermost loop if it is not already one. If reference-invariant loop does not exist in the loop nest, no loop interchange will take place.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

4.5 Unroll-And-Jam

Unroll-and-jam is a kind of loop transformation that unrolls a loop body several times [13, 15]. It can be used in conjunction with scalar replacement to improve register allocation [16]. Although unrolling multiple loops does not cause any legality problem, jamming the unrolled loop body into the innermost loop is not always legal. Legality test has to be performed for unroll-and-jam by following the legality rules in Section 4.3. For an outer loop, unroll-and-jam is allowed when the outer loop can be interchanged to the innermost loop (legality rule (iii)). Otherwise, rule (ii) applies.

The most important problem in unroll-and-jam is to determine the unrolling vector of the loop nest [39, 40, 41]. For a loop nest with depth M, the unrolling vector is denoted as loop tiling size $\vec{v} = [v_1 \quad v_2 \quad \ldots \quad v_m \quad \ldots \quad v_M]$. In the area of applying high-level transformations to improve register allocation, Callahan et al. [16] showed how to enhance scalar replacement by unroll-and-jam. Later on, Carr and Kennedy [12] applied unroll-and-jam to improve the balance of loops. However, they do not present a method to compute unroll amounts automatically. More recently, Carr and Guan [13] improved unroll-and-jam by using the reuse model proposed in [27] and its associated linear algebra framework. At the same time, they also limit their scope to uniformly generated set [27], i.e. the subscript expressions of any two array references in the group are only different with a constant term, e.g. $x[i]$, $x[i+1]$ and $x[i+5]$. [13] left the task of selecting unrolling factors to the compiler rather than the programmer, meaning more automatic. With the understandings of the multiple-index subscripted variables, we develop a parameter estimation model to determine the unrolling vector for loop nests containing these variables. This estimation model takes into account the distance vectors of the reuses, the boundaries and the incremental step size of the loop nest, and the number of available registers. In our method, the extended code size from unroll-and-jam is not considered since it is not within the scope of this thesis. However, extended code size can be incorporated into our method without much modification.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

4.5.1 Parameters Estimation For Optimal Unrolling

In order to determine the optimal unrolling vector for a given loop nest, a number of parameters has to estimated. These parameters are:

- The total number of reuse exposed by unroll-and-jam, Ne. Reuse is the read after the first access.
- The number of scalar variables required by scalar replacement, Nr. The number of registers required cannot be greater than the number of available registers to prevent register spilling.
- The total number of memory access after the optimization, Na. This computes the total number of memory accesses after scalar replacement in round robin fashion.
- The ratio of the memory access after and before the optimization, R. This parameter reflects the effect of optimization. This is the objective to minimize.

During the computations, the following three conditions are assumed to be satisfied.

(i) Reuse exists only when the distance vector $\vec{d}$ is the multiple of the incremental step size vector $\vec{t}$. This is the necessary condition for a multiple-index subscripted variable to generate temporal reuse.

(ii) Unrolling vector $\vec{v}$ should always be a multiple of the incremental step size $\vec{t}$. This relation ensures the correctness of the unrolled loop nest.

(iii) A multiple-index subscripted variable's reuse is revealed only when $(v_m - t_m) \geq d_m$ for all $d_m \neq 0$. The condition implies that loop unrolling can be applied if $v_m > t_m$. When $v_m = t_m$ there is no unrolling.

Total number of reuse N_e

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

The total number of reuse (N_e) includes the reuse within one iteration and the reuse across iterations after applying unroll-and-jam. Both types of reuse exist in the code in Figure 4.5(a). These temporal reuses are exposed in Figure 4.5(b) after unroll-and-jam. (Both inner loop and outer loop are unrolled 3.) For example, the two references $x[i+j+3]$ having the same subscript expression generate reuse within an iteration. Another type of reuse is from $x[i+j+1]$ and $x[i+j+4]$ since these two references have data dependence across an unrolled iteration. To remove both types of reuses in the loop nest, a round robin fashion of scalar replacement is applied as shown in Figure 4.7(b). Scalar replacement is discussed in more details in Section 4.6.

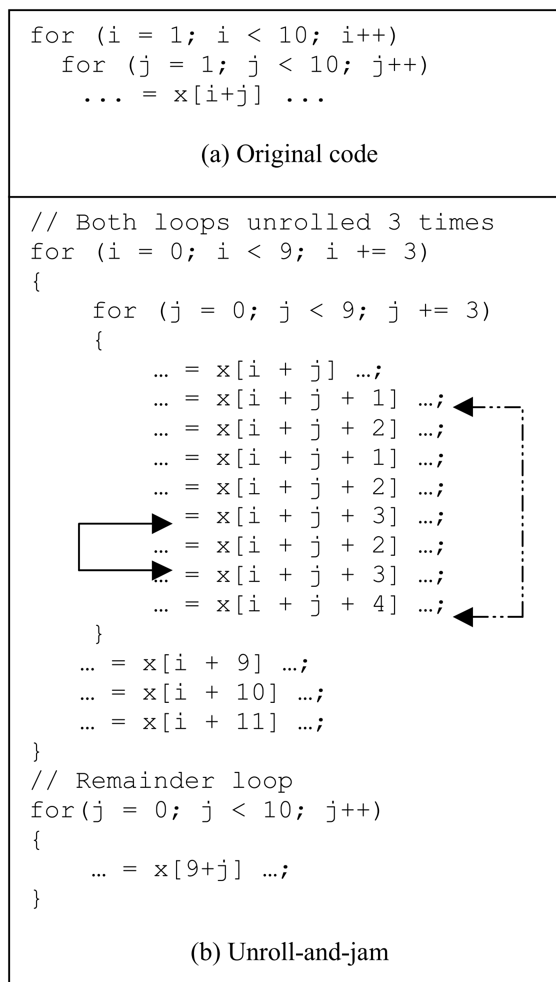


Figure 4.5: Reuse within one iteration and reuse across iterations.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

$$\prod \left(\frac{v_m - |d_m|}{t_m} \right) \tag{4.5.1}$$

Self-temporal reuse generated by a multiple-index subscripted reference might result from several loops. Let L be the number of these loops. For a loop m, $(v_m - |d_m|)/t_m$ is the number of reuse carried by this loop. This expression measures the difference of loop m's unrolling factor v_m and the distance of reuse carried by loop m in terms of the incremental step size t_m. For L number of loops, expression (4.5.1) is the product of reuses exposed by these loops. For example, array reference $x[i][j + 2k]$ has distance vector $\vec{d} = \begin{bmatrix} 0 & 2 & -1 \end{bmatrix}$. There are two loops carrying the dependence, i.e. $L = 2$. The incremental step size for $\begin{bmatrix} i & j & k \end{bmatrix}$ is $\vec{t} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$. The unrolling vector is chosen to be $\vec{v} = \begin{bmatrix} 0 & 5 & 3 \end{bmatrix}$. Hence, the value of (4.5.1) in the unrolled iteration is $(5 - 2) \times (3 - 1) = 6$.

