Enhancing the performance of Decoupled Software Pipeline through Backward Slicing

Alwan, Esraa

Award date:
2014

Awarding institution:
University of Bath

Enhancing the performance of Decoupled Software Pipeline through Backward Slicing

submitted by

Esraa Hadi Obead Alwan

for the degree of Doctor of Philosophy

of the

University of Bath

Department of Computer Science

January 2014

COPYRIGHT

Signature of Author ....

Esraa Hadi Obead Alwan

Abstract

The rapidly increasing number of cores available in multicore processors does not necessarily lead directly to a commensurate increase in performance: programs written in conventional languages, such as C, need careful restructuring, preferably automatically, before the benefits can be observed in improved run-times. Even then, much depends upon the intrinsic capacity of the original program for concurrent execution.

Using software techniques to parallelize the sequential application can raise the level of gain from multicore systems. Parallel programming is not an easy job for the user, who has to deal with many issues such as dependencies, synchronization, load balancing, and race conditions. For this reason the role of automatically parallelizing compilers and techniques for the extraction of several threads from single-threaded programs, without programmer intervention, is becoming more important and may help to deliver better utilization of modern hardware.

One parallelizing technique that has been shown to be an effective for the parallelization of applications that have irregular control flow and complex memory access patterns is Decoupled Software Pipeline (DSWP). This transformation partitions the loop body into a set of stages, ensuring that critical path dependencies are kept local to a stage. Each stage becomes a thread and data is passed between threads using inter-core communication. The success of DSWP depends on being able to extract the relatively fine-grain parallelism that is present in many applications.

Another technique which offers potential gains in parallelizing general purpose applications is slicing. Program slicing transforms large programs into several smaller ones that execute independently, each consisting of only statements relevant to the computation of certain, so-called, (program) points.

This dissertation explores the possibility of performance benefits arising from a secondary transformation of DSWP stages by slicing. To that end a new combination method called DSWP/Slice is presented. Our observation is that individual DSWP stages can be parallelized by slicing, leading to an improvement in performance of the longest duration DSWP stages. In particular, this approach can be applicable in cases where DOALL is not. In consequence better load balancing can be achieved between the DSWP stages.

Moreover, we introduce an automatic implementation of the combination method using Low Level Virtual Machine (LLVM) compiler framework. This combination is particularly effective when the whole long stage comprises a function body. More than one slice extracted from a function body can speed up its execution time and also increases the scalability of DSWP.

An evaluation of this technique on six programs with a range of dependence patterns leads to considerable performance gains on a core-i7 870 machine with 4-cores/8-threads. The results are obtained from an automatic implementation that shows the proposed method can give a factor of up to 1.8 speed up compared with the original sequential code.

Acknowledgements

In the name of Allah the Beneficent, the Merciful.

First and before everything, I would like to express my great thanks to Allah for his mercy and blessing.

Undertaking this PhD has been a truly life-changing experience for me, and it would not have been possible to do without the support and guidance that I received from many people.

I heartily thank my supervisors Prof. John Fitch (University of Bath) and Dr. Julian Padget (University of Bath) for their over-seeing, guidance, commitment, patience and suggestions, providing me with invaluable comments and insights to improve my thesis and shape this research into what is has become today.

I warmly thank Prof. McCusker (University of Bath) for his valuable advice and friendly help, and his extensive discussions throughout one year of his supervision. I will forever be thankful to my dear Prof. Mohammed Hashim Matloob (Babylon University), who was a major influence in my decision to apply to Bath University.

I will forever be thankful to my colleagues Dr. Shadi Basurra, Dr. Firas Albadran and Mr. JeeHang Lee for their valuable advice and support throughout the duration of my study at Bath. A very special thank you to Mr. Max Harris for proofreading the thesis.

I am, as ever, especially indebted to my brothers and sisters, for their love, prayers and remarkable support throughout my whole life. My biggest regret in my life is that my beloved parents did not live to see me complete my PhD. No amount of words could fully convey my

gratitude to them.

I'm also grateful to my dear friends Asieh, Fatemeh and Ta (Pawitra) for their encouragement and understanding during my life in academia.

Finally, I gratefully acknowledge the funding received towards my PhD from the Ministry of Higher Education and Scientific Research (MoHESR) in Iraq.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many improvements in both commodity hardware systems and compiler optimization techniques have occurred in the last decade, thus raising application performance. A range of microarchitectural techniques such as the superscalar issue, out-of-order execution, on-chip caching, and deep pipelines supported by sophisticated branch predictors, have been used to enhance the performance of single thread applications in a highly effective way (Spracklen and Abraham, 2005). By continuously doubling the number of transistors in each processor generation, as shown in figure 1-1, the increment in the performance of single thread programs has become clear especially due to recent increases in processor clock speed.

According to figure 1-2, which illustrates the evaluation of a range of processors using the SPEC benchmark suite (SPEC, 2013), there was a continuous improvement in the performance of single-threaded application until about 2004, but after this the increase stopped. Many factors caused this, however, two of them have had a major impact. The first one relates to processor clock speed, where it is impossible for the processor designer to increase the processor clock speed without overcoming the barriers of power and thermal design. The second factor is that although there was stability in increasing the number of transistors, the creation or development of a new microarchitcture was becoming impossible without addressing power and heat budgets (Bridges, 2008).

A new strategy has been introduced by processor designers that has involved steadily increasing in the number of transistors, through which many cores are placed in one chip. These multiple cores on a single chip combine to replicate the performance of a single faster processor. The individual cores on a multi-core processor do not necessarily run as fast as the highest performing single-core processors, but they do improve overall performance by handling more tasks in parallel (Venu, 2011). Moreover, if one core in the multi core system stops working the rest will continue without reduction in the overall performance in a full working system. This property is termed fault-tolerance. Also, static power consumption can be reduced because if one of the cores is not needed it can be switched off (Han, 2010).

Recently, many general purpose processors have been marketed with four, twelve or sixteen cores. For example, Intel introduced Core-i5 and Core-i7 processors having four cores placed in

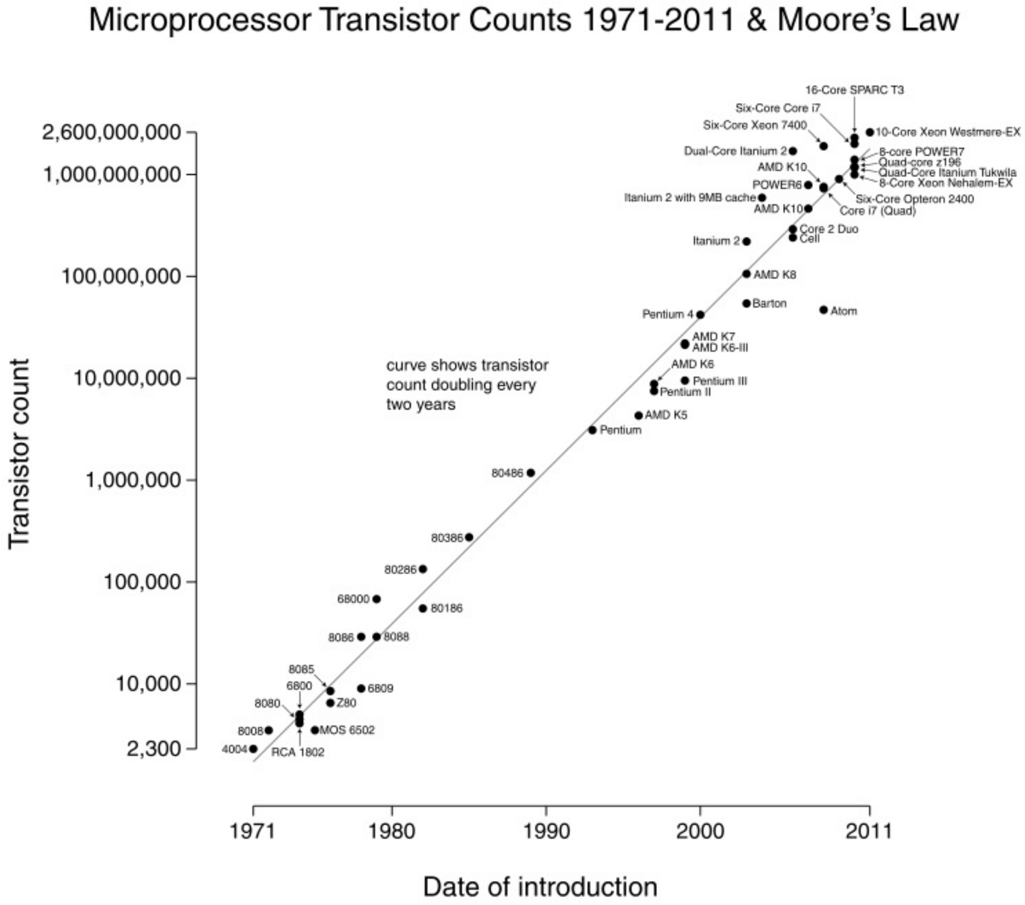


Figure 1-1: Number of transistors integrated per die for Intel x86 processors. Adapted from (Wikipedia, 2011) , updating Intel (Intel-Corporation, 2002)

the same die, respectively, while the Opteron^{TM} processor from AMD has sixteen cores (AMD, 2013). In addition, the POWER processor with twelve cores per socket has been released by IBM (IBM, 2013) and finally, UltraSPARC T1 and T2 processors[2] from SUN have eight cores each (Vachharajani, 2008), with more being expected in the future.

While incremental increases in the number of cores improve the performance of multi threaded applications, no gains have been made from this strategy for single thread application. Various techniques have been presented to discover the parallelism in such applications. Regarding this, in the late 1960s much research began in the field of parallel languages, compilers and distributed techniques. However, while they made noticeable progress on instruction level parallelism where multiple low-level instructions are executed at the same time, parallelizing general program application still requires more work.

There are a number of reasons making parallel programming difficult (Bliss, 2007):

1- The process of converting a sequential program to parallel is complex as it needs: par-

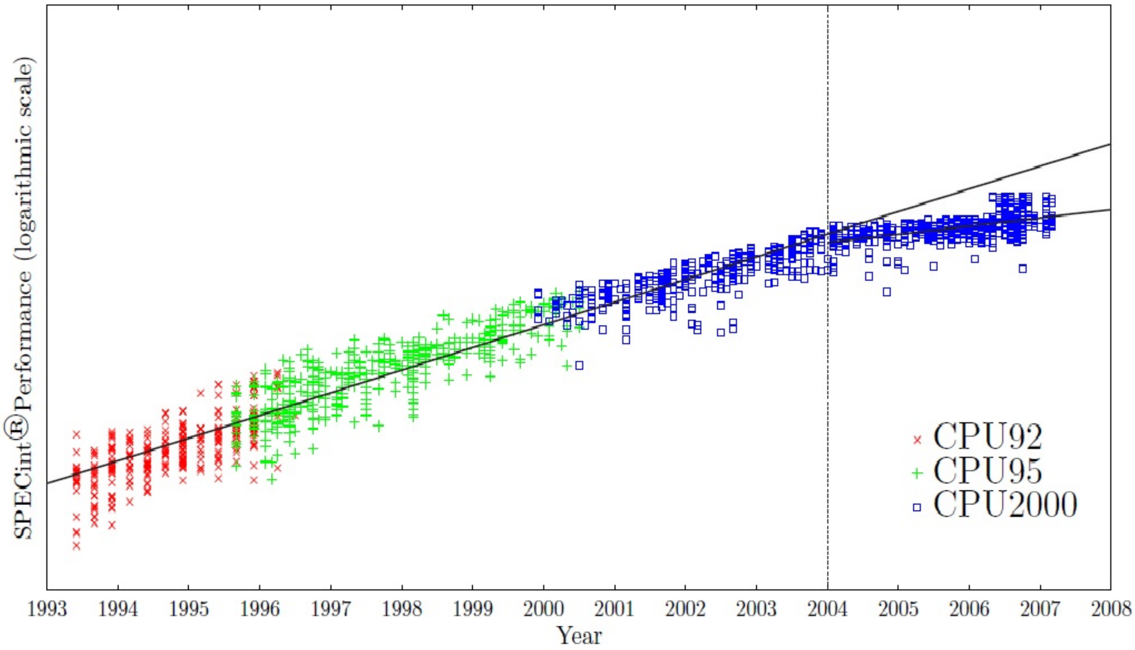


Figure 1-2: Normalized SPEC scores for all reported confgurations beween 1993 and 2007. Adapted from (SPEC, 2013)

titioning and mapping the sequential program with its complex data structure to the parallel parts.

2- Synchronization techniques are required to coordinate the processor work.

3- Writing an efficient parallel program is the hardest task and significant attention needs to be paid to balancing program communication and computation.

Two predominant methods have been put forward to enhance the performance of the single threaded program by translating it into a multi threaded application, first, by having the programmer accomplish this and, second by assigning this responsibility to the compiler.

Rewriting single-threaded applications as multi-threaded applications by the programmer is one solution that has been used to speed up single-threaded applications. Unfortunately, many difficulties have emerged from the rewriting process, not least because it is an error-prone task. Consequently, although this rewriting process can give more parallelism and hence better results, if it is undertaken by an expert programmer, it is still a costly process and time consuming (Tournavitis et al., 2009). Another challenge that the developer has to address is that of testing and debugging the program. Finally, additional concerns regarding synchronization issues, such as deadlock, race conditions, etc. have to be overcome.

Many languages have been designed to help the programmer to parallelize single thread application by providing them with the constructs and annotations. However, the job of identifying the right place for these constructs to express the parallelism is still left to the programmer. Examples of these languages and language annotations are High Performance Fortran (Loveman, 1993), MPI (MPI, 2013), OpenMP (OpenMP, 2013), Cilk (Frigo et al., 1998), StreamIt (Thies et al., 2002) and Atomos (Carlstrom et al., 2006). It is easy for the programmer to use these constructs with regular and structured parallelism, but for general purpose programs that do not have this type of parallelism the effectiveness of applying these structures is very limited (Raman, 2009). The gains from manual rewriting can be spectacular [table2,(Bischof et al., 2012)] but the cost (6 or more skilled person months) are great. Hence we ask whether we can achieve (some of) these gain automatically.

The alternative approach to a multi-threaded application is using an automatic parallelizing compiler. Through this the difficulties and costly efforts of the previous solution are alleviated by giving the job of translating the single threaded application to a multi-threaded one to the compiler without programmer intervention. The compiler will look at all the sequential code in the single threaded program and try to find the best program part that is suitable to execute in parallel. Moreover, this compiler is not only concerned with developing a new parallel program, but it can also help a legacy program to take advantage of the multi-core system (Kim et al., 2000; Vachharajani, 2008).

Additionally, it allows the compiler to adjust automatically the amount and type of parallelism extracted based on the underlying architecture, just as instruction-level parallelism(ILP) optimizations relieve programmers of the burden of targeting their applications to complex single-threaded architectures. The process of converting the sequential program to a parallel one can be divided into three phases. First, it needs some compiler techniques to analyse the relationship between program statements, to find data and control dependency. Second, it has to employ the information collected from the first phase to discover the program sections that have potential parallelism and finally the parallel code needs to be produced (Kim et al., 2000). The essential problem is getting the correct transformation to respect the dependencies in the program. In general, the greater the accuracy of the dependency information that is discovered, the more effective the parallel coding that can be produced. Thus, a great deal of effort has been spent on developing dependence representations that can provide accurate results. One of the most efficient and powerful program representations that can help in the program analysis and capturing dependent information is the Program Dependency Graph (PDG) (Zhao and Rinard, 2003), which has the ability to represent both data and control dependencies. Then the potential parallelism can be revealed through the dependency relationship between program instructions (Ferrante et al., 1987).

The richest part in the program as a source of parallelism is the loop body, because it is executed repeatedly. That is, many loop iterations can be separated across multiple cores and executed in parallel. Different cores can execute different loop iterations simultaneously.

This will be a straightforward process if there are no dependencies between the loop iterations. Many automatic techniques with different degrees of applicability and efficiency regarding the extraction of parallelism from the loop body have been proposed. In particular, significant progress was achieved in parallelizing the loop body from the 1980s to 1990s by using the DOALL and DOACROSS techniques (Tournavitis et al., 2009).

DOALL is one of the techniques that extracts parallelism by executing loop iterations in parallel and has achieved success in scientific and numerical applications. However, it is inapplicable when there is an inter-iteration dependency between loop iterations. The other technique is DOACROSS, which works in a similar way to DOALL, but has the ability to handle any inter-iteration dependency that occurs, when the current loop iteration is dependent on the result from the previous one. Both DOALL and DOACROSS along with several other techniques that have the ability to discover the parallelism and convert a single threaded application to a multi threaded one have been integrated in some research compilers such as the Fortran-D compiler (Hiranandani et al., 1993), SUIF (Hall et al., 2005), Polaris (Blume et al., 1994), etc (Raman, 2009).

However, most of these previous techniques that operate on an array with regular access have limitation in applicability, because many general purpose applications have irregular control flow and complex memory access. Thus, parallelizing this type of application is still an incomplete research area. One of the techniques that has shown the power to parallelize irregular application patterns is the Decoupled Software Pipeline (DSWP), which addresses the problem in a different way. That is, instead of separating loop iterations on different cores by giving each to one thread, DSWP partitions the loop body into pipeline stages, thus ensuring that critical path dependences are kept stage-local. Each of these stages will be given to different threads and the dependency between threads is broadcast using inter core communication (Bridges, 2008; Li et al., 2011).

Another technique which has shown potential in parallelizing general purpose applications is slicing, more precisely, the “Backward slice” which is appropriate for multiprocessors, because it has the ability to decompose such applications into independent slices that are executed in parallel.

Once the statement that represents a seed for backward slicing has been identified, the statements that have dependency relations with this seed statement are extracted and can execute in parallel (Wang et al., 2009). In the next section more details are provided on the problem of extracting parallelism and a solution is put forward.

1.1 Problem Description

As explained above, DSWP represents the most successful method, for parallelizing general purpose applications that have irregular control flow and complex memory access patterns using pointers. Unlike the other parallelizing techniques, namely DOALL and DOACROSS, that work by separating the loop iterations on available cores, it divides the loop body into several stages and gives each stage to the thread where each stage can be independently executed forming a pipeline.

The way that the loop body is divided in this method guarantees that the critical path dependency will be thread local. Moreover, the dependency between DSWP stages should be respected in order to get correct execution. Both long communication latencies between threads and the variable latency within each thread can be tolerated by keeping the critical path dependency thread local and decoupling the execution of the threads, respectively (Bridges, 2008; Li et al., 2011). Although DSWP has the ability to parallelize a wide range of applications, its performance is degraded when there are unbalanced stages. Under these circumstances, because each stage of the pipeline is given to a single thread and the amount of work given to these threads is unequal, then some DSWP threads will take more time to finish their work than others. Also, this time can grow proportionally with the amount of work leading to the negation of any benefit from using DSWP.

The solution proposed in this thesis involves focusing on exploring whether and how DSWP can be enhanced to enable more transformation in combination with other parallelizing techniques. More precisely, the main feature of the proposed method is applying backward slicing to the longest stage emerging from the DSWP transformation. The slicing technique is a decomposition technique that involves the extracting a group of statements for a given variable in a program. This group of statements has an influence or likely influence on computing a variable at a certain point in the program (Binkley and Gallagher, 1996), where these extracted slices offer scope for execution in parallel.

Moreover, it is shown that DSWP/Slice combination provides a new alternative in cases where the previous methods such as DSWP+ (Huang et al., 2010) are inapplicable and also that the approach presented here can increase the scalability of these methods. This is particularly effective when the whole stage represents a function body. That is, instead of giving the whole of stage that represents the function body to one thread, it can be distributed across n threads, depending on the number of slices extracted, within this case, one thread running the first stage and n more running $S1, S2, \ldots, Sn$ (the extracted slices from the second stage).

As a consequence the contributions of this thesis are as follows

1- It gives a new alternative method to improve the performance of DSWP in cases where the previous DSWP+ (Huang et al., 2010) is inapplicable.

2- It introduces an initial implementation of the automatic compilation of the DSWP/Slice method.

3- It evaluates this method with several programs, both artificial and real.

4- It evaluates the effect of using the lock-free buffer.

5- It implements the whole process in the framework of the LLVM compiler.

1.2 Outline

The thesis is organised as follows. Chapter 2 sets out the background and includes a description of some of the concepts and essential information regarding graph theory as well as the terminology that will be employed throughout the work. It also includes a brief explanation of static single assignment, program dependency graph representation along with providing an introduction to the LLVM Low Level Virtual Machine) architecture and its intermediate representation. It also describes some of the LLVM optimization passes and classes that have been used in this thesis. Chapter 3 presents an overview of several parallelizing techniques and is divided into two parts. The first starts by considering the slicing technique and its various forms, then it describes two methods that have been used to extract slices which are the control flow graph and program dependency graph. Finally, there is a review of the research literature that has used the slicing technique in parallelism. The second part of chapter 3 introduces loop parallelizing techniques, specifically: DOALL, DOACROSS, and DSWP. Also, it reviews some analytical studies regarding these methods that have revealed the difficulties that have prevented their being used in general purpose programs as well as examining the difference between DSWP and these other techniques. Finally, it considers some of the research literature that has applied DSWP techniques. In chapter 4, the implementation of the proposed method is presented. The work that has been carried out by (Fuyao Zhao, 2011) to implement the DSWP in LLVM (see section 2.4) has been adapted to make it suitable for the DSWP/Slice technique. Chapter 5 contains an evaluation of the automatic implementation of the proposed method presented in chapter 4. Several programs are used to evaluate and discuss the results obtained from applying this method, with some being artificial and the others real. This chapter also shows how this method enables the automatic extraction of parallelism. The evaluation is divided into three parts, with the first discussing the effect of the Lock-Free buffer technique on the performance of DSWP. The second shows how the DSWP/Slice technique can improve the performance of long stage DSWP with different program patterns and finally, there is consideration of the effect of buffer size and the slice length on the performance of the DSWP/Slice technique. Finally, chapter 6 contains a summary of the research, its contribution to the field of computer science and suggestions for future work as well as the final conclusion.

Chapter 2

Background

This chapter provides background information and terminology regarding graph theory concepts. It also includes a discussion of the fundamental principles in the field of program representation. In particular, it explains Static Single Assignment (SSA) and the Program Dependency Graph (PDG) representation that are both going to be employed throughout the thesis. Moreover, there is a brief introduction to the LLVM architecture and some of its classes that have been used to implement the research work.

2.1 Definition and Terminology

2.1.1 BasicBlock

A *Basic block* is linear sequence of instructions that has a single entry and exit point. The semantic of basic blocks are that all the instructions in this block are executed sequentially. Every basic block starts with a label and ends with a branch instruction and there is no other branch or label. For example, consider the code in figure 2-1 the instructions at lines 9,10 and 11 form a basic block, whilst that at line 7 starts another basic block that ends with a conditional branch that has two operands which represent the two outputs of the `while` statement (Muchnick, 1997; Appel and Palsberg, 1998).

2.1.2 Directed Graph

- **Graph G:** A graph is a finite set of ordered pairs G(V,E), where V represents the set of the vertices and E set of the edges,
- **Vertex V:** A vertex, V, represents a node in a graph. In program representations such as a Control Flow Graph (CFG) or Program Dependency Graph (PDG), every statement, predicate (e.g. if-then, while, etc.) or basic block is represented by a vertex.
- **Edge E:** In graphical representations, edges represent relationships between two vertices written as $v_i \longrightarrow v_j$

- **Path P:** A path is a sequence of vertices $v_i, v_{i+1}, \ldots, v_j$, where there is a directed edge between each consecutive pair of vertices $\langle v_k, v_{k+1} \rangle$.

- **Predecessors/Successors:** If there is a directed edge between two vertices $v_i \longrightarrow v_j$, v_i is a predecessor of vertex v_j and v_j a successor of vertex v_i (Muchnick, 1997).

2.1.3 Control Flow Graph (CFG)

A Control Flow Graph (CFG) is a common intermediate representation for a program. Each edge in this graph represents a flow of control for the execution of the program and each node represents a basic block. Two special vertices or nodes are contained in this graph, START and STOP. Through the former node the control can enter the first basic block in the control graph and is also the predecessor of the first basic block. Through the latter the control can leave the control graph and it represents the successor of the last basic block in the control flow graph. The edge $v_i \longrightarrow v_j$ represents the control flow edge signifying that the program is going to execute vertex v_j after vertex v_i. Each node has no more than two outgoing edges and if they are both present this means a test is carried out to decide which control edge is taken. The outgoing edges can be labelled "True" or "False" depending on the test outcome. Figure 2-2 represents the CFG for the code in figure 2-1. All the edges in this graph represent the control flow for this program, while the triangular boxes represent basic blocks or nodes.

Several of the additional relationships between CFG vertices need to be defined (Muchnick, 1997) :

- **Dominance:**

  A vertex v_i dominates a vertex v_j in the CFG if all control flow paths from BEGIN to v_j pass through v_i.

- **Post Dominance:**

  A vertex v_j postdominates a vertex v_i if all control flow paths from v_i to END pass through v_j.

- **Strict (Post) Dominance:** A vertex v_i strictly (post)dominates a vertex v_j if v_i (post)dominates v_j and $v_i \neq v_j$.

- **Immediate (Post) Dominance:**

  A vertex v_i is the immediate (post)dominator of a vertex v_j if v_i strict(post)dominates v_j and no other vertex v_c exists such that v_i strict(post)dominates v_c and v_c (post)dominates v_j

- **Dominance Frontier:**

  A dominance frontier for vertex v_i contains all vertices v_j, such that v_i dominates an immediate predecessor of v_j , but v_i does not strictly dominate v_j.

```
int main()
{
  int n,
  reverse = 0;
  scanf("%d\n",&n);

  while (n != 0)
   {
    reverse = reverse * 10;
    reverse = reverse + n/ 10;
    n = n/10;
   }
   printf( "%d\n",reverse);
   return 0;
}
```

Figure 2-1: source code

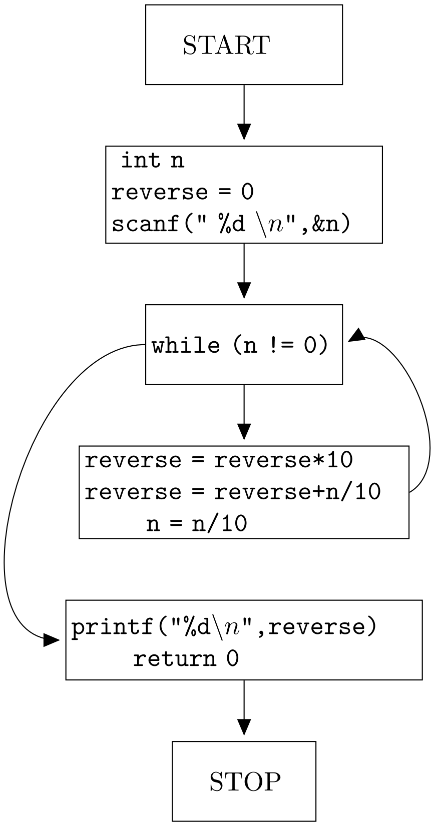


Figure 2-2: CFG

- **Def-Use chain**

  A Definition-Use (DU chain) is a data structure for a variable, X, where U represents all its uses and D represents all its definitions, that is the definition of this variable X can reach its use without an intervening definition.

2.1.4 Strongly Connected Component (SCC)

A strongly connected component (SCC) is a sub-graph of the original directed graph in which for each node n there is a path to all other nodes. For example, see the graph in figure 2-3. In this graph there are two SCCs. The first comprises the nodes 1, 2 and 3 where there is a path from node 1 to node 2, from node 2 to node 3 and finally from node 3 to node 1 which creates a cycle. The second comprises the nodes 4 and 5.

2.1.5 Directed Acyclic Graph (DAG)

Any directed acyclic is contained in an SCC, so if every SCC represent one node, the directed graph will be acyclic (Direct Acyclic Graph:DAG). Figure 2-4 represents the DAG that is

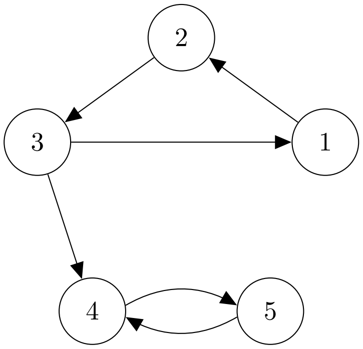


Figure 2-3: Strong Connected Component

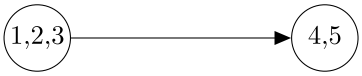


Figure 2-4: Direct Acyclic Graph for Strong Connected Component

obtained from the figure 2-3 after merging all the nodes that configure SCC in one node.

2.2 Program Dependency Graph (PDG)

A program dependency graph (PDG) is an intermediate program representation, which is a pair of graphs: the data and the control dependency graphs. Unlike the flow graph, a PDG has the important property that it can represent all the data and control dependencies in a program. A PDG represents a program as a graph, each operator and operand being corresponding nodes in the graph and the dependency between these are corresponding edges. Two types of dependency edges are found in a PDG, the first being data dependency edges, which occur when a variable defined in one statement is used in another and the output will be incorrect if the order of these statements is reversed. For example in:

$$A = B * C \tag{S1}$$

$$D = A * E + 1 \tag{S2}$$

the output of the second statement S2 is dependent on the statement S1 and so if the statement order is changed the value of D will be incorrect. The second type is control dependency edges, which occur when the execution of the second statement is controlled by the first. For example in the code below:

$$\begin{aligned} & if(A)then && \text{(S1)} \\ & B = C * D && \text{(S2)} \\ & endif && \end{aligned}$$

The execution of statement S2 is controlled by the execution of the predicate A in statement S1. In the next subsection the main two components of a PDG, which are the data dependency graph and control dependency graph, are defined (Ferrante et al., 1987).

2.2.1 Data Dependency Graph

In this graph, each node represents a statement in the program and the directed edges represent the dependency. The data dependency definition comes from the idea that there is one statement assigning a value to the variable and another statement uses this value later on. More formally, a data dependency exists between two statements S1 and S2 with respect to the variable `x`, if this variable x satisfies the following three conditions.

1- Variable `x` is defined at statement S1.

2- Variable `x` is used at S2 and,

3- There is a path in the program from S1 to S2 that does not have another definition for the `x` variable.

Three types of data dependency can be found in a program:

- **Flow Dependence** Dependency from a write to a read of either a register or a memory location, termed true dependency, which cannot be broken by renaming. For example look at the two statements below where S2 depends on the statement S1.

$$x = 10 \tag{S1}$$

$$y = x + c \tag{S2}$$

- **Anti Dependence** Dependency from a read to a write of either a register or a memory location is a false dependence, called an anti dependence, because it can be broken by making the write use a different name than the read. In the example below the variable y exists in S1 and S2 and this dependency can be broken by renaming this variable.

$$x = y + c \qquad (S1) \qquad\qquad x1 = y1 + c1 \qquad (S1)$$

$$y = 10 \qquad (S2) \qquad\qquad y2 = 10 \qquad (S2)$$

- **Output Dependence** Dependency from a write to a write of either a register or a memory location is also a false dependence, but called output dependence, which can be broken by making one of the writes use a different name. In the following example, renaming the variable x can break the dependency (Ferrante et al., 1987).

$$x = 10 \qquad (S1) \qquad\qquad x1 = 10 \qquad (S1)$$

$$x = 20 \qquad (S2) \qquad\qquad x2 = 20 \qquad (S2)$$

Note that we may need to introduce a Φ-function (see 2.3) Additional classifications can be added to the DDG which are:

- **loop-carried dependency** A loop-carried dependency occurs because of the loop iterations, with the latter iteration depending on the former one. Consider the piece of code below. In each loop iteration except the first, the execution of `S2` depends on the value

```
1.   int i=0;
2.   int sum = 0;
3.   while (i<8){
4.   sum += i;
5.   i++;
     }
6.   print sum ;
```

Figure 2-5: Code Example

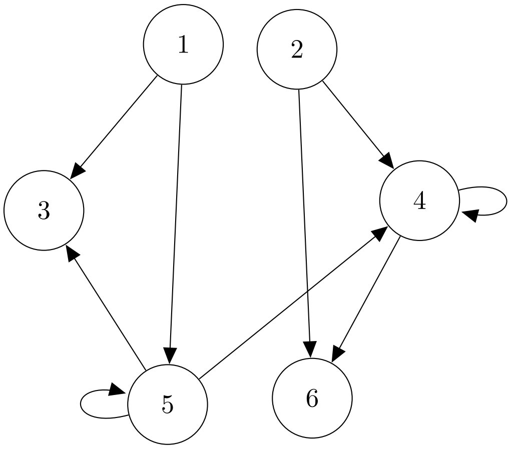


Figure 2-6: Data Dependency Graph

of A[i] that has been computed from the previous iteration. The same applies for the statement S1: the new value for A[i+1] depends on the value of F[i] that has been computed in the previous iteration. For this, loop iterations can not execute in parallel, this because both of these statements have loop carried dependency (Kennedy and Allen, 2002; Tip, 1995).

```
for (i=1 ;i<N; i++)
    {
     A[i+1]=F[i];     S1
     F[i+1]=A[i];     S2
    }
```

- **loop-independent** A loop-independent term means the execution of the loop iterations does not depend on their order. The loop iterations in the code below have the ability to execute in parallel, because the execution of the statement S2 depends on the value of A[i] that has been calculated at statement S1 at the same iteration (Tip, 1995).

```
for (i=1 ;i<N; i++)
    {
     A[i] = B[i] + C     S1
     D[i] = A[i] + E     S2
    }
```

Figure 2-6 shows the data dependency graph for the piece of code in figure 2-5.

2.2.2 Control Dependency Graph(CDG)

In this subsection the Control Dependency Graph (CDG) is defined in terms of the Control Flow Graph (CFG) and dominators. That is, the Control Flow Graph of a program P is a directed graph(N,E,START,STOP), where N and E represent the set of nodes and edges respectively. Each node n in this graph G represents either a statement or a control predicate and each edge $(n, m)\epsilon E$ indicates the possible flow of control from node m to n. In addition, two unique nodes, START and STOP, represent the entry and exit nodes in the CFG graph and each node in the CFG has at most two successors. The node v is post-dominated by a node w, if all paths from v to the STOP node contain w. Moreover, in the control flow graph G, if x, y and z are nodes, then y is control dependent on x if and only if (Ferrante et al., 1987; Sreedhar et al., 1999):

- There is a direct path p from x to y and this path has another node called z (which is different than x and y) where node y post-dominates by z.
- Node y does not post-dominate x.

Two exit nodes for node x must be available if y is control dependent on x. One of these exits always leads to y, however the other exit may not lead to y.

Construction of the CDG from the CFG of graph G (Appel and Palsberg, 1998) is achieved as follows:

- Construct the reverse control-flow graph $G^{'}$ where every $y \longrightarrow x$ edge will be transformed to a $x \longrightarrow y$ one and the START point in the G will be the STOP point in the $G^{'}$.
- Computing the dominator tree for $G^{'}$, where computing the post-dominator in the control flow graph is equivalent to computing the dominator tree for the reverse control flow graph.
- For all nodes of $G^{'}$ dominance frontiers $DF_{G'}$ should be calculated.
- The edge $x \longrightarrow y$ is one of the CDG edges if the node $x \in DF_{G'}[y]$. Figure 2-7 shows the CDG for the program in figure 2-5.

