CHALMERS

Increasing parallelizing compiler efficiency using commutative functions

Master of Science Thesis Integrated Electronic System Design

JACOB LIDMAN

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2011

Increasing parallelizing compiler efficiency using commutative functions

JACOB LIDMAN

Examiner: Sally A. McKee

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2011

Abstract

The need for suitable software abstractions is proportional to the complexity of the underlying computer architecture. Even though language technology has made significant contributions over the years, many hard decisions have been delegated to the compiler. Naturally, compilers are only as efficient as the algorithms they contain.

In this thesis we consider an approach to detecting commuting operations that give the same result independent of execution order. Our approached is based on analyzing commutativity of any pair of statements rather than just consecutive function calls. This will allow us to detect opportunities for parallelization that previous conservative approaches would ignore or miss because of context-insensitivity.

Acknowledgements

I would foremost like to thank my adviser, Prof. Sally A. McKee for allowing me to proceed with my idea concerning usage of commutative statements in automatic parallelizing compilers. A lot of obstacles would have been nearly impossible to breach within the limited time of this thesis without her guidance. Furthermore I would like to thank Anders Widen, Bhavishya Goel, and Chen Guancheng for valuable insights during many and sometimes late discussions in the office.

Foreword

Imagine yourself preparing dinner for some friends, a few of them have agreed to help. You know that dinner will be delicious if you follow a recipe. But, as always, you don't want to spend more time then necessary (author's opinion). So how would you delegate the subtasks of the recipe to your friends so that the time requirement is minimized? Looking at the tasks you quickly figure out that **chop onion, carrots and lime** and **prepare a white roux** can be done by two different friends at the same time since they are independent. But there are also instructions such as **pout warm milk into the frying pan** and **put grated cheese into the frying pan**, these are not independent since both uses the frying pan. The important question here is: *"Will the state of the frying pan be the same if these are performed by different friends¿'*.

A very similar problem happens to be the topic of this thesis.

Contents

Abstract **II**

Acknowledgements **III**

Foreword **III**

List of Figures **VI**

Abbreviations **VII**

1 Introduction **1**
- 1.1 Report outline . . . 2

2 Background information **3**
- 2.1 Analysis phase . . . 4
  - 2.1.1 Data-flow analysis . . . 4
  - 2.1.2 Data dependency analysis . . . 5
  - 2.1.3 Pointer analysis . . . 7
  - 2.1.4 Call graph generation . . . 8
  - 2.1.5 Constant propagation . . . 8
- 2.2 Optimization phase . . . 9
  - 2.2.1 Global value numbering . . . 11
- 2.3 Generation phase . . . 11
- 2.4 Symbolic analysis . . . 12
  - 2.4.1 Basic statements/blocks . . . 12
  - 2.4.2 Conditional statements . . . 13
  - 2.4.3 Loops and recurrences . . . 13
  - 2.4.4 Symbolic analysis in parallelizing compilers . . . 14
- 2.5 Summery . . . 14

3 Commutativity analysis **15**
- 3.1 Fine-grain commutativity analysis . . . 17
  - 3.1.1 Global value vector . . . 18
  - 3.1.2 Revisiting a failed reordering . . . 19
- 3.2 Summery . . . 22

4 Implementation **23**
- 4.1 Host compiler . . . 23
- 4.2 Intermediate language transformation . . . 24
  - 4.2.1 Global transformation . . . 24
  - 4.2.2 Local transformations . . . 25
  - 4.2.3 Intermediate language . . . 25

4.3 SAFT . . . 25
4.3.1 Primitive representations . . . 26
4.3.2 Symbolic expression solving and manipulation . . . 27
4.3.3 Organization of SAFT . . . 27
4.4 Summery . . . 27

5 Results **29**
5.1 Microbenchmarks . . . 30
5.2 Problems . . . 30
5.2.1 Alias analysis . . . 30
5.2.2 Recursive equations . . . 31
5.3 Evaluation . . . 32
5.4 Summery . . . 33

6 Conclusion **34**
6.1 Future work . . . 34
6.2 Discussion and reflections . . . 34
6.3 Final thoughts . . . 35

Bibliography **35**

List of Figures

2.1 Compilation flow in 2.1(a) and midend flow in 2.1(b) 3
2.2 CFG of a toy function and table showing its corresponding reaching definitions IN[S] and OUT[S] 5
2.3 Example where reordering statements break dependencies but are still valid . . . 6
2.4 Dependency analysis of loop represented as iteration space and iteration vector . 7
2.5 Call graph (middle) statically generated from sourcecode (left) and (right) 9

3.1 Retry map after insertion of A, A/B and A/B/C 21

4.1 Implemented framework flowchart 28

Abbreviations

AST	**A**bstract **S**yntax **T**ree
CFG	**C**ontrol **F**low **G**raph
DLP	**D**ata **L**evel **P**arallelism
IL/IR	**I**ntermediate **L**anguage / **I**ntermediate **R**epresentation
ILP	**I**nstruction **L**evel **P**arallelism
SSA	**S**ingle **S**tatic **A**ssignment
TLP	**T**hread **L**evel **P**arallelism
XML	e**X**tensive **M**arkup **L**anguange

Chapter 1

Introduction

Processors are a part of everyday life, ranging from small embedded computers locked away in devices of various shapes and sizes (such as TVs, DVD-players, and cars) to large supercomputers used by the scientific community for simulation purposes. Processors have since the beginning made use of inherent parallelism of the micro-operations within the instructions themselves in a similar fashion to how cars are manufactured. As the demand for more computational power grew, engineers added more execution units for common operations, decreased the clock cycle period, etc., all to transparently allow for the implicit performance increase expected by the user community. At some point, fishing for more parallelism among instructions grew too expensive, and power consumption and heat dissipation became a problem. One of the solutions was then to add more processor cores (each operating at a lower frequency then before) to the chip to exploit parallelism between tasks and map sequential program to one or more of these to realize performance improvements.

This, however, posed a new problem in that programmers now had to consider the architecture to a larger extent. The issue of increasing performance was now switching from a hardware to a software problem that programmers must address. Writing good parallel programs is a daunting task even when the number of cores is relatively small [1].

In the extreme, creating parallel programs from sequential versions can be done either manually or automatically. The benefit of manual parallelization lies in changing the algorithm to map better onto the architecture, but at the cost of increased development time. Manual parallelization can ultimately prove to be an inefficient usage of personnel [2]. The second option requires the compiler writer to come up with ways of detecting and exploiting parallelism. The clear advantage is that from the programmer's view the target platform is still a uniprocessor. In real life a combination of these options is used, since the tasks associated with software development require knowledge that the compiler lacks.

This thesis addresses one of the fundamental problems in parallel compiler theory: detecting which parts of the program must execute sequentially. To do this, we use the algebraic property of commutativity. Two statements in the source code can be said to "commute" if they complete with the same result irrelevant of execution order. If two statements commute they can be executed in parallel, assuming they are atomic.

We assume that the reader has an understanding of computer architecture, and in particular parallel architectures where Thread Level Parallelism (TLP) is of central concern. The standard textbooks on the subject [3, 4] explain central issues and terminology of the field.

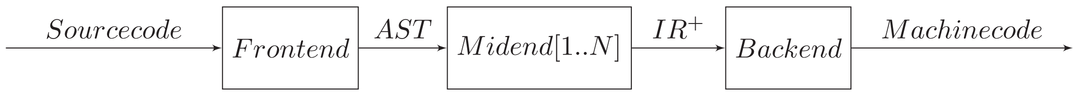
1.1 Report outline

This report starts by introducing the reader to the necessary background information in compilation and the algorithms used in Chapter 2. We primarily deal with the analysis phase of compilation (explained below), and the emphasis of the discussion will be on methods from this and neighboring phases. Earlier and later parts of compilation will only be touched on lightly but names and references to important work are cited for completeness. Chapter 3 introduces (fine-grain) commutativity analysis, an analysis algorithm for deciding whether two statements commute. Chapter 4 discusses the design choices and implementation details concerning the developed algorithm and framework. Chapter 5 includes the results of algorithm, evaluated on microbenchmarks. Finally Chapter 6 discusses further work to improve the algorithm, reflections about the thesis, and conclusion.

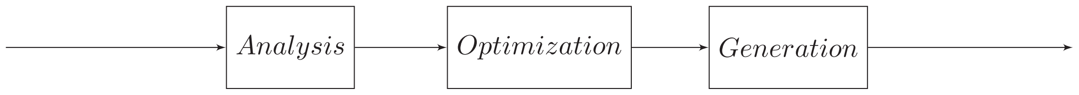
Chapter 2

Background information

Compilation is the process in which source-code is translated to machine-dependent instructions. The assembler later translates (often called "assembles") the instructions into binary code/data, and the linker produces the executable. Figure 2.1(a) shows a textbook example of the flow of compilation with three primary parts described below: the frontend, midend(s), and backend. The midend and backend have similar functions but operate on different Intermediate Representations (IRs); and for this reason we discuss them as one entity.



(a) Compilation flow



(b) Midend flow

Figure 2.1: Compilation flow in 2.1(a) and midend flow in 2.1(b)

The choice of IR depends on the function of the midend/backend and the information needed by these and later parts. It can be as simple as the RTL IR used in the GCC backend or as complex as the C language used as one of the IRs in Haskell. Compilers may further include various numbers of IRs ranging from one in a simple compiler operating only on the Abstract Syntax Tree (AST) itself to GCC with many IRs.

- Common to all compilation is the frontend in which a lexical analyzer (lexer) takes the input sourcecode and translates it to an ordered set of tokens. The resulting AST is then forwarded to a parser, which performs syntactic analysis to determine the grammatical structure with respect to the source language. The AST is then verified by the typechecker for semantic consistency with the type rules of the source language. Once passed through all frontend processes, the AST is generally normalized and/or converted to an IR used in a possible midend/backend.
  The frontend mechanisms are often mechanically generated by syntax-directed translation

[5], where the syntax (and possibly type information) is described in a domain-specific metalanguage that represents the source language in Backus-Naur Form (or any extension such as LBNF, XBNF, ABNF, etc.). Terminals of the language are given as regular expressions to the lexer-generator. These are then transformed to minimized deterministic finite automata (e.g., Thompson's algorithm [6] $\rightarrow$ subset construction algorithm [7] $\rightarrow$ Hopcroft's algorithm [8]). The non-terminals are transformed to a LR(k) parser, such as the LALR(1) parser [5] used by the parser-generator (such as Yacc or Bison).