The amount of reuse across iterations by scalar variables is computed as expression (4.5.2). It counts the number of data shifts among scalar variables in an iteration. h_m is the coefficient of loop m in the subscript expression of a reference. It should be noticed that loop M is the innermost loop. It is assumed that the array reference is originally located in the innermost loop. Using the $x[i][j + 2k]$ example again, the value of (4.5.2) is $((5 - 1) \times 1 + (3 - 1) \times 2 - 3)/1 + 1 = 6$. With both the amount of reuse inside an iteration (4.5.1) and the amount reuse across iterations (4.5.2) computed, the total number of reuses, Ne, is computed as in Equation 4.5.3. N_m^{new} denotes the number of iterations for the unrolled loop m. N_m^{new} is computed by Equation 4.5.4. $(b'_m + 1)$ is the upper bound of the original loop m. v_m is the unrolling factor for loop m.

$$\frac{\sum \left(\frac{v_m - t_m}{t_m} \times |h_m| \right) - v_M}{t_M} + 1 \tag{4.5.2}$$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

$$N_e = \left(\prod \left(\frac{v_m - |d_m|}{t_m} \right) + \frac{\sum \left(\frac{v_m - t_m}{t_m} \times |h_m| \right) - v_M}{t_M} + 1 \right) \times \prod_{m=1}^{M} N_m^{new} \tag{4.5.3}$$

where

$$N_m^{new} = \frac{((b'_m + 1) - (b'_m + 1)\% v_m)}{v_m} \tag{4.5.4}$$

Number of scalar variables N_r

Beside the estimation of reuses, the number of scalar variables (N_r) used for scalar replacement has to be computed. This is to ensure that the transformation would not require more registers than that available. The number of scalar variables introduced by scalar replacement is computed by summing up the number needed for reuse within an iteration and that across iterations. N_r can be easily estimated by a data dependence graph for the loop nest after unroll-and-jam (as in Figure 4.5(b)).

Number of memory accesses N_a

The number of memory access (N_a) refers to the number of array reference access after optimization. It includes two parts. The first part is the original amount of memory access less the scalar variable access in the new loop nest. The second part is the additional memory access for the initializations of scalar variables. N_a is calculated by Equation 4.5.5, where N_m is the number of iterations in loop m before loop unrolling, N_i is the number of array reference accesses during the initialization of scalar variables, and N_e is the reuse computed in Equation 4.5.3 which is the amount of scalar variable access.

$$N_a = \prod_{m=1}^{M} N_m - N_e + N_i \tag{4.5.5}$$

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

where

$$N_m = \left\lfloor \frac{b'_m - b_m}{t_m} \right\rfloor + 1$$

Ratio R

The ratio of the memory access before and after optimization is denoted by R. This ratio indicates the effectiveness of different unrolling vectors. The ratio is computed by dividing the remaining amount of memory accesses, Na, by the original amount of memory access as given by Equation 4.5.6.

$$R = \frac{N_a}{\prod_{m=1}^{M} N_m} \tag{4.5.6}$$

4.5.2 Unrolling vector

This section presents the algorithm for selecting an optimal unrolling vector based on the parameter estimations. The algorithm is presented in Figure 4.6.

The first two steps apply the detection process to find the distance vector sets D_{total} and D_{reuse}. Set D_{total} contains all the distance vectors of a given loop nest. It is for legality test in the next steps and to ensure all the data dependency are preserved. Set D_{reuse} contains all distance vectors having constant elements ($D_{reuse} \subseteq D_{total}$). This is to seek for the opportunities to apply loop interchange, unroll-and-jam and scalar replacement. If $D_{reuse} = 0$, there is no space for scalar replacement and the procedure ends.

Step 3 is to find out the loops carrying data dependency in D_{reuse}. If a loop carries data dependency in D_{reuse}, this loop is a candidate for loop interchange and unroll-and-jam. Hence the legality of permuting this loop to the innermost loop is tested against all dependency in D_{total}. Step 4 is to find the maximum unrolling factor, $v^{\max}$, for all

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

1. Compute all the distance vectors in a given loop nest, forming the distance vector set D_{total}.
2. From D_{total}, pick out the distance vectors with constant values and classify them to a new set D_{reuse} ($D_{reuse} \subseteq D_{total}$). If $D_{reuse} = 0$, the algorithm ends.
3. For each loop *m*, if there is a distance vector $\vec{d} \in D_{reuse}$ and its element $d_m \neq 0$, the legality of permuting loop *m* to the innermost loop is tested against all $\vec{d} \in D_{total}$ (legality rule 1).
4. Compute the maximum unrolling factor $v^{\max}$ for every loop carrying temporal reuses (legality rules 2 & 3).
5. If all the non-zero elements of $\vec{d} \in D_{reuse}$ are less than the corresponding maximum unrolling factors $v^{\max}$, $\vec{d}$ is a legal temporal reuse and should remain in the set D_{reuse}. Otherwise, $\vec{d}$ is taken out from $\vec{d} \in D_{reuse}$. If $D_{reuse} = 0$, the algorithm ends.
6. Find out the reference-invariant loop by examining D_{reuse} and access matrix *H*. If this kind of loops exists, go to step 7. Otherwise, go to set 8.
7. Select an invariant loop, assume it is the innermost loop and compute amount of accesses generated. The invariant loop with the minimal amount of accesses and satisfying legality rule 1 is permuted to be the innermost loop.
8. For every loop carrying legal temporal reuses, its unrolling factor v is initialized to the corresponding t_m. The minimal ratio R^{min} is initialized to 1. The optimum unrolling factor v^{opt} is initialized to t_m.
9. Assuming there are L loops carrying legal temporal reuses,

```
for m=1, to L
    for v_m = t_m to v_m^max
            compute the number of reuses, N_e.
            compute the number of registers, N_r.
            if (Nr < the number of registers available)
                    break;
            endif.
            compute the total number of memory accesses ,N_a.
            compute the number of memory access ratio R.
            if (R < R^opt || ((R==R^opt) && ( (v_1 × v_2 ... × v_m ... × v_L) < (v_1^opt × v_2^opt ... × v_m^opt ... × v_L^opt) )))
                    R^opt = R.
                    Optimal unrolling vector is assigned with the current unrolling vector.
            endif
    endfor
endfor
```

Figure 4.6: Algorithm of selecting optimal unrolling vector.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

the loops carrying data dependency in D_{reuse}. If a loop passes the test in step 3, its $v^{\max}$ depends on the boundary values of the loop. Otherwise, legality rule 2 is applied to the loop to find the value of $v^{\max}$.

Step 5 is to check any exploitable distance vector in D_{reuse}. For a distance vector $\vec{d} \in D_{reuse}$, if all its non-zero elements are less than their corresponding unrolling factors $v^{\max}$, $\vec{d}$ is exploitable. If set D_{reuse} has no exploitable distance vector, the procedure ends.

Step 6 and 7 select the loop to be the innermost loop. Step 6 finds all reference-invariant loops. Step 7 decides which reference-invariant loop will generate minimal amount of memory accesses if being innermost. This loop is then permuted to be the innermost loop if not illegal. If such a loop does not exist, no loop interchange is performed.

Step 8 does some initialization work for the last step. It initializes both the unrolling vector (for searching) and the default optimal unrolling vector to the original incremental step size vector $\vec{t}$. Step 9 finds the optimal unrolling vector iteratively. It performs the measurements for every possible unrolling vector. The unrolling vector leading to minimal memory accesses under the register restriction will be chosen. If there is more than one such unrolling vector, the one with smaller code size will win. If no unrolling vector satisfies the register restriction, the original incremental step size is the optimal unrolling vector. The last step is to perform scalar replacement to store all the reused data into registers.

4.6 Scalar Replacement

Scalar replacement replaces subscripted variables (array references) by temporary scalar variables to effect reuse [12, 16]. Scalar replacement is the last transformation to be performed in our method. This transformation increases the register usage for most

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

normal compiler with coloring-based register allocators and is the most effective way of reducing memory operands [18]. Register operand also has shorter run times due to the reduction of potential stalls and cache misses.

When applying scalar replacement, there is the issue of legality. Scalar replacement is performed on variables having flow dependence (read-after-write) or input dependence (read-after-read). When the reuse is flow dependent, it should ensure that there is no data update between the write and the read of the datum. When the reuse is input dependent, scalar replacement will not cause any legality problem. For multiple-index subscripted variables, the reuse generated by multiple loop indices is self-temporal reuse, which is always input dependence. Hence, there is no legality problem during scalar replacement.