2.3 Static Single Assignment

Static single assignment (SSA) is a common and efficient intermediate representation that simplifies the implementation of program optimizations. It is used as the base stage for a variety of optimization algorithms for producing optimization compilers (Sreedhar et al., 1999; Srinivasan and Grunwald, 1991). The SSA algorithm traverses the control flow graph (CFG) and transforms the input code to SSA form. A CFG is a directed graph, with every sequence of consecutive statements representing a basic block in a program, which corresponds to a node in the graph. The most powerful feature of SSA is that each variable in a program has been assigned once. That is, a single value is given to each variable and every use of that variable has exactly one reaching definition (Srinivasan and Grunwald, 1991). To produce an SSA form for a set of statements, the left-hand side variable for the first statement is given a unique

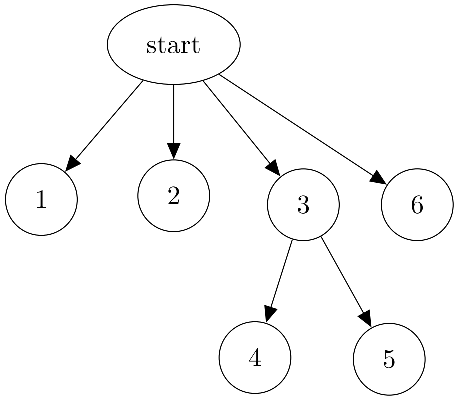


Figure 2-7: Control Dependency Graph for the program in figure 2-5

(a)	(b)
$A = 1$	$A_1 = 1$
$B = A + 2$	$B_1 = A_1 + 2$
$A = B$	$A_2 = B_1$
(a)	(b)

Figure 2-8: (a) Original code fragment. (b)SSA form

name one time during its life and all uses of this variable by other statements are consequently renamed (Brandis and Mössenböck, 1994). For example, figure 2-8a and figure 2-8b represent the set of statements before and after converting this set to SSA form. As can be seen, there are two definitions for the variable `A` in statements 1 and 3. The first variable is assigned the name `A1` and all uses before the second definition, which is above statement 3, are assigned this name. Subsequently, the second definition of a variable is assigned the name `A2` and likewise, all its uses employ this name. That is, both the names `A1` and `A2` are value names and all uses for these should be given the same name (Srinivasan and Grunwald, 1991). SSA introduces another function called the Φ-function for programs that have branch or join nodes. This is because such programs do not know in advance which branch will be taken as they only deal with static data flow. The Φ-function can merge distinct values and produce a new value name, which is used to indicate that the exact value of a variable is no longer known (Srinivasan and Grunwald, 1991; Brandis and Mössenböck, 1994).

In the example shown in figure 2-9, without knowing the value of `P` at compilation time, it is impossible to determine which path will be taken, in other words, which value for `X` reaches the end of the conditional branch. Therefore, the Φ-function is inserted in the merge node, node (5), which introduces the new value, named X_4, which highlights the value of X_2 or X_3 that will be taken. The SSA form has two main properties. The first one is that the use of a specific variable has only one definition, and the Φ-function is inserted in the join point if there are

```
I=1
X=5
IF  P < 100 then
I=I+1
X=X+(I+2)
Else
    I=I+3
    X=X+(4*I)
K= X
```

(a)

```
$I_1$ = 1
$X_1$ = 5
IF  P < 100  then
    $I_2$ =  $I_1$+1
    $X_2$ = $X_1$+( $I_2$+2 )
Else
    $I_3$ = $I_1$+3
    $X_3$ = $X_1$+(4*$I_3$)
$I_4$ =  $\Phi$($I_2$,$I_3$)
$X_4$ = $\Phi$($X_2$,$X_3$)
$K_1$ = $X_4$
```

(b)

Figure 2-9: (a) Original code fragment. (b) Final SSA form

multiple definitions to the same variable coming from different branches. The second, is that the definitions dominate all uses, whereby any statement that uses the X variable must come from the unique definition of that variable. For these reasons, many efficient optimization algorithms use the SSA form as a platform, such as: constant propagation, dead-code elimination, code motion, and so on (Brandis and Mössenböck, 1994).

2.4 LLVM

The Low Level Virtual Machine (LLVM) is a compiler framework that uses a combination of a low level virtual instruction set with a high level information type. Development of this compiler began in 2000, whereas the popular open source compiler, GCC, was established in 1985. The LLVM was developed as research infrastructure by Vikram Adve and Chris Lattner at the University of Illinois and is written in C++. It involves a compilation strategy organized as multi-stage optimization, performed at compile-time, link-time, runtime, and offline. An important part of the LLVM design is the intermediate representation (IR), which is the output of the compiler front-end and this fornt-end is responsible for parsing the source code and converting it into the intermediate representation code. This intermediate representation allows for many traditional optimizations to be applied to LLVM code. Moreover, the LLVM IR is a low level RISC like virtual instruction set that captures the key operations of ordinary processors, like: add, subtract, compare and branch. That is, it presents an infinite set of typed virtual registers, which can carry integer, floating and pointer values (Neustifter, 2010; Lattner and Adve, 2002).

One of the benefits of the LLVM compiler is its ability to provide continuous optimization throughout all the compilation stages (Ye, 2011). In the following subsections, first, there is a brief introduction to the LLVM system architecture and its various stages (front end, LLVM optimization and back end). In addition, further detail regarding IR and the uses of the static single assignment form in the LLVM are provided. Finally, some of the LLVM passes and

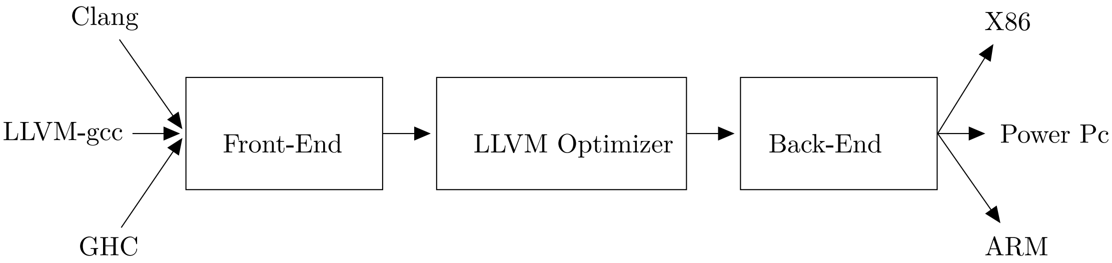


Figure 2-10: LLVM's Implementation of the Three-Phase Design (Brown and Wilson, 2008)

classes used in the programming of DSWP and slicing, developed in this thesis, are described.

2.5 LLVM System Architecture

LLVM was designed to support multi-phase compilation which is the most common way to design the traditional static compiler. The front end stage was constructed so as be able to accept multiple languages. Moreover, this stage parses, validates and diagnoses errors in the input code and then translates it into the LLVM virtual instruction set. During the second stage, optimization, the LLVM IR passes through many analysis and optimization steps so as to optimize the input code ready for the back end, also called the code generator, which is responsible for producing the native machine code. Figure 2-10 illustrates these three stages (Brown and Wilson, 2008).

2.5.1 Front-End

One of the advantages of the LLVM compiler is its variety of front ends, which has the capability of supporting various programming languages, such as: Ada, C, C++, Objective C, Fortran, Ruby and Python. The LLVM front-end translates the source language program into the LLVM virtual instruction set. The front-end analyses the input program, performs architecture-independent optimization and generates the intermediate representation (IR). Examples of LLVM front-end compilers are LLVM-gcc and Clang.

The LLVM front-end is not responsible for producing directly an SSA form of the program. Rather, this is achieved by the LLVM stack promotion (which promotes the stack allocation of scalar values to reside in SSA registers) and scalar expansion (which expands local structures to scalars) passes, which generate efficient SSA for the variables that can be allocated on the stack (which of itself is not in SSA form). After expanding local structures to scalars with the scalar expansion pass, their fields can then be mapped to the SSA registers (Lattner and Adve, 2004; Neustifter, 2010).

Intermediate Representation (IR)

As explained above, the LLVM IR captures the basic operations of ordinary processors, like add, subtract etc. and this instruction set is hardware independent. It uses the Static Single Assignment (SSA) form and high level language independent type information. The LLVM instruction set has 31 op-codes so as to avoid multiple ones for the same operation. In addition, most operations in the op-codes are overloaded, which means that the same instruction can be used with integer and floating point operands. This greatly reduces the number of distinct op-codes as it is not necessary use different ones for unsigned integers and single or double precision floating point values. The semantics of the operations and the type of result depends entirely on the types of operands, regarding which a three-address code is used to represent the operations, consisting of one or two operands and producing one result (Lattner, 2002; Lattner and Adve, 2002). All function representations that result from the LLVM intermediate representation have an explicit CFG which consists of a set of basic blocks. Each contains a sequence of instructions and terminate with a special instruction, such as branch, return, which has the ability to determine its successor. In addition, the first basic block in each function is called the entry block, while the last is the return block (Lattner and Adve, 2004). Figure 2-11 illustrates simple C source code and the LLVM IR that corresponds to this is given in figure 2-12

Language-Independent Type System

The Language-Independent Type System is one of the important properties in the LLVM design. There is a type associated with each SSA register and the memory object as well as all the operations have to follow these strict type rules. Through the connection between the information type and instruction op-code, the right semantics for each instruction will be defined, for example, the difference between the floating point add and integer add.

LLVM contains a source-language-independent primitive types and four derivative types. The former has predefined size, such as void, bool, signed/unsigned integer ranging from 8 to 64 bits and with single, double-precision floating-point types, while the derivative types include pointers, array, struct and functions. Consequently, high level language types can be constructed based on the primitive and derivative LLVM types, for example, the C++ classes. Figure 2-13 shows some examples illustrating the types in LLVM.

The instruction operands in LLVM have restrictions in that all these instructions are strictly typed, which helps to simplify the transformation and maintain type correctness. For clarification, both operands of the add operation have to have same type, for example, both of them could be int or float and the result would be either int or float, respectively. The same thing applies for load and store instructions. The load instruction needs a pointer operand to load from while the store instruction needs value to store and pointer to location where this value store. If the value has specific type the stored location has to be a pointer to that type (Lattner and Adve, 2002).

```
unsigned add1(unsigned a, unsigned b) {
  return a+b;
}
 // Perhaps not the most efficient
 // way to add two numbers.
unsigned add2(unsigned a, unsigned b) {
  if (a == 0) return b;
  return add2(a-1, b+1);
}
```

Figure 2-11: C source code. Adapted from (Brown and Wilson, 2008)

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}
define i32 @add2(i32 %a, i32 %b) {
entry:
  %tmp1 = icmp eq i32 %a, 0
  br i1 %tmp1, label %done, label %recurse
recurse:
  %tmp2 = sub i32 %a, 1
  %tmp3 = add i32 %b, 1
  %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
  ret i32 %tmp4
done:
  ret i32 %b
}
```

Figure 2-12: Intermediate Representation. Adapted from (Brown and Wilson, 2008)

[40 x int]	Array of 40 integer values.
[4 x int]*	Pointer to array of four int value.
float(int,int*)*	Pointer to function that takes an int and a pointer to int, returning float
{float, %funptr}	Structure where the first element is float and the second element is pointer to function

Figure 2-13: Examples of LLVM types

```
uint %testfunc() {
  %v = load int 4            ; Must load through a pointer
  store int 42, float* %fptr ; Cannot store int through float pointer
  %val = add int %val, 0     ; Definition does not dominate use
  ret int* null              ; Cannot return int* from function returning uint

}
```

Figure 2-14: LLVM types

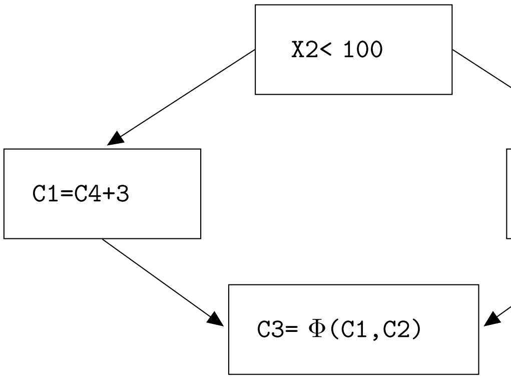


Figure 2-15: Static Single Assignment

Using SSA in LLVM

SSA is essential to code representation in LLVM and as described in section 2.3, it is an intermediate format that helps in optimization and increases instruction level parallelism (Lattner and Adve, 2002; Neustifter, 2010). Figure 2-15 shows SSA form as applied to the LLVM IR. For each instruction that computes a value, the LLVM implicitly creates a new virtual register holding this. For example, the instruction (load int* % ret) converts to (`% C1= load int* % ret`)and this property ensures that the traversal of the "def-use" chain is efficient. When handling control flow, which path will be taken is initially unknown and therefore, SSA defines the Φ node which is the special command that its value is to be determined depending on previous control flow. In LLVM this command is the corresponding phi instruction and depending on which basic block the control came from the incoming value can be determined (Lattner and Adve, 2002; Neustifter, 2010).

The syntax of this instruction is: $< result >= phi < ty > [< val0 >, < label0 >], \ldots$

In figure 2-16 C3 is assigned the value of C1, if the control reaches this instruction from the basic block labelled BB1 or C2, if the control reaches this instruction from the basic block

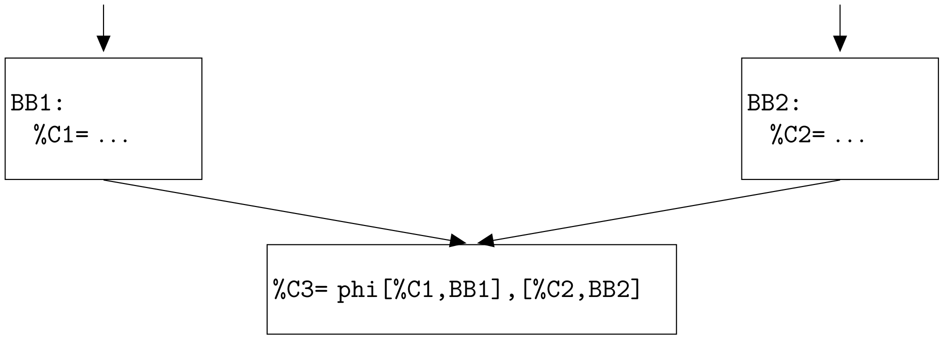


Figure 2-16: LLVM representation

labelled BB2.

2.5.2 Optimization and the Pass system

The most powerful feature of the LLVM is the pass system. In fact, all of its analysis, optimization, transformation, IR and machine code generation is carried out through passes, with each being written as a C++ class that is derived (indirectly) from the pass class. Moreover, there is dependency between the LLVM passes in that work in the current pass depends on that which has been undertaken by a previous one. For this reason, the pass order should be respected so as to prevent the current pass from destroying any work from previously run passes. The LLVM pass manager is used to solve the interdependency between passes and to determine the pass schedule. One of the main restrictions that LLVM passes must obey is that they only have the right to modify the element inside the object (function, basic block) to which the pass is applied. For example, a loop pass has the right to modify basic blocks inside the current loop, but not at the same or a higher level, which thus provides the pass manager the ability to schedule the passes in parallel. The LLVM optimizer has many different passes, which read the LLVM IR, work a bit on it and then produce the optimized and faster codes. The optimization structure has three parts:

1- search the program body to find the part that is wanted to be transformed.

2- confirm that this transformation is correct or safe.

3- perform the transformation.

In figure 2-17, the first part represents the most trivial transformation which is pattern matching transformation on arithmetic identities and the second part shows it representation in LLVM

Instruction simplification interface that is used as utilities by the different of higher level transformations have been provided by the LLVM for these type of peep-hole transformation.

```
  x-x  is  0 ;
  x-0  is  x  and
  (x * 2) - x  is  x

===== first part


and the following shows how these expressions are represented in the LLVM
temp1 = sub i32 %x , %x
temp2 = sub i32 %x , 0
temp3 = mul i32 %x , 2
temp4 = sub  i32  %temp3 , % x

====== second part
```

Figure 2-17: arithmetic identities and their intermediate
representation in LLVM

```
// X - 0 -> X
if (match(Op1, m_Zero()))
  return Op0;
// X - X -> 0
if (Op0 == Op1)
  return Constant::getNullValue(Op0->getType());
// (X*2) - X -> X
if (match(Op0, m_Mul(m_Specific(Op1), m_ConstantInt<2>())))
  return Op1;
...
return 0;  // Nothing matched, return null to indicate no transformation.
```

Figure 2-18: Pattern matching Optimization

These transformations that illustrate in figure 2-18 are contained in the **SimplifySubInst** function .

`Op1` and `Op0` represent the two operands of a subtract instruction, left and right. However, the two functions match and `m_` help to carry out a declarative pattern matching operation on LLVM IR code. For example, if the left hand side of a multiply operation is the same as `Op1` then the `m_Specific` function returns that there is a match between the `Op0`, which represents the `(X*2)` and the output of the multiply operation. All these previous cases are performing such pattern matching. These three cases of pattern matching, and all the others, are called from a **SimplifyInstruction** transformation function that subsequently, returns the replacement, if there is a simplification, but otherwise it returns a null pointer. Figure 2-19 clarifies how this function is used. It scans all the instructions in a BasicBlock to see if there is any optimization or simplification it can achieve. Where there is a simplification the **replaceAllUsesWith** method will be called to update the code with the simpler form (Lattner, 2002).

```
for (BasicBlock::iterator I = BB->begin(), E = BB->end(); I != E; ++I)
 if (Value *V = SimplifyInstruction(I))
   I->replaceAllUsesWith(V);
```

Figure 2-19: SimplifyInstruction used to apply transformations

The compilation strategy in LLVM has the ability to support multi-stage optimization throughout the life time of the program, such as compile-time, link-time, runtime, and offline. Below illustrates the multi-stage optimisation of the LLVM compiler:

1- Compile Time: Front-end and Static Optimizer

The period in which the compiler translates the source code to machine code is called compile-time and during this time the compiler does as much static optimization to alleviate the work of the link-time optimizer. As illustrated before, the main job of the LLVM front-end is to translate the source language to the LLVM virtual instruction set. In addition, it can do some language specific optimization such as transforming the call of print function, for example, the `printf("hello \n")` in C and C++ languages to `puts("hello")`. In addition, all the optimization or transformation passes in LLVM are built into libraries, so the static compilers have ability to use some or all of them for interprocedural optimization to enhance the code generation capabilities (Lattner, 2002).

2- Link Time: Linker and Interprocedural Optimizer

Similar to static optimization, link-time optimization also operates on the LLVM bytecode file (This file is used to store the intermediate representation on disk in compacted form). To speed up the link-time analysis and optimization, the link-time optimizer uses the interprocedural summaries that have been computed at compile time as an input instead of recomputing them. The interprocedural summaries are computed for each function in the program and attached to its LLVM bytecode. Because most of the program parts are present during this time it is the right place to accomplish aggressive interprocedural optimization.

Both interprocedural analysis such as Data Structure Analysis, call graph construction, and Mod/Ref analysis, as well as interprocedural transformation, such as inlining, dead type elimination, dead argument elimination (Kowshik et al., 2002), constant propagation, array bounds check elimination, simple structure field reordering and automatic pool allocation (which is used to control the pool allocation library instead of general heap allocation) are included in the LLVM (Lattner, 2002; Lattner and Adve, 2004).

3- Run Time: Profiling and Reoptimization

This optimization works by collecting profile information during program runtime. Then it recompiles and reoptimizes the LLVM bytecode by using this information. There are different techniques that the runtime optimizer uses to collect the profile information, examples which are: the PC sampling (Anderson et al., 1997)(which is used to find the hot functions and loops) and path profiling (which detects the hot paths through complex region code). This LLVM runtime optimizer has the ability to choose whether to undertake aggressive optimization, which updates the LLVM bytecode and regenerates machine code for it or perform light-weight optimization, which works directly on the recompiled native machine code and which at the same time it refer to the LLVM bytecode which has the high-level information, such as data flow and types (Lattner, 2002; Lattner and Adve, 2004).

4- Idle Time: Offline Reoptimizer

This type of optimization works during the idle time of the user's computer. The reason for this is that some of the application programs are large and none of the program part are "hot"(it has the highest execution time) so the run time optimizer will not spend time to improve them. The Offline optimizer is used to assist this type of application and also to assist other optimizations that need expensive analysis. Although the run-time optimizer cannot find the hot piece of code appropriate for this optimization, it still has the ability to find the most frequent paths executed. The profile information of the program that has been collected from the run time optimizer with its LLVM bytecode is used by the Offline optimizer to accomplish aggressive profile driven interprocedural optimization so as to recompile and reoptimize this application (Lattner, 2002; Lattner and Adve, 2004).

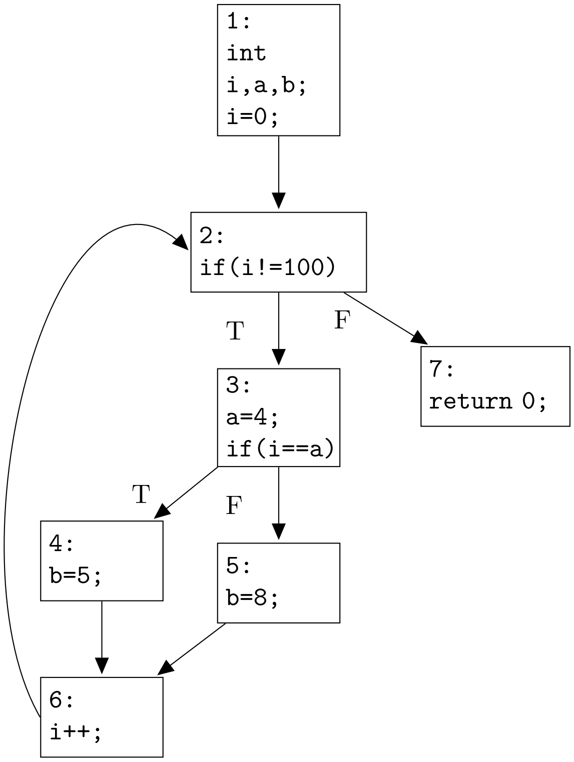
2.5.3 Back-end

The back end or code generator is responsible for transforming the LLVM IR into the best machine code for a given target. In other words, the job of the code generator is to produce assembly or object code for different architectures that have different specifications. Two types of passes need to be available to generate machine code for a target machine, the built in passes, which run by default and individual passes, like: instruction selection, register allocation, scheduling, code layout optimization, and assembly emission. Moreover, the target author has the ability to select from default passes that suit his target machine requirements.

The above features give the target author the power to use these default passes to construct the target machine code needed without having to write these passes right from the beginning. For example, since the X86 machine has very few registers, it uses the register-pressure reducing back-end scheduler (which is beyond the scope of this thesis), whilst the PowerPC machine uses a latency optimizing back-end scheduler because it has many registers. However, this mix and match process brings forward the problem that these shared components need to have some information about the target machine, for example, the number of available registers. LLVM solves this problem by supporting several description files that define the properties of a certain architecture, which are processed by the `tablegen` tool.

```
void foo(){
int i,a,b;
  for (i=1 ;i!=100; i++)
    {
    a=3;
    if(i==a)
       b=5;
    else
       b=8;
     }
    return 0;
  }
```

(a)



(b)

Figure 2-20: (a) Piece of code. (b) Control flow graph

2.5.4 Analysis Passes

As pointed out above, a key part of the LLVM system is its passes, which carry out most of the work of the compiler, such as transformations and optimizations. Moreover, LLVM introduces some analysis passes that have been used to analyse LLVM IR and collect the information to be used during LLVM optimization. Below, some analysis passes that are employed in the current work, where the piece of code in figure 2-20 and its CFG will be used for clarification.

1- Dominator Tree

This pass describes the relation among CFG basic blocks by providing dominance information that will be used by other passes as a common analysis. By applying the terminology presented in section 2.1 in the figure 2-20 it can be determined that basic block 2 dominates basic block 4, because all paths from 2 to 4 pass through 2, while basic block 3 represents the intermediate dominator of basic block 4.

2- Loop Information

LLVM offers analysis to diagnose a natural loop (the loop that has one entry and can cause cycles in CFG) and the basic block that is contained within this loop. The loop entry is called the header and it can be shared with other loops if there are nested ones. Through the piece of code below the concept of back edge is illustrated (Grosser, 2011). Consider figure 2-20, which represents the control flow of a piece of code, the edge 6-2 is the back edge if basic block 2 is dominating all the basic blocks that can reach 6 without passing through 2. The nodes (basic blocks) 2, 3, 4, 5, 6 all represent a natural loop, where node 2 is the loop header. In addition, if a basic block presents in two loops `L1` and `L2` this means that one of these loops is the child of the other.

3- Alias Analysis

LLVM introduces several alias analysis passes that work together to deliver accurate analysis information about the relationship between two memory accesses. The input to this pass will be two memory addresses and the output result has one of the following responses to describe the nature of the relationship (LLVM Project, 2013).

- no alias

  means that these two pointers do not access the same memory location.

- may alias

  means that these are incomplete or that there is no information about memory access (two pointers might refer to the same memory address).

- partially alias

  means that these two pointers do access the same memory object but do not start at the same address.

- must alias

  means that these two pointers do access the same memory location.

2.6 Some of the LLVM Classes and Pass

One of the first things that needs to be done when designing a new pass is to decide what class should be used. In this section, some classes that have been used for this research to implement the DSWP/Slice technique are discussed (LLVM Project, 2012).

- LoopPass class

  The LoopPass works in reverse order such that it executes the inner most loop first and the outer most last, with each loop in the function being considered independently of other loops in that function. By using the LPPassManager interface, LoopPass subclasses have the ability to update the loop nest. Three virtual methods belonging to the LoopPass class are overlapped to do LoopPass work and their return values are true if they modify the program of interest, but false otherwise. These methods are:

  - doInitialization(Loop, LPPassManager) method

    This method assigns initial variable types and operation values. It works independently and does not rely on the functions being processed. Moreover, the LPPassManager interface should be used to access function or module level analysis information.

  - The runOnLoop(Loop, LPPassManager) method

    The runOnLoop is implemented to perform whatever action is necessary for the specified Loop, such as transformation or analysis work. The return value of this function is true if this loop is modified, but false otherwise and the LPPassManager interface should be used to update a loop nest.

  - The doFinalization() method

    The doFinalization method is called for every loop in the program after the runOnLoop has finished its work.

- MemoryDependencyAnalysis class

  This class is an analysis class that determines, for a given memory operation, which preceding memory operations it depends on. The memdep pass uses alias analysis to provide high-level dependence information about memory-using instructions and will inform which store feeds into a load.

  Some of the MemoryDependencyAnalysis class members:

  - getDependency (Instruction %*QueryInst)

    This method returns the instruction on which a memory operation depends.

  - getInst()

    If this is a normal dependency, it returns the instruction that is depended upon, but otherwise, the return is null.

  - isDef()

    Returns true if this MemDepResult represents a query that is an instruction definition dependency.

  Figure 2-21 below shows a piece of code that illustrates how to collect different types of dependency (true, output and anti dependences) from programs.

- Value Class

  All values computed by the program that may be used as operands to other values use the Value class as a base class. In addition, some classes use the Value class as a super class, such as instruction or function and hence, this Value class is very important to these classes. All values have a type as well as having a `Use list` that keeps track of which other values are using this particular value and this `Use list` is called the def-use chain. Figure 2-22 shows how the def-use chain can be used to determine which use employs a

```
MemDepResult mdr = mda.getDependency(inst);
 if (mdr.isDef()) {
    Instruction *dep = mdr.getInst();
    if (isa<LoadInst>(inst)) {
       if (isa<StoreInst>(dep)) {
          addEdge(dep, inst, DTRUE);
//READ AFTER WRITE
         }
      }
    if (isa<StoreInst>(inst)) {
     if (isa<LoadInst>(dep)) {
       addEdge(dep, inst, DANTI);
//WRITE AFTER READ
        }
    if (isa<StoreInst>(dep)) {
       addEdge(dep, inst, DOUT);
//WRITE AFTER WRITE
       }
  }
```

Figure 2-21: Memory dependency

```
for(BasicBlock::iterator ii = BB->begin(); ii != BB->end(); ii++)
 {
  Instruction *inst = &(*ii);
  for(Value::use_iterator ui = ii->use_begin(); ui != ii->use_end(); ui++)
   { if (Instruction *user = dyn_cast<Instruction>(*ui))
     addEdge(inst, user, REG);
   }
 }
```

Figure 2-22: Register dependency

particular value (in this research this piece of code is used to find register dependency). For instance, if an instruction named inst is assigned to a particular basic block BB and we want to find all of the instructions that use it, then this is as simple as iterating over the def-use chain of inst:

- Function Class

  In LLVM, a function body consists of a list of basic blocks, with each having a set of instructions. The last of these is a special one called the terminator instruction, which represent either a branch instruction or a function return. Moreover, a function body has two special basic blocks, an entry block which is the first block in it and an exit block which is the last one. The entry block in a function body has two special properties:

- It represents the entrance to the function body and it is the first block that will be executed.
- It can not have a predecessor and also cannot have any phi nodes. Moreover, it is not allowed to have branches to the entry block of a function body.

Some function class members that are used in passes for this research are:

- getFunctionType()

  this function returns the type of the function.
- setCallingConv(CallingConv::ID CC)

  sets the calling convention of this function call.
- getBasicBlockList()

  Obtains the basic block contained in this function and the basic block list is empty for external functions.

2.7 Summary

This chapter has introduced some of the concepts and essential information that are employed throughout this thesis. To start with, graph theory definitions were given and some appropriate terminology, e.g. CFG, SCC, DAGscc, dominator, post dominate have been discussed, which represent the basis of understanding the PDG. Subsequently, some details of the CDG and DDG were presented as well as explanation being provided regarding how they can be used to construct the PDG that will be employed to design both Slice and DSWP techniques in chapter 4. Moreover, a description was given of SSA representation and its uses in the LLVM. Finally, there was an overview of the LLVM architecture and its three phase design which are Front-End, Optimization and Back-End. Also some of the LLVM classes that have been used to implement this current research have been presented. The next chapter discusses the slicing technique and also presents some loop parallelizing techniques.

Chapter 3

Parallelizing Techniques

Several parallelizing techniques have been proposed to enhance the performance of sequential legacy applications by transforming them into parallel ones. These techniques focus on converting single-threaded applications to semantically equivalent multi-threaded versions that can deliver a good performance across a wide range of applications. This chapter presents an overview of several parallelizing techniques and it divides into three sections which are slicing, extracting parallelism and the summary. The first section dealing with slicing technique which is organized into nine (sub)sections. Section 3.1.1 focuses on the types of slices and section 3.1.2 discusses dynamic and static slices. Also two methods to compute slices depending on the CFG and PDG are presented in subsections 3.1.3 and 3.1.4. Sections 3.1.5, 3.1.6 and 3.1.7 deal with intraprocedure and interprocedure slices as well as interprocedural slicing in the presence of aliasing. Section 3.1.8 considers the parallel execution of slices. And finally subsection 3.1.9 describes the application of program slicing. The second section presents some of the loop parallizing techniques and is organized into four subsections. Sections 3.2.1 and 3.2.2 focus on Instruction Level Parallelism (ILP) and Fine-grain Thread Level Parallelism (TLP). Section 3.2.3 discusses Loop level parallelism (LLP) and presents three methods to parallelize the loop body which are: Independent Multi-Threading (IMT), Cyclic Multi-Threading (CMT) and Pipeline Multi-Threading(PMT). And finally section 3.2.4 describes the usage of the DSWP. And last there is the summary section which introduces a brief overview of all parallelizing techniques presented in this chapter.

3.1 Slicing

Given a statement s and subset v of the variables in the program p, the program slice `(s,v)` is the set of all statements whose execution may be necessary for the correct computation of the values of v (Binkley and Gallagher, 1996). It is computed by removing all the statements from the program that have no effect on the slice criterion and is a pair`(s,v)`. The extracted statements are called a slice and the technique is called program slicing (Harman and Hierons, 2001). The slice concept was originally introduced by (Weiser, 1984), whose slice was called the

executable backward static slice. The first term, **executable**, was because of the Weiser requirement for the slice to be executable whilst still producing the same original program behaviour and the **backward** term describes the way of computing the slice. That is, a control-flow graph (CFG) was used as an intermediate representation to get a slice by traversing the graph edge in a backward manner. Finally, he referred to it as a **static** slice because its computation depended on static analysis of the program without considering the program input(Binkley and Gallagher, 1996). In 1990 Horwitz et al. (1990) introduced the concept of the forward slice, which is able to address the question ”What statements are affected by the value of variables v at statement s?”.

Various slightly different notations of program slices have since been proposed, as well as a number of methods to compute them. These varieties have emerged because different applications require different properties (Weiser, 1984; Tip, 1995). Figure 3-1 illustrates slicing of a sample program for a given slicing criterion. To see how slicing works, consider the simple example fragment in figure 3-1 (a). The statements which are significant in this case are those that affect the variable *lines* until statement number *16* of the program. Figure 3-1(b) shows the extracted slice.

```
 1 *  read(text);                       1  read(text);
 2    read(n);
 3    lines = 1;                        3  lines = 1;
 4    chars = 1;
 5    subtext = "";
 6    c = getChar(text);                6  c = getChar(text);
 7    while (c != ’\eof’)               7  while (c != ’\eof’)
 8    if (c == ’\n’)                    8  if (c == ’\n’)
 9    then lines = lines + 1;           9  then lines = lines + 1;
10  * chars = chars + 1;
11    else chars = chars + 1;
12    if (n != 0)
13  * then subtext = subtext + c;
14    n = n - 1;
15    c = getChar(text);                15 c = getChar(text);
16    write(lines);                     16 write(lines);
17    write(chars);
18  * write(subtext)

       (a)                                     (b)
```

Figure 3-1: (a) An example program. (b) A slice based on slicing criterion $< 16, \{lines\} >$. Adapted from (Silva, 2012)

3.1.1 Types of Slices

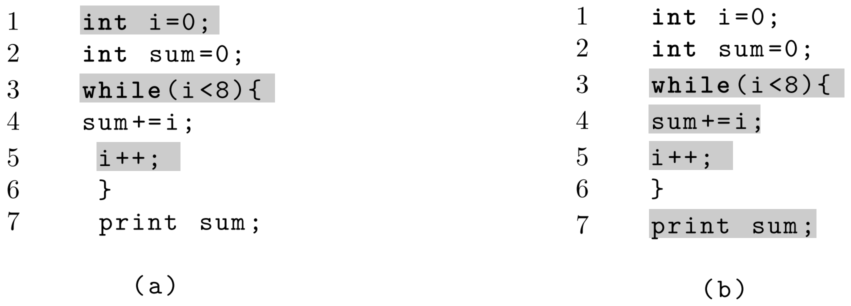
There are several types of program slicing. **Backward slicing** at program point p is computed by working backwards from this point and extracting all the instructions that may affect p and removing any other ones (Tip, 1995; Zilles and Sohi, 2000). **Forward slicing** is the opposite of backward slicing and in fact, many of the algorithms for the latter can be simply reversed to work for the former. A forward slice at program point `p` is the program subset that may be affected by `p` (Tip, 1995; Harman and Hierons, 2001).

A third type of program slicing which combines both backward and forward slicing is called **Chopping**, being used in a wide range of applications (e.g. path condition, input validation, and reducing a program for model checking). To compute chopping, slice $chop(s,t)$ for a given program P1, where s represents the source statement and t the target statement, which is accomplished by collecting all the statements influenced by s, but those influencing t and then identifying the intersections between them (Giffhorn, 2009).

In addition to these described types there are other ways of restricting slices, such as **Dicing** and **Barrier** slicing (Tip, 1995; Harman and Hierons, 2001). **Dicing** was introduced by Weiser as a debugging methodology, which recognizes a set of statements in a slice that are likely to contain a bug. Lyle and Weiser defined this process as: "remove those statements of a static slice of variable that appears to be correctly computed from the static slice of an incorrectly valued variable" (Chen and Cheung, 1993). After performing several tests for the program output, some will be correct and others not. The statements that participate in the incorrect slice and are not contained in the correct one will be more likely to contain the error. For example, consider the program in figure 3-1, by specifying many of the criterion slicing points like $< 16, \{lines\} >, < 17, \{chars\} >$. Assume that the slice for point $< 16, \{lines\} >$ produces incorrect output and is represented by the statement numbering $1, 3, 6, 7, 8, 9, 15$ and 16, while the point $< 17, \{chars\} >$ that produces the correct output is represented by the statement numbering $1, 4, 6, 7, 8, 10, 11, 15$ and 17. The dicing slice pertaining to the statements that cause the error will contain 3, 9 and 16 (Silva, 2012).

Barrier is a filtering slice which is an extension of a chopping slice, whereby a barrier is introduced which is not allowed to be passed during slice computation, thus adding an additional restriction or additional knowledge during the chopping slice computing. Using this barrier the programmer can identify which parts of the program will be visited and which not. For example, during program debugging, some parts are bug free so they should be excluded from the slice so as to make the process more focused (Krinke, 2003). A set of nodes or edges in the PDG will be associated with the slice barrier that will represent the fence that the computation should be stopped at when reached. As a consequence the barrier slicing criterion will have three entries $< p, v, b >$. The first two (p, v) have the same meaning as a static slice, while (b) is a set of statement numbers that symbolizes a barrier (Silva, 2012).

In this thesis, a static backward slice is used because it preserves the original program behaviour of any input and gives an executable slice as shown in figure 3-2.



