

RICE UNIVERSITY

**Parameterization and Adaptive Search for Graph  
Coloring Register Allocation**

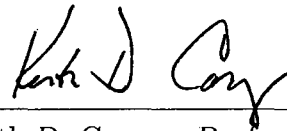
by

**Donghua Liu**

A THESIS SUBMITTED  
IN PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE

**Master of Science**

APPROVED, THESIS COMMITTEE:



---

Keith D. Cooper, Professor, Chair  
Computer Science



---

William Scherer, Faculty Fellow  
Computer Science



---

Vivek Sarkar, Professor  
Computer Science

Houston, Texas

October, 2009

UMI Number: 1486055

All rights reserved

**INFORMATION TO ALL USERS**

The quality of this reproduction is dependent upon the quality of the copy submitted.

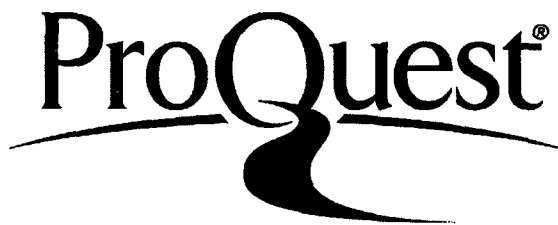
In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI 1486055

Copyright 2010 by ProQuest LLC.

All rights reserved. This edition of the work is protected against unauthorized copying under Title 17, United States Code.



ProQuest LLC  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106-1346

# Parameterization and Adaptive Search for Graph Coloring Register Allocation

Donghua Liu

## Abstract

Graph coloring register allocators use heuristics for register coalescing and allocation, which are relevant to the number of physical registers that a group of virtual registers will use after allocation. They cannot be determined accurately in allocation, thus we made them tunable by introducing new parameters as the thresholds for coalescing and the thresholds for defining constrained live intervals in simplification. Experiments demonstrated neither the aggressive method nor the conservative method can outperform the other for all tests and the best parameters vary significantly among programs. This parameterization is profitable because the best running time reached by varying the parameters is up to 16% faster than the best of fixed-parameter methods. Hill-climbing and random probe algorithms were used to find good parameters, and the later performed better. Further analysis reveals the search space has many irregular fluctuations that are not suitable for the hill-climber.

## Acknowledgments

Firstly, I wish to thank Prof. Keith D. Cooper, the committee chair, who taught me compiler courses and directed my research. Prof. Cooper showed me the beauty of compiler world through comp 412 and 512 courses. He is the best teacher I ever met - you can learn not only knowledge, but also passion. In my research, he gave me huge help and encourage. Each time after we talked, I saw the light leading out of swamp. Although being very busy, Prof. Cooper scrutinized my thesis and gave many comments. I learned many from him, through what he taught and what he did.

I also express my appreciation to other committee members, Prof. Vivek Sarkar and Dr. Bill Scherer. They reviewed my work and gave advices on some issues, necessary for the final completeness of the thesis. Prof. Sarkar's comment on complexity let me pay attention to this issue to avoid unclearness. I also need to mention Prof. Kenneth Kennedy. I took his comp 515, an compiler course on vectorization and parallelization, and learned precious knowledge that is hard to get elsewhere. Although he passed away, his contribution and personality will be remembered forever.

People who supported my study and research behind must receive praise. Ms. Belia Martinez helped the whole thesis procedure and other graduate student affairs. Ms. Darnell Price and Ms. Lena Sifuentes processed the administrative paperwork during my whole study. Dr. Adria Baker and Ms. Lily Lam from the Office of International Students & Scholars took all efforts to maintain my international student status. Their excellent work made me concentrate on my study and work.

Finally, I must thank my wife and my parents, who are a part of my life. Without the understanding, encourage and love from them, I cannot be here. I owe them a lot, far beyond a thank.

# Contents

Abstract	ii
Acknowledgments	iii
List of Illustrations	vi
List of Tables	viii
<b>1 Introduction</b>	<b>1</b>
1.1 Register Allocation in Compilers . . . . .	1
1.2 Adaptive Compilers and Tunable Parameters . . . . .	2
1.3 Goals and Tasks . . . . .	2
<b>2 Background</b>	<b>4</b>
2.1 Graph-Coloring Register Allocation . . . . .	4
2.2 Complexity in Graph Coloring Register Allocation . . . . .	8
2.3 LLVM Compiler . . . . .	9
2.4 Engineering Environment . . . . .	10
<b>3 Tunable Parameters for Graph Coloring Register Allocator</b>	<b>16</b>
3.1 Implementation of the Register Allocator . . . . .	16
3.2 Dominative Choices on Parameters . . . . .	21
3.3 Tunable Parameters for Performance . . . . .	22
<b>4 Properties of the Performance on Parameter Space</b>	<b>24</b>
4.1 Experiments for Data Analysis . . . . .	24

4.2	Performance Range Analysis . . . . .	25
4.3	Performance Tendency Analysis . . . . .	31
4.4	Compilation Procedure Analysis . . . . .	39
4.5	Independence between Parameters . . . . .	41
<b>5</b>	<b>Adaptive Search with Tunable Parameters</b>	<b>44</b>
5.1	Hill Climbing Search . . . . .	44
5.2	Experiments and Analysis . . . . .	47
5.3	Performance Stability over Parameter Changes . . . . .	60
<b>6</b>	<b>Summary and Conclusions</b>	<b>69</b>
	<b>Bibliography</b>	<b>71</b>

# Illustrations

2.1	Integer Registers for Allocation . . . . .	12
2.2	Average Running Time vs the Number of Simultaneous Processes . .	15
3.1	Graph-coloring register allocator . . . . .	16
4.1	Mesh Graphs of scimark Running Time . . . . .	33
4.2	Mesh Graphs of pift Running Time . . . . .	34
4.3	Mesh Graphs of nasa Running Time . . . . .	35
4.4	Mesh Graphs of spice Running Time . . . . .	36
4.5	Mesh Graphs of analyzer Running Time . . . . .	37
4.6	Mesh Graphs of lambda Running Time . . . . .	37
4.7	Mesh Graphs of llu Running Time . . . . .	38
4.8	Mesh Graphs of sim Running Time . . . . .	38
4.9	Correlation between the interval statistics and running time for scimark	40
4.10	Performance graphs for different floating-point parameters . . . . .	42
4.11	Performance graphs for different integer parameters . . . . .	43
5.1	Hill Climbing Algorithm . . . . .	45
5.2	Value on each step in a searching . . . . .	49
5.3	Value pair (#steps, best value) of every restart . . . . .	52
5.4	Histogram for distribution on best values . . . . .	53
5.5	Histogram for distribution on steps in a restart . . . . .	53

5.6	Histogram and probability distribution of random points in scimark . . .	58
5.7	Histogram and probability distribution of random points in pifft . . .	58
5.8	Histograms of random points in spice and nasa . . . . .	59
5.9	Probability of reaching top 10% . . . . .	60
5.10	Histogram of the degree of all initial intervals . . . . .	68



## Tables

2.1	Benchmark Programs . . . . .	11
2.2	Errors in running time measurement for scimark . . . . .	14
2.3	Errors in running time measurement for pifft . . . . .	15
4.1	Running time on parameter space for integers and floats . . . . .	28
4.2	Running time on parameter space for integers only . . . . .	29
4.3	Comparing the best results in $[0, 20] \times [0, 20]$ parameter space with the conservative and aggressive coalescing methods . . . . .	30
5.1	Hill-Climbing searching on scimark at 25%, 50%, 100% patience . . . . .	50
5.2	Hill-Climbing searching on pifft at 25%, 100% patience . . . . .	54
5.3	Hill-climbing performance for scimark . . . . .	56
5.4	Hill-climbing performance for pifft . . . . .	56
5.5	$\Delta$ of spilled intervals in scimark when coalescing threshold changes . . . . .	64
5.6	$\Delta$ of spilled intervals in scimark when simplifying threshold changes . . . . .	64
5.7	$\Delta$ of coalesced intervals in scimark when coalescing threshold changes . . . . .	65
5.8	$\Delta$ of coalesced intervals in scimark when simplifying threshold changes . . . . .	65
5.9	$\Delta$ of spilled intervals in pifft when coalescing threshold changes . . . . .	66
5.10	$\Delta$ of spilled intervals in pifft when simplifying threshold changes . . . . .	66
5.11	$\Delta$ of coalesced intervals in pifft when coalescing threshold changes . . . . .	67
5.12	$\Delta$ of coalesced intervals in pifft when simplifying threshold changes . . . . .	68

# Chapter 1

## Introduction

### 1.1 Register Allocation in Compilers

A compiler transforms the source code of a program into another form, mostly, in a machine-executable format. As part of a compiler, the register allocator plays dual roles for code-generation and performance optimization. In most programming paradigms, programmers can define an unlimited number of variables, but processors only provide a limited set of physical registers. Thus, the register allocator must determine which variables will reside in the physical registers and which register will hold each variable. In most cases, the physical registers cannot hold all variables through the lifetime of the program execution, so some variables must be spilled into memory when the physical registers are not sufficient and be reloaded when they are used again.

From the programming view, most instructions in a program are operations on single or multiple variables; from the hardware view, the access time of memory is much greater than the access time of registers. Thus, the quality of register allocation has a strong impact on the execution performance of the compiled code. A global register allocator is not easy, because of the complexity of control flow structures. Theoretically, a global register allocation problem can be transformed into a graph coloring problem. And coalescing and spilling are often needed, increasing the complexity of allocation. Unfortunately, the graph coloring problem is NP-complete, so approximate heuristic techniques are used to solve the problem.

## 1.2 Adaptive Compilers and Tunable Parameters

Traditional compilers using fixed optimizations are tuned for a good average performance on a large body of programs, but often fail to reach the best result for an individual program or for different optimization goals, such as code size, memory limitation, or power consumption. To solve this problem, adaptive compilers that can change their optimization method for specific applications or target hardware have been studied. Researchers have demonstrated the adaptive method can produce better code relative to an external objective function than traditional fixed-behavior compilers. Most of the work focused on selecting an optimal sequence of optimizations for a specific application or a specific optimization goal, and has gained significant improvements through adaptive selection of optimizations [19, 20, 1, 2, 25, 15, 26, 16, 18].

However, these adaptive systems are based on existing compiler platforms, which are not designed for use in an adaptive compiler. For example, they do not expose adequate parameters to allow an adaptive compiler to vary significantly the behavior of the optimizations so that it can make larger improvements. More fine-grained controlling parameters for the compiler are necessary. This approach also means a challenge for the algorithms used in adaptive compilers, because there are more parameters to select and tune. First, we need to revise the existing algorithms to provide an effective parameterization and construct an expressive scheme to expose those parameters for explicit control in adaptive compilers. Next we need to test search approaches on the parameterized algorithms to evaluate the effect. In this thesis, we will focus on a representative register allocation method, the graph-coloring register allocation with live range coalescing, to find the tunable parameters and evaluate its performance.

## 1.3 Goals and Tasks

The general goal of this work is to parameterize the existing graph-coloring register allocator and find an effective strategy for setting those parameters. This work

consists of three parts.

First, we will review the graph-coloring register allocation algorithm to find parameters that can be varied. Then, we will examine if the parameters have a significant impact on code quality. If varying a parameter cannot change the results significantly, it is not profitable. If one choice on a parameter outperforms other choices for most programs, the compiler should fix that good choice in the algorithm, thus there is no need for adaptation. If some choices on a parameter may win over other choices for some programs but cannot win for most programs, this parameter may be a good candidate for parameterization. When finishing this work, we will have a list of parameters for adaptive register allocation.

Next, we will study the impact of the parameters. We will explore the parameter space to find its properties. This study will show the theoretical maximum improvement and the difficulty of reaching it. It will also reveal if the parameterization is profitable enough to pursue. Also, the study on the structure of the parameter space can expose relationships among the parameters and the changing tendency over the parameters, which may help us on searching algorithms.

Finally, we will run some search methods on the new tunable parameters, and know how fast they can reach good results. The search may need the knowledge from the previous steps and it may give feedback for the parameterization. The parameterization creates the possibility for improvement; the search turns it into reality. If the adaptive compiler cannot do the final job well, the parameterization is not practical. As a result of tradeoff, we may try to get back a little from the best possible result but get closer to the best feasible or acceptable result, or shrink the parameter space for a quicker search.

## Chapter 2

### Background

#### 2.1 Graph-Coloring Register Allocation

The task of a register allocator is to map the variables defined and used in a program to physical registers. In general, we call the variables as *virtual registers*, in contrast with physical registers. Because the physical registers are scarce resources, the register allocation is not an easy job. A naive method can do it by storing a value residing in a physical register to memory to make a physical register available, and reloading it later when it is accessed again. Obviously, the performance of the method is very poor, due to frequent memory loads and stores. Based on a model that precisely describes the relations among virtual registers and physical registers, a graph-coloring register allocator can produce better results. Because the graph-coloring problem is intractable, the graph-coloring register allocator solves the problem with heuristics.

To formulate this, we need to introduce some concepts for virtual registers. In a well-formed program, a virtual register must be defined or assigned a value at first; and there should be some uses after the assignment. We say a virtual register is *live* between its definitions and its uses along the program execution paths, and the entire region where the virtual register is live is its *live range (LR)* (or *live interval*). Because programs have control structures, the live range often is not a single straight segment, which means, it may branch and join together. No matter how complicated the structure of a live range is, whether or not two live ranges are both live at some point is definite. If their life ranges overlap, we say they *interfere*. Thus, we can derive an *interference graph (IG)* that encodes the interferences between any two live ranges. In the interference graph, each node represents a live range, and if two

live ranges interfere, the two corresponding nodes are connected by an edge. If we use different colors to represent the different physical registers, the register allocation needs to color the nodes and ensure any two connected nodes do not receive the same color. In this way, we model register allocation as a graph-coloring problem.

In graph theory, a  $k$ -coloring of a graph is an assignment of  $k$  colors to the nodes, such that adjacent nodes are assigned different colors. The minimum  $k$  for which a  $k$ -coloring exists is known as the *chromatic number* of the graph. If there are  $k$  physical registers, then a  $k$ -coloring of the interference graph shows how to allocate live ranges to physical registers in a way that avoids spilling any live range to memory. But finding the minimal graph coloring is NP-complete, though some powerful heuristics for efficient coloring exist in practice.

In many cases, the chromatic number of the IG is greater than the number of available physical registers, so some virtual registers must be spilled into memory. The choice of which registers should be stored/reloaded and when they should be stored/reloaded has a heavy impact on performance. The dynamic nature of the real stores/reloads adds more difficulty for this problem.

Chaitin *et al.* were the first to implement a graph-coloring register allocator [14, 13]. Chaitin's coloring heuristic is simple and relies on the graph theoretic property:

*Given a graph  $G$  and a node  $v$  such that  $\text{degree}(v) < k$ , then  $G$  is  $k$ -colorable if and only if  $G - v$  is  $k$ -colorable.*

Chaitin's algorithm uses this property to recursively *simplify* the interference graph by removing *unconstrained* nodes ( $\text{degree} < k$ ) until the graph is empty or all the remaining nodes in the reduced graph are *constrained* ( $\text{degree} \geq k$ ). If the graph becomes empty, the algorithm inserts the removed nodes into the graph in the reverse order of removing (popped off from a stack), and assigns each node a color not used by any of its neighbors. The above graph theoretic property guarantees that a color is available for each node.

Often, however, the graph cannot be reduced to empty. Here, Chaitin's algorithm

assumes the graph is not  $k$ -colorable pessimistically, and selects one of the constrained nodes and removes it from graph, marking it for later spilling. The heuristics for this spilling are minimizing the spilling cost and reducing the degrees of other nodes mostly, so the node associated with the smallest spilling cost divided by the current degree is selected at first. After the node is removed and marked, the simplification procedure may continue until another node must be marked for spilling or the graph becomes empty. Eventually, the graph will be empty. Spill code will be inserted for the live ranges representing the nodes marked for spilling. Because the spill code uses some physical register resources, some unconstrained node before inserting spill code may become uncolorable. The entire process of building interference graph and simplifying will be repeated until no further spilling happens. Typically, this process converges in two to four passes.

Chaitin's assumption that a node with  $k$  or more neighbors is uncolorable is pessimistic, because some nodes among the neighbors may receive the same color and make the node become colorable finally. Briggs *et al.* proposed an improvement, *optimistic coloring* [10, 8, 11], by removing the pessimistic assumption. In Briggs' algorithm, a node is considered uncolorable only if its neighbors have used all colors. Instead of marking a constrained node for spilling, Briggs' method optimistically places it into the stack, just as it puts the unconstrained nodes into the stack. After the graph becomes empty, Briggs' method tries to find colors for the nodes, by popping them one by one from the stack and looking for a color. For constrained nodes, this method still has a chance to find a color for them, based on the coloring of their neighbors. If a node does not receive a color, it will be marked for spilling and will not be inserted back into the graph. After the stack is empty, spill code will be inserted for the nodes marked for spilling. The optimistic method may color more live ranges than the pessimistic method.

There is another important transformation on live ranges that the allocator can perform, named *coalescing*. If two live ranges are joined by a move instruction and

they do not otherwise interfere, they can be coalesced. After coalescing, the two live ranges are joined into one, and the copy instruction is eliminated. Coalescing can decrease memory accesses directly by eliminating copy instructions. The impact of coalescing can be considerable. Briggs showed some examples where coalescing eliminates up to one-third of live ranges [8]. Besides this, coalescing has both negative and positive impacts on the colorability of the interference graph. The coalesced node often has a greater degree than each of the nodes before coalescing, so the new coalesced node may become uncolorable. On the other hand, coalescing may reduce the degree of adjacent nodes in the graph, increasing the chance that those nodes receive a color. It is hard to predict the overall effect of coalescing.

Chaitin used an *aggressive coalescing* method that coalesces all possible live ranges. Briggs *et al.* proposed an approach named *conservative coalescing* [11]. If the degree of coalesced node is less than the number of colors, the coalescing is always beneficial, since it cannot make the coalesced node uncolorable. To avoid the possibility of making the coalesced node uncolorable, the conservative coalescing coalesces two live ranges only when the conservative condition is satisfied. George and Appel introduced *iterated coalescing* [24], which performs conservative coalescing iteratively, interleaved with simplify stage to expose more chances for conservative coalescing.

As we have seen, the number of colors or the number of available physical registers,  $k$ , plays an important role in coloring and coalescing. In both coloring and coalescing, we use  $k$  for the assumptions that guarantee that the register allocator makes “safe” decisions. More precisely, in coloring, the best  $k$  should be related to the number of the distinct colors of the neighbors, not the number of the neighbors. The case in coalescing is similar. But the problem is when we need the value about the  $k$ , we do not know about neighbor’s colors.

This thesis focuses on improvements that can be made by considering  $k$  as a tunable parameter. The experimental results in the next chapters will show that adjusting  $k$  can lead to significant improvements and that the appropriate value of  $k$



varies from procedure to procedure.

## 2.2 Complexity in Graph Coloring Register Allocation

The introduction above showed the transformation from a register allocation problem into a graph coloring problem. Chaitin *et al.* demonstrated that all graphs can arise from register allocation, and proved the general register allocation problem is NP-complete [14].

There are some transformations on the live ranges before the register allocation pass, such as the transformation into SSA form [3, 34, 21, 9]. These transformations may add constraints to interference graphs such that they are a proper subset of arbitrary graphs. For example, the interference graph based on post-SSA live ranges is a chordal graph [6, 12, 31, 27] and there exists a polynomial algorithm to color such graphs [23]. More strongly, the set of all chordal graphs and the set of the interference graphs of all SSA-form programs are equal [32]. However this does not mean the coloring of the interference graph derived in original register allocation problem is tractable. Because the SSA transformation cannot be reversed precisely, the coloring problems for the original interference graph and the post-SSA interference graph are not equivalent.

No matter whether it is SSA-based, the register allocation in real world often requires spilling, and the optimization goal is to minimize the spilling cost. This optimization problem is NP-complete [22]. If we only consider the number of spilled live ranges instead of the spilling cost, this problem is still NP-complete for chordal graphs. But for interval graphs, it can be solved in polynomial time [39].

The coalescing is like an inversion of splitting. Bouchez *et al.* proved that all the best known variations of the coalescing problem are NP-complete [7].

Other requirements may add complexity. The pre-colored registers and register aliasing result in NP-complete complexity of register allocation, even if it is SSA-based [29, 5, 30]. The register aliasing is a source of the irregularity of register files

and makes it hard to estimate the number of available physical registers, and we will discuss it in the following sections.

## 2.3 LLVM Compiler

LLVM (*Low Level Virtual Machine*) [28] is a compiler framework designed to support transparent, life-long program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs. This framework defines a common, low-level code representation in Static Single Assignment (SSA) form, as the key to support its capabilities. The LLVM toolkit is implemented in the C++ language.

LLVM has a pass manager and implements many passes, such as the passes for optimization, code-generation, and auxiliary support. In its implementation, a linear scan register allocation algorithm [33] is used for global register allocation, for better just-in-time performance. we implemented a graph-coloring register allocator, which uses support from some LLVM analysis passes, including *LiveVariables*, *PhiElimination*, *TwoAddressTransform* and *LoopInfo*.

The *LiveVariables* pass derives variable information from an SSA structure, containing the definition of a node, the basic blocks it lives through, and the killing sites.

The *PhiElimination* pass eliminates the necessity of *PhiFunctions* in SSA by adding copy instructions at the end of the predecessor basic blocks of the *PhiFunctions*. So it creates lots of live ranges connected by copy instructions, increasing the chances and importance of coalescing.

The *TwoAddressTransform* pass transforms three-address instructions used in the intermediate representation of LLVM into two-address instructions used by the machine instruction on some target platforms, such as X86. *PhiElimination* and *TwoAddressTransform* break SSA properties, because they introduce multiple definitions.

The *LoopInfo* is used to compute the depth of loop for all instructions, which is

required for spill cost calculation.

The graph-coloring register allocator also needs a data structure to maintain live ranges, which is a group of live variables connected by PhiFunctions. A live range stores a vector of live ranges addressed with instruction indexes. The live range analysis is implemented inside the graph-coloring register allocation pass, for easy dynamic updating. This register allocator also needs an interference graph, which is maintained as neighbor vectors in all live range objects in the implementation.