When performing scalar replacement after unroll-and-jam, the temporary scalar variables are used in a round robin fashion. For example, the code in Figure 4.7(a) can be optimized in such a way. Firstly, a loop transformation unroll-and-jam is performed. This transformation is to expose the temporal reuse generated by the multiple-index subscripted array reference $x[i + j]$. In Figure 4.7(b), both loop i and loop j are unrolled by an unrolling factor of 3. Temporal reuse happens in the references having the same subscripts. This introduces redundant accesses which are then reduced by scalar replacement. In Figure 4.7(c), scalar replacement is applied in a round robin fashion by introducing temporary variables $x0$, $x1$, $x2$, $x3$ and $x4$. The assignments of $x3$ to $x0$ and $x4$ to $x1$ clear the data in $x3$ and $x4$, thus $x3$ and $x4$ can be loaded with new values. To enable the round robin fashion, $x3$ and $x4$ are initialized in the outer loop. In the outer loop, there is no data passing between $x3$ and $x4$, since the distance of two array references carried by loop i is 1, which is less than the incremental step size of 3. Hence, the round robin fashion is not applied in the outer loop. From this example, it can be see that the round robin fashion is applied from the innermost loop towards outer level of loops, and it stops at the loop whose incremental step size is bigger than the distance

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

```
for (i = 1; i < 10; i++)
  for (j = 1; j < 10; j++)
   ... = x[i+j] ...
```

(a) Original code

```
// Both loops unrolled 3 times
for (i = 0; i < 9; i += 3)
{
    for (j = 0; j < 9; j += 3)
    {
        … = x[i + j] …;
        … = x[i + j + 1] …;
        … = x[i + j + 2] …;
        … = x[i + j + 1] …;
        … = x[i + j + 2] …;
        … = x[i + j + 3] …;
        … = x[i + j + 2] …;
        … = x[i + j + 3] …;
        … = x[i + j + 4] …;
    }
   … = x[i + 9] …;
   … = x[i + 10] …;
   … = x[i + 11] …;
}
// Remainder loop
for(j = 0; j < 10; j++)
{
    … = x[9+j] …;
}
```

(b) Unroll-and-jam

```
for (i = 0; i < 9; i += 3)
{
    x3 = x[i];
    x4 = x[i + 1];
    for (j = 0; j < 9; j += 3)
    {
        x0 = x3;
        x1 = x4;
        x2 = x[i + j + 2];
        x3 = x[i + j + 3];
        x4 = x[i + j + 4];
        … = x0 …;
        … = x1 …;
        … = x2 …;
        … = x1 …;
        … = x2 …;
        … = x3 …;
        … = x2 …;
        … = x3 …;
        … = x4 …;
     }
    … = x[i + 9] …;
    … = x[i + 10] …;
     … = x[i + 11] …;
}
for(j = 0; j < 10; j++)
{
    … = x[9+j] …;
}
```

(c) Scalar replacement

Figure 4.7: Scalar replacement after unroll-and-jam.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

value carried by it. When performing the scalar replacement in the round robin fashion, the number of array references is greatly reduced. This is because the data with lifetime across iterations are passed through scalar variables.

One further issue with scalar replacement is the number of temporary scalar variables generated. Since the scalar variables will be allocated to register by normal compilers with coloring-based register allocators, the number of scalar variables is restricted by the number of available registers. Under this restriction, the number of reuse exposed by unroll-and-jam determines the amount of temporary scalar variables, as explained in Section 4.5.

4.7 Complexity

The algorithm has to first detect the distance vectors in a loop nest. Specifically, the process of searching multiple-index subscripted variable is found in $O(n)$ time, where n is the number of subscripted array references. After detection, the algorithm performs the test for loop interchange. This test is usually fast and requires $O(M)$ time, where M is the depth of the loop nest. Following loop interchange, the algorithm searches the loop index space for optimal unrolling vector based on the parameter estimations. This process requires $O(MK)$ time, where M is the depth of a loop nest and K is the number of iterations in a loop. In practice, the algorithm runs surprisingly fast. This might be due to the acceptable complexity of practical applications.

4.8 Signal Processing Benchmarks

Our proposed method is applied to the six benchmark programs obtained from the signal processing library of Texas Instruments Inc. [42]. A summary of these programs is shown in Table 4.1. These benchmarks are typical array-intensive programs which involve heavy data accesses inside loop nests. Moreover, the array references in these programs are

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

multiple-index subscripted references. The purpose of our experiments described are to (1) measure the performance benefits of the register allocation strategy presented in previous sections; (2) understand the effectiveness of the method on different kind of multiple-index subscripted references; and (3) determine the accuracy of the parameter estimations model in the method. The experiment results are obtained by executing the benchmarks on "simplescalar" [43], a tool which profiles loads and stores of references.

Table 4.1: Benchmarks from signal processing library

Benchmark	**Description**
DSP_autocor	This routine performs an autocorrelation of an input vector.
DSP_fir_cplx	This complex FIR computes complex output samples using complex coefficients.
IMG_corr_gen	This routine performs a generalized correlation with a 1 by M tap filter.
DSP_minerror	Performs a dot product on 256 pairs of 9 element vectors and searches for the pair of vectors which produces the maximum dot product result.
IMG_mad_8x8	This routine returns the location of the minimum absolute difference between a 8x8 search block and some block in a (h + 8) x (v + 8) search area.
IMG_mad_16x16	This routine returns the location of the minimum absolute difference between a 16x16 search block and some block in a (h + 16) x (v + 16) search area.

4.8.1 Performance

To measure the performance of the method, the original code, and the transformed code of the benchmarks are executed on simplescalar respectively. The simulator profiles the total number of loads and stores executed. By improving register allocations using our method, the number of loads and stores should be reduced. Table 4.2 shows the number of loads and stores for the original benchmark programs.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 4.2: Number of loads and stores of original code

Benchmark	**Load/Store**
dsp_autocor	810
dsp_fir_cplx	1640
img_corr_gen	14220
dsp_minerror	4608
img_mad_8*8	4114
img_mad_16*16	2033602

The transformation is restricted by the number of available registers. For the sake of evaluation, the number of registers is set to 32. However, the principle can be applied to other values depending on the number of registers available in a particular processor. For each program, we perform loop interchange (if any), unroll-and-jam and scalar replacement. For the step of unroll-and-jam, the optimal unrolling vector of each program is determined by using the iterative search algorithm presented in Figure 4.6. The results are shown in Table 4.3.

Table 4.3: Results after transformations

benchmark	**unrolling vector**	**load/store**	**reduced by %**	**No. of reg.**
dsp_autocor	[32 1]	362	55.2	32
dsp_fir_cplx	[26 4]	597	63.6	30
img_corr_gen	[28 1]	3356	77.4	29
dsp_minerror	[32 1]	1929	58.1	30
img_mad_8*8	[12 1 1 1]	2407	41.5	13
img_mad_16*16	[16 1 1 1]	1124581	44.7	17

The results show that various amounts of reductions of load/store have been achieved for different benchmark programs. It is interesting to note that the unrolling vectors have minimal incremental step sizes for the inner loops and bigger step sizes for the outer loops. This is due to the round robin fashion of scalar replacement. For example,

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

in the program *dsp_autocor*, a multiple-index subscripted reference $x[k-i]$ in Figure 4.8(a) possess reuses. After unroll-and-jam with an unrolling vector [32 1], the program looks like Figure 4.8(b). Without unrolling loop k, there is no reuse within one iteration. However, there are nine reuses across iterations. This is shown in Figure 4.8(c). By applying these transformations, the reads of references are moved from an inner loop to an outer loop. Hence, the number of load/store is reduced.