```
int i=0;
int sum=0;
while(i<8){
sum+=i;
 i++;
 }
 print sum;
```

(a)

```
int i=0;
int sum=0;
while(i<8){
sum+=i;
i++;
}
print sum;
```

(b)

Figure 3-2: Figure 1 :(a) Backward slicing . (b) Forward slicing. Adapted from (Zilles and Sohi, 2000)

3.1.2 Static and Dynamic Slice

The main two ways to compute a slice are static and dynamic.

Static slice : According to Weiser's definition, is a set of statements S in program P that may affect the value of variable `v` at some point `x`. After constructing control and data dependency for a program, a static slice can be computed by finding consecutive sets of indirectly relevant statements. Under these circumstances, just static information will be used to compute the slice and this preserves the effect of the code for every possible execution of the program, thus resulting in the slice being correct for all inputs (Tip, 1995). The disadvantage with the static slice is the resulting slice tends to be large (Lucia, 2001).

A **Dynamic slice** exploits specific information available about the input during its execution. Thus, through the execution of the program some statements are executed many times with different values. For this reason the slice criterion that has been used in a static slice is not enough to extract the dynamic slice. So, both of traditional slicing criterion and the information on a particular execution of the program are used to extract dynamic slice. As a consequence, the extracted slice will be correct for certain inputs.

That is, the only dependencies(data and control) that happen in the specific execution of the program (for certain inputs) are taken into account (Tip, 1995), where the extracted slice is smaller compared to the static slice. But there is a price to pay in terms of space and time.The main benefit of forward slicing in dynamic execution is that the space complexity is bounded as compared with the backward method, which needs to record the execution trace of the program. That is, in a backward dynamic algorithm the execution trace of the program is first recorded and then dynamic dependence relationships are driven by tracing backwards the execution trace. Subsequently, these dynamic dependence relationships are used to compute the dynamic slice. In contrast, a forward slice algorithm computes the dynamic slice during the program execution trace without recording the execution trace (Korel and Yalamanchili, 1994). Figure 3-3(b) shows the dynamic slice for the source code in Figure 3-3(a) with respect to the sum variable for input `n=0`. Thus, the extracted dynamic slice will be statements numbered 4

```
 1  main()
 2  { int N,i,sum;
 3    iread(0,&N);
 4    sum=0;
 5    i=1;
 6    while(i <= N){
 7       sum +=i;
 8       i++;
 9      }
10   printf("%i\n",sum);}
```

(a)

```
main()
{    int    N, i, sum;
     iread(0,&N);
     sum=0;
     i=1;
    while ( i <= N ) {
              sum +=i;
              i++;
        }
  printf("%i\n" , sum) ; }
```

(b)

```
 1  main()
 2  {
 3
 4  sum=0;
 5
 6
 7
 8
 9
10  printf("%i\n",sum);}
```

(c)

Figure 3-3: (a) An example program. (b) A static slice of a program with respect to the final value of a variable sum.(c) A dynamic slice of program with respect to the final value sum for input n=0. Adapted from (Tip, 1995)

and 10.

Most forms of program slicing are syntax preserving. That is, they leave the syntax of the original program largely untouched and simply remove statements to create a program slice. The exception to this rule is that if removing statements may cause a compilation error then they may be altered (for example in Java, removing the `'then'` part of an `if` statement, where the if statement does not use a block statement).

In most cases, dynamic slicing produces smaller slices than the static approach. However, because the former needs to record program execution and analyses every statement in the program, this makes run time overheads larger than for the latter. That is, a static slice is faster because it is just needs to perform static analysis (Kumar, 2001). There are other ways to compute slices, three of which are worthy of note here: **Hybrid, Conditional** and **Amorphous** slicing.

Hybrid slice: Because of the disadvantages of the previous dynamic and static slicing methods in terms of the large time and space consumption for the dynamic slice and the inaccurate and long slicing for the static one, Gupta et al. (1997) proposed the Hybrid slice. This works as a combination of the last two slices, being more precise and smaller than the static one and also less expensive than the dynamic one. The concept behind the hybrid slice is using dynamic information in static analysis to eliminate a number of statements that will not have been executed through a particular execution. This information that will be given to the slicer is a set of breakpoints. Consider the program in figure 3-1, where there are four breakpoints marked with '*'. By giving this information to the slicer it can predict which parts (depend on the '*' sign) are going to be executed and so the set of breakpoints $\{1, 13, 13, 13, 10, 13, 18\}$ can tell that both of the if statement branches will be executed. There are three entries in the hybrid slicing criterion $\langle p, s, \{b_1, \cdots, b_n\}\rangle$, where the first two have the same meaning as the static slice, while the $\{b_1, \cdots, b_n\}$ symbolize sequence of breakpoints. Thus, the extracted slice for the slicing criterion is $\langle 17, \{chars\}\rangle$ as shown in figure 3-4 (Silva, 2012).

Conditional slice: It works as a link between a dynamic slice, which wants to know everything about a program's inputs, and a static slice which does not want to know any of this. That is, it wants to know about the input without being so specific as to give precise values, so the extracted slice depends on a condition. The first to start work with this slice were Ning et al. (1994), while it was formally introduced for the first time by Canfora et al. (1994). The slicing criterion for a conditional slice contains four entries $< i, F, p, v >$, where the i symbolizes the subset of the input variables of the program, F is logic formula (set of potential inputs of the program, where this input can be infinite), p is a statement of the program and finally, v is a set of the variables that will be sliced. For example, the conditional slice that will be extracted from the program in figure 3-1(a) based on slicing criterion $\langle(test, n), F, 18, \{subset\}\rangle$ for $F = (\forall c \in test, c \neq' \backslash n'.n > 0)$ is illustrated in figure 3-5.

In contrast to all the slicing techniques that depend on keeping the part of the semantic program and its syntax, **Amorphous slicing** allows for making syntactic changes as long as the relevant semantics are preserved (Harman and Hierons, 2001). That is some program transformation is used to produce this slice, which depends on simplifying the program and preserving the semantics for specified slicing criterion. The obtained slice will be significantly smaller than the traditional slice, because of the syntactic freedom giving permission to these transformations to perform more simplifications. For example, consider the program in figure 3-6(a). With respect to the slicing criterion $\langle 3, \{chars\}\rangle$, 3-6(b) will represent the interprocedural slice (to be explained later) after removing statements 4, 5 and 6, while 3-6(c) represents the amorphous slice after performing some simplification and evaluation.

3.1.3 Slicing Control Flow Graph

This section introduces the control flow graph as an intermediate representation that is used as the basis of the process of computing the extracted slice where for each node in the CFG there are two sets associated with it, which are:

```
1   read(text);
2
3
4   chars = 1;
5
6   c = getChar(text);
7   while (c != '\eof')
8   if (c == '\n')
9
10
11   else chars=chars+1;
12
13
14
15   c = getChar(text);
16
17   write(chars);
```

Figure 3-4: Hybird slice based on slicing criterion $< 17, \{chars\} >$. Adapted from (Silva, 2012)

```
1   read(text);
2   read(n);
3
4
5   subtext = "";
6   c = getChar(text);
7   while (c != '\eof')
8   if (c == '\n')
9
10
11
12  if (n != 0)
13   then subtext=subtext++c;
14   n = n - 1;
15   c = getChar(text);
16
17
18   write(subtext);
```

Figure 3-5: Conditional slice of figure 3-1(a) with respect to slicing criterion $\langle (test, n), F, 18, \{subset\} \rangle$ for $F = (\forall c \in test, c \neq' \backslash n'.n > 0)$. Adapted from (Silva, 2012).

```
1  program main
2  chars = sum(chars,1);
3  end
4  procedure increment(a)
5  sum(a,1);
6  return
7  procedure sum(a,b)
8  a = a + b;
9  return
```

(a) program 1

```
1  program main
2  chars = sum(chars,1);
3  end
4
5
6
7  procedure sum(a,b)
8  a = a + b;
9  return
```

(b) program 2

```
1  program main
2  chars = chars+1;
3  end
```

(c) program 3

Figure 3-6: Example of amorphous slicing. Adapted from (Silva, 2012)

1- $REF(n)$ which represents the set of variables whose values are referenced at n.

2- $DEF(n)$ which represents the set of variables whose values are defined at n.

Computing a slice using a CFG consists of two steps: First, after the data flow information is computed, this is used to extract a slice and is represented by a set of relevant variables. This set of relevant variables for a certain point in the program $< s, v >$, represents all the variables that affect directly or transitively the computation v at s. The second step is responsible for determining the statements of the slice. So, to compute a slice for a straight line code, which consists of a set of statements executed sequentially, the relevant set of information for a certain point in the program has to be computed which comprises four steps:

- The relevant set is initialized to the empty set.
- Variable v is added to the relevant set.
- For two statements n and m, where n represents the intermediate predecessor for m, $relevant(m)$ is:

$$(relevant(n) - DEF(m)) \cup (REF(m) \textit{ if } relevant(n) \cap DEF(m) \neq \emptyset)$$

- The third step is repeated for all of m's predecessors and terminates when it reaches $n_{initial}$, working backwards.

Consider column 5 in the table in figure 3.1, which represents the relevant set of a slice with respect to $< 8, a >$. This relevant set is computed in backward order. That is, the relevant set for statement 8 will be computed first, followed by that for statement 7 and the process will be continuous until the first statement, when it stops. For example (Binkley and Gallagher, 1996):

$$\begin{aligned}
&relevant(7) = (relevant(8) - DEF(7)) \cup (REF(7) \textit{ if } relevant(8) \cap DEF(7) \neq \emptyset)\\
&= (\{a\} - \{a\}) \cup (\{b, c\} if \{a\} \cap \{a\} \neq \emptyset)\\
&= \{b, c\}
\end{aligned}$$

After the relevant set has been defined these statements are accumulated to configure the final slice and hence, this has the statements in lines 7, 6, 2 and 1. Both the relevant set and the slice statements are computed in a single pass. For the structured control flow the previous algorithm needs to be updated to manage this. That is, three things have to be added:

1- A new set, which is associated with the statement n and is called the control set, which represents the predicate statements that control the execution of n's statement directly.

2- An extra rule has to be added, which is related to a join statement (two nodes have the same predecessor). That is, to compute the relevant set for this join node, both the relevant sets for the nodes (statements) that are the join statement represents their predecessor should be combined.

Table 3.1: Relevant Sets for <8,a>

n	statement	refs(n)	defs(n)	relevant(n)
1	b=1		b	
2	c=2		c	b
3	d=3		d	b,c
4	a=d	d	a	b,c
5	d=b+d	b,d	d	b,c
6	b=b+1	b	b	b,c
7	a=b+c	b,c	a	b,c
8	print a	a		a

Table 3.2: Relevant Sets for <13,a>

n	statement	refs(n)	defs(n)	control(n)	relevant(n)
1	b=1		b		a
2	c=2		c		a,b
3	d=3		d		a,b
4	a=d	d	a		a,b
5	if(a)then	a			b,c,d
6	d=b+d	b,d	d	5	b,d
7	c=b+d	b,d	c	5	b,c
8	else			5	c,d
9	b=b+1	b	b	8	c,d
10	d=b+1	b	d	8	c,d
11	endif				b,c
12	a=b+c	b,c	a		b,c
13	print a	a			a

3- Repeat calculating the relevant set. This happens in the presence of the loop and this repetition occurs over the loop body until does not change the relevant set or slice.

Figure 3.2 shows the relevant set for the statement in column 2 in the presence of the if statement with respect to <13,a>. This set is computed in backwards order, from line 13 until line 1. So the statements in lines 8 and 5 are included in this slice because *relevant*(10) contains *control*(8) and statement 8 contains *control*(5). Both lines 5 and 8 are included in this slice (Binkley and Gallagher, 1996).

3.1.4 Slicing with the PDG

Ottenstein and Ottenstein (1984) concluded that the PDG is the best and most efficient intermediate representation to extract slices, stating that “explicit data and control dependency make the PDG ideal for constructing a program slice”. It gives the ability to slice program in linear time on the number of nodes in the PDG. Both algorithms used to compute intraprocedure and interprocdure slicing have three steps (Binkley and Gallagher, 1996):

1- Build the PDG for a given program (see chapter 2)

2- Slice the graph obtained from the first step.

3- Get the slice from the sliced graph.

Horwitz et al. (1989) introduced a adapted version of a PDG to that described by Ottenstein, where each vertex in the directed graph represents an assignment statement or control predicate and connects with other vertices by several kinds of edges. It has a special vertex called the entry vertex. Moreover, the edges between two vertices represent control or flow dependencies and the control dependency ones are labelled true or false. Control dependency happens if one of the following conditions is true:

1- There are two vertices, u, which represents the entry vertex and v, which represents a component in the program, which is not nested within a loop or condition. This edge is given a true label.

2- There are two vertices, u, which represents the control predicate for a loop or condition statement L and v, which represents a component immediately nested within L. For a while-loop this edge is given a true label and for a conditional statement it is given a true or false one.

Flow dependency happens between two vertices, v and u, if both of the following conditions are true:

1- v defines a variable x and u uses it.

2- There is a path from v to u and this path does not have another definition for x.

In addition, two types of flow dependency can be seen in the PDG: loop carried and loop independent.

1- Loop carried dependency, denoted by $u \rightarrow_{lc(L)} v$, exists if the flow dependency inside a loop L has a back edge to the loop predicate.

2- Loop independent, denoted $v \rightarrow_{li} u$, exists if the flow dependency inside a loop L has no back edge to the loop predicate .

The extracted slice with respect to vertex s, denoted by $Slice(G, s)$, is a graph that contains all vertices that can reach s via the flow ($_c$) and/or control ($_f$) edges.

$$V(Slice(G, S)) = \{v \in V(G) \mid v \rightarrow^*_{c,f} s\} \tag{3.1}$$

The edges of $Slice(G, s)$ are those in the subgraph of G included in $V(Slice(G, s))$:

$$E(Slice(G, S)) = \{v \rightarrow_f u \in E(G) \mid v, u \in V(Slice(G, S))\} \tag{3.2}$$

$$U\{v \rightarrow_c u \in E(G) \mid v, u \in V(Slice(G, S))\} \tag{3.3}$$

Figure 3-8 shows the PDG for the source code in figure 3-1 and the extracted slice with respect to the program criterion ($product, 10$). The nodes in this graph represent the statements in the program. The bold vertices represent the extracted slice while the black, light blue and dark blue arrows represent control dependency, loop independent dependency and loop carried dependency respectively. The black arrow between the statement that has label $While(I > N)do$ and second the statement that has label $i := i + 1$ means that the execution of the second statement is controlled by the execution of the first (Binkley and Gallagher, 1996).

Choi and Ferrante (1994) showed that the classical PDG is not an efficient representation for constructing executable slices for programs that have arbitrary control flow (programs containing **goto**). Consequently, they presented a modified version focused on making a change in the PDG, but not the slicing algorithm. This new version they called the argument program dependency graph (APDG), which has a set of fall-through edges that represent the lexical successor of the **goto** statement in the source code. Because under this method slices are obtained by removing some statements (like the **goto** statement), the fall-through edges are added to the control flow graph (ACFG) to maintain the correct execution path after these statements have been deleted. An alternative method that uses a separate graph to store additional information without changing the PDG was introduced by Agrawal (1994). Because a jump statement does not assign a value to any variables and has no predicate, there is no data dependency and control dependency between program statements and it. Therefore under this modified version a lexical successor tree and post-dominator tree are used to identify an appropriate jump statement to include in a slice. If the nearest node to the jump statement in the post-dominator tree in a slice is the same as the nearest node to it in the lexical successor in the slice, then the jump statement will be omitted from the slice, otherwise it will be included (Agrawal, 1994).

3.1.5 Intraprocedure Slice

Intraprocedure slice concerns slicing the program without taking into account collecting information beyond the boundary of the main program (do not slice crosses the boundaries of procedure calls). The Weiser slice was classified as an intraprocedural slice, however it was not accurate enough to slice multi-procedure program. As consequence, it does not fail to slice crosses the boundaries of procedure calls; consider the program in figure 3-9. By applying Weiser's algorithm to extract a slice with respect to slice criterion $< 16, \{x\} >$ the whole program will be included in this slice except statement (11), (12) and (13). This incorrect output happens because the Weiser algorithm does not slice the `sum(x,1)` statement alone but slices all other statements that call the sum function, even the ones that are unrelated like statement `sum(lines,1)`. This happened due to this algorithm not memorizing the procedure call information through traversing the program (Silva, 2012).

```
begin
  read(n);
  i:=1;
  sum:=0;
  product:=1;
  while (i>n) do
  begin
    i:=i+1;
    sum:=sum+1;
    product:=product+1;
  end;
  write(sum);
  write(product);
end;
```

Figure 3-7: An example program.

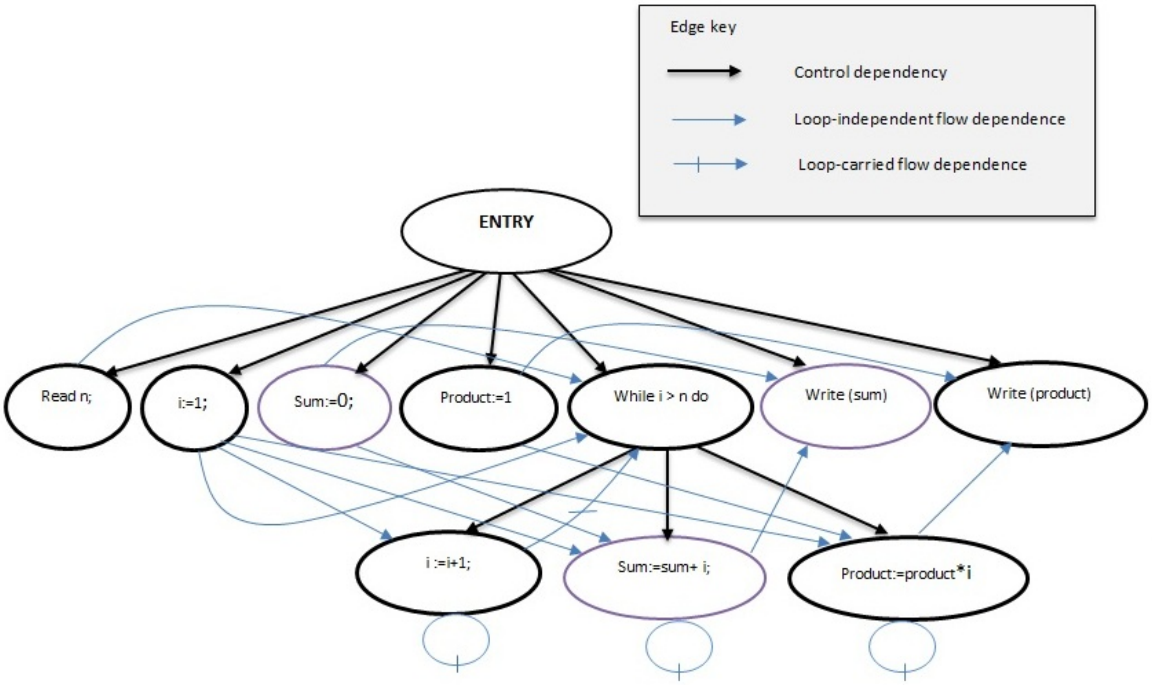


Figure 3-8: shows the PDG for the source code in figure 3-7 and the extracted slice of the program criterion (product, 10). The bold vertices represent the extract slice.

3.1.6 Interprocedural slice

Here, a brief overview of interprocedural slicing is provided, although it is largely outside the scope of this thesis (this work focuses on intraprocedural slicing). This type of slicing deals

```
1     program main
2     read(text);
3     lines = 1;
4     chars = 1;
5     c = getChar(text);
6     while (c != ’\eof’)
7     if(c== ’\n’)
8     then sum(lines,1);
9     else increment(chars);
10    c = getChar(text);
11    write(lines);
12    write(chars);
13    end
14
15    procedure increment(x)
16    sum(x,1);
17    return
18
19    procedure sum(a, b)
20    a = a + b;
21    return
```

Figure 3-9: An example of multi-procedure program. Adapted from (Silva, 2012)

```
1     program main
2     read(text);
3
4     chars = 1;
5     c = getChar(text);
6     while (c != ’\eof’)
7     if (c == ’\n’)
8
9     else increment(chars);
10    c = getChar(text);
11
12
13    end
14
15    procedure increment(x)
16    sum(x,1);
17    return
18
19    procedure sum(a, b)
20    a = a + b;
21    return
```

Figure 3-10: The slicing of a multi-procedure program , program criterion $< 16, \{x\} >$ Adapted from (Silva, 2012)

with generating a slice for the entire program cross boundaries of procedure call (Lakhotia, 1992). Horwitz et al. (1990) presented an algorithm to compute interprocedure static slicing, which depends on extension of the PDG, called a system dependency graph(SDG). The SDG consists of a combination of a PDG for the main program and a procedural dependency graph for each procedure, where the term system referring to a multi-procedure program (Binkley and Gallagher, 1996).

The SDG is constructed from a collection of the producer dependency graphs which are connected by interprocedural control and flow dependency edges. The SDG is similar to the PDG, however several types of edges and vertices are added to it which do not exist in the PDG, which are

1- Call statement, which is represented by the call vertex.

2- Parameter passing which is represented by four types of vertices. The first two of these vertices represent the call site, actual-in/actual-out that symbolize actual parameters passing. The second two represent the called procedure which are formal-in/formal-out vertices, symbolizing the formal parameter passing.

3- Transitive dependence edges which represent the flow of dependency caused by called procedure. These edges also called summary edges which connect between actual-in and actual-out vertices.

In addition, three more edges are added to the procedural dependency graphs to produce the SDG:

1- The edge connecting the call-site vertex and entry vertex is called the control dependence edge.

2- The edge connecting actual-in and formal-in is called a parameter-in edge.

3- The edge connecting actual-out and formal-out is called a parameter-out edge.

Figure 3-10 shows the slicing of a multi-procedure program, program criterion $< 16, \{x\} >$ Horwitz et al. (1990); Silva (2012). Two pass interprocedural slicing over the graph is used to obtain a more accurate slice. With respect to variable s, if it is in the procedure P, the first pass concerns slicing the procedure P by identifying all vertices that reach s (which are contained in P or in the procedure that is called P) without going through the procedures called by P or called by P caller, whilst the second pass concerns the slicing vertices in the called procedure, particularly those vertices that the summary edges use to move to the call site in the first pass (Binkley and Gallagher, 1996).

3.1.7 Interprocedural slicing in the presence of aliasing

The aliasing concept means that there is more than one variable and these have different names that can access the same memory location. Horwitz et al. (1990) introduced two methods to deal with slicing programs that have call-by-reference and aliasing. That is, some adaptations need to be made in the presence of aliasing, but the specific details of these methods are beyond the scope of this thesis. The first method works by transforming the problem of interprocedural slicing in the presence of aliasing to one that is alias free. This goal can be achieved by mimicking the calling behaviour of the system to find out how variables are aliased for each instance of procedure call. The advantage of this method is that it can provide a more accurate slice, however, more time and space is needed to accomplish this conversion. The second method depends on the generalization of the flow dependency to cover all dependencies that happen owing to the presence of aliasing. Flow dependency exists between two vertices if all the following conditions are true:

1- v and u are vertices, v defines variable x and u uses variable y.

2- x and y are potential aliases.

3- In the control flow graph there is a path from v to u and along this path there is no other definition for x or y

The disadvantage of this method is that the produced slices are inaccurate and there may be unrelated statements attached to them (Horwitz et al., 1990).

```
Original Program

1    FOR I:=7 UNTIL 13
              DO
2            IF PRIME(I)
3            THEN WRITE ("A")
4            ELSE WRITE ("B")
              ENDIF
      ENDFOR
      (a)
```

```
SLICE 1 (PROJECTION "A" )

1    FOR I:=7 UNTIL 13
              DO
2            IF PRIME(I)
3            THEN WRITE ("A")
              ENDIF
      ENDFOR


SLICE 2 (PROJECTION "B" )

1    FOR I:=7 UNTIL 13
              DO
2            IF PRIME(I)
4            ELSE WRITE ("B")
              ENDIF
      ENDFOR

      (b)
```

Figure 3-11: (a) An example program. (b) a two slice extract from the original program. Adapted from (Weiser, 1983)

3.1.8 Parallel execution of a slice

Weiser (1984) showed how parallel slicing can be executed for a program. He argued that this form of slicing is appropriate on multiprocessors, because of its ability to decompose the program into independent slices executed in parallel without synchronization or shared memory, by duplicating the computation in each slice. In general, the slices are shorter and executed faster than the original program. However, the arbitrary difference in the speed of slice execution time leads to a **splicing problem**, which is the problem of finding at runtime the correct order of slice output. That is, after getting the output of each slice this output needs to be reordered so as to keep the original program behaviour (Weiser, 1983). Wesier introduced a method to solve the **splicing problem** by merging the statement execution sequences of a set of slices into a single one exactly like the original execution of the program. That is, he made all the sequences of the execution statements into regular expressions so that all possible paths through the program flow graph were represented as regular expressions. Figure 3-11 (a) shows the original program and (b) shows a two slices extracted from the original program. The regular expression for the original program is f(i(a |b)f)*, where f symbolizes the FOR statement, i the IF statement and a and b symbolize two write statements. The regular expressions f(i(a |λ)f)* and f(i(λ |b)f)* represent slices S_1 and S_2, respectively, where λ represents an empty string. Figure 3-12 shows the output after merging that of two slices together by applying the splicing solution B algorithm, which is used to reconstruct the slice output. The algorithm

R=f(i(a|b)f)*

S_1=fiafif

S_2=fifibf

N	S_1	S_2	Output(m)
{ f }	fiafif	fifibf	
{ i }	iafif	ifibf	f
{ a,b }	afif	fibf	i
{ f }	fif	fibf	a
{ i }	if	ibf	f
{ a,b }	f	bf	i
{ f }	f	f	b
{ i }	λ	λ	f

Figure 3-12: An example of solution B. Adapted from (Weiser, 1983)

input represents the regular expression for the original program and for the extracted slices. This method implies that the original program has a reducible graph and the disadvantage of it is the large amount of information that each slice needs to send during its running. For example, if two slices are executed in parallel then when each addresses one statement it should send the symbol that represents this statement to the splicer and depending on the information that has been received, the splicer can determine the order of these statements and then drive the original program behaviour, all of which takes a long time. Badger and Weiser introduced a technique to accomplish a merging operation with minimal communication. Unlike a previous method (Weiser, 1983), that made each slice transmit an amount of information proportional to the running time, this technique transmits a small amount of information and uses the dataflow path to order the computation. For this process a small group of counting variables are added to each slice, which record the information related with the slice progress during its execution and when each slice meets an output statement it sends it to the splicer pairs that represent the slice output and counter variable. Depending on the received information from two slices the splicer will decide which slice output should come out before the other. For example, if there are two slices, $S1$ and $S2$, and set I, which represents the set of the branch nodes that affect the execution of both $S1$ and $S2$, at the beginning the counter variables for each slice are assigned a zero value. Each time the slice passes a node in the I set the value of counter variables for this slice is increased. Moreover, when $S1$ or $S2$ encounter the output statement they will send their information (output, counter variables) to the splicer that makes the slice with the smaller counter variable output appear before that with the larger one (Badger and Weiser, 1988).

Wang et al. (2009) put forward a dynamic framework to parallelize a single threaded binary program using speculative slicing and the major contribution of this work can be summarized as follows:

- It introduces a dynamic parallelization framework for parallelizing binary code transparently for a multicore system, which works by combining the slicing-based parallelizing algorithm with the classical two-phase dynamic optimization scheme. This consists of two phases for identifying and optimizing the often executed code, with the first phase being called the profiling phase and the second the optimization phase.

- Instead of finding the backward slice for the whole program they identified the slice for the hot region (most frequently executed path). In addition, they used a loop unrolling transformation that can help in finding more loop level parallelism in backward slices even in the presence of loop carried dependencies and they proposed an algorithm to determine automatically the optimal unrolling factor. Then, they illustrated how this factor can affect the parallelism.

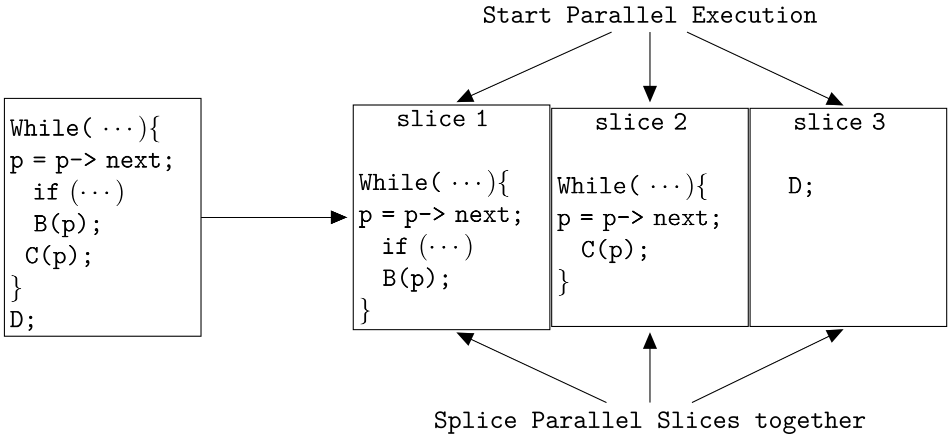
- They developed the slicing based parallelism to work with an irreducible control flow graph. That is, they defined the backward slice using a PDG instead of a program regular expression and added an `Allow list` (commit list). This list, based on the post-dominator relationship, was used to determine the priority of the instruction in each slice. Thus, this `Allow list` was able to solve the ambiguity problem that existed in previous splicing solutions (Weiser, 1983). For example, considering the code in figure 3-13, if the `Allow list` initializes at `w` and if the program is executed at the `while` statements, then after `while` commit then the question will be raised, which of the statements `p` or `d` should follow the `w` statement? This problem is solved by giving priority to the Allow list, which is dependent on the post-dominator relationship.

- Moreover, three types of speculation namely are memory, control and branch, have been added to enhance slice extraction and then increase the parallelism which can be accomplished by breaking the rare dependency that can be found between program instructions.

3.1.9 Application of Program Slicing

This subsection discusses how program slicing can be used in different applications and how these various applications give rise to the need to develop the notation of program slicing.

- Debugging



```
While( ···){
p = p-> next;
  if (···)
  B(p);
 C(p);
}
D;
```

```
While( ···){
p = p-> next;
  if (···)
  B(p);
}
```

```
While( ···){
p = p-> next;
  C(p);
}
```

```
D;
```

Figure 3-13: An example of solution B. Adapted from (Wang et al., 2009)

The original idea of program slicing was to mimic the method that an experienced programmer used to debug a program to find errors. Several slicing tools are beneficial for debugging as they give the programmer a chance to focus on those statements that have caused errors. One of these types of slicing methods is dynamic slicing, which, as explained above, can produce smaller slices than the static form and it can help the programmer to find the bug that is caused by a particular execution of the program at hand. Other types of slices that can be used in debugging are dicing and chopping.

- Differencing and integration

  Program differencing is responsible for identifying a set of components that represent the difference between two versions of a program: the old and the new. Instead of testing the whole new version of the program, only the differencing components need to be tested. This process is accomplished by partitioning the old and new versions of the program and then comparing the behaviour of two components. If they have equivalent behaviour this means they belong to the same partition. An integration algorithm is used to compare the extracted slices to determined if their behaviour is equivalent (Tip, 1995).

- Software maintenance

  Software maintenance focuses on whether the modification of some of a program's components affect the behaviour of other parts and a decomposition slice is used to investigate this. This is a static slice as presented by Gallagher and Lyle (Gallagher and Lyle, 1991), which decomposes the program into a set of components, with each of these capturing the original program behaviour (Tip, 1995). The decomposition slice for one program component x represents all the statements that affect it at some point. The other program components form what is called the complement, which results after removing the decomposition slice from the original program (Gallagher and Lyle, 1991). Furthermore, two slices are independent if they do not have a shared statement and they will be strongly dependent if one of them is a part of the other (Tip, 1995).

- Software comprehension

  Software comprehension is the first and essential step in software maintenance and for a legacy program where the program writer is no longer available and the program documentation no longer exists this of paramount importance. Many types of program slices have been used in software comprehension, such as conditional and constrained slices, extraction of which depends on condition and traditional static slice criteria. This property gives the maintainer the ability to analyse the program from different angles (Xu et al., 2005).

3.2 Extracting parallelism

Much of the research relating to compiler parallelization techniques and parallel architectures has been developed to improve the performance of a wide range of applications (Zhong, 2008). For instance, major microprocessor companies have put many processors into a single chips to increase the throughput and efficiency by executing program computations in parallel (Zhong et al., 2007). The challenge with exploiting these multicore systems in an efficient way is to find more parallelism in the program. This section presents some of the loop parallizing techniques, in particular: DOALL, DOACROSS, and DSWP. Moreover, it introduces some general concepts and reviews several examples of previous research that have used these techniques for extracting parallelism.

3.2.1 Instruction level Parallelism (ILP)

Exploiting instruction level parallelism (ILP) that exists in single thread applications by modern architectures, like Superscalar and VLIW, improves program performance (6-8 instructions per cycle). However, the whole gain from ILP is still low as a result of frequent memory stalls. In a multicore system, low latency communication of operands between cores should exist so as to respect the dependency between instructions. In most architectures these communications are achieved through a shared register file and bypass network, which do not exist in standard multi-core systems (Zhong, 2008; Zhong et al., 2007).

3.2.2 Fine-grain Thread Level Parallelism (TLP)

This parallelism is achieved by partitioning the program into small chunks or fine-grain threads to execute in parallel. The main challenge of extracting fine-grain parallelism, is to determine which operations can be executed in parallel. That is, the communication overhead of moving values between cores and also cache layout need to be taken into account. A poor decision in program partitioning can lead to performance degradation. Moreover, both concurrent execution and overlapping memory latencies (overlapping cache misses or misses with computation have a large potential) are permitted in fine-grain TLP (Chu and Mahlke, 2007; Zhong et al., 2007).

3.2.3 Loop level parallelism (LLP)

Extracting parallelism from loops in an ordinary program has considerable effect on speeding up the program and it has been successfully used in scientific computations (Bridges, 2008). Two reasons make the loop body the best candidate part of a program for parallel execution. First, the iterative nature of a loop results in the same piece of code being executed many times. Second, the fact that most programs spend their execution time inside the loop body, makes it the hottest part in the program that needs parallel execution (Raman, 2009).

By executing loop iterations in parallel, fully utilizing of all cores is achieved. Depend on how these cores handle the dependency between loop iterations, there are three type of parallelism that can extracted from the loop: Independent Multi-threading(IMT), Cyclic Multi-threading (CMT), and Pipelined Multi-threading(PMT). Below a detailed explanation of these types is presented.

Independent Multi-threading (IMT)

This type of parallelism was first introduced to parallelize array-based scientific programs and the most popular IMT technique is DOALL, which introduces a high quantity of parallelism to be utilized in many significant applications. This method works by giving each loop iteration to a thread and then executing all these threads in parallel without cross dependency between them. Several reasons have led to DOALL techniques becoming well-known and suitable for paralleling, these being (Jeon, 2009):

- In many applications, most of a program's execution time is spent inside loop parts and more speedup can gained by parallelizing this part following Amdahl's law.
- In DOALL, each loop iteration should be independent and hence there is neither data communication nor loop carried dependency between loop iterations.
- There is no load balance problem in DOALL, because all loop iterations have the same number of instructions.
- Finally, DOALL can scale linearly with the number of available threads.

To see how this method works consider the program fragment in figure 3-14. If `f` does not present any loop carried-dependence (by modifying p), then all loop iterations can be execute in parallel. Unfortunately, outside the scientific domain and numerical applications, DOALL does not to seem work effectively, for the presence of arbitrary control flow and irregular memory access via pointers make it inapplicable to most loops.