- The midend/backend phases can be broken down into analysis, optimization, and generation, as shown in Figure 2.1(a). The analysis phase deduces implicit information from the IR to provide the optimization phase with the necessary information to improve the computation. The generation phase finally transforms the IR to another IR or to a machine-dependent language.

2.1 Analysis phase

The analysis phase deduce information from the sourcecode that later guide the optimization phase. In this section we describe conventional program analysis methods that influence our approach. Although symbolic analysis, explained in Section 2.4, belongs to this group it is of central concert to this thesis and hence given a separate section.

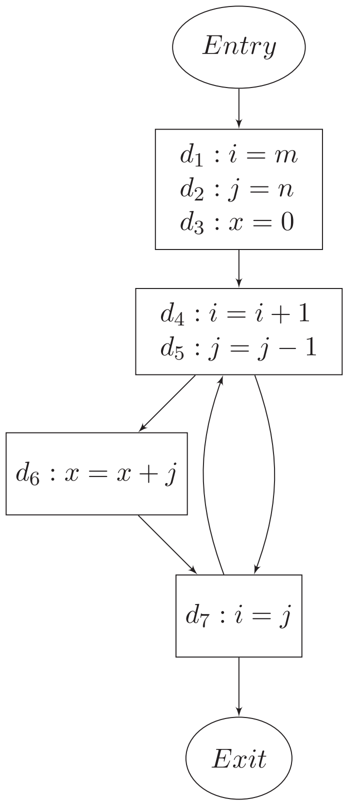
2.1.1 Data-flow analysis

Data-flow analysis is performed via a set of functions that gather information about the data calculated at various points in the program as a function of previous calculations, hence the flow of data. For example, *Reaching Definitions* (RD) determine what variables reach a certain instruction without being altered. This information can then be used to produce the use-def and def-use chains which describe all the definitions for a certain use of a variable or all uses of a particular definition, respectively.
The RD sets of a basic block S are described by Equation 2.1, where the GEN$_n$ are all the assignments y of the basic block not killed by previous basic blocks and KILL$_n$ refers to the killed definitions (= redefined) of the previous basic blocks (that are killed).

$$RD_{in}(S) = \begin{cases} \emptyset & \text{if S is ENTRY} \\ \bigcup_{p \in pred(S)} RD_{out}(p) & \text{otherwise} \end{cases} \tag{2.1}$$

$$RD_{out}(S) = \bigcup_{p \in pred(S)} GEN_p \cup (RD_{in}(p) \setminus KILL_p) \tag{2.2}$$

Computing data-flow equations, including RD, is done iteratively. For RDs the algorithm starts by computing GEN$_n$ and KILL$_n$ of each basic block S. It then compute the in/out-state until it converges, and no changes occur to $RD_{out}(S)$ in an iteration. As an example, consider the CFG shown in Figure 2.2; by computing the $RD_{in}(S)/RD_{out}(S)$ we end up with the result shown in the tables of the same figure. The contents of the $RD_{out}(S)$ then show which definitions exit the block. RD can be used for data dependency analysis and constant propagation.



IN[b]	
b_1	$\emptyset$
b_2	$d_1, d_2, d_3, d_5, d_6, d_7$
b_3	d_3, d_4, d_5, d_6
b_4	d_3, d_4, d_5, d_6
EXIT	d_3, d_5, d_6, d_7

OUT[b]	
b_1	d_1, d_2, d_3
b_2	d_3, d_4, d_5, d_6
b_3	d_4, d_5, d_6
b_4	d_3, d_5, d_6
EXIT	d_3, d_5, d_6

FIGURE 2.2: CFG of a toy function and table showing its corresponding reaching definitions IN[S] and OUT[S]

Another useful data-flow analysis is *live variable analysis*, in which the compiler deduces, for each point in the program which variables that later will be read before written. This can later be used for *dead-code elimination*, described in Section 2.2.

2.1.2 Data dependency analysis

A very central problem to compilation of formal languages is which statements depend on one another. A semantics-preserving compiler can figure out which statements can be reordered. A necessary step is thus to find the constraints (= dependencies) that prevent legal reordering. From Bernstein's conditions [10] two references depend on one another if the following hold:

- They reference the same memory location.
- At least one of the references is writing.
- The two associated statements are executed.

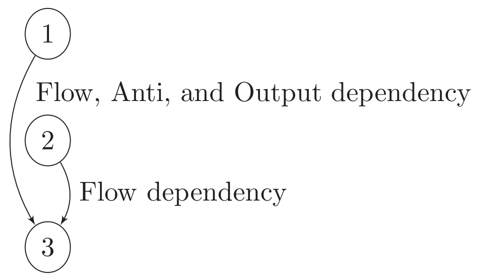
By referring to the first reference in program order as A and the next as B it is possible to name and categorize the dependencies according to:

```
a = 2a;
b = 2b;
a = a+b;
```

Sourcecode

```
a = a+b;
a = 2a;
b = 2b;
```

Possible reordered sourcecode



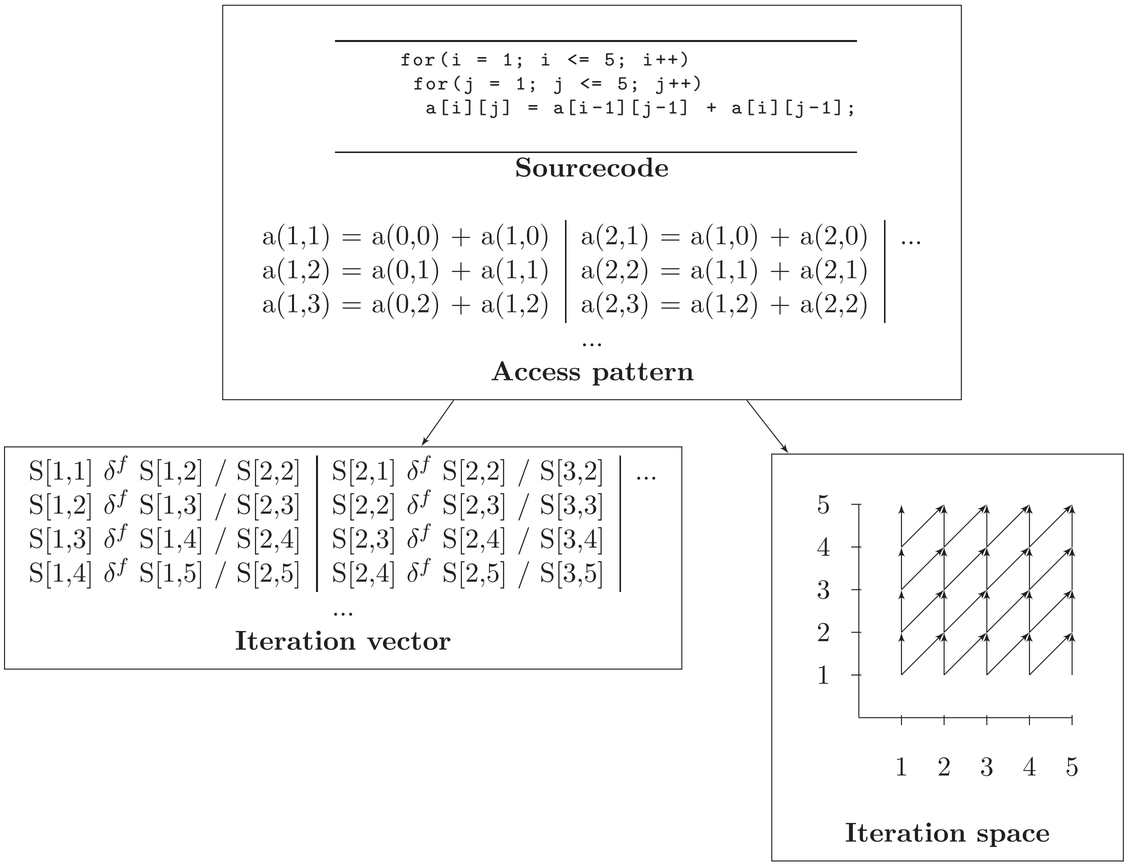
Dependency graph of sourcecode

FIGURE 2.3: Example where reordering statements break dependencies but are still valid

- Flow dependence (True dependence, Read-after-write (RAW) [9]): A assigns a variable and B reads it.
- Anti-dependence (Write-after-read (WAR) [9]): A uses the variable and B reassigns it.
- Output-dependence (Write-After-Write (WAW) [9]): A assigns a variable and B reassigns it.

Both anti and output dependencies represent false dependencies which arise when a variable is reused or reassigned. One should note that dependency analysis is a necessary but not sufficient test for reordering as exemplified in Figure 2.3. As shown, the third statement depends on both the first and the second, yet it is possible to swap the first and third statements with equal results. Data dependency analysis can either use values (contents) or addresses of variables to discover these constraints. Value-based algorithms are a subset of address-based algorithms, and are generally more accurate but take more time [11]. The output of data dependency analysis is a data dependency graph that describes these constraints. This is, however, not enough to handle loops or arrays which are the primary source for coarse-grain concurrency today. Instead, dependencies among loop iterations are used. The most fundamental descriptions of iteration dependencies are the iteration space and iteration vector [11].

Based on these, the compiler can form the expanded dependency graph (EDG), as shown in Figure 2.4. The EDG cannot be used as input to a parallelization algorithm, since it usually becomes too large or cannot be created at compile-time due to expressions with unknown results [12]. This calls for an approximation of the EDG, namely the Reduced Dependency Graph (RDG). But since the decision of approximation can lead to two source codes having the same RDG it generally is not used either. Instead, an extension of the RDG often used is the Polyhedral Reduced Dependency Graph (PRDG), or simply the Polyhedral Dependency Graph. The choice of dependency graph is further complicated by loop transformation algorithms that require specific dependency graphs [12].



```
for(i = 1; i <= 5; i++)
 for(j = 1; j <= 5; j++)
  a[i][j] = a[i-1][j-1] + a[i][j-1];
```

Sourcecode

a(1,1) = a(0,0) + a(1,0) | a(2,1) = a(1,0) + a(2,0) | ...
a(1,2) = a(0,1) + a(1,1) | a(2,2) = a(1,1) + a(2,1)
a(1,3) = a(0,2) + a(1,2) | a(2,3) = a(1,2) + a(2,2)
...