The round robin fashion of scalar replacement tends to produce bigger unrolling factors for outer loop and smaller unrolling factors for inner loop. However, the selected unrolling vector is based on the evaluation of all possible unrolling vectors. This can be seen from the evaluation results of the *dsp_autocor* in Table 4.4. For clarity, only a small portion of the selection results is shown. It can be seen that method does evaluate the unrolling of inner loop. It selects the vector [32 1] because it has the minimal memory access ratio R comparing to the original program. Another benefit of round robin fashion of scalar replacement is that it leads to a smaller amount of unrolling. For example, the vector [32 1] produces an extended code with a factor of 32×1, which is much smaller than the factors of many other unrolling vectors such as [26 3]'s factor 26×3. One important fact of round robin fashion of scalar replacement is that it introduces register-to-register transfers. It can be seen that the vector [32 1] leads to the largest number of register-to-register transfers.

To understand the accuracy of the estimation model in the method, unrolling vectors are randomly selected to transform the program *dsp_autocor*. Their results from the simulator are compared with those from the selection process. This is shown in Table 4.5. The results show that estimated access ratio matches the practical access ratio with high accuracy.

In Section 4.1, it is mentioned that not all multiple-index subscripted references permit reuse. Reuse is determined by both a reference's subscript expressions and the boundary values of the loop nest. In the experiment, the benchmark *dsp_minerror* has this kind of

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

```
for (i = 0; i < nr; i++)
{
    ….
    for (k = nr; k < nx+nr; k++)
    {
        ...=... x[k-i];
    }
    ...
}
```

(a)

```
for (i = 0; i < nr; i+=32)
{
    ...
    for (k = nr; k < nx+nr; k++)
    {
        ...=... x[k-i];
        ...=... x[k-i-1];
        ...=... x[k-i-2];
        ...=... x[k-i-3];
        ...=... x[k-i-4];
        ...=... x[k-i-5];
        ...=... x[k-i-6];
        ...=... x[k-i-7];
        ...=... x[k-i-8];
        ...=... x[k-i-9];
        ...

    }
    ...
}
```

(b)

```
for (i = 0; i < nr; i+=32)
{
   x8 = x[nr-i-9];
   x7 = x[nr-i-8];
   x6 = x[nr-i-7];
   x5 = x[nr-i-6];
   x4 = x[nr-i-5];
   x3 = x[nr-i-4];
   x2 = x[nr-i-3];
   x1 = x[nr-i-2];
   x0 = x[nr-i-1];
   ...

    for (k = nr; k < nx+nr; k++)
   {
       x9 = x8;
       x8 = x7;
       x7 = x6;
       x6 = x5;
       x5 = x4;
       x4 = x3;
       x3 = x2;
       x2 = x1;
       x1 = x0;
       x0 = x[k-i];
       ...=...x0;
       ...=...x1;
       ...=...x2;
       ...=...x3;
       ...=...x4;
       ...=...x5;
       ...=...x6;
       ...=...x7;
       ...=...x8;
       ...=...x9;
       ...

    }
    ...
}
```

(c)

Figure 4.8: A for loop in *dsp_autocor*

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Table 4.4: Selection of unrolling vector

unrolling vector	**Ratio**	**No. of reg.**	**Reg. to reg. transfer**
[16 14]	0.677	30	190
[16 16]	0.677	32	170
[18 11]	0.657	29	255
[18 12]	0.657	30	240
[18 13]	0.633	31	223
[18 14]	0.645	32	210
[26 1]	0.617	26	390
[26 2]	0.617	28	344
[26 3]	0.610	30	289
[30 1]	0.510	30	440
[30 2]	0.507	32	360
[32 1]	0.429	32	580

Table 4.5: Comparison of estimation and simulation

Unrolling vector	**Estimated R**	**Simulator R**
[12 18]	0.533	0.549
[15 15]	0.567	0.586
[16 14]	0.658	0.672
[26 3]	0.610	0.599
[32 1]	0.429	0.448

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

reference.

```
for (i = 0; i < GSP0_NUM; i++)
{
    for (val = 0, j = 0; j < GSP0_TERMS; j++)
        val += GSP0_TABLE[i*GSP0_TERMS+j] * errCoefs[j];
    ...
}
```

Figure 4.9: *dsp_minerror*

In Figure 4.9, reference $GSP0_TABLE[i * GSP0_TERMS + j]$ is a multiple-index subscripted reference. $GSP0_TERMS$ is a constant value. It has a distance vector $[1 \quad -GSP0_TERMS]$. However, the inner loop j has a boundary value of $GSP0_TERMS$, which restrict the value of loop index j to be less than $GSP0_TERMS$. This prohibits any reuse from the reference $GSP0_TABLE$.

Another reference $errCoefs[j]$ is a reference which is invariant with loop i. It seems that loop i should be the innermost loop to utilize the reuses of $errCoefs[j]$. By doing so, more temporary variables are generated for the scalar val. A compromise is to unroll the outer loop i to a certain level such that it utilizes the reuses from the reference $errCoefs[j]$ without introducing too many temporary variables.

From the analysis of the results, it can be concluded that our method can achieve significant memory access reduction for programs with multiple-index subscripted references as long as reuse exists. The amount of reduction is determined by the amount of reuse inherent to the computations, the number of temporary registers available and data dependence in the program. This method has also been applied to the WB-AMR speech decoder as well. In Section 3.4.3, the filtering operations in functions *Filt_6k_7k* and *Pred_lt4* involve multiple-index subscripted references as shown in Figure 3.14. Applying the method to these functions contributes about 15.1% towards the total memory access reduction.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Chapter 5

Conclusions

In the thesis, a systematic three-step optimization procedure has been developed for reducing data access for a given data-intensive application software. It is particularly useful for multimedia processing software such as the WB-AMR speech decoder. The three steps are profiling, inlining and global transformation. Memory related bottlenecks of the program is first revealed through profiling. After identifying the bottleneck functions, they are inlined to enlarge the exploration space. A call-based inlining scheme is adopted. The criteria of inlining is that it should facilitate the global transformation step within acceptable code size limits. The global transformation step provides the real improvements to the program. In this step, different combinations of fundamental transformations are applied to the WB-AMR speech decoder. The most effective transformations, in terms of the reduction in memory access, are loop merging, loop unrolling and scalar replacement. The transformations optimize the code under constraints such as code size, number of registers available and legality of the transformations. By balancing the performance gain and the constraints, transformation parameters such as unrolling factor are determined.

The effectiveness of this procedure has been demonstrated through applying it to the WB-AMR speech decoder. The results of the optimized program for all nine modes of the WB-AMR speech decoder is reported. The reduction in memory access varies from

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

27% to 31%. This result is achieved without rigorous optimizations.

A new automated method to exploit multiple-index subscripted variables is proposed. Assuming coloring-based register allocation, this method improves the likelihood of register allocation for multiple-index subscripted references. It has the advantage that the exploration space is broadened compared with previous works by considering temporal reuse generated by multi-index subscripts. Furthermore, this approach performs a sequence of high-level compiler techniques to maximize the amount of temporal reuse. Experimental results on a number of benchmarks show that the reductions in memory access vary from 34% to 69%. This confirms that significant improvements can be achieved for data-intensive programs using our method.

Since power consumption is directly related to the number of memory access, the methods developed in this thesis will be useful in minimizing power of data-intensive software.

