Compile-Time Characterization of Recurrent Patterns in Irregular Computations

Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science at the Graduate School of The Ohio State University

By

Arjun Jagadeesh Singri

Graduate Program in Computer Science and Engineering

The Ohio State University

2010

Thesis Committee

P. Sadayappan, Advisor

Atanas Rountev, Co-Advisor

Abstract

There are compiler techniques using which efficient communication statements can be generated with user supplied data-distributions. These techniques work for regular array access functions but many applications use irregular access functions. It is difficult to generate efficient communication statements for irregular accesses as compile time characterization of the computation structure of such applications is infeasible. In many applications, the irregular computational patterns recur a number of times during execution. This recurring section of an application can be analyzed once at run time to determine its computation structure and then the information collected can be used to generate an efficient communication schedule for it. This model of compiling a section of a program is an example of Inspector-Executor compilation. This thesis aims at identifying such sections in a program automatically at compile-time. The algorithms needed for this are implemented using the Low Level Virtual Machine (LLVM) compiler framework and are used to detect such sections in the SPEC benchmarks. The relative running time of all such sections is calculated with respect to that of the entire program by using HPCToolkit.

Acknowledgements

I would like to thank my advisors, Dr. Sadayappan and Dr. Atanas Rountev for allowing me to work on this project which has helped me get a better understanding of program analysis and its application in different areas.

I would like to thank my parents and my brother for their continuous support throughout my stay at Ohio State University.

Vita

Sept 2001 – June 2005……..Bachelor of Engineering

Computer Science and Engineering,

R V College of Engineering,

Bangalore, India

June 2005 – August 2006……..National Instruments, Bangalore India

August 2006 – January 2007……..SAP Labs India, Bangalore, India

January 2007 – July 2008……..National Instruments, Bangalore, India

Fields of Study

Major Field: Computer Science

Contents

List of Figures

Figure **Page**

List of Tables

Table **Page**

Chapter 1

Introduction

Modern multiprocessors have the potential to significantly speed up scientific and engineering applications. But writing parallel programs using the message passing paradigm is currently impeding this. With the help of user specified data-distributions, a compiler can automatically generate communication statements needed for parallelization in shared-space programming models (5, 8). Though there are compiler techniques for generating efficient communication for programs with regular access functions (6, 7, 9), many applications use irregular access functions and compile time characterization of the computation structure of such applications is infeasible which makes it difficult to generate efficient communication statements. The paper by Eswar et al., [2] addresses an important issue pertaining to efficient parallel implementation of scientific and engineering computations with irregular array accesses. In many engineering applications, the irregular computational patterns recur a number of times during the execution [4, 10]. This recurring section of an application can be analyzed once at run time to determine its computation structure and then use the information collected to generate an efficient communication schedule for it. This model of compiling a section of a

program is an example of Inspector-Executor compilation. The analysis is performed by the Inspector and the execution by the Executor. In this context, the recognition of sections of programs with repetitive computation patterns is clearly very important.

The Inspector-Executor model has been used in PARTI [3] which consists of a set of primitives that can be embedded in distributed-memory parallel programs which help in determining the communication required for arrays used in a parallel loop. The ARF compiler [11] can also generate Inspectors and Executors with help from user for specifying the parallel loops and their data distributions. PARTI primitives are embedded in the generated Inspectors and Executors.

This thesis addresses the issue of automatic compile-time identification of sections of a program that exhibit recurring computational patterns using the algorithm from [2]. The relative execution time for all such sections with respect to that of the whole program is shown for each of the SPEC 2000 and 2006 benchmarks.

1.1 Inspector Executor Model

In order to parallelize certain loops, enough information must be available about the required elements belonging to the loop body so that they can be sent to different processors. Some of this information may not be available at compiler time, but it can be obtained at run time by emulating one iteration of the loop body and information obtained can then be used to actually execute the loop in parallel. The code section that emulates the loop body to

find the information is called the Inspector and the code section that actually executes the loop body with the information supplied by the Inspector is called the Executor. This model of executing programs is called the Inspector-Executor (IE) model and the code sections are called IE-sections. Eswar et al., [2] deals with identification of sections of a program for which the Inspector-Executor model can be applied. The algorithm finds all loop bodies whose statement dependence graph remains invariant in a given execution of that loop. Such loop bodies are called IE/L-sections. Since the statement dependence graph remains invariant in a given execution of the loop, we can extract the information required by emulating one iteration and the cost for that can be amortized over that particular execution of the loop. However, the actual statement dependence graph cannot be computed until runtime so we cannot check if the actual dependence graph is invariant. But we can impose certain conditions (that can be checked at compile-time) on the structure of the loop that is sufficient to conclude that the dependence graph will be invariant at runtime.

1.2 Compile-time identification of IE/L-sections

If L is the loop under question, then let β denote its body. To describe the conditions to be checked, a function SI/L(β, v, S) is defined, where SI stands for Sequence Invariant. When applied to the use of a variable v in a statement S, it returns true or false depending on whether v's value remains invariant during a particular instance of execution of the loop or not. The conditions are from [2] and are listed below for completeness. For β to be an IE/L-section, the following conditions have to be true:

a. For every control statement S in β and for every variable v used in its condition P, $SI/L(\beta, v, S) = true$.
b. For every variable v used in the array subscript expression of a non-control statement S in β, $SI/L(\beta, v, S) = true$.

In order to do a compile-time test to check if a given loop-section is an IE/L-section, we can check the above conditions on all possible (v, S) pairs in β. If both the conditions are true for that β, then it is considered an IE/L-section and not otherwise. Several definitions from the area of compiler design [1] will be used in the following for the computation of SI/L which are described below. Some definitions are from the compilers class at Ohio State University taught by Prof. Atanas Rountev.

IE/L-sections are defined with respect to loops and hence we have to first recover the loops from the generated llvm bytecode. This is done by constructing a control flow graph.

1.2.1 Control flow graph

A control flow graph is a graph made up of Basic Blocks. Basic blocks are maximal sequences of consecutive three-address instructions with the following properties:

1. There cannot be any jumps in the middle of the block and control flow can enter the block only through the first instruction in the block.
2. The conditional or unconditional jump statement can only be at the end of the basic block.

There is an edge from one basic block in the control flow graph to another if there is a jump statement, either conditional or unconditional, from one basic block to another or the last instruction in the first block immediately precedes the first instruction in the second basic block in the input bytecode. The first block will be called the predecessor of the second block and the second block the successor of the first block.

The control structure of the input bytecode can be represented using a control flow graph. Consider the code section below that shows a bytecode representation of an if-else statement. The entry basic block branches to either basic block bb or bb1 depending upon the value in %3. Both of these blocks will then branch the basic block bb2. Its control flow graph will be as shown but without the three address instructions for simplicity –

```
entry:
      ...
  br i1 %3, label %bb, label %bb1
bb:
      ...
  br label %bb2
bb1:
      ...
  br label %bb2
bb2:
      ...
  br label %return
```

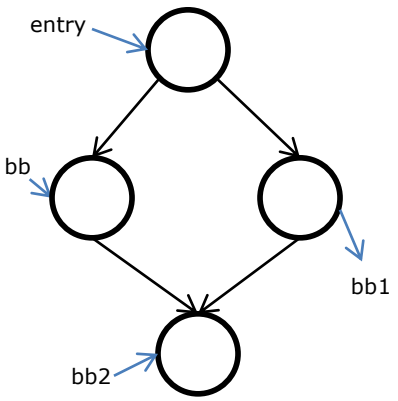

Figure 1: LLVM bytecode and its control flow graph

1.2.2 Loops

A loop in the control flow graph is a strongly connected component with a single entry point. A strongly connected component is a maximal set of nodes such that each node in the set is reachable from every other node in the set.