Access pattern

S[1,1] δ^f S[1,2] / S[2,2] | S[2,1] δ^f S[2,2] / S[3,2] | ...
S[1,2] δ^f S[1,3] / S[2,3] | S[2,2] δ^f S[2,3] / S[3,3]
S[1,3] δ^f S[1,4] / S[2,4] | S[2,3] δ^f S[2,4] / S[3,4]
S[1,4] δ^f S[1,5] / S[2,5] | S[2,4] δ^f S[2,5] / S[3,5]
...

Iteration vector

FIGURE 2.4: Dependency analysis of loop represented as iteration space and iteration vector

```
a = &b;
b = &d;
c = &d;
```

Explicit	<*a,b>, <*b,d>, <*c,d>, <**a,d>, <**a, *b>, <**a, *c>, <*b, *c>
Compact/equivalence sets	<*a, b>, <*b,*c,d,**a>
Points-to pairs	(a $\rightarrow$ b), (b $\rightarrow$ d), (c $\rightarrow$ d)

TABLE 2.1: Alias analysis example code (left) and representation (right)

2.1.3 Pointer analysis

Pointer analysis refers to a set of analysis techniques (for language paradigms that include pointers) to deduce information via *alias analysis* (what memory location is the pointer pointing to), *liveness analysis* (which variables are live at the program exit), and *escape analysis* (what is the dynamic scope of the pointer).

Alias / points-to analysis Over the years a lot of work has gone into developing algorithms that can deduce alias information [13], a problem that is undecidable, in general [14]. Algorithms are categorized based on:

- **Flow-sensitivity** — Does the algorithm take advantage of the CFG? Otherwise one solution is calculated for the whole program/function. Notice that this does not include whether the algorithm unifies results for consecutive pointer assignments as one (this is a further question of flow-insensitive algorithms). This is called unification based (or Steensgaard-style [15]), while keeping separate information is called inclusion based (or Andersen-style [16]). These represent two extremes of flow-insensitivity.
- **Context-sensitivity** — Does the algorithm use information in the calling context?
- **Aggregate/Field -sensitivity** — Are different fields of a structure given different treatment or is the structure handled as one unit?
- **Heap-sensitivity** — Is a dynamically allocated memory area treated as one unit?
- **Alias representation** — How are the aliases represented? Table 2.1 illustrates varies alias representations.
- **Definiteness** — Is the result definitely true (must) or might it be true (may)?

2.1.4 Call graph generation

A call graph is a direct (a)cyclic (depending on whether recursive functions exist) multi-graph where each vertex represents a function and an edge e:A $\rightarrow$ B represents that function A is calling B. The problem with generating a call graph lies in whether the language supports function pointers (and whether these can be passed as formal parameters) to another function. Furthermore the precision in the call graph depends on whether context is taken into account; for instance, function A might call function B in one path and C in another mutual exclusive path. Research has shown that even scalable alias analyzers can achieve good precision [17].
Call graphs can be generated statically (at compile time) and dynamically (at runtime). Dynamic generation has the highest precision but may only be valid for a limited input set. Generating an exact call graph statically is undecidable. With low precision comes the ambiguity problem, as shown by the call graph in Figure 2.5 (middle), which when generated statically could come from any of the source codes in Figure 2.5 (left) or (right).

2.1.5 Constant propagation

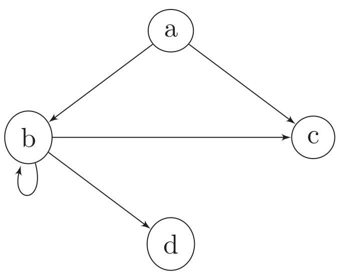
Constant propagation looks for operands that are unchangeable, in which case it substitutes the operand to the evaluated value. It is generally implemented using reaching definitions, where a variable is considered constant if all reaching definitions are the same assignment which assigns the same constant to the variable. By using symbolic analysis (as outlined in Section 2.4) on an SSA IR it is possible to create sparse conditional constant propagation [21, 22], which is strictly more powerful then applying dead-code elimination and constant propagation in any order and with any number of repetitions.

```
int a(int x) {
 int ack;
 c();
 for(i = 0; i < LIMIT; i++)
  ack += b(i);
 return ack;
}

int b(int x) {
 ...
 if(x == 0)
  c();
 else if(x == 1)
  return d();
 else
  return b(x-1)+d();
}

int c() {
 ...
}

int d() {
 ...
}
```



```
int a(int x) {
 int ack;
 c();
 for(i = 0; i < LIMIT; i++)
  ack += b(i);
 return ack;
}

int b(int x) {
 ...
 if(x == 0)
  c();
 else
  return b(x-1)+d();
}

int c() {
 ...
}

int d() {
 ...
}
```

FIGURE 2.5: Call graph (middle) statically generated from sourcecode (left) and (right)

2.2 Optimization phase

Crucial questions when trying to transform a certain code block to something else is that of legality and benefit. This is particularly true for parallel execution where code might be scheduled to execute nondeterministically compared to the sequential schedule. With legality it is meant that the transformed code needs to produce the same result as the original for all possible inputs, although this is sometimes relaxed to mean within a certain difference of the original because of the presence of floating point operations. The benefit of a transformation decides what is gained by the optimization in units of memory accesses, I/O operations or, more general, in performance (latency, throughput), power, etc.
Neither of these properties is easy to guarantee. There might have been undetected dependencies, in the program and the benefit may very well diminish depending on execution context, for instance, due to synchronization requirements.

Although global value numbering (GVN), explained in Section 2.2.1, are of practical importance in this thesis other optimization techniques include, but are far from limited to:

- **Loop invariant code motion** detects statements within loops that is not dependent on the iteration, allowing them to be moved outside the loop. This is the case for variable A in Listing 2.3 which only depends on variable B that is not updated in the loop body.
- **Dead-code-elimination** tries to remove code whose results are not used in later computation, hence it remove redundant computation.
- **Induction variable substitution** refers to the process of finding alternative expressions for recurrence (or induction) variables that are changed in a loop iteration. This can sometimes make the loop iterations independent of each other. Consider Listing 2.1, the

loop cannot be parallelized because of variable A which is flow dependent on the assignment of the previous iteration. In Listing 2.2 the recurrence relation for variable A has been rewritten to a closed form which is now only dependent on the current value of variable I and hence the loop is parallel.

- **Loop transformations** change one loop construct to another equivalent but improved loop construct. This to optimize for locality[1] and alter dependency directions. Since loops make up the single biggest source of TLP in todays programs, a large number of loop transformations have been developed. Table 2.2 shows a few of them. One of the most common is loop interchange; unrolling is also very common, since it may add additional instructions that may benefit scheduling.

Transformation name	**Initial sourcecode**	**After transformation**

Fusion (Combining)	`for(i = 1; i < N; i++)`
	`  Y[i] = 2*Z[i];`
	`for(j = 1; j < N; j++)`
	`  X[j] = 2*Y[j];`
		`for(i = 1; i < N; i++) {`
		`  Y[i] = 2*Z[i];`
		`  X[i] = 2*Y[i];`
		`}`

Unrolling (Unwinding)	`for(i = 1; i < N; i++)`
	`  Y[i] = Z[i] * X[i];`
		`for(i = 1; i < N/2; i+=2) {`
		`  Y[i] = Z[i] * X[i];`
		`  Y[i+1] = Z[i+1] * X[i+1];`
		`}`

Reversal	`for(i = 1; i < N; i++) {`
	`  Y[i] = Z[i]+1;`
	`  X[i] = Y[i]/2;`
	`}`
		`for(i = N-1; i > 0; i--) {`
		`  Y[i] = Z[i]+1;`
		`  X[i] = Y[i]/2;`
		`}`

Interchanging	`for(i = 1; i < N; i++)`
	`  for(j = 1; j < N; j++)`
	`    Y[i][j] = Z[i][j];`
		`for(j = 1; j < N; j++)`
		`  for(i = 1; i < N; i++)`
		`    Y[i][j] = Z[i][j];`

Skewing	`for(i = 1; i <= N+M+1; i++)`
	`  for(j = max(1,i+N);`
	`    j <= min(i,M);`
	`    j++)`
	`    Z[i][j] = Z[i-1][j-1];`
		`for(i = 1; i <= N; i++)`
		`  for(j = 1; j <= M; j++)`
		`    Z[i][j-1] = Z[i-1][j-1];`

Table 2.2: Common loop transformations

[1]That is, by minimizing the access distance in time or space. The spatial distance is sometimes referred to as stride, where stride-1 would be a distance of one word.

```
A = 10
while(I < LIMIT) {
 A = A + 5
 heavy_computation(A)
}
```

Listing 2.1: Induction variable substitution

```
A = 10
while(I < LIMIT) {
 A = 10 + 5*I
 heavy_computation(A)
}
```

Listing 2.2: Induction variable substitution with A expression rewritten

```
A = 10
while(I < LIMIT) {
 A = 5*B
 ...
}
```

Listing 2.3: Loop invariant code detection

2.2.1 Global value numbering

Global value numbering (GVN) is a method for "naming" an expression, is useful for discovering redundant/constant computations. This information can then used by optimization techniques such as common sub-expression elimination to factor out redundant computations.
Traditionally, value numbering was performed separately on each (extended) basic block using hashing [18] where each instruction, including its operands, is added (and given a value number) if it does not exist in the hashmap. Later approaches [19], extend this work and instead of associating assignments with a expression graph use a direct-cyclic graph (representing the program) and make use of Hopcroft's algorithm to minimize it before comparison.
In this thesis we use value numbering for comparing symbolic expressions, a problem which in general is undecidable [20].