5.1 Future Work

Further research on this project can be conducted in the following aspects. First, the procedure described in Chapter 3 should be formalized for quantitative analysis and comparison and could be applied to codes automatically. Second, the method in Chapter 4 should prove its optimality of the unroll-and-jam algorithm and the effectiveness on memory power reduction. Third, the approaches described in this project can be extended to consider cache policy. When cache is considered, both temporal locality and spacial locality can be improved. In addition, cache size poses a restriction on inlining and unrolling. In the optimization of source code, control dependency in non-perfect loop nests are found to be obstacles during transformations. A systematic method can be developed to remove or reduce these obstacles. Finally, from the embedded system design point of view, the original work of transforming source codes to reduce array access can be utilized to optimize the memory architecture for the system.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

References

[1] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design.* Boston/Dordrecht/London: Kluwer Academic Publishers, 1998.

[2] R. Graybill and R. Melhem, *Power Aware Computing.* New York: Kluwer Academic Publishers, 2002.

[3] R. Graybill and R. Melhem, *Power Aware Computing.* New York: Kluwer Academic Publishers, 2002.

[4] R. Graybill and R. Melhem, *Power Aware Computing.* New York: Kluwer Academic Publishers, 2002.

[5] K. R. Wadleigh and I. L. Crawford, *Software Optimization for High-performance Computing.* Hewlett-Packard Company, Prentice-Hall, 2000.

[6] R. P. Wilson, R. S. French, C. S. Wilson, J. M. A. Saman P. Amarasinghe, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "Suif: An infrastructure for research on parallelizing and optimizing compilers," *In ACM SIGPLAN Notices*, vol. 29, pp. 31–37, December 1994.

[7] M. S. Lam, "A retrospective: a data locality optimizing algorithm," in *20 Years of Programming Language Design and Implementation (1979-1999): A Selection*, June 2003.

[8] F. Catthoor, K. Danckaert, S. Wuytack, and N. Dutt, "Code transformations for data transfer and storage exploration preprocessing in multimedia processors," *IEEE Journal on Design and Test of Computers*, vol. 18, no. 3, pp. 70–82, May-June 2001.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[9] R. Gonzales and M. Horowitz, “Energy dissipation in general-purpose microprocessors,” *IEEE Journal of Solid-state Circuit*, vol. SC-31, no. 9, pp. 1277–1283, September 1996.

[10] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye, “Influence of compiler optimizations on system power,” in *Proceedings of the 37th Design Automation Conference*. ACM, June 2002, pp. 304–307.

[11] The DTSE methodology, http://www.imec.be/design/dtse/.

[12] S. Carr and K. Kennedy, “Improving the ratio of memory operations to floating-point operations in loops,” *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1768–1810, November 1994.

[13] S. Carr and Y. Guan, “Unroll-and-jam using uniformly generated sets,” in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, December 1997, pp. 349–357.

[14] A. Koseki, H. Komastu, and Y. Fukazawa, “A method for estimating optimal unrolling times for nested loops,” *Information Processing Society of Japan Journal*, vol. 37, no. 06, pp. 376–382, 1997.

[15] V. Sarkar, “Optimized unrolling of nested loops,” in *Proceedings of the 14th international conference on Supercomputing*. ACM, June 1997, pp. 153–166.

[16] D. Callahan, S. Carr, and K. Kennedy, “Improving register allocation for subscripted variables,” in *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. ACM, June 1990, pp. 53–65.

[17] F. Harmsze, A. Timmer, and J. van Meerbergen, “Memory arbitration and cache management in stream-based systems,” in *Special Interest Group on Design Automation*, January 2000.

[18] V. Tiwari, S. Malik, and A. Wolfe, “Power analysis of embedded software: a first step towards software power minimization,” in *Proceedings of the IEEE Conference on Computer Aided Design*. IEEE, November 1994, pp. 384–390.

[19] N. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” in *International Symposium on Computer Architecture*, June 1990, pp. 364–373.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[20] S. Palacharla and R. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *International Symposium on Computer Architecture*, April 1994.

[21] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji, "Adapting cache line size to application behaviour," in *ACM International Conference on Supercomputing*, May 1997.

[22] S. Adve, D. Burger, R. Eigenmann, A. Rawsthorne, M. Smith, C. Gebotys, M. Kandemir, D. Lilja, A. Choudhary, J. Fang, and P.-C. Yew, "The interaction of architecture and compilation technology for high-performance processor design," in *IEEE Computer Magazine*, vol. 30, no. 12, December 1997, pp. 15–58.

[23] S.-M. Moon and K. Ebcioglu, "A study on the number of memory ports in multiple instruction issue machines," in *International Symposium on Microarchitecture*, December 1993, pp. 49–58.

[24] M. E. Wolf, *Improving parallelism and locality in nested loops.* Stanford University: Ph.D. thesis, August 1992.

[25] P. Panda, N. Dutt, and A. Nicolau, *Memory Issues In Embedded Systems-On-Chip, Optimizations and Exploration.* Kluwer Academic Publishiers, 1999.

[26] R. Niemann, *Hardware/software Co-Design for Data Flow Dominated Embedded Systems.* Kluwer Academic Publishers, 1998.

[27] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991, pp. 30–44.

[28] U. Banerjee, *Loop Transformations for Restructuring Compilers: The Foundations.* Boston/Dordrecht/London: Kluwer Academic Publishers, 1993.

[29] The 3rd Generation Partnership Project, file server area, http://www.3gpp.org/ftp/.

[30] *Technical Specification 3GPP TS 26.173: AMR Wideband Speech Codec; ANSI-C code.* The 3rd Generation Partnership Project, Mar 2002.

[31] *Technical Specification 3GPP TS 26.201: AMR Wideband Speech Codec: Feasibility study report.* The 3rd Generation Partnership Project, April 2000.

[32] The Atomium tool suite, http://www.imec.be/design/atomium/toolsuite.shtml.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

[33] *ATOMIUM 1.1.6 User's Manual, Version 1.1.* The ATOMIUM Club, The Interuniversity Microelectronics Centre, December 2000.

[34] R. W. Schiefler, "An analysis of inline substitution for a structured programming language," in *Communications of the ACM*, vol. 20, no. 9, September 1977, pp. 647–654.

[35] J. Xue and C.-H. Huang, "Reuse-driven tiling for improving data locality," *International Journal of Parallel Programming*, vol. 26, no. 6, pp. 671–696, 1998.

[36] M. J. Wolfe, "Techniques for improving the inherent parallelism in programs," *Technical Report UIUCDCS-R-78-929, University of Illinois*, 1978.

[37] F. Irigoin and R. Triolet, "Computing dependence direction vectors and dependence cones," *Technical Report E94, Centre D'Automatique et Informatique*, 1988.

[38] J. R. Allen and K. Kennedy, "Automatic loop interchange," in *Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction*, June 1984, pp. 233–246.

[39] M. J. Wolfe, "Iteration space tiling for memory hiearchies," in *In Third SIAM Conference on Parallel Processing for Scientific Computing*, 1987, pp. 357–361.

[40] F. Irigoin and R. Triolet, "Supernode partitioning," in *In the Proceedings of the ACM Symposium on Principles of Programming Languages*, 1988, pp. 319–329.

[41] J. Xue, "On tiling as a loop transformation," in *Parallel Processing Letters*, vol. 7, no. 4, 1997, pp. 409–424.

[42] Texas Instrument official website, http://www.ti.com/.

[43] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," June 1995.