A backedge in the control flow graph is an edge (a, b) in that control flow graph such that a is dominated by b. A node x dominates another node y in the control flow graph, if all paths from the entry node to node y have go through node x. The natural loop for a back edge (a, b) is the set of all nodes m that can reach node n without going through node b. All such nodes m are dominated by b. For the natural loop formed by the backedge (a, b), node b will be the header.

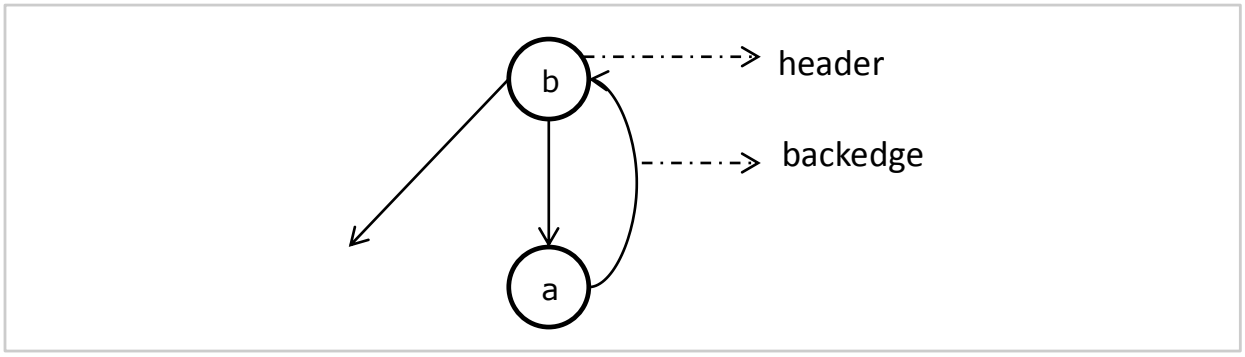

Figure 2: Structure of a loop showing its backedge and header node

Thus using these concepts we can recover the loop structure of the input source code.

1.2.3 Reaching definitions

A statement *S* that assigns a value to a variable *v*, scalar, array or structure variable, is called the definition of that variable. The first step of the algorithm requires the analysis of every use in every statement in the loop body. For each such use, all the definitions that are reaching it are computed. This is done using reaching definition analysis which is a type of dataflow analysis.

If p is a point in the program and d is a definition of a variable x then d reaches p if there is a path in the control flow graph from p to d and if there are no other definitions for that variable x along that path. When there is a definition of the same variable along that path, the former definition is killed by the latter definition.

1.2.4 Control Dependence

Simply stated, a node n is control dependent on another node c in the control flow graph if the decision made at c decides whether node n is executed at runtime or not. Formally, there is a control dependence of node n on node c if,

1. c has two edges e_1 and e_2 and one edge, say e_1, causes n to be executed at runtime.
2. Following edge e_2 avoids the execution of n.

Since a basic block consists of statements, it can be said that all the statements of node *n* are control dependent on the conditional statement in *c*.

1.3 Finding IE/L-sections

The algorithm for identifying the IE/L-sections is a combination for computing the backward program slice and loop invariants for a program. It is from [2] and shown below for completeness.

Call the loop L and its body as β.

Step 1

Check every variable v used in a statement S in β, and assign $SI/L(\beta, v, S)$ a false value if v has definitions reaching S from both inside and outside of the loop L and true otherwise.

Step 2

Let RD be the set of definitions of v under question that are inside L and reaching a statement S under question

Let CP be the set of control predecessors of a given statement S inside

The following is repeated as long as there is a change to any SI/L value,

for each v and S such that $SI/L(\beta, v, S) = dontknow$ do

if any use in any statement in RD or CP has a false SI/L value then set

$SI/L(\beta, v, S) = false$

Step 3

Change $SI/L(\beta, v, S)$ to true for any pair (v, S) such that $SI/L(\beta, v, S) = dontknow$

Chapter 2

Implementation Using LLVM

The Low Level Virtual Machine (LLVM) compiler framework has been used for implementing the algorithms described in other chapters. LLVM has frontends for compiling C, C++ and FORTRAN programs to the LLVM bytecode format. LLVM also has libraries for reading the bytecode programs and performing analysis, transformation and code-generation using it. The analysis part of LLVM has been used in this implementation.

2.1 LLVM Passes

Any analysis on the bytecode program is done through an LLVM pass. Even LLVM optimizations are implemented as passes. Analysis passes do not change the input program and just compute information that other passes can use. So one pass can depend on another pass and this dependency has to be defined when the pass is defined. There are several main LLVM passes available. A custom LLVM pass has to inherit from one of these passes. The main passes are:

1. Module pass

   This pass as the name suggests runs on a module. Using this we can perform analysis on all the functions in the program.

2. Call Graph SCC pass

   This pass is used to traverse the call graph of the program in a bottom up fashion.

3. Function Pass

   This pass visits every function in the input program once.

4. LoopPass

   This pass visits every loop in the program once. If a loop is nested, then each of its sub-loops is also visited.

5. BasicBlockPass

   This pass visits every basic block in the control flow graph of the input program.

6. MachineFunctionPass

   This pass executes on the machine dependent-representation of an LLVM function in the program.

Function Pass is used for implementing the algorithm for finding the IE/L-sections. Even though it would be easier to implement using the LoopPass, FunctionPass is chosen because control dependence and reaching definition computation are done using a FunctionPass and LLVM does not allow a LoopPass to be dependent on a FunctionPass.

To start defining a function pass, we have to inherit from the class FunctionPass. The construct for FunctionPass takes an identifier as its argument which must be passed from our pass. The functions

runOnFunction, getAnalysisUsage are virtual functions that have to be overridden in the derived class. The runOnFunction is the function where the actual implementation of the analysis will be. This function is called for every function in the input program and gets an object of type Function that represents the current function. The getAnalysisUsage method is used to define the dependency between various passes – either custom or built-in. The pass for finding IE/L-sections depends on the control dependence pass and the reaching definition pass so this dependency is established as shown-

```
void SIL::getAnalysisUsage(AnalysisUsage &AU) const
{
   AU.setPreservesAll();
   AU.addRequired<ControlDependence>();
   AU.addRequired<ReachingDef>();
   AU.addRequired<LoopInfo>();
 }
```

Figure 3: Defining interaction between passes

The `LoopInfo` pass gives information about the loops and is also used to visit every loop in the program without having to depend on `LoopPass`. The custom pass also has to be registered so that it can be represented using a command line argument and provided with a description that can be accessed from the command line using the `opt` tool. Registration is done as shown:

```
static RegisterPass<SIL> sil("iel", "find all IE/L sections");
```

Here `iel` is the name using which the SIL pass can be accessed from opt as described in sections ahead.

LLVM is flexible enough to allow custom options to be passes to the opt tool so that they can be interpreted by custom passes. This is done by using the cl::opt template class.

```
cl::opt<bool>
printRejected("iel:print-rejected",
cl::desc("Print rejected candidates for IE/L-sections"));
```

This defines the option "`iel:print-rejected`" that when passed to opt will print the rejected candidates for IE-L/sections along with some brief information that describes why it was rejected. The object `printRejected` defined here can be treated as a `boolean` variable whose value will be true if the above flag was passed and false otherwise. The string passed to `cl::desc`'s constructor is description for this option that is printed when `opt` is executed with the flag `--help`.