## 2.4 Engineering Environment

### Hardware/Software Environment and Benchmark Programs

All experiments are run on an Intel Core 2 Quad CPU Q6700 2.66GHz machine running a 32-bit Redhat Linux operating system. The running time is based on the user time outputted by *time* program under linux. The experiments and evaluation will be based on the running time mainly, since it is single-valued and most directly related to performance. Other measurements are collected for evaluation and analysis, when needed.

The LLVM version in use is 1.9. It uses the *llvm-gcc* front-end to transform C/C++ source code to LLVM byte code format, for later use by LLVM compiler *llc*. For the source code in Fortran, we used *f2c* to transform them into C language at first.

The benchmark programs come from SPEC 95 and the LLVM test suite, including four programs that have many floating-point operations and four programs that have few floating-point operations. The following table lists the program names, the operation types, the test suites where they come from, the programming languages, the number of source code lines, and the brief descriptions.

Name	Type	Test Suite	Language	Lines	Intervals
analyzer	Integer	LLVM/freebench	C	923	498
	A dependency analyzer				
lambda	Integer	LLVM	C++	2182	2896
	Lambda calculus interpreter				
llu	Integer	LLVM	C	191	185
	A linked list traversal micro-benchmark				
sim	Integer	LLVM	C	1596	2214
	Finding k best non-intersecting alignments between two sequences or within one sequence using dynamic programming techniques				
nasa	Integer +Float	SPEC 95	Fortran	1106	3183
	7 kernels for numerical computing that are heavily floating point intensive				
pifft	Integer+Float	LLVM/freebench	C	4185	5223
	Calculation of PI(= 3.14159...) using FFT				
scimark	Integer +Float	LLVM	C	1233	903
	SciMark2 Numeric Benchmark				
spice	Integer +Float	SPEC 95	Fortran	18414	31173
	General purpose circuit simulation program for nonlinear dc, non-linear transient, and linear ac analyses				

Table 2.1 : Benchmark Programs

### Available physical registers for register allocation in the programs

The running environment is a 32-bit X86 machine, which provides 8 32-bit integer registers, 8 80-bit floating-point registers ST0 through ST7, and 8 128-bit SSE registers XMM0 through XMM7.

The integer registers are complicated because of different register sizes and shared hardware units. The 32-bit registers also provide spaces for 16-bit registers and 8-bit registers. Each 32-bit register can also be used to hold one 16-bit register. For example, EAX provides the space for AX. Each of the four general-purpose 16-bit registers can also be used to accommodate two 8-bit registers. Thus, some registers share the same physical units, and exclude the use of other registers that share the same space. And the two 8-bit registers in the same 16-bit registers do not exclude each other. Figure 2.1 illustrates the physical layout of these registers.

#### General Purpose Registers (EAX, EBX, ECX and EDX)

byte 3	byte 2	byte 1	byte 0
<i>EAX EBX ECX EDX</i>			
NOT USED		<i>AX BX CX DX</i>	
NOT USED		<i>AH BH CH DH</i>	<i>AL BL CL DL</i>

#### Pointer Registers (ESP and EBP)

byte 3	byte 2	byte 1	byte 0
<i>ESP EBP</i>			
NOT USED		<i>SP BP</i>	

#### Index Registers (ESI and EDI)

byte 3	byte 2	byte 1	byte 0
<i>ESI EDI</i>			
NOT USED		<i>SI DI</i>	

Figure 2.1 : Integer Registers for Allocation

Beside four general-purpose integer registers, two index registers can also be used for general allocation. In LLVM compiler, pointer register ESP cannot be used for general allocation; but the other pointer register EBP can be used for general allocation if the specified function does not need a dedicated frame pointer register. When the function has variable sized allocas or frame pointer elimination is disabled through command-line argument, a dedicated frame pointer is needed and the register EBP is used as the dedicated frame pointer, because the address in the frame cannot be determined at the compilation time.

The floating-point registers ST0 through ST7 are not accessible directly, but are accessible as a LIFO stack. However, the register allocator allocates all registers as they can be accessed directly, so the allocator only use 7 such registers, with one reserved for later FP stackifier pass that transforms the direct register access into stack mode with the help of the reserved register. If the SSE registers are available, the LLVM compiler only uses the eight SSE registers for floating-point allocation. Because the hardware in use supports SSE registers, the register allocator used the eight SSE registers in the experiments.

These complexities in physical registers add some difficulties to register allocation. For example, we cannot use a fixed value for  $k$ , the number of allocatable integer registers. Such irregular architectures exist in other target machines, so register allocation algorithms and implementations should consider them [37, 35, 36, 38]. In the terms used for generalized register allocation, *RegisterClass* refers to all registers that have the same type and size, and *AliasSet* means all registers that share the same physical space. For example, in X86,  $\{AL, AX, EAX\}$  makes an *AliasSet*, and  $\{AH, AX, EAX\}$  is also an *AliasSet*. *AH* and *AL* are not in the same *AliasSet*, but belong to the same *RegisterClass*. All registers in a *RegisterClass* provide the same functionality, so a virtual register can be allocated to any available register in a *RegisterClass*. All registers in an *AliasSet* are names for the same physical register unit, so only one name from the set can be used. Since the *RegisterClass* describes

the type and size properties of registers, it is convenient to use the RegisterClass designation on virtual registers as well.

### Measurement errors in experiments

The measurement of running time may be disturbed, resulting in errors of the performance results. In order to evaluate the effect of such errors, we ran scimark with 5 different sets of the parameters for integer repeatedly and continually. This test is run under three modes: single process for 8 times, two simultaneous processes for 4 times on every processing unit, and four simultaneous processes for 2 times on every processing unit. The following table contains the statistics from the measured results. According to the data, for most cases, the error range is less than 0.82%, but some relatively larger error appears occasionally, which is the reason for some big values ( $> 2.75\%$ ) of the  $\frac{\text{max}-\text{min}}{\text{min}}$ .

params	1-simultaneous			2-simultaneous			4-simultaneous		
	mean	stderr	$\frac{\text{max}-\text{min}}{\text{min}}$	mean	stderr	$\frac{\text{max}-\text{min}}{\text{min}}$	mean	stderr	$\frac{\text{max}-\text{min}}{\text{min}}$
{0,0}	29.2063	0.0211	0.20%	29.2028	0.0697	0.72%	29.2345	0.0329	0.34%
{0,16}	26.4126	0.2391	2.75%	26.3150	0.0490	0.57%	26.3334	0.0734	0.82%
{16,0}	26.8559	0.6644	7.37%	26.6241	0.0561	0.66%	26.6985	0.0681	0.77%
{16,16}	30.0738	0.0483	0.50%	30.0936	0.0295	0.30%	30.1023	0.0313	0.29%
{8,8}	25.7332	0.0520	0.55%	25.8278	0.2685	3.02%	25.7890	0.0484	0.54%

Table 2.2 : Errors in running time measurement for scimark

We noticed the simultaneous execution of scimark does not affect the performance. But there are large errors from simultaneous execution in another program. The following table is from the similar running of program pifft. The standard error is still low, but the average running time changes greatly when the number of simultaneous processes changes. The greater the number of simultaneous processes is, the slower the program runs. The following figure shows it clearly. We also found that though the performance changes over the degree of simultaneity, the error for multiple runs

under one mode is still low. Thus, the experiments on this program should be run in non-simultaneous mode, in order to output comparable results.

params	1-simultaneous			2-simultaneous			4-simultaneous		
	mean	stderr	$\frac{max-min}{min}$	mean	stderr	$\frac{max-min}{min}$	mean	stderr	$\frac{max-min}{min}$
{0,0}	23.2684	0.0435	0.53%	25.2118	0.0973	1.02%	41.9229	0.1080	0.73%
{0,16}	28.7048	0.0399	0.45%	29.9639	0.0896	0.81%	43.6674	0.1528	1.17%
{16,0}	22.4638	0.0568	0.85%	24.5705	0.1183	1.11%	41.5265	0.1245	0.80%
{16,16}	25.6861	0.0460	0.48%	27.2284	0.1723	1.61%	42.6561	0.1064	0.73%
{8,8}	22.6915	0.0306	0.42%	24.7787	0.1193	1.30%	41.7195	0.1143	0.90%

Table 2.3 : Errors in running time measurement for pifft

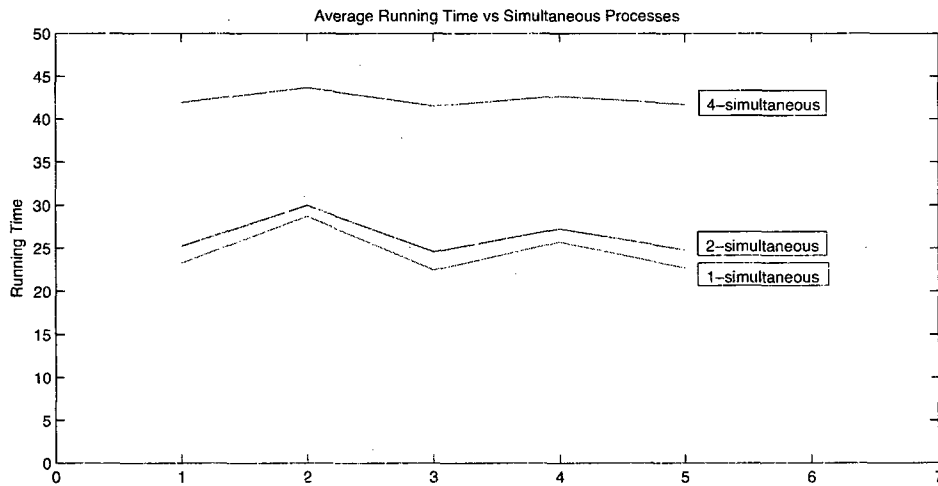


Figure 2.2 : Average Running Time vs the Number of Simultaneous Processes



## Chapter 3

# Tunable Parameters for Graph Coloring Register Allocator

### 3.1 Implementation of the Register Allocator

In order to find tunable parameters, we need to get into the details of the graph coloring register allocation algorithms. The implementation details of the algorithm and the hardware environment can help us to understand the factors that might introduce good parameters. The implementation uses some facilities provided by the LLVM platform. This section will present the implementation details, the opportunities for tunable parameters, and the dominative choices.

Figure 3.1 shows the structure of the graph-coloring register allocator in our implementation. We will discuss the implementation details in this section.

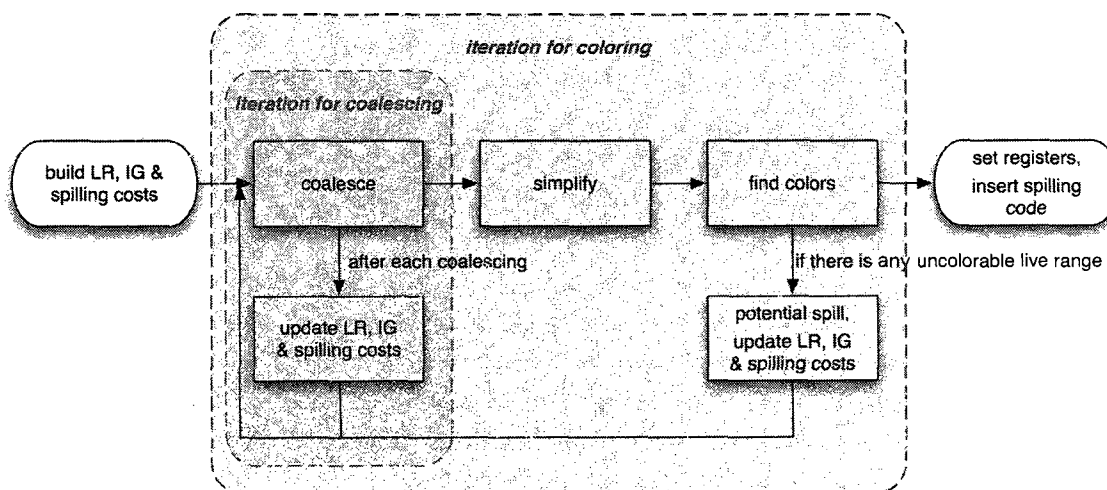


Figure 3.1 : Graph-coloring register allocator

## Building Live Ranges

The compiler scans all the basic blocks in a function and all the instructions in each basic block for definitions. Once a definition is found, it uses *LiveVariables* information to find where it is live and inserts the ranges into live range structure. If the live range for this definition has been initialized, which may result from multiple definitions generated by either *PhiElimination* or *TwoAddressTransform*, then the compiler combines the live ranges for these definitions together.

When a live range is spilled, it becomes a series of tiny ranges. The allocator directly updates the live range structure to reflect these changes, so there is no need to re-generate the live ranges.

The data representation used in live ranges can facilitate interference computation. Because it uses numbered indexes to describe the ranges, number comparisons are enough for interference computation. The overlapping method is not the most precise method for interference computation, but it is simple enough.

## Building the Interference Graph

Interference can be calculated by checking the overlap between live ranges. But in the real world, determining interferences is not so easy. On the X86, for example, the registers do not have identical functions. The LLVM compiler uses *RegisterClass* and *AliasSet* to define the relations among registers. If two virtual registers belong to the same *RegisterClass*, they may interfere. But this is not adequate. We need to define a new concept, *ConflictSet*, which contains all the registers that may occupy the same physical unit. The *ConflictSet* can be derived from *RegisterClass* and *AliasSet*. Put all *RegisterClasses* and *AliasSets* together, and if there are two sets that have at least one element in common, replace the two sets with the union of them, until no more union can be made. Strictly, if two intervals belong to one *ConflictSet*, they may interfere, and if not, they never interfere. For 32-bit X86 target machine, there are two *ConflictSets*, integer and floating point. Thus, the allocator can build separate

interference graphs for each ConflictSet, significantly reducing the cost of building the interference graphs. [17]

### Calculating the Degree in Interference Graph

The computation of “degree” in interference graph is a little subtle. According to the usage of the concept, the term “degree” is not the same as the degree in graph theory, because this “degree” is used to determine how many physical register units can accommodate its neighbors. The difficulty comes from two reasons - one is the complex relation among physical registers, the other is some neighbors may be allocated to the same physical register.

Because of the use of RegisterClass and AliasSet, the precise calculation is complicated. Allocating a virtual register to different physical registers may result in different numbers of available registers. For example, allocating two 8-bit virtual registers to  $\{AH, AL\}$  and  $\{AL, BL\}$  leads to different numbers of available 16-bit and 32-bit physical registers. Furthermore, even if we know how many physical registers are used for the neighbors, we do not know whether the current register can be allocated in some cases. For example, if all 32-bit physical registers have been used and the virtual register that needs to be allocated is an 8-bit register, then this register still has a chance to be allocated, in case that one of physical registers only contains one 8-bit virtual register. To handle such cases, the allocator needs detailed knowledge about the current register allocation. But in practice, we often ignore this, because it is not realistic and the advantage is small.

After allocation, some neighbors may receive the same physical register, so the real physical unit occupation may be less than the earlier estimate. If the estimate of the number of physical registers that will be used by a group of virtual registers could be more accurate, the chance of profitable coalesces will increase. More coalesces may lower the demand of registers. The difference between the number of used physical registers and the number of virtual registers can help improve the estimate, but the

difference varies among programs, and even within a single. A guess can be made for the number of physical register needed. Even an upper bound or a lower bound may be helpful, because sometimes it can lead to a “safe” decision. If the lower bound is greater than the number of physical registers, it is uncolorable; if the upper bound is less than the number of physical registers, it is colorable. For other cases, we may use probability. This can be regarded as a general model for the degree estimation used in coalescing. Aggressive and conservative coalescing are special cases of this model, for they set some strong assumptions for the upper bound or lower bound.

There are some cases where we can improve the degree estimation. Spilling creates new smaller intervals. The new intervals are so tiny that they are only live within one instruction, so only tiny intervals in one instruction may interfere. As a result, two physical registers are enough for all the intervals from spilling, if the processor uses two-address instructions. The more knowledge we have about the relationship among the neighbors, the better estimate of degree we can make. Next, we will see an experiment on the degree calculation for the tiny intervals from spilling.

We tested three methods for degree estimation - assigning all tiny intervals into one physical register, assigning all tiny intervals into two physical registers, and a precise method that checks the interference among the tiny intervals to determine how many registers they require. Experimental results indicate that the precise method is slightly better. So our implementation adopted the precise method.

### Calculating Spilling Cost

The number of the executions of a loop is estimated as 10. So the computation of spilling cost is

$$SpillingCost(LR) = \sum_{def, uses \in LR} 10^{\text{depth of the loop}}$$

The very short intervals, such as use-after-def and the intervals from spilling, are assigned a very large spill cost, so that they will not be spilled.

## Coalescing

In the implementation, each coalesce needs to erase one of the coalesced intervals, transfer its information to the other, and update the adjacency relation in interference graph. After coalescing, the degree of related intervals will change, creating new chances for coalescing. An iterative method is used for coalescing.

## Simplifying

To simplify a graph, the allocator takes nodes off the graph and pushes them onto a stack to construct a coloring order for later use. This procedure can be divided into two stages. The first is to push all unconstrained nodes whose spill cost is not very large, and the second is to push all other nodes in the incremental order of costs. As described above, a large spill cost means the node cannot be spilled. Here we see “unconstrained” again. The traditional algorithm compares the degree in interference graph and the number of available physical registers to determine if a node is unconstrained. As we see in coalescing, we do not know accurately how many physical registers will be needed for neighbors finally, so this determination is also based on heuristics, which creates the chance to introduce some tunable parameters.

## Finding Colors

The approach of coloring the nodes is used to find an available register for each live range. The procedure is to pop the stack and look for an available color for the popped node. If a node cannot be colored, it will be spilled.

During the coloring procedure, using different policies for ordinary intervals and the tiny intervals from spilling is reasonable, because the properties of the two types of intervals are very different. My implementation assigns the first available color for an ordinary interval and the last one in the available color vector for a tiny interval. It means to push the two kinds of intervals to the two ends of the physical register vector, such that the possibility that they interfere is reduced. The method makes an

apparent improvement, compared with assigning color in the same manner.

### **Assigning Virtual Registers to Physical Registers**

Once all intervals receive colors, the register allocation can start to assign physical registers and add spill instructions. In current implementation, load or store instructions are inserted to every tiny interval from spilling. This is a little inefficient, because some load instructions may be unnecessary.[4]

### **Inserting Spilling Code**

The current implementation spills the intervals that do not get a color. Such intervals are broken into small intervals, which are marked “spilled”, and assigned a very large spill cost. Load/store instructions are not inserted at this time, in order to avoid updating the instruction index in live ranges frequently. After the intervals are broken into small ones, the register allocator enters next coloring iteration, where the ordinary intervals and the tiny intervals from the spilled intervals will be put together and be colored again, until no new spilling happens. With the iterative method, the allocator need not reserve registers for spilling.

## **3.2 Dominative Choices on Parameters**

By studying the details of the implementation, we found some points where different methods or different arguments are possible, which make the candidates for parameterization. At first, we need to identify some parameters where one choice wins for most case. For these parameters, using the best choice can result in better performance and smaller parameter space. As described above, the following parameters have dominative choice:

### **Degree estimation for a set of tiny intervals from spilling**

The level of interferences among such intervals is low and calculable, so the

allocator can calculate the number of physical register they really need, which is 1 or 2 for two-address processors, much less than the total number of the tiny intervals.

### **Register assignment policy for tiny intervals from spilling**

Try to keep the tiny intervals in a small set of physical registers. On average, it can create more continuous spaces in physical registers such that more ordinary intervals can be accommodated.

## **3.3 Tunable Parameters for Performance**

In a graph-coloring register allocator, the criterion for unconstrained/constrained and the estimation of spill costs is critical for the performance of compiled programs. However, it is very hard to calculate them when they are needed. Only after the whole allocation is finished, does the allocator know if an interval is unconstrained, i.e., colorable. For spill costs, it is dynamic in execution, so computing a precise value at compilation time is impossible. In order to solve these problems, people use some heuristics, as mentioned in previous chapters. My new method turns these heuristics into tunable parameters, creating a chance for better performance.

We noticed that the number of available physical registers plays an important role in each of the heuristics. This value is used to make some important estimates and decisions in coalescing and simplification.

In coalescing, the number is important for evaluating the profitability of a specific coalesce decision. One difficulty is that the number of available physical registers is not certain for the processor we are using, because of the complicated register layout. The other is that we do not know how many physical registers will be needed for a group of virtual registers, and this value will be compared with the number of available physical registers. If we define the number of physical registers as a parameter, we can lessen the trouble from the first problem. As for the second, it is about the

comparison between two values, so making either side tunable is fine.

In coloring, the number is used to divide the two stages of simplification. The boundary between the stages is determined by the number of available physical registers, because it defines what is unconstrained. Like the inaccuracy in coalescing, the number of physical registers does not divide them accurately, but we cannot calculate an accurate value for it. We are not even sure whether one value is better than another. Therefore, we also let it be a tunable parameter.

Now we have two parameters for adaptive control. Considering integer and float register classes do not intersect, we divide each parameter into two, one for integer and the other for float.

We also tried different methods for spilling cost calculation and add it the parameter list. Actually, the calculation for total spilling cost is identical, but the values used to order the intervals in the stack may contain other heuristics. My implementation use three method -  $cost$ ,  $\frac{cost}{degree}$ , and  $\frac{cost}{degree^2}$  - where the cost is the estimated total spilling cost of the interval and the degree is number of interfered intervals. Previous research demonstrated no one wins for all application [4], so it is suitable as a tunable parameter.

Finally, we have five parameters for adaptive searching. The new command line arguments are

```
-coalescing-threshold-integer=< int >  -stack-threshold-integer=< int >
-coalescing-threshold-float=< int >    -stack-threshold-float=< int >
-weight-method={0|1|2}
```



## Chapter 4

# Properties of the Performance on Parameter Space

### 4.1 Experiments for Data Analysis

We compiled the benchmark programs with different arguments in the parameter space and ran them. The four arguments for thresholds vary from 0 to 20, and the weight-method parameter has 3 choices. Therefore, the size of the parameter space is  $21 \times 21 \times 21 \times 21 \times 3$ . But, the experiments in this chapter only use sub-spaces, by fixing the parameters for integer intervals when varying the parameters for floating-point intervals or fixing the parameters for floating-point intervals when varying the parameters for integer intervals. From the experiments, we collected the values of running time, the number of spilled intervals, the number of coalesced intervals, and others. The log files of the compiler contain more details, so we are able to get more values from them when we need. The data were analyzed for the following goals:

**Performance range** - At first, we want to know if the extended parameter space creates any chance of better performance than fixed parameter method. This is about the probability of better performance.