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Appendix A

Optimizations to WB-AMR Speech Decoder

1. Inlining and Loop Merging

Caller: *decoder*
Callee: *Copy*, *Scale_sig*

Original Code:

```
Copy(&exc[i_subfr], exc2, L_SUBFR);
Scale_sig(exc2, L_SUBFR, -3);
```

New Code:

```
for (i = 0; i < L_SUBFR; i++)
{
    L_tmp = L_deposit_h(exc[i + i_subfr]);
    L_tmp = L_shl(L_tmp, -3);
    exc2[i] = round(L_tmp);
}
```

Caller: *decoder*
Callee: *Copy*

Original Code:

```
for (i = 0; i < L_SUBFR; i++)
{
    L_tmp = L_mult(5898, exc[i - 1 + i_subfr]);
    L_tmp = L_mac(L_tmp, 20972, exc[i + i_subfr]);
    L_tmp = L_mac(L_tmp, 5898, exc[i + 1 + i_subfr]);
    code[i] = round(L_tmp);
}
Copy(code, &exc[i_subfr], L_SUBFR);
```

New Code:

```
exc_temp1 = exc[i_subfr - 1];
exc_temp2 = exc[i_subfr];
for (i = 0; i < L_SUBFR; i++)
{
    exc_temp0 = exc_temp1;
```

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

```
    exc_temp1 = exc_temp2;
    exc_temp2 = exc[i + 1 + i_subfr];
    L_tmp = L_mult(5898, exc_temp0);
    L_tmp = L_mac(L_tmp, 20972, exc_temp1);
    L_tmp = L_mac(L_tmp, 5898, exc_temp2);
    exc[i + i_subfr] = round(L_tmp);
}
```

Caller: *decoder*
Callee: *Copy*
Original Code:

```
Copy(&exc[i_subfr], exc2, L_SUBFR);
for (i = 0; i < L_SUBFR; i++)
{
    L_tmp = L_mult(code[i], gain_code);
    L_tmp = L_shl(L_tmp, 5);
    L_tmp = L_mac(L_tmp, exc[i + i_subfr], gain_pit);
    L_tmp = L_shl(L_tmp, 1);
    exc[i + i_subfr] = round(L_tmp);
}
max = 1;
for (i = 0; i < L_SUBFR; i++)
{
    tmp = abs_s(exc[i + i_subfr]);
    if (sub(tmp, max) > 0)
    {
        max = tmp;
    }
}
```

New Code:

```
max = 1;
for (i = 0; i < L_SUBFR; i++)
{
    exc_temp0 = exc[i + i_subfr];
    exc2[i] = exc_temp0;
    L_tmp = L_mult(code[i], gain_code);
    L_tmp = L_shl(L_tmp, 5);
    L_tmp = L_mac(L_tmp, exc_temp0, gain_pit);
    L_tmp = L_shl(L_tmp, 1);
    exc_temp0 = round(L_tmp);
    exc[i + i_subfr] = exc_temp0;
    tmp = abs_s(exc_temp0);
    if (sub(tmp, max) > 0)
    {
        max = tmp;
    }
}
```

Caller: *synthesis*
Callee: *Syn_filt_32*, *Deemph_32*, *HP50_12k8*, *Dot_product12*

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Original Code:

```
Syn_filt_32(Aq, M, exc, Q_new, synth_hi + M, synth_lo + M, L_SUBFR);
…
Deemph_32(synth_hi + M, synth_lo + M, synth, PREEMPH_FAC, L_SUBFR,
&(st->mem_deemph));
…
HP50_12k8(synth, L_SUBFR, st->mem_sig_out);
…
ener = extract_h(Dot_product12(exc, exc, L_SUBFR, &exp_ener));
```

New code:

```
for (i = 0; i < L_SUBFR; i++)
{
    L_tmp = 0;
    L_tmp_hi = 0;
    for (j = 1; j <= M; j++)
    {
        Aqtemp = Aq[j];
        L_tmp = L_msu(L_tmp, sig_lo[i - j], Aqtemp);
        L_tmp_hi = L_msu(L_tmp_hi, sig_hi[i - j], Aqtemp);
    }
    L_tmp = L_shr(L_tmp, 16 - 4);       /* -4 : sig_lo[i] << 4 */
    L_tmp=L_add(L_tmp, L_tmp_hi);
    exc_temp = exc[i];
    L_tmp = L_mac(L_tmp, exc_temp, a0);
    exc_temp = round(exc_L_tmp);
    exc[i] = exc_temp;
    L_sum = L_mac(L_sum, exc_temp, exc_temp);
    L_tmp = L_shl(L_tmp, 3);            /* ai in Q12 */
    sig_hi_temp = extract_h(L_tmp);
    sig_hi[i] = sig_hi_temp;
    L_tmp = L_shr(L_tmp, 4);            /* 4 : sig_lo[i] >> 4 */
    sig_lo[i] = extract_l(L_msu(L_tmp, sig_hi_temp, 2048));
    L_tmp = L_shl(L_tmp, 7);
    L_tmp = L_mac(L_tmp, synth_temp, fac);
    L_tmp = L_shl(L_tmp, 1);
    synth_temp = round(L_tmp);
    x2 = x1;
    x1 = x0;
    x0 = synth_temp;

    /* y[i] = b[0]*x[i] + b[1]*x[i-1] + b[2]*x[i-2]  */
    /* + a[1]*y[i-1] + a[2] * y[i-2];  */
    L_tmp = 16384L;
    L_tmp = L_mac(L_tmp, y1_lo, a50[1]);
    L_tmp = L_mac(L_tmp, y2_lo, a50[2]);
    L_tmp = L_shr(L_tmp, 15);
    L_tmp = L_mac(L_tmp, y1_hi, a50[1]);
    L_tmp = L_mac(L_tmp, y2_hi, a50[2]);
    L_tmp = L_mac(L_tmp, x0, b50[0]);
    L_tmp = L_mac(L_tmp, x1, b50[1]);
    L_tmp = L_mac(L_tmp, x2, b50[2]);
    L_tmp = L_shl(L_tmp, 2);            /* coeff Q12 --> Q14 */
    y2_hi = y1_hi;
```

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

```
    y2_lo = y1_lo;
    L_Extract(L_tmp, &y1_hi, &y1_lo);
    L_tmp = L_shl(L_tmp, 1);
    synth[i] = round(L_tmp);
}
st->mem_deemph = synth_temp;

st->mem_sig_out[0] = y2_hi;
st->mem_sig_out[1] = y2_lo;
st->mem_sig_out[2] = y1_hi;
st->mem_sig_out[3] = y1_lo;
st->mem_sig_out[4] = x0;
st->mem_sig_out[5] = x1;

sft = norm_l(L_sum);
L_sum = L_shl(L_sum, sft);
exp_ener = sub(30, sft);                        /* exponent = 0..30 */
ener = extract_h(L_sum);
```

Caller: *synthesis*
Callee: *HP400_12k8*

Original Code:

```
HP400_12k8(synth, L_SUBFR, st->mem_hp400);
L_tmp = 1L;
for (i = 0; i < L_SUBFR; i++)
    L_tmp = L_mac(L_tmp, synth[i], synth[i]);
L_tmp = 1L;
for (i = 1; i < L_SUBFR; i++)
    L_tmp = L_mac(L_tmp, synth[i], synth[i - 1]);
```

New Code:

```
y2_hi = st->mem_hp400[0];
y2_lo = st->mem_hp400[1];
y1_hi = st->mem_hp400[2];
y1_lo = st->mem_hp400[3];
x0 = st->mem_hp400[4];
x1 = st->mem_hp400[5];

L_sum = 1L;
L_sum2 = 1L;
for (i = 0; i < L_SUBFR; i++)
{
    x2 = x1;
    x1 = x0;
    x0 = synth[i];
    /* y[i] = b[0]*x[i] + b[1]*x[i-1] + b140[2]*x[i-2]   */
    /* + a[1]*y[i-1] + a[2] * y[i-2];   */
    L_tmp = 16384L;
    L_tmp = L_mac(L_tmp, y1_lo, a[1]);
    L_tmp = L_mac(L_tmp, y2_lo, a[2]);
    L_tmp = L_shr(L_tmp, 15);
    L_tmp = L_mac(L_tmp, y1_hi, a[1]);
```

