Thèse

High Performance by Exploiting Information Locality through Reverse Computing

Présentée et soutenue publiquement le 21 décembre 2011 par

Mouad Bahi

pour l'obtention du Doctorat de l'université Paris-Sud

Jury

Dr.	Christine	Eisenbeis	INRIA Saclay - Île-de-France	Directrice de thèse
Prof.	Claire	Hanen	Université Paris-Ouest Nanterre La Défense	Rapporteur
Dr.	Erven	Rohou	INRIA Rennes - Bretagne Atlantique	Rapporteur
Prof.	Jean-Luc	Gaudiot	University of California - Irvine	Examinateur
Prof.	Yannis	Manoussakis	Université Paris-Sud	Examinateur
Dr.	Claude	Tadonki	Mines ParisTech	Examinateur

INRIA Saclay - Île-de-France
Université Paris-Sud

Alchemy project
Ecole doctorale LRI

Acknowledgments

I would like to thank many people who provided valuable support during the years in which I was working on this thesis. I would like to express my appreciation and thanks to my advisor Christine Eisenbeis for being a great source of inspiration, for her guidance, and for her patience. I acknowledge the PetaQCD project for providing financial support. I thank the members of my thesis committee for their reviews and suggestions concerning this research. I would also like to thank Françoise Combes from Observatoire-de-Paris for her willingness to hire me before I was quite finished and her patience while I finished the writing of my dissertation. I am also thankful to my friends, in and out of INRIA, for their friendship and support. Last but not least, I express my gratitude to my parents, my brothers and sisters for their constant support.

Abstract

The main resources for computation are time, space and energy. Reducing them is the main challenge in the field of processor performance.

In this thesis, we are interested in a fourth factor which is information. Information has an important and direct impact on these three resources. We show how it contributes to performance optimization. Landauer has suggested that independently on the hardware where computation is run information erasure generates dissipated energy. This is a fundamental result of thermodynamics in physics. Therefore, under this hypothesis, only reversible computations where no information is ever lost, are likely to be thermodynamically adiabatic and do not dissipate power. Reversibility means that data can always be retrieved from any point of the program. Information may be carried not only by the data but also by the process and input data that generate it. When a computation is reversible, information can also be retrieved from other already computed data and reverse computation. Hence reversible computing improves information locality.

This thesis develops these ideas in two directions. In the first part, we address the issue of making a computation DAG (directed acyclic graph) reversible in terms of spatial complexity. We define energetic garbage as the additional number of registers needed for the reversible computation with respect to the original computation. We propose a reversible register allocator and we show empirically that the garbage size is never more than 50% of the DAG size.

In the second part, we apply this approach to the trade-off between recomputing (direct or reverse) and storage in the context of supercomputers such as the recent vector and parallel coprocessors, graphical processing units (GPUs), IBM Cell processor, etc., where the gap between processor cycle time and memory access time is increasing. We show that recomputing in general and reverse computing in particular helps reduce register requirements and memory pressure. This approach of reverse rematerialization also contributes to the increase of instruction-level parallelism (Cell) and thread-level parallelism in multicore processors with shared register/memory file (GPU). On the latter architecture, the number of registers required by the kernel limits the number of running threads and affects performance. Reverse rematerialization generates additional instructions but their cost can be hidden by the parallelism gain. Experiments on the highly memory demanding Lattice QCD simulation code on Nvidia GPU show a performance gain up to 11%.

Abstract (in French)

Les trois principales ressources du calcul sont le temps, l'espace et l'énergie, les minimiser constitue un des défis les plus importants de la recherche de la performance des processeurs.

Dans cette thèse, nous nous intéressons à un quatrième facteur qui est l'information. L'information a un impact direct sur ces trois facteurs, et nous montrons comment elle contribue ainsi à l'optimisation des performances. Landauer a montré que cest la destruction - logique - dinformation qui coûte de l'énergie, ceci est un résultat fondamental de la thermodynamique en physique. Sous cette hypothèse, un calcul ne consommant pas d'énergie est donc un calcul qui ne détruit pas d'information. On peut toujours retrouver les valeurs d'origine et intermédiaires à tout moment du calcul, le calcul est réversible. L'information peut être portée non seulement par une donnée mais aussi par le processus et les données dentrée qui la génèrent. Quand un calcul est réversible, on peut aussi retrouver une information au moyen de données déjà calculées et du calcul inverse. Donc, le calcul réversible améliore la localité de l'information.