**Surface properties** - We would better know the changing tendency of performance over tunable parameters. If the surface of the performance in the multi-dimension space is smooth, the adaptive searching can find the best value quickly. The surface properties have an impact on the selection and design of the searching algorithm. This is about the feasibility of better performance.

**Intermediate variables** - We know lots of variables are correlated. In this experiment, we have an initial input and get a compiled program, then get the running

time of the compiled program finally. Besides the initial input and final output, there are some intermediate variables in the compilation procedure. If we can establish some links between the input and the intermediate variables, or the links between the intermediate and the out, we may accumulate more knowledge on how the input affects the output and more heuristics for better searching strategy. This is about the path to better performance.

**Relation among parameters** - If we can find the parameters are independent from each another, we may transform this problem into the optimizations of several sub-problems to improve searching efficiency. This is based on an observation that the allocations of integers and floats are independent, but experiments are needed to prove it. On the other hand, if we find redundant parameters, we can shrink the searching space. This is also about the path to better performance.

## 4.2 Performance Range Analysis

Table 4.1 presents the statistical values from the running time of the programs for integer and floating-point performance. The four major rows represent four programs. Each contains two major columns for data. The left column labeled with *integer* means changing each threshold parameter for integer intervals from 0 to 20, when fixing the threshold parameters for floating-point intervals to {8,8}. Similarly, the right column labeled with *float* means changing the parameters for floating-point intervals and fixing the parameters for integer interval to {8,8}. Each major column has three minor columns, representing the choices of the parameter for spilling weight calculation. In each group of experiments on a program with a spilling weight method, there are  $21 \times 21 = 441$  executions. The table shows the best, the worst, and the average values from the 441 running time data, as well as the percentage representing the difference between the best and the worst values. The numbers in bold means the best(minimum) values from the three choices on weight calculation.

Table 4.2 lists the values for the programs only on integer performance. It is similar as Table 4.1, except it does not contain the float column.

We also ran the algorithms with conservative coalescing and aggressive coalescing. In our parameter model, the threshold parameters for conservative coalescing are  $\{8,8,8,8\}$ ; and the threshold parameters for aggressive coalescing are  $\{\infty,8,\infty,8\}$ . Although the number of available integer physical registers cannot be set precisely as discussed above, we assigned the closest estimate for it in the conservative coalescing method. Table 4.3 compares the conservative and aggressive methods with the best results in the  $[0, 20] \times [0, 20]$  parameter space for the 8 benchmark programs with 3 different spill-cost calculation choices. The table also contains the worst results in the parameter space for comparing. The percentage values in the table represent the relative difference to the best values that are benchmark values(100%). There are some cases where the aggressive method is better than the best value in the parameter space. This is reasonable, considering the best values are from a sub-space. We also found neither the conservative method nor the aggressive method wins over the other one. The percentage values in bold means the aggressive coalescing method performs better than the best values in the row. Though in some rows of this table the aggressive method performs better, it is possible that varying the thresholds for simplification can make better results than fixing them to 8.

From these tables, we can get the opportunity of performance improvements using parameterization and other important properties. The following lists the results.

**The parameterization is profitable.** We can make this statement from table 4.3.

The maximum improvement of the tunable parameter method over the better one of the conservative and the aggressive methods is 16%(scimark, cost method). Among all the 8 programs, 2 programs have significant improvement and 3 programs have not so large improvement with the tunable method. If a larger parameter space is used, the parameterization will make more improvements. And the effects of the improvement vary from program to program, so

there is not a method that always wins.

**Adaptive searching is necessary for the best parameters.** For this conclusion, we need to see what parameters can give the best running times. Apparently, the parameters are very different among programs. In next section, we will see some figures showing more detailed data and clearly conclude that no one parameter can give good results for all programs. Therefore, since every program has its own best parameters, the compiler needs adaptive search to find it.

**No choice on weight calculation always wins.** The best values are in bold. We can find every data column contains numbers in bold, so no one spill heuristic wins always. This result confirms the experiments presented in [4]. Also, in some programs, the performance difference between certain choices on weight calculation is large, especially, between *cost* and  $\frac{\mathit{cost}}{\mathit{degree}}$ . Overall, the  $\frac{\mathit{cost}}{\mathit{degree}}$  method is better than the *cost* method, and is close to the  $\frac{\mathit{cost}}{\mathit{degree}^2}$  method.

Program		Integer			Float		
		<i>cost</i>	$\frac{cost}{degree}$	$\frac{cost}{degree^2}$	<i>cost</i>	$\frac{cost}{degree}$	$\frac{cost}{degree^2}$
spice	best	6.857 {16,8}	<b>6.853</b> {16,9}	<b>6.853</b> {19,12}	<b>6.851</b> {20,6}	6.906 {20,10}	6.903 {20,10}
	worst	7.985 {3,20}	7.979 {2,20}	<b>7.929</b> {4,16}	<b>7.071</b> {3,20}	7.138 {16,19}	7.121 {15,20}
	mean	7.21638	<b>7.19807</b>	7.21828	<b>6.90894</b>	6.9619	6.95663
	$\frac{worst-best}{worst}$	14.13%	14.11%	13.57%	3.11%	3.25%	3.06%
nasa	best	2.527 {13,6}	<b>2.512</b> {13,9}	2.532 {13,20}	2.532 {18,9}	<b>2.499</b> {15,10}	2.546 {16,13}
	worst	2.755 {8,16}	2.696 {2,0}	<b>2.678</b> {11,3}	2.662 {3,5}	<b>2.656</b> {0,4}	2.695 {0,7}
	mean	2.58356	<b>2.55726</b>	2.59058	2.56462	<b>2.53881</b>	2.58917
	$\frac{worst-best}{worst}$	8.28%	6.82%	5.45%	4.88%	5.91%	5.53%
pifft	best	27.431 {18,4}	<b>21.810</b> {20,6}	21.947 {20,4}	27.895 {5,2}	<b>23.553</b> {11,18}	24.271 {11,18}
	worst	36.450 {2,19}	<b>32.558</b> {6,20}	32.961 {9,20}	32.479 {19,19}	<b>29.137</b> {17,19}	29.370 {4,19}
	mean	31.2261	<b>26.7503</b>	26.8491	29.5385	<b>25.0144</b>	25.4652
	$\frac{worst-best}{worst}$	24.74%	33.01%	33.42%	14.11%	19.16%	17.36%
scimark	best	25.258 {6,6}	<b>22.710</b> {10,17}	22.826 {9,17}	29.051 {18,8}	<b>25.564</b> {15,16}	25.597 {14,12}
	worst	36.043 {6,15}	32.228 {13,20}	<b>31.941</b> {13,17}	36.833 {1,16}	<b>30.387</b> {2,18}	32.846 {1,20}
	mean	30.6128	27.1435	<b>26.3835</b>	32.2859	<b>26.5814</b>	26.5861
	$\frac{worst-best}{worst}$	29.92%	29.53%	28.54%	21.13%	15.87%	22.07%

Table 4.1 : Running time on parameter space for integers and floats

Program		Integer		
		<i>cost</i>	$\frac{cost}{degree}$	$\frac{cost}{degree^2}$
analyzer	best	<b>41.187</b> {12,12}	42.967 {17,14}	42.263 {10,3}
	worst	50.240 {13,9}	51.955 {19,0}	<b>46.457</b> {2,6}
	mean	44.7724	44.9981	<b>44.0471</b>
	$\frac{worst-best}{worst}$	18.02%	17.30%	9.03%
lambda	best	<b>6.089</b> {7,4}	<b>6.089</b> {9,20}	6.115 {6,19}
	worst	6.673 {3,17}	<b>6.591</b> {10,12}	7.061 {8,3}
	mean	<b>6.27299</b>	6.31798	6.32343
	$\frac{worst-best}{worst}$	8.75%	7.62%	13.40%
llu	best	<b>13.346</b> {14,20}	13.376 {6,11}	13.456 {18,10}
	worst	15.145 {0,13}	<b>15.087</b> {0,17}	15.240 {4,12}
	mean	14.1464	<b>14.1434</b>	14.1553
	$\frac{worst-best}{worst}$	11.88%	11.34%	11.71%
sim	best	10.859 {7,14}	<b>9.496</b> {1,16}	9.652 {3,15}
	worst	12.581 {18,9}	<b>10.634</b> {5,3}	10.978 {14,1}
	mean	11.3301	<b>9.95391</b>	10.0838
	$\frac{worst-best}{worst}$	13.69%	10.70%	12.08%

Table 4.2 : Running time on parameter space for integers only

Program		best	worst	aggressive	conservative
spice	<i>cost</i>	6.851 {8,8,20,6}	7.985(117%) {3,20,8,8}	6.828 ( <b>100%</b> )	6.883 (100%)
	$\frac{\text{cost}}{\text{degree}}$	6.853 {16,9,8,8}	7.979(116%) {2,20,8,8}	6.863 (100%)	6.881 (100%)
	$\frac{\text{cost}}{\text{degree}^2}$	6.853 {19,12,8,8}	7.929(116%) {4,16,8,8}	6.860 (100%)	6.955 (101%)
nasa	<i>cost</i>	2.527 {13,6,8,8}	2.755(109%) {8,16,8,8}	2.717 (108%)	2.563 (101%)
	$\frac{\text{cost}}{\text{degree}}$	2.499 {8,8,15,10}	2.696(108%) {2,0,8,8}	2.795 (112%)	2.559 (102%)
	$\frac{\text{cost}}{\text{degree}^2}$	2.532 {13,20,8,8}	2.695(106%) {8,8,0,7}	2.747 (108%)	2.573 (102%)
pift	<i>cost</i>	27.431 {18,4,8,8}	36.450(133%) {2,19,8,8}	26.778 ( <b>98%</b> )	29.470 (107%)
	$\frac{\text{cost}}{\text{degree}}$	21.810 {20,6,8,8}	32.558(149%) {6,20,8,8}	23.982 (110%)	24.845 (114%)
	$\frac{\text{cost}}{\text{degree}^2}$	21.947 {20,4,8,8}	32.961(150%) {9,20,8,8}	24.163 (110%)	25.655 (117%)
scimark	<i>cost</i>	25.258 {6,6,8,8}	36.833(146%) {8,8,1,16}	32.979 (131%)	29.299 (116%)
	$\frac{\text{cost}}{\text{degree}}$	22.710 {10,17,8,8}	32.228(142%) {13,20,8,8}	29.119 (128%)	25.701 (113%)
	$\frac{\text{cost}}{\text{degree}^2}$	22.826 {9,17,8,8}	32.846(144%) {8,8,1,20}	28.959 (127%)	25.939 (114%)
analyzer	<i>cost</i>	41.187 {12,12,8,8}	50.240(122%) {13,9,8,8}	34.085 ( <b>83%</b> )	42.178 (102%)
	$\frac{\text{cost}}{\text{degree}}$	42.967 {17,14,8,8}	51.955(121%) {19,0,8,8}	34.469 ( <b>80%</b> )	43.775 (102%)
	$\frac{\text{cost}}{\text{degree}^2}$	42.263 {10,3,8,8}	46.457(110%) {2,6,8,8}	34.004 ( <b>80%</b> )	43.693 (103%)
lambda	<i>cost</i>	6.089 {7,4,8,8}	6.673(110%) {3,17,8,8}	6.373 (105%)	6.232 (102%)
	$\frac{\text{cost}}{\text{degree}}$	6.089 {9,20,8,8}	6.591(108%) {10,12,8,8}	6.502 (107%)	6.310 (104%)
	$\frac{\text{cost}}{\text{degree}^2}$	6.115 {6,19,8,8}	7.061(116%) {8,3,8,8}	6.274 (103%)	6.253 (103%)
llu	<i>cost</i>	13.346 {14,20,8,8}	15.145(113%) {0,13,8,8}	13.497 (101%)	13.656 (102%)
	$\frac{\text{cost}}{\text{degree}}$	13.376 {6,11,8,8}	15.087(113%) {0,17,8,8}	13.729 (103%)	13.785 (103%)
	$\frac{\text{cost}}{\text{degree}^2}$	13.456 {18,10,8,8}	15.240(113%) {4,12,8,8}	13.031 ( <b>97%</b> )	13.758 (102%)
sim	<i>cost</i>	10.859 {7,14,8,8}	12.581(116%) {18,9,8,8}	9.614 ( <b>89%</b> )	11.064 (102%)
	$\frac{\text{cost}}{\text{degree}}$	9.496 {1,16,8,8}	10.634(112%) {5,3,8,8}	8.867 ( <b>94%</b> )	9.660 (102%)
	$\frac{\text{cost}}{\text{degree}^2}$	9.652 {3,15,8,8}	10.978(114%) {14,1,8,8}	9.159 ( <b>95%</b> )	9.884 (102%)

Table 4.3 : Comparing the best results in  $[0, 20] \times [0, 20]$  parameter space with the conservative and aggressive coalescing methods

### 4.3 Performance Tendency Analysis

Figure 4.1-4.8 graph the running time against the thresholds for each program. These executions are the same as those in Section 4.2. Every program contains three or six graphs, depending on whether the program contains integer performance only. The multiple graphs for a benchmark program correspond to the different spill cost calculation methods. For the programs on integer and floating-point, the left column means the changing on the integer thresholds, and right for changing on floating-point thresholds. From top to bottom, the weight calculation method is  $cost$ ,  $\frac{cost}{degree}$ , and  $\frac{cost}{degree^2}$  respectively. In the graphs for an individual program, the value ranges of all three axes are identical, for easy comparison. In each graph, the X-axis and Y-axis are for the varying parameters, and the Z-axis is for the running time.

From these graphs, we can easily see how the performance changes when the parameters vary. The properties of the surface are very helpful for evaluating the difficulty of space searching and designing a suitable algorithm. In these graphs, there are different properties of the surface. Also, the surface in a graph displays a composition of different properties.

From a large scale, some graphs display a property of *monotonicity*. The integer graphs of spice (Figure 4.4) and pift (Figure 4.2) are typical over the entire X-Y scope. Some other graphs show monotonicity for a part of the entire X-Y scope. For example, some areas on the integer graphs of scimark (Figure 4.1) is almost even, and the boundaries is monotonic.

From a small scale, we noticed that some surfaces have a property of *smoothness*. Perfect smoothness is rare in these graphs. Some areas in the graphs for scimark (Figure 4.1) are close to this concept. For other graphs, we can classify the local properties of the surfaces. In some graphs, there are high steps, even forming a spike with a small top. There also exist some uneven areas where the changes are small. The two types of changes may come from different reason. Some of small changes are from measurement errors. The sudden changes may come from the instability



of algorithms. In the register allocation, it is possible that a small difference on parameters or internal states leads to a big difference on the final allocation result.

Monotonicity and smoothness are good for searching. For other situations, some adjustments on searching algorithms are necessary for good performance.

A composition of different properties is common. Some graphs are roughly monotonic at a large scale, and not smooth at a small scale. Actually, for small changes among neighbors, a low-pass filter can make it smoother and reveal more large-scale tendency.

The large-scale predictability and the small-scale unpredictability in some graphs imply the different consequences of the parameterization and the different reasons for better results. This parameterization creates a large space that may expand to a “good” area. On the other hand, it also introduces disturbance, resulting in good or bad results, in an unpredictable way. These mean two categories of good points. One is reachable using a searching algorithm; the other can often be reached by chance. It is unfair to state the first is good and the second is bad. In fact, the disturbance also expands the range of the performance space, bringing a probability of better result. The disadvantage is the disturbance may hidden the predictable properties of the surface and become an obstacle for reaching the first category of good points. Adaptive searching is suitable for the first category of good points, while random probing can be used for the second category. The average time of adaptive searching is relative to the algorithm itself; and the average time of random probing relies on the statistical property of the target space. Actually, most searching algorithms does both tasks. The searching algorithm for a certain problem may need a deliberate tradeoff between seeking a tendency and catching a chance.

We also noticed these graphs show no diagonal symmetry, so the two thresholds do not have similar meaning, and should be treated separately.