2.2 Control flow graph

LLVM has a pass named `BasicBlockPass` that visits every basic block of the control flow graph. Another way of traversing the control flow graph is by manually visiting the successors or predecessors of a given node. We can start with the entry node which can be obtained for a given function in the input program using the following method on the LLVM `Function` object:

```
const BasicBlock &getEntryBlock() const
```

There are several ways of accessing the successors predecessors of a given node in the control flow graph. A convenient way is shown below:

accessing successors

```
for (succ_iterator i = succ_begin(node); i != succ_end(node); ++i)
{
      //...
}
```

accessing predecessors

```
for (pred_iterator i = pred_begin(node); i != pred_end(node); ++i)
{
      //...
}
```

Figure 4: Accessing control flow graph in LLVM

2.3 Loops

LLVM has a pass that makes it easy for traversing the loops in a control flow graph. The `LoopInfo` class provides information about a loop and it can also be used to iterate through all the loops in the input program. There are several ways by which we can get access to the loops in the program using the `LoopInfo` class. They are described below:

1. Iterate through every basic block in the control flow graph and use the `getLoopFor` method. If that basic block is contained in a loop then that loop is returned otherwise `NULL` is returned. This way we can access every loop in the program but since a loop can contain more than one

basic block, we will be visiting the loops more than once. So some kind of flag must be used to avoid this.

2. Use the iterators in `LoopInfo` to iterate through all outer loops in the program. If a loop is nested then its sub-loops can be accessed using `getSubLoops` method on the `Loop` class and this process can be repeated recursively to access all the loops in the program.

```
void processLoop(Loop* loop)
{
   //do some work here
   const std::vector<Loop*>& subLoops = loop->getSubLoops();
   for (std::vector<Loop*>::const_iterator j =
                subLoops.begin(); j != subLoops.end(); ++j)
   {
      Loop* subLoop = *j;
      processLoop(loop);
   }
}

void processLoops(LoopInfo& loopInfo)
{
      for (LoopInfo::iterator i = loopInfo.begin(); i !=
                                          loopInfo.end(); ++i)

      {
            Loop* loop = *i;
            processLoop(loop);
      }
 }
```

Figure 5: Accessing loops in LLVM using LoopInfo

The `LoopInfo` object can be obtained by using the `getAnalysis` function as shown:

```
LoopInfo& loopInfo = getAnalysis<LoopInfo>();
```

The `LoopPass` class can be used to visit every loop in the program once but LLVM does not allow a `LoopPass` to be dependent on a `FunctionPass` and since the control dependence and reaching definition passes are function passes, `LoopInfo` was used.

2.4 Reaching definitions

This section describes the algorithms used for computing reaching definitions and show how they differ when dealing with the specifics of LLVM.

2.4.1 Reaching definitions for scalars

The data flow analysis for reaching definitions can be represented using the following equations:

$$IN[n] = \bigcup_{m \in Predecessors(n)} OUT[m]$$

$$OUT[ENTRY] = \emptyset$$

$$OUT[n] = (IN[n] - KILL[n]) \cup GEN[n]$$

where,

n is a basic block,

Predecessors(n) is the predecessors of that basic block,

ENTRY is the entry edge of the control flow graph,

$KILL[n] = KILL[s_1] \cup KILL[s_2] \cup KILL[s_3] \cup \ldots \cup KILL[s_m]$

and s_1, s_2, $\ldots s_m$ are the instructions in the basic block n and

KILL[s_k] is set of all definitions of the variable whose value is changed by s_k which is basically the set of instructions killed by instruction s_k.

GEN[n] = GEN[s_m] U (GEN[s_{m-1}] - KILL[s_m]) U (GEN[s_{m-2}] - KILL[s_{m-1}] - KILL[s_m]) U ...U (GEN[s_1] – KILL[s_2] - KILL[s_3] - … - KILL[s_m])

where,

GEN[s_k] is the singleton set containing the definition s_k and GEN[n] is the set of all instructions that are downwards exposed i.e., not killed by a subsequent definition in the basic block n.

To calculate KILL[s_k], all the instructions in the basic block can be examined and if a variable is found to be defined more than once then the kill sets can be modified accordingly. For example:

d_1: a = b + c

d_2: d = a + 10

d_3: a = e + f

Variable a is defined twice and the definition at d_3 kills the definition at d_1 and vice versa. So, the kill set KILL[d_1] will contain d_3 and KILL[d_3] will contain d_1.

Once the KILL and GEN sets have been constructed, the IN and OUT sets can be constructed using the following iterative algorithm:

set OUT[ENTRY] to ∅

for every basic block b other than ENTRY

    set OUT[b] to ∅

Repeat the following until there are no changes to any OUT

    for every basic block b other than ENTRY

        set IN[b] to $\cup_{m \text{ is a Predecessor of } b}$ OUT[m];

        set OUT[b] to GEN[b] $\cup$(IN[b] – KILL[b]);

Figure 6: Algorithm for computing reaching definitions

The registers in LLVM bytecode representation use the SSA format. SSA stands for Static Single Assignment, which is an intermediate representation where every variable is assigned only once. Since every variable is assigned only once, reaching definition analysis becomes easy. LLVM uses this representation for registers. Finding all the definitions reaching an instruction using this technique is shown below:

```
for (use_iterator i = value.op_begin(); i != value.op_end(); ++i)
{
      //...
}
```

The loop above iterates through every operand of an instruction. Since every register operand has only one definition, this directly finds all the definitions of registers that are reaching this instruction.

2.4.2 Variables with loads and stores

In some cases LLVM resorts to loads and stores for memory variables but this can be changed by running the pass `mem2reg` which promotes memory references to register references. However, arrays and structures are not promoted by this form and they are manipulated using loads and stores as shown:

```
1  %a = alloca [10 x i32]
2  %3 = getelementptr inbounds [10 x i32]* %a, i64 0, i64 %2
3  %4 = load i32* %3, align 4

4  %1 = getelementptr inbounds [10 x i32]* %a, i64 0, i64 %0
5  store i32 undef, i32* %1, align 4
```

Figure 7: Load and store instructions for arrays

`%a` is an array of size 10 whose elements are of 32 bits each. Line 2 gets the address of this array by indexing at location pointed by register `%2`. Line 3 loads the value at this location and stores it into register `%4`.

LLVM uses the SSA form only for registers and since reaching definitions for arrays and structures are also needed, it has to be implemented. This can be done by treating every load as a read and every store as a write. However, for arrays we will not know the exact element that is being written to until

run time and we would like to perform reaching definition analysis at compile time. For example:

```
d1: a[0] = ...

d2: ... = a[i]
```

Here we cannot tell if `a[0]` is being used at d_2 or not until we know the value of `i` and that value may not be available until runtime. Treating a load and a store of an array or a structure element as load and store of the array and the structure itself will get around this problem. This will result in a conservative analysis and a few actual IE/L-sections will be missing from the reported IE/L-sections. For example:

```
1   a[0] = ...

2   for (iter = 0; iter < 10; ++iter)
3   {
4        for (i = 1; i < 10; ++i)
5        {
6             a[j] = a[j - 1] + ...
7        }
8   }
```

The outer loop is a valid IE/L-section because at run-time, the only definition reaching line 6, from outside the loop, is `a[0]`. So a[1] uses a[0], a[2] uses a[1] and so on. At any point during the execution of the loop the value reaching a line is either from inside the loop or outside the loop but not both. But treating every read and write of an array element as read and write of the array itself will effectively change the above code from the perspective of reaching definition analysis into:

```
1    a = ...

2    for (iter = 0; iter < 10; ++iter)
3    {
4         for (i = 1; i < 10; ++i)
5         {
6              a = a + ...
7         }
8    }
```

The loops above are clearly not IE/L-sections and this is mainly because of the transformation of read/write of array elements to read/write of arrays. Reaching definition analysis for arrays and stores treated as one variable does not differ much from that for scalar variables but there are a few changes. Described below is the procedure for this analysis along with the necessary changes.