La thèse développe ces idées dans deux directions. Dans la première partie, partant d'un calcul, donné sous forme de DAG (graphe dirigé acyclique), nous définissons la notion de garbage comme étant la taille mémoire – le nombre de registres – supplémentaire nécessaire pour rendre ce calcul réversible. Nous proposons un allocateur réversible de registres, et nous montrons empiriquement que le garbage est au maximum la moitié du nombre de noeuds du graphe.

La deuxième partie consiste à appliquer cette approche au compromis entre le recalcul (direct ou inverse) et le stockage dans le contexte des supercalculateurs que sont les récents coprocesseurs vectoriels et parallèles, cartes graphiques (GPU, Graphics Processing Unit), processeur Cell dIBM, etc., où le fossé entre temps d'accès à la mémoire et temps de calcul ne fait que s'aggraver. Nous montons comment le recalcul en général, et le recalcul inverse en particulier, permettent de minimiser la demande en registres et par suite la pression sur la mémoire. Cette démarche conduit également à augmenter significativement le parallélisme d'instructions (Cell BE), et le parallélisme de threads sur un multicore avec mémoire et/ou banc de registres partagés (GPU), dans lequel le nombre de threads dépend de manière importante du nombre de registres utilisés par un thread. Ainsi, l'ajout d'instructions du fait du calcul inverse pour la rematérialisation de certaines variables est largement compensé par le gain en parallélisme. Nos expérimentations sur le code de Lattice QCD porté sur un GPU Nvidia montrent un gain de performances atteignant 11%.

Contents

I Introduction 17

1 Introduction 19
- 1.1 Context 20
- 1.2 Contribution 22
- 1.3 Organization of the Thesis 22

II Reversible Computing & Information Conservation 25

2 Reversible Computing: Definition and Motivation 27
- 2.1 Power Consumption and Heat Dissipation 28
- 2.2 Energy Dissipation and Reversibility 29
- 2.3 Reversible Computing 30
  - 2.3.1 Reversible Operations 31
  - 2.3.2 Reversible Logic 33
  - 2.3.3 Reversible Logic Gates 34
- 2.4 Reversible Architecture 36
- 2.5 Reversible Software 37
- 2.6 Summary 38

3 Reversible Computing for Information Conservation 39
- 3.1 Cost of Reversibility and Algorithm 40
  - 3.1.1 Reversible Operations 41
  - 3.1.2 Algorithm 43
- 3.2 Reversible DAG and Register Reuse DAG - Lower Bound 48
- 3.3 Reversibility and Values Lifetime 50
- 3.4 Experimental Results and Upper-Bound for the Garbage Size 52
- 3.5 Summary 54

III Using Reverse Computing to Improve Performance 55

4 Register Allocation Overview 57
- 4.1 Register Allocation Architecture 58
  - 4.1.1 Data Dependency 58
  - 4.1.2 Data Dependency Graph 59
  - 4.1.3 Basic Block 59
  - 4.1.4 Interference Graph 59

4.1.5 Meeting Graph 60
4.1.6 Register Requirements 60
4.1.7 Register Saturation 60
4.1.8 Register Pressure 60
4.1.9 Live Range Splitting 61
4.1.10 Coalescing 61
4.1.11 Register Spilling 61
4.1.12 Register Rematerialization 62
4.2 Different Register Allocation Approaches 62
4.2.1 Register Allocation via Graph Coloring 62
4.2.2 Linear Scan Register Allocation 63
4.2.3 Register Allocation based on Register Reuse Chains 63
4.2.4 Register Allocation via Integer Linear Programming 64
4.3 Register Allocation and Instruction Scheduling 64