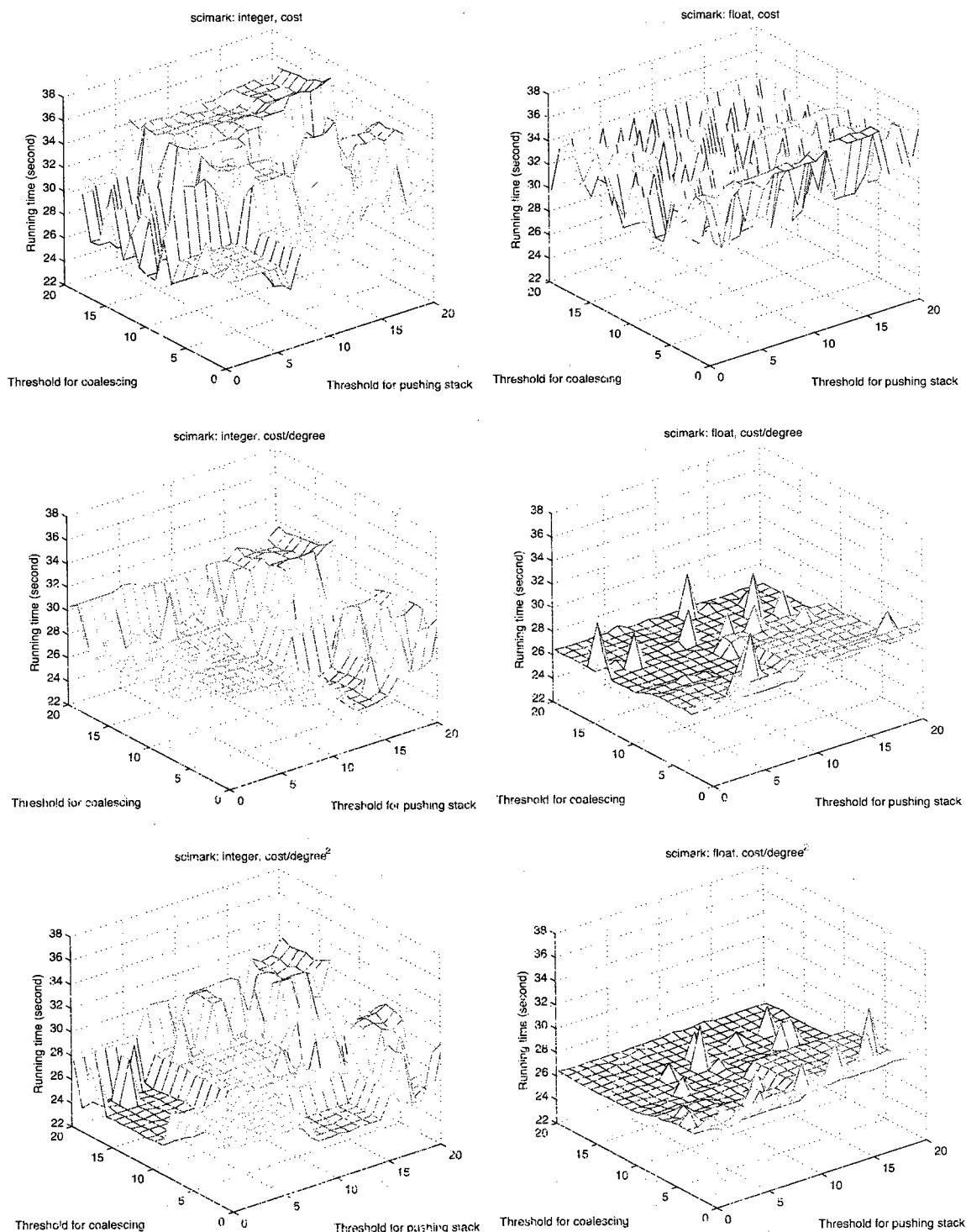


Figure 4.1 : Mesh Graphs of scimark Running Time

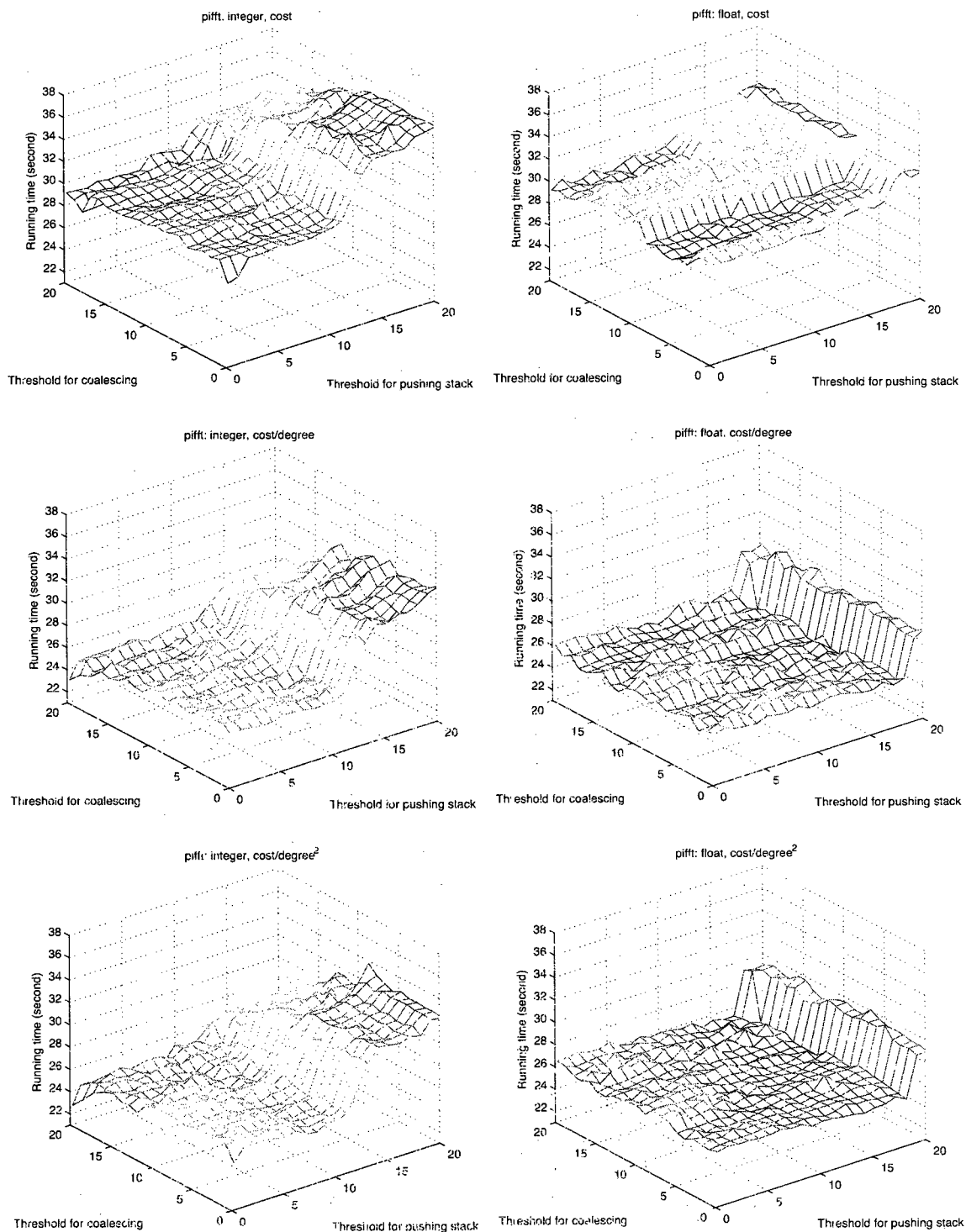


Figure 4.2 : Mesh Graphs of pifft Running Time

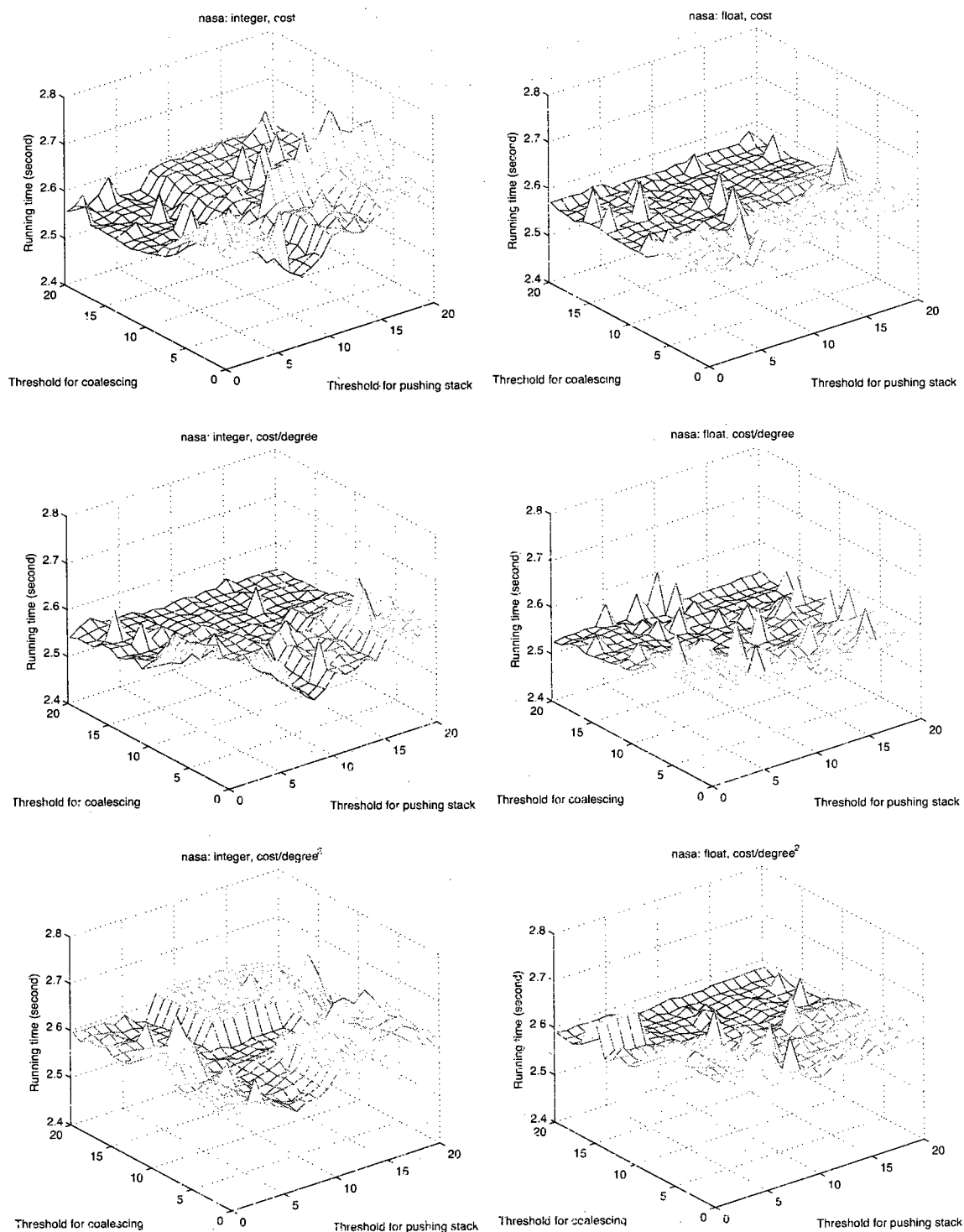


Figure 4.3 : Mesh Graphs of nasa Running Time

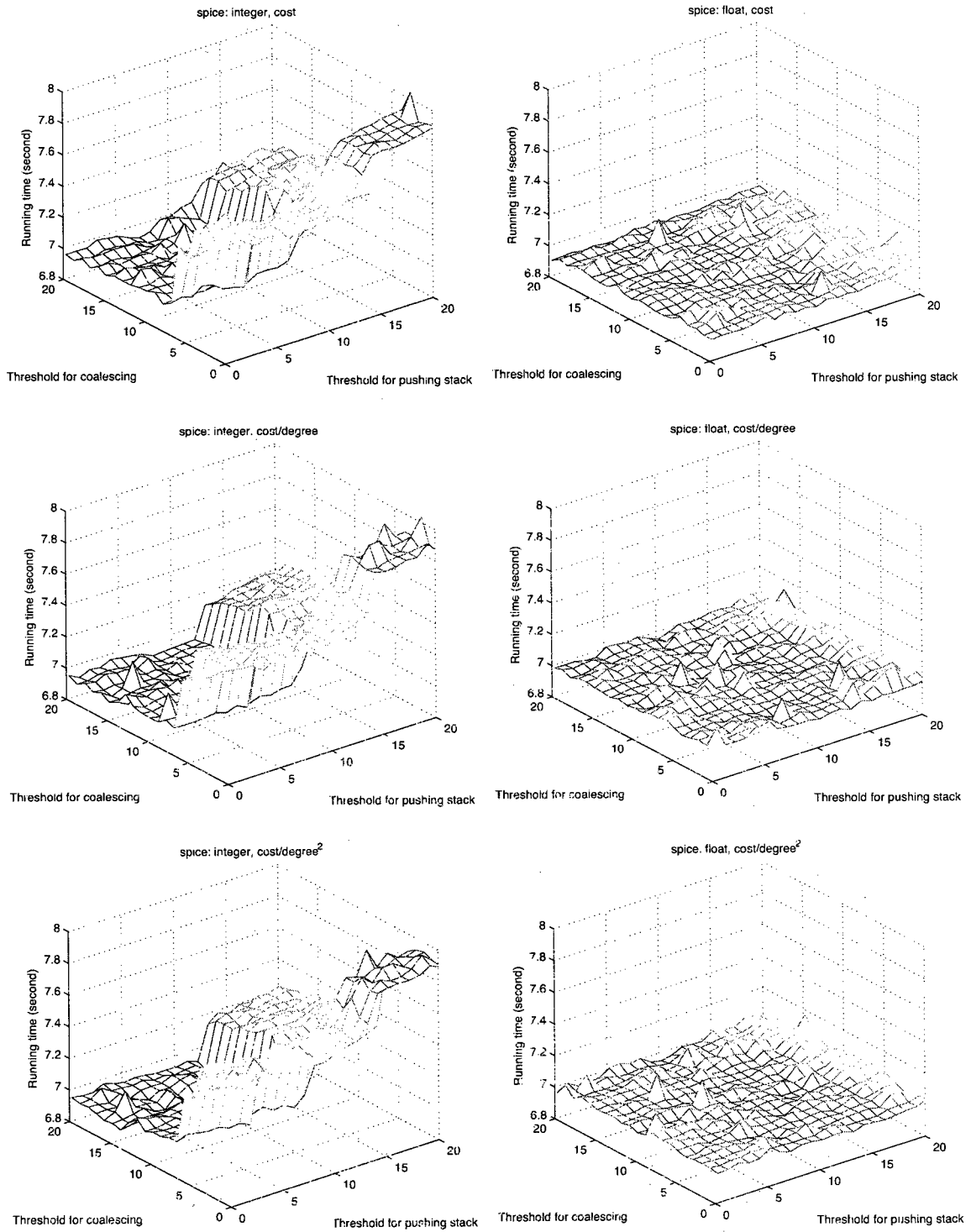


Figure 4.4 : Mesh Graphs of spice Running Time

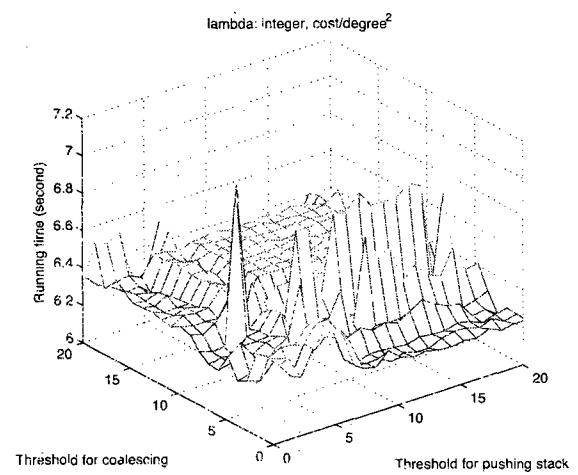
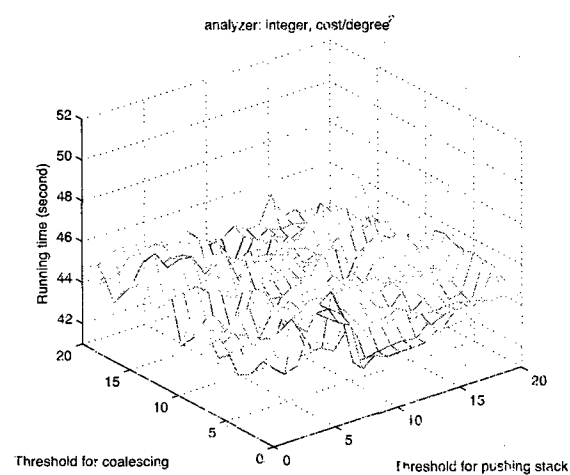
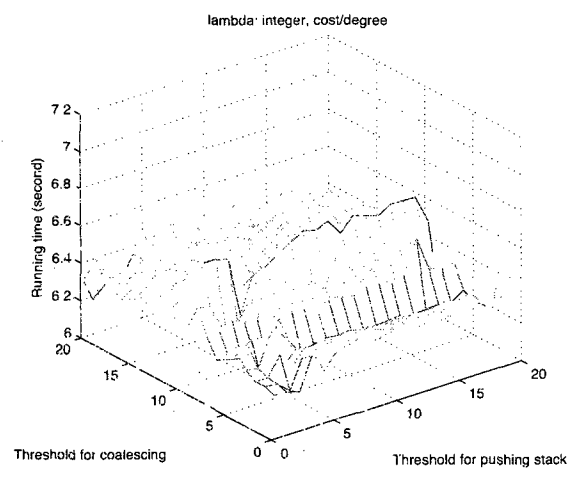
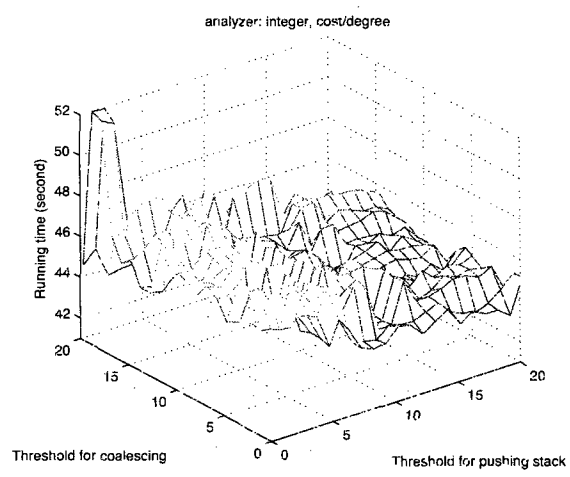
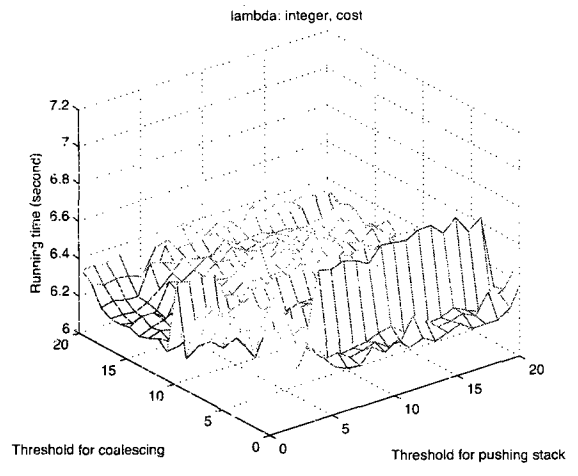
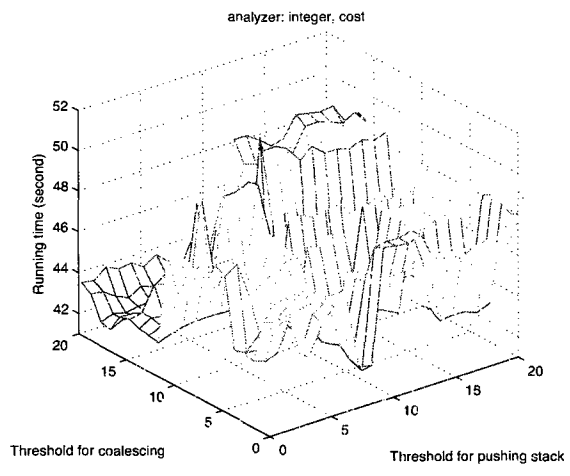


Figure 4.5 : Mesh Graphs of analyzer Running Time

Figure 4.6 : Mesh Graphs of lambda Running Time

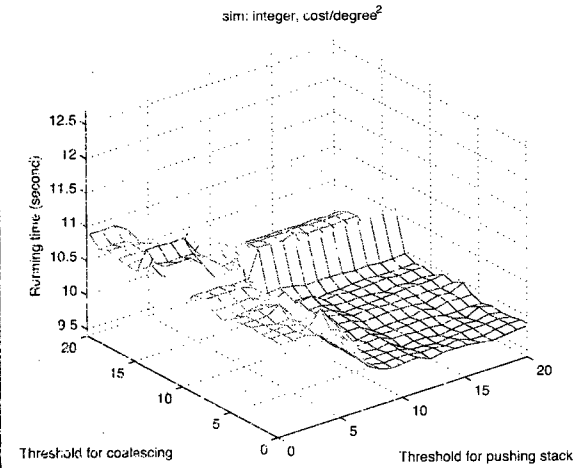
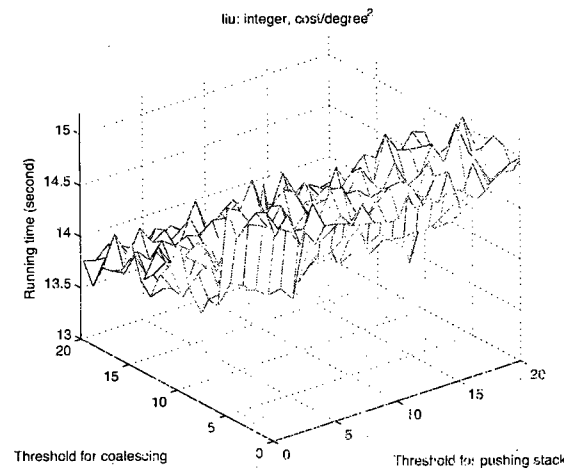
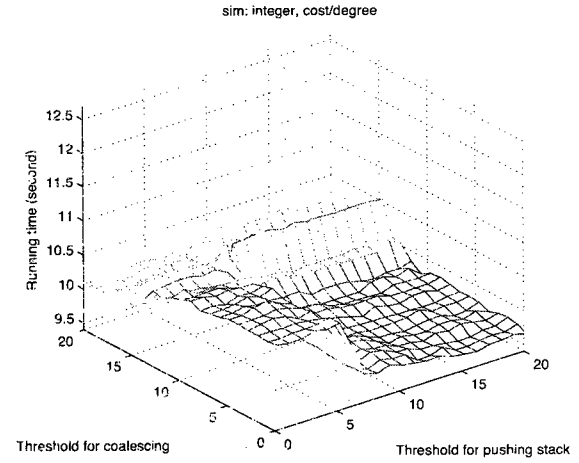
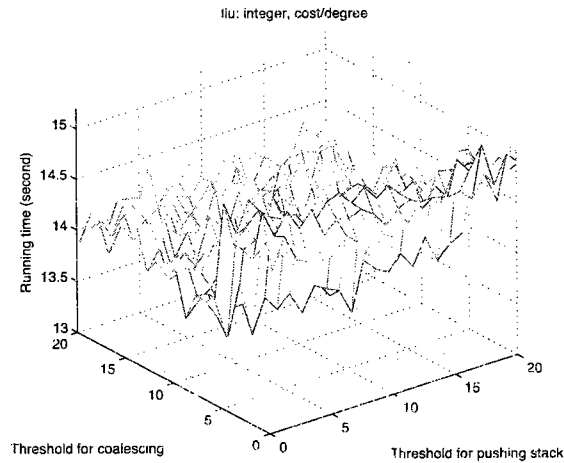
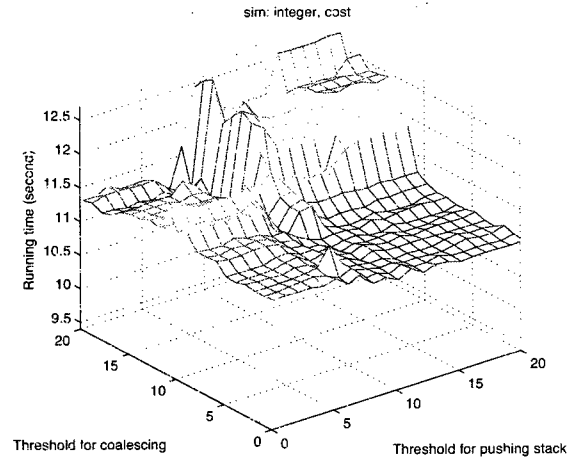
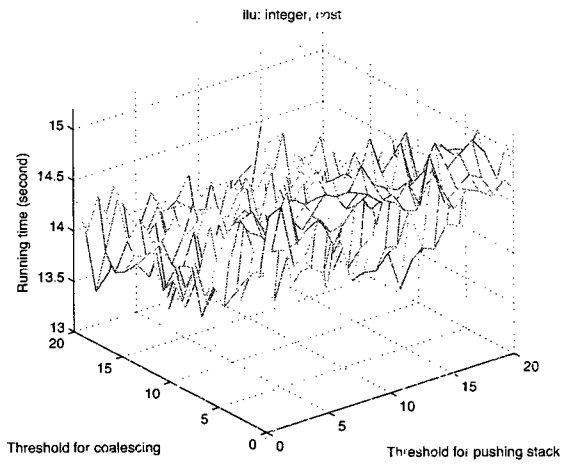


Figure 4.7 : Mesh Graphs of llu Running Time

Figure 4.8 : Mesh Graphs of sim Running Time

## 4.4 Compilation Procedure Analysis

We can construct a chain from the parameters to the final performance, and there may be some links between the two ends. The coalescing threshold parameter and pushing-stack threshold parameter have a direct effect on the number of allocation results, including coalesced intervals and spilled intervals. And the differences of the allocation results play a major role for the running time of the compiled code. If we model the relationship among them, it will be helpful for adaptive search, because we can estimate the final performance without running the code.

We collected the number of spilled intervals, the number of coalesced intervals, the number of loads/stores added and the estimated spill costs, then plotted them to study the relation between them and the running time. Figure 4.9 contains the mesh graphs for these data and running time of scimark. The graph for the number coalescing is very regular and smooth, but it looks not correlated well with the running time graph. The spill cost graph is also regular and smooth, and the area where the thresholds are small shares the same surface as the running time. The graphs for spilled intervals, stores, and loads are complicated and it is hard to correlate them to the entire running time. These last four graphs that are relevant to spilled intervals all have small values when the pushing-stack threshold parameter take values near 10.

We also made a linear regression for these data. The results show that the inputs and the targets have some correlations at large scale, but the regression results and the real results are not close enough. Especially, there are fluctuations at small scale that the regression cannot capture. Thus the statistics of the interval allocation cannot provide enough information for the performance estimation. Chapter 5 will discuss this problem with the details of the intervals. There are also some factors related to real running, such as the dynamic running counts of basic blocks and the memory/cache effect, which also make the running time estimation is hard.