2.4.2.1 Kill sets for arrays and structures

For arrays and structures, the read and write of any array or structure element is treated as the read and write of the entire array. So an array or structure definition cannot be killed because of a subsequent array or structure definition. For example:

```
d1   a[i] = …
d2   b = a[i] + …
d3   a[j] = …
d4   c = a[i] + a[j]
```

If d_1 is killed because of d_3 and if i and j are different, then the definitions reaching d_4 will be computed as d_3 which is incorrect. Hence, both d_1 and d_3 reach d_4 and d_1 is not killed by d_3 and vice versa. This requires that the kill set for arrays and definitions be empty sets and all the array and structure definitions be downward exposed.

LLVM uses loads and stores for certain variables that are not arrays or structures. For these variables we can use the original equations for reaching definitions but for arrays and structures they are modified as shown:

$$OUT[ENTRY] = \emptyset$$

$$OUT[n] = IN[n] \cup GEN[n]$$

where,

$$GEN[n] = GEN[s_m] \cup GEN[s_{m-1}] \cup GEN[s_{m-2}] \cup \ldots \cup GEN[s_1]$$

2.4.3 Generalized equations

Generalizing this by combining it for all variables,

$$IN[n] = \cup_{m \in Predecessors(n)} OUT[m]$$

$$OUT[ENTRY] = \emptyset$$

$$OUT[n] = (IN[n] - KILL[n]) \cup GEN[n]$$

where n is a basic block,

Predecessors(n) is the predecessors of that basic block,

ENTRY is the entry edge of the control flow graph,

$$KILL[n] = KILL[s_1] \cup KILL[s_2] \cup KILL[s_3] \cup \ldots \cup KILL[s_m]$$

where s_1, s_2 …sm are the instructions **that do not operate on arrays and structures** in the basic block n and

KILL[s_k] is set of all definitions of the variable whose value is changed by s_k is basically the set of instructions killed by instruction s_k.

GEN[n] = GEN[s_m] U (GEN[s_{m-1}] - KILL[s_m]) U (GEN[s_{m-2}] - KILL[s_{m-1}] - KILL[s_m]) U **...**U (GEN[s_1] – KILL[s_2] - KILL[s_3] - … - KILL[s_m])

where,

GEN[s_k] is the singleton set containing the definition s_k and GEN[n] is the set **containing any array, structure definitions in basic block n** and all other instructions that are downwards exposed i.e., not killed by a subsequent definition in the basic block n.

2.5 Control Dependences

Algorithm for finding all control dependences is described below:

1. Consider all control flow graph edges *(c, b)* such that *b* does not post-dominate *c*. This implies that *c* is a branch node containing a conditional statement.
2. Traverse the post-dominator tree bottom-up, starting at *b* and stopping immediately before the parent of *c*.
3. For every visited node *n*, report that *n* is control dependent on *c*

The above algorithm is based on the post-dominator tree of the control flow graph. A node x is said to post-dominate a node y if all paths to the exit node of the control flow graph starting from node y must go through node x. A

node x strictly post-dominates node y if x post-dominates y and x **is not equal to** y. The immediate post-dominator of a node x is the post-dominator of x that does not strictly post-dominate any other strict post-dominators of x. The post-dominator tree is a tree made up of the nodes from the control flow graph such that every parent node post-dominates all its children.

LLVM has a pass named postdomtree represented by the class PostDominatorTree that computes the post dominator tree. So the algorithm for computing the control dependences can be implemented using this pass.

The control dependences can be obtained in terms of basic blocks or instructions. So if a basic block X is control dependent on basic block Y, then all instructions in X are control dependent on the conditional branch instruction in Y. This branch instruction will be the last instruction of that basic block which is also called as Terminator Instruction in LLVM. The three types of Terminator instructions that are relevant to control dependences are described below [12] –

1. conditional br instruction

   `br i1 <cond>, label <iftrue>, label <iffalse>`

   This instruction causes the control flow to transfer to basic block labeled by <`iftrue`> if the condition `<cond>` is true and to basic block labeled `<iffalse>` if the condition is false.

2. switch instruction

   `switch <intty> <value>, label <defaultdest> [ <intty> <val>, label <dest> ... ]`

This instruction transfers control to one of several different basic blocks and specifies a table of values and destinations. This control transfer is dependent on the outcome of the comparison of `<value>`. If this value matches any of the values in the table then the control is transferred to the basic block that has the corresponding label. If the value is not found in the table then control is transferred to the basic block labeled `<defaultdest>`.

3. invoke instruction

```
<result> = invoke [cconv] [ret attrs] <ptr to function ty>
<function ptr val>(<function args>) [fn attrs]
                to label <normal label> unwind label
<exception label>
```

This instruction transfers control to a function that is specified as part of the instruction but the control may also be transferred to `<normal label>` if the callee returns with the ret instruction or `<exception label>` if the callee returns with the unwind instruction.

2.6 Running the tool

The tool to find the IE/L sections in a program is built using LLVM's system of passes. LLVM's passes are run using the LLVM tool named `opt`. This tool takes the name of the pass to run as its argument along with the file on which this pass is to run. The file has to be in the LLVM's bytecode format. LLVM has compilers to translate C, C++, FORTRAN source into LLVM

bytecode format. So if we want to find the IE/L sections in a C source file named input.c, it first needs to be compiled to the LLVM bytecode format using the command:

```
$llvm-gcc –g –emit-llvm –c input.c –o input.bc
```

The –g debug flag has to be included so that there is information about the source code line numbers in the resulting input.bc file. This information will be used while reporting the selected IE/L-sections.

The tool assumes that mem2reg has been run on this resulting input.bc file. So mem2reg also needs to be applied to input.bc as shown-

```
$opt –mem2reg input.bc –o input.bc –f
```

If the source code for the tool has been installed with the rest of llvm then the resulting input.bc file has to be passed to opt as shown below –

```
$opt –iel input.bc
```

otherwise we have to load all the custom-built passes that iel is dependent upon line reaching definition analysis and control dependence analysis.

```
$opt –load <llvm_installation>/Debug/lib/reaching-def.so \
-load <llvm_installation>/Debug/lib/control-dependence.so \
-load <llvm_installation>/Debug/lib/iel.so -iel input.bc
```

where `<llvm_installation>` is the absolute/relative path of the llvm install directory.

The output of this will be a list of IE/L-sections.