```
int *p=&y[7];
for(i=0;i<N;i++)
 x[i] +=f(p);
```

Figure 3-14: Example code

```
for(i=1;i<6 ;i++) // A(i)
  {
   statements;    // B(i)
 }
```

Figure 3-15: DOALL example code. Adopted from (Raman, 2009)

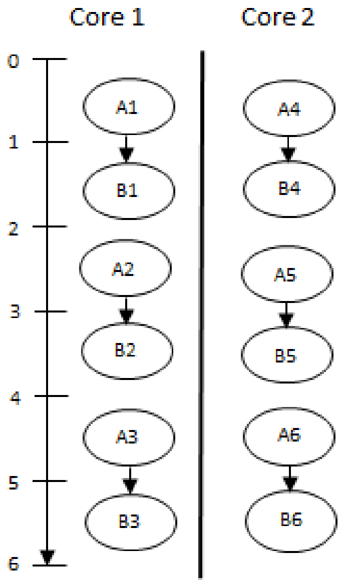


Figure 3-16: applying DOALL. Adopted from (Raman, 2009)

Figure 3-16 illustrates the execution schedule for the program in figure 3-15 which has six loop iterations executed on two cores using two threads where A(i) represent the `for` statement and the B(i) represent the computed statements. The first thread executes the iterations from 1-3 and the second those from 4-6. The loop carried dependency for the value of i can be ignored by making the i value in the second thread equal to $N/2$. In a system that has the number of threads equal to the number of available cores, let Li represent latency to execute one loop iteration, n represent the number of loop iterations and m the number of available threads, then the execution time for a sequential program will be equal to $Li * n$. If the loop is parallelized with m concurrent threads and the $m > n$ then the execution time will be Li, whereas alternatively if $m \leq n$ then the execution time will be (Jeon, 2009)

$$Li * \left\lceil \frac{n}{m} \right\rceil^{1} \tag{3.4}$$

Cyclic Multi-Threading (CMT)

DOACROSS is the most popular CMT transformation which allows cross-thread dependence to exist and can be used to extract parallelism from general purpose applications with a complex dependence pattern. This technique works similar to DOALL, by giving each iteration to a thread and these threads are executed on multi-core in round-robin fashion. In contrast with the DOALL technique, however, there are data and control dependencies crossing loop iteration boundaries. Therefore, to get correct results the dependency should be respected and the synchronization should be in the right order so as to ensure that the following iteration receives the correct value (Bridges, 2008; Vachharajani, 2008). Unfortunately, this puts the inter-thread communication on the critical path and hence binds the success of this technique with the efficiency of the communication techniques. With the increased number of cores integrated on the same die, the communication latencies between them become more significant, with their values ranging from a few tens of cycles to even a few hundreds. As a result, this leads to reduced performance of CMT, for unlike with IMT the increased number of threads in the system does not always lead to a linear increase in performance. Figure 3-18 shows an example of this technique. DOACROSS schedules each loop iteration on an alternate thread and communicates the dependency from thread to thread in a cyclic fashion due to the pointer chasing in statement A. All odd iterations of the loop are executed by the first thread and even ones by the second (Rangan et al., 2008; Adve and Boehm, 2010). The total time to execute the DOACROSS parallelized loop equals:

$$L_{par} = \frac{n}{m} * (L_i + SC) \tag{3.5}$$

where n represents the number of loop iterations, m the number of available threads, $iter1, iter2, iter3, iter4$ represent loop iterations, L_i the execution time for one loop iteration and SC the stall cycle. Figure 3-19 shows the synchronization and associated stalls that happen during the execution of the DOACROSS technique. C represents the consumer point and P the producer point, whilst t2 is the execution time between a consumer point and producer point in the same thread. Depending on this figure the SC between $iter1(i)$ and $iter4(i+m)$ is equal to

$$(t2_1 + CL_1) + (t2_2 + CL_2) + (t2_3 + CL_3) \tag{3.6}$$

or

$$m * (t2 + CL) \tag{3.7}$$

where CL is the communication latency between two threads. Finally, the speed up that will be gained from this technique is:

$$Speedup = \frac{n * L_i}{L_{par}} \tag{3.8}$$

[1] thanks for Dr Des Watson for pointing out the need for $\lceil \; \rceil$

```
while( ptr = ptr->next)//A(i)
  ptr->data += 1;  //B(i)
```

Figure 3-17: DOACROSS example code. Adopted from (Raman, 2009)

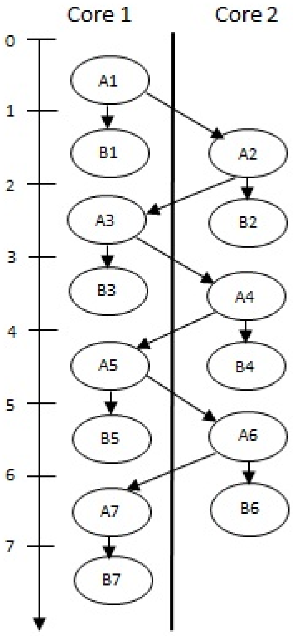


Figure 3-18: CMT Execution example. Adopted from (Raman, 2009)

Pipeline Multi-Threading(PMT)

PMT is another technique for parallelizing loops with inter-thread iteration dependencies, which does not need to separate the critical path of a loop across multiple threads and hence guarantees that this path will be local to one thread. The most popular transformation of PMT is decoupled software pipelining (DSWP) which was developed from the earlier DOPIPE[2] (Davies, 1981) technique.

DSWP approaches the problem differently from the above techniques, for instead of partitioning a loop by placing distinct iterations in different threads, it partitions the loop body for pipeline execution. More specifically, this technique works by automatically extracting a long thread from the loop body and partitioning this into stages that work in a pipeline form and subsequently, communicating the dependency between threads using inter core communication. By exploiting fine grain parallelism that exists in most applications, DSWP can improve program performance. There are three main points of difference between the traditional software pipeline (SWP) and DSWP, these being (Bridges, 2008; Li et al., 2011):

- DSWP depends on thread level parallelism while SWP depends on instruction level.

[2] starts working with the DOACROSS to parallelize the scientific code that has loop carried-dependency which is different than the DSWP that has the ability to parallize the general purpose program. DOPIPE can partition the loop body among multiple threads, however, this number of threads is determined at the compilation time. Moreover, DOPIPE works in a way that the dependency between loop iterations can form a cycle (Mason et al., 2009).

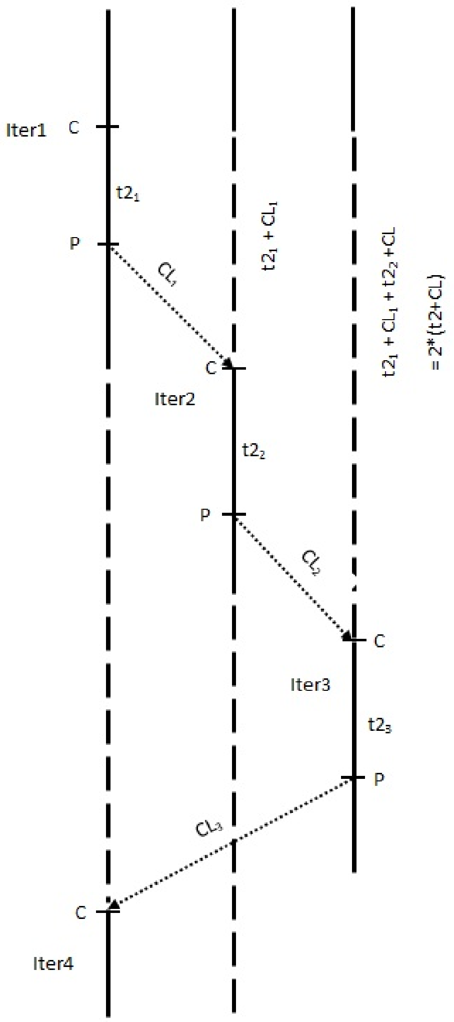


Figure 3-19: This figure shows the synchronization and associated stalls in DOACROSS. Adapted from (Raman, 2009)

- DSWP parallelizes loops with complicated control flow by partitioning the loop body into stages that have independent control flow and gives each stage to thread.
- DSWP stages are decoupled and communicate using a software implemented queue so a stall in one stage does not affect another.

Figure 3-20 shows how DSWP parallelizes the loop in figure 3-17 (Bridges, 2008; Li et al., 2011). A brief description of how this technique works is as follows. First, the PDG is constructed for the candidate loop and is used to identify the strongly connected components (SCCs). Subsequently it is transformed into another graph called the Direct Acyclic Graph (DAG), where each SCC is represented by one node and the nodes in the DAG are allocated to threads, which then communicate the dependencies between threads.

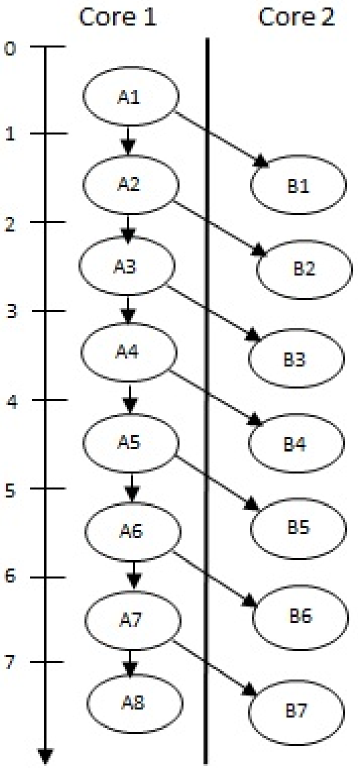


Figure 3-20: The execution schedule of the loop in figure 3-17 parallelized by DSWP. Adopted from (Raman, 2009)

To calculate the execution time for the parallelizing loop in DSWP techniques, let $C_1, C_2, \ldots, C_k$ represent the number of the SCCs and $L(C1)$ the latency of execution time for a set of operations inside each SCC. In addition, it is assumed there is no variation in the execution time latencies across loop iterations. Also, let $S_1, S_2, \ldots, S_m$ represent the number of dependencies inside each thread that need to synchronised and communicated and m be the number of available threads. $C_{1,j}, C_{2,j}, \ldots$ represent the number of SCCs inside thread j. The execution time for each j^{th} thread will be equal to:

$$L(T_j) = L(C_{1,j} \cup C_{2,j} \cup ..C_{2,j} \cup S_j) \tag{3.9}$$

The overall of execution time of the parallelizing loop in this technique equals that of the thread that takes the longest.

$$L_{par} = Max_j(L(T_j) \tag{3.10}$$

$$= Max_j(L(C_{1,j} \cup C_{2,j} \cup \ldots C_{2,j} \cup S_j)) \tag{3.11}$$

and the speedup that is gained from this method is equal to:

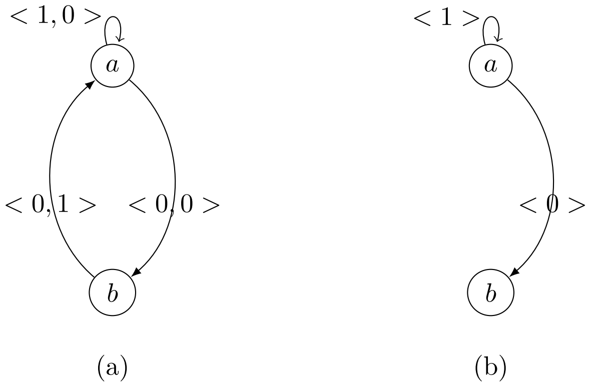
```
L1: for(i1 =0;i1<N1;i++){
L2: for(i2=0;i2<N2;i2++)
     {
      a:U[i1+1][i2]=V[i1][i2]+U[i1][i2];
      b:V[i1][i2+1]=U[i1+1][i2];
    }
   }
```

Figure 3-21: Sequential version of program. Adapted from (Rong et al., 2007)

$$speedup = \frac{L_{seq}}{Max_j(L(C_{1,j} \cup C_{2,j} \cup \ldots C_{2,j} \cup S_j))} \tag{3.12}$$

3.2.4 Usage of DSWP

Rong et al. (2007) proposed a method for constructing a software pipeline to extract instruction level parallelism (ILP) from an arbitrarily deep loop nest, where the traditional software pipeline is applied to the inner most loop or from the inner most to the outer loops. Figure 3-21 shows the sequential version of the program. This approach is called the single dimensional software pipeline (SSP), so named because it came from the conversion of a multi-dimensional data dependency graph (DDG) to a 1-D DDG and consists of three steps.



(a) (b)

Figure 3-22: Converting a multi-dimension data dependency graph (DDG) to a single dimension (DDG). Adapted from (Rong et al., 2007)

1- Loop Selection:

Every loop level is inspected and the most profitable one is selected to apply the software pipeline schedule, which works by overlapping successive iterations of a loop. Two criteria can be used to determine which loop is more profitable to the software pipeline schedule, these being the initiation rate, which determines the number of iteration points per cycle and data reuse, which is the average number of memory access per iteration point.

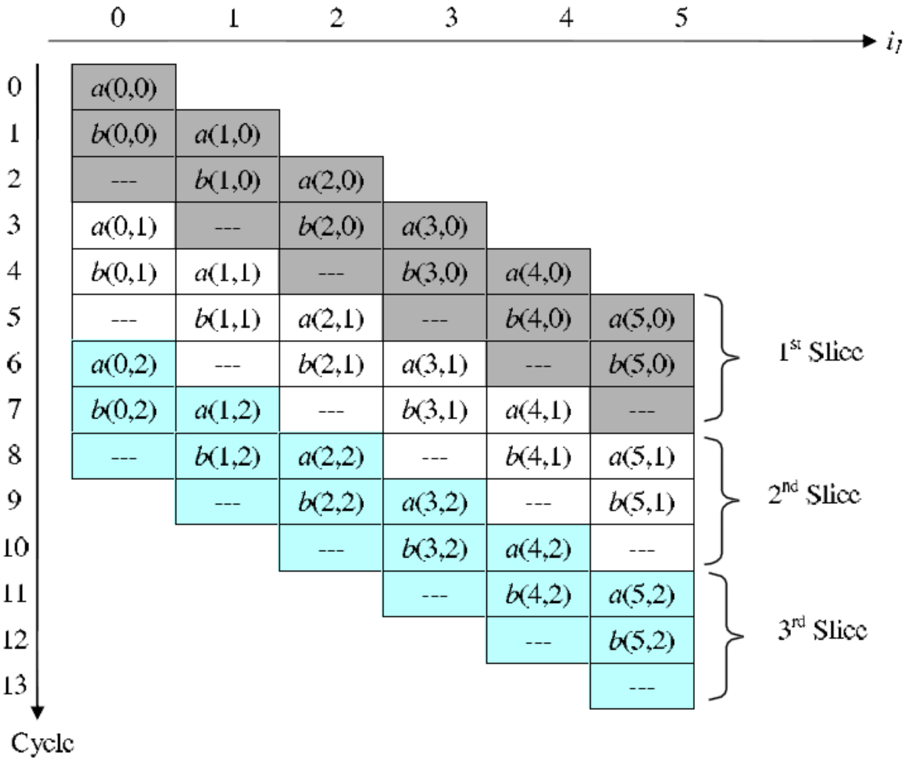
2- Dependency Simplification:

Involves simplifying the dependency for the selected loop L_x from the multi-dimension data dependency graph (DDG) into a single dimension. There a two legal types of dependency that can exist, which are represented by the distance vector $\langle d_1, d_2 \ldots d_n \rangle$. The first is positive dependency, which is the dependency across two slices and its vector is $\langle d_1, d_2 \ldots d_n \rangle$, where $d_1 \geq 0$, and $\langle d_2, \ldots d_n \rangle$ is a positive vector. This dependency is already respected, because all the slices execute sequentially. The second type is zero dependency, which happens inside each slice and is the only one that needs to be considered if a software pipeline is used to execute the slices dependency vector $\langle d_1, d_2 \ldots d_n \rangle$, where $d_1 \geq 0$, and $\langle d_2, \ldots d_n \rangle$ is a zero vector. By removing the positive dependency that exists between slices the multi-dimension data dependency graph (DDG) is simplified into a single dimension as illustrated in figure 3-22(a) and 3-22(b). Another type of dependency is negative dependency which is illegal, which happens between the current and previous slices and if it occurs needs to be converted to zero or positive dependency using loop skewing.

3- Final Schedule Computation:

After obtaining the simplified DDG, the iteration point in a loop nest is allocated to the slice, that is, for any $i1 \in [0, N1]$, iteration point $(i1, 0, .., 0, 0)$ is assigned to the first slice, $(i1, 0, .., 0, 1)$ to the second, and so on. All $L1$ iterations can be executed in parallel, if there are no dependences between them and no limits on the resources. However, if there is dependency between the iterations, they will be executed using the software pipeline method, as illustrated in figure 3-23. To address resource limitations, the set of slices are split into groups and pushed down to the next group until some resources are free.

Rangan et al. (2004) introduced a new technique to utilize a decoupled software pipeline for optimizing the performance of recursive data structures (RDS)(e.g., linked lists, trees and graphs). For this kind of structure difficulties have been encountered when trying to execute it in parallel, because the instructions of a given iteration of a loop depend on the pointer value that is loaded from a previous iteration. Therefore to address this problem, a decoupled software pipeline has been used so as to avoid stalls that are happening with the long variable-latency instruction in RDS loops. It is clear that RDS loops consist of two parts. The first contains the traversal code (critical path of execution) and the second represents the computation that should be carried out on each node traversed by the first part. By determining which program part is responsible for the traversal of the recursive data structure, the backward slice for this part can be identified and then decoupled software pipeline techniques can be used to parallelized these parts. The first part will be given to one thread and the second part to another. As the data dependency between these parts is unidirectional (the computation chain in the first part depends on the traversing chain in the second, but not vice-versa), the producer



Cycle	0	1	2	3	4	5
0	a(0,0)					
1	b(0,0)	a(1,0)				
2	---	b(1,0)	a(2,0)			
3	a(0,1)	---	b(2,0)	a(3,0)		
4	b(0,1)	a(1,1)	---	b(3,0)	a(4,0)	
5	---	b(1,1)	a(2,1)	---	b(4,0)	a(5,0)
6	a(0,2)	---	b(2,1)	a(3,1)	---	b(5,0)
7	b(0,2)	a(1,2)	---	b(3,1)	a(4,1)	---
8	---	b(1,2)	a(2,2)	---	b(4,1)	a(5,1)
9		---	b(2,2)	a(3,2)	---	b(5,1)
10			---	b(3,2)	a(4,2)	---
11				---	b(4,2)	a(5,2)
12					---	b(5,2)
13						---

i_1
1st Slice
2nd Slice
3rd Slice

Figure 3-23: the final schedule of cutting and pushing down the slice. Adapted from (Rong et al., 2007)

instruction is inserted in the first part and the consumer one in the second. Figure 3-24 clarifies how DSWP techniques split the RDS body into two parts, thus ensuring that the dependence recurrences in a loop remain local to a thread. The primary experiments to parallelize DSWP on the Pentium 4 Xeon processor have illustrated that the operating system and spinlock synchronization overhead can negate any speed up gained from applying the pipelining. This is because the trap and return time that these synchronization methods use is large. Moreover, extra time is needed for thread communication through shared memory that will cause an increase the demand for the shared memory port and cache pollution. Rangan et al. (2004) have put forward a hardware synchronization array (SA). This is a non-memory-based structure consisting of a set of blocking queues being used for communication between various hardware threads on CMP and SMT machines. These queues are accessed by producer and consumer instructions, with the former taking two operands, the immediate dependence number and register, in order to get access to the SA which gives the same dependent array entries to the dependent consumer and producer instructions. Consequently, the value in the register will be sent to a virtual queue defined by the dependence number and the same thing will be happen with the return value by a consumer instruction. Figure 3-25 illustrates the structure of a synchronization array (Rangan et al., 2004). When the producer executes, it first checks the

```
while( ptr = ptr->next){
ptr->data += 1;
}
```

a) Recursive Data Structure loop

```
while( ptr = ptr->next){
  Producer(ptr)
 }
```

b) Traversal Loop

```
while( ptr = consumer()){
  ptr->data += 1;
 }
```

c) Computation Loop

Figure 3-24: Splitting RDS Loops. Adopted from (Rangan et al., 2004)

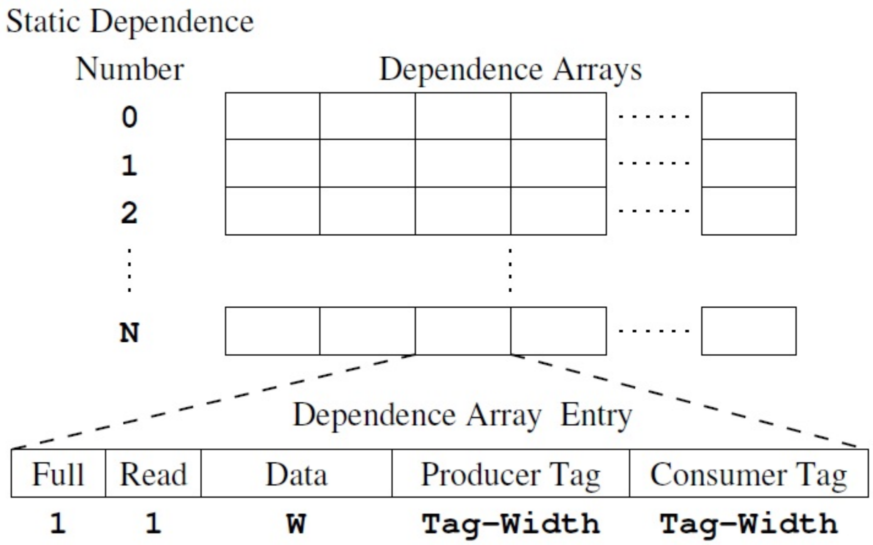


Figure 3-25: Synchronization Array structure Adapted from (Rangan et al., 2004)

SA flag to see if the data field is empty (Full flag equals zero) before writing its data. If it is so, it writes its value, write its tag to the producer tag field and changes the SA flag (setting it to one). The same thing happens for the consumer, whereby if it starts to execute it checks the full flag to know if there is any data ready to consume and if the flag value equals 1 then it starts reading this value and change architecture state (dependence array fields). When there are no data in the data field the SA will keep remembering the consumer tag until it finds the corresponding producer that will give it the producer value. For each unique core that

```
DSWP(Loop L)

$G \longleftarrow$Build-Dependency-Graph(L)
$SCC \longleftarrow$find-Strong-Connected-Component(G)
If| $SCCs$ |=1 then return
$DAG_{scc} \longleftarrow$Coalesce-SCCs(G,SCCs)
$P \longleftarrow$TPP-algorithm($DAG_{scc}$,L)
If | $P$ |=1 then return
Split-Code-into-loops(L,P)
Insert-necessary-flow(L,P)
```

Figure 3-26: DSWP algorithm. Adapted from (Ottoni et al., 2005)

produces data and the one that consumes them there is a given dependency number associated with it. This particular dependency number that is allocated to the consumer and producer pair of cores can be changed only after all the producer and consumer instructions that have been issued are retired. Moreover, the array dependency in this SA architecture is designed to support multiple dependencies between two threads.

Whilst the Rangan technique focuses on the RDS code, Ottoni et al. (2005) have presented a fully automatic DSWP approach for generic program loops. Instead of identifying the critical path in loop bodies, this automatic method tries to identify more generic program recurrence and to achieve acyclic dependency among threads. That is, by finding out the hidden fine grain parallelism in most applications, DSWP can extract long-running concurrently executing threads and figure 3-26 shows the DSWP algorithm steps. The input to this algorithm is the loop's dependency graph, which contains the different types of dependency that are going to exist in the loop, like: memory, register and control dependencies.

Two types of threads are created by this algorithm, the main one and the auxiliary ones, which are spawned by the former at the beginning of the program. For an optimized loop, the compiler creates a function that has code that will be executed by the auxiliary thread and passes its address to it. To communicate values between cores a simple message passing mechanism has been used, which is similar to that used in a scalar operand network. This involves communicating one word per message and implementing this through software. Ottoni et al. used a shared queue for two types of instructions: producer and consumer. The former were used to send data and the latter for receiving them. In addition, they avoided a synchronization overhead by making these two instructions block only when adding value to the full queue and deleting value from the empty one (Ottoni et al., 2005). The experiments showed the ability of this method to handle all kinds of dependencies and also its efficiency of exploiting the the pipeline parallelism found in ordinary, general-purpose applications. However, one of the main problems that limits its performance is the slower stage DSWP.

Cluster Decoupled Software Pipelining (CDSWP) was introduced by Zhang et al. (2008) as an extension to the conventional DSWP. It works almost exactly like DSWP, however, it communicates a *set* of dependent data rather than a single item. Thus, after the DSWP partition loop body to n stages, the second stage cannot start its execution until the first stage has finished writing all data dependencies into the queue buffer. Then, after the first stage has collected all of its dependent data in one buffer, the data in this buffer are communicated as a single unit. This procedure is repeated throughout all the DSWP stages. Clustering of dependent date helps reduce the number of communications and as a result leads to a reduction in the average overhead for each iteration. False sharing, which happens when two threads try to access the same cache line at the same time, and cache latency, are two factors that will affect the performance of DSWP, if inter-thread synchronization is implemented in the software. Zhang et al. eliminated the former by controlling Cluster Data Size(CDS), such that both the producer and the consumer work on different cache lines. Also, this can reduce the number of times the cache line is accessed by each stage and as a result the cache latency overhead is reduced. While choosing the right size can improve the performance, however, the big cluster size makes the extracted thread execute sequentially and also reduces the benefit from the parallelism. Figure 3-27 illustrates their proposed method. CDSWP partitions the loop body into three threads and the data dependency is divided into two data-sets. The first thread executes loop iteration `k` times and then the dependency data set is written to the queue buffer. The same thing is repeated for the rest of the iterations. The producer will execute spin-wait if there is not enough space to add data and the consumer also will execute this when there is no data in the buffer. Moreover, balanced threads sizes have a big impact on CDSWP performance where unbalanced threads size can reduce the benefit of parallelization. Vachharajani et al. (2007) proposed a method to improve the performance of DSWP by using speculation, which they added to break the rare dependencies. The loop body will be transformed to pipeline threads, where each thread has a set of loop body instructions. Certain dependencies should be evaluated speculatively to make sure that there is no dependency between the later and earlier iterations. The transformation method has several steps:

1- Build the PDG for the chosen loop.

2- Choose the suitable dependency edge for speculation.

3- Remove the selected edge from the PDG.

4- Transform the loop using DSWP.

5- The code that is responsible for detecting misspeculation[3]has to be inserted.

6- The misspeculation recovery code needs to be inserted.

It is started by temporarily speculating all the highly anticipated dependencies that will exist in the loop body. Then after a heuristic partition of these instructions among threads,

[3] Sic. meaning an incorrect case of speculation

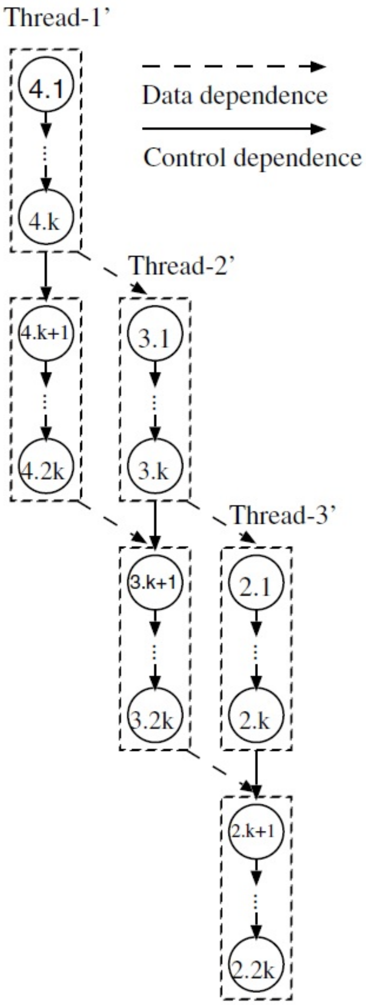
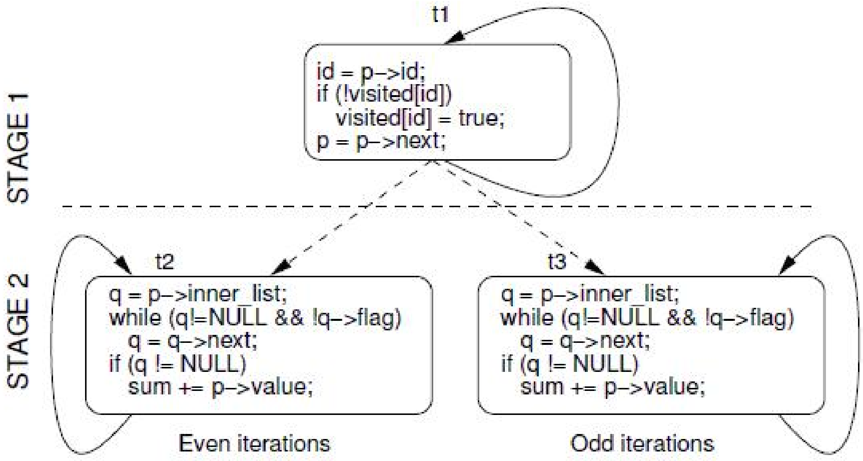


Figure 3-27: Parallel-stage DSWP Adapted from (Raman et al., 2008)

the dependencies that are thread local, are removed from the speculation set and the only ones that are kept are cross-thread dependencies, especially those that happen between the later and earlier threads. These cross-thread dependencies are the only ones that need to speculate to guarantee that there are no cyclic communications between threads. Threads created by SpecDSWP will check the architecture state and when misspeculation is discovered several actions need to be taken. First, the recovery waits until the threads that started before misspeclation finish their work and second the speculative state needs to be removed and exchanged with a non speculative one. Third the misspeculation thread needs to be re-executed and finally, after that the speculative execution can start. Although the performance of SpecDSWP without misspeculation will be equal to that of DSWP, it will be degraded in the presence of poor partitioning, a large amount of communication and a high rate of misspeculation.



```
id = p->id;
if (!visited[id])
   visited[id] = true;
p = p->next;
```

```
q = p->inner_list;
while (q!=NULL && !q->flag)
   q = q->next;
if (q != NULL)
   sum += p->value;
```

```
q = p->inner_list;
while (q!=NULL && !q->flag)
   q = q->next;
if (q != NULL)
   sum += p->value;
```

Figure 3-28: Parallel-stage DSWP . Adapted from (Raman et al., 2008)

Raman et al. (2008) introduced a new technique called the parallel stage decoupled software pipeline (PS-DSWP), which is a combined of DSWP and DOALL. The reason for this combination is that the slowest stage of DSWP limits the speed of this technique and this method exploits the ability to execute some stages of DSWP using DOALL as illustrated in figure 3-28. That is, PS-DSWP enables such a concurrent execution by replicating the strong connected components (SCCs) that form the entire inner loop which has high iteration counts and hence, can take a long time to execute. That is, multiple threads concurrently execute the SCCs that represent this code and replicating this stage leads to having many copies of the same code being executed for different data. However, this replication is only possible if there is no dependency between the inner and outer loops. Figure 3-29 shows the execution schedule of the loop in figure 3-28 (Raman et al., 2008). For performance reasons, Raman et.al. used dedicated inter-core communication hardware (Synchronization Array) to communicate between threads. A special queue-based register is added to each processor core so as to get the physical queue number, by adding the value in this register to the virtual queue number in the producer/consumer instructions.

In the parallel stage of DSWP, the first thread has the odd iterations while the second has the even ones. This way of distributing loop iterations makes the ownership of the cache line move between the two threads and as a result this could cause the false sharing problem, which can negate any benefit from the PS-DSWP. The false sharing problem happens when two processors want to write to a different location on the same cache line and this can increase the latency to access to the same cache line. For more clarification consider the example in

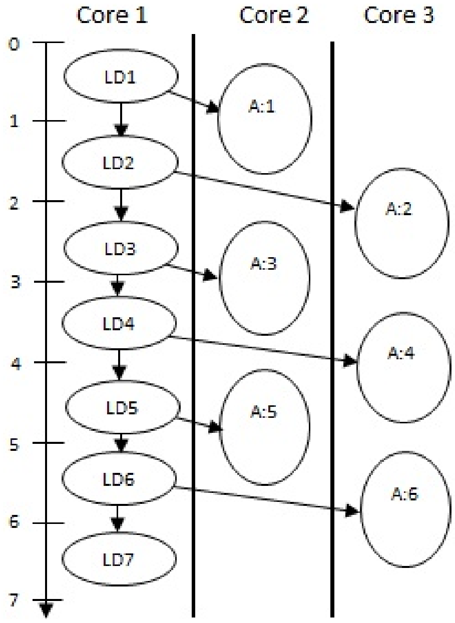


Figure 3-29: Parallel-stage execution schedule of the loop in figure 3-28. Adapted from (Raman et al., 2008)

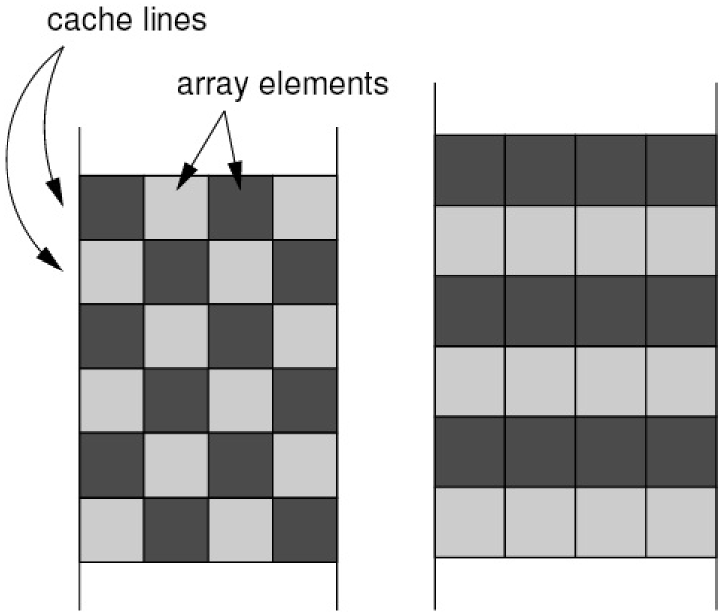
figure 3-30(a). The parallel stage represents the two statements `A` and `B`, while the sequential one represents statement `C`. Figure 3-30(b) illustrates how the cache will look after it is accessed by two threads. The shaded square represents the cell that has been written by the first thread (has odd iterations) `a[1]`, `a[2]`, `a[5]`, while the light one represents the second thread (even iterations) `a[2]`, `a[4]`, `a[6]`, where each cell represents array elements. To solve this problem, instead of distributing loop iterations as odd and even and making threads work in a round-robin fashion, a set or chunk of continuous iterations will be given to each thread to make them write in a different cache line. Figure 3-30(c) illustrates how the cache is written after chunking the loop iterations. Simple heuristics were used by Raman et al. to calculate the chunk size.

$$chunksize(cs) = \frac{cls}{stride * size}$$

cls represents the cache line size, *stride* the induction variable stride and finally *size* is the variable size. The original induction variable without chunking will take the value $i_0, i_{0+1}, i_{0+2}, \cdots i_{0+k}$ and after adding this chunk with size equals 2, the increment will be $i_0, i_{0+1}, i_{0+4}, i_{0+5}$ for the first thread and $i_2, i_{2+1}, i_{2+4}, i_{2+5}$ for second one, but the chunk size can reduce the opportunity to extract the parallelism.