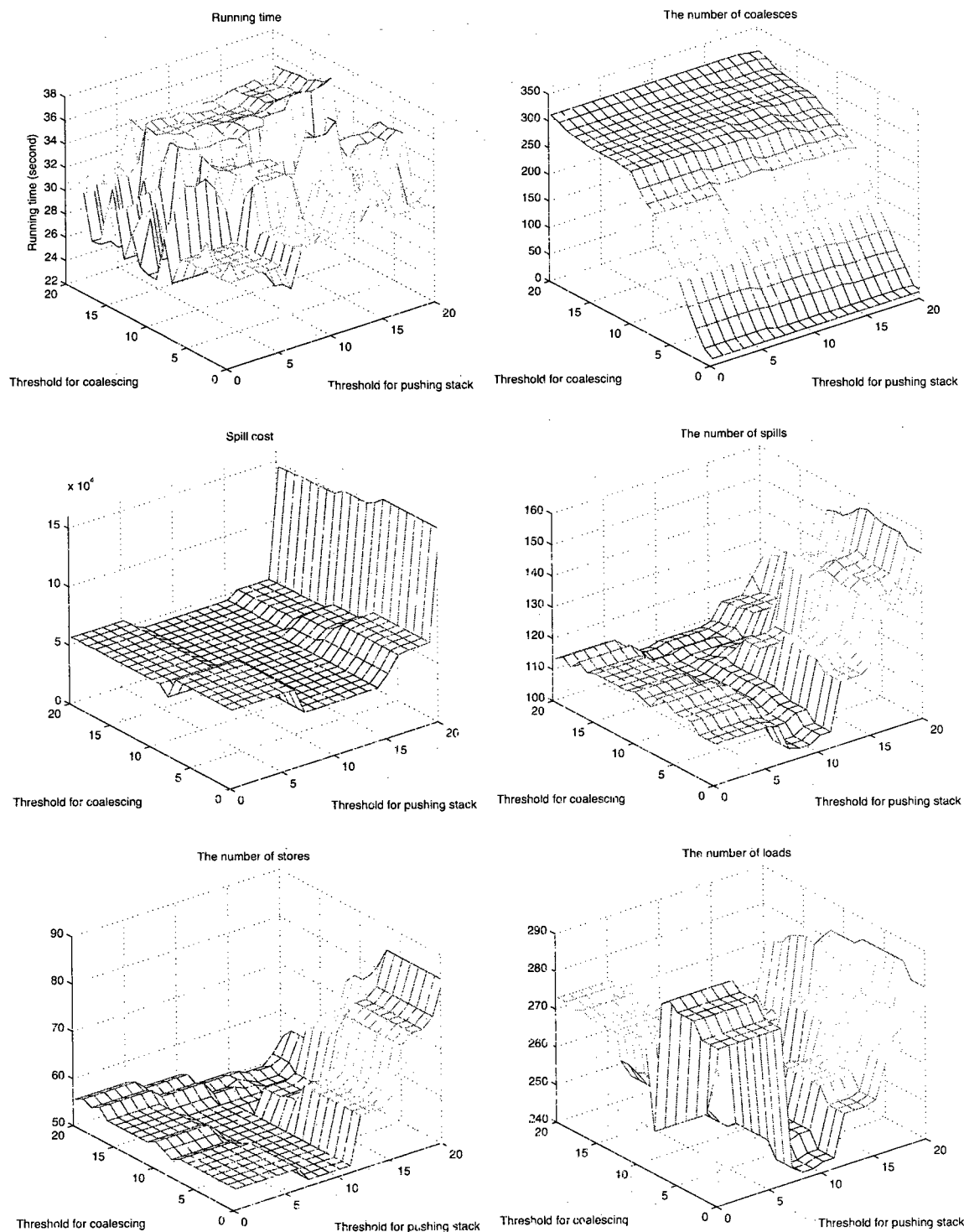


Figure 4.9 : Correlation between the interval statistics and running time for scimark

## 4.5 Independence between Parameters

We guessed that the parameters for integers and the parameters for floats may be independent, so we designed an experiment for it. When setting the floating-point threshold parameters to  $\{0,0\}$ ,  $\{4,4\}$ ,  $\{8,8\}$ ,  $\{12,12\}$ , and  $\{16,16\}$  respectively, we ran scimark program on  $\{0-20,0-20\}$  parameter space for integers. Similarly, we also fixed the integer parameters and varied the floating-point parameters. We compared the graphs to see how similar the surfaces are. In other word, we want to know if there exists a linear correlation between the surfaces. For the graphs where the integer parameters vary, although the graphs have some similarities at a large-scale, it is different at small-scale. For the graphs where the floating-point parameters vary, the differences are significant.

On the other hand, we want to know if the combination of good integer parameters and good floating-point parameters can make good overall parameters. We formulated it as this problem - if  $RunningTime(a', b', c, d)$  and  $RunningTime(a, b, c', d')$  are less than  $RunningTime(a, b, c, d)$  adequately, is  $RunningTime(a', b', c', d')$  less than the above three values? The experiments in the 4-dimentional space in the next chapter can answer this question. By looking for all cases where  $RunningTime(a', b', c, d)$  and  $RunningTime(a, b, c', d')$  are less than  $RunningTime(a, b, c, d)$  adequately and comparing  $RunningTime(a', b', c', d')$  with the other three values, we found the probability that  $RunningTime(a', b', c', d')$  is the lest is not very high.

Although the register allocation for integer intervals and floating-point intervals are separated and independent, the running time from the mixing of varying the integer parameters and varying the floating-point parameters is much complicated. So we should not divide the parameters into subspaces to get a global optimal result.

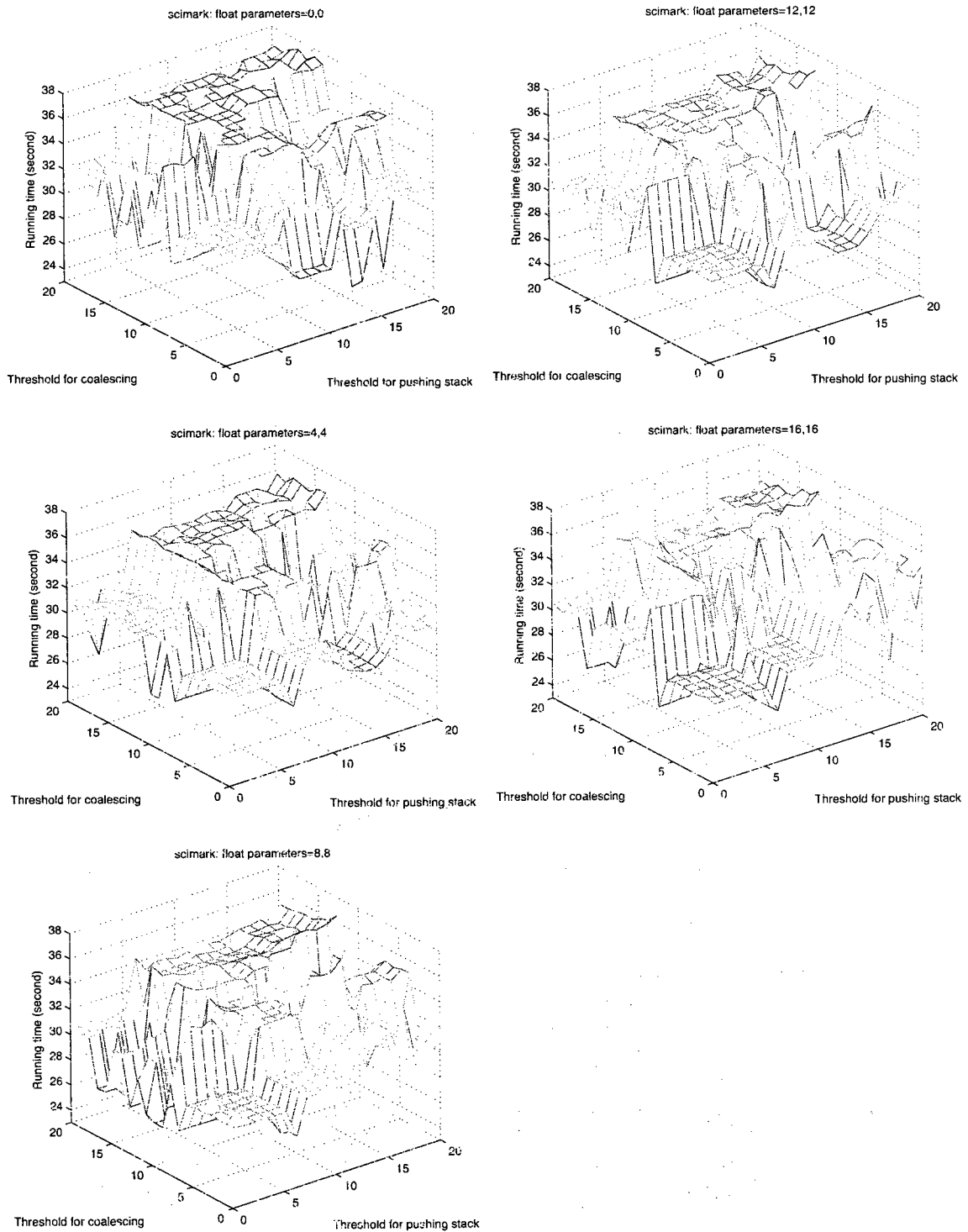


Figure 4.10 : Performance graphs for different floating-point parameters

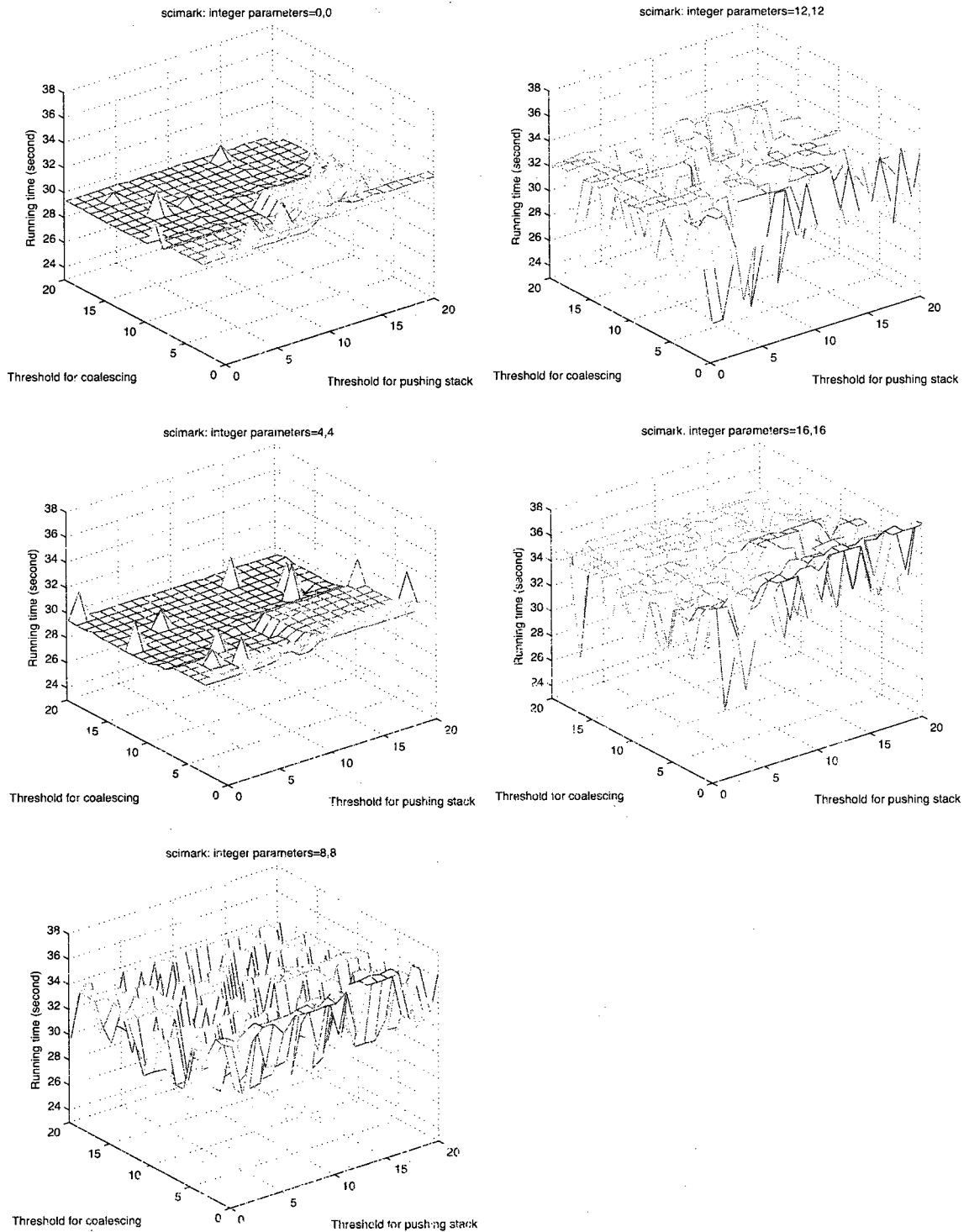


Figure 4.11 : Performance graphs for different integer parameters

## Chapter 5

### Adaptive Search with Tunable Parameters

#### 5.1 Hill Climbing Search

The *hill-climbing* search algorithm is a loop that continually moves in the direction of increasing value until reaching a peak where no neighboring point has a higher value. If the goal is to find the smallest value, just change the sense of comparison. A hill-climbing algorithm often fails to find a global maxima or minima because it may get stuck on a local maxima or minima. So *random restart* is needed to get out of the local maxima or minima and continue the searching. The efficiency of hill climbing depends on the shape of the search space: if there are few local maxima/minima and plateaux, random-restart hill climbing can find a good solution quickly. Few local maxima/minima can reduce the opportunity of restart, and few plateaux can make a rapid ascending or descending. In practical hill-climbing, the goal is often a point that is good enough, instead of the best one. So the number distribution of the good points also matters to the search time. A high percentage of the good points means a quick search. The position distribution of the good points in the search space is also relevant to the searching efficiency. In general, a sparse distribution can make the searching end successfully with less restarts. A hill-climber walks to a neighbor in each step, except restarts. Thus, the definition of neighborhood is also a key to the shape of the search space, especially in some high-dimentional spaces. A good neighborhood definition can keep the climber on long paths with a good ascending/descending rate, and reduce the number of evaluations.

The random restart in hill-climbing is like the *random probe* approach, which probes a node that is randomly selected in each step. The efficiency of random probe

```

initialization
repeat {
    pick a start point at random
    min = the value of the start point
    up_times = 0
    repeat {
        select (patience * #neighbors) unvisited neighbors randomly
        measure the values of the selected neighbors
        min_neighbor = the minimum value of the neighbors
        if (min_neighbor <= min*tolerance){
            if(min_neighbor > min){
                if(up_times < allowed_ups){
                    up_times++
                    move to the neighbor and continue the loop
                }else{
                    terminate current path, exit this loop
                }
            }else{
                up_times = 0 and min = min_neighbor
                move to the neighbor and continue the loop
            }
        }else{
            terminate current path, exit this loop
        }
    }until(terminate current path)
    if necessary, update the global minimum of all searching paths
}until(reach finish conditions)

```

Figure 5.1 : Hill Climbing Algorithm

is determined completely by the number distribution of the points, i.e. the percentage of the good points. Because the hill-climber can walk along a monotonically ascending or descending path, it may reach the goal with less steps than random probe. But if the search paths cannot get close to the goal in an efficient way, the hill-climbing may degenerate towards random probe, depending on the position distribution of the points. A extreme example is a completely random point distribution, where each climbing does not make a meaningful move towards the goal, except in the sense of random.

The hill climbing algorithm has many variants. Other than searching all neighbors exhaustively, a hill-climber may act impatiently, by checking only a part of its neighbors, resulting in decrease time consumption at each searching point. However, this may miss some paths that can reach the goal quickly, leading to more restarts. The level of impatience can be defined using the percentage of the neighbors that the climber tries.

To apply a hill-climbing algorithm to register allocation, we need to investigate the properties of register allocation over tunable parameters. A problem arises from the error of the measurement of running time, discussed in Chapter 2. Though the error is not very large, it is enough to change the results of a comparison, especially, when the imprecision is as large as the difference between two neighboring points. This may let the climber terminate the current search path even though the real values of the path are monotonic. For example, there is a monotonic path with 0.1 increment at every step, but the measurement has an error up to 0.2. Therefore, the climber may find no higher neighbors and abort current path prematurely. Another trouble is the climber may go back to the point it has visited, because the multiple executions of one program have slightly different running time, which may let the climber think it is still on an ascending or descending way.

To lessen these difficulties, the hill-climber tolerates small moving-up, though its goal is to move down. Also, it visits each point only one time. The climber records the

smallest value it has met on current searching path. If the smallest neighbor value is less than the recorded smallest value multiplied by a coefficient that is slightly greater than one, the climber will move to the neighbor. This brings another problem - the climber may traverse an even area and does not terminate. So we set a limitation on the continuous steps that a climber can move up, in order to stop aimless roving. Figure 5.1 describes the pseudo code for the algorithm.

As for the neighborhood, it is not very complicated, since the space is low-dimensional and the input values vary in a continuous manner. In our implementation, two points are neighbors when only one parameter differs and that difference is  $\pm 1$ . And we fixed the parameter for spill cost calculation method in a search. So, the number of neighbors for a point is 8, if not on the boundary.

## 5.2 Experiments and Analysis

In the experiments, we used the hill-climbing algorithm to search in the parameter space of the four threshold parameters. Because within one program, the different choices for the spill cost calculation often separate the result sets on other parameters distantly, it is not profitable to put this parameter together with other parameters in the search. Thus we just fixed this parameter in a search. Therefore, the search space is 4-dimensions, each of which is in the range of non-negative integer. The start and restart points are randomly selected from the range of 0-20; but the hill-climber may move out of this range. We tried different numbers for patience, 25%, 50% and 100%; that is to say, we evaluated 2, 4 and 8 neighbors for one move. The tolerance for up-climbing is 1.65%, derived from the measurement errors discussed in Chapter 2. The maximum allowed number of continuous steps that are higher than the minimum in current search path is 3.

During the experiments and analysis, a *restart* or a *path* refers to the entire procedure from randomly starting a point to terminating current search sequence. A *step* means making a decision on moving. Choosing an adjacent point and calculating its



value makes an *evaluation* or a *try*.

We ran two types of experiments. The first is to run the hill-climber to a fixed number of steps, so that we can study the properties of the search paths. The second is to run the hill-climber to a fixed goal, in order to get the efficiency directly.

### Experiments based on a fixed number of steps

The following data and analysis are based on program scimark with the spill cost calculation method fixed to *cost*. In these experiments, the termination condition is a path has just been terminated and the total number of steps from start has reached 100. Thus, the number of steps in a searching may be greater than 100; the benefit is we always have a complete path for each ‘climbing’. We ran the hill-climber 8 times for each patience level, due to time limitation. These experiments are aimed at the behavior in every step, rather than the final result.

Figure 5.2 illustrates the entire procedure in some searches, by which we can easily understand how it works. The horizontal axis means each step from start to end; the height of the squares on curves gives the best value found in that step. A continuous curve is a complete searching path. So, we can see how the hill-climber moves up and down, restarts and terminates, in the entire searching.

Table 5.1 lists the key data for all  $8 \times 3$  searches, including the best value, the number of restarts, the number of steps, and the number of evaluations in each entire searching. For every patience level, it gives the average length of paths, the average number of steps in a path, and the average number of evaluations in a step, based the accumulated data from 8 runs. From these data, the high patience level has a potential to find better result than others, through it needs more time. We also found the best results from 25% patience and 50% patience is close; the 25% is even better. As expected, the search with higher patience can go farther in a searching path, so it has more chances to reach better values. The number of evaluation in a step is very different, approximately proportional to the values of their patience.

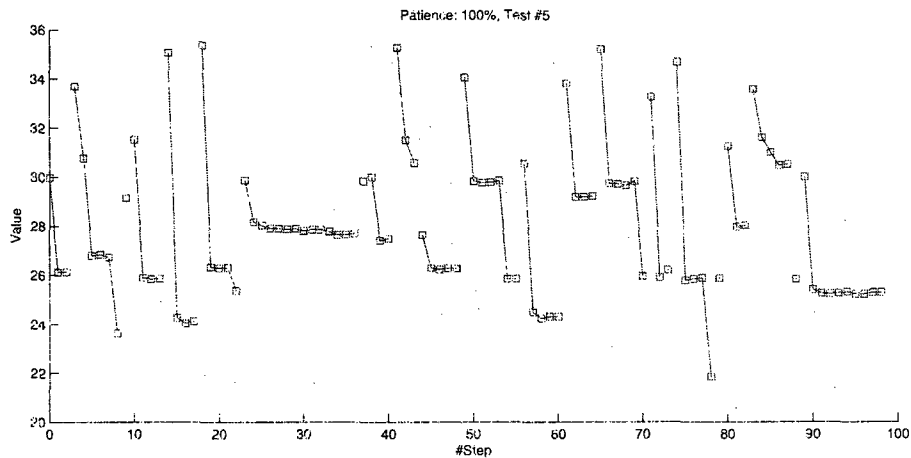
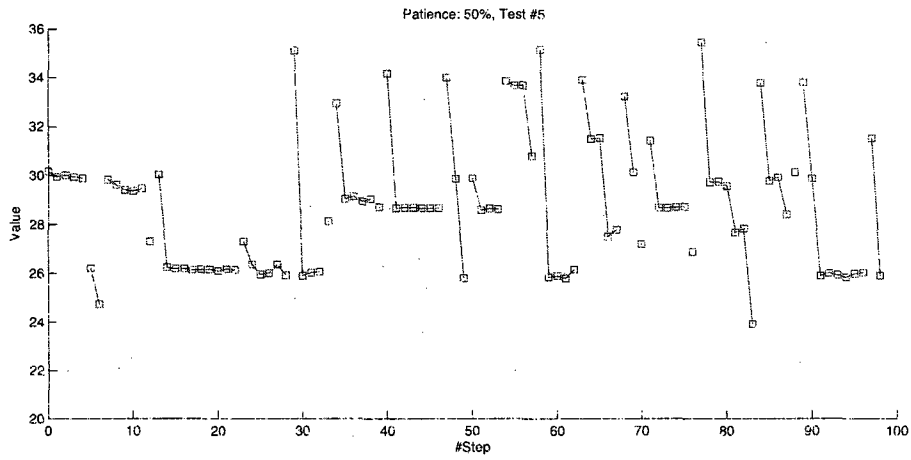
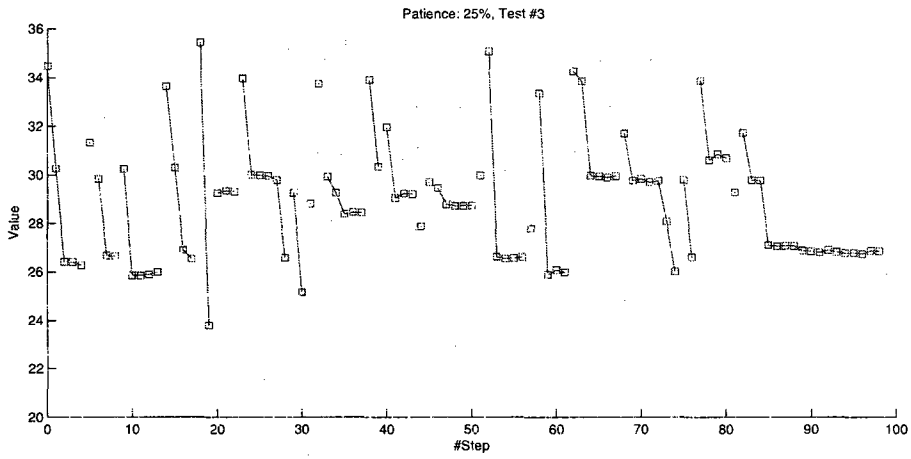


Figure 5.2 : Value on each step in a searching

patience	best	restarts	steps	evaluations	
25%	23.916	26	100	226	Average restarts
	24.174	39	105	249	32.2500
	<b>23.800</b>	28	102	232	Steps per restart
	25.111	35	104	243	3.1938
	24.284	34	107	248	Evaluations per restart
	25.195	30	104	238	7.3876
	24.241	34	101	236	Evaluations per step
	24.351	32	101	234	2.3131
50%	25.065	22	115	482	Average restarts
	25.080	32	100	432	28.5000
	23.991	29	101	433	Steps per restart
	24.661	33	105	453	3.6711
	<b>23.920</b>	26	100	426	Evaluations per restart
	24.213	37	109	473	15.6842
	25.282	22	100	422	Evaluations per step
	24.115	27	107	455	4.2723
100%	22.791	27	102	705	Average restarts
	24.526	25	107	709	24.5000
	24.020	25	102	705	Steps per restart
	23.911	24	100	694	4.2041
	<b>21.841</b>	24	109	725	Evaluations per restart
	24.611	27	101	687	28.5204
	22.529	23	103	700	Evaluations per step
	24.017	21	100	665	6.7839

Table 5.1 : Hill-Climbing searching on scimark at 25%, 50%, 100% patience

The first random point in a restart is counted into the evaluations, so the number of evaluations in a step for 25% patience may be over 2. The hill-climber does not go back to the visited points, so the value for 100% patience is less than 8. In the 8 executions at a patience level, the numbers of restarts, steps, and evaluations do not vary much, indicating statistics rules works.