2.7 Interpreting the output

The reported IE/L-sections will have enough information to be mapped back to the corresponding source code loops. Here is an example –

```
Line no: 37
Range: 37-38
Source file: av.c
Function: Perl_av_reify
Loop header: bb2
```

Header	Description
`Line no`	Line number of the loop whose body has been idenfitied as an IE/L section
`Range`	Line numbers where this loop is spread across
`Source file`	The source file containing this loop
`Loop header`	The label of the header of this loop in LLVM bytecode
`Function`	The function that contains this loop

Table 1: Description of the IE-L/section report

If we want information about the rejected candidate loops then the option `iel:print-rejected` has to be passes to opt. This is mainly used for debugging. Here is an example of this output –

```
Branch
Line: 2829
Loop: bb175
```

```
Line: 2829
Value:   %378 = load i8* %p.1, align 1
Step2a: 2829
Step1: 2824     2827    2830
```

Header	Description
`Branch`	This indicates that rejection's cause is a branch instruction that mapped to a conditional statement in the source code. This corresponds to Proposition 2'a
`Line`	Loop's line number in the source code
`Loop`	Loop header's label in in LLVM bytecode
`Line`	Line number of the branch instruction containing the value that lead to the rejection of this loop.
`Value`	The value in LLVM bytecode whose SI/L value was false which lead to the rejection of this loop
`Step1, Step2a, Step2b`	Trace to the rejection source. This shows different steps in the algorithm when a false value is assigned to an SI/L parameter. The source line numbers next to these show the propagation path of this value. If a parameter with false value at line number xyz is responsible for setting this parameter's value to false in Step2a, then that line number is included here.

Table 2: Description of rejected candidates for IE/L-section

A count of the total number of loops in the file along with the numberof IE/L-sections can be obtained by passing the option `-iel:print-count`.

2.8 Example

Shown below is a modified code snippet from the jacobi-1d-imper test case from polyrose tests. Call the body of the loop at line 1 as β_1. Since the algorithm is run on the bytecode, shown below Figure 8 is the bytecode for C code.

```
1    for (t = 0; t < tsteps; t++)
2    {
3        for (i = 2; i < n - 1; i++)
4            B[i] = 33 * A[i];
5    }
```

Figure 8: Example code based on jacobi-1d-imper

```
1   entry:
2     %A = alloca [10 x i32]
3     %B = alloca [10 x i32]
4     %"alloca point" = bitcast i32 0 to i32
5     br label %bb4

6   bb:
7     br label %bb2

8   bb1:
9     %0 = sext i32 %i.0 to i64
10    %1 = getelementptr inbounds [10 x i32]* %A, i64 0, i64 %0
11    %2 = load i32* %1, align 4
12    %3 = mul nsw i32 %2, 33
13    %4 = sext i32 %i.0 to i64
14    %5 = getelementptr inbounds [10 x i32]* %B, i64 0, i64 %4
15    store i32 %3, i32* %5, align 4
16    %6 = add nsw i32 %i.0, 1
17    br label %bb2
18
19  bb2:
20    %i.0 = phi i32 [ 2, %bb ], [ %6, %bb1 ]
21    %7 = sub nsw i32 4096, 1
22    %8 = icmp sgt i32 %7, %i.0
23    br i1 %8, label %bb1, label %bb3

24  bb3:
25    %9 = add nsw i32 %t.0, 1
26    br label %bb4

27  bb4:
28    %t.0 = phi i32 [ 0, %entry ], [ %9, %bb3 ]
29    %10 = icmp slt i32 %t.0, 10000
30    br i1 %10, label %bb, label %bb5

31  bb5:
32    br label %return

33  return:
34    ret i32 0
```

Figure 9: LLVM bytecode for C code in Figure 8

(β, v, S)	Reaching values	RD	CP	s_0	s_1
(β_1, %t.0, 25)	0, %9	25		F	F
(β_1, %7, 22)		22		D	T
(β_1, %i.0, 22)	2, %6	16,		D	T
(β_1, %8, 23)		23		D	(T)
(β_1, %i.0, 9)	2, %6	16	23	D	T
(β_1, %A, 10)		-	23	D	T
(β_1, %0, 10)		10	23	D	(T)
(β_1, %1, 11)		11	23	D	T
(β_1, %2, 12)		12	23	D	T
(β_1, %i.0, 13)		9	23	D	T
(β_1, %B, 14)		-	23	D	T
(β_1, %4, 14)		14	23	D	T
(β_1, %3, 15)		15	23	D	T
(β_1, %5, 15)		15	23	D	T
(β_1, %i.0, %6)	2, %6	9	23	D	T

Table 3: Trace of SI/L values for the bytecode in Figure 9

The first column shows the SI/L-parameters in β_1. The second column labeled *Reaching values* is the set of definitions reaching the instruction for that SI/L parameter. RD and CP are as described from Section 1.3. s_0 and s_1 show the SI/L values for all the parameters for the different steps and iterations in the algorithm. The circled values are the ones that are finally checked to see if β_1 is an IE/L-section or not. These values are picked on the basis of conditions in Section 1.2.

Since an IE/L-section was found its details are printed so that it can be correlated with the source code as shown:

```
Line no: 25
Range: 25-28
Source file: test.c
Loop header: bb4
Function: main
```

Description of each of the above field is in Section 2.7.

Chapter 3

Characterization Of IE/L-Sections

This sections shows the relative running time of all the IE/L-sections in the program with respect to that of the entire program. The programs selected are from SPEC 2000 and SPEC 2006 benchmarks. HPCToolkit was used to measure the performance of the program.

3.1 Using HPCToolkit

HPCToolkit[13] reports the relative performance of individual loops in the program with respect to that of the entire program. HPCToolkit integrates with PAPI and allows specification of PAPI events. The list of PAPI events can be obtained by running the commands papi_avail and papi_native_avail. The event used in this performance analysis is PAPI_TOT_CYC which represents the total cycles. Specifiying a period of say 1000000 for this event in HPCToolkit samples the running application every 10000000 cycles.

HPCToolkit can report the performance in three ways - calling context view, callers view and flat view. Flat view is used in this performance analysis as it shows the total cost of a loop for the entire duration of the program. So even if two different procedures have a call to another procedure containing

a loop, then the cost of the loop in Flat view will be the aggregate of the cost of the loop in each of those calls. HPCToolkit comes with hpcviewer that shows all three views in a GUI. However, for this performance analysis hpcviewer was not used and instead hpcrun-flat and hpcproftt were used that provide the flat view of performance in a textual dump which is easier for analysis using shell scripts.

The steps in obtaining the flat view performance using hpcrun-flat and hpcproftt are shown below:

1. Compile the input program with debug information.
2. Run the program using hpcrun-flat. This creates a file containing the performance information. A new file is created for every run.

   $hpcrun-flat -e PAPI_TOT_CYC:period -- binary_name binary_arguments
3. Run hpcstruct with the binary to be profiled as its argument. This recovers the structure of the program in a file named *binary_name.hpcstruct.*

   $hpcstruct binary_name
4. Run hpcproftt by providing the name of the files created by (1) and (2). There is an option to specify that we want the performance summary of just loops and performance information alongside source. This is done by using the option `–src=l,src`. `l` represents the loops. Other options can be found in hpcproftt's man page. This creates a textual dump of the flat view performance summary. The -I option provides the location of the source code for this binary if hpcproftt cannot find the location but is not always required.

```
$hpcproftt --src=l,src  file_created_by_hpcrun-flat  -S
                file_created_by_hpcstruct    -I path_to_source_code
```

3.2 Building SPEC benchmarks

Compiling the benchmarks without using the benchmark's makefiles can be tricky. SPEC has a simple method for building these benchmarks. Since we need debug information to be enabled we have to modify the makefiles. SPEC uses a common makefile for all the benchmarks that is in the folder *spec-installation*/benchspec and is named Makefile.defaults. This has separate sections for specifying the flags for C, C++ and FORTRAN programs. The EXTRA_CFLAGS, EXTRA_CXXFLAGS and EXTRA_FFLAGS were modified by adding the "-g" flag which is for debug information in the binaries. Also, the optimizations were disabled by providing the –O0 flag in the EXTRA_OPTIMIZE field.