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

```
    L_tmp = L_mac(L_tmp, y2_hi, a[2]);
    L_tmp = L_mac(L_tmp, x0, b[0]);
    L_tmp = L_mac(L_tmp, x1, b[1]);
    L_tmp = L_mac(L_tmp, x2, b[2]);
    L_tmp = L_shl(L_tmp, 1);           /* coeff Q12 --> Q13 */

    y2_hi = y1_hi;
    y2_lo = y1_lo;
    L_Extract(L_tmp, &y1_hi, &y1_lo);
    synth_temp = round(L_tmp);
    L_sum = L_mac(L_sum, synth_temp, synth_temp);
    if(i==0)
        previous_synth_temp = synth_temp;
    else
    {
        L_sum2 = L_mac(L_sum2, synth_temp, previous_synth_temp);
        previous_synth_temp = synth_temp;
    }
}
st->mem_hp400[0] = y2_hi;
st->mem_hp400[1] = y2_lo;
st->mem_hp400[2] = y1_hi;
st->mem_hp400[3] = y1_lo;
st->mem_hp400[4] = x0;
st->mem_hp400[5] = x1;
```

2. Loop Merging and Scalar Replacement

In *decoder*, two loops: pitch enhancement and build excitation

Original Code:

```
/*pitch enhance */
L_tmp = L_deposit_h(code[0]);
L_tmp = L_msu(L_tmp, code[1], tmp);
code2[0] = round(L_tmp);
for (i = 1; i < L_SUBFR - 1; i++)
{
    L_tmp = L_deposit_h(code[i]);
    L_tmp = L_msu(L_tmp, code[i + 1], tmp);
    L_tmp = L_msu(L_tmp, code[i - 1], tmp);
    code2[i] = round(L_tmp);
}
L_tmp = L_deposit_h(code[L_SUBFR - 1]);
L_tmp = L_msu(L_tmp, code[L_SUBFR - 2], tmp);
code2[L_SUBFR - 1] = round(L_tmp);

/* build excitation */
gain_code = round(L_shl(L_gain_code, Q_new));
for (i = 0; i < L_SUBFR; i++)
{
    L_tmp = L_mult(code2[i], gain_code);
    L_tmp = L_shl(L_tmp, 5);
```

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

```
    L_tmp = L_mac(L_tmp, exc2[i], gain_pit);
    L_tmp = L_shl(L_tmp, 1);
    exc2[i] = round(L_tmp);
}
```

New Code:

```
exc_temp1 = code[0];
L_tmp = L_deposit_h(exc_temp1);
L_tmp = L_msu(L_tmp, code[1], tmp);
L_exc2 = L_mult(round(L_tmp), gain_code);
L_exc2 = L_shl(L_exc2, 5);
L_exc2 = L_mac(L_exc2, exc2[0], gain_pit);
L_exc2 = L_shl(L_exc2, 1);
exc2[0] = round(L_exc2);
exc_temp2 = code[1];
for (i = 1; i < L_SUBFR - 1; i++)
{
    exc_temp0 = exc_temp1;
    exc_temp1 = exc_temp2;
    exc_temp2 = code[i+1];
    L_tmp = L_deposit_h(exc_temp1);
    L_tmp = L_msu(L_tmp, exc_temp2, tmp);
    L_tmp = L_msu(L_tmp, exc_temp0, tmp);
    L_exc2 = L_mult(round(L_tmp), gain_code);
    L_exc2 = L_shl(L_exc2, 5);
    L_exc2 = L_mac(L_exc2, exc2[i], gain_pit);
    L_exc2 = L_shl(L_exc2, 1);
    exc2[i] = round(L_exc2);
}
L_tmp = L_deposit_h(exc_temp2);
L_tmp = L_msu(L_tmp, exc_temp1, tmp);
L_exc2 = L_mult(round(L_tmp), gain_code);
L_exc2 = L_shl(L_exc2, 5);
L_exc2 = L_mac(L_exc2, exc2[L_SUBFR - 1], gain_pit);
L_exc2 = L_shl(L_exc2, 1);
exc2[L_SUBFR - 1] = round(L_exc2);
```

In *Isf_isp*, two loops: copy and compute isp.

Original Code:

```
for (i = 0; i < m - 1; i++)
{
    isp[i] = isf[i];
}
isp[m - 1] = shl(isf[m - 1], 1);
for (i = 0; i < m; i++)
{
    ind = shr(isp[i], 7);
    offset = (Word16) (isp[i] & 0x007f);
    L_tmp = L_mult(sub(table[ind + 1], table[ind]), offset);
    isp[i] = add(table[ind], extract_l(L_shr(L_tmp, 8)));
}
```

New Code:

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

```
for (i = 0; i < m-1; i++)
{
    isf_temp = isf[i];
    ind = shr(isf_temp, 7);
    offset = (Word16) (isf_temp & 0x007f);
    L_tmp = L_mult(sub(table[ind + 1], table[ind]), offset);
    isp[i] = add(table[ind], extract_l(L_shr(L_tmp, 8)));
}
isf_temp = shl(isf[m - 1], 1);
ind = shr(isf_temp, 7);
offset = (Word16) (isf_temp & 0x007f);
L_tmp = L_mult(sub(table[ind + 1], table[ind]), offset);
isp[m-1] = add(table[ind], extract_l(L_shr(L_tmp, 8)));
```

In lagconc, two loops: find smallest history lag and biggest history lag

Original code:

```
/***********SMALLEST history lag***********/
minLag = lag_hist[0];
for (i = 1; i < L_LTPHIST; i++)
{
    if (sub(lag_hist[i], minLag) < 0)
    {
        minLag = lag_hist[i];
    }
}
/*******BIGGEST history lag*******/
maxLag = lag_hist[0];
for (i = 1; i < L_LTPHIST; i++)
{
    if (sub(lag_hist[i], maxLag) > 0)
    {
        maxLag = lag_hist[i];
    }
}
```

New code:

```
minLag = lag_hist[0];
maxLag = minLag;
for (i = 1; i < L_LTPHIST; i++)
{
    lag_hist_temp = lag_hist[i];
    if (sub(lag_hist_temp, minLag) < 0)
    {
        minLag = lag_hist_temp;
    }
    if (sub(lag_hist_temp, maxLag) > 0)
    {
        maxLag = lag_hist_temp;
    }
}
```

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

3. Loop Unrolling and Scalar Replacement

In function *Filt_6k_7k*:

Original code:

```
for (i = 0; i < lg; i++)
{
    L_tmp = 0;
    for (j = 0; j < L_FIR; j++)
        L_tmp = L_mac(L_tmp, x[i + j], fir_6k_7k[j]);
    signal[i] = round(L_tmp);
}
```

New code:

```
c1 = lg - lg % 3; c2 = L_FIR - L_FIR % 3;
for (i = 0; i < c1; i += 3)
{
    L_tmp1 = 0;
    L_tmp2 = 0;
    L_tmp3 = 0;
    for (j = 0; j < c2; j += 3)
    {
        x1 = x[i + j + 1];
        x2 = x[i + j + 2];
        x3 = x[i + j + 3];
        f0 = fir_6k_7k[j];
        f1 = fir_6k_7k[j + 1];
        f2 = fir_6k_7k[j + 2];
        L_tmp1 = L_mac(L_tmp1, x[i + j], f0);
        L_tmp1 = L_mac(L_tmp1, x1, f1);
        L_tmp1 = L_mac(L_tmp1, x2, f2);
        L_tmp2 = L_mac(L_tmp2, x1, f0);
        L_tmp2 = L_mac(L_tmp2, x2, f1);
        L_tmp2 = L_mac(L_tmp2, x3, f2);
        L_tmp3 = L_mac(L_tmp3, x2, f0);
        L_tmp3 = L_mac(L_tmp3, x3, f1);
        L_tmp3 = L_mac(L_tmp3, x[(i + 2) + (j + 2)], f2);
    }
    for(j = c2; j < L_FIR; j++)
    {
        f0 = fir_6k_7k[j];
        L_tmp1 = L_mac(L_tmp1, x[i + j], f0);
        L_tmp2 = L_mac(L_tmp2, x[(i + 1) + j], f0);
        L_tmp3 = L_mac(L_tmp3, x[(i + 2) + j], f0);
    }
    signal[i] = round(L_tmp1);
    signal[i+1] = round(L_tmp2);
    signal[i+2] = round(L_tmp3);
}
for (i = c1; i < lg; i++)
{
    L_tmp1 = 0;
```