Without loss of adequate accuracy, the paths can be regarded as independent on each another, considering the space is large enough. We put the paths in all 8 runs together, to get more samples for statistical analysis. Therefore, we can focus on the set of 200+ paths, instead of on the 8 runs. Each path is associated with a best value and its length, which is the base for further analysis.

Graphs in Figure 5.3 draw the length and best value of all paths from a patience level on one plane. Each pair of the values is a point on the 2-dimension plane, with the values as its coordinates. The curves describe the means of the best values at each length of paths. In these graphs, the best values often do not correspond to the long paths. Some samples indicate the longer paths tend to be on even area, which also accounts for their longer length. Longer paths consume more computing resources and often do not produce better results, so the hill-climber may stop some searches if it is long enough.

Figure 5.4 and Figure 5.5 are the histogram graphs describing the distribution of the best value and the length respectively. The overall position of best values for 100% patience is on the left of those of other patience levels. The differences between the path lengths are apparent. The searches at low patience level tend to stop earlier than others.

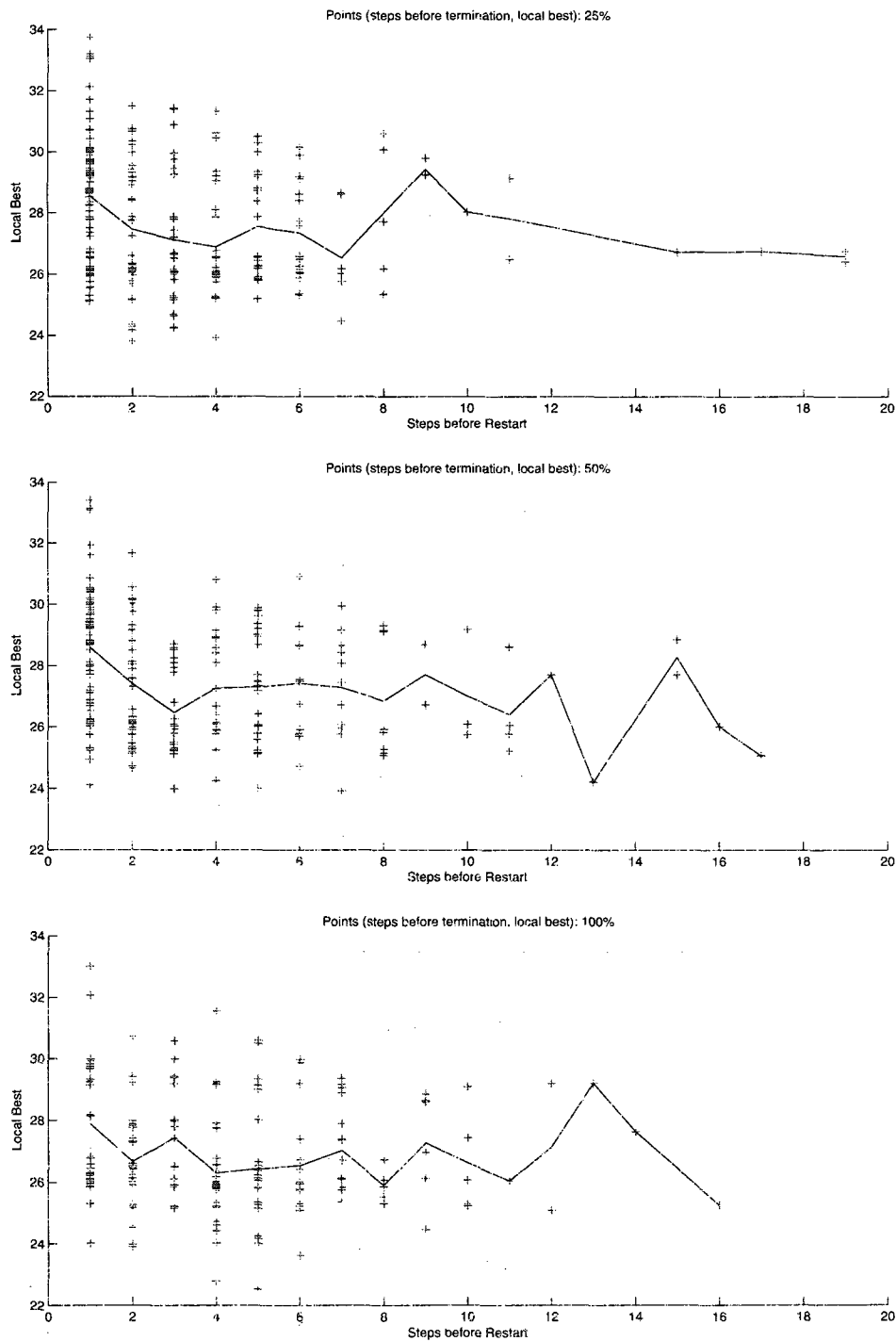


Figure 5.3 : Value pair (#steps, best value) of every restart

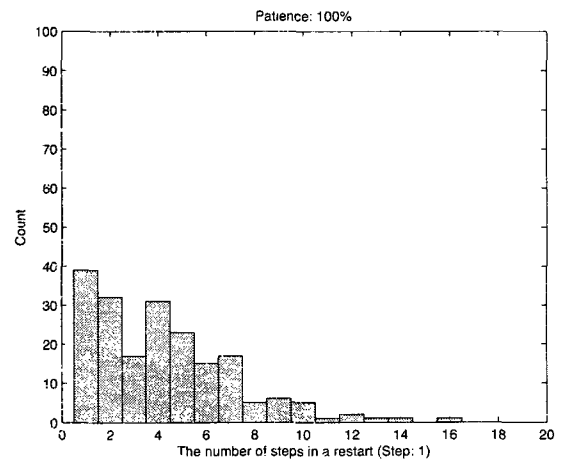
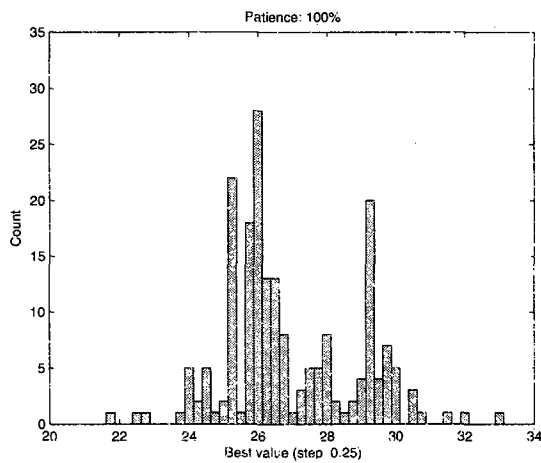
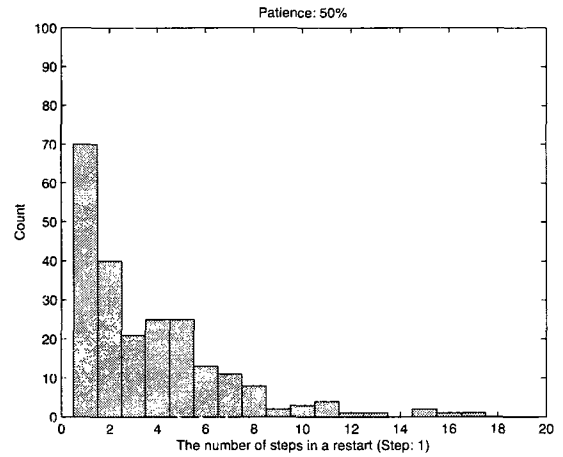
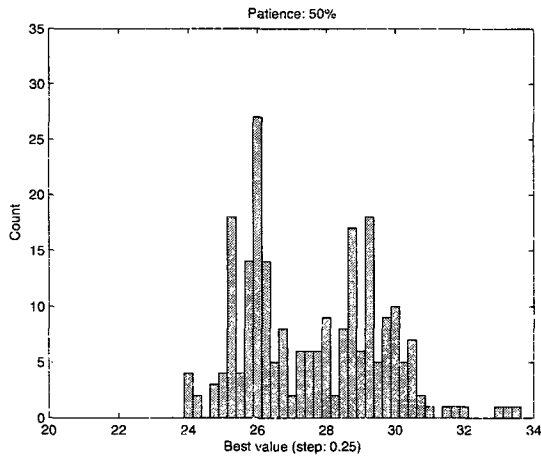
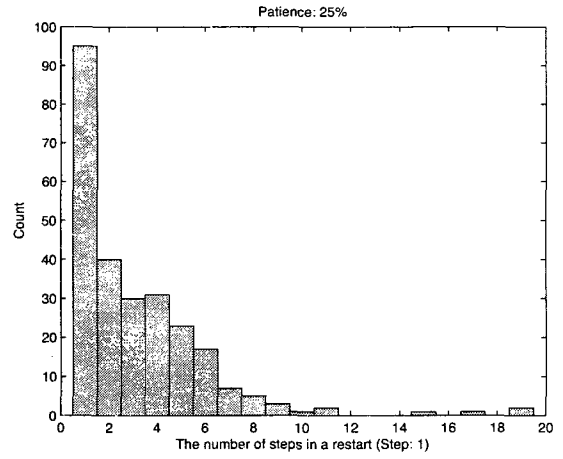
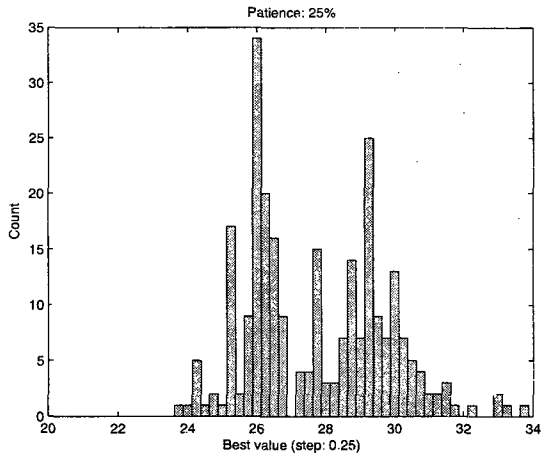


Figure 5.4 : Histogram for distribution on best values

Figure 5.5 : Histogram for distribution on steps in a restart

The next is the similar experiment on program pifft for 25% and 50% patience levels with the spill cost method argument set to  $\frac{cost}{degree}$ . The number of minimum tries is 110 in this experiment, instead of 100 in the experiment on scimark. Table 5.2 lists the data for the two patience levels. The best values in this experiment are not much better than the best value 21.810 in Chapter 4 where only the parameters for integers vary.

patience	best	restarts	steps	evaluations	
25%	<b>21.277</b>	14	113	240	Average restarts
	21.563	18	116	250	18.8750
	21.412	21	111	243	Steps per restart
	22.027	22	114	250	5.9868
	21.988	21	113	247	Evaluations per restart
	21.671	19	112	243	12.9735
	21.775	17	113	243	Evaluations per step
	21.508	19	112	243	2.1670
100%	<b>20.932</b>	21	110	740	Average restarts
	22.276	21	115	759	16.7500
	21.220	13	110	717	Steps per restart
	21.488	16	122	779	6.8358
	21.653	18	116	778	Evaluations per restart
	21.685	15	115	755	44.9104
	21.581	16	111	722	Evaluations per step
	21.783	14	117	768	6.5699

Table 5.2 : Hill-Climbing searching on pifft at 25%, 100% patience

## Experiments based on fixed goals

In order to evaluate the final performance of the hill-climber and the effect of the patience level, we ran the hill-climber with fixed goals as the termination condition. Because the traversal over the entire parameter space is not realistic, the best the running time is unknown. Thus we used the best values on the 2-dimensional  $21 \times 21$  grids from the experiments in Chapter 4, instead of the best running time. We also varied these best values by 5% and used them as the goals to evaluate the change of the search performance.

The first experiment is also based on program *scimark* and uses two goals respectively for two groups of experiments. In the first group, we set the running time goal to 25.258s, the best running time for *scimark* in Table 4.1 at the same parameter configuration. And this group contains 150 runs. In the second group, we set the goal to  $25.258 \times 95\% = 23.995s$ , and the total number of runs is 100. In each group, we used three patience values, 25%, 50% and 100%. In order to avoid too long running time, the hill-climber will terminate after 200 tries, no matter whether the goal has been reached.

The second experiment is for program *pifft*. We got the goal 21.810s from the experiments in Chapter 4. Because this goal is hard to reach, we used an easy goal for comparison,  $21.810 \times 105\% = 22.901s$ . The running time for *pifft* is too long, so we only ran 50 times. As the above *pifft* experiments, we used two patience levels, 25% and 100%.

The points that the random start stage selects can be regarded as a random set, since the probability that the random start selects a visited point is very low. We accumulated a point set for *scimark* with a size of 2052, and the set for *pifft* with a size of 971. Then we ran a random probe algorithm on these point sets to calculate the number of the tries that the random probe needs to reach a goal. The random probe simulator ran 100000 times on the point set to get the average performance. Actually, the random probe performance can be calculated, if the percentage of points



that are better than the goal is available.

Table 5.3 and 5.4 list the the average number of steps used to reach the goals and the number of runs that did not reach the goals. The computation of the average number of steps only counts the runs that reached the goals. The two major data columns represent two goals, 25.258s and 23.995s, and the data rows are for different patience levels and random probe.

Patience	25.258s		23.995s	
	average tries	unsuccessful	average tries	unsuccessful
25%	16.22	0/150	23.94	0/100
50%	20.14	0/150	37.55	0/100
100%	23.30	0/150	37.54	10/100
random	10.47	0/100000	16.83	2/100000

Table 5.3 : Hill-climbing performance for scimark

Patience	22.901s		21.810s	
	average tries	unsuccessful	average tries	unsuccessful
25%	20.36	0/50	80.08	26/50
100%	35.40	2/50	98.47	33/50
random	6.48	0/100000	71.30	15424/100000

Table 5.4 : Hill-climbing performance for pifft

From these tables, we found that the different patience levels make different performance for different goals and 25% patience is better than others. For different programs, the performance data shows different pattern.

We also found a very important fact - the random probe algorithm performed better than the hill-climbing algorithm, and it has similar pattern as the hill-climber for the programs. According to the above data and analysis on the properties of the search paths, the hill-climber does not show a gradual and apparent ascending/descending property, which is good for hill-climbing search. As showed in Figure 5.2, big drops and wandering around a value occupy most search paths. The drops play a positive role to reach a goal, but they do not often happen in the search. In these figures, there are a few paths that drop apparently in most of their steps, but the improvement they contribute is small. The wandering is very negative for this search, because it results in many futile tries, which consume lots of time but contribute little for approaching the goal. In the search space, the neighbors often have similar values, so to visit these points neither makes effective ascending/descending paths to the goal, nor expands the value range of the visited points as random method does. In this case, the search performance is even worse than random probe. If the wanderings dominate the hill-climbing search, they offset the positive effect of the searches along a monotonic direction. Thus, we can conclude that the success of the hill-climber in these experiments should be attributed to the random parts of the algorithm, such as random restart and random neighbor-selection, and the futile tries may make its performance worse than random method.

Figure 5.6 and 5.7 contain the histogram figures and the probability distribution figures for the random point sets we discussed above. The sets represent the properties of the whole search spaces approximately. We found the set for program scimark and the set for program pifft have different distribution properties. This can explain the different performance patterns of these two programs.

The performance of the random probe is solely determined by the value distribution of the running time. If the distribution has some normal properties, such as not sparse or separate, the average tries of the random probe will not be too high, depending on the position of the goal. For example, if the goal is top 10%, the average

number of tries is around 10. The random probe can work as a feasible method for not too high goals.

These figures also show the varying ranges of the running time of the compiled codes over the four parameters, as the data and figures do over two parameters in Chapter 4. They confirm again that the varying ranges are large enough to make the adaptive search profitable. Also, the values distribute over the entire varying ranges, not falling into several small ranges.

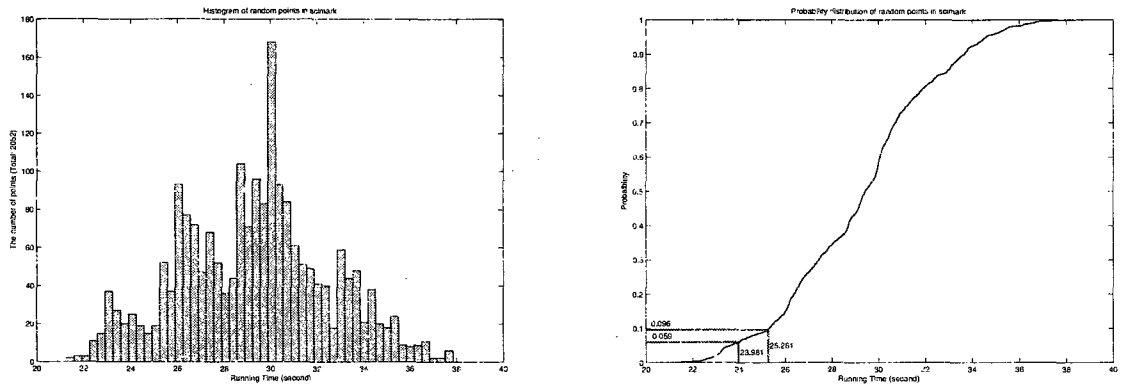


Figure 5.6 : Histogram and probability distribution of random points in scimark

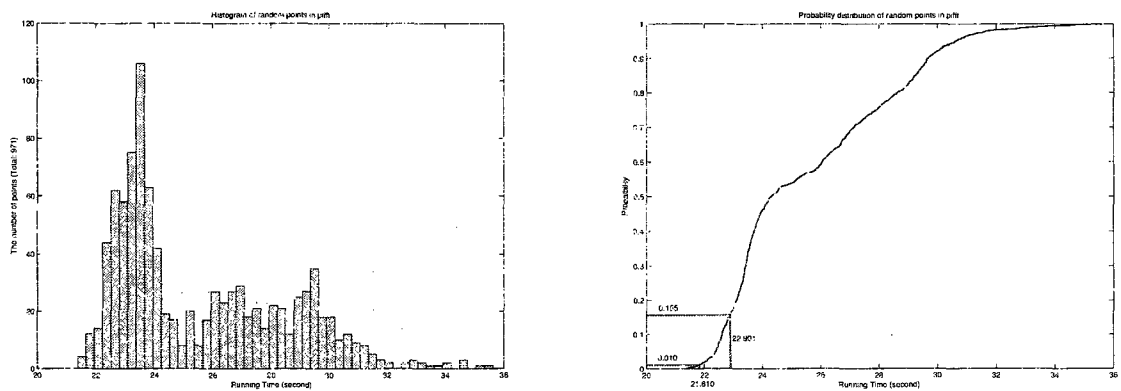


Figure 5.7 : Histogram and probability distribution of random points in pift

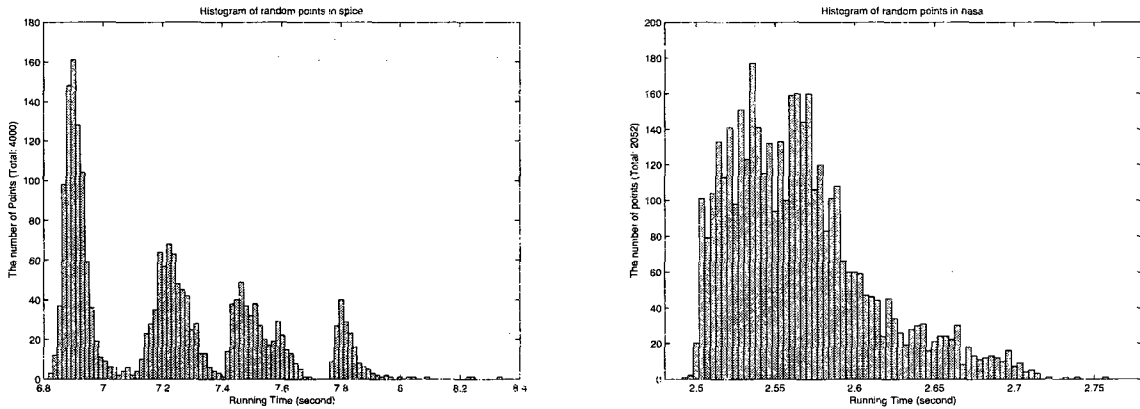


Figure 5.8 : Histograms of random points in spice and nasa

Figure 5.8 shows more histogram figures for the random point sets. We found the distributions of the random points in the search space are very different from program to program.