3.3 Identifying IE/L-sections in SPEC benchmarks

Since HPCToolkit reports the loop performance by providing the line number of the loop and the source file containing the loop, it becomes necessary to provide these pieces of information while printing out the IE/L-sections as well. This allows correlation of the idenfiied IE/L-sections with the output from HPCToolkit. This information is provided from llvm-2.7 onwards.

The LLVM Instruction class has methods named getMetadata and setMetadata that can be used for managing information about every instruction that is available across different llvm passes. The debug information is also provided using this mechanism. If inst is an llvm

Instruction then `inst->getMetadata("dbg")` returns an object of type MDNode. The line and source file information can then be obtained as shown-

```
MDNode* mdNode = inst->getMetadata("dbg");
DILocation loc(mdNode);
const char* file = loc.getFilename().str().c_str();
int line = loc.getLineNumber();
```

For getting the line number of a loop we can use the first instruction in the header node of the loop. We can also print out the range of line numbers of a loop by computing the min and max of all line numbers for the instructions in the loop by either going through all the basic blocks or just examining the header node and the node connected to the header node through the backedge.

Since we want the performance of all IE/L-sections in the program, the outer loops of the program are first checked for IE/L-sections. If this check fails then its sub-loops are analyzed recursively. Thus even if all the loops in a nested loop are IE/L-sections, including the outer most loop, then only that loop is reported. This makes it easy for correlating the data from HPCToolkit automatically as HPCToolkit reports the performance of each loop in a nested loop separately. Not including the option `iel:outer-loops` in `opt` will report the nested loops as well.

3.4 Results

The following graphs show the relative execution time of IE/L-sections with respect to the whole program for SPEC 2006 and SPEC 2000 benchmarks.

The bars in red represent FORTRAN benchmarks. Some of FORTRAN's DO loops are translated by LLVM such that the source code body becomes part of the header of the control flow graph loop. Since the IE/L-section analysis is on the body of the loop and since there are no statements in the body that cause a rejection, all such loops are counted as IE/L-sections even though they are not IE/L-sections when their source code is analyzed. Since we are doing the analysis on the bytecode, we have to go by the definition of the loop with respect to a control flow graph. These loops are not reported in the figures below.

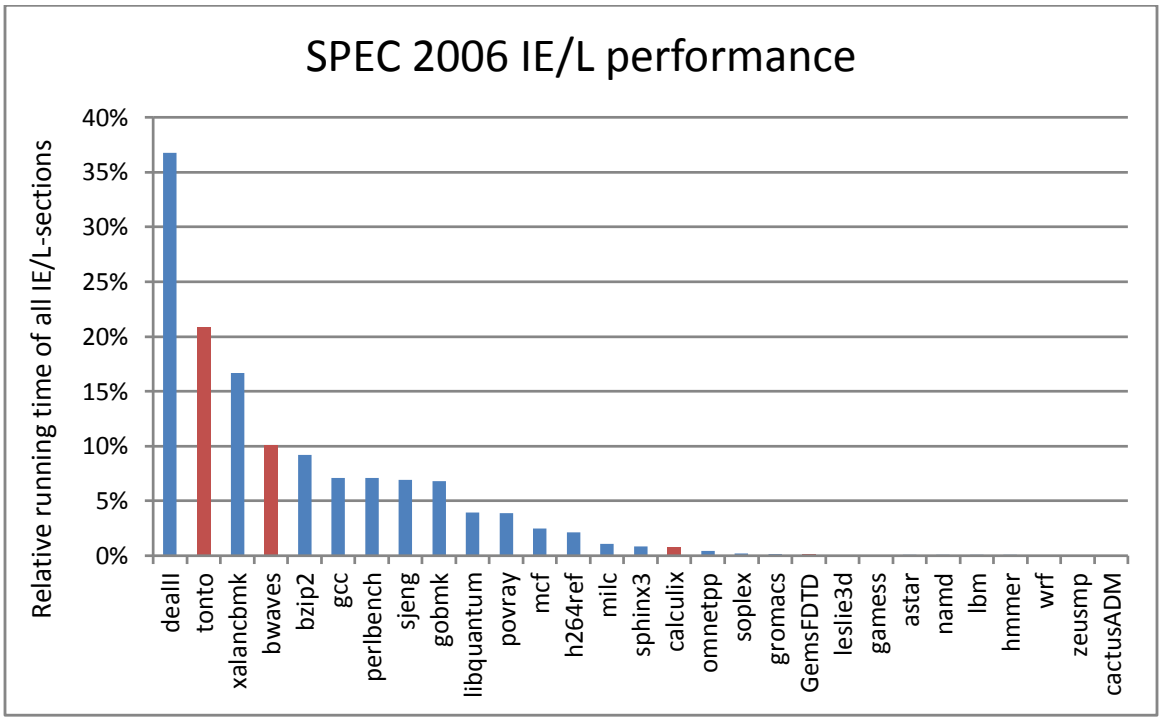

Figure 10: Performance of IE/L-sections in SPEC 2006 benchmarks

SPEC 2006 benchmark	IE/L-sections' relative execution time
dealII	36.77%
xalancbmk	16.65%
bzip2	9.17%
perlbench	7.11%
gobmk	6.77%
povray	3.85%
h264ref	2.11%
sphinx3	0.83%
omnetpp	0.44%
gromacs	0.17%
leslie3d	0.1%
astar	0.01%
lbm	0.01%
wrf	0%
cactusADM	0%

SPEC 2006 benchmark cont.…	IE/L-sections' relative execution time
tonto	20.85%
bwaves	10.07%
gcc	7.12%
sjeng	6.89%
libquantum	3.95%
mcf	2.46%
milc	1.09%
calculix	0.76%
soplex	0.19%
gemsFDTD	0.17%
gamess	0.02%
namd	0.01%
hmmer	0%
zeusmp	0%

Table 4: Performance of IE/L-sections in SPEC 2006 benchmarks

The following figure shows the nesting depths of IE/L-sections in the SPEC 2006 benchmarks.