```
   int a[N], b[N], c[N];
A: for(i =0;i<N;i++){
B: a[i] = a[i]*b[i];
C: c[i] = c[i-1]+a[i];
   }
```

(a) loop affected by false sharing



(b) Without chunking

(c) With chunking

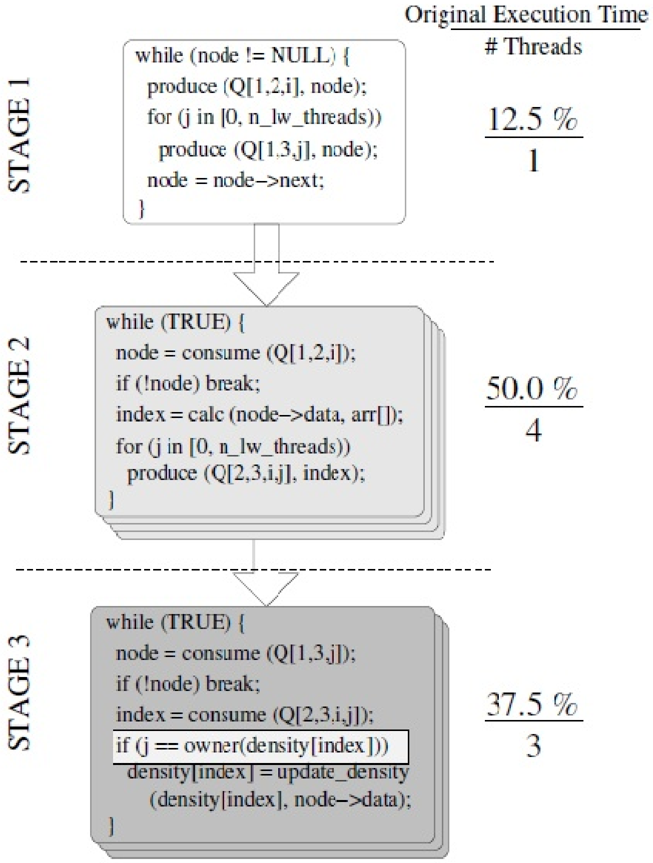
Figure 3-30: False sharing adapted from (Raman et al., 2008)

While Raman et al.'s technique focuses on parallelizing one stage of DSWP, Huang et al. (2010) introduced a manual parallelization technique that can parallelize more than one stage. The numbers and the size of the SCCs can degrade the performance of DSWP. As a consequence the DSWP overall speed up depends on its slowest stage which is equal to:

$$Speedup = \frac{1}{T_{SlowestStage}}$$

The technique called DSWP+ and can improve the performance of DSWP if it is employed along with another techniques such as: DOALL, LOCALWRITE and SpecDOALL[4]. That is DSWP+ divides the loop body into stages, for the ones that take a long time hence limiting the performance of DSWP and so, needs to be parallelised using other techniques.

[4] Speculative DOALL



```
while (node != NULL) {
  produce (Q[1,2,i], node);
  for (j in [0, n_lw_threads))
    produce (Q[1,3,j], node);
  node = node->next;
}
```

```
while (TRUE) {
  node = consume (Q[1,2,i]);
  if (!node) break;
  index = calc (node->data, arr[]);
  for (j in [0, n_lw_threads))
    produce (Q[2,3,i,j], index);
}
```

```
while (TRUE) {
  node = consume (Q[1,3,j]);
  if (!node) break;
  index = consume (Q[2,3,i,j]);
  if (j == owner(density[index]))
    density[index] = update_density
      (density[index], node->data);
}
```

Figure 3-31: DSWP+DOALL applied to the second stage and SpecDOALL applied to the third stage Adapted from (Huang et al., 2010)

The first combination, DSWP+DOALL, works similarly to Raman et al.'s method, whereas the second involves partitioning an array into blocks and giving each one ownership of a thread. Consequently, only the thread that has ownership of a particular block has the ability to change the array value. In addition, LOCALWRITE scales linearly with the number of additional threads if the array distribution is uniform. The third technique combines DSWP with speculative DOALL, which speculatively removes the rare dependencies from loop iterations that make DOALL inapplicable. Their results showed that DSWP+ provides more speedup than DSWP and other parallelizing techniques, if they are used alone. Figure 3-31 shows the code transformation for DSWP+ with DOALL being applied to the second stage and SpecDOALL to the third. Four threads are assigned to the former stage and three to the latter. So the speed up for the second stage will be equal to the stage time divided by the number of threads allocated to this stage = $\frac{50.0}{4}$ (Huang et al., 2010). The key limitation with LOCALWRITE is that, it is a parallelizing method for irregular reduction where the elements that need to be reduced are not located in consecutive order and also its scalability depends on the distributed array index across the array block. Moreover, DSWP+SpecDOALL performs well when the probability of dependency among loop iterations is significantly smaller than 1, whereby the excessive miss-speculation can negate any benefit that can be gained from the speculation. Software implementation has been used to communicate the data and control dependency between threads, with the cache-aware, concurrent lock-free buffer algorithm being used to program the producer and consumer primitives (Huang et al., 2010).

3.3 Summary

This chapter has presented an overview of compiler-based automatic parallelization techniques. Essential information and concepts that are employed in the proposed method have been introduced. The key points that have been focused on are, first static backward slicing because it preserves the original program behaviour of any input and it generates an executable slice. Second DSWP because of its has ability to achieve broadly applicable parallelizations that are not limited by communication latency. That is a parallelization paradigm is needed that does not spread the critical path of a loop across multiple threads and so this has been introduced to allow cross-thread dependences while ensuring that the dependences between threads flow in only one direction. This ensures that the dependence recurrences in a loop remain local to a thread, and also that communication latency does not become a bottleneck.

The structure of this chapter has been divided into three sections. The first section covered some concepts and definitions relating to different slicing methods. Then, the techniques of interprocedural and intraprocedural slicing were discussed. Moreover, two methods to compute slices depending on the CFG and PDG were presented. And finally, several forms of parallel slicing techniques, their execution and applications were explained. In the second section, several techniques in relation to loop level parallelism were presented and their limitations discussed. In addition, some concepts and ideas regarding DOALL, DOACROSS and DSWP were covered as well as prior research on how to perform parallel execution in the latter most being reviewed. In the next chapter there is an investigation into how the combination of these two techniques (slicing and DSWP) can give better performance than using each of them on their own.

Chapter 4

Implementation of DSWP and Slicing Techniques

This chapter presents the DSWP/Slice technique. First, a simple illustrative code example to demonstrate the principle is presented and later on a more complicated and realistic case is used to describe the proposed method.

Previous work that has been carried out by Fuyao Zhao (2011) to implement DSWP in LLVM has been adapted to make it suitable for the DSWP/Slice technique, but it has only been demonstrated on a very few, simple, artificial programs with a synchronization achieved by pthread_mutex. In contrast, our work extends and adapts the approach, uses low overhead lock-free buffers, and demonstrates its effectiveness using real-world programs The chosen programs are executed sequentially until they reach the part that is suitable for applying the DSWP/Slice technique, when the parallel execution is begun.

This chapter consists of four sections. The second section introduces the DSWP/Slice algorithm and consist of three subsections which are determining a thread assignment, slice extraction and code generation. The third section describes the compiler implementation of the proposed method and some supported libraries, and finally, the last section provides a summary of the work presented in this chapter.

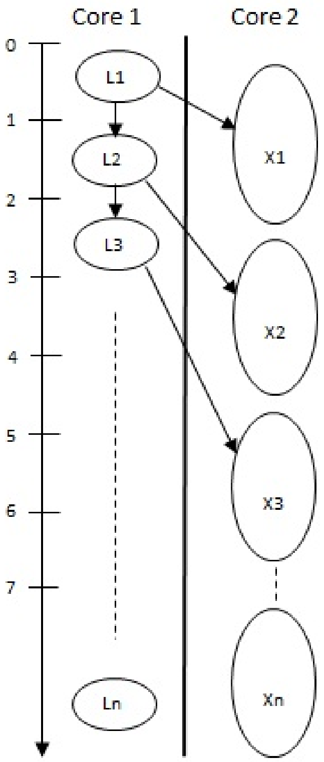
4.1 Motivation

To illustrate the proposed method, some assumptions need to be introduced:

- The execution time of each pipeline stage remains unchanged across different iterations of the loop.
- Most of the execution time of the loop goes on the function body called inside the loop body and this function body is not recursive.

```
Work(cur)
  {
   Statements
  }
main()
.....
List *cur = head;
for (; cur != NULL;
   cur = cur->next;) \\L
    Work(cur);       \\X
```

(a) Simple Code



(b)Two stages DSWP

Figure 4-1: Conventional DSWP. Adapted from (Raman, 2009)

- For simplicity, it is assumed that the loop to which DSWP is applied has only one back-edge and a single header (no jump statement from the middle of the loop to the loop header or from anywhere in the program to the loop).
- Also the partition problem is simplified by allowing only *one* of the pipeline stages to be assigned to more than one thread.
- We assume that the number of threads is known in advance.

Consider the example in Figure 4-1(a). Conventional DSWP partitions the loop body into the two parts labelled `L` and `X`, with the first part being given to the first thread and the second to the second thread.

As can be seen from figure 4-1(b), the implementation of DSWP in this way does not make any improvement to its performance since most of the execution time goes on in the function body. The second stage DSWP (second part) is much longer than the first and consequently this leads to an unbalanced load between these stages.

```
Work(cur)
   {
    var1; \\S1
    varn; \\Sn
   }
   List *cur = head;
for(;cur != NULL;
   cur = cur->next;)\\L
     Work(cur); \\X
```

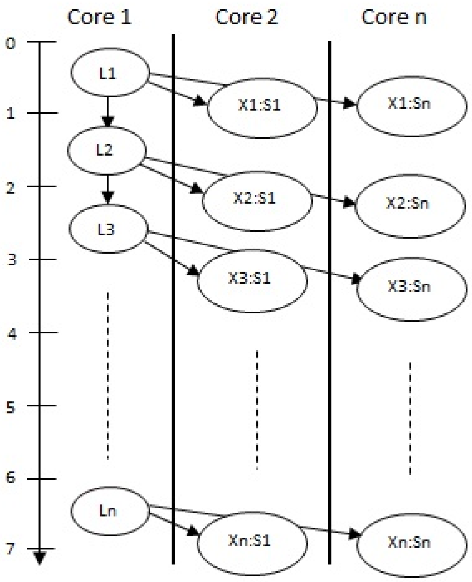


Figure 4-2: Application of slice technique to the long stage DSWP

The main feature of the proposed method is the application of backward slicing to the longer stage emerging from the DSWP transformation. This is particularly effective when the whole stage represents a function body. The reasons that the function body is chosen to apply the slicing technique are:

1- The whole function represents the indivisible SCC which is the bottleneck to improving DSWP performance and this is generally true when the execution time for this function is large.

2- The likelihood of getting more than one slice from the function body (not inlining) that comprimises the DSWP stage is higher than the possibility of getting a slice from the DSWP stage without function call. This is generally true in practice since all the SCCs that have allocated to the DSWP stage (without function call) are highly dependent on each other. In consequence the possibility to extract more than one slice is reduced.

That is, instead of giving the whole of stage X to one thread, it can be distributed across n threads, depending on the number of slices extracted, with in this case, one core running L (the first stage) and n more running $S_1, S_2, \ldots, S_n$ (the slices from the second stage).

By extracting several slices from the function body and distributing them among several threads a better load balancing between the DSWP stages can be achieved. For example if two slices from the long stage DSWP that represents function body are extracted, each of them has half the execution time of the original function body and means speeding up this stage approximately a factor of two, as illustrated in figure 4-2.

However, while there are potential gains from splitting the loop body into several concurrent threads, there is still the cost of synchronization and communication between threads to take into account. In general there is no hardware that can provide fast inter-core communication similar to the synchronization array that is proposed by (Ottoni et al., 2005), and using lock-based synchronization such as pethread-mutex-lock has a high overhead and can negate any benefit from DSWP. Moreover lock-based synchronizations come with several difficulties, such as deadlocks, livelocks and so on (Giacomoni et al., 2008). Instead a software technique called lock-free buffers (Giacomoni et al., 2008) is used to communicate these data and minimize the overheads.

The synchronization is implemented through the FastForward circular lock-free queue algorithm, which enhances the performance of Lamport's queue (Giacomoni et al., 2008) where it fails under many weaker consistency models that result in strong coupling of the control `(head/tail)` and data `buffer` in a single operation. The FastForward circular lock-free queue algorithm uses the entry buffer itself to determine the state of the circular queue where both control variables (i.e., head and tail) are thread local. Using this algorithm the overhead of accessing the control variables is reduced (Chen et al., 2010). As a result, the producer and consumer can access the queue concurrently, via the enqueue and dequeue operations, which makes it possible for them to operate independently as long as there is at least one data element in the queue. More details of the FastForward circular lock-free queue algorithm are provided in section 4.2.3.

4.2 DSWP/Slice combination technique

DSWP/Slice combination technique has three parts:

1- Determining a thread assignment

2- Extract slices

3- Code generation

which are now explained in detail.

4.2.1 Determining a Thread Assignment

This part has four steps which are based on the DSWP algorithm as proposed by Ottoni (Ottoni et al., 2005):

Below, these steps are explained in detail.

Identify the candidate loop for DSWP

This step looks for the most profitable loop to apply DSWP/Slice by accumulating some information about the program. That is, the heuristic computes the estimated cycles necessary

Algorithm 1 DSWP algorithm that is proposed by Ottoni (Ottoni et al., 2005)

```
Step 1: Identify the candidate loop for DSWP.
Step 2: Build the program dependency graph (PDG).
Step 3: Build the strongly connected components (SCCs) and the directed acyclic graph (DAG).
Step 4: Assign SCCs.
```

to execute all instructions in every loop in the program by considering the instruction latency (see appendix B). The chosen loop should have the following properties:

- The greatest execution time among the loops in the program
- This loop contains a call to a function which accounts for a significant proportion of the execution time. As explained earlier (see 4.1), the transformation is most beneficial when the body of a loop is a function because the likelihood of getting more than one slice from the function body (not inlining) that comprises the DSWP stage is higher than the possibility of getting a slice from the DSWP stage without function call.

Build the program dependency graph (PDG)

Building the PDG that contains several types of dependency is the second step of this technique. These types of dependencies that exist between instructions are collected in a PDG table with an entry for each instruction in the loop body. Then it associates with each entry a set of instructions that have dependency with it. The dependency arc in the PDG can represent either a data dependency, which covers both register and memory dependency or a control dependency. LLVM uses a `Use` class to collect the register dependency, while a `MemoryDependenceAnalysis` class that depends on an alias analysis class is employed to find memory dependency (if there is dependency between the Store and Load memory instructions as illustrated in subsection 2.6). For example, consider the code in figure 4-3, which traverses a linked list (statements 2, 5), with the data field of each node being used in the `Calc(P,m)` function together with the value of `m`. Statement (3) calls the function `fact(P)` that generates a factorial value for `m`. The Intermediate Representation (IR) and PDG for this code is given in figure 4-4, with the red, blue and black arrows representing the: control, register, and memory dependencies, respectively.

Build the strongly connected components (SCCs) and the directed acyclic graph (DAG)

In order to keep all the instructions involved in the dependency cycle thread-local, SCCs have to be constructed for them. Then, by merging each of these into a single node a Directed Acyclic Graph (DAG_{SCC}) can be obtained. The grey node in Figure 4-4 represents the SCC. After gathering all the nodes that participate in the dependency cycle into one SCC, a DAG_{SCC} is generated, as illustrated in figure 4-5.

```
                                     7   Cal (list P , int  m){
1   P = list->head;                  8    a[0]=0;
2   While (P ! = NULL){              9    for(i=1;i<P->data;i++){
3     m = fact(P);                  10       v= m +seq(i);
4     Cal(P,m)                      11       a[i]=a[i-1]+cos(v);
5     P = P->next;                  12       b[i]=b[i]+sin(v);
6   }                               13    }
                                    14   }
```

Figure 4-3: source code

Assign SCCs

The problem that arises with the assignment of SCCs is when their number is more than that of the available threads. In this case, it is necessary to merge several SCCs until there are as many as threads. Then, the thread partition method can be used to find equal weight threads (load balance) by distributing the SCCs between them.

Because the function call that represents one SCC is indivisible, a balance of threads cannot be achieved if the execution time of this function is much higher than the rest of the instructions in the loop body. By way of explanation, going back to the example in figure 4-3, if it is assumed that the statement 4 accounts for 75%, statement 3 accounts for 15% and statements 2 and 5 account for 10% of loop body execution times, giving the statements 2, 3, 5 to the first stage and statement 4 to the second stage will result in an imbalanced DSWP. Consequently, the second stage will be a critical one that negates any benefit from applying DSWP. The proposed solution to this problem is to apply the slicing technique to the long stage DSWP. However, with this solution two cases have to be considered:

1- The first case is when the function does not return value to the loop body. In this case there is no problem because this will give us a guarantee that the DSWP will work in one direction. The dependence arcs between the blocks in the partition do not form a cycle and this is similar to what is mentioned in (Raman et al., 2008) (property 1, condition 1) which states that

> For $i \neq j$, if there is some dependence from B_i to B_j, then there are no $B_{k1}, B_{k2} \cdots B_{kn}$, where $k1 = j$ and $kn = i$, such that there is some dependence arc from every B_{kl} to B_{kl+1}. In other words, the dependence arcs between the blocks in the partition do not form a cycle

   where B represents a basic block. Then the function body is assigned the second stage DSWP and under this circumstance all the instructions in the loop body are given to the first stage DSWP, except that which calls the function.

2- The second case is when the function returns a value to the loop body

```
  @p = alloca %struct.reco**
   m = alloca i32
bb3:

  %t3 = load %struct.reco** @p, a
  %t4 = call i32 @fact(%struct.re
  St1 store i32 %t4, i32* %m, ali
  %t6 = load %struct.reco** @p, a
  %t7 = load i32* %m, align 4
  call double @cal(%struct.reco*%
  %t9 = load %struct.reco** @p, a
  %t10 = getelementptr inbounds %
         %t9, i32 0, i32 2
  %t11 = load %struct.reco** %t10
   St2 store %struct.reco* %11, %
        reco** @p , align 4
bb4:

  %t0 = load %struct.reco** @p, a
  %t1 = icmp ne %struct.reco* %t0
  br i1 %t1, label %bb3, label %b

bb5:
 ret
```

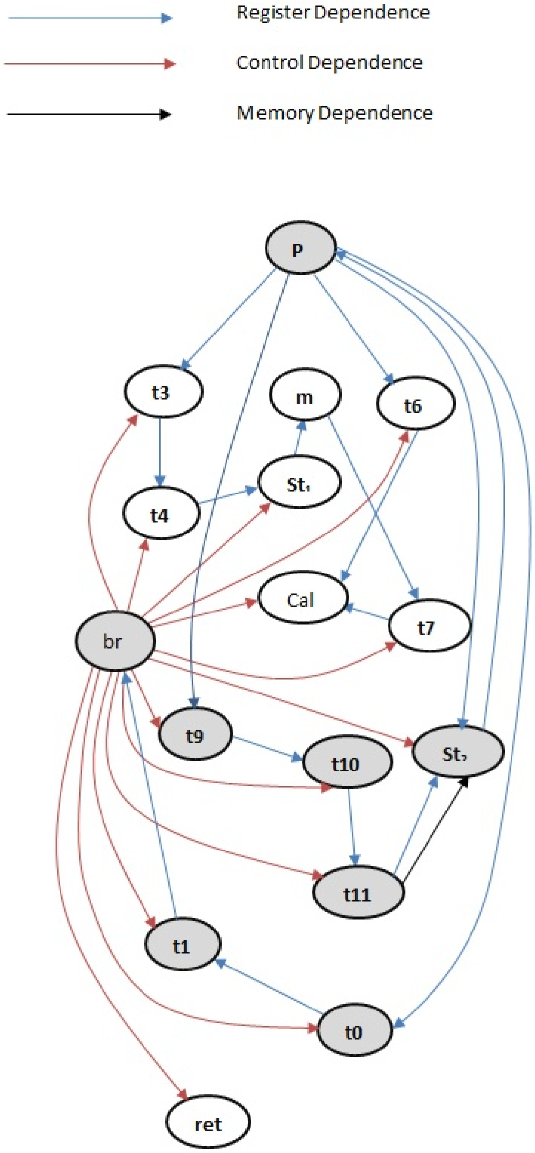


Figure 4-4: The IR and PDG of the source code in figure 4-3

In this case the loop is partitioned by giving the SCCs that represent the function call and all instructions that have dependency on it to the third stage DSWP. After partitioning is completed, the function body is sliced, with the allocated number of threads being equal to that of the extracted slice. So, in addition to the two threads that represent the first and second stages of DSWP, a number of threads equal to the number of extracted slices is needed.

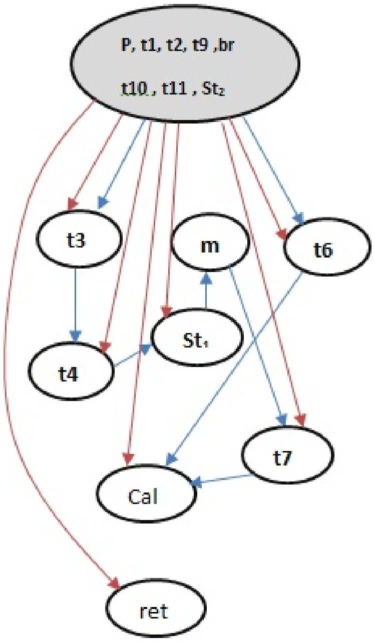


Figure 4-5: The DAG graph for SCCs

> Moreover, the second stage thread will have the instructions (or the strong connected component) that have dependency with the return value from the extracted slices (function body), which happens because DSWP works in one direction and thus avoids having a cyclic pattern. According to this partition, the first will represent the producer and the second a consumer, consuming the retrieved data from the first stage and also from the *n* slice, while the extracted slices work as consumer and producer at the same time. Figure 4-6 illustrates this.

Unlike the condition 2 that is mentioned in (Raman et al., 2008)(property 1, condition 2),which states that:

> For every (B_i, T_i), with $|T_i| > 1$, the DAG_{SCC} nodes in B_i must be **doall** nodes,
> and none of the dependence arcs among the nodes in Bi is loop-carried

where T represents a thread. Even in the presence of loop-carried dependency that makes another transformation for this stage inapplicable. Slicing techniques can still be applied to the function body and reduce it is execution time. However, this still depends on the availability of some factors as mentioned in section 5.3 which can determine the most suitable slice that can improve the long stage DSWP performance.

Relying on the above assumption that most of the loop body execution time is spent inside the function body and because the SCC that represents this body cannot be divided, the

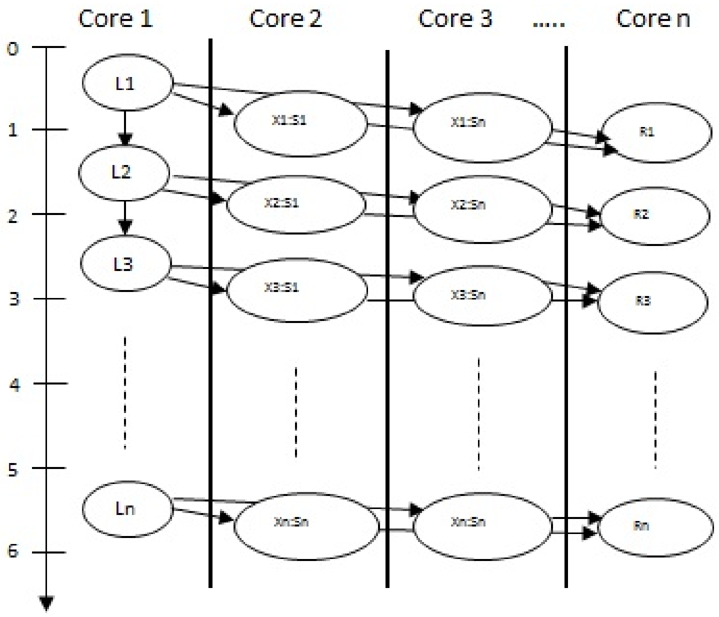


Figure 4-6: return value from the extract slice to the second stage

function call inside the loop body is treated by applying the slicing technique to the function body and extracting n slice executions in parallel. In the next subsection the slicing process is described in detail.

4.2.2 Extracting Slices

In this part, a small slicing program is designed that has the ability to extract slices for the limited range of the current case studies. To this end, the algorithm 2 that is illustrated below is used to compute an intra-procedural static slice, this algorithm was discovered independently, but is similar to the one in (Al Dallal, 2005).

N static slices from the function body are extracted as follows: in the first step, the PDG is built for the function body by drawing up the dependency table that has both control and data dependency (similar to the one above used to determine thread assignment). Secondly, the entry block for the function body is examined so as to identify the variables to be sliced and then the names of these are collected, being put on a slicing list. The aim is to extract a slice for every listed variable. Next, these variables are filtered, whereby those that are assigned a value once and loop induction variables, are dropped from the list. That because these variables are going to be included in more than one slice. Also, the function arguments allocation variables (`var_addr` where, `var` represents any function arguments) are dropped so as to reduce the range of the variables to be sliced.

Then, an attempt is made to isolate the control statement parts, such as loop or if state-

Algorithm 2 The ComputeAllSlice algorithm. A similar algorithm is in (Al Dallal, 2005)

```
Input: A PDG, node n, An empty set associated with each
       node to collect all data and control dependency.
Output: Extracted slice for a particular node $n$.

Algorithm:
  Make all PDG nodes as not visited.
  ComputeASlice(node $n$)
```

Procedure ComputeASlice(node n)

```
if the node is not visited then
    Mark node $n$ as visited
    Add the identifier of $n$ to the set associated with node $n$
    for each node $m$ on which node $n$ depends do
        ComputeASlice($m$)
        Add the content of the set associated with node $m$
            to the set associated with node $n$
    end for
end if
```

ments, into another table called the control table. After collecting the control part instructions, these are added to the extracted slice, if one of the slice instructions is contained in these control parts. To make this idea clear see the example in figure 4-7(a). After building the PDG for this function and before extracting slices for the two arrays `a[i]` and `b[i]`, all the statements that represent the control parts `for(i=0;i<p->data;i++)` and `if( v < 0.5 )` will be added in the control parts table. After extracting slice for each array variable `a[i]`, the control part statements for (`for` and `if` statement) will be added to slice if there is a control dependency between the `a[i]` variable and these control parts, as illustrated in figure 4-7(b).

For each filtered variable in the slicing variables list, first, an empty list is associated with it and subsequently, all the PDG table entries are scanned into it to find which one matches the slicing variable. If one is found, then all the instructions that have data or control dependency are added to the associated list. This procedure is repeated for all the instructions in the associated list and their operands and it finishes when all the instructions and their operands are contained in this list or all the variables that represent the loop induction variables have been reached. Figure 4-8 shows the entry block before and after the filtering process

Another difficulty is separating slices, particularly when two of them share the same loop induction variable. For example consider the the code in figure 4-7, the extract slice for the `a[i]` array will be included in the array `b[i]`, because the loop induction variable `i` is shared between these two arrays. To solve this problem the loop induction variables have to be isolated in a separate list and then the `computeASlice` procedure stops when its reaches these variables.

After a set of slices has been extracted from the function body, they are filtered to remove redundant ones so as to avoid repeated calculation, which will happen if all the instructions in one of them have been included in another. For example, if there are two slices and the first is completely contained in the second, which is longer than the first, then the former are

```
fun(int m ){
 ....
for(i=0;i<p->data;i++){
   v=m+seq(i);
   a[i]=a[i-1]+cos(v);
   b[i]=b[i]+sin(v);
   if( v < 0.5 )
     {
     dx=(val-(v*v)/2.0*v);
     a[i]=a[i]+dx;
    b[i]=b[i]+dx;
     }
 }}
```

(a)

```
fun( int m )
  ....
for(i=0;i<p->data;i++){
  v=m+seq(i);
  a[i]=a[i-1]+cos(v);

  if( v < 0.5 )
    {
     dx=(val-(v*v)/2.0*v);
     a[i]=a[i]+dx;

  }
}}
```

(b)

Figure 4-7: (a)The source code. (b) The extracted slice for `a[i]` array variable

removed and the latter kept. This procedure is repeated for all slices until the correct number is obtained. Figure 4-9 illustrates the extracted slices for the program source code in figure 4-1, where the function `seq` does not have any side effects. Also, figures 4-10 and 4-11 show the intermediate representation of these slices in LLVM.

4.2.3 Code Generation

In this part, the Multi-Threaded Code Generation (MTCG) algorithm, which has been introduced by (Ottoni et al., 2005) is used to generate threaded code for DSWP stages. The input to this algorithm is the PDG for the original loop body with its control flow graph (CFG). Furthermore, the set of the SCCs that represent the DSWP partitions and extracted slices also is given. The thread module in this work assumes that the execution of the program will be sequential until it reaches the loop that has been chosen to execute using DSWP. When execution reaches this point, auxiliary threads are spawned and each of them executes the CFG that has been created by the (MTCG) algorithm. Once these threads finish their work they are terminated and the sequential execution starts again. Because these auxiliary threads are created once, the cost of creating them will be reduced. The created threads wait until they are invoked to execute the code (function) that was allocated to them statically. An outline of the (MTCG) algorithm (algorithm 3) steps is provided below with their detailed description.

Before creating a function for each thread, the DSWP transformation inserts code in the main module (loop parent) to initialize each thread. In the loop header basic block the branch instruction will be reset to make it point to the new basic block that is responsible for creating these threads and giving each its job. This new basic block, called loop-replace, has the call site of the external function “sync_delegate” that is called to create threads. Figure 4-12 illustrates this loop-replace basic block.

```
define i32 @fun(i32 %d) nounwind
entry:
 %d_addr = alloca i32, align 4
 %retval = alloca i32
 %a1 = alloca [1000000 x i32]
 %b = alloca [1000000 x i32]
 %i = alloca i32
 %"alloca point" = bitcast i32 0 to i32
 store i32 %d, i32* %d_addr
 %0 = getelementptr inbounds[1000000 x i32]* %a1, i32 0, i32 0
 store i32 0, i32* %0, align 4
 %1 = getelementptr inbounds[1000000 x i32]* %b, i32 0, i32 0
 store i32 200, i32* %1, align 4
 store i32 1, i32* %i, align 4
 br label %bb1
```

(a)

```
define i32 @fun(i32 %d) nounwind
entry:

 %retval = alloca i32
 %a1 = alloca [1000000 x i32]
 %b = alloca [1000000 x i32]


 store i32 %d, i32* %d_addr
 %0 = getelementptr inbounds[1000000 x i32]* %a1, i32 0, i32 0
 store i32 0, i32* %0, align 4
 %1 = getelementptr inbounds[1000000 x i32]* %b, i32 0, i32 0
 store i32 200, i32* %1, align 4
 store i32 1, i32* %i, align 4
 br label %bb1
```

(b)

Figure 4-8: (a) Function entry block before filtering. (b) Function entry block after filtering

Creating Thread Basic Blocks

This step is the first step in the MTCG algorithm. After partitioning the loop body between DSWP stages, a new CFG_i, where i represents the partition number, for each partition P_i, is constructed. This CFG_i has a set of basic blocks, which are related to those in the original code (original CFG), whereby for each block B relevant to P_i a new basic block B́ is created in

```
Slice_1(p,m)
  {
   for(i=0;i<p->data;i++){
     v = m+seq(i);
     a[i]=a[i-1]+cos(v);
    }
  }
```

```
Slice_2(p,m)
 {
  for(i=0;i<p->data;i++){
    v=m+seq(i);
    b[i]=b[i]+sin(v);
   }
 }
```

Figure 4-9: The extracted slice after applying the slicing technique

```
%v = alloca i32
%a = alloca [1000 x double], align 8
%i = alloca i32
store %struct.reco* %d, %struct.reco** %d_addr
store i32 %m, i32* %m_addr
store i32 1, i32* %i, align 4
br label %bb1
%0 = load i32* %i, align 4
%1 = call i32 @seq(i32 %0) nounwind
%2 = load i32* %m_addr, align 4
%3 = add nsw i32 %1, %2
store i32 %3, i32* %v, align 4
%4 = load i32* %i, align 4
%5 = load i32* %i, align 4
%6 = sub nsw i32 %5, 1
%7 = getelementptr inbounds [1000 x double]* %a, i32 0, i32 %6
%8 = load double* %7, align 4
%9 = load i32* %v, align 4
%10 = sitofp i32 %9 to double
%11 = call double @sin(double %10) nounwind readonly
%12 = fadd double %8, %11
%13 = getelementptr inbounds [1000 x double]* %a, i32 0, i32 %4
store double %12, double* %13, align 4
%25 = load %struct.reco** %d_addr, align 4
%26 = getelementptr inbounds %struct.reco* %25, i32 0, i32 0
%27 = load i32* %26, align 4
%28 = load i32* %i, align 4
%29 = icmp sgt i32 %27, %28
 br i1 %29, label %bb, label %bb2
 br label %return
```

Figure 4-10: LLVM representation of Slice 1 in figure 4-9

CFG_i. Each basic block that originally belongs to the original CFG and also belongs to the P_i partition is called a *relevant* basic block. MTCG adds arcs to connect these basic blocks in the new CFG_i, which are inserted in order to guarantee equivalence between the condition of

```
 %v = alloca i32
 %b = alloca [1000 x double], align 8
 %i = alloca i32
 store %struct.reco* %d, %struct.reco** %d_addr
 store i32 %m, i32* %m_addr
 store i32 1, i32* %i, align 4
 br label %bb1
 %0 = load i32* %i, align 4
 %1 = call i32 @seq(i32 %0) nounwind
 %2 = load i32* %m_addr, align 4
 %3 = add nsw i32 %1, %2
 store i32 %3, i32* %v, align 4
 %14 = load i32* %i, align 4
 %15 = load i32* %i, align 4
 %16 = getelementptr inbounds [1000 x double]* %b, i32 0, i32 %15
 %17 = load double* %16, align 4
 %18 = load i32* %v, align 4
 %19 = sitofp i32 %18 to double
 %20 = call double @cos(double %19) nounwind readonly
 %21 = fadd double %17, %20
 %22 = getelementptr inbounds [1000 x double]* %b, i32 0, i32 %14
 store double %21, double* %22, align 4
 %23 = load i32* %i, align 4
 %24 = add nsw i32 %23, 1
 store i32 %24, i32* %i, align 4
 br label %bb1
 %25 = load %struct.reco** %d_addr, align 4
 %26 = getelementptr inbounds %struct.reco* %25, i32 0, i32 0
 %27 = load i32* %26, align 4
 %28 = load i32* %i, align 4
 %29 = icmp sgt i32 %27, %28
 br i1 %29, label %bb, label %bb2
 br label %return
```

Figure 4-11: LLVM representation of Slice 2 in figure 4-9

```
loop-replace:
 call void @ini_qu1()
 call void @sync_delegate(i32 0, i8* (i8*)* @"8_subloop_0")
 call void @sync_delegate(i32 1, i8* (i8*)* @"8_subloop_1")
 call void @mas_exit()
 br label %bb5
```

Figure 4-12: Loop- replace basic block

Algorithm 3 Multi-Threaded Code Generation (MTCG) algorithm. Adapted from (Ottoni et al., 2005)

```
Step 1: Create Thread Blocks.
Step 2: Move Instructions.
Step 3: Insert Synchronization.
Step 4: Create Control Flow.
```

```

entry:
 .
 .

bb:
 %0 =call i8* @genrate_address()
 store i8* %0, i8** %k, align 4
 %1 = load i32* %i, align 4
 %2 = inttoptr i32 %1 to i8*
 store i8* %2, i8** %k, align 4
 %3 = load i32* %i, align 4
 %4 = add nsw i32 %3, 1
 store i32 %4, i32* %i, align 4
 br label %bb1

bb1:
 %5 = load i32* %i, align 4
 %6 = icmp sle i32 %5, 4
 br i1 %6, label %bb, label %bb2

return:
 %retval3 = load i32* %retval
 ret i32 %retval3
```

(a) The Original CFG for first partition of DSWP

```
 new_entry_1:
 %retval_1 = alloca i32
 %k_1 = alloca i8*
 %i_1 = alloca i32
  br label bb_1

bb_1:
 %0_1 =call i8* @genrate_address()
 store i8* %0, i8** %k_1, align 4
 %1_1 =load i32* %i_1, align 4
 %2_1 =inttoptr i32 %1_1 to i8*
 store i8* %2_1, i8** %k_1, align 4
 %3_1 =load i32* %i_1, align 4
 %4_1 =add nsw i32 %3_1, 1
 store i32 %4_1, i32* %i_1, align 4
 br label %bb1_1

bb1_1:
 %5_1 = load i32* %i_1, align 4
 %6_1 = icmp sle i32 %5_1, 4
 br i1 %6_1, label %bb_1,label %bb2_1

new_exit_1:
 %retval3_1 = load i32* %retval_1
 ret i32 %retval3_1
```

(b) New CFG for first partition

Figure 4-13: Building a new CFG for the first partition of DSWP

the execution of each new basic block and its corresponding block in the original *CFG*.

Also, two new entry and exit (`new_entry_i` and `new_exit_i`) basic blocks are created for each new CFG_i, as illustrated in figure 4.13(a), which represents the original CFG, while figure 4.13(b) is the constructed one, where `i` represents the thread or slice number.

After creating the relevant basic blocks for the threads, all the instructions are cloned from the original to a new one, such as the basic block `bb` in figure 4-13 and all the instructions in this basic block are cloned to the new one `bb_1`. These instructions are inserted in the new basic block in the same order as in the original one.

Insert synchronization

To guarantee that DSWP works in the right way, it is necessary to respect the dependencies between threads. That is, efficient pipeline parallelism requires a buffering communication mechanism to provide a core-to-core communication that incurs minimal overheads. As mentioned in section 4.1, inter-thread communication has been implemented by using the FastForward circular lock-free queue algorithm. Two communication primitives `producer` and `consumer` are inserted in the source and the destination of the dependency. These communication pairs are inserted statically where each `producer` feeds one `consumer` and each `consumer` is fed by one `producer`. According to the location that the communication primitives are inserted in the code, three types of communication, `intra`, `initial` and `final` are defined which are now described in detail.

intra-communication: To determine the source and the destination of dependency between the DSWP stages, both the loop and function body have to be inspected. In the first stage DSWP the actual communication instructions for the `producer` are inserted in the same position as the instruction that corresponds to the call site. The function arguments represent the data that will be put in the communication buffers. In most of the current case studies one variable is transferred from the first thread, which represents the loop body, to the second, which signifies the function body.

In the case that the called function has n arguments and to avoid calling the enqueue and dequeue functions many times (which will be equivalent to the number of function arguments), a list of pointers is built, each node in the list pointing to the allocated memory location that has the function argument value. That is, the value of the first argument is assigned to the first memory location and the second argument to the second and so on. This design makes the extracted slice shorter and also it avoids the impact of calling the enqueue and dequeue functions many times during slice execution.

In addition, the instruction that represents the function call site in the first stage of DSWP is deleted and the set of instructions that define the enqueue call site are inserted at the same location. Figure 4-14 shows the code of the calling site for both the producer and consumer functions that are inserted in DSWP/Slice threads to communicate data dependencies.

Figure 4-15 shows the LLVM intermediate representation of the code in figure 4-14(a). Two basic blocks have been created and inserted in the first stage thread at the point where the function body is defined in the code, as can be seen in figure 4-15. This piece of code is replicated in the first thread depending on the number of extracted slices.

In each extracted slice, three basic blocks are created that represent the piece of code in figure 4-14(b), with the intermediate representation of these basic blocks being illustrated in figure 4-16. These three basic blocks are inserted after the entry block of each extracted slice and these instructions in the slice entry block are divided in two parts. The first part is kept in the entry block which represents the allocation instructions, whilst all

```
 while(ret!=0)
   {
    ret=enqueue_nonblock(&dd[1],&dd[0],first);
   }
```

```
(a) producer
```

```
  while(ret1==1)
   {
    ret1=dequeue_nonblock(&dd[2],&dd[0],&gg);
   }
```

```
(b) consumer
```

Figure 4-14: enqueue and dequeue function

```
bb_0_1:
; preds = %bb_0_2
  %P6 = load i32* %x_0
  %P7 = inttoptr i32 %P6 to i8*
  %P9 = getelementptr inbounds [10 x i8*]* @dd, i32 0, i32 0
  %P8 = getelementptr inbounds [10 x i8*]* @dd, i32 0, i32 1
  %P10 = call i32 @enqueue_nonblock(i8** %P8, i8** %P9, i8* %P7)
  store i32 %P10, i32* %ret_0
  br label %bb_0_2

bb_0_2:
; preds = %bb_0_1, %bb_0_0
  %P11 = load i32* %ret_0
  %P12 = icmp eq i32 %P11, 1
  br i1 %P12, label %bb_0_1, label %bb_0
```

Figure 4-15: Intermediate representation of the code that is used to call the enqueue function

instructions that give initial values, which represents the second part of the instructions, have to be moved from the slice entry block to the third basic block. In addition, this third block will work as an entry block that resets the value to the slice variables and gives them initial values after each call. Figures 4-16 and 4-17 show the third basic block after moving the second part of the instructions from the entry block in figure 4-8.

The extracted slice is inspected in order to be able to allocate where to put the retrieve data from the dequeue function so that the DSWP/Slice stages work in the correct way.