5 Using Reverse Computing to Decrease Spill Code 65
5.1 Problem Statement: Register Allocation 67
5.1.1 Recomputing vs. Storage 67
5.1.2 Aggressive Register Reuse 69
5.2 Rematerialization rules and guidelines 69
5.2.1 Building Register Reuse Chains 70
5.2.1.1 Register Reuse between Dependent Values 70
5.2.1.2 Register Reuse between Independent Values 74
5.2.2 Detecting Excessive Registers 81
5.2.3 Discovering Rematerializable Values 81
5.2.3.1 Rematerialization Decision 81
5.2.4 Graph Transformation 84
5.2.5 More Opportunities for Reverse Computing than for Direct Computing 85
5.3 Experimental Results 85
5.3.1 Lattice QCD Computation 86
5.3.2 Register Requirements 87
5.3.3 Spill Costs 88
5.3.4 Run-Time Performance 89
5.3.5 Inverse Precision 90
5.4 Summary 90

6 Using Reverse Computing to Increase Intruction Level Parallelism 97
6.1 What is Instruction-Level Parallelism? 98
6.1.1 Instruction-Level Parallelism Challenges 98
6.1.1.1 Instruction-Level Parallelism within Basic Blocks 99
6.1.1.2 Instruction-Level Parallelism across Basic Blocks 100
6.2 Cell BE Implementation 101
6.2.1 Cell BE Architecture Overview 101
6.2.2 Programming Cell BE 102
6.2.2.1 Code SIMDization 102
6.2.2.2 Code Partitioning 104
6.2.2.3 Communication and Data Transfer 104
6.2.3 Performance Measurement 105

6.3 Summary 106

7 Using Reverse Computing to Increase Thread Level Parallelism 107

7.1 GPU Architecture and Programming Model 108

7.1.1 Memory Hierarchy 109

7.1.2 Thread-Level Parallelism 110

7.1.3 Register Usage, Rematerialization and Performance 110

7.2 Experimental Results 113

7.3 Analysis of Results 116

7.3.1 Limitations 117

7.4 Summary 117

IV Conclusion 119

8 Conclusion and Future Work 121

8.1 Conclusion 122

8.2 Future Work 123

List of Figures

2.1 Simple model of CMOS system . . . 28
2.2 State transition: reversible computing vs. irreversible computing . . . 30
2.3 Thermodynamic dissipation vs. technological dissipation . . . 31
2.4 Feynman gate . . . 34
2.5 Toffoli gate . . . 34
2.6 Fredkin gate . . . 34
2.7 Example of reversible logic circuit. . . . 35

3.1 Illustrative example of reversible code and garbage generation. . . . 43
3.2 Scheduling algorithm diagram. . . . 44
3.3 Example of register reuse limitation for reversing a DAG . . . 48
3.4 Example of register reuse limitation between dependent values for reversing a DAG . . . 49
3.5 Example of register reuse for reversing a DAG . . . 50
3.6 Example comparing number of live values and their lifetime between reversible and irreversible execution . . . 51
3.7 Example comparing the number of live values at each computation step for the same code in reversible and irreversible execution (corresponding to the code of Example 2 Figure 3.3) . . . 52
3.8 Upper-bound to the garbage size . . . 53
3.9 Number of graphs randomly generated according to the garbage size . . . 53
3.10 upper-bound to the garbage size . . . 54

4.1 (a) 3 address code and (b) its corresponding data dependency graph . . . 59
4.2 Example of interference graph . . . 60
4.3 (a) Pseudo code. (b) Live-ranges of variables. (c) Interference graph before splitting live-range of a. (d) Pseudo code after splitting. (e) New presentation of live-ranges. (f) New interference graph. . . . 61
4.4 Example of graph coloring . . . 63
4.5 Example of register reuse chains . . . 64

5.1 Memory Access vs CPU Speed . . . 66
5.2 GPU architecture: Memory Access vs CPU Speed . . . 66
5.3 Reducing register pressure using reverse rematerialization. . . . 68
5.4 Recomputing Vs Storage. . . . 68
5.5 Example of the relation "*can reuse*" . . . 70
5.6 Example of killing and earliest ultimate killing nodes . . . 71
5.7 Initial Data Dependency Graph . . . 73