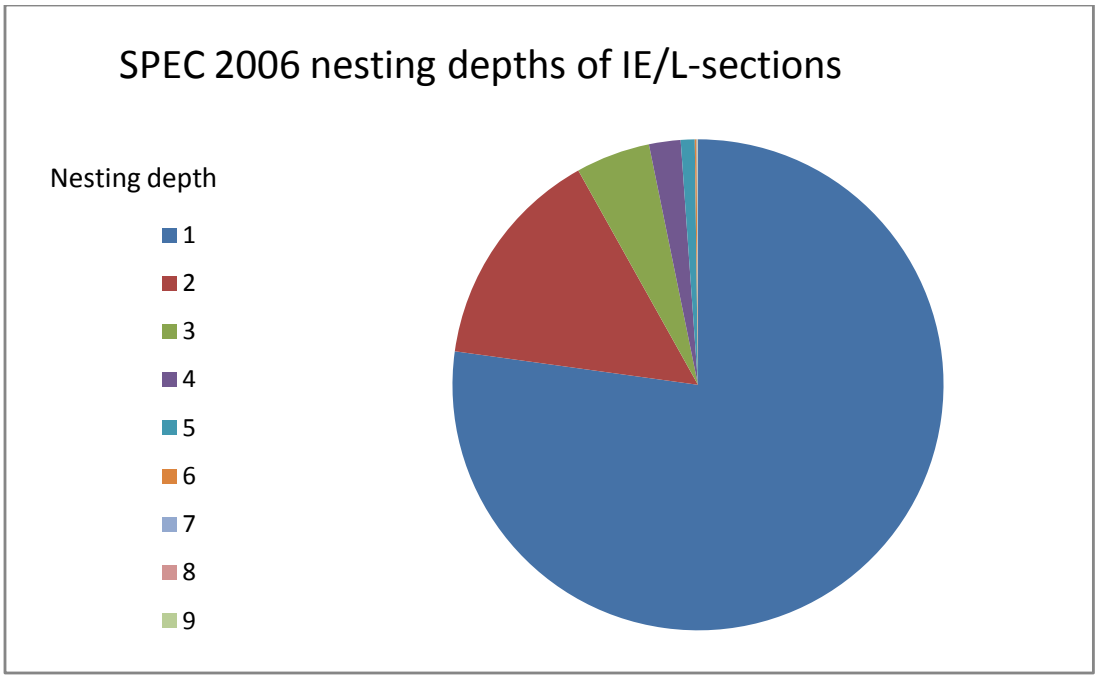

Figure 11: Nesting depths of IE/L-sections in SPEC 2006 benchmarks

SPEC 2006 benchmark	Number of IE/L-sections
1	6569
2	1252
3	417
4	177
5	78
6	10
7	5
8	1
9	2

Table 5: Nesting depths of IE/L-sections in SPEC 2006 benchmarks

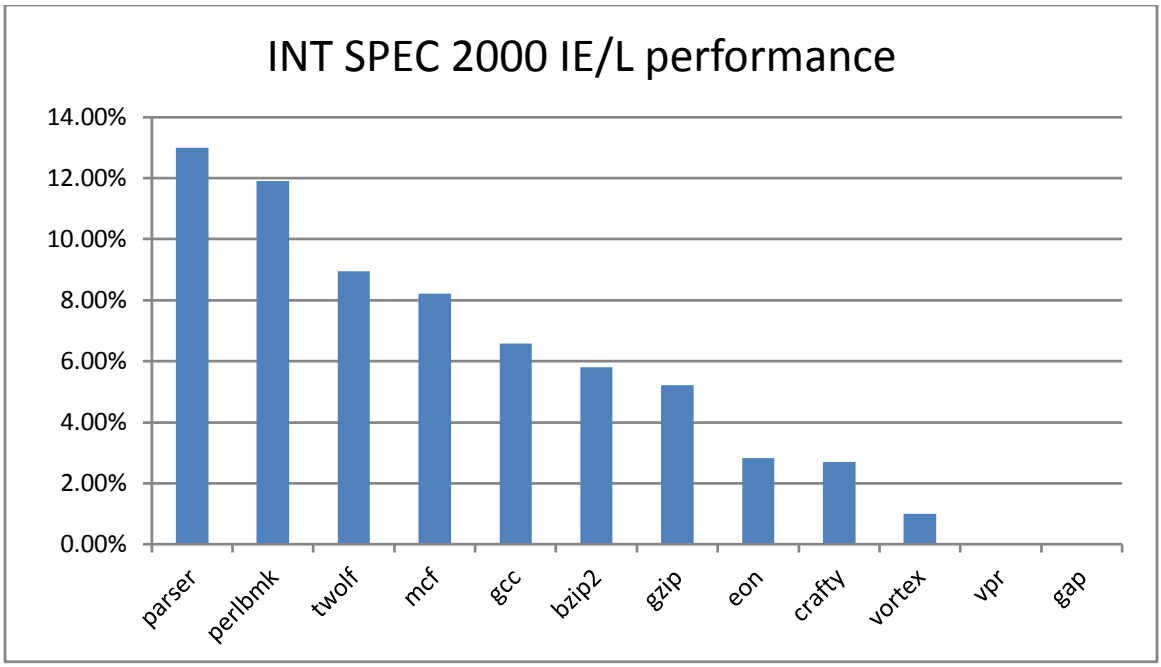

Figure 12: Performance of IE/L-sections in SPEC 2000 Int benchmarks

SPEC 2000 **Integer** benchmark	IE/L-sections' relative execution time
parser	12.99%
perlbmk	11.91%
twolf	8.95%
mcf	8.22%
gcc	6.59%
bzip2	5.80%
gzip	5.21%
eon	2.83%
crafty	2.70%
vortex	1.00%
vpr	0.03%
gap	0.00%

Table 6: Performance of IE/L-sections in SPEC 2000 Int benchmarks

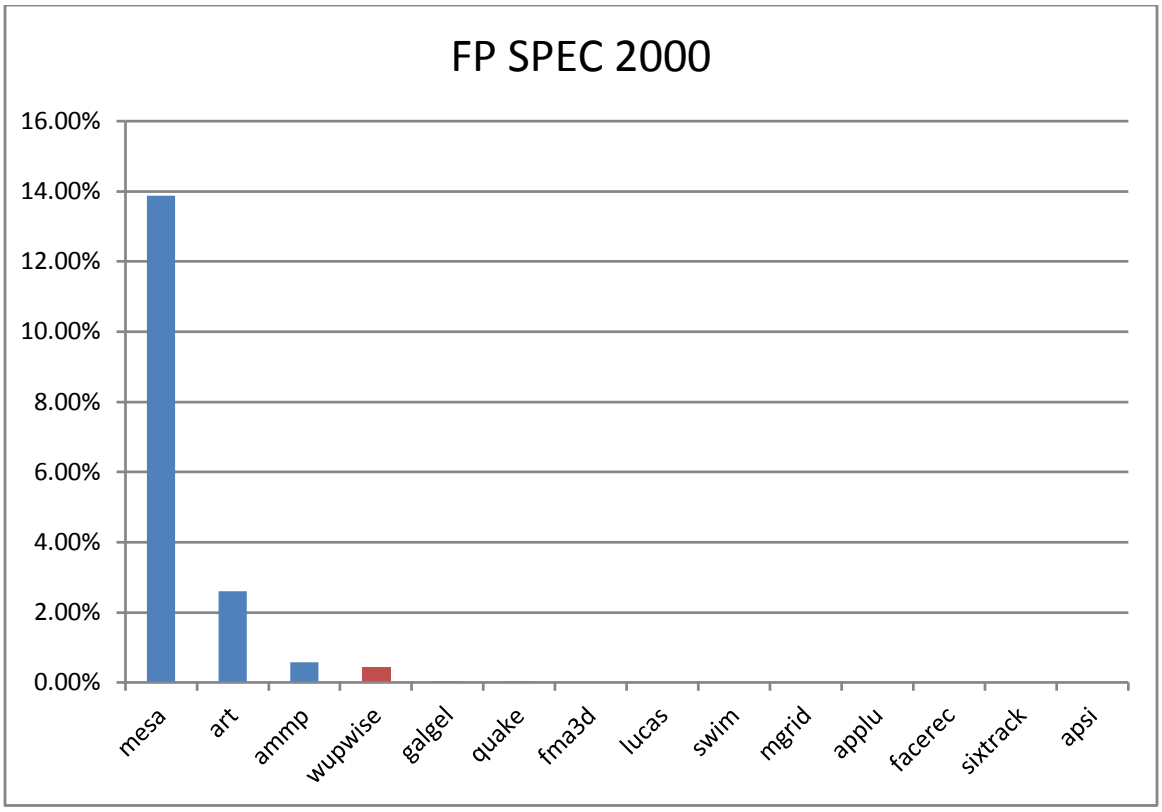

Figure 13: Performance of IE/L-sections in SPEC 2000 FP benchmarks

SPEC 2000 **Floating point** benchmark	IE/L-sections' relative execution time
mesa	13.88%
art	2.61%
ammp	0.58%
wupwise	0.46%
galgel	0.05%
quake	0.02%
fma3d	0.01%
lucas	0.01%
swim	0%
mgrid	0%
applu	0%
facerec	0%
sixtrack	0%
apsi	0.00%

Table 7: Performance of IE/L-sections in SPEC 2000 FP benchmarks

The following fitures show the nesting depths of IE/L-sections in the SPEC 2000 Integer and Floating point benchmarks. The first one shows that among the IE/L-sections that were found in the Integer benchmarks, 62% of them had a nesting depth of 1, 26% of them had a nesting depth of 2 and so on.