2.3 Generation phase

Generation phase concerns converting the IR to machine-code. The difficulty in this lies in:

1. Deciding how an abstract operation should be executed, or selecting the particular instruction.
2. Deciding the order of instructions executed, or the execution schedule.
3. Since by the cache principle only a small amount of memory (registers) is directly accessible for computations, the compiler also has to decide which operands stay in registers and which are stored on the stack at a particular point in the program. This is called register allocation.

s$_0$	$krand = , randVarS = \nabla 1$
s$_1$	$\delta(s_0; krand = \nabla 1/C1)$
s$_2$	$\delta(s_1; randVarS = C2 * (randVarS - (\nabla 1/C1) * C1) - ((\nabla 1/C1) * C3))$

Table 2.3: Symbolic analysis of Listing 2.4 statements 1-3

All these tasks are just as complicated as any of the previous but since it is of little concern to this thesis topic interested readers are referred to textbooks on the subject [5, 9, 11, 12].

2.4 Symbolic analysis

Symbolic analysis is an implementation of abstract interpretation [23], which is the theory of sound approximation of program semantics. By this we mean that abstract interpretation is a partial execution of the computer program without using the actual values in the computation. This allows for understanding of the computation itself, to some extent.
In symbolically executing a statement one obtains a *program context*, a 3-tuple [s,t,p] where s is the state, t is the state condition, and p is the path condition. The state is a set of variable/expression pairs $v_1 = e_1, v_2 = e_2, ...$ where v_x is the variable (memory location) and e_x is the expression unambiguously describing the abstract value of v_x. The state condition includes constraints on the execution order and variable value such as implied by loops, variable declarations, and user assertions. Path condition describes the condition for the execution of a certain statement. Different statements have different impact (or require different representation) on the variable state, as presented below.

2.4.1 Basic statements/blocks

```
int Yacm_random()
{
    register int krand ;
    krand = randVarS / C1 ;
    randVarS = C2 * (randVarS - krand * C1) - (krand * C3) ;
    if( randVarS < 0 ){
           randVarS += C4 ;
    }
    return( randVarS ) ;
} /* end acm_random */
```

Listing 2.4: Park-Miller pseudo-random number generator

In a basic statement, a set of outputs is assigned values based on operations on a set of inputs. Symbolically executing a basic statement hence yields a state as a function of previous state $\delta(previous\ state, changes\ to\ the\ current\ state)$. Consider Listing 2.4. In symbolically analyzing statements 1-3 (line 3-5) we obtain the results shown in Table 2.3. Initially the value of `krand` is set to a default value () while `randVarS`, being a global has an unknown value ($\nabla 1$). `krand` is then updated and finally `randVarS` is updated in the third state as a function of `krand`.

s$_3$	$\delta(s_2, randVarS = \varphi((C2*(randVarS-(\nabla 1/C1)*C1)-((\nabla 1/C1)*C3)<0):$ $C2*(randVarS-(\nabla 1/C1)*C1)-((\nabla 1/C1)*C3)+C4,$ $C2*(randVarS-(\nabla 1/C1)*C1)-((\nabla 1/C1)*C3)))$

TABLE 2.4: Symbolic analysis of Listing 2.4 statements 4-5

2.4.2 Conditional statements

In order to proceed, in the symbolic analysis of Listing 2.4 we need a way to deal with statements constrained by conditions where the new path condition forms a predicate of the execution. For this we use the function $\varphi(condition, true-part, false-part)$ which assigns an expression to a state variable based on the condition. Table 2.4 shows the results of symbolically executing statements 4-5.

2.4.3 Loops and recurrences

Loops are statements for describing recurrence relations. Such as the loop "***for(i** = 0; **i** < LIMIT; **i++**)*", where the induction variable `i` starts at the value 0 (its boundary condition) and is increased by 1 as long as it is below LIMIT (its recurrence condition). A expression describing the recurrence relation (excluding the recurrence condition) is hence

$$i(x) = \begin{cases} 0 & x = 0; \\ i(x-1)+1 & x > 0. \end{cases}$$

To handle loops within a symbolic analyzer more easily, one would like to obtain a closed form of the expression (which here is $i(x) = x$ where x is the iteration number). The number of iterations, as determined by the recurrence condition is given by the number of times **i** can increase while being less then LIMIT hence $\lceil \frac{LIMIT}{1} \rceil = LIMIT$ times. The complete equation of **i** is thus $1 * LIMIT = LIMIT$.

```
while(I < M) {
        F = E*F
        I = I+B
}
```

LISTING 2.5: While-loop

As an example of a complete run of symbolic analysis on a loop, consider Listing 2.5. The variable **I** increases arithmetically ($i_{n+1} - i_n = B$) and F increases geometrically ($f_{n+1}/f_n = E$). Assuming an initial program context of $\{B = b,\ I = b,\ E = x,\ F = y,\ M = m\}$, the symbolic analyzer will run the loop body with temporary values for the variables (it is possible that these are in fact conditional expressions) for simplicity, with path condition set to the loop recurrence condition. At the end of the loop body the initial state is inserted and solved to closed-form if possible. As shown in the result in Table 2.5 we use the function $\nu(Instance, (Boundary, Recurrence), Condition)$ for this with the last state showing **I** solved

s_0	$B = b, I = b, E = x, F = y, M = m$
s_1	$B = B', F = F', I = I', E = E', M = M', p_1 = I' < M'$
s_2	$\delta(s_1; F = E' * F'), p_2 = p_1$
s_3	$\delta(s_2; I = I' + B'), p_3 = p_2$
s_4	$\delta(s_3; I = (I, s_0, [s_3, p_3]), F = (F, s0, [s_3, p_3]), B = b, M = m)$
s_4	$\delta(s_3; I = \nu(\$1, (b, I(\$1 - 1) + b), I(\$1) < m), F = \nu(\$1, (y, E * F(\$1 - 1)), I(\$1) < m))$
s_4	$\delta(s_3; I = b\lceil(m + 1)/b\rceil, F = y * x^{\lceil(m+1)/b\rceil}, B = b, M = m)$

Table 2.5: Symbolic analysis of Listing 2.5

to $b\lceil(m + 1)/b\rceil$ and **F** as $y * x^{\lceil(m+1)/b\rceil}$.

2.4.4 Symbolic analysis in parallelizing compilers

Symbolic analysis has long been used by parallelizing compilers [20, 24], such as the Parafrase-2 compiler [25] and SUIF [26], for various tasks including constant propagation, induction variable substituion, and loop invariant code detection, to name a few.
In this thesis we use it to obtain expressions of a particular ordered set of statements so that a change to the order can be compared for correctness.

2.5 Summery

In this chapter we have introduced methods in compilation that are important to this thesis (and some surrounding topics). In particular the following are of importance.

- A textbook model of compilation can be broken down into a frontend, zero or more midends and one backend phase. Although the control flow tends to be more complicated, particular between midend and backend, this model emphasize that analysis precedes optimization.
- Data dependency analysis is a necessary but not sufficient test in deciding legality of a particular reordering. Although simple for most basic statement, deciding loop-carried dependencies is troublesome.
- Alias analysis is undecidable in general and its approximation algorithms can be categories by six features. Scalable algorithms are fast but conservative.
- Value numbering is a technique to assign a value to expressions and hence ease deciding equivalence between two symbolic expressions. The general case is undecidable.
- The optimization phase deals with problems regarding legality and benefit of transformations.
- Symbolic analysis is a powerful static analysis method to compute symbolic states at a particular program points.

Chapter 3

Commutativity analysis

Deciding commutativity between blocks of statements for the use in automatic parallelization was first shown by Diniz and Rinard [27]. In their work object-oriented applications written in C++ was analyzed to detect instance variable access restrictions and the invoked operations. Two operations were said to commute if the new value of each instance variable of the receiver objects and the invoked operations where the same in both executions orders. The algorithm to detect and verify possible commuting operations worked in the following way:

1. Traverse the call graph to identify variables that the two operations read but do not alter (called extent constant variables).
2. Traverse the call graph again to divide the call sites into auxiliary call sites (= functions that can return values, read/write reference parameters, invoke other auxiliary methods but does not compute values based on state modified by other operations) or full call sites (= functions that may access objects, read reference parameters or invoke full/auxiliary methods but do not return values or change reference parameters).
   (a) Full call sites are visited to make sure that all variables passed by reference into full methods only contain extent constant values.
   (b) Auxiliary call sites are checked to make sure that all auxiliary methods only compute extent constant values.
3. All methods are checked to make sure that they can be divided into a object- and invocation- section. The object section performs all access to the receiver while the invocation section invokes operations and does not access the receiver. It is of course allowed for the object section to pass values to the invocation section.
4. Independence testing and symbolic analysis is then used to verify commutativity according to above criteria.

The practical limitation of this approach came from the symbolic analysis algorithm used as well as limitations on the application it analyzed.

Aleen and Clark [28] extended this work by recognizing that the approach depended on the

fact that operations had to be commuting right after the completion of the last operations. It is however possible that later operations does not care about the difference (a subset of the expression). A conceptually simple case of this is when two functions return values inserted into a array and then consequently compute the difference between these. Depending on the order they are inserted a certain pair may result in x or -x and hence will not be commuting. If however the absolute is applied to this result then both cases will produce the value x.
The approach in [28] depended on random interpretation [29], a combination of random testing and abstract interpretation to verify commutativity. The algorithm worked in the following way:

1. Generate two sets of random inputs used by a function F, called I_1 and I_2, as well as a random memory map M for the outputs.
2. Run random interpretation on F(I_1, M) generating memory values M_1 which are used as context for a second run: F(I_2, M_1) generating memory map M_12.
3. Redo the execution, but now switch order (equivalent to swapping I_1 and I_2): Run F(I_2, M) = M_2 and then F(I_1, M_2) = M_21.
4. If $M_12 = M_21$ then the function F is commuting otherwise go on and recursively try readers of changed memory location to see if their result is altered by the new input.

Notice that the "otherwise" part in Item 4 is what's new. This adds another dimension to the problem in that results have to be rechecked by all readers of the result. In the case of Aleen and Clark [28] the algorithm iterated over all readers and if a certain reader did not accept the change then the readers reader was checked and so on. This recursive behavior was limited to not explode.