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

```
    for (j = 0; j < L_FIR; j++)
        L_tmp1 = L_mac(L_tmp1, x[i + j], fir_6k_7k[j]);
    signal[i] = round(L_tmp1);
}
```

In function *Filt_7k*:

Original code:

```
for (i = 0; i < lg; i++)
{
    L_tmp = 0;
    for (j = 0; j < L_FIR; j++)
        L_tmp = L_mac(L_tmp, x[i + j], fir_7k[j]);
    signal[i] = round(L_tmp);
}
```

New code:

```
c1 = lg - lg % 3; c2 = L_FIR - L_FIR % 3;
for (i = 0; i < c1; i += 3)
{
    L_tmp1 = 0;
    L_tmp2 = 0;
    L_tmp3 = 0;
    for (j = 0; j < c2; j += 3)
    {
        x1 = x[i + j + 1];
        x2 = x[i + j + 2];
        x3 = x[i + j + 3];
        f0 = fir_7k[j];
        f1 = fir_7k[j + 1];
        f2 = fir_7k[j + 2];
        L_tmp1 = L_mac(L_tmp1, x[i + j], f0);
        L_tmp1 = L_mac(L_tmp1, x1, f1);
        L_tmp1 = L_mac(L_tmp1, x2, f2);
        L_tmp2 = L_mac(L_tmp2, x1, f0);
        L_tmp2 = L_mac(L_tmp2, x2, f1);
        L_tmp2 = L_mac(L_tmp2, x3, f2);
        L_tmp3 = L_mac(L_tmp3, x2, f0);
        L_tmp3 = L_mac(L_tmp3, x3, f1);
        L_tmp3 = L_mac(L_tmp3, x[(i + 2) + (j + 2)], f2);
    }
    for(j = c2; j < L_FIR; j++)
    {
        f0 = fir_7k[j];
        L_tmp1 = L_mac(L_tmp1, x[i + j], f0);
        L_tmp2 = L_mac(L_tmp2, x[(i + 1) + j], f0);
        L_tmp3 = L_mac(L_tmp3, x[(i + 2) + j], f0);
    }
    signal[i] = round(L_tmp1);
    signal[i+1] = round(L_tmp2);
    signal[i+2] = round(L_tmp3);
}
for (i = c1; i < lg; i++)
```

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

```
{
    L_tmp1 = 0;
    for (j = 0; j < L_FIR; j++)
        L_tmp1 = L_mac(L_tmp1, x[i + j], fir_7k[j]);
    signal[i] = round(L_tmp1);
}
```

In function *Pred_lt4:*

Original code:

New Code:

```
c1 = L_subfr - (L_subfr % 3);
c2 = 2 * L_INTERPOL2 - ((2 * L_INTERPOL2) % 3);
for (j = 0; j < c1; j += 3)
{
    L_sum1 = 0L;
    L_sum2 = 0L;
    L_sum3 = 0L;
    for (i = 0, k = sub(sub(UP_SAMP, 1), frac); i < c2; i+=3, k +=
3*UP_SAMP)
    {
        x1 = x[i + j + 1];
        x2 = x[i + j + 2];
        x3 = x[i + j + 3];
        inter0 = inter4_2[k];
        inter1 = inter4_2[k + UP_SAMP];
        inter2 = inter4_2[k + 2*UP_SAMP];
        L_sum1 = L_mac(L_sum1, x[i + j], inter0);
        L_sum1 = L_mac(L_sum1, x1, inter1);
        L_sum1 = L_mac(L_sum1, x2, inter2);
        L_sum2 = L_mac(L_sum2, x1, inter0);
        L_sum2 = L_mac(L_sum2, x2, inter1);
        L_sum2 = L_mac(L_sum2, x3, inter2);
        L_sum3 = L_mac(L_sum3, x2, inter0);
        L_sum3 = L_mac(L_sum3, x3, inter1);
        L_sum3 = L_mac(L_sum3, x[(i + 2) + (j + 2)], inter2);
    }
    for(i = c2, k = sub(sub(UP_SAMP, 1), frac) + c2*UP_SAMP; i < 2 *
L_INTERPOL2; i++, k+=UP_SAMP)
    {
        inter0 = inter4_2[k];
        L_sum1 = L_mac(L_sum1, x[i + j], inter0);
        L_sum2 = L_mac(L_sum2, x[i + (j + 1)], inter0);
        L_sum3 = L_mac(L_sum3, x[i + (j + 2)], inter0);
    }
    L_sum1 = L_shl(L_sum1, 1);
    L_sum2 = L_shl(L_sum2, 1);
    L_sum3 = L_shl(L_sum3, 1);

    exc[j] = round(L_sum1);
    exc[j + 1] = round(L_sum2);
    exc[j + 2] = round(L_sum3);
}
for (j = c1; j < L_subfr; j++)
{
```

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

```
    L_sum1 = 0L;
    for (i = 0, k = sub(sub(UP_SAMP, 1), frac); i < 2 * L_INTERPOL2;
i++, k += UP_SAMP)
    {
        L_sum1 = L_mac(L_sum1, x[i + j], inter4_2[k]);
    }
    L_sum1 = L_shl(L_sum1, 1);
    exc[j] = round(L_sum1);
}
```

4. Other Global Transformations

In *decoder* function:

Original code:

```
Decoder()
{
    Word16 code[L_SUBFR];
    …
    D_gain2(code,…);
    …
    Voice_factor(code,…);
    …
}

D_gain2(Word16 *code, …)
{
     …
     L_tmp= Dot_product(code);
     …
}
voice_factor(Word16 *code, …)
{
     …
     L_tmp= Dot_product(code);
     …
}
```

New code:

```
Decoder()
{
    Word16 code[L_SUBFR];
    Word32 L_tmp;
    …
    L_tmp = Dot_product(code);
    D_gain2(L_tmp, …);
    …
    Voice_factor(code, …);
    …
}
```

ATTENTION: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library

Appendix B

List of Author's Publications

[1] Shan Li, E.M-K.Lai, and Mohammed Javed Absar, "Minimizing Embedded Software Power Consumption through Reduction of Data Memory Access", *Proceedings of International Conference on Information and Communications Systems*, December 2003, Singapore.

[2] Shan Li and Mohammed Javed Absar, "Minimizing Memory Accesses By Improving Register Usage Through High-level Transformations", Accepted by *First International Workshop on Power-Aware Real-Time Computing (PARC 2004)*, September 2004, Pisa, Italy.

[3] Shan Li, E.M-K.Lai, and Mohammed Javed Absar, "A Power Efficient Software Implementation of the Wideband Adaptive Multirate (WB-AMR) Speech Decoder", Submitted to *IEEE Transactions on Consumer Electronics*, June 2004.

[4] Shan Li, E.M-K.Lai, and Mohammed Javed Absar, "Improve Register Allocation For Multiple-Index Subscripted Reference", Submitted to *ACM Transactions on Architecture and Code Optimization*, September 2004.