Nesting depth	Number of IE/L-sections
1	2048
2	1026
3	385
4	99
5	1

Table 8: Nesting depths of IE/L-sections in SPEC 2000 Int benchmarks

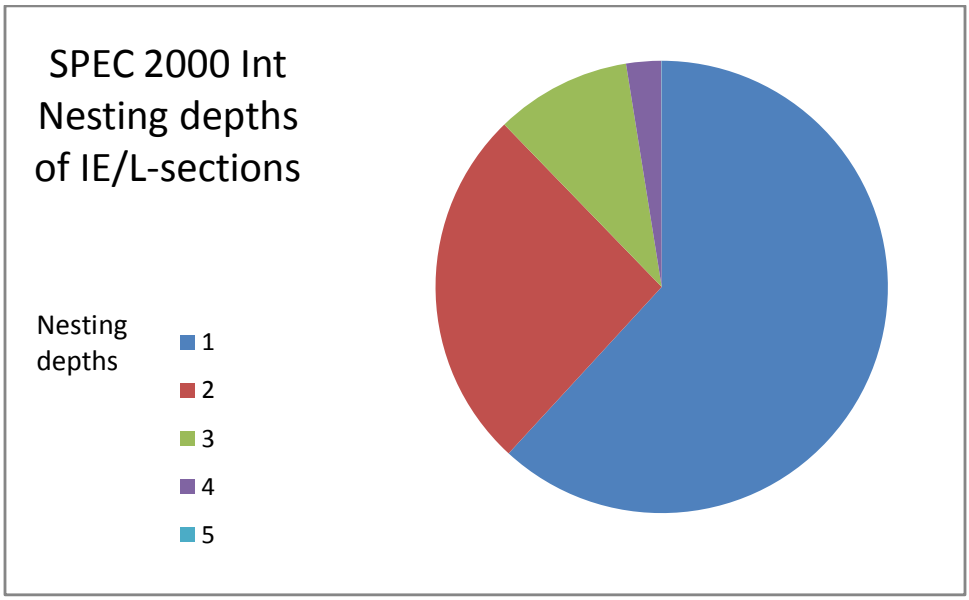

Figure 14: Nesting depths of IE/L-sections in SPEC 2000 Int benchmarks

Nesting depth	Number of IE/L-sections
1	100
2	66
3	39
4	2

Table 9: Nesting depths of IE/L-sections in SPEC 2000 FP benchmarks

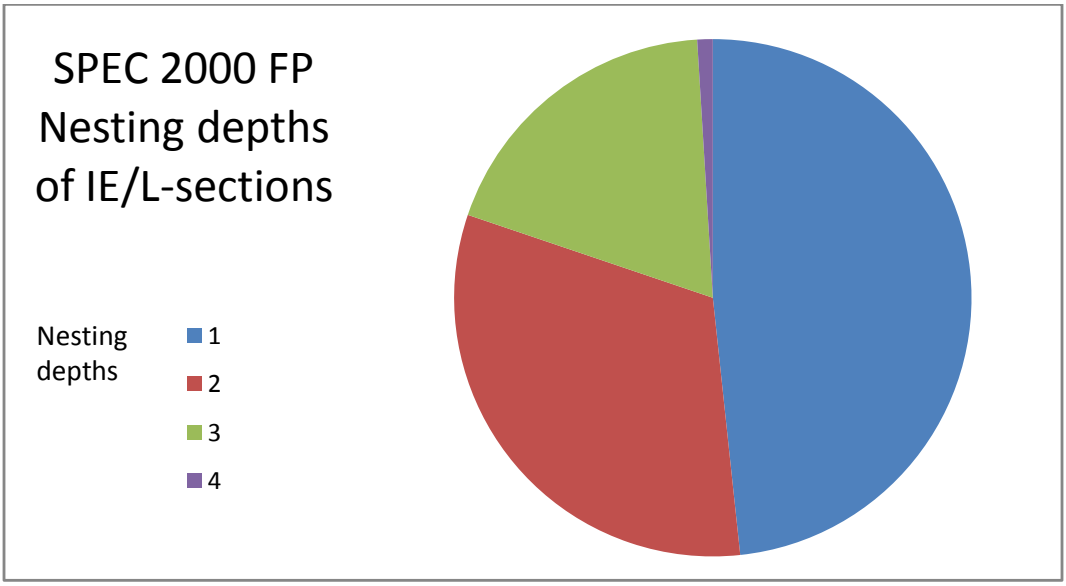

Figure 15: Nesting depths of IE/L-sections in SPEC 2000 Int benchmarks

Bibliography

1. **A. Aho, M. Lam, R. Sethi, and J. Ullman**, "*Compilers: Principles, Techniques and Tools*" Pearson Education, 2007.

2. **K. Eswar, P. Sadayappan, C. Huang**, "Compile-time Characterization of Recurrent Patterns in Irregular Computations"

3. **H. Berryman, J. Saltz, and J. Scroggs**, "*Execution time support for adaptive scientific algorithms on distributed memory machines*". Concurrency: Practice and Experience, Vol. 3, pp. 159-178, 1991.

4. **R. Das, et al.,** "*The design and implementation of a parallel unstructured Euler solver using software primitives*". ICASE Report No. 92-12, Institute for Computer Applications in Science and Engineering, Rice University, 1992.

5. **G. Fox, et al.,** "*Fortran D Language Specification*". Rice COMP TR90-141, Department of Computer Science, Rice University, 1990.

6. **M. Gupta and P. Banerjee**, "*A methodology for high level synthesis of communication on multicomputers*". Sixth ACM International Conference on Supercomptuting, 1991.

7. **S. K. S. Gupta, et al.,** "*On the generation of efficient data communication for distributed-memory machines*". Proceedings of the International Computing Symposium, Vol. 1, pp. 504-513, 1992.

8. High Performance Fortran Forum, *High Performance Fortran Language Specification*, Version 1.0, Draft, 1993

9. **S. Hiranandani, K. Kennedy and C. Tseng**, "*Compiler support for machine-independent parallel programming in Fortran D*", In Compilers and Runtime Software for Scalable Multiprocessors (J. Saltz and P. Mehrotra, editors), Elsevier, 1991.

10. **P. Sadayappan and V. Visvanathan**, "*Circuit simulation on shared-memory multiprocessors*". IEEE Trans. Comput., Vol. 37, pp. 1634-1642, 1988

11. **J. Wu, J. Saltz, H. Berryman, and S. Hiranandani**, "Distributed memory compiler design for sparse problems." ICASE Report No. 91-13, Institute for Computer Applications in Science and Engineering, 1991

12. Low Level Virtual Machine (http://llvm.org/)
13. HPCToolkit (http://hpctoolkit.org/)