The performance of the random probe solely depends on the percentage of points better than the goal, but the distributions are so different that there is not a correspondence between the absolute value and the relative percentage, so it is hard to estimate the random tries to reach a goal given with an absolute value or base on a fixed value, such as the best in the search space. If we set the goal as a top percentage, the tries can be calculated in a simple way. The probability of reaching the top percentage  $t$  in  $k$  random tries is  $1 - (1 - t)^k$ . The following figure illustrates the function figure where  $t$  is 10%. The average number of tries to reach the goal is 9.84.

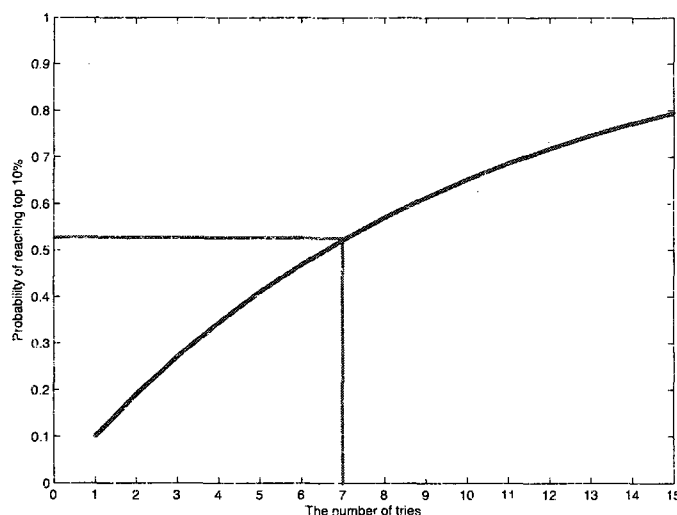


Figure 5.9 : Probability of reaching top 10%

### 5.3 Performance Stability over Parameter Changes

The previous experiments reveal some important information about the tunable parameters and adaptive search. The overall properties of the search space tell us that the tunable parameters provide a big potential for performance improvements. But the hill-climbing search fails to reach a good solution efficiently, because the microstructure of the search space is not suitable for hill-climbing search.

The running time data in the space are not good enough for an efficient hill-climbing search, since they have more random nature and plateaus than regular slopes. In other words, the relationship among the running time results of the neighboring points is not good for an adaptive search. In some cases, there is a large running time difference between two adjacent points. In some other cases, the running time varies randomly over an area. The differences of the running time come from the difference of the initial inputs, i.e. the tunable parameters. However, the parameters are very regular over the space, since they are monotonic and changed by one. If we consider

the compiler as a black box, the inputs of the box are continuous, but the outputs are not continuous, in large and small scale. The question is why the continuous inputs lead to such irregular results. In order to study this, we traced from the parameters, to the detailed list of the coalesced intervals and the allocated intervals, and finally to the running time, and investigate the changes.

The following data are from program *scimark* with the spilling cost method set to *cost*, and program *pift* with the spilling cost method set to  $\frac{cost}{degree}$ , as used in the hill-climbing search. In order to capture more details of the changes of the spilled intervals and coalesced intervals, we compared the lists of intervals to find out which intervals in one list are not in the other list, instead of only comparing the number of the list elements. Thus, we got two numbers for each comparison, the number of intervals that are deleted from the first list and the number of intervals that are added to the second list. For each program, we collected the differences of spilled intervals and spilled intervals when the parameter for coalescing threshold or pushing-stack threshold increases from 0 to 20 by 1.

Table 5.5 - 5.8 are for program *scimark*, and Table 5.9 - 5.12 are for program *pift*. The two numbers in the tables are in the form of *number/number* - the first number means the deleted intervals from the list whose parameter is less, and the second number is for the added intervals to the list whose parameter is greater. The different rows mean the varying of coalescing threshold, and the different columns are for the varying of pushing-stack threshold. In these figures, some threshold labels are not aligned with the data rows or columns, because these thresholds are varying in these tables and the data represent the differences between the thresholds. If a data line lies between two labels, this data line contains the differences between these two labels. And the table borders represent the varying direction. In order to make the tables more readable, 0 is displayed using an empty cell in some tables.

In Chapter 4, we found the number of coalesced intervals increases as the coalescing threshold increases. Here, we also find this changes have two parts - new coalesced

intervals, and the intervals that will not be coalesced. In some data, the number of the intervals that will not be coalesced is considerable. In each step, some come, and some go. Thus, these changes are greater than the changes of the total number can tell solely. The sequence of the number of the coalesced intervals as the coalescing threshold increases has peak values. The positions of such peaks are different in the two programs. After studying the distribution of the degrees of all intervals, showed in Figure 5.10, we guess they are relevant to the distribution of the degrees, because they have similar peaks.

The changes of the spilled intervals also mix the adding of new intervals and the deleting of old intervals. There is a difference from the coalesced intervals - the change of the coalesced intervals is almost monotonic, but the change of the spilled intervals is not. As the pushing-stack threshold increases from 0, the number of spilled thresholds has a tendency of decreasing at the beginning and increasing later. The turning point is near 7 or 8 roughly, which indicates the choice of the number of physical registers as the pushing-stack threshold is logically reasonable. However, the running time does not follow these changes strictly, since the running time is determined not only by the total number of spilled intervals, but also by the detailed list of spilled intervals and others. The deleting and adding of the spilled intervals augment the changes and differences, leading to more fluctuations, i.e. instability over small areas.

We have studied the effect of the coalescing threshold over the coalesced intervals and the effect of the pushing-stack threshold over the spilled intervals, which are intuitive. The tables also show the coalescing threshold has an effect not only over the coalesced intervals, but also over the spilled intervals, though the former is major. The same happens to the pushing-stack threshold. Because the graph-coloring register allocator uses an iterative procedure, the results of coalescing can change the results of pushing-stack in following iteration, and *vice versa*. The two effects also have some difference. The effect on itself is direct and major, but the effect on the other is erratic and minor. The former contributes more to the tendencies over large scale,

and the later is a source of the irregular fluctuations.

Because of the complicated nature of the graph-coloring register allocation algorithm, to change the parameters may produce irregular and unpredictable results, which can be showed through the change details of the coalesced intervals and the spilled intervals. This creates a chance to extend the range of code performance, but may become an obstacle for efficient searches. Therefore, this method of tuning parameters has two sides. If potential is more important than efficiency, chaos may create such a chance. But if efficiency is the main goal, everything should be kept neat. In order to improve the efficiency, the algorithm should try to keep the stability throughout the entire procedure of the algorithm, and make the different parts less entangled.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
0	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/
1	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/
2	/3	/3	/3	/3	/3	/3	/3	/3	/3	/3	/3	/3	/3	/3	/3	/3	/3	/3	/3	/3	/3	1/4
3	/	/	/	/	/	/	/1	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/
4	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/
5	/2	/2	/2	/2	/2	/2	1/2	/1	/1	/1	/1	/1	/1	/1	/1	/1	/1	/1	/1	/1	/1	/1
6	1/1	1/1	1/1	1/1	1/1	1/1	2/1	/1	/1	/1	/1	/1	/1	/1	/1	/1	/1	/1	/1	/1	/1	/1
7	5/1	5/1	5/1	5/1	5/1	5/1	4/1	1/	/	/	/	/	5/	5/	5/	5/	3/1	3/1	3/1	3/1	5/2	6/5
8	1/2	1/2	1/2	1/2	1/2	1/2	1/	/	/	/	/	/	1/	1/	1/	6/3	7/4	7/4	7/4	7/4	7/4	6/3
9	/1	/1	/1	/1	/1	/1	/1	/	/	/	/	/	2/	2/	2/	2/	1/1	1/1	1/1	1/1	1/1	2/
10	1/	1/	1/	1/	1/	1/	1/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/
11	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	2/1	2/1	2/1	2/1	2/1	2/1
12	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	2/1	2/1	2/1	2/1	2/1	2/1
13	/	/	/	/	/	/	/	/	/	/	/	1/	7/3	7/3	7/3	9/4	13/1	13/1	13/1	13/1	13/1	13/1
14	/	/	/	/	/	/	/	/	/	/	/	1/	3/2	3/2	3/2	3/2	10/6	11/5	11/5	12/6	12/8	12/8
15	/	/	/	/	/	/	/	/	/	/	1/	1/	1/	1/	2/1	5/3	5/3	5/3	5/3	5/3	5/3	5/3
16	2/1	2/1	2/1	2/1	2/1	2/1	2/1	1/1	1/1	1/1	1/1	3/3	3/1	3/1	3/1	3/1	3/1	3/1	3/1	3/1	3/1	4/2
17	4/6	4/6	4/6	4/6	4/6	4/6	4/6	4/5	4/5	4/5	4/5	4/5	4/5	4/5	3/3	4/6	5/5	5/5	5/5	5/5	6/6	6/6
18	4/1	4/1	4/1	4/1	4/1	4/1	4/1	3/1	3/1	3/1	5/1	3/1	3/1	3/1	2/2	3/2	3/2	3/2	3/2	3/2	3/3	7/4
19	/2	/2	/2	/2	/2	/2	/2	/1	/1	/1	/1	/1	/1	/1	1/1	/	/	/	/	/	1/	1/
20	3/1	3/1	3/1	3/1	3/1	3/1	3/1	2/1	2/1	2/1	2/1	2/1	2/1	2/1	2/1	2/1	2/1	2/1	2/1	2/1	3/2	3/1

Table 5.5 :  $\Delta$  of spilled intervals in scimark when coalescing threshold changes

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	/	/	/	/	/	1/	8/	1/	/	/2	/	2/17	/	/	3/6	3/17	1/3	/	1/2	4/16	4/16
1	/	/	/	/	/	1/	8/	1/	/	/2	/	2/17	/	/	3/6	3/17	1/3	/	1/2	4/16	4/16
2	/	/	/	/	/	1/	8/	1/	/	/2	/	2/17	/	/	3/6	3/17	1/3	/	1/2	4/16	4/16
3	/	/	/	/	/	1/	8/	1/	/	/2	/	2/17	/	/	3/6	3/17	1/3	/	1/2	4/16	4/16
4	/	/	/	/	/	/	9/	1/	/	/2	/	2/17	/	/	3/6	3/17	1/3	/	1/2	4/16	4/16
5	/	/	/	/	/	/	9/	1/	/	/2	/	2/17	/	/	3/6	3/17	1/3	/	1/2	4/16	4/16
6	/	/	/	/	/	1/	9/	1/	/	/2	/	2/17	/	/	3/6	3/17	1/3	/	1/2	4/16	4/16
7	/	/	/	/	/	2/	7/	1/	/	/2	/	2/17	/	/	3/6	3/17	1/3	/	1/2	4/16	4/16
8	/	/	/	/	/	2/1	5/	/	/	/2	/	2/12	/	/	3/6	2/19	1/3	/	1/1	5/19	5/19
9	/	/	/	/	/	3/	4/	/	/	/2	/	2/11	/	/	1/2	3/20	1/3	/	1/1	5/19	5/19
10	/	/	/	/	/	3/	5/	/	/	/2	/	2/9	/	/	1/2	3/22	1/3	/	1/1	5/17	5/17
11	/	/	/	/	/	3/	4/	/	/	/2	/	2/9	/	/	1/2	3/22	1/3	/	1/1	5/17	5/17
12	/	/	/	/	/	3/	4/	/	/	/2	/	2/9	/	/	1/2	3/21	1/3	/	1/1	5/17	5/17
13	/	/	/	/	/	3/	4/	/	/	/2	1/	3/5	/	/	4/15	1/3	/	1/1	5/17	5/17	5/17
14	/	/	/	/	/	3/	4/	/	/	/2	/	2/4	/	/	3/11	/	/	1/1	5/19	5/19	5/19
15	/	/	/	/	/	3/	4/	/	/	1/	/	2/4	/	/	1/1	2/9	/	/	1/1	5/19	5/19
16	/	/	/	/	/	3/	4/1	/	/	1/	2/2	/	/	/	1/1	2/9	/	/	1/1	5/19	5/19
17	/	/	/	/	/	3/	5/1	/	/	1/	2/2	/	/	2/1	1/3	1/6	/	/	1/1	5/19	5/19
18	/	/	/	/	/	3/	4/1	/	/	1/	2/2	/	/	1/1	1/2	1/6	/	/	1/2	5/16	5/16
19	/	/	/	/	/	3/	5/1	/	/	1/	3/2	/	/	/	1/2	1/6	/	/	1/1	5/16	5/16
20	/	/	/	/	/	3/	4/1	/	/	1/	2/2	/	/	/	1/1	1/6	/	/	1/6	5/10	5/10

Table 5.6 :  $\Delta$  of spilled intervals in scimark when simplifying threshold changes

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0/11	0/11	0/11	0/11	0/11	0/11	0/11	0/5	0/5	0/5	0/5	0/5	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/2
1	0/29	0/29	0/29	0/29	0/29	0/29	0/29	0/31	0/31	0/31	0/31	0/31	0/30	0/30	0/30	0/26	0/26	0/26	0/26	0/26	0/26
2	0/32	0/32	0/32	0/32	0/32	0/32	0/30	0/26	0/26	0/26	0/26	0/26	0/26	0/26	0/26	0/26	0/26	0/26	0/26	0/24	0/24
3	0/38	0/38	0/38	0/38	0/38	0/38	0/40	0/31	0/31	0/31	0/31	0/31	0/32	0/32	0/32	0/32	0/28	0/28	0/28	0/30	0/31
4	3/21	3/21	3/21	3/21	3/21	3/21	2/35	2/35	2/35	2/35	2/35	2/28	2/28	2/28	2/28	2/32	2/32	2/31	2/31	2/31	1/26
5	2/29	2/29	2/29	2/29	2/29	2/29	2/29	2/22	2/22	2/22	2/22	2/22	2/23	2/23	2/23	2/23	2/29	2/30	2/30	2/30	2/39
6	2/61	2/61	2/61	2/61	2/61	2/61	2/61	1/28	1/22	1/22	1/22	1/22	0/27	0/27	0/27	0/27	0/27	0/27	0/27	0/29	0/31
7	4/15	4/15	4/15	4/15	4/15	4/15	3/17	5/50	5/56	5/56	5/59	5/59	5/52	5/52	5/52	5/52	5/47	4/46	4/46	4/44	4/32
8	9/29	9/29	9/29	9/29	9/29	9/29	9/26	8/28	8/28	8/28	8/24	8/24	7/22	7/22	7/22	7/24	7/27	7/24	7/24	7/24	3/16
9	6/32	6/32	6/32	6/32	6/32	6/32	6/32	5/32	5/32	5/32	5/32	5/32	5/34	5/34	5/34	5/33	6/24	6/17	6/17	6/17	4/19
10	4/5	4/5	4/5	4/5	4/5	4/5	4/5	2/8	2/8	2/8	2/8	2/8	2/7	2/7	2/7	2/7	2/9	2/14	2/14	2/14	2/14
11	2/5	2/5	2/5	2/5	2/5	2/5	2/5	2/6	2/6	2/6	2/6	2/6	3/8	3/8	3/8	3/8	3/6	2/5	2/5	2/5	2/5
12	2/5	2/5	2/5	2/5	2/5	2/5	2/5	2/2	2/2	2/2	2/2	2/3	4/3	4/11	4/11	6/12	6/17	5/19	5/19	5/18	5/18
13	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	7/6	7/6	7/6	7/6	9/10	8/11	8/11	8/12	8/12
14	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	7/7	10/10	10/10	10/10	10/10	10/10
15	5/8	5/8	5/8	5/8	5/8	5/8	5/8	5/8	5/8	5/8	5/8	5/8	4/9	4/9	4/9	5/10	4/9	4/9	4/9	4/9	4/9
16	8/10	8/10	8/10	8/10	8/10	8/10	8/10	5/10	8/10	8/10	8/10	8/10	8/10	8/10	6/9	10/10	11/14	11/14	11/14	11/14	11/14
17	1/4	1/4	1/4	1/4	1/4	1/4	1/4	1/4	1/4	1/4	1/4	1/4	1/4	1/4	2/3	2/4	2/4	2/4	2/4	2/3	2/5
18	1/9	1/9	1/9	1/9	1/9	1/9	1/9	1/9	1/9	1/9	1/9	1/9	1/9	1/9	1/10	0/10	0/10	0/10	0/10	0/10	0/10
19	3/9	3/9	3/9	3/9	3/9	3/9	3/9	3/9	3/9	3/9	3/9	3/9	3/9	3/9	3/9	2/9	2/9	2/9	2/9	3/7	1/7
20																					

Table 5.7 :  $\Delta$  of coalesced intervals in scimark when coalescing threshold changes

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/
1	/	/	/	/	/	/	6/	/	/	/	/	1/	/	/	/	/	/	/	/	/	2/
2	/	/	/	/	/	/	4/	/	/	/	/	2/	/	/	4/	/	/	/	/	/	2/
3	/	/	/	/	/	2/	8/	/	/	/	/	2/	/	/	4/	/	/	/	2/	2/	2/
4	/	/	/	/	/	/	17/	/	/	/	/	1/	/	/	4/	4/	/	/	/	/	1/
5	/	/	/	/	/	/	2/	/	/	/	/	8/	/	/	/	4/	1/	/	/	/	6/1
6	/	/	/	/	/	/	9/	/	/	/	/	7/	/	/	/	4/6	/	/	/	/	1/5
7	/	/	/	/	/	/	41/	6/	/	/	/	2/1	/	/	1/1	2/4	/	/	/2	/6	/6
8	/	/	/	/	/	/3	10/	/	/	/3	/	8/	/	/	1/1	7/4	1/1	/	/	9/3	9/3
9	/	/	/	/	/	/	7/	/	/	1/	/	9/	/	/	1/3	8/8	3/	/	/	12/2	12/2
10	/	/	/	/	/	/	6/	/	/	1/	/	9/2	/	/	1/2	15/5	10/	/	/	6/	6/
11	/	/	/	/	/	/	1/	/	/	1/	/	10/2	/	/	1/2	14/6	5/	/	/	6/	6/
12	/	/	/	/	/	/	/	/	/	1/	/	10/3	/	/	1/2	16/6	6/1	/	/	6/	6/
13	/	/	/	/	/	/	/	/	/	/	2/	2/3	/	/	/	9/4	3/1	/	1/	6/	6/
14	/	/	/	/	/	/	/	/	/	/	/	2/	/	/	/	7/4	/	/	/	6/	6/
15	/	/	/	/	/	/	/	/	/	/	/	2/	/	/	2/2	6/3	/	/	/	6/	6/
16	/	/	/	/	/	/	/	/	/	/	/	/	/	/	1/1	5/2	/	/	/	6/	6/
17	/	/	/	/	/	/	/	/	/	/	/	/	/	/	1/2	4/1	/	/	/	6/	6/
18	/	/	/	/	/	/	/	/	/	/	/	/	/	/	1/	3/1	/	/	/	1/	4/
19	/	/	/	/	/	/	/	/	/	/	/	/	/	/	1/	/	/	/	/	6/1	6/1
20	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	3/	3/

Table 5.8 :  $\Delta$  of coalesced intervals in scimark when simplifying threshold changes

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
0	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/
1	/	/	/	/	/1	/	/	/	/	/	/	/	/	1/1	1/1	/	/	1/1	1/1	1/2	/	/
2	1/2	1/2	1/2	2/2	1/2	1/2	2/2	2/2	/	/	/	/	/	/	/	/	/	/	/	/	/	/
3	2/2	2/2	2/2	3/3	1/2	2/2	1/2	1/3	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	2/2	2/2	2/2	2/2	2/2
4	3/3	3/3	3/3	3/3	1/5	4/4	3/3	2/2	4/4	5/5	5/5	4/4	4/4	4/5	5/3	4/3	4/3	4/3	4/3	4/3	4/3	5/4
5	1/3	1/3	1/3	1/4	3/2	2/1	1/1	1/1	3/3	1/1	1/2	2/2	2/4	2/4	4/10	4/12	2/10	2/10	2/10	2/10	2/10	2/21
6	4/4	4/4	4/4	4/5	1/6	5/5	4/4	1/4	2/2	2/4	2/2	3/4	3/4	8/5	6/4	6/4	6/4	5/2	7/4	7/4	7/4	18/6
7	5/1	5/1	5/1	5/1	4/1	4/1	5/2	2/4	2/7	2/1	5/4	4/2	5/2	6/3	6/5	6/3	6/3	7/5	7/5	7/5	7/5	8/9
8	5/	5/	5/	6/1	4/1	6/1	5/2	5/	5/2	1/5	2/11	2/11	4/16	4/14	8/15	10/15	12/12	12/12	12/12	12/12	12/12	12/12
9	4/6	4/6	4/6	3/5	2/5	2/4	1/4	2/2	2/2	2/2	3/3	3/1	8/1	9/1	6/	6/	7/1	7/1	7/1	7/1	7/1	7/1
10	10/5	10/5	10/5	10/5	11/5	10/4	8/2	6/2	7/2	9/4	8/2	9/9	13/15	17/17	21/15	19/17	19/21	19/21	20/22	20/22	20/22	20/22
11	3/2	3/2	3/2	3/2	2/2	2/2	1/2	1/1	2/1	2/1	2/2	10/6	10/6	14/10	16/13	17/13	17/14	17/13	16/14	16/14	16/14	16/14
12	4/1	4/1	4/1	4/1	4/1	4/3	4/3	4/5	5/7	3/6	6/8	17/10	25/14	33/30	32/17	30/14	30/14	37/16	37/16	39/17	39/17	39/17
13	2/1	2/1	2/1	2/1	2/1	2/1	3/1	2/1	3/2	2/3	8/5	15/6	15/12	26/16	23/14	22/13	22/13	22/13	30/15	30/16	30/16	30/16
14	3/4	3/4	3/4	3/4	4/4	3/4	3/4	3/4	4/6	5/6	10/8	9/9	18/14	38/16	40/19	26/15	27/15	26/16	27/15	27/17	27/19	27/19
15	4/6	4/6	4/6	4/6	4/6	4/6	5/5	4/6	4/6	5/6	10/8	8/6	20/6	23/8	25/9	23/8	23/8	23/8	23/9	26/10	25/9	25/9
16	4/1	4/1	4/1	3/1	4/4	4/1	4/1	6/2	7/2	7/3	9/4	10/5	15/11	16/21	15/20	14/5	15/11	13/5	12/4	16/7	15/6	15/6
17	7/6	7/6	7/6	7/7	8/5	6/6	6/7	7/5	6/5	5/4	5/4	5/4	10/18	20/13	19/11	30/14	24/9	24/10	22/8	22/8	22/8	22/8
18	8/4	8/4	8/4	8/4	8/5	10/7	8/6	11/6	11/6	13/10	13/9	13/12	17/16	24/20	19/15	20/16	25/16	25/15	26/18	25/14	23/13	23/13
19	5/1	3/1	4/3	4/3	4/3	4/3	4/1	2/1	2/1	4/2	4/2	4/2	6/4	4/3	5/4	4/6	4/4	4/4	9/7	4/4	4/4	4/4
20																						