5.8 Initial Register Reuse Chains 73
5.9 Building register reuse chains 74
5.10 Building Register Reuse Chains 75
5.11 Live ranges and distance between nodes 77
5.12 Live ranges and distance between nodes 78
5.13 Values lifetime before rematerialization 82
5.14 Values lifetime after rematerialization. 83
5.15 Data dependency graph after register rematerialization. 83
5.16 Graph transformation. 84
5.17 Local register pressure reduction 85
5.18 Contribution of reverse rematerialization to execution time 89
5.19 Hopping_Matrix_k: data dependency graph 92
5.20 Hopping_Matrix_l: data dependency graph 93
5.21 su3_multiply: data dependency graph 94
5.22 complex_times_vector: data dependency graph 95

6.1 Using reverse computing to increase instruction level parallelism 99
6.2 Using reverse computing to increase instruction level parallelism 100
6.3 Cell Broadband Engine Architecture 102
6.4 Data organization for SIMD operations (SM) 103
6.5 Different data structures used in LQCD 103
6.6 Double buffering 104
6.7 Scalability of performance with the count of the used SPEs for different SIMDization techniques 106

7.1 Fermi Streaming Multiprocessor (SM) 109
7.2 Overview of the rematerialization algorithm 110
7.3 Reverse rematerialization : performance vs. register requirements 111
7.4 Cost of reversibility: additional operations 112
7.5 Reverse rematerialization : performance vs. available registers 112
7.6 HMC code fragment: original implementation 113
7.7 HMC code fragment: code splitting and reordering - optimized implementation 114
7.8 Reverse rematerialization: global performance on NVIDIA GPU (gflops) . . . 116
7.9 Reverse Rematerialization: single thread performance on NVIDIA GPU (gflops) 116

List of Tables

2.1 Reversible simulation of irreversible computing 32
2.2 Irreversible vs. Reversible function 35

3.1 Original, reversible and reverse function 42
3.2 An execution of the algorithm described by relabeling and electing rules (corresponding to the code of Example 2 Figure 3.3) 47

5.1 List of killer and EUK nodes 73
5.2 Contribution of reverse rematerialization to the minimization of register requirements 87
5.3 Cost of the reversibility: number of additional operations 88
5.4 Contribution of reverse rematerialization to minimize spill operations 89

6.1 Contribution of reverse rematerialization to improve performance 100
6.2 Hopping_Matrix_k 105
6.3 Hopping_Matrix_l 105

7.1 Contribution of reverse rematerialization to increase thread level parallelism 115
7.2 Contribution of reverse rematerialization to increase performance on NVIDIA GPU (gflops) - original implementation - 115
7.3 Contribution of reverse rematerialization to increase performance on NVIDIA GPU (gflops) - optimized implementation - 115
7.4 Single thread performance on NVIDIA GPU (gflops) 115
7.5 Cell BE Vs NVIDIA GPU: single thread performance (gflops) 117
7.6 Cell BE Vs NVIDIA GPU: global performance (gflops) 117

Part I

Introduction

Chapter 1

Introduction

Contents

1.1 Context 20
1.2 Contribution 22
1.3 Organization of the Thesis 22

1.1 Context

For the three resources of computation - time, space, and energy, it has become a challenge for researchers and engineers in different fields to minimize them. However, minimizing one of these resources at all costs requires a disproportionately large amount of the other resources. The memory usage can be reduced at the cost of slower program execution, and the computation time can be reduced at the cost of increased power consumption. This is why finding a trade-off between these three factors has become the challenge of theory of computation.

In this research we are interested in a fourth factor which is information. Information has an important and direct impact on power consumption, memory space requirements, and execution time. Landauer [44] showed that loss of information should release dissipated energy. Therefore only reversible programs where no information is ever lost, are likely to be thermodynamically adiabatic. Reversibility means that data can always be retrieved from any point of the program. This was our primary motivation to study the relationship between reversible computing and information measurement in the context of information theory, and thereafter to study the relationship between information, space and time.