Example
Consider the sourcecode in Listing 3 which shows the computation of a histogram of a image which is also sharpened by a Laplacian convolution kernel. The two for loops in the **histf()** function are commuting but not independent because of the update of the global hist variable, neither PoCC [30] or the automatic parallelization program in ROSE [31] can detect this. To reach the conclusion that the loop iterations are commuting we will need to unroll the loop (for simplicity only the inner loop is used in this example) and compare the context for iteration order i;i+1 (orginal) to i+1;i (swapped). The expressions can be obtained as per Section 2.4 and are shown below. As shown both contexts are equal and the result will be the same independently of loop iteration order.

$$i;i{+}1 = \begin{cases} i = IMAGE\_SIZE\_X \\ j = IMAGE\_SIZE\_Y \\ hist[(-I[j+1][i-1] - I[j+1][i] - I[j+1][i+1] - I[j][i-1] + 8 * I[j][i] - ...] + + \\ hist[(-I[j+2][i-1] - I[j+2][i] - I[j+2][i+1] - I[j+1][i-1] + 8 * I[j+1][i] - ...] + + \end{cases}$$

$$i{+}1; i = \begin{cases} i = IMAGE\_SIZE\_X \\ j = IMAGE\_SIZE\_Y \\ hist[(-I[j+2][i-1] - I[j+2][i] - I[j+2][i+1] - I[j+1][i-1] + 8 * I[j+1][i] - ...] ++ \\ hist[(-I[j+1][i-1] - I[j+1][i] - I[j+1][i+1] - I[j][i-1] + 8 * I[j][i] - ...] ++ \end{cases}$$

```c
int hist[MAX_INTENSITY-MIN_INTENSITY];
int I[IMAGE_SIZE_Y+2][IMAGE_SIZE_X+2];

int laplacian(int x, int y) {
  //Laplacian sharpening kernel [-1 -1 -1; -1 8 -1; -1 -1 -1]
  return -I[y+1][x-1]-I[y+1][x]-I[y+1][x+1] +
         -I[y][x-1]+8*I[y][x]-I[y][x+1] +
         -I[y-1][x-1]-I[y-1][x]-I[y-1][x+1];
}
int histf() {
  //Create histogram...
  int i,j;
  for(i = 1; i <= IMAGE_SIZE_X; ++i)
    for(j = 1; j <= IMAGE_SIZE_Y; ++j) {
      hist[laplacian(i,j)-MIN_INTENSITY]++;
    }
  return 0;
}


int main() {
  printf("Histf: \%d\n", histf());
  ...
}
```

3.1 Fine-grain commutativity analysis

In our approach we instead try to detect commutativity on a finer granularity. The motivation behind this is:

- This allows us to identify functions that are commuting in one context but not in another. As a practical example of this consider a function for creating linked list nodes. This function is not commuting. But if it is used to create containers for hashmap entries this call can however be considered commuting, if its also used to store nodes in a standard queue then it will not be commuting in this context. An approach only focusing on detecting commutativity on a function level will not be able to discover this as both contexts will be unified.
- One may always find cases where a few statements are between two calls to different functions, the question is do these statements alter the possibility of the calls to be commuting? In the event that these statements are independent of the calls - no, but if they commute (and are not independent of course) with the state variables of the calls you need a more sophisticated approach to detect this.

In contrast to previous coarse-grain approaches the output of our algorithm is not a list of commuting functions but rather a lower triangular matrix of size proportional to the number of statements of interest. All possible permutations of a function can be expressed by pairwise swapping statements [32]. A certain reordering is possible (or alive) if the result (or program context in terms of symbolic analysis) is the same for both the reordered and original order, otherwise it is not possible (or dead). As discussed above we also need to reconsider dead reorderings later on, hence a certain reordering can fail at first (conditional dead) and later be revived (conditional alive) or succeed in the first place (unconditional alive).

To begin developing an algorithm consider Algorithm 1. In this algorithm all statements are traversed and for each statement there is one more possible reordering than for the previous statement (the first statement can't be moved upwards). In total there are $\sum_{i=0}^{N} \binom{N}{i} = 2^N$ possible permutations for N statements. As per Menon et al. [32] we can generate each permutation of a set of statements by sequences of adjacent transpositions (i.e., iteratively swap two elements). This leaves N^2, but due to symmetry (i.e., reordering statements (i,j) and (j,i), where $i, j \in \{1, N\}$) and redundant swaps (i.e., (i,i)) the total is $\frac{N^2-N}{2}$ possible reorderings. For each possible reordering we check if it is independent in which case it is also commutative. Otherwise the algorithm tries to swap the order of the two statements (called s_i and s_j in program order) involved in the reordering, symbolically execute s_j, all dependent statements in between these two and finally symbolically execute s_i. If the expressions for all altered variables are equal to that of the original order then the reordering is commutative. After each reordering for a particular statement has been considered the algorithm needs to go back and redo all failed reorderings for the previous statements, a very costly operation.

Compared to Algorithm 1 the improved Algorithm 2 does not "go back" and redo the calculations for all failed reorderings. In order to skip this stage we need to be able to deduce properties about the program context and its variables, properties that will give us an idea of when a certain reordering is possible.

When a certain reordering is found not to be commuting the algorithm creates a global value vector (more discussed in Section 3.1.1) for each variable that describes the state of that particular variable in terms of composite functions. The global value vector is then added (together with the global value vector of the context of the original order) to a list in such a way that it will only be revisited if its possible that a later executed statement canceled that difference (more discussed in Section 3.1.2). At the very end of the algorithm the list is revisited and reorderings that can be revived are rechecked.

3.1.1 Global value vector

A global value number is a index for a certain operation as mentioned in Section 2.2.1, a complex expression will be composed of many numbers. Such as $2x + \frac{3y}{z}$ = +(*(2,x),/(*(3,y),z)) = A(B(C,D),E) where A refers to the addition, B to the 2x multiplication and E to division. Applying $\frac{2x}{z} + 2x$ = +(/(*(2,x),z),*(2,x)) to the same algorithm with the previous dictionary would yield G(F,B(C,D)) where the new operations: the first addition G and division F, has been added to the dictionary.

Global value vector on the other hand represent an expression by naming each subexpression.

Algorithm 1: Main algorithm (Naive)

```
Input: S - set of N statements
Input: $C_0$ - Input expressions of dependent variables
Output: A commutativity matrix CM
begin
    forall the Stm $s_i$ in S[1 to N] do
        $C_i \longleftarrow$ RSSymbolicAnalyse($s_i, C_{i-1}$)
        forall the Stm $s_j$ in S[2 to i] do
            if IsIndependant($s_i, s_j$) then
                $CM(i,j) = UNCOND\_ALIVE$
            else
                $C_{tmp} \longleftarrow$ RSSymbolicAnalyse($s_j, C_{i-1}$)
                forall the DepStm $d_i$ in S[i+1 to j-1] do
                    $C_{tmp}$ = RSSymbolicAnalyse($d_i$, $C_{tmp}$)
                $C_{tmp} \longleftarrow$ RSSymbolicAnalyse ($s_i$,$C_{tmp}$)
                if $C_{tmp} = C_i$ then $CM(i,j) = UNCOND\_ALIVE$
                else $CM(i,j) = COND\_DEAD$
        forall the j in [1 to i] do
            forall the k in [2 to j] do
                if $CM(j,k) = COND\_DEAD$ then
                    $C_{tmp} \longleftarrow$ RSSymbolicAnalyse($s_k, C_{j-1}$)
                    forall the DepStm $d_i$ in S[j+1 to k-1] do
                        $C_{tmp}$ = RSSymbolicAnalyse($d_i$, $C_{tmp}$)
                    $C_{tmp} \longleftarrow$ RSSymbolicAnalyse ($s_j$,$C_{tmp}$)
                    forall the DepStm $d_i$ in S[j+1 to i] do
                        $C_{tmp}$ = RSSymbolicAnalyse($d_i$, $C_{tmp}$)
                    if $C_{tmp} = C_i$ then $CM(i,j) = UNCOND\_ALIVE$
```

Global value numbering is applied as above but the result is represented as a vector. In above we have 7 (A to G) (sub)expressions each is given a component i in a vector. If an expression includes the subexpression then the component i will be 1 else 0. For the above expression the global value vector is <1,1,1,1,1,0,0> and <0,1,1,1,0,1,1> respectively. Algorithm 3 performs the above calculation which forms a description of an expression.

3.1.2 Revisiting a failed reordering

As mentioned previously we need more than simply knowing whether that a reordering failed or not. The very reason that a reordering is possible is because for one/many variables the state was different compared to the original execution order.