Table 5.9 :  $\Delta$  of spilled intervals in pift when coalescing threshold changes

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	/	/	5/	1/	3/4	8/1	13/4	4/7	9/3	5/16	8/20	11/39	11/56	6/21	5/16	1/6	2/10	5/13	4/7	3/2	3/2
1	/	/	5/	1/	3/4	8/1	13/4	4/7	9/3	5/16	8/20	11/39	11/56	6/21	5/16	1/6	2/10	5/13	4/7	3/2	3/2
2	/	/	5/	1/1	4/4	8/1	13/4	4/7	9/3	5/16	8/20	11/39	11/56	6/21	5/16	1/6	3/11	5/13	3/7	3/1	3/1
3	/	/	6/	1/2	4/4	7/1	13/4	5/6	9/3	5/16	8/20	11/39	11/56	6/21	5/16	1/6	3/11	5/13	3/7	3/1	3/1
4	/	/	5/	1/	3/4	8/1	13/5	5/5	9/3	5/16	8/20	11/39	11/56	6/20	5/16	1/6	3/11	5/13	3/7	3/1	3/1
5	/	/	5/	2/2	2/3	9/1	14/5	3/1	12/6	5/16	8/20	11/39	11/57	7/18	5/17	1/6	3/11	5/13	3/7	2/	2/
6	/	/	4/	5/1	2/3	7/1	14/5	6/3	9/4	4/15	8/21	11/39	11/57	4/19	7/21	2/7	3/11	5/13	3/7	2/11	2/11
7	/	/	3/	5/1	2/3	8/1	17/7	9/3	9/6	3/12	8/22	11/39	12/54	3/19	7/21	2/7	5/12	4/12	3/7	/	/
8	/	/	3/	4/1	2/3	9/2	11/6	4/1	14/5	5/14	6/19	12/39	10/52	3/21	9/21	2/7	3/11	4/12	3/7	1/4	1/4
9	/	/	4/1	2/1	2/1	8/3	10/3	6/5	5/3	7/21	6/19	11/41	10/50	6/21	7/17	4/4	3/11	4/12	3/7	1/4	1/4
10	/	/	3/	2/2	3/1	7/3	11/3	6/5	5/3	7/19	8/19	11/36	10/49	5/22	7/17	5/5	3/11	4/12	3/7	1/4	1/4
11	/	/	3/	2/1	4/2	5/1	9/3	7/5	8/6	3/14	4/21	13/40	8/45	5/16	5/19	5/9	3/11	4/12	3/7	1/4	1/4
12	/	/	3/	2/2	5/3	4/1	10/3	7/4	9/7	3/15	3/16	12/39	8/45	5/17	6/19	5/10	3/10	4/14	4/8	1/4	1/4
13	/	/	3/	2/2	3/3	4/1	8/3	8/6	4/3	4/15	2/6	11/34	13/58	6/6	7/19	5/10	1/3	4/14	5/8	1/4	1/4
14	/	/	3/	2/2	4/4	5/1	7/2	8/5	3/4	4/13	2/7	12/41	12/50	3/4	7/19	5/10	1/3	2/6	4/8	1/4	1/4
15	/	/	3/	2/1	2/3	5/3	10/4	7/6	4/4	3/7	4/4	8/33	9/29	4/6	8/30	4/8	2/6	3/5	3/9	2/7	2/7
16	/	/	3/	2/1	4/5	7/3	10/6	8/7	5/4	1/2	1/1	4/17	9/28	6/7	14/37	4/8	2/6	3/6	5/9	2/7	2/7
17	/	/	2/	2/3	4/2	7/3	11/6	9/7	5/5	1/1	1/1	9/23	10/38	4/5	5/14	6/15	/	1/4	1/4	2/7	2/7
18	/	/	3/2	2/	3/4	9/6	11/3	8/7	7/7	1/1	1/1	13/36	2/15	9/9	7/8	5/15	1/2	6/9	4/8	6/10	6/10
19	/	/	3/2	1/	1/1	8/6	14/3	8/7	5/7	3/2	3/3	13/36	10/20	3/3	2/3	3/8	/	4/9	6/7	2/7	2/7
20	/	1/2	3/2	1/	1/1	8/4	12/3	8/7	4/5	3/2	3/3	13/36	10/21	2/2	1/5	3/6	/	1/4	1/4	2/7	2/7

Table 5.10 :  $\Delta$  of spilled intervals in pift when simplifying threshold changes

	0	1	2	3	4	5	6	7	8	9	10
0	0/24	0/24	0/24	0/24	0/24	0/24	0/23	0/22	0/22	0/21	0/20
1	0/98	0/98	0/98	0/96	0/105	0/91	0/88	0/75	0/75	0/75	0/60
2	4/103	4/103	4/103	4/100	4/102	4/101	4/105	4/91	4/85	4/86	2/84
3	9/128	9/128	9/128	9/131	9/114	9/127	9/121	8/137	8/135	8/134	5/136
4	5/183	5/183	5/183	6/185	6/196	6/188	5/179	5/174	5/182	5/177	6/175
5	14/122	14/122	14/122	14/119	20/114	20/123	15/129	14/115	14/113	14/117	11/118
6	20/133	20/133	20/133	20/136	20/135	20/144	20/134	19/135	19/120	19/120	20/125
7	25/137	25/137	25/137	25/138	25/139	25/133	25/136	25/147	25/157	26/133	24/127
8	17/80	17/80	17/80	17/82	17/87	17/85	17/87	16/93	16/86	14/111	14/122
9	16/70	16/70	16/70	16/70	14/67	14/75	14/79	14/75	14/89	15/82	18/89
10	14/69	14/69	14/69	14/69	14/70	14/61	14/62	14/75	14/73	16/54	18/48
11	21/52	21/52	21/52	21/52	21/51	25/43	26/46	26/46	26/47	17/62	18/53
12	15/28	15/28	15/28	15/28	15/28	16/41	16/42	16/41	16/29	21/39	17/41
13	18/22	18/22	18/22	18/22	18/22	17/20	17/20	17/21	15/32	18/33	23/42
14	19/17	19/17	19/17	19/17	19/17	18/17	18/17	18/16	18/16	16/18	17/19
15	10/11	11/12	11/12	11/12	11/12	10/13	11/13	11/13	11/12	14/11	14/14
16	14/22	15/23	15/23	15/23	15/26	15/25	15/26	15/23	15/23	13/22	13/22
17	13/18	12/17	12/17	12/17	13/15	13/14	13/14	12/17	13/19	12/18	12/18
18	17/26	17/26	17/26	17/26	17/27	16/27	16/26	16/25	16/25	17/26	17/26
19	6/20	6/20	6/20	6/20	6/19	6/19	6/24	6/25	6/21	6/20	6/20
20											

	11	12	13	14	15	16	17	18	19	20
0	0/20	0/18	0/16	0/16	0/14	0/14	0/14	0/14	0/14	0/9
1	0/58	0/56	0/56	0/71	0/73	0/73	0/69	0/69	0/69	0/74
2	2/85	2/85	2/85	2/80	4/82	4/82	4/79	4/77	4/77	4/77
3	5/136	5/133	5/131	5/125	8/126	8/126	7/118	7/113	7/113	7/108
4	6/174	6/166	6/164	5/155	6/159	6/159	7/161	7/162	7/161	7/154
5	11/123	10/121	8/110	8/112	11/113	11/113	10/118	10/119	10/120	10/124
6	20/124	19/127	17/134	16/131	15/131	17/130	16/138	16/142	16/140	16/148
7	24/127	26/139	24/136	24/144	24/139	24/133	24/133	24/135	24/135	24/133
8	13/109	14/100	14/93	16/79	13/84	12/89	13/89	13/89	13/88	13/90
9	16/91	13/91	13/85	12/81	12/81	12/83	13/80	13/76	13/77	13/77
10	20/50	21/44	22/36	18/37	20/43	21/42	21/41	21/41	21/48	21/48
11	21/64	23/51	20/60	19/49	17/45	17/45	17/46	17/55	19/50	19/49
12	15/35	16/56	31/51	14/55	14/47	13/53	13/56	15/58	16/59	16/60
13	21/44	26/44	17/50	19/51	16/52	17/53	17/53	21/49	19/44	19/44
14	17/17	19/27	17/36	17/35	17/33	18/34	18/34	18/34	20/34	20/34
15	13/13	10/22	10/26	10/27	11/31	12/26	12/26	12/26	11/27	11/27
16	13/22	13/30	13/28	11/26	10/20	10/21	10/21	9/20	10/23	10/23
17	12/18	14/23	12/28	13/29	13/33	12/27	12/28	11/28	10/26	10/27
18	17/26	18/28	18/32	18/31	19/32	17/35	17/34	19/33	17/34	16/33
19	6/20	7/20	7/15	7/16	6/16	6/16	6/16	7/19	6/16	6/16
20										

Table 5.11 :  $\Delta$  of coalesced intervals in pift when coalescing threshold changes

0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/
1	/	/	/	/	/	1/	1/	/	1/	1/	/	2/	2/	/	2/	/	/	/	/	/	5/
2	/	/	2/	/9	14/	4/	14/	/	1/	16/	2/	4/	2/	1/16	/	/	4/	/	/	/	/
3	/	/	5/	/11	15/	/	29/1	6/	/	18/2	2/1	6/2	2/	2/12	2/2	/	7/	2/	/	/	/
4	/	/	2/	6/	2/	6/	14/3	8/	1/	13/2	1/	12/5	5/1	7/11	4/2	/	15/1	7/	/	5/	/
5	/	/	1/	/5	11/1	14/	18/2	/	8/2	15/1	2/	16/1	12/6	9/5	2/3	1/1	13/	6/	1/	12/	/
6	/	/	4/	6/	1/	3/	31/2	2/	2/	11/1	/3	16/	19/4	4/2	2/1	/	8/1	5/	/	8/	/
7	/	/	1/	7/	/8	13/	28/1	17/	2/	9/3	/2	15/3	14/8	6/2	3/2	3/1	3/5	2/1	2/	/	/
8	/	/	2/	4/	/2	10/	16/	7/	27/	13/3	/2	12/10	13/6	1/5	8/2	8/	2/4	/1	2/	2/	/
9	/	/	1/1	2/3	/	10/2	10/1	15/1	2/2	8/9	11/1	18/6	19/5	14/2	1/3	3/1	1/2	/1	3/	/	/
10	/	/	/	/	1/9	4/	13/	/	10/2	2/7	15/9	14/5	28/8	16/1	1/3	1/1	3/	3/	2/	/	/
11	/	/	/	/1	4/3	3/	/	2/	29/	5/2	13/7	21/5	33/4	11/1	4/10	2/	4/	3/	2/7	/	/
12	/	/	/	/	14/1	1/	/	1/	16/11	18/5	10/12	35/4	25/8	21/1	4/8	2/	3/	3/9	6/4	1/	/
13	/	/	/	/	3/2	/	1/	13/	3/3	15/8	9/7	16/5	39/2	2/3	9/5	1/6	/	3/9	5/3	/	/
14	/	/	/	1/1	3/1	/	/	/	2/	4/1	2/4	18/2	29/7	/	10/10	1/6	/	4/2	9/4	/	/
15	/	/	/	/	1/	/	1/	/	/2	3/	3/3	13/5	17/6	2/1	11/9	1/6	/	3/1	9/2	/	/
16	2/2	/	/	/	1/2	1/	2/1	1/	3/1	/	/	2/6	13/6	2/2	10/11	2/1	/	4/2	8/3	/	/
17	/	/	/	/3	1/1	/	4/	1/	1/	/	/	1/13	12/3	1/1	6/2	2/2	/	2/	4/1	/	/
18	1/1	/	/	/	2/1	/	/	1/1	2/1	/	/	1/16	2/	2/2	6/1	1/2	4/3	5/1	1/2	/	/
19	/	/	/	/1	/	1/	1/	/	1/	/	/	/16	1/3	1/	1/1	/	/	5/1	4/3	/1	/
20	/	/	/	/	/	/4	/	4/	2/	/	/	1/16	5/2	/	/1	/	/	2/	3/	/1	/

Table 5.12 :  $\Delta$  of coalesced intervals in pifft when simplifying threshold changes

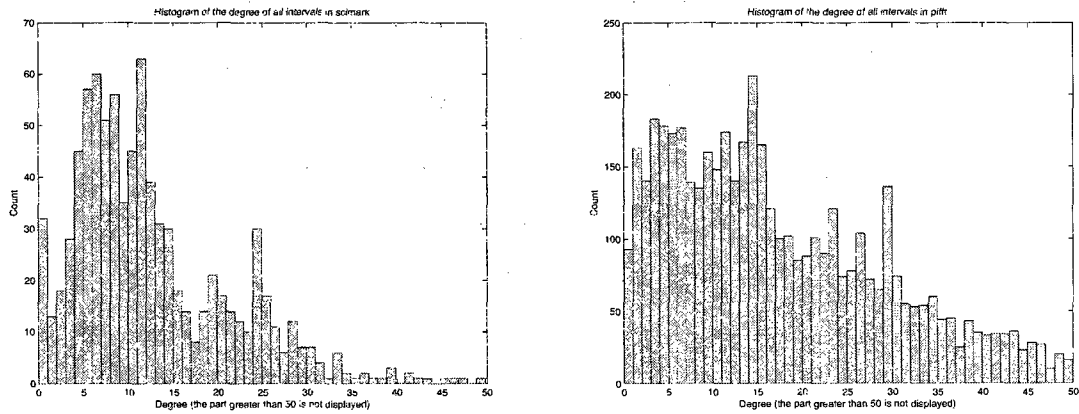


Figure 5.10 : Histogram of the degree of all initial intervals

## Chapter 6

### Summary and Conclusions

This work is about tuning the graph coloring register allocator to obtain a better code quality. It consists of two parts, adding tunable parameters to the current algorithm and looking for good parameters for individual programs.

The existing graph coloring register allocators use some heuristics based on the number of available physical registers. The heuristics are used for live interval coalescing and determining the order of live interval coloring. But no heuristics can guarantee the best results for different programs. Theoretically, making the heuristics more precise can improve the results. The problem is the precision is hard to obtain. Thus, we used the tunable parameters instead of the fixed heuristics. These tunable parameters include the threshold for coalescing, the threshold to separate the constrained and unconstrained variables in simplification. The thresholds for integers and floating-point numbers are separated. And the spill cost calculation method is also a parameter. Thus there are five tunable parameters in total, which are set as command line arguments of the LLVM compiler. The experiment results show the varying ranges of the running time are large - the difference between the best and the worst is up to 30%. So the parameterization provides a potential for better code. The best parameters also differ among the programs, so a search method for individual program is needed.

In order to find good parameters, we ran an adapted hill-climbing algorithm over these parameters at three patience levels. The results indicated the lowest patience levels(25%) works better than others(50% and 100%). These runs also generated a set of points that are selected randomly in the parameter space, which can represent the

properties of the whole space approximately. We ran a random probe simulator on the random point set. However, the results indicate the random probe outperforms the hill-climbing. By analyzing the search paths, we found the structure of the search space is not good enough for hill-climbing search, because there are more random fluctuation and less regular slopes. Therefore, the hill-climbing search often wastes time in wandering over almost-even areas.

Then we studied the reason of the poor space structure, by tracing the input parameters to the internal states of the register allocator. Instead of comparing the number of coalesced and spilled intervals, we focused on the changes of the coalesced and spilled interval lists when a parameter increases by one and found two factors can contribute to the instability of the running time. First, the changes of the coalesced or spilled intervals often contain both adding new elements and removing old elements, which means more changes in the interval lists than only adding or only deleting, leading to large and irregular changes. Second, the two thresholds for coalescing and separating constrained/unconstrained intervals not only have direct effects on the coalesced intervals and spilled intervals respectively, but also have effects on the other in an erratic way. The iteration in the algorithm creates this entanglement and makes it hard to trace. Roughly, the effects on its own part are major and regular, but the effects on the other's part are minor and irregular.

This study presents a method that creates a chance for better results; and the experiments on the adaptive search reveal some issues in practical application. These make a good feedback for further improvements. In further work, we may try to modify the algorithm to reduce the randomness and instability. This may shrink the range of the running time, or change the distribution. It is possible to deliver a good overall performance in limited search time if the adaptive search works well.

## Bibliography

- [1] Lelac Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steve Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Compilation order matters: Exploring the structure of the space of compilation sequences using randomized search algorithms, 2003.
- [2] Lelac Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems(LCTES '04)*, pages 231–239, 2004.
- [3] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of values in programs. In *Conf. Rec. Fifteenth ACM symp. on Principles of Programming Languages*, pages 1–11, 1988.
- [4] David Bernstein, Matin C. Golumbic, Yishay Mansour, Ron Y. Pinter, Dina Q. Goldin, Hugo Krawczyk, and Itai Nahshon. Spill code minimization techniques for optimizing compilers. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 258–263, 1989.
- [5] M. Biró, M. Hujterand, and Zs. Tuza. Precoloring extension. i: Interval graphs. *Discrete Mathematics*, 100(1-3):267–279, 1992.
- [6] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastell. Register allocation and spill complexity under ssa. Technical Report RR2005-33, LIP, ENS-Lyon, France, 2005.



- [7] Florent Bouchez, Alain Darté, and Fabrice Rastello. On the complexity of register coalescing. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*, pages 102–114. IEEE Computer Society Press, March 2007.
- [8] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, Houston, TX 77005, April 1992.
- [9] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software—Practice and Experience*, 28(8):859–881, 1998.
- [10] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. *ACM SIGPLAN Notices*, 24(7):275–284, July 1989.
- [11] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, May 1994.
- [12] Philip Brisk, Foad Dabiri, and Jamie Macbeth. Polynomial time graph coloring register allocation. In *15th International Workshop on Logic and Synthesis*, 2005.
- [13] Gregory J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM Symposium on Compiler Construction*, pages 98–105, June 1982.
- [14] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.
- [15] Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steve Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Exploring the structure of

- the space of compilation sequences using randomized search algorithms. *The Journal of Supercomputing*, 36(2):135–151, May 2006.
- [16] Keith D. Cooper, Tim Harvey, and Jeff Sandoval. Tuning an adaptive compiler. In *First Workshop on Statistical and Machine learning approaches applied to ARchitectures and compilaTion (SMART 07)*, 2007.
- [17] Keith D. Cooper, Timothy J. Harvey, and Linda Torczon. How to build an interference graph. *Software-Practice and Experience*, 28(4):425–444, 1998.
- [18] Keith D. Cooper, Timothy J. Harvey, and Todd Waterman. *An Adaptive Strategy for Inline Substitution*, volume 4959/2008 of *Lecture Notes in Computer Science*, pages 69–84. Springer Berlin / Heidelberg, April 2008.
- [19] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *In Workshop on Languages, Compilers, and Tools for Embedded Systems*, 1999.
- [20] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23(1):7–22, 2002.
- [21] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451 – 490, 1991.
- [22] Martin Farach and Vincenzino Liberatore. On local register allocation. In *The 9th ACM-SIAM symposium on Discrete Algorithms*, pages 564–573, 1998.
- [23] Fanica Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *Journal of combinatoric*, B(16)(46–56), 1974.

- [24] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):300–324, May 1996.
- [25] Alexander Grosul. *Adaptive Ordering of Code Transformations in an Optimizing Compiler*. PhD thesis, Rice University, April 2005.
- [26] Yi Guo, Devika Subramanian, and Keith D. Cooper. An effective local search algorithm for an adaptive compiler. In *First Workshop on Statistical and Machine learning approaches applied to ARchitectures and compilaTion (SMART 07)*, 2007.
- [27] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in ssa-form. In *15th International Conference on Compiler Construction (CC 2006)*, pages 247–262, 2006.
- [28] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, CA, April 2004.
- [29] Jonathan K. Lee, Jens Palsberg, and Fernando Magno Quintão Pereira. Aliased register allocation for straight-line programs is np-complete. *Theoretical Computer Science*, 407(1-3):258–273, 2008.
- [30] Danial Marx. Precoloring extension on unit interval graphs. *Discrete Applied Mathematics*, 154(6):995–1002, 2006.
- [31] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *The Third Asian Symposium on Programming Languages and Systems.*, pages 315 – 329, 2005.

- [32] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation by puzzle solving. In *ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI)*, 2008.
- [33] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.
- [34] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Conf. Rec. Fifteenth ACM symp. on Principles of Programming Languages*, pages 12–27, 1988.
- [35] Johan Runeson and Sven-Olof Nyström. Generalizing chaitin’s algorithm: Graph-coloring register allocation for irregular architectures. Technical Report Technical Report 2002-021, Uppsala University, Department of Information Technology, 2002.
- [36] Johan Runeson and Sven-Olof Nyström. Retargetable graph-coloring register allocation for irregular architectures. In *SCOPES*, 2003.
- [37] Michael D. Smith and Glenn Holloway. Graph-coloring register allocation for irregular architectures. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI’01)*, 2001.
- [38] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. In *Proceedings of the ACM SIGPLAN ’04 Conference on Programming Language Design and Implementation*, pages 277–288, 2004.
- [39] Mihalis Yannakakis and Fanica Gavril. The maximum k-colorable subgraph problem for chordal graphs. *Information Processing Letters*, 24(2):133–137, 1987.