```
bb_1_0:
; preds = %bb_1_1
  %C7 = bitcast i32** %ret_value1 to i32*
  %C9 = getelementptr inbounds [10 x i8*]* @dd, i32 0, i32 0
  %C8 = getelementptr inbounds [10 x i8*]* @dd, i32 0, i32 2
  %C10 = call i32 @dequeue_nonblock1(i8** %C8, i8** %C9, i32* %C7)
  store i32 %C10, i32* %retdq_1
  br label %bb_1_1

bb_1_1:
; preds = %bb5_1_1, %bb_1_0
  %C13 = load i32* %retdq_1
  %C14 = icmp eq i32 %C13, 1
  br i1 %C14, label %bb_1_0, label %bb_1_2

bb_1_2:
; preds = %bb_1_1
  %C16 = load i32* %C7
  store i32 %C16, i32* %t39_1
  br label %bb1_1
```

Figure 4-16: Intermediate representation of the code that is used to call the dequeue function

For each slice there is a communication buffer associated with it. The first thread (which represents the first stage DSWP) will put the communicated data in n buffers and each slice retrieves a copy from the buffer that is allocated to it.

initial communication: The second type of communication is the initial one where live-in variables, those defined outside the loop body and used inside it that are not global variables, need to be communicated. However, these variables are not going to affect the execution time of DSWP because it just happens once before calling the first thread. In addition, the initial value of the loop induction variable has to be reset. That is, this variable is defined and its initial value set outside the loop basic blocks, but it has to be redefined and reset inside the first stage DSWP thread.

final communication: The third type is a final communication where at the end of the second stage (there are just two stages in this work) some of the extracted slices need to communicate their values back to the main thread. For this type of communication a new buffer is added and also, both `producer` and `consumer` instructions have to be added at each side. Two simple library functions have been written in C to support Fastforward lock-free buffer enqueue, dequeue routines along with the functions library that have been developed by Fuyao Zhao (2011) which in turn uses the pthread library.

As mentioned before a FastForward circular lock-free queue algorithm is used in this research. In this algorithm, the data store in the buffer has been used to determined the queue state,

```
bb_1_0:
; preds = %bb_1_1
  %C7 = bitcast i32** %ret_value1 to i32*
  %C9 = getelementptr inbounds [10 x i8*]* @dd, i32 0, i32 0
  %C8 = getelementptr inbounds [10 x i8*]* @dd, i32 0, i32 2
  %C10 = call i32 @dequeu_nonblock1(i8** %C8, i8** %C9, i32* %C7)
  store i32 %C10, i32* %retdq_1
  br label %bb_1_1

bb_1_1:
; preds = %bb5_1_1, %bb_1_0
  %C13 = load i32* %retdq_1
  %C14 = icmp eq i32 %C13, 1
  br i1 %C14, label %bb_1_0, label %bb_1_2

bb_1_2:
; preds = %bb_1_1
  %C16 = load i32* %C7
  store i32 %C16, i32* %t39_1
  store i32 %d, i32* %d_addr
  %0_1 = getelementptr inbounds [1000000 x i32]* %a1, i32 0, i32 0
  store i32 0, i32* %0_1, align 4
  %1_1 = getelementptr inbounds [1000000 x i32]* %b, i32 0, i32 0
  store i32 200, i32* %1_1, align 4
  store i32 1, i32* %i, align 4
  br label %bb1_1
```

Figure 4-17: Intermediate representation of the dequeue blocks after instructions moving

instead of a control variable (head/tail). This has the advantage of reducing the overhead of accessing the control variable.

We made adaptation to the dequeue procedure related with the termination of the consumer thread. We decoupled the comparision of two values which are the `data` and `flag`. If the `data` field is equal to NULL and the `flag` value does not equal 100 the consumer will execute busy wait until this data field value equal to NULL or the value of flag is equal to 100. When both conditions are true the consumer will stop (no more data in the queue). Further explanation of second stage termination is illustrated in subsection 4.2.3.

Create control flow

The last step in the code generation algorithm inserts the control flow in each thread and this can be partitioned into two parts. The first relates to the re-targeting of some of the branch instructions in the created thread. This is because of the fact that not all the basic blocks are going to be copied or involved in the created thread. For example, some threads have only one or two basic blocks, such as the second and the fourth. Therefore, in this case it is necessary

```
enqueue_nonblock(data)
 {
  if (NULL != buffer[head]) {
      return 1;
   }
   buffer[head] = data;
   head = NEXT(head);
    return 0;
}
```

Enqueue Function

```
dequeue_nonblock(data)
 {
  data = buffer[tail];
  if (NULL == data) {
  return 1;
  }
  if (NULL == data)&&(flag==100) {
  return 2;
  }
  buffer[tail] = NULL;
  tail = NEXT(tail);
  return 0;
}
```

Dequeue Function

Figure 4-18: FastForward Lock-free buffer algorithm, after (Giacomoni et al., 2008)

to redirect the branch instructions to guarantee that the constructed CFG_i works like the original program and for more clarity on this matter, see figure 4-13. A new entry block called `new_entry_1` is created and its branch instruction has to be redirected so as to make it point to block `bb_1`.

Another issue related to the termination of the second stage DSWP, is that because the loop induction variable already controls the first stage, it is necessary to use a control variable to inform the second stage that there are no more values in the buffer and so the thread that represents the second stage should be stopped. To this end, the dequeue and enqueue functions has to be adapted by using a global variable called a flag with its initial value set at zero. When the first stage finishes its work it sets the flag value at 100. The second stage checks the flag value every time it retrieves a value from the buffer. When the value of flag equals 100 the return value from the dequeue function is equal to 2 and at this point means the second stage

```
while( ret1 != 2 )
  {
   ret1=dequeu_nonblock(&dd[2],&dd[0],&gg);
   }
```

Figure 4-19: Ending the execution slice

will finish its work as there are no more values that are going to be consume by this slice. All the code in figure 4-19 is inserted at the end of the extracted slice and the branch instruction in the entry block has to be redirected. Also, the branch instruction in the last inserted block has to be re-targeted to make it point to the exit block, if the condition is true and if it is false, the branch instruction re-directs to the block that checks the buffer to see if there is more data to retrieve and consume by the consumer.

4.3 Compiler implementation

The DSWP/Slice technique has been implemented in the framework of the Low Level Virtual Machine (LLVM) compiler infrastructure and with the support of the general-purpose POSIX threading library (pthread). Our implementation operates as a pass of the LLVM compiler.

Synchronization primitives (producer and consumer) have been written as a separate external library, which is linked with the source code at compile time, to supply the lock-free queue abstraction. If a queue is empty, the consumer thread busy waits until a value is written into the buffer and if a queue is full the producer busy waits until space is available.

The synchronization primitives are inserted into the program where they are used control the storage and retrieval of values to and from the communication buffer. The program executes sequentially until it reaches the parallelization point (the start of the chosen loop), at which point the threads are created (the external library function "sync_delegate" is called to create threads for each of the DSWP stages and each of the extracted slices) and each is given the address of the function assigned to it. For each thread dependency, a buffer is allocated, so each producer/consumer pair operates on the same buffer.

In addition, another library function is called to allocate a set of list of pointers that buffer data entries refer to it. Each node in the list points to the memory location for a function argument value. Thus, we avoid repeated calling the enqueue and dequeue functions for each function argument.

4.4 Summary

This chapter has introduced the automatic implementation of the DSWP/Slice technique that is proposed to alleviate the bad effect of the long stage DSWP using a slicing technique. Instead of giving the whole long stage DSWP to one thread, it can be distributed across n threads, depending on the number of slices extracted from this stage. In consequence, better load balancing can be achieved between the DSWP stages. An adaptation of the method of Fuyao Zhao (2011) has been created to achieve this optimisation.

This chapter started with the motivating example that illustrates the proposed methods showing how unbalanced DSWP stages can negate any benefit from the DSWP and introduced the slicing technique as an alternative solution to alleviate the bad effect of the long stage DSWP in case other transformation become inapplicable. This introduction was followed by a description of the DSWP/Slice algorithm, which consisted of three parts determining a thread assignment, slice extraction and code generation. The first relates to dividing the loop body between DSWP stages and involves several steps: choosing the most profitable loop for this method by accumulating the estimated cycles necessary to execute all instructions in every loop in the program and choosing the loop that represents the hot region or has the highest execution time in the program. The second step builds the PDG and then constructs the DAG graph for SCCs as well as assigned these SCCs to their thread. The second part has described the extracting slice process.

It started with building the PDG and then the entry block of function body has been examined as well as the slicing variables being defined. Then some of filtering process that were applied to define the accurate numbers of the extracted slices were explained. Finally, the third part has illustrated the process of generating code for these threads, which as with part one also involves several stages: creating thread blocks, moving instructions, inserting synchronization and subsequently, creating control flow.

Chapter 5

Evaluation of DSWP/Slicing Transformation

This chapter presents an evaluation of the automatic implementation for the combination method that has been presented in chapter 4. It discusses the results obtained from applying this technique to several programs that are suitable as case studies. Some of these programs are artificial and the others are taken from the website (Burkardt, 2012; Pang, 1997) (The programs used are given in appendix A). The chosen case studies can, in some cases, be parallelized using other techniques, such as DOALL. However they are also considered here as useful examples with which to demonstrate the applicability of our proposed method. Before starting with evaluating the results, some details are provided about the evaluation machine in Table 5.1, that is, it has an Intel Corei7 processor with a speed of 2.93GHz and 4GB RAM.

Table 5.1: Platform Details

Processor	Intel(R) Core(TM) i7 CPU
Processor speed	2.93 GHz
Processor Configuration	1 CPU, 4 Core, 2 threads per Core
L1d Cache size	32 k
L1i Cache size	32 k
L2 Cache size	256 k
L3 Cache size	8192 k
RAM	4.GB
Operating System	SUSE
Compiler	GCC 4.2.1 and LLVM 2.8

This chapter also shows how this method enables the automatic extraction of parallelism. The evaluation is divided into three parts, with the first discussing the effect of the Lock-Free buffer technique on the performance of DSWP, the second part shows how the DSWP/Slice technique can improve the performance of long stage DSWP with different program patterns and finally, the third considers the effect of buffer size and slice length on the performance of DSWP/Slice. At the end of this chapter, some comparisons between this work and that done by Huang et al. (Huang et al., 2010) are provided and also the effect of inlining techniques on the slicing is discussed.

5.1 Communication Overhead

In this subsection the effect of the communication overhead and the buffer size on the performance of DSWP are investigated, a starting with the former. Considering the program in figure 5-1, the aim is to execute this program by applying DSWP on the loop that takes up the most execution time of the program.

To start with, this program is split into two parts, with each being assigned to a thread, which are then executed as a pipeline. The first thread has the statements from lines 4–15 and the second those from lines 17–28. The two parameters that have a vital role in improving the performance of the DSWP technique are `N` and `M`, where the former refers to data transfer between (in this experiments one value per iteration was transferred) threads and the latter represents the amount of work inside each of them.

Figure 5-2 shows how changing the value of `N` from 1-320 iterations can improve the execution time of DSWP compared with the sequential program. From the first iteration to the fifth the sequential version of the program runs faster than the DSWP one, with `M` = 25,600 iterations where both numbers 320 and 25,600 are sufficient to show the required behaviour. This is because of the overhead between two threads as the second thread does not start until the first one has finished its job. At iteration 10 the performance of the sequential program becomes equal to the DSWP one and from the 20th iteration to the 320th the effect of overlap becomes more obvious.

Figure 5-3 shows the amount of work that each thread has to do to improve the performance of DSWP. The value of `N` is fixed to 1000 (less than 1000 the execution time is too small) iterations and the value of `M` from 100 to 25,600. At `M` equals 800 the execution time of sequential program is equal that of DSWP, while with a large number of `M` the performance of DSWP becomes better (DSWP takes less execution time compared with the sequential program).

From both figures 5-2 and 5-3 we can notice the following points:

1- When the amount of computation inside each stage is large enough (for one iteration) the overhead of transferring data between DSWP stages becomes inconsequential (one data per iteration). However, this claim can not be generalised when there are many data per iteration needing to be transferred to respect the dependency between threads. This is because the overhead of calling the enqueue and dequeue functions many times needs to be added.

2- Small amounts of work inside the DSWP stages, with a small number of loop iterations

```
rows=N;
 for(i1=1; i1 < rows; i1++)
   {
   for(z=1;z<M;z++);
      {
       sum = 0;
       for(a=1; a<10; a++)
          sum = sum + image[i1]*mask_1[a];
        if(sum > max)
          sum = max;
        if(sum < 0)
           sum =10;
     if(sum < out_image[i1])
        out_image[i1]= sum;
        }