In Section 3.1.1 we used two expressions to show how a global value vector was represented. The difference between the two is <1,0,0,0,1,1,1> or A, E, F and G. In order for a later statement to justify this reordering the previous components of the difference would need to be 0 (or nonexistent). One should note that a reordering that failed because subexpression A was included will be more likely to regain alive status then one where B and A was missing. In fact a reordering

Algorithm 2: Main algorithm

```
Input: S - set of N statements
Input: $C_0$ - Input expressions of dependent variables
Output: A commutativity matrix CM
begin
    forall the Stm $s_i$ in S[1 to N] do
        $C_i \longleftarrow$ RSSymbolicAnalyse($s_i, C_{i-1}$)
        $V_i \longleftarrow$ GlobalValueVector($C_i$)
        forall the Stm $s_j$ in S[2 to i] do
            if IsIndependant($s_i, s_j$) then
                $CM(i,j) = UNCOND\_ALIVE$
            else
                $C_{tmp} \longleftarrow$ RSSymbolicAnalyse($s_j, C_{i-1}$)
                forall the DepStm $d_i$ in S[i+1 to j-1] do
                    $C_{tmp}$ = RSSymbolicAnalyse($d_i$, $C_{tmp}$)
                $C_{tmp} \longleftarrow$ RSSymbolicAnalyse ($s_i$,$C_{tmp}$)
                if $V_i$ = GlobalValueVector($C_{tmp}$) then
                    $CM(i,j) = UNCOND\_ALIVE$
                else
                    $CM(i,j) = COND\_DEAD$ AddRetryEntry($V_i$,
                    GlobalValueVector($C_{tmp}$))
        forall the ApplicableReorderings ($V_1 j$,$V_2 k$) in RetryMap[Component] do
            $C_{tmp} \longleftarrow$ RSSymbolicAnalyse($s_k, C_{j-1}$)
            forall the DepStm $d_i$ in S[j+1 to k-1] do
                $C_{tmp}$ = RSSymbolicAnalyse($d_i$, $C_{tmp}$)
            $C_{tmp} \longleftarrow$ RSSymbolicAnalyse ($s_k$,$C_{tmp}$)
            forall the DepStm $d_i$ in S[k+1 to i] do
                $C_{tmp}$ = RSSymbolicAnalyse($d_i$, $C_{tmp}$)
            if $V_i$ = GlobalValueVector($C_{tmp}$) then $CM(j,k) = COND\_ALIVE$
```

Algorithm 3: GlobalValueVector - Generation of global value vector

```
Input: E - Variable state
Input: Map - Map of previous named expressions
Output: A expression description V
begin
    forall the Children $C_i$ in E.Children [1 to N] do
        $E_{desc}$[i] = GlobalValueVector($C_i$)
    if Map.Lookup(E($E_{desc}$[1 to N])) != 0 then
        return Map.Lookup(E($E_{desc}$[1 to N]))
    else
        Map[E($E_{desc}$[1 to N])] = AssignNewIndex()
        return Map[E($E_{desc}$[1 to N])]
```

with an A extra has to be alive before A and B. For that reason, failed reorderings are added to a list based on how many subexpressions it needs as well as which expressions it needs. Algorithm 4 shows how this is done. Consider three different reorderings to the same variable: in the first case the difference was A, the second A and B and the third A, B and C. The structure of the retrymap for that variable will now look as in Figure 3.1. If a later statement would change the variable in such a way that A is removed then the first reordering will be rechecked, if that passes the second and later the third. If B is removed then the second reordering will be rechecked and later the third reordering. If C is removed then the third reordering is rechecked.

Algorithm 4: AddRetryEntry - Adding expression to retry buffer

```
Input: ($V_1$, $V_2$) - Variable state encoded as a set of value numbers
Input: Var - Variable name
begin
    $\Delta V \longleftarrow |V_1 - V_2|$
    $(NumComp, Comp) \longleftarrow$ CountComponents($\Delta V$)
    forall the Component $C_i$ in Comp do
        Index = 0
        while Index != NumComp do
            if RetryLink[i][Index] = UNSET —— NumComp then
                RetryLink[i][Index-1] = Index; RetryLink[i][Index] = NumComp
                Index = NumComp
            else if RetryLink[i][Index] ¡ NumComp then
                Index = RetryLink[i][Index]+1
            else if RetryLink[i][Index] ¿ NumComp then
                RetryLink[i][NumComp] = RetryLink[i][Index]
                RetryLink[i][Index] = NumComp
        if Component in $V_1$ then
            RetryMap[i][NumComp] = ($V_1$, $V_2$)
        else
            RetryMap[i][NumComp] = ($V_2$, $V_1$)
```

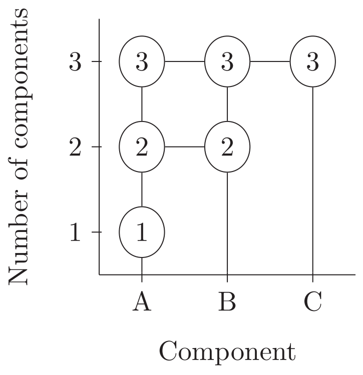


FIGURE 3.1: Retry map after insertion of A, A/B and A/B/C

3.2 Summery

A context-sensitive algorithm to determine commutativity between blocks of statements has been presented. Its introduction is primarily motivated by:

1. Blocks (such as functions) can be commuting in one context but not in another.
2. Current approaches may have problems dealing with seemingly irrelevant but still interfering statements when pattern-matching for statements applicable to their analysis.

The analysis starts from the conclusion of Aleen and Clark [28] but uses symbolic analysis to construct a comparable symbolic expressions of the original and altered state. In contrast to their approach this method does not iterate recursively to all readers of the altered variables, but rather in a bottom-up approach construct summaries in the form of symbolic differences that are later revisited.

Chapter 4

Implementation

The following chapter discusses the implemented framework and design decisions related to it. In order to analyze a program it has to be translated to a representation it can manipulate, something that is done by the frontend of a compiler. Section 4.1 outlines the various host compilers that were considered for this task. A dedicated intermediate language was also defined and is, together with the transformation from the host compilers IR described in Section 4.2. Finally, Section 4.3 describe the framework and its main components.

4.1 Host compiler

Since our approach uses dependence- and alias- analysis it was also necessary for the chosen compiler to easily export such functions. The following compilers were investigated with ROSE as the chosen host mainly because of its clear focus towards researching algorithms as this and its well documented architecture.

- **Trimaran** [33] is a integrated compiler and simulation infrastructure. The compilation process begins with profiling and inlining followed by applying various analysis methods. Optimizations such as loop optimizations, constant folding and simdization (vectorize code using SIMD instructions) are then applied. Finally the application is simulated to obtain runtime results.
- The **LLVM** compiler framework was designed to provide transparent program analysis and transformation during the whole life-time of the application. This “incremental” optimization is achieved using a abstract RISC-like instruction set with 31 opcodes that includes type information, control flow graphs and a dataflow representation using an infinite, typed register set in SSA form [34]. The LLVM IL is very low-level in that it does not contain high-level constructs such as classes or exception handling (such as Microsoft CLI or JVM).
  LLVM include various alias analyzers and methods to generate call graphs, perform liveness analysis, (loop) dependency analysis, constant propagation and induction variable

detection. Supported optimizations include various forms of dead code elimination, constant folding, library-aware compilation and loop optimizations. However LLVM is not a parallelizing compiler; the optimizations done are for uniprocessor systems.

- **ROSE** is a source-to-source compiler framework meant for building program transformations and analysis tools. It support Fortran, C/C++, OpenMP and Unified Parallel C as input languages. The IL used in ROSE is named after its frontend, SageIII and includes all the details of the sourcecode within a AST to not lose information for possible later analysis/transformations. The ROSE framework is compiled to a library which later is called upon from the application. From the application it is possible to control which analysis and transformation passes to use. Included analysis methods include various graph generation functions (such as call-, program-, data- and control- graph), alias analysis, data flow analyses such as live variable and def use/use def. Supported optimizations are more limited then both LLVM and Trimaran but include partial redundancy elimination (a version of common subexpression elimination), constant folding and loop transformations.

The SageIII IR is the complete AST of the input source-code. To simplify SAFT the IR was converted to another, more normalized representation (i.e., converted to a canonical form) that ignore redundant nodes (e.g., comments, variable declarations). The transformation step is outline below in Section 4.2.

4.2 Intermediate language transformation

The framework component "RoseToSAFT" is responsible for converting Sage III IR to an easier manipulated IR. In the process a number of other methods are invoked to retrieve information that is needed by the analyzer. The different transformation phases are classified as either global or local.

4.2.1 Global transformation

The global transformations include transformations done on a project level (a set of source-files). This includes alias analysis and generation/minimization of readers/writers.
The readers are calculated based on inter-procedural data-flow analysis starting with a static callgraph the strongly connected components are computed. Each component will reflect the functions called in a cyclic recursive manner. The strongly connected graph (SCG) is traversed in a reverse topological order. At each component the readers and writers are propagated upwards to the caller. For a component with more than one member the readers and writers are collected in a worst case manner and hence not subject to minimization. The output of this phase is a readers/writers map.
Minimization occurs in the sense that a certain function may have outputs that are only written by itself in a possible execution path. By removing these outputs the symbolic analyzer can safely ignore comparing the expression for this variable. In general this kind of optimization may prove worthwhile for object-oriented languages rather than C as investigated here. This

step was however included for future use.

4.2.2 Local transformations

Local transformations occur on a function basis. The following passes are included:

- Transform sourcecode to canonical form (Desugaring/Normalization).
  - Variable names are renamed in such a way that scopes can be removed.
  - For- and Do-while- loops are converted to while loops.
  - Switch cases are represented as consecutive if-else.
  - Subtraction is converted to addition where the second operand is negated.
- Dependence analysis is being performed so that independence can be determined easily.
- Filters can be applied to limit which reorderings the analyzer considers later.

Dependence analysis is performed on the constructs of the IR. This means (in contrast to conventional approaches) that a a loop reading/assigning the variable X is dependent on a previous assignment of X. The reason for this is that no reorderings occur across control dependencies and hence additional information (such as which statements in the loop) is unnecessary. Since the number of reorderings for a program of n statements in a basic block is $\frac{n^2-n}{2}$ a filter is applied to limit the number. The filter includes which type of statement/expression to pass such as calls, loops etc. This can later be extended to automatically create a filter based on profiling results. Finally statements are grouped based on the control structure they belong to.

4.2.3 Intermediate language

The language is organized into 5 different constructs as detailed below.

1. **Basic** constructs include expression statements such as assignment and calls.
2. **Group** represents a set of other statements.
3. **If** and **Loop** has a direct equivalence to same construct in C. Loop represents a while-loop or normalized do-while-/for- loop.
4. **Function** represent a named group of constructs.

The syntax of the language is further described by the context insensitive description of Table 4.1.

4.3 SAFT

Symbolic Analysis Framework-T (SAFT) performs the computation of the program context for each IR construct. Given a construct it recursively creates the program context by symbolic

Basic $\rightarrow$ Expression
Group $\rightarrow$ (Group | If | Loop | Basic)*
If $\rightarrow$ Basic Group Group
Loop $\rightarrow$ Basic Group
Function $\rightarrow$ Group

TABLE 4.1: CFL description of the SAFT IR

analysis as outlined in Section 2.4. Each state is then subject to simplification and reduction to further improve comparison. Both the simplifier and reducer are important and could be implemented using a Computer Algebra System (CAS), however the cost of performing more elaborate operations may be severe.

4.3.1 Primitive representations

Primitives in SAFT include equation trees and program contexts. The data structures needs to be efficient both in space and execution time since many functions either concatenate or copying them.

Expression
Expressions in SAFT are represented as a direct acyclic graph. Currently nodes are either internal for operations or external for operands. In some cases it might be necessary to have a few special cases, i.e. nodes that represent functions calls (the first operand is the function name) and conditional assignment (first operand is condition, second is true-part and third is false-part) with dedicated nodes. In order to handle conditions, expressions are extended to binary decision diagrams (BDD) with (recurrence) expressions as terminals.
The two most important expressions in the symbolic analysis are the call and assignment expression. Both of them can incur a state update. Since at all times a fully expanded program context is going to be a function of its inputs, globals and various constants (in contrast to a partial expanded program context where locals can exist) it is necessary to have an efficient implementation of lazy-copy for expression trees. Without this, an expression tree would have to be deep copied (in contrast to shallow copied) every time an assignment or (possibly) a call was executed. SAFT implements an observer pattern to handle this across the different layers. As a parent node adds a new child it tells the observer that it is interested in knowing about updates to that child. If the child is also found and changed as part of another parents tree it will duplicate (deep-copy) itself and send a message to the first parent with the copy before updating itself.

Program context
Many applications written in the imperative programming paradigm lack recursive functions of higher order (one seldom find anything higher then self-recursion). Together with the fact that the callgraph of the applications in this paradigm has a very tree-like structure another optimization was added. Besides having the same support for lazy-copy as the expressions, the program context (together with the state) also supports (de)serialization to/from disk depending on when it was previously used. This decreases the memory requirement as it is otherwise quite large when using BDDs.

4.3.2 Symbolic expression solving and manipulation

Simplifier The simplifier propagates not/negations down the expression tree to terminal or non distributive operations (i.e. $-\frac{x}{y} \neq \frac{-x}{-y}$), expand distributive operations and boolean expressions subject to DeMorgans law.

Reducer The reducer uses an interpreter to precompute constant operations as well as removing terminals subject to cancellation (i.e. x-x).

Recurrence equation Solving recurrence equations to closed forms is central to obtaining easier to manipulate expressions. As this is an elaborate task [35] it was decided to delegate this to a library or tool. The following were investigated

- Parma University's Recurrence Relation Solver (PURRS) [36] is a recurrence relation solver library in C++. It supports a number of recurrence equation types (some are limited to special cases) such as:
  - Linear Recurrences of (in)finite order with constant/variable coefficients (e.g., x(n) = 2x(n-1)+1, x(n) = a(n)*x(n-1)+p(n))
  - Generalized recurrences (e.g., x(n) = a * x(n/b) + p(n)) which are very common in divide-and-conqueror functions.
  - Non-linear recurrence relations of finite order (ex. x(n) = x(n-1)*x(n-2))
  - Multivariate recurrences (ex. x(m,n) = f(x(m-1,n-1))).
- Mathematica [37] is a CAS that include automatic simplification/solver, (partial/total) differentiation, (definite/indefinite) integration, (un)constrained optimization as core functions. Mathematica is really the name of the front end linked using a socket protocol called MathLink [38] to the Mathematica kernel. The internal recurrence solver RSolve has previously been used in symbolic analysis engines [39].

Since PURRS has open bugs and is no longer maintained it was decided to go with Mathematica.

4.3.3 Organization of SAFT

The implemented framework is shown in Figure 4.1. As mentioned ROSE IR is transformed to SAFT simplified representation. The fine-grain commutativity analysis module then iterate on program context returned from SAFT for each statement block in sourcecode.

4.4 Summery

This chapter outlined the implementation specific details of the framework. The host compiler ROSE transforms the sourcecode to its IL which is normalized to SAFT IR. The fine-grain

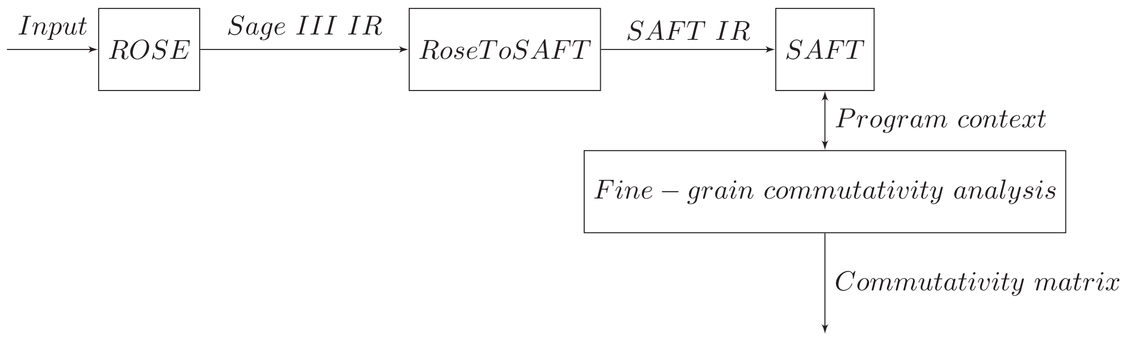


Figure 4.1: Implemented framework flowchart

commutativity analysis then work together with SAFT to produce the commutativity matrix for each block of statements.

The primitives in SAFT are optimized and its IL include the structures Basic, Group, If, Loop and Function.

Chapter 5

Results

The main result from this thesis is the symbolic analysis framework with the following properties:

- Support for C constructs with its own simplified IR.
- Modular design to support different compilers frontends.
- Performance enhanced context maps.
- Reader/Writer identification and minimization.
- Simplification & Reduction (Interpreter based) routines.
- Solves, when possible, recurrence equations to their closed form using Mathematica.

Corner cases of the analysis have been verified using microbenchmarks. Due to various issues (presented below) it has been impossible to verify the presented commutativity analysis approach on any larger piece of software, such as benchmark applications.

5.1 Microbenchmarks

The microbenchmarks include code-snippets to test that the symbolic simplifier/reducer is able to rewrite the state expressions to a form that allows the symbolic comparator to detect commutativity. Each microbenchmark includes one or a few cases that we expected it to find. This includes various recursive functions (such as naive Fibonacci sequence computation), loop constructs, basic statements (such as a variation of Listing 5.1) and kernels (such as IIR). Although the core functionality of SAFT is capable of analyzing and producing valid program contexts for almost all the microbenchmarks the simplifier/reducer is yet not capable normalizing the expressions to a form that yields successful comparison.
To emphasize further the problems, some of microbenchmarks are discussed together with their respective problem in Section 5.2. The results of the evaluation is presented in Section 5.3.

Listing 5.4, 5.2 and 5.1 (without pointers) are part of the microbenchmark suite together with seven other kernels, which although similar in structure includes statements that implement more complicated expressions for the symbolic engine to handle. As such all microbenchmarks are represented depending on how similar they are to one of the given three cases. Table 5.1 outlines the various kernels.

Microbenchmark ID	**Similar to/different from**
1	Listing 5.1 but only a basic block without pointers.
2	Listing 5.2.
3	Listing 5.2 but using arrays rather than scalars.
4	IIR kernel, similar to ID 3.
5	Listing 5.1 without pointers.
6	similar to ID 1 but uses pointers.
7	Listing 5.2 but more complicated.
8	Listing 5.1 but uses compound types such as structs and arrays.
9	Listing 5.4.
10	similar to ID 8 but more complicated.
11	similar to ID 7 but more complicated.

Table 5.1: Microbenchmark categorization

5.2 Problems

5.2.1 Alias analysis

Pointers are modeled as a set of variables in SAFT, writing to pointers are seldom expanded (as variables primitive types) but rather kept as a set to minimize storage space. Without alias analysis SAFT will not be able to deduce which variables a pointer could refer to and will hence operate on null-pointers. The implemented alias analysis in ROSE uses Stensgaard's algorithm, which unfortunately is too conservative to allow detecting the calls to `a()` in Listing 5.1 as commutative. Our approach can however detect commutativity if the definition and declaration of `a()` is changed so that the result is returned rather then passed through a pointer. Naturally this example could be reduced with aggressive constant propagation/folding to a single `printf("1");`, but it still illustrates the problem.

```
int global_1, global_2;

void a(int *d) {
  int tmp = global_2 % 2;
  if(tmp == 0) {
    (*d) = global_1;
  else
    (*d) = -(global_1 + 7 + tmp);
  global_1++;
}

void b(int e, int f) {
  return e-f;
}

int c(int g) {
  if(g < 0)
    return -g;
  return g;
}

int main() {
  int foo, bar;
  a(&foo);
  a(&bar);
  printf("%d", c(b(foo, bar)));
  return 0;
}
```

Listing 5.1: Example from [28]

```
int main() {
  int i, a = 0;

  for(i = 0; i < 10; i++) {
    a += func_a() - i; //pure function
  }
  a += func_b(); //pure function
  return a;
}
```

Listing 5.2: Multiple recurrence equation example

Due to issues with the alias analysis implementation support for pointers was restricted in SAFT and consequently, sourcecodes that include pointers are termed unanalyzable.

5.2.2 Recursive equations

Many of the recursive equations encountered from loops lack closed-form. As such they are hard to manipulate symbolically to discover commutativity. Another case is when a induction variable is updated by two variables in the loop such as in Listing 5.2. In this case statement 3 (the for loop) is commuting with the assignment in statement 6. Although recurrence equation of the loop counter (variable i) can be resolved to its final expression $n - 1$ (n is the loop iteration number) the reducer can't yet successfully combine the expression with that of the variable a to produce the final expression $\frac{41n-n^2}{2}$.
Listing 5.4 show the example of a naive Fibonacci sequence generator. Since SAFT currently does not include functionality to detect and handle recursive functions it is not able to compute the program context for this example. Instead it expands the function calls till the maximum call depth is reached, where it returns an uninterpreted function with the proper arguments as results. This expression can of course not be solved by the recurrence equation solver.
Although related to the issue with pointers, the problem of determining the index of an array access mostly manifests itself when the index is a recurrence equation. Consider Listing 5.3 which

is a simplified version of the critical loop in the kmeans application from the Rodinia benchmark suite [40]. The loop has loop-carried dependencies given that the variable `index` maybe equal in two different iterations. The problem lies in that the index cannot be resolved to a proper comparable expression.

```
    for(i = 0; i < LIMIT_1; i++) {

      index = func(...); //Pure function

      array1[index]++;
          for (j=0; j< LIMIT_2; j++)
        array2[index][j] += data[i][j];
    }

```

Listing 5.3: Array access example

```
int fib2(int n) {
  int a = fib(n-1);
  int b = fib(n-2);
  return a + b;
}

int fib(int n) {
  if(n == 0)
    return 1;
  else if(n == 1)
    return 1;
  else
    return fib2(n);
}
```

Listing 5.4: Recursive function example

5.3 Evaluation

Table 5.2 shows the results of evaluating the microbenchmarks on the given framework and algorithm. **Failed due to core** refers to errors within the program context generation engine. In this particular case (with the mentioned microbenchmark kernels) it refers to problems with pointers as well as recursive functions. **Failed due to simplifier/reducer** refers to cases where the simplifier/reducer was not able to normalize the state expressions sufficiently. **Failed since storage location could not be determined** refers to examples where for instance array indexes could not be determined and the framework reverted to conservative assumptions. **Successfully determined commutativity** is of course success in all stages of the framework.