  for(z1=1;z1<M;z1++);
      {
      sum1 = 0;
      for(a1=1; a1<10; a1++)
           sum1 = sum1 + image[i1] * mask_2[a1];
        if(sum1 > max)
           sum1 = max;
        if(sum1 < 0)
           sum1 = 10;
      if(sum1 > out_image[i1])
         out_image[i1]=sum1;
       }
  }
```

Figure 5-1: Sequential program

and also with one function argument being transferred between DSWP stages can degrade its performance.

3- It can be seen that a high number for both N and M gives better results, because the overlap between the loop partitions can obscure the side effect of communication overhead, as observed in figure 5-4, where the X-axis in this figure represents the number of iterations in the inner loop (M).

5.2 DSWP/Slice

This section presents the results that were obtained from applying a slicing technique to the long stage DSWP. Most of the case study programs have unbalanced DSWP stages (i.e the number

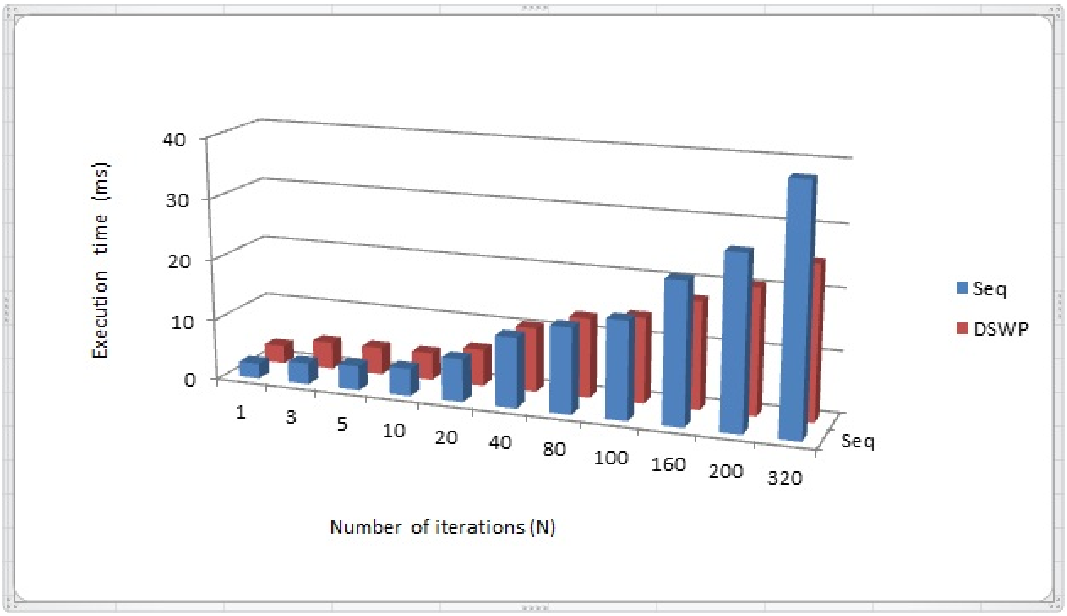


Figure 5-2: The effect of `N(M=25,600)` on the DSWP

Table 5.2: Case Studies Details

Program name	function name	function execution time(ms)
simple4.c	fun	0.77
linked2.c	calculate	0.79
linked3.c	calculate	0.82
fft.c	fft	0.85
pro-2.4.c	three	0.82
test666.c	spherical_harmonic	0.85

of instructions inside the outer loop is less than the number in the function body). Table 5.2 illustrates the amount of execution time spent inside each function and that by adding the slicing technique a balance can be achieved

Six case study programs were used to evaluate the proposed method, three being artificial (`simple4.c`, `linledlist2.c` and `linkedlist2.c`) and the other three (`fft.c`, `pro_ 2.4.c` and `test666.c`) real problems. For more details about these programs see appendix A. For all the case study programs two slices were extracted from the function body and the maximum

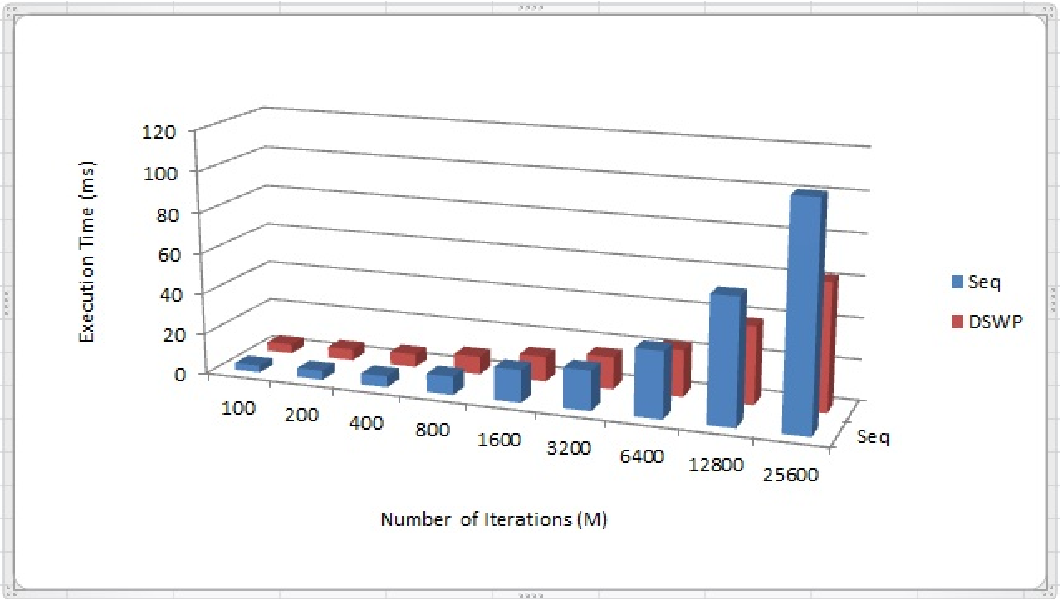


Figure 5-3: The effect of `M(N=1,000)` on the DSWP

number of threads for these programs is four. In addition, the maximum amount of transferred data between threads was four values for each iteration with there being different data types, for example int value, double and struct value (e.g. linkedlist). Moreover, from experience using these programs it was clear that heavy calculation inside the function body leads to better results, as might be expected.

In addition to the LLVM compiler, a GCC compiler was used to implement this method manually and subsequently, a comparison between the manual and automatic results was made.

The automatic method has two passes. The first is called the ANAL pass which carries out some static analysis for a given case study program and traverses all the loops in program. For each loop it accumulates a static execution time for each instruction as well as execution time for the function body and put these results in table. The second pass is called the DSWP pass, with the first step of this being traversing the previous table. That is, depending on this information the DSWP pass will have the ability to chose which loop is more suitable to be implemented with the proposed method. Two conditions have to be available in the chosen loop. Firstly, it should have the highest execution time of all the loops and secondly, it should have a function call inside it with the proportion of work inside this function higher than the rest of the loop instructions altogether.

Before evaluating the results, some clarification relating to the columns heading of the

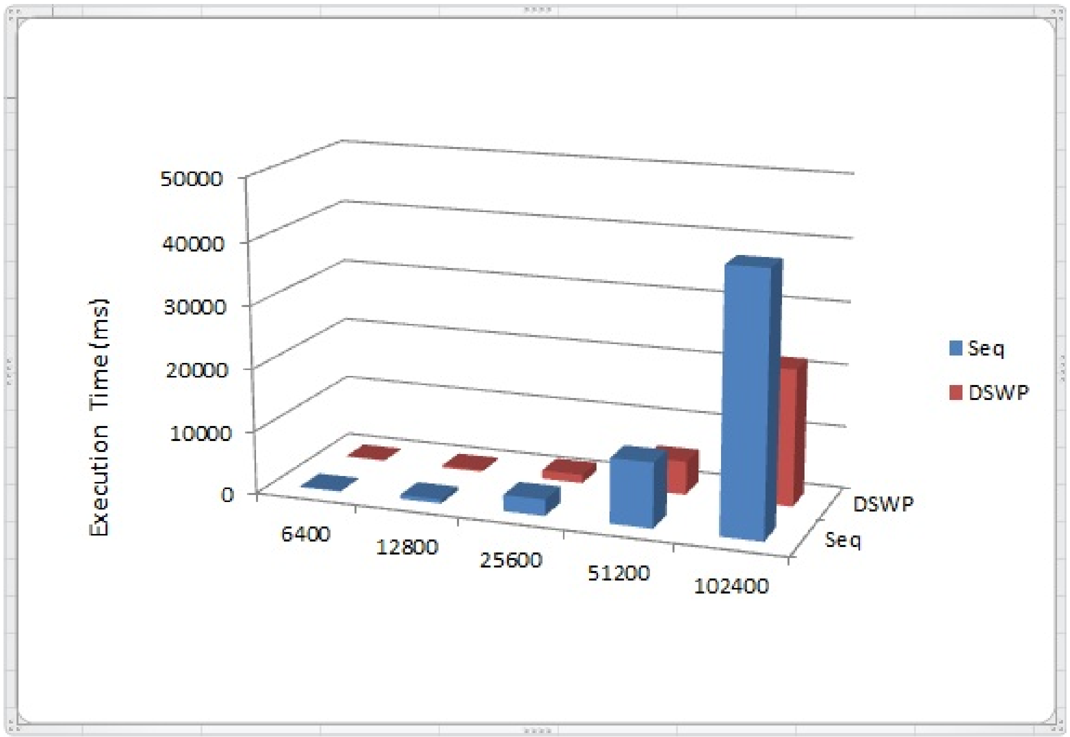


Figure 5-4: The effect of large `M(N=25600)`

results tables is illustrated below:

1- Column one (iter) shows the loop iterations.

2- Column two (Seq+O) shows the results for the sequential program compiled with LLVM (with optimization).

3- Column three (Seq-O) shows the results for the sequential program compiled with LLVM (no optimization).

4- Column four (DSWP-Slice+O) shows the results for the program compiled with LLVM with DSWP/Slice (with optimization).

5- Column five (DSWP-Slice-O) shows the results for the program compiled with LLVM with DSWP/Slice (no optimization).

6- Column six (Seq-O) shows the results for the sequential program compiled with GCC (no optimization)

	LLVM				GCC		
Iter.	Seq+O	Seq-O	DSWP-Slice+O	DSWP-Slice-O	Seq-O	DSWP-Slice-O	DSWP-O
50	0.364	0.385	0.217	0.222	0.434	0.229	0.410
100	0.698	0.731	0.396	0.412	0.863	0.446	0.823
150	1.029	1.090	0.575	0.589	1.270	0.678	1.243
200	1.384	1.455	0.767	0.823	1.740	0.9	1.675
250	1.695	1.800	0.960	0.992	2.109	1.119	2.055
300	2.041	2.159	1.150	1.193	2.520	1.355	2.472
350	2.400	2.481	1.325	1.390	2.930	1.575	2.875
400	2.729	2.873	1.521	1.584	3.339	1.819	3.311

Table 5.3: Execution times for program simple4.c

7- Column seven (DSWP-Slice-O) shows the results for the program compiled with GCC with DSWP/Slice (implemented manually, no optimization)

8- Column eight (DSWP-O) shows the results for the program compiled with GCC with DSWP (implemented manually, no optimization).

Next, each of the chosen programmes is described and their performance is analysed.

5.2.1 simple4.c program

As mentioned earlier, this program is an artificial program, being made to illustrate the speed up that can be obtained from a combination of DSWP and slicing. First, the loop body was executed with the DSWP technique. This splits the loop into two stages: the function call instruction (which represents the SCC has a higher latency than the other) was allocated to the second stage DSWP and the rest of the loop SCCs to the first. Then slicing was applied to the function body. For this, two slices were extracted from the function `Fun` by traversing all variables that were allocated in the function entry block. The next step was to filter these slices by deleting some of them that were included in others (see section 4.2.2). For the `simple4.c` program two slices were extracted after the filtering process; one representing array `a[ ]` and the second array `a1[ ]`. The number of threads was three, the first pertaining to the producer and the second and third the consumers. Also, two buffers are used to transfer data from the former to the latter.

Table 5.3 presents the manual and automatic results with different numbers of iterations. Columns 3 and 5 represent the execution time of the sequential and parallelized (LLVM-DSWP-Slice) versions of the `simple4.c` program, respectively. They show that the automated method gives ≈ 1.75 speed-up compared to the sequential one in the LLVM environment without optimization. By comparing the execution time of column 2 with columns 4 and 5 it can be seen that the execution times of the LLVM-DSWP-Slice+O and LLVM-DSWP-Slice-O are better than the sequential version with optimization in the LLVM environment.

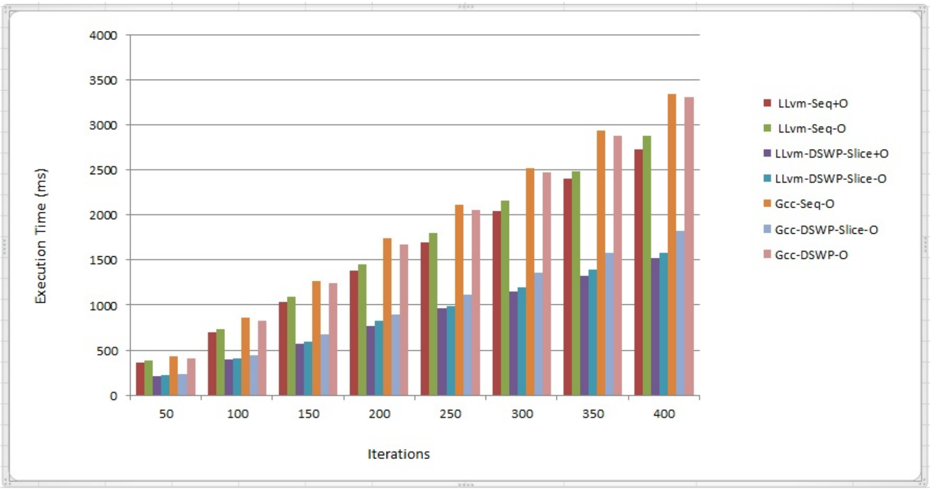


Figure 5-5: Loop speed-up with three threads for `simple4.c` program

Columns 6, 7 and 8 represent the manual results for the execution time for the sequential program, execution time after applying the slicing technique to the DSWP and the execution time for DSWP without slicing, respectively, under the GCC environment. It is observed that DSWP alone does not give a noticeable speed-up, only ≈ 1.09, whilst it becomes ≈ 1.9 after applying the slicing technique.

Figure 5-5 shows loop speed-up using DSWP/Slice. For this figure it can be seen that from iterations (50-400) the performance of DSWP-Slice (manual and automatic with or without optimization) is better than the sequential version and also shows a good improvement over DSWP alone with GCC. Moreover, there are no big differences in the execution time between the optimized and unoptimized one in the LLVM environment, whereas both are better than the unoptimized one in the GCC environment.

5.2.2 linkedlist.c program

The second program is also an artificial one that is traversing a linked list and has two versions.

1- The first is `linked2.c` that has a function that does not return values to the candidate loop, two slice were extracted. The first slice represented array `b1[ ]` and the second was array `b2[ ]`. The number of threads is three, the first pertaining to the producer and the second and third the consumers. The value of `x` represents the number of nodes in the linked list being traversed (from 5-35) and the value of `z` represents the number of nodes in the inner linked list. The value of `z` was set to 1,000 for the first experiment and then this was changed to 2,000 (it is enough to show an improvement) for the second. Also, two buffers were used to transfer one value from the producer to the consumers.

(i) First experiment result (`linked2-exp1.c`)

	LLVM				GCC		
Iter.	Seq+O	Seq-O	DSWP-Slice+O	DSWP-Slice-O	Seq-O	DSWP-Slice-O	DSWP-O
5	0.151	0.157	0.105	0.110	0.153	0.104	0.188
10	0.320	0.330	0.190	0.200	0.330	0.210	0.369
15	0.487	0.493	0.283	0.290	0.520	0.312	0.545
20	0.637	0.660	0.354	0.370	0.705	0.407	0.733
25	0.811	0.830	0.439	0.460	0.870	0.505	0.909
30	0.975	1.002	0.530	0.560	1.056	0.607	1.090
35	1.135	1.172	0.620	0.650	1.245	0.710	1.265

Table 5.4: Execution times for program `linked2-exp1.c` (without returning value)

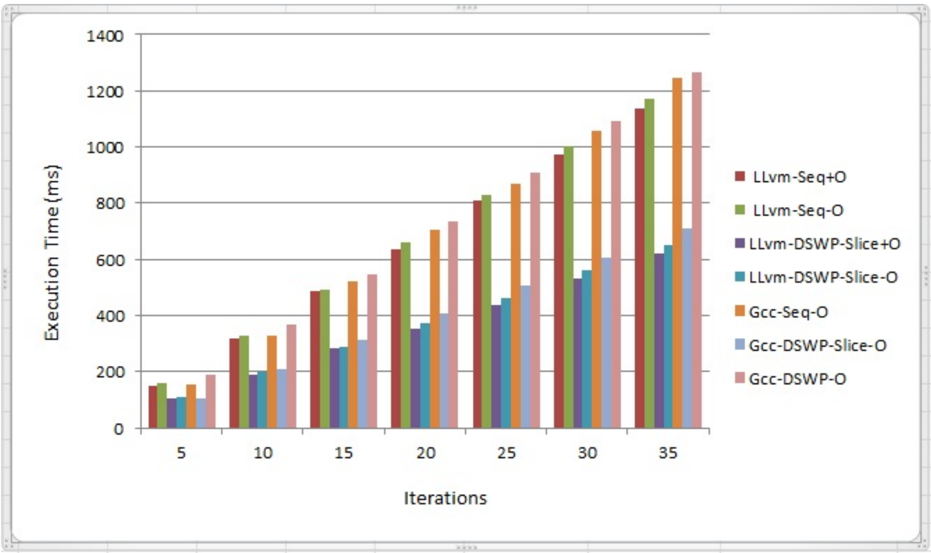


Figure 5-6: linkedlist2-exp1.c program

Table 5.4 shows the improvement that is achieved from the linkedlist program with the method employed (experiment 1). Columns 3 and 5 represent the execution times for the sequential and LLVM-DSWP-Slice versions, respectively of the `linked2.c` and illustrate that the automated method gives ≈ 1.8 speed-up compared with the sequential one in the LLVM environment without optimization. Similar to the previous program, the execution time of the LLVM-DSWP-Slice with optimization is better than the Seq+O one. Regarding manual execution in the GCC environment, as shown in columns 6, 7 and 8, the DSWP performance is worse than the sequential one (because the amount of work in the first thread is very little compared with the second stage), while the speed-up becomes ≈ 1.75 after applying slicing. Figure 5-6 shows loop speed-up using the DSWP/Slice technique. In this figure it can be seen

	LLVM				GCC		
Iter.	Seq+O	Seq-O	DSWP-Slice+O	DSWP-Slice-O	Seq-O	DSWP-Slice-O	DSWP-O
5	0.550	0.559	0.307	0.320	0.600	0.393	0.721
10	1.194	1.227	0.651	0675	1.290	0.780	1.442
15	1.850	1.900	0.985	1.025	2.002	1.165	2.130
20	2.491	2.545	1.332	1.390	2.720	1.550	2.860
25	3.134	3.225	1.659	1.737	3.459	1.925	3.560
30	3.794	3.915	2.100	2.011	4.153	2.340	4.304
35	4.475	4.592	2.330	2.433	4.838	2.710	5.007

Table 5.5: Execution times for program `linked2-exp2.c` (without returning value)

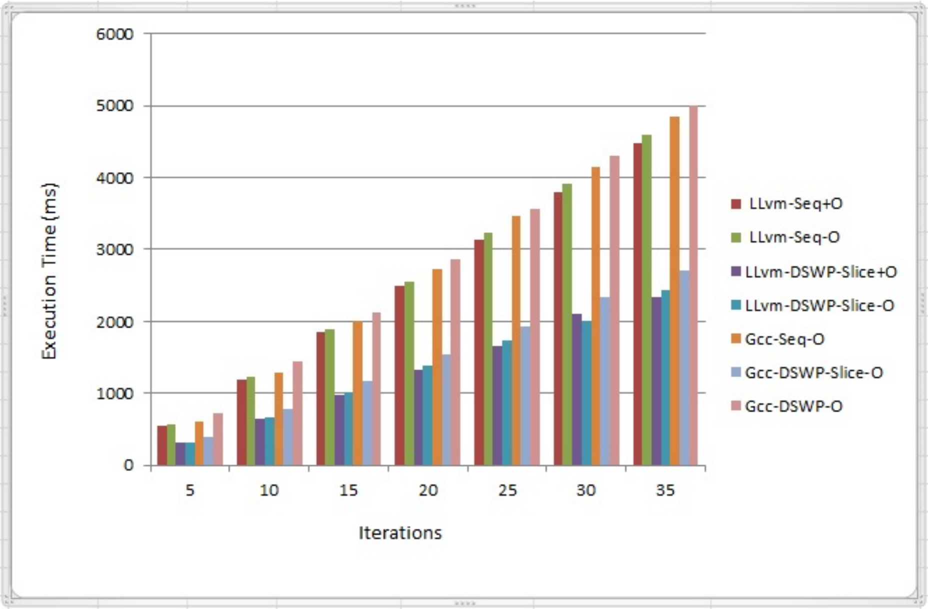


Figure 5-7: linkedlist2-exp2.c program

that with a small iterations count (5) the execution time of DSWP-Slice (with or without optimization) is approximately close to the execution time of the sequential program, while the difference between them becomes clearer with higher iterations (15-35). Also there is no significant difference between the manual DSWP-Slice and the automatic one.

(ii) Second experiment result (`linked2-exp2.c`)

For the second experiment, the value of `z` was set at 2,000. By comparing the results in column 5 in the tables (5.5 and 5.4), it can be seen that the speed-up is ≈ 1.92, which is higher than for the previous experiment(≈ 1.8 with `z`=1,000). That is, a better result can be achieved with a larger amount of calculations inside each slice,

	LLVM				GCC		
Iter.	Seq+O	Seq-O	DSWP-Slice+O	DSWP-Slice-O	Seq-O	DSWP-Slicing-O	DSWP-O
5	0.150	0.156	0.106	0.110	0.160	0.111	0.188
10	0.321	0.331	0.192	0.202	0.335	0.210	0.369
15	0.490	0.502	0.282	0.295	0.530	0.299	0.545
20	0.658	0.667	0.360	0.386	0.710	0.408	0.733
25	0.815	0.855	0.465	0.483	0.880	0.507	0.909
30	0.980	1.008	0.540	0.570	1.065	0.611	1.090
35	1.151	1.176	0.629	0.655	1.250	0.712	1.265

Table 5.6: Execution times for program `linkedlist3.c` (with returning value)

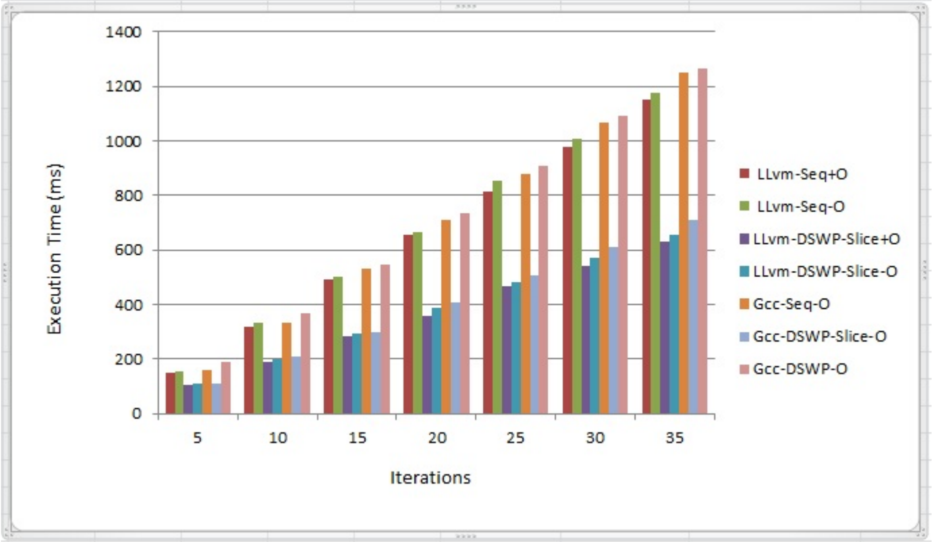


Figure 5-8: `linkedlist3.c` program

as illustrated in figure 5-7.

2- The second version of the linkedlist program has a function that returns values to the candidate loop.

In this case the second stage DSWP represents the function body together with the SCCs that have data dependency with it. After applying the slicing technique to the function body, the number of extracted slices is two and the number of worker threads four. The first represents the producer and the second and third the consumers. Also, the slice that returns a value to the loop body plays the part of a producer providing value to the fourth thread. Moreover, another buffer is added so the total number is three. Table 5.6 illustrates how the extra thread and buffer can affect the execution time of the linkedlist program. By comparing the results in column 5 in the tables 5.6 and table 5.4, it is found

	LLVM				GCC		
Iter.	Seq+O	Seq-O	DSWP-Slice+O	DSWP-Slice-O	Seq-O	DSWP-Slice-O	DSWP-O
5	0.370	0.702	0.396	0.406	0.700	0.310	0.558
10	0.730	1.375	0.765	0.780	1.391	0.690	1.244
15	1.080	2.058	1.130	1.155	2.078	1.069	1.934
20	1.430	2.750	1.488	1.532	2.770	1.453	2.625
30	2.133	4.106	2.240	2.272	4.130	2.214	3.972
40	2.830	5.474	2.930	3.013	5.530	2.954	5.390

Table 5.7: Execution times for program `fft.c`

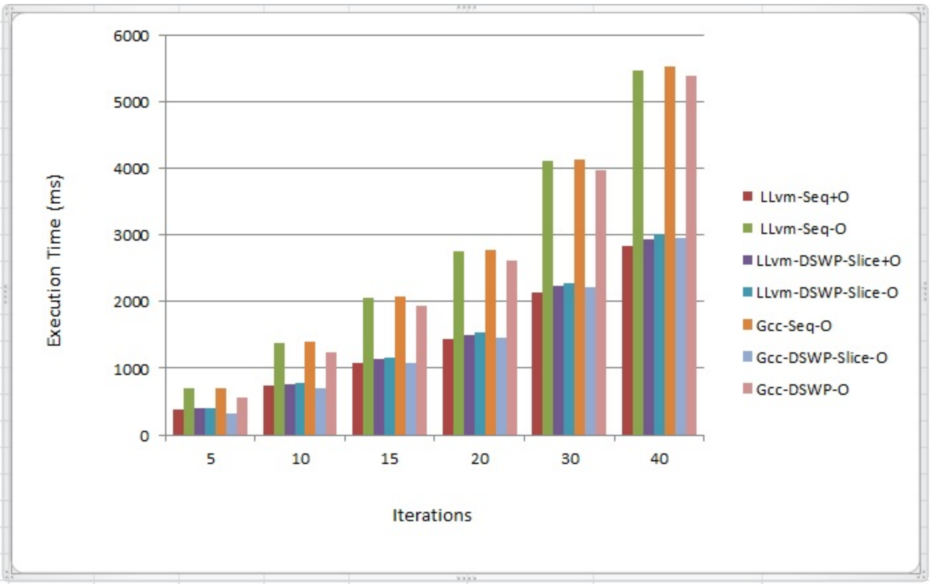


Figure 5-9: `fft.c` program

that the extra thread and buffer do not have a significant effect on the execution time.

Figure 5-8 shows loop speed-up of `linkedlist3.c` using the DSWP/Slicing technique

5.2.3 fft.c program

The third program computes the Discrete Fourier transform, which was generalized to make it work with N functions with `NMAX`=1024 and `MMAX`=8. The outer loop was given to the first thread and the `DFT` function to the second. After applying the slicing technique to the second stage, which represented long stage DSWP, two slices were extracted from the function body, the first pertaining to the real part, `gr[ ]`, and the second the imaginary one, `gi[ ]`. The number of threads was three with two communication buffers.

By considering table 5.7 showing the results obtained by the automatic and manual im-

	LLVM				GCC		
Iter.	Seq-O	Seq+O	DSWP-Slice-O	DSWP-Slice+O	Seq-O	DSWP-Slice-O	DSWP-O
5	0.088	0.052	0.066	0.055	0.083	0.042	0.058
10	0.153	0.080	0.100	0.081	0.153	0.077	0.103
15	0.227	0.110	0.130	0.108	0.220	0.101	0.145
20	0.290	0.140	0.153	0.134	0.292	0.134	0.188
25	0.353	0.170	0.180	0.150	0.365	0.168	0.230
30	0.419	0.198	0.217	0.179	0.450	0.210	0.275

Table 5.8: Execution times for program `pro-2.4.c`

plementation of the sequential and DSWP/Slice versions, it can be seen how the imbalance of long stage DSWP can affect performance in that that for DSWP is very close to that for the sequential program. Columns 3 and 5 illustrate the execution times for the sequential and LLVM-DSWP-Slice versions of the `fft.c` program and reveal that the automated method gives ≈ 1.79 speed-up when compared with the sequential program in an LLVM environment without optimization.

Taking columns 6 and 7 in the GCC environment, it can be seen that DSWP alone does not give a noticeable speed-up, only ≈ 1.2, while this becomes ≈ 1.94 after applying slicing. Figure 5-9 shows loop speed-up using the DSWP/Slice technique. Even with a small number of iterations (5) the performance of the unoptimized version of LLVM-DSWP-Slice-O is better than the unoptimized sequential program; however, its performance is close to the optimized sequential one and it stays steady through other iterations.

5.2.4 pro-2.4.c program

This fourth program computes the derivative of N functions. `F1` represents the first derivative, `F2` the second, `D1` is the error in `F1` and `D2` is that in `F2`. Similar to the previous program two main slices were extracted from the function body after giving the function body to the second stage DSWP. Some adaptations were added to the program in that it was generalized so as to make it work with N number of functions. A value of `NMAX` = 100,000 was set with a value of `M` varying between `M`=5 and `M`=30. Table 5.8 gives the manual and automatic execution times for the sequential, DSWP and DSWP/Slice applications.

The obtained results from the sequential and DSWP/Slicing versions show that the automated method gives ≈ 1.8 speed-up as compared with the sequential program in the LLVM environment without optimization (see columns 2 and 4 in the table 5.8). Columns 3 and 5 show that the performance of DSWP-Slice+O improves over Seq+O with a high number of iterations, which means that the LLVM optimization can have a noticeable affect with a small number of iterations. Under the GCC environment, as columns 6 and 7 show, the speed-up becomes ≈ 2.2 after applying the slicing technique while it is 1.5 after applying the DSWP alone because of long stage DSWP. Figure 5-10 shows loop speed-up of the `pro-2.4.c` program

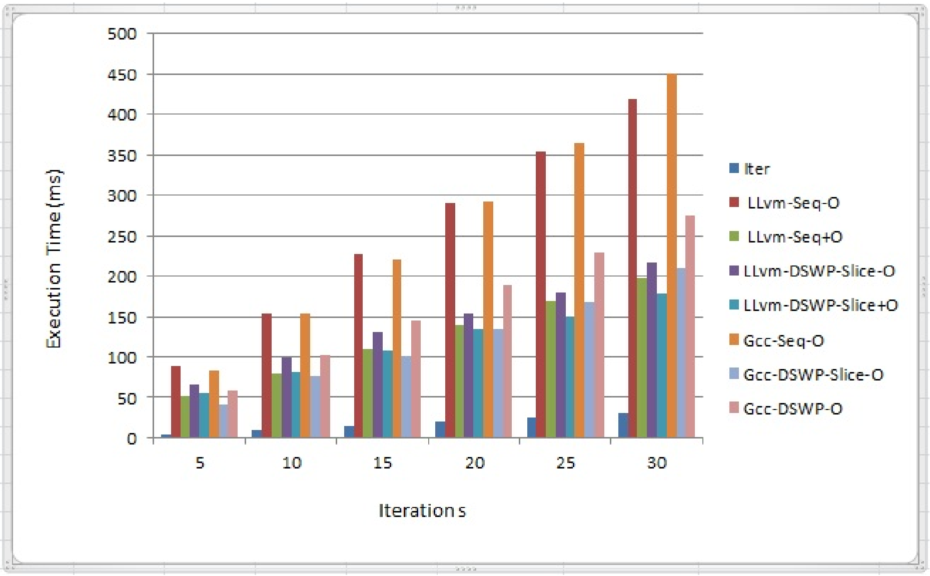


Figure 5-10: `pro-2.4.c` program

	LLVM				GCC		
Iter.	Seq-O	Seq+O	DSWP-Slice-O	DSWP-Slice+O	Seq-O	DSWP-Slice-O	DSWP-O
2	0.135	0.125	0.119	0.115	0.370	0.272	0.304
5	0.215	0.194	0.173	0.140	0.628	0.420	0.483
7	0.250	0.220	0.190	0.150	0.875	0.602	0.667
9	0.360	0.235	0.260	0.227	1.140	0.775	0.866
11	0.410	0.366	0.263	0.230	1.387	0.954	1.046
13	0.523	0.463	0.366	0.306	1.651	1.115	1.242

Table 5.9: Execution times for program `test0697.c`

using the DSWP/Slice technique.

5.2.5 test0697.c program

This program computes the spherical harmonics function, which is used in many physical problems ranging from the computation of atomic electron configuration to the representation of the gravitational and magnetic fields of planetary bodies. It has two function calls inside the loop body. The first, called the `spherical-harmonic-value`, gives the initial value to the second function argument, with this function being called the `spherical-harmonic`. The loop was divided into two parts, depending on the instruction latency execution time. The second function call, which represents the spherical-harmonic computation was allocated to the second thread, whilst the rest of the loop body containing the first function call is assigned to the first

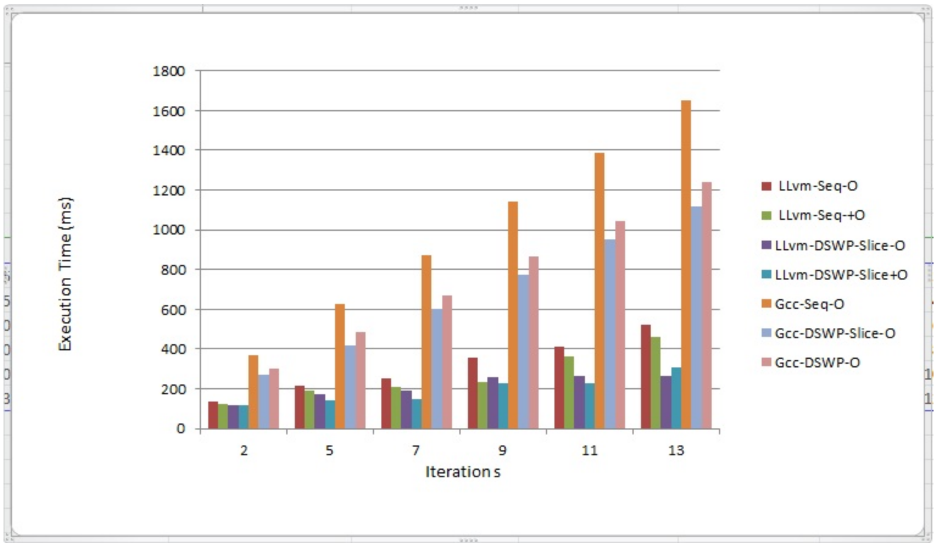


Figure 5-11: Loop speed-up with three threads for `test0697.c` program

thread Subsequently, two slices, `c[ ]` and `s[ ]`, were extracted from the second function call by applying slicing technique on this part alone. With high values (40000) of `L` and `M` the execution time of this combination is better than for the sequential program. The number of threads was three with two communication buffers and the number of transferred function arguments was four.

The results obtained by automatic and manual implementation for the sequential and DSWP/Slice versions, show that the former method gives ≈ 1.4 speed-up compared with the sequential program in the LLVM environment without optimization (see columns 2 and 4 in the table 5.9). Also, comparing columns 3 and 5 it is clear that the performance of DSWP-Slice+O (with optimization) is better than that for Seq+O. Moreover, columns 6 and 7 under the GCC environment show that the speed-up becomes ≈ 1.5 after applying the slicing technique, while that for DSWP alone is only ≈ 1.3.

5.3 Buffer size and Slice length

In this subsection the effect of the buffer size on the performance of DSWP is examined, for which the same program as in figure 5-1 was employed. However this time the value of `N` was fixed to 1,000 and `M` to 10,000 and the only parameter changed was the buffer size. That is, it varied between 10 and 1,000, with the execution time of the program being only slightly changed during the execution (2 to 5 ms) which is attributable to the construction of the longer queue. As a result, it can be concluded for this experiment that the effect of buffer size on DSWP is minor.

	LLVM				GCC	
Iter.	Seq+O	Seq-O	DSWP-Slice+O	DSWP-Slice-O	Seq-O	DSWP-Slice-O
10	0.670	0.694	0.687	0.690	0.780	0.760
50	0.341	0.360	0.365	0.365	0.393	0.379
100	0.670	0.694	0.687	0.690	0.780	0.760
200	1.322	1.375	1.410	1.432	1.561	1.506
300	1.980	2.060	2.145	2.159	2.336	2.599
400	2.628	2.700	2.877	2.897	3.160	3.727
500	3.290	3.425	3.610	3.630	3.900	4.841

Table 5.10: Execution times for program `simple4.c`

Next, the effect of unequal length slices on the performance of the proposed method is investigated. To this end, a modified version of the simple4.c program with two extracted slices was used, with the length of one of the slices being small and that of the second being close to the length of the original function. The table 5.3 shows the results obtained in the LLVM and GCC environments. By comparing the results in column 5 with those in column 3, it is observed that in the LLVM environment unequal slices make the performance of the proposed method worse than for the sequential method, while the table 5.3 shows that the balanced slices (the slices that have approximately the same amount of work) provide noticeable speed-up. The reason for this is that the number of instructions inside one of the extracted slices is very close to the number in the original function body and after adding the producer instructions to the extracted slice this will make the number of instructions in the extracted slice higher than that in the original function. In addition to this reason, there is the time that this slice needs to retrieve data from the buffer. Taken together these two factors will add more execution time to the extracted slice. The same thing happens in the GCC environment and the performance worsens with a high number of iterations.

Determining the perfect length of a slice is a difficult process and from many experiments, it is concluded that there are three parameters that have a big impact on choosing the right one:

- The presence in the slice of instructions with high execution times or the use of math library functions such as sin, cos, etc.
- The presence in the slice of a loop with a large iteration count.
- Slice length – which only matters in the presence of the first two – which should be between a half and three quarters of the length of the original function body.

Another problem with the backward slice is where in some cases most of program will represent one slice. This problem can mitigate against the potential parallelism that can be extracted from function body. As illustrated by Weiser (1983), who showed that the average

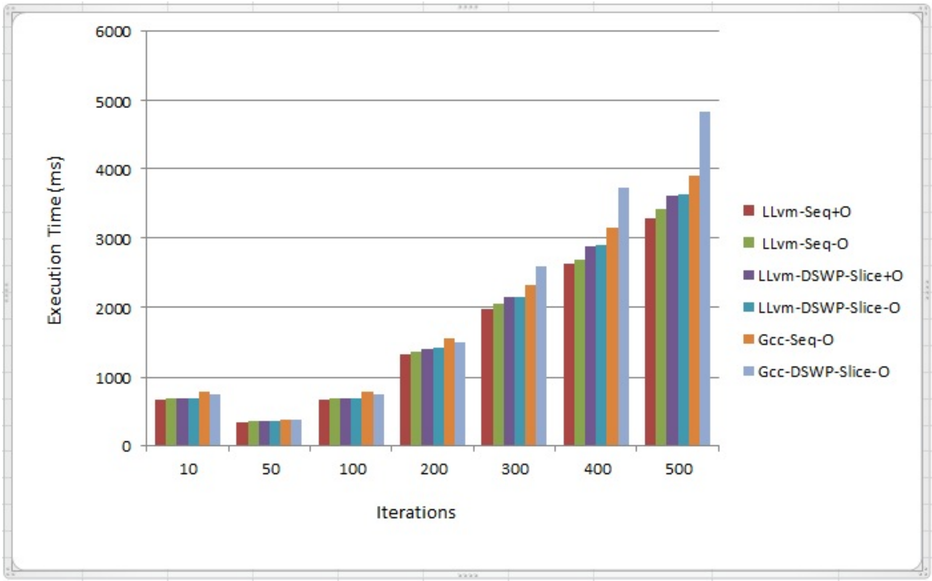


Figure 5-12: Loop speed-up with three threads for `simple4.c` program

length of extracted backward slices for three programs is equal to 58% (compared with the program length) and this percentage will be increased to reach 100% in the worst case.

5.4 Discussion

In this section two issues are discussed, the first one concerns comparing the proposed method with other work and the second relates to studying the effect of inlining on the proposed method.

5.4.1 comparing result

This subsection introduces a comparison between the proposed method and the work that has been presented by Huang et al. (2010), in which they tried to parallelize DSWP long stages using different parallelizing techniques. Moreover, it is demonstrated how this new DSWP/Slice combination provides a new alternative in the cases where the previous methods become inapplicable and also that the approach presented here can increase the scalability of DSWP.

Taking the example in figure 5-13, it can be seen that the loop carried dependency in stage 1 is manifest, while that in stage 3 depends on the input. The aforementioned authors proposed three techniques to parallelize these stages: DOALL, LOCALWRITE and SpecDOALL. Because stage 2 has no loop carried dependency (the existence of loop carried dependency makes the DOALL inapplicable), they used DOALL as a suitable strategy to parallelize stage 2, which comprises the statement `index = calc(node->data, arr[ ])`.

```
node=list->head;
while (node!=NULL){
 index= calc(node->data,arr[]);
 density[index]=update_density
  (density[index],node->data);
  node=node->next;
 }
```

(a) original

```
node=list->head;
while (node!=NULL){
   produce (Q[1,2],node);
   produce (Q[1,3],node);
   node=node->next;
   }
```

(b) Stage 1

```
while(true){
   node = consume (Q[1,2]);
   if(!node) break;
   index= calc(node->data
    ,arr[ ]);
   produce (Q[2,3], index);
 }
```

(c) Stage 2

```
while(true){
   node = consume (Q[1,3]);
   if(!node) break;
   index= consume (Q[2,3]);
   density[index]=update_density
   (density[index],node->data);
 }
```

(d) Stage 3

Figure 5-13: DSWP stages adapted from (Huang et al., 2010)

For stage 3, which comprises the statement `density[index] = update_density(density[index], node->data)`, they chose the LOCALWRITE and SpecDOALL as suitable parallelizing strategies for this stage. LOCALWRITE is a parallelizing method for irregular reduction where the elements that need to be reduced are not located in consecutive order. It works by partitioning the array into blocks and assigning the ownership of each to a different thread. LOCALWRITE has two flaws, which result in this technique performing poorly:

1- In most cases studies in this work, this method require the availability of an array and an ownership checker to be inserted which has to be executed every time a new index is generated and the checker output decides which thread is executed.

2- Redundant computation, which has a significant negative effect on performance

The third method that has been used as an alternative to LOCALWRITE is speculative DOALL. As mentioned earlier, there are two loop carried dependencies in this statement, the first one between array `density[index]` with itself and the second one is `node->data` with the previous one. To start with the statement `density[index]` is executed using SpecDOALL, by giving this stage to two threads and executing both of them using SpecDOALL (each thread executes a different iteration). If the value of the `index` for both iterations is different then there is no problem, and both can be executed speculatively. However the problem arises if the `index` for both iterations has the same values, which means the output of the second thread will be wrong (misspeculation). Furthermore the second thread needs to re-execute after the first one has finished. This scenario looks fine if this type of dependency rarely happens, however, excessive miss-peculation will cause a problem, for the overhead of misspeculation recovery can negate any benefits of parallelization.

Because of all these drawbacks, it is contended that the proposed combination in this thesis is an alternative that is effective in some cases when the existing methods are inapplicable. That is, even with the presence of loop carried dependency, under the technique put forward, if there is enough work inside the function body that covers the time communication overhead then it will be effective.

5.4.2 Inlining effect

Another issue relates to the inlining technique and how performing this on the function body inside the DSWP stages can affect the slicing technique. This effect on the backward slicing can be explained using the example in figure 5.14(a). The main program in this example has two function calls where the `do_cal` function represents the long stage DSWP. After inlining the `do_cal` function inside the second stage DSWP, the body of the called function will be substituted with the actual call to that function and also if the variables named in the called function are distinct from the caller, the compiler needs to rewrite this piece of code so as to capture the semantics of the parameters of the function without altering the original parameters passed to it. Figure 5.14(a) shows the original code before applying inlining and figure 5.14(b) what this loop looks like after applying the inlining technique. From this figure it can be seen that it is difficult to extract three slices after applying inlining. The last statement `k[i] = c[i]+s[i]+d[i]` gathers all arrays, `c[i]`, `s[i]` and `d[i]`, into one statement. So if the intention is to slice `k[i]`, then all three arrays `c[i]`, `s[i]` and `d[i]` will be involved in the extracted slice, but only one slice is extracted from this part.

In addition to these concerns, applying inlining makes the program code bigger and degrades the program performance by increasing the page fault and cache misses (Dr.Dobb's, 2002). From the preceding discussion it can be concluded that applying slicing techniques for DSWP stages in the presence of inlining can reduce the probability of extracting more than one slice from these stages and that therefore inlining and slicing within DSWP does not work well. Moreover, it is observed that all the methods described above (DOALL, SpecDOALL, LOCALWRITE) have their advantages and disadvantages and perform better or worse depending on the characteristics of the program, whereas the DSWP/Slice approach is effective in all of the scenarios explored with the above methods, as long as good slices are available (see section 5.3).

5.5 Summary

This chapter has presented both manually and automatically acquired results from applying the DSWP/Slice combination to a set of case studies. It started by explaining the evolution of this technique according to three aspects. The first pertained to the effect of the Lock-Free buffer technique on the performance of DSWP; this showed that the effect of the software implementation of lock-free buffer will be low if there are large amounts of computations inside the DSWP stages otherwise it will be noticeable. Also the small amount of work can cause poor-performance even with one function argument transfer between DSWP stages and with small amounts of loop iterations. Moreover, the presence of a large amount of work in these stages can gave good performance even with large loop iterations.

Whilst the second showed how the DSWP/Slice technique can improve the performance of long stage DSWP with different program patterns. The amount of speed-up that is obtained for the first five programs on average was ≈ 1.78 while for the last one it was ≈ 1.4 (measured with disabling the LLVM optimization), while for the optimized versions the speed-up, on average, was ≈ 1.45. Finally, the third aspectwas with regards to the effect of buffer size and slice length on the performance of the DSWP/Slice, which shown the buffer size does not have any significant effect on DSWP performance.

Through our experiments that have been done before, it is observed there are three factors that have a big impact on choosing the right slice. These are: (i) the availability of instructions that have high execution time or some of the maths library in the extract slices, (ii) there is a loop inside the extract slice and, (iii) the length of extracted slices have to be between 50% and 75% of the original program length.

Subsequently, the proposed method was compared with the work in (Huang et al., 2010) that tried to parallelize DSWP long stages using different techniques: DOALL, LOCALWRITE and SpecDOALL. It was shown that all these methods can perform better or worse depending on the characteristics of the program, whilst as long as good slices are available, the method presented in this work can deliver good performance.

Finally, the last part of this chapter demonstrated the effect of the inlining technique on the proposed method, where, according to previous discussion, it shown that applying the slicing technique after inlining function body in the long stage DSWP reduces the chances of extracting more than one slice. This means there is no improvement achieved from inlining the function body in this stage.

```
 do_cal(int l1,int m1,
 double c[],double s[],double d[])
.....
 angle=(double)(m1)*phi;
  if(0<=m)
  {
    for(i=0;i<=l1;i++)
    {
      c[i]=plm[i]*cos(angle);
      s[i]=plm[i]*sin(angle);
      d[i]=plm[i]*tan(angle);
    }
  }
  else
  {
    for(i=0;i<=l1;i++)
    {
      c[i]=-plm[i]*cos(angle);
      s[i]=-plm[i]*sin(angle);
      d[i]=plm[i]*tan(angle);
    }
  }

main()
.....
 for(i=0;i<13;i++)
  {
   initial(&n_data,&l,&m);

   if(n_data==0)
   {printf("the n_data equal zero");
       }
   do_cal(l,m,c,s,d);
   k[i]=c[i]+s[i]+d[i];
  }
```

(a) Before inlining

```

/* add and unify the passing
 parameters and the function
 parameters of the function
 entry block to the loop
 parent entry block.  */
------------------------------
main()
.....
//-------- first stage DSWP
 for(i=0;i<13;i++)
  {
   initial(&n_data,&l,&m);
   if(n_data==0)
    {printf("the n_data equal zero");
       }

//-------- second stage DSWP
/* ------ start call function */

  int l,int m;
  double c[],double s[],double d[];
 ......
  angle=(double)(m)*phi;
  if(0<=m)
  {
    for(i=0;i<=l;i++)
    {
      c[i]=plm[i]*cos(angle);
      s[i]=plm[i]*sin(angle);
      d[i]=plm[i]*tan(angle);
    }
  }
  else
  {
    for(i=0;i<=l;i++)
    {
      c[i]=-plm[i]*cos(angle);
      s[i]=-plm[i]*sin(angle);
      d[i]=plm[i]*tan(angle);
    }
/*----- end call function */

   k[i]=c[i]+s[i]+d[i];
  }
}
```

(b) After inlining

Figure 5-14: Sequential version of program before and after inlining

Chapter 6

Conclusion and Future Directions

The multi-core computing system promises to be the most powerful means to deliver more performance in the near future, as the most obvious outcome of increasing transistors density. The correctness of this assumption depends on how well the application can be parallelized. The task of producing an efficient sequential program is challenging and error prone. Although converting a sequential program to a parallel one by the professional programmer can deliver more parallelism, it costs a lot of effort and time. Moreover, programming is already a complex task and expert programmers are not always available. In addition, this solution does not work well with the large body of existing sequential programs. On the other hand, automatic parallelization has proved to be an effective alternative that can reduce the cost and time, whilst at the same guaranteeing the correctness of the resulting code.

This thesis introduces the idea of DSWP applied in conjunction with slicing, by splitting up loops into new loops that are amenable to slicing. Also, it is shown that the DSWP/Slice combination provides a new alternative in cases where the previous methods that have been used in DSWP+ (Huang et al., 2010) such as DOALL, SepcDOALL, etc., are inapplicable.

Furthermore automatic implementation of the proposed method, based on the work that has been carried out by Fuyao Zhao (2011), that is used to alleviate the bad effect of the long stage DSWP using the slicing technique has been introduced. Instead of giving the whole long stage of DSWP that comprise the function body to one thread, it can be distributed across n threads, depending on the number of extracted slices from this stage. In consequence, better load balancing can be achieved between the DSWP stages. This combination is particularly effective when the whole long stage comprises a function body.

An evaluation of this technique for six programs with a range of dependence patterns leads to considerable performance gains on a Core-i7 870 machine with 4-core/8-threads. The results obtained from an automatic implementation that show that the proposed method can give a factor of up to 1.8 speed up compared with the original sequential code.

Moreover, this dissertation investigates the effect of the communication overhead on the performance of DSWP by using Lock-free buffer and it demonstrates that there is inverse proportionality between the amount of computation inside each stage and the communication overhead which becomes insignificant when there is heavy computation inside the DSWP stages. This communication overhead can negate any benefit from DSWP when there is only a small amount of work inside the DSWP stages even with small loop iterations and one function argument transferred between these stages.

Furthermore this study analyses the effect of unequal length slices on the performance of the proposed method where determining the perfect length of a slice is a difficult process and from many experiments it is concluded that there are three factors that have a big impact on choosing the right one namely: (i) the availability of instruction that have high execution time or some of the maths library in the extracted slices, (ii) there is a loop inside the extracted slice, (iii) in the case of extraction of two slices, we have established that the extracted slices should be between 50% and 75% of the original program. The case of the larger numbers of slices and their effective length remain a question for farther research.

Finally, the work that has been carried out by (Fuyao Zhao, 2011) is employed as basis for automating the DSWP/Slice combination. Some adaptations are added to make it a suitable for the proposed method.

From all the above it can be concluded that this thesis has proposed and automated a new combination, designed to improve the performance of the long stage of DSWP. Whilst this work shows that the performance of DSWP alone does not tend to give impressive improvement, however, it shines when the DSWP is coupled with Slicing techniques.

6.1 Future Directions

While the two combination method presented in this dissertation shows significant advantage, however there is much to work to be done in respect of improving the combination to deliver more parallelism. This section discuss some further research directions related to the these methods.

1- Firstly, the aim to increase the potential parallelism that can be extracted from the long stage DSWP. One of major issues with backward slicing is the longest critical path (slice) that creates a limit on parallelism. Insight from (Wang et al., 2009) suggests that parallelism can be increased (number of extracted slices) by combining loop unrolling with backward slicing in the presence of loop carried dependencies as mentioned in section 3.1.8. That is they show that applying loop unrolling to a long critical part of backward slice can break a long dependency cycle into smaller independent cycles where this can increase the usable parallelism. Also, to reduce significantly the length of critical path of the parallel slices, speculation can be exploited to cut rare dependences, and as a result well-designed program transformations can be used to expose parallelism. Based on (Wang et al., 2009) two types of speculation were applied to cut rare dependencies which are memory control and branch prediction speculation . To illustrate how speculation works look at the example in figure 6-1. Without speculation the whole program will construct one slice. However, if *r6 rarely aliases with *r5 then two slices can be extracted as shown

```
1  r1=...
2  r2=...
3  r3=r1+1
4  if(r1>0)
5    r4=r2+1
6  if(r2==0)
7    r4=r4*r3
8  *r5=r3*2
9  r7=*r6+r4
```

(a) original program

```
1   // Slice 1
2   r1=...
3   r3=r1+1
4   *r5=r3*2
5
6   //slice 2
7   r1=...
8   r2=...
9   r3=r1+1
10  if(r1>0)
11    r4=r2+1
12  if(r2==0)
13    r4=r4*r3
14  r7=*r6+r4
```

(b)

```
1  // Slice 1
2   r1=...
3   r3=r1+1
4   *r5=r3*2
5
6   //slice 2
7   r1=...
8   r2=...
9   r3=r1+1
10  if(r1>0)
11    r4=r2+1
12  if(r2==0)
13   rgn exit
14  r7=*r6+r4
```

(c)

```
1  // Slice 1
2   r1=...
3   r3=r1+1
4   IP:
5   if(r1>0)
6   *r5=r3*2
7
8   //slice 2
9   r1=...
10  r2=...
11  r3=r1+1
12  brpred(IP)
13    r4=r2+1
14  if(r2==0)
15   rgn exit
16  r7=*r6+r4
```

(d)

Figure 6-1: Speculation Example (Wang et al., 2009)

in figure 6.1(b). Furthermore if the branch instruction in line 6 executes rarely then this branch will be cut and the instruction will be moved to the exit region as illustrated in figure 6.1(c). Moreover, if the branch instruction in line 4 is highly predictable this instruction can stop being executed and moved from the critical slice as can seen in figure 6.1(d)

2- There is still the need to test the implementation introduced in this work more on complex programs, across multiple platforms. This can reassure the correctness of the automatic implementation for wide range of programs and also providing better comparative analysis between this method that depends on software implementation and other methods that use hardware support to implement the DSWP.

Table 6.1: Third Program Execution Time

Program Name.	Execution type	Time(ms).	No. of thread
Just.c	serial	0.369	1
Dswpjust.c	using DSWP	0.198	3
Justslice.c	using SLICE	0.197	2
DswpjustDoall.c	using DSWP+DOALL	0.108	5
dswpjustDoallSlice.c	(DSWP + DOALL+SLICE)	0.078	9

3- The effect of the memory coherence on the performance of the proposed method needs to be studied. As mentioned in the subsection 3.2.4 the solution that has been introduced by Raman et al. (2008) can be adapted to make two threads work on different cache lines. Moreover the method that has been presented by Chen et al. (2010) could also be employed. To avoid memory traffic, they introduced a cache-friendly approach. This approach communicates the dependency between threads through introducing a C++ template class called QueueBuffer as a lock-free, cache friendly software queue designed for DSWP. Two data members for this class have been declared as order atomic by using the Intel Threading Building Blocks library template class atomic<T>. Chen et al. (2010) try to execute two operations, the first checking of the availability of empty space in the queue and the second writing data in this space, without any interrupts. As a consequence the second thread has no right to access the queue until the first one finishes its work (checking and writing data). Also they try to avoid the false sharing problem by enqueueing or dequeueing multiple data elements at one time that their size in total equal a multiple of the cache-line size.

4- In addition, we can increased the scalability of the proposed method by adding slicing technique to the DSWP stage that has been transformed earlier by DOALL technique. So the new combination will be DSWP+DOALL+Slice. Through our manual experiment we observed that this combination can increase scalability of the DOALL technique and decrease the execution time for each iteration. So from the table 6.1 we get 0.108ms by execution this program using (DSWP + DOALL) and 0.078ms by executed this program using (DSWP + DOALL + Slice).

   However, more analysis and comparison are needed to decide which one of these two cases is better or more effective for a machine that has a limited number of threads :

   1- Increasing number of loop iterations chunks to make it equal to the number of available threads without applying slicing.

   2- Or reducing these loop iteration chunks with application of slicing.

Finally, as the number of cores is continuously increasing, good usages for these cores by the programs are needed. As we mentioned earlier that we believed automatic thread extraction techniques are the best way to reach this goal and therefore that there remains much to be gained from continued exploration of automatic parallelization. In conclusion this dissertation has shown that the combination of slicing with a decoupled software pipeline has significant benefits over either technique alone, and this transformation can be made automatically without user intervention.

Bibliography

Adve, S. V. and Boehm, H.-J. (2010). Memory models: a case for rethinking parallel languages and hardware. *Commun. ACM*, 53(8):90–101.

Agrawal, H. (1994). On Slicing Programs with Jump Statements. In Sarkar, V., Ryder, B. G., and Soffa, M. L., editors, *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 302–312. ACM.

Al Dallal, J. (2005). An Efficient Algorithm for Computing all Program Static Slices. In *Proceedings of the 4th WSEAS International Conference on Software Engineering, Parallel & Distributed Systems*, SEPADS'05, pages 27:1–27:5, Stevens Point, Wisconsin, USA. World Scientific and Engineering Academy and Society (WSEAS).

AMD (2013). AMD Opteron MT Processors Model Numbers. `http://www.amd.com/uk/products/server/processors/Pages/model-numbers.aspx`, retrieved <2013.11.13>.

Anderson, J.-A. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., Leung, S.-T., Sites, R. L., Vandevoorde, M. T., Waldspurger, C. A., and Weihl, W. E. (1997). Continuous Profiling: Where Have All the Cycles Gone? *ACM Trans. Comput. Syst.*, 15(4):357–390.

Appel, A. W. and Palsberg, J. (1998). *Modern Compiler Implementation in C.* Cambridge University Press.

Badger, L. and Weiser, M. (1988). Minimizing Communication for Synchronizing Parallel Dataflow Programs. In *International Conference of Parallel Processing (2)*, pages 122–126. IEEE Computer Society.

Binkley, D. and Gallagher, K. B. (1996). Program Slicing. *Advances in Computers, San Diego, California, Academic Press*, 43:1–50.

Bischof, C., an Mey, D., and Iwainsky, C. (2012). Brainware for green HPC. *Computer Science — Research and Development*, 27(4):227–233.

Bliss, N. (2007). Addressing the Multicore Trend with Automatic Parallelization. *Lincoln Laboratory Journal*, 17(1):187–198.

Blume, W., Eigenmann, R., Faigin, K., Grout, J., Hoeflinger, J., Padua, D., Petersen, P., Pottenger, B., Rauchwerger, L., Tu, P., et al. (1994). Polaris: The Next Generation in Parallelizing Compilers. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, pages 141–154. Springer-Verlag,Berlin/Heidelberg.

Brandis, M. M. and Mössenböck, H. (1994). Single-pass Generation of Static Single-Assignment Form for Structured Languages. *ACM Trans. Program. Lang. Syst.*, 16(6):1684–1698.

Bridges, M. J. (2008). *The VELOCITY Compiler: Extracting Efficient Multicore Execution from Legacy Sequential Code*. PhD thesis, Department of Computer Science, Princeton University, New Jersey, United States. `ftp://ftp.cs.princeton.edu/techreports/2008/835.pdf`, retrieved <2012.02.20>.

Brown, A. and Wilson, G. (2008). *The Architecture of Open Source Applications Structure, Scale, and a Few More Fearless Hacks*, volume 2. lulu.com. `http://www.worldcat.org/isbn/9781105571817`, retrieved <2011.12.17>.

Burkardt, J. (2012). C Source Codes. `http://people.sc.fsu.edu/~jburkardt/c_src/c_src.html`, retrieved <2012.11.25>.

Canfora, G., Cimitile, A., Lucia, A. D., and Lucca, G. A. D. (1994). Software Salvaging Based on Conditions. In *Proceedings of the International Conference on Software Maintenance*, ICSM '94, pages 424–433, Washington, DC, USA. IEEE Computer Society.

Carlstrom, B. D., McDonald, A., Chafi, H., Chung, J., Minh, C. C., Kozyrakis, C., and Olukotun, K. (2006). The Atomos Transactional Programming Language. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 1–13, New York, NY, USA. ACM.

Chen, T. Y. and Cheung, Y. Y. (1993). Dynamic Program Dicing. In Card, D. N., editor, *ICSM*, pages 378–385. IEEE Computer Society.

Chen, W. R., Yang, W., and Hsu, W. C. (2010). A lock-free cache-friendly software queue buffer for decoupled software pipelining. In *Computer Symposium (ICS), 2010 International*, pages 997–1006. IEEE.

Choi, J.-D. and Ferrante, J. (1994). Static Slicing in the Presence of Goto Statements. *ACM Trans. Program. Lang. Syst.*, 16(4):1097–1113.

Chu, M. L. and Mahlke, S. A. (2007). Code and Data Partitioning for Fine-grain Parallelism. In Pande, S. and Li, Z., editors, *Languages, Compilers, Tools and Theory for Embeded System*, pages 161–164. ACM.

Davies, J. R. B. (1981). Parallel Loop Constructs for Multiprocessors. Master's thesis, University of Illinois at Urbana-Champaign.

Dr.Dobb's (2002). The New C:Inline Functions. http://www.drdobbs.com/the-new-c-functions/184401540.

Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The Program Dependence Graph and Its use in Optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349.

Frigo, M., Leiserson, C. E., and Randall, K. H. (1998). The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA. ACM.

Fuyao Zhao, M. H. (2011). Decoupled Software Pipelining in LLVM. Technical Report 15-745 Final Project, Carnegie Mellon University. http://www.cs.cmu.edu/~fuyaoz/courses/15745/report.pdf, retrieved < 2012.01.26>.

Gallagher, K. B. and Lyle, J. R. (1991). Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering*, 17:751–761. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.39.1532&rep=rep1&type=pdf, retrieved <2013.03.12>.

Giacomoni, J., Moseley, T., and Vachharajani, M. (2008). FastForward for Efficient Pipeline Parallelism: a cache-optimized concurrent lock-free queue. In Chatterjee, S. and Scott, M. L., editors, *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 43–52. ACM.

Giffhorn, D. (2009). Chopping Concurrent Programs. In *9th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 13–22. IEEE.

Grosser, T. C. (2011). Enabling Polyhedral Optimizations in LLVM. http://polly.llvm.org/publications/grosser-diploma-thesis.pdf, retrieved <2013.06.23>.

Gupta, R., Soffa, M. L., and Howard, J. (1997). Hybrid Slicing: Integrating Dynamic Information with Static Analysis. *ACM Trans. Softw. Eng. Methodol.*, 6(4):370–397.

Hall, M. W., Amarasinghe, S. P., Murphy, B. R., Liao, S.-W., and Lam, M. S. (2005). Interprocedural Parallelization Analysis in SUIF. *ACM Trans. Program. Lang. Syst.*, 27(4):662–731.

Han, W. (2010). *Multi-core Architectures with Coarse-grained Dynamically Reconfigurable Processors for Broadband Wireless Access Technologies.* PhD thesis, The University of Edinburgh. https://www.era.lib.ed.ac.uk/bitstream/1842/3812/1/Han2010.pdf , retrieved <2012.10.05>.

Harman, M. and Hierons, R. M. (2001). An Overview of Program Slicing. *Software Focus*, 2(3):85–92. http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=981AD210AC5185425E386BE982F1674D?doi=10.1.1.22.8393&rep=rep1&type=pdf, retrieved<2011.03.12>.

Hiranandani, S., Kennedy, K., and Tseng, C.-W. (1993). Preliminary Experiences with Fortran D Compiler. In *Supercomputing '93. Proceedings*, pages 338–350. IEEE.

Horwitz, S., Prins, J., and Reps, T. (1989). Integrating Noninterfering Versions of Programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387.

Horwitz, S., Reps, T., and Binkley, D. (1990). Interprocedural Slicing using Dependence Graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60.

Huang, J., Raman, A., Jablin, T. B., Zhang, Y., Hung, T.-H., and August, D. I. (2010). Decoupled Software Pipelining Creates Parallelization Opportunities. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 121–130, New York, NY, USA. ACM.

IBM (2013). Processor Value Unit [PVU] licensing for Distributed Software. `http://www-01.ibm.com/software/lotus/passportadvantage/pvu_licensing_for_customers.html`, retrieved <2013.11.23>.

Intel-Corporation (2002). PRESS KIT Moore's Law 40th Anniversary. `http://www.intel.com/pressroom/kits/events/moores_law_40th`, retrieved<2013.07.22>.

Jeon, D. (2009). Compiler Parallelization Techniques for Tiled Multicore Processors. `http://cseweb.ucsd.edu/~djeon/re_dhjeon.pdf`, retrieved <2011.06.22>.

Kennedy, K. and Allen, J. R. (2002). *Optimizing Compilers for Modern Architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Kim, H.-S., Yoon, Y.-H., Na, S.-O., and Han, D.-S. (2000). ICU-PFC: An Automatic Parallelizing Compiler. In *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, volume 1, pages 243–246. IEEE.

Korel, B. and Yalamanchili, S. (1994). Forward Computation of Dynamic Program Slices. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '94, pages 66–79, New York, NY, USA. ACM.

Kowshik, S., Dhurjati, D., and Adve, V. (2002). Ensuring code safety without runtime checks for real-time control systems. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '02, pages 288–297, New York, NY, USA. ACM.

Krinke, J. (2003). Barrier Slicing and Chopping. In *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*, pages 81–87. IEEE Computer Society.

Kumar, L. (2001). Software Comprehension and Program Slicing. Master's thesis, Concordia University. `http://spectrum.library.concordia.ca/1662/1/MQ68469.pdf`, retrieved <2013.12.01>.

Lakhotia, A. (1992). Improved Interprocedural Slicing Algorithm. Technical Report CACS TR-92-5-8, University of Southwestern Louisiana. http://www.cacs.louisiana.edu/~arun/papers/TR-92-5-8.pdf, retrieved <2012.10.06>.

Lattner, C. (2002). LLVM : An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL. http://llvm.cs.uiuc.edu, retrieved <2011.07.16>.

Lattner, C. and Adve, V. (2002). The LLVM Instruction Set and Compilation Strategy. Technical Report UIUCDCS-R-2002-2292, University of Illinois at Urbana-Champaign, Computer Science Department. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.58.40&rep=rep1&type=pdf,retrieved <2012.05.07>.

Lattner, C. and Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–86, Palo Alto, California. IEEE Computer Society.

Li, F., Pop, A., and Cohen, A. (2011). Advances in Parallel-Stage Decoupled Software Pipelining Leveraging Loop Distribution, Stream-Computing and the SSA Form. In *Proceedings of the Workshop on Intermediate Representations*, pages 29–36, Chamonix, France. Florent Bouchez and Sebastian Hack and Eelco Visser.

LLVM Project (2012). Programmer's Manual. http://llvm.org/docs/ProgrammersManual.html, retrieved <2012.1.07>.

LLVM Project (Last updated on 2013). LLVM Alias Analysis Infrastructure. http://llvm.org/docs/AliasAnalysis.html, retrieved <2012.05.09>.

Loveman, D. B. (1993). High Performance Fortran. *IEEE Parallel Distrib. Technol.*, 1(1):25–42.

Lucia, A. D. (2001). Program Slicing: Methods and Applications. In *SCAM*, pages 144–151. IEEE Computer Society.

Mason, T. R., David, A., and August, I. (2009). LAMPVIEW: A Loop-Aware Toolset for Facilitating Parallelization. Master's thesis, Princeton University, New Jersey, United States. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.155.3358&rep=rep1&type=pdf, retrieved <2012.11.17>.

MPI (2013). The Message Passing Interface (MPI) Standard. http://www-unix.mcs.anl.gov/mpi, retrieved <2012.08.23>.

Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Neustifter, A. (2010). Efficient Profiling in the LLVM Compiler Infrastructure. Master's thesis, TU Wein. http://llvm.org/pubs/2010-04-NeustifterProfiling.pdf, retrieved <2012.11.15>.

Ning, J. Q., Engberts, A., and Kozaczynski, W. V. (1994). Automated Support for Legacy Code Understanding. *Commun. ACM*, 37(5):50–57.

OpenMP (2013). The OpenMP API Specification for Parallel Programming. `http://www.openmp.org`, retrieved <2013.08.12>.

Ottenstein, K. J. and Ottenstein, L. M. (1984). The Program Dependence Graph in a Software Development Environment. In Riddle, W. E. and Henderson, P. B., editors, *Software Development Environments (SDE)*, pages 177–184. ACM.

Ottoni, G., Rangan, R., Stoler, A., and August, D. I. (2005). Automatic Thread Extraction with Decoupled Software Pipelining. In *ACM International Symposium on Microarchitecture*, pages 105–118. IEEE Computer Society.

Pang, T. (1997). *An Introduction to Computational Physics*. Cambridge University Press.

Raman, E. (2009). *Parallelization Techniques with Improved Dependence Handling*. PhD thesis, Princeton University. Dept. of Computer Science. `http://liberty.princeton.edu/Publications/phdthesis_eraman.pdf`, retrieved <2013.02.27>.

Raman, E., Ottoni, G., Raman, A., Bridges, M. J., and August, D. I. (2008). Parallel-stage Decoupled Software Pipelining. In Soffa, M. L. and Duesterwald, E., editors, *Code Generation and Optimization*, pages 114–123. ACM.

Rangan, R., Vachharajani, N., Ottoni, G., and August, D. I. (2008). Performance Scalability of Decoupled Software Pipelining. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(2).

Rangan, R., Vachharajani, N., Vachharajani, M., and August, D. I. (2004). Decoupled Software Pipelining with the Synchronization Array. In *IEEE PACT*, pages 177–188. IEEE Computer Society.

Rong, H., Tang, Z., Govindarajan, R., Douillet, A., and Gao, G. R. (2007). Single-dimension Software Pipelining for Multidimensional Loops. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(1).

Silva, J. (2012). A Vocabulary of Program Slicing-based Techniques. *ACM Comput. Surv.*, 44(3):12:1–12:41.

SPEC (2013). Standard Performance Evaluation Corporation (SPEC). `http://www.spec.org/`, retrieved<2013.02.11>.

Spracklen, L. and Abraham, S. G. (2005). Chip Multithreading: Opportunities and Challenges. In *HPCA*, pages 248–252. IEEE Computer Society.

Sreedhar, V. C., Ju, R. D.-C., Gillies, D. M., and Santhanam, V. (1999). Translating Out of Static Single Assignment Form. In Cortesi, A. and Filé, G., editors, *SAS*, volume 1694 of *Lecture Notes in Computer Science*, pages 194–210. Springer.

Srinivasan, H. and Grunwald, D. (1991). An Efficient Construction of Parallel Static Single Assignment Form for Structured Parallel Programs. Technical report, University of Colorado at Boulder. `http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=3977DCE3E2BC8969AFC4130FA58A2B99?doi=10.1.1.48.2778&rep=rep1&type=pdf`, retrieved <2012.07.03>.

Thies, W., Karczmarek, M., and Amarasinghe, S. P. (2002). StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, UK. Springer-Verlag.

Tip, F. (1995). A Survey of Program Slicing Techniques. *Journal of programming languages*, 3(3):121–189. `http://www.cse.buffalo.edu/LRG/CSE605/Papers/slicing-survey-tip.pdf`, retrieved <2011.08.15>.

Tournavitis, G., Wang, Z., Franke, B., and O'Boyle, M. F. (2009). Towards a Holistic Approach to Auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 177–187, New York, NY, USA. ACM.

Vachharajani, N., Rangan, R., Raman, E., Bridges, M. J., Ottoni, G., and August, D. I. (2007). Speculative Decoupled Software Pipelining. In *Parallel Architecture and Compilation Techniques*, pages 49–59. IEEE.

Vachharajani, N. A. (2008). *Intelleigent Speculation for Pipelined Multithreading*. PhD thesis, Departement of Computer Science, Princeton University. `http://liberty.princeton.edu/Publications/phdthesis_nvachhar.pdf`, retrieved <2012.11.08>.

Venu, B. (2011). Multi-core Processors - An Overview. *CoRR*, abs/1110.3535. `http://arxiv.org/ftp/arxiv/papers/1110/1110.3535.pdf`, retrieved <2012.02.25>.

Wang, C., Wu, Y., Borin, E., Hu, S., Liu, W., Sager, D., Ngai, T.-f., and Fang, J. (2009). Dynamic Parallelization of Single-threaded Binary Programs using Speculative Slicing. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 158–168, New York, NY, USA. ACM.

Weiser, M. (1983). Reconstructing Sequential Behavior from Parallel Behavior Projections. *Inf. Process. Lett.*, 17(3):129–135.

Weiser, M. (1984). Program Slicing. *IEEE Trans. Software Eng.*, 10(4):352–357.

Wikipedia (2011). `http://en.wikipedia.org/wiki/Moore%27s_law`, retrieved <2014.04.02>.

Xu, B., Qian, J., Zhang, X., Wu, Z., and Chen, L. (2005). A Brief Survey of Program Slicing. *ACM SIGSOFT Softw. Eng. Notes*, 30(2):1–36.

Ye, G. (2011). Getting to know the LLVM Compiler. Master's thesis, University of Edinburgh. `http://www.epcc.ed.ac.uk/sites/default/files/Dissertations/2010-2011/GuobinYe.pdf`, retrieved <2012.09.10>.

Zhang, Y., Ootsu, K., Yokota, T., and Baba, T. (2008). Clustered Decoupled Software Pipelining on Commodity CMP. In *Parallel and Distributed Systems, 2008. ICPADS '08. 14th IEEE International Conference on*, pages 681–688.

Zhao, J. and Rinard, M. (2003). System Dependence Graph Construction for Aspect-Oriented Programs. Technical Report MIT-LCS-TR-891, Laboratory for Computer Science Massachusetts Institute of Technology 200 Technology Square, Cambridge, MA 02139, USA. `http://cse.sjtu.edu.cn/~zhao/pub/pdf/mit-lcs-tr-891.pdf`, retrieved <2012.03.20>.

Zhong, H. (2008). *Architectural and Compiler Mechanisms for Accelerating Single Thread Applications on Multicore Processors*. PhD thesis, Computer Science and Engineering in The University of Michigan. `http://cccp.eecs.umich.edu/theses/hongtaoz-thesis.pdf`, retrieved <2011.05.22>.

Zhong, H., Lieberman, S. A., and Mahlke, S. A. (2007). Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications. In *HPCA*, pages 25–36. IEEE Computer Society.

Zilles, C. B. and Sohi, G. S. (2000). Understanding the Backward Slices of Performance Degrading Instructions. In Berenbaum, A. D. and Emer, J. S., editors, *27th International Symposium on Computer Architecture (ISCA)*, pages 172–181. IEEE Computer Society.

Appendix A

Programs

1- **simple.c program**

```
fun(int d) {
   int a1[1000000],b[1000000];
   int i;
   a1[0]=0;
   b[0] = 200;
   for (i = 1; i < d; i++) {
       a1[i-1]=cos(i);
       a1[i] = a1[i-1]+((i*i)/100)*2;
       a1[i]=a1[i]+ sin(i);
       b[i-1]=sin(i);
       b[i] = b[i-1]+ cos(i)*2;
      }
 }

 int main() {
   int res,x,i,m,k;
   long m1=0;
   for(i=1;i<100;i++)
   {x=x+i;
     }
   for(m=10000;m<10020;m++)
    {
      for(k=1;k<2000;k++)
        { m1=0;
          m1=k+ sin(k);}
       fun(m);
     }
   return 0;
 }
```

2- **linkedlist2.c**

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
struct _list1
{
int data;
struct _list1 *next1;
};
typedef struct _list1 list1;
struct _reco {
int data;
list1 *right ;
struct _reco *next;
};
typedef struct _reco reco;
reco *p,*q, *p1,*p2;
reco * makelist(int x)
{
 int i;
  p=NULL;
  p1=malloc(sizeof(reco));
  p1->data=0;
  p=p1;
  for(i=1;i<x;i++)
     {
      p2=malloc(sizeof(reco));
      p2->data=i;
      p1->next=p2;
      p2->next=NULL;
      p2->right=NULL;
      p1=p2;
    }
return p;
}
```

```

list1 * makelist2(int x)
{
  list1 *pw,*pw1,*pw2;
 int i;
  pw=NULL;
  pw1=malloc(sizeof(list1));
  pw1->data=1;
  pw=pw1;
  for(i=1;i<x;i++)
     {
      pw2=malloc(sizeof(list1));
      pw2->data=i+1;
      pw1->next1=pw2;
      pw2->next1=NULL;
      pw1=pw2;
    }
return pw;
}

 void calculate(reco *d)
{
list1 *p11;
double a[100000],a1[100000];
  int i,vll;
   double sum=0;
   double sum1=0;
  p11=d->right;
  int k=0;
  while (p11!=NULL)
   {
     if(p11->data >0)
      {
        for (i=1;i<p11->data;i++)
          {
            sum=sum+sin(i);
            sum1=sum1;//+cos(i);
          }
      }
    a[k]=sum;
    a1[k]=sum1;
    sum=0;
    sum1=0;
    k=k+1;
    p11=p11->next1;
    }
  }
int main()
{
 int x,z;
 x=5;
 list1 *w;
 p=makelist(x);
 q=p;
 z=1000;
 while(q!= NULL)
   {
     w=makelist2(z);
     q->right=w;
     q=q->next;
   }
 int k1=0;
 while (p!= NULL) // main loop
   {
    calculate(p);
    p=p->next;
   }
}
```

3- **linkedlist3.c program**

```

#include<stdio.h>
struct _list1
{
int data;
struct _list1 *next1;
};
typedef struct _list1 list1;
struct _reco {
int data;
```

```
list1 *right ;
struct _reco *next;
};
typedef struct _reco reco;

reco *p,*q, *p1,*p2;
reco * makelist(int x)
{
 int i;
  p=NULL;
  p1=malloc(sizeof(reco));
  p1->data=1;
  p=p1;
  for(i=1;i<x;i++)
     {
      p2=malloc(sizeof(reco));
      p2->data=i;
      p1->next=p2;
      p2->next=NULL;
      p2->right=NULL;
      p1=p2;
    }
return p;
}
 double calculate(reco *d)
{
list1 *p11;
double a[20000],a1[20000];
 int i;
 double sum=0;
 double sum1=0;
 p11=d->right;
 int k=0;
 while (p11!=NULL)
   {
      if(p11->data >0)
      {
       for (i=1;i<p11->data;i++)
         {
          sum=sum+sin(i);
          sum1=sum1+cos(i);
         }
     a[k]=sum;
     a1[k]=sum1;
     sum=0;
     sum1=0;
     k=k+1;
    }
  p11=p11->next1;
  }
  return sum;
  }

list1 * makelist2(int x)
{
  list1 *pw,*pw1,*pw2;
 int i;
  pw=NULL;
  pw1=malloc(sizeof(list1));
  pw1->data=1;
  pw=pw1;
  for(i=1;i<x;i++)
     {
      pw2=malloc(sizeof(list1));
      pw2->data=i;
      pw1->next1=pw2;
      pw2->next1=NULL;
      pw1=pw2;
    }
return pw;
}


int main()
{
  int x,z;
double cal;
  x=5;
  list1 *w;
  p=makelist(x);
  q=p;
  z=1000;
  while(q!= NULL)
   {
     w=makelist2(z);
     q->right=w;
     q=q->next;
    }
 double k1=0;
 while (p->next!= NULL) // main loop
   {
    cal=calculate(p);
    p=p->next;
    k1=k1+cal;
   }
}
```

4- fft.c program

```
#include <stdio.h>
#include <math.h>
#define NMAX 1024
#define MMAX  8
int n = NMAX;
int m = MMAX;
double fr[100][NMAX],fi[100][NMAX];
double f0,h,rn;
 dft (int x){
int i,j;
double pi,x1,q;
double gr[NMAX],gi[NMAX];
pi =4*atan(1);
x1  =2*pi/2056;
for (i = 0; i < n; ++i)
 {
  gr[i] = 0;
  gi[i] = 0;
  for (j = 0; j < n; ++j)
    {
     q = x1*j*i;
     gr[i] = gr[i]+fr[x][j]*cos(q)+fi[x][j]*sin(q);
     gi[i] = gi[i]+fi[x][j]*cos(q)-fr[x][j]*sin(q);
    }
   gr[i] = f0*gr[i];
   gi[i] = f0*gi[i];
 }
}

main(){
int i,x,x1,i1;
double xm;
h = 1.0/(n-1);
rn = n;
f0 = 1/sqrt(rn);
for(x=0;x<15;x++)
  {
   for (i = 0; i < n; ++i)
    {
     xm = h*x;
     fr[x][i] = sin(xm)*(1-xm);
     fi[x][i] = 0;
    }
   dft (x);
 }
}
```

5- **pro-2.4.c program**

```
#include <stdio.h>
#include <math.h>
#define NMAX 100000
#define M 25
int n,m;
double pi,h;
double x[M][NMAX],f[M][NMAX],f1[M][NMAX],f2[M][NMAX];

 three(int j){
 double d1[NMAX],d2[NMAX];
 int i;
for (i = 1; i < n-1; ++i)
  {
  f1[j][i] = (f[j][i+1]-f[j][i-1])/(2*h);
  f2[j][i] = (f[j][i+1]-2*f[j][i]+f[j][i-1])/(h*h);
  }
f1[j][0]   = 2*f1[j][1]-f1[j][2];
f1[j][n-1] = 2*f1[j][n-2]-f1[j][n-3];
f2[j][0]   = 2*f2[j][1]-f2[j][2];
f2[j][n-1] = 2*f2[j][n-2]-f2[j][n-3];
for (i=0; i < n; ++i)
  {
  d1[i] = f1[j][i]-cos(x[j][i]);
  d2[i] = f2[j][i]+sin(x[j][i]);
  }
}

main(){
pi = 4*atan(1);
n = NMAX;
m =M;
h  = pi/(2*(n-1));
int i,j;
for (j=0; j < m; ++j)
{
  for (i=0; i < n; ++i)
     {
      x[j][i]  = h*i;
      f[j][i]  = sin(x[j][i]);
      f1[j][i] = 0;
      f2[j][i] = 0;
     }
   three(j);
   }
}
```

6- **test0697.c program**

```

#include <stdio.h>
#include <math.h>
#define N_MAX 15
#define MAX1 500000
double phi_vec[N_MAX];
double theta_vec[N_MAX] ;
double yi_vec[N_MAX] ;
double yr_vec[N_MAX];
int l_vec[N_MAX];
int m_vec[N_MAX];
double k=0.5;
void spherical_harmonic_values( int *n_data, int *l,
 int *m,double *theta,  double *phi, double *yr, double *yi )
{
int j;
  if ( *n_data < 0 )
  {*n_data = 0;}
  *n_data = *n_data + 1;
  if ( N_MAX < *n_data )
  { *n_data = 0;
    *l = 0;
    *m = 0;
    *theta = 0.0;
    *phi = 0.0;
    *yr = 0.0;
    *yi = 0.0;
  }
  else
   { *l = l_vec[*n_data-1];
    *m = m_vec[*n_data-1];
    *theta =theta_vec[*n_data-1];
    *phi = phi_vec[*n_data-1];
    *yr = yr_vec[*n_data-1];
    *yi = yi_vec[*n_data-1];
  }
  return;
}
void spherical_harmonic ( int l, int m,
double theta , double phi )
{
 double c[MAX1+1],s[MAX1+1];
 double angle;
 int i;
 int m_abs;
 double *plm;
 double x,factor;
 int mm;
 double pi = 3.141592653589793;
 double somx2;
 double value,value1;
 m_abs = abs ( m );
 x=cos(theta);
 plm = ( double * ) malloc ( ( l + 1 ) * sizeof ( double ) );
 if ( m_abs < 0 )
  {
    //fprintf ( stderr, "\n" );
   // fprintf ( stderr, "LEGENDRE_ASSOCIATED_NORMALIZED
   //  - Fatal error!\n" );
   // fprintf ( stderr, "  Input value of M is %d\n", m );
    //fprintf ( stderr, "  but M must be nonnegative.\n" );
   exit ( 1 );
 }

if ( l <m_abs )
 {
   // fprintf ( stderr, "\n" );
   // fprintf ( stderr, "LEGENDRE_ASSOCIATED_NORMALIZED
   // - Fatal error!\n" );
   // fprintf ( stderr," Input value of M = %d\n", m );
   // fprintf ( stderr," Input value of N = %d\n", n );
   // fprintf ( stderr,"  but M must be less than or
   // equal to N.\n" );
 exit ( 1 );
 }
if ( x < -1.0 )
  {
   // fprintf ( stderr, "\n" );
   // fprintf ( stderr, "LEGENDRE_ASSOCIATED_NORMALIZED
   //- Fatal error!\n" );
   // fprintf ( stderr,"  Input value of X = %f\n", x );
   // fprintf ( stderr," but X must be no less than -1.\n" );
    exit ( 1 );
  }
 if ( 1.0 < x )
  {
   // fprintf ( stderr, "\n" );
   // fprintf ( stderr, "LEGENDRE_ASSOCIATED_NORMALIZED
   // - Fatal error!\n" );
   // fprintf ( stderr, "  Input value of X = %f\n", x );
   // fprintf ( stderr," but X must be no more than 1.\n" );
    exit ( 1 );
  }
 for ( i = 0; i <= m_abs-1; i++ )
  {
    plm[i] = 0.0;
  }
  somx2 = sqrt ( 1.0 - x * x );
  factor = 1.0;
 for ( i = 1; i <=m_abs; i++ )
  {
    plm[m_abs] = -plm[m_abs] * factor * somx2;
    factor = factor + 2.0;
  }

 for ( i =  m_abs+2; i <= l; i++ )
  {
    plm[i] = ( ( double ) ( 2 * i  - 1 ) * x * plm[i-1]
            + ( double ) ( - i - m_abs + 1 ) * plm[i-2] )
            / ( double ) (  i -  m_abs  );
  }
/*
  Normalization.
*/
 value=1;value1=1;
for ( i = 1; i <= l; i++ )
    value = value * ( double )(i);

for ( mm = m_abs; mm <= l; mm++ )
  {
     for ( i = 1; i <= mm + m_abs ; i++ )
       value = value * ( double ) ( i );

   for ( i = 1; i <= mm - m_abs ; i++ )
    value = value * ( double ) ( i );

  for ( i = 1; i <= mm-m_abs ; i++ )
   value1 = value1 * ( double ) ( i );

   factor = sqrt ( ( ( double ) ( 2 * mm + 1 )
  * value1 )/( 4.0 * pi *value));
  plm[mm] = plm[mm] * factor;
  }
  angle = (double)(m) * phi;
  if ( 0 <= m )
  {
    for ( i = 0; i <= l; i++ )
    {
      c[i] = plm[i] * cos ( angle );
      s[i] = plm[i] * sin ( angle );
    }
  }
  else
  {
    for ( i = 0; i <= l; i++ )
    {
      c[i] = -plm[i] * cos ( angle );
      s[i] = -plm[i] * sin ( angle );
    }
  }
  free ( plm );
  return;
}
void main ( )
{
double theta, phi;
int m;
int n_data;
  int l,i;
  double yi;
  double yi2;
  double yr;
  double yr2;
  int j;
for(j=0;j<N_MAX;j++){
 l_vec[j]=400000;
 m_vec[j]=400000;
}
for(j=0;j<N_MAX;j++){
 phi_vec[j] =0.7;//random(k);
 theta_vec[j] =0.1;//random(k);
 yi_vec[j] =0.2;// random(k);
 yr_vec[j] =0.2;//random(k);
}
  n_data = 1;
  for (i=0 ;i<13;i++ )
  {
    spherical_harmonic_values (&n_data,
    &l, &m, &theta, &phi, &yr, &yi );
    if ( n_data == 0 )
    {
   break;
    }
    spherical_harmonic( l,m,theta,phi);
  }
  return;
}
```

Appendix B

Instructions Latency

This appendix contains the table of the instructions latency that we have use in the implementation chapter. This table adapted from the work that is presented by Zhao et al. (Fuyao Zhao, 2011). We keep this table as it is because of the final results will not be effected if we change it with the Core i7 instructions latency table.

Table B.1: Instruction Latency / part 1. Adapted from (Fuyao Zhao, 2011)

Inst. Name	Inst. Latency
Return	1
Branc	0
Switch	0
Add	1
FAdd	4
Sub	1
FSub	4
Mul	3
FMul	4
UDiv	17
SDiv	17
FDiv	24
URem	17
SRem	17
FRem	24
Shift left	7
Shift right	7
Shifr right	7

Table B.2: Instruction Latency / part 2. Adapted from (Fuyao Zhao, 2011)

Inst. Name	Inst. Latency
And	1
Or	1
Xor	1
load	2
Store	2
GetElementPtr	1
Truncate integers	1
Zero extend integers	1
Sign extend integers	1
FPtoUI	4
FPtoSI	4
UItoFP	4
SItoFP	4
FPTrunc	4
FPExt	4
PtrToInt	2
IntToPtr	2
Type cast	1
Integer compare	1
Float compare	1
PHI node	1
Call function (variable)	50