Result status	List of microbenchmark IDs
Failed due to core	{6,9}
Failed due to simplifier/reducer	{2,3,4,7,11
Failed since storage location could not be determined	{4,8}
Successfully determined commutativity	{1,5}

Table 5.2: Evaluation results

5.4 Summery

In evaluating the algorithm and framework 11 microbenchmarks were created. These include one or more cases of commuting operations. Due to problems with alias analysis, pointers are not supported and recursive functions is yet not handled. Furthermore, the support for recurrence equations is yet limited and in particular poses problems when arrays are accessed by a loop variable.

Chapter 6

Conclusion

The benefit of multicores diminishes if applications can't take advantage of the parallel architecture. As such is it necessary to develop algorithms that are able to identify code-blocks that can run in parallel. This thesis has explored an approach to context insensitive and fine-grain commutativity analysis. Section 6.1 discusses improvements that can be done to improve the results of our approach. Section 6.2 reflects over the progress in this thesis and Section 6.3 concludes this thesis.

6.1 Future work

Although this thesis has addressed issues with commutativity analysis that were previously overseen there are still a number of issues to deal with. As seen the framework fails to discover commuting operations mostly due to limited symbolic simplification/reduction. This is obvious a question of accuracy versus speed, still one could have multiple algorithms to decide equality, ranging from fastest but least accurate to slowest but most accurate.
It is also a question of whether all failed reorderings should be reconsidered, if the comparator can tell that these expressions never will be the same. A common case is when the state expression is equal to a constant in one order but another expression in the opposite order. Since constants are static expressions such as these will always correspond to non-commutativity.
Finally, it is necessary to come up with decision rules on when to exploit detected commutating operations. As the cost of guaranteeing atomicity can be quite substantial such a decision may well have to be delayed till runtime when the environment is known.

6.2 Discussion and reflections

A lot of time of the thesis was devoted to on perfecting the SAFT modules. Design decisions critical to the overall running time and memory consumption was early prioritized to allow for scalability. Symbolic analysis is an elaborate method in many aspects; unfortunately it comes with multiple non-trivial design tradeoffs.

Although problems exist with the current framework, it was decided that these were outside the scope of this thesis.

6.3 Final thoughts

As seen more aggressive alternatives exist to dependency analysis, a keystone in compilation and particular parallelizing compilation. The cost of a less conservative analysis is in general a higher price in the form of execution time. Whether the gain outweight the cost depends on the execution context and the application. Even though commutativity analysis may well find additional opportunities the cost of synchronization to grantee atomicity of the dependent and commuting operations may be high. In the other extreme few/small commuting operations in a otherwise independent iteration space may well enhance execution time.
The algorithm introduced here can detect commuting operations even in the context where they are not immediately visible.

Bibliography

[1] Johan de Galas. The quest for more processing power: is the single core cpu doomed? http://www.anandtech.com/show/1645/3, February 2005.

[2] Wen mei Hwu, S. Ryoo, Sain-Zee Ueng, J.H Kelm, I. Gelado, S. S. Stone, R. E. Kidd, S. S. Baghsorkhi, A. A. Mahesri, S. C. Tsao, N. Navarro, S. S. Lumetta, M. I. Frank, and S. J. Patel. Implicitly parallel programming models for thousand-core microprocessors. In *ACM/IEEE Design Automation Conference*, 2007.

[3] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A quantitative approach*. Morgan Kaufmann, 2007.

[4] D. E. Culler and J. P. Singh. *Parallel computer architecture: A hardware/software approach*. Morgan Kaufmann, 1999.

[5] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Pearson/Addison-Wesley, 2007.

[6] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/363347.363387.

[7] J. E. Hopcraft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation (3:rd edition)*. Pearson/Addison-Wesley, 2006.

[8] John E. Hopcroft. An nlogn algorithm for minimizing states in a finite automaton. Technical report, Stanford University, Stanford, CA, USA, 1971.

[9] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures*. Morgan Kaufmann, 2001.

[10] A. J. Bernstein. Analysis of programs for parallel processing. *Electronic Computers, IEEE Transactions on*, EC-15(5):757 –763, oct. 1966. ISSN 0367-7508. doi: 10.1109/PGEC.1966.264565.

[11] M. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, 1996.

[12] Santosh Pande and Dharma P. Agrawal. *Compiler Optimizations for Scalable Parallel Systems Languages, Compilation Techniques, and Run Time Systems*. Springer, 2001.

[13] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Workshop on Program Analysis for Software Tools and Engineering*, 2001.

[14] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4): 323–337, 1992. ISSN 1057-4514. doi: http://doi.acm.org/10.1145/161494.161501.

[15] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3. doi: http://doi.acm.org/10.1145/237721.237727.

[16] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, DIKU, University of Copenhagen, Copenhagen, Denmark, 1994.

[17] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise call graphs for c programs with function pointers. *Automated Software Engineering*, 11:7–26, 2004. ISSN 0928-8910. URL `http://dx.doi.org/10.1023/B:AUSE.0000008666.56394.a1`. 10.1023/B:AUSE.0000008666.56394.a1.

[18] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Softw. Pract. Exper.*, 27(6):701–724, 1997. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/(SICI)1097-024X(199706)27:6⟨701::AID-SPE104⟩3.3.CO;2-S.

[19] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7.

[20] Mohammad R. Haghighat and Constantine D. Polychronopoulos. *Symbolic analysis for parallelizing compilers*. Kluwer academic, 1995.

[21] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/103135.103136.

[22] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17(2):181–196, 1995. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/201059.201061.

[23] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Pro- gramming Languages*, pages 238–252. ACM, 1997.

[24] Thomas Fahringer and Bernhard Scholz. *Advanced symbolic analysis for compilers: new techniques and algorithms for symbolic program analysis and optimization*. Springer-Verlag, Berlin, Heidelberg, 2003. ISBN 3-540-01185-4.

[25] Constantine D. Polychronopoulos, Milind B. Girkar, Mohammad Reza Haghighat, Chia Ling Lee, Bruce Leung, and Dale Schouten. Parafrase-2: an environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. *Int. J. High*

Speed Comput., 1(1):45–72, 1989. ISSN 0129-0533. doi: http://dx.doi.org/10.1142/S0129053389000044.

[26] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Interprocedural parallelization analysis in suif. *ACM Trans. Program. Lang. Syst.*, 27 (4):662–731, 2005. ISSN 0164-0925. doi: http://doi.acm.org.proxy.lib.chalmers.se/10.1145/1075382.1075385.

[27] Pedro C. Diniz and Martin C. Rinard. Commutativity analysis: a new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19 (6):942–991, 2000.

[28] Farhana Aleen and Nathan Clark. Commutativity analysis for software parallelization: letting program transformations see the big picture. In *International conference on Architectural support for programming languages and operating systems*, 2007.

[29] S. Gulwani. *Program analysis using random interpretation.* PhD thesis, University of California Berkeley, 2005.

[30] Louis-Nol Pouchet. *Iterative Optimization in the Polyhedral Model.* PhD thesis, University of Paris-Sud XI, Orsay, France, January 2010.

[31] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *Joint Modular Languages Conference held in conjunction with EuroPar'03*, 2003.

[32] Vijay Menon, Keshav Pingali, and Nikolay Mateev. Commutativity analysis: a new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 25(6):776–813, 2003.

[33] Lakshmi N. Chakrapani, John Gyllenhaal, Wen-mei W. Hwu, Scott A. Mahlke, Krishna V. Palem, and Rodric M. Rabbah. Trimaran: An infrastructure for research in instruction-level parallelism. In Rudolf Eigenmann, Zhiyuan Li, and Samuel P. Midkiff, editors, *Languages and Compilers for High Performance Computing*, volume 3602 of *Lecture Notes in Computer Science*, pages 32–41. Springer Berlin / Heidelberg, 2005.

[34] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9.

[35] George S. Lueker. Some techniques for solving recurrences. *ACM Comput. Surv.*, 12(4): 419–436, 1980. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/356827.356832.

[36] Roberto Bagnara, Andrea Pescetti, Alessandro Zaccagnini, and Enea Zaffanella. Purrs: Towards computer algebra support for fully automatic worst-case complexity analysis. *CoRR*, abs/cs/0512056, 2005.

[37] Stephen Wolfram. *The Mathematica Book, Fifth Edition.* Wolfram Media, August 2003. ISBN 1579550223.

[38] Chikara Miyaji and Paul Abbott. *Mathlink Network Programming in Mathematica*. Cambridge University Press, New York, NY, USA, 2001. ISBN 0521641721.

[39] M. Scheibl, A. Celic, and T. Fahringer. Interfacing mathematica from the vienna fortran compilation system. Technical report, University of Vienna, December 1996.

[40] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC '09: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-5156-2. doi: http://dx.doi.org/10.1109/IISWC.2009.5306797.

[41] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. Sd-vbs: The san diego vision benchmark suite. In *IISWC '09: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-5156-2. doi: http://dx.doi.org/10.1109/IISWC.2009.5306794.