This dissertation presents a new method for improving program performance by exploiting information locality through reverse computing. The reason why we use the term "information locality" instead of "data locality" is because information may be carried not only by a data but also by the process and input data that generate it. Therefore information can be present in different forms. When computations are reversible there are more ways for retrieving information, not only from input data and direct computation but also from other already computed data and reverse computation. Hence reversible computing improves information locality. For example, for the instruction $c{:=}a{+}b$, the information in *(a,b,c)*, *(a,b)*, *(a,c)* or *(b,c)* is the same, so no need to keep the three values alive at the same time, because we can always compute one value from the two others. a can be retrieved from *(b,c)* and b from *(a,c)* by a simple subtraction. Hence, values that carry the same information can share the same register. Therefore, information locality can even optimize the storage space.

To take advantage of information locality, we study the conservation of information during an irreversible process. Conventional computing is a priori irreversible and does not have to conserve information. Adding two numbers together, for example, destroys information, unless we retain one of the numbers. This leads to the question *"what is the minimum amount of information we must retain to generate all - input, intermediate and output - values without any additional memory space demands?"*. Since the advantage of reversible computing is its ability to conserve information, we address the issue of making a program reversible in term of memory space.

Reversibility of programs has been studied by Bennett [10]. A first easy way to make a program reversible is to record the history of intermediate variables along the execution, but then the issue of erasing - forgetting - that information, called garbage, remains. That garbage can be used as an estimation of the intrinsics energy consumption of programs and *minimizing it is the objective.* Bennett[10] proved that if the input can be computed from the output then there is a reversible way of computing the output from the input while eliminating the garbage. This may be at the price of large space usage for storing all the intermediate states during the computation. This is why we take interest in studying the *spatial complexity of reversible programs.*

High performance by exploiting information locality may come from decreasing register pressure and as consequence decreasing spill code. In this work, we revisit register allocation issues from the reversible computing angle. While being a very old computer science problem, register allocation is always an important issue in architectures where memory access time and communication time have ever been increasing with respect to computing time. In register allocation one can use rematerialization instead of spilling, meaning that we recompute some value v instead of keeping it live in memory. Hence, the vertical memory hierarchy is stressed as little as possible. Recomputation is performed from values still stored in registers, in the same way as specified in the program. But there is a part of information of v carried by other values w that have been computed directly or indirectly from v. Hence this gives new opportunities for recovering the v value: undoing the computation from the w values, or in other words, reversely computing v. Therefore one of the questions that we address in this dissertation is *whether rematerialization by reverse computing – reverse rematerialization – can help improving register allocation.*

Register pressure is also a big issue for multi-core processors with shared memory or shared register file like GPUs. The number of registers and the amount of shared memory required by the kernel affects the number of running threads and thus performance. The optimization we found to be important, as GPUs runs most efficiently with a large numbers of threads, is to use rematerialization, that helps to minimize register requirements per thread. We mean that all registers containing values that can be recomputed, can be re-used. Register rematerialization generates additional instructions and we have to trade instruction count against register usage. We show that rematerialization with reverse computing is less sensitive than ordinary rematerialization with direct recomputing. This increases register reuse and reduces register demand per thread that should increase the number of active threads.

Another kind of parallelism that can take advantage of information locality is instruction-level parallelism. Parallelism is limited among instructions not only by data dependencies, but also by resource availability. By exploiting information locality, reverse rematerialization can contribute to increase instruction-level parallelism by increasing register reuse to allow more instructions to be executed simultaneously. In loop-level parallelism, even if a large register file can hold all values of one iteration, the processor might occasionally stall as a result of data dependencies and branch instructions. Rematerialization through reverse operations helps to reduce register usage per iteration to keep pipeline full by unrolling loops without any spill operations and exploit parallelism among instructions by finding sequences of unrelated instructions from different iterations that can be overlapped in the pipeline.

This work is done in the framework of the PetaQCD project that aims at designing architecture and program for the LQCD (Lattice Quantum ChromoDynamics) application. The lattice of sites on which computations are performed is a 4D lattice that is splitted into sublattices, each of which is managed by one processor. Since LQCD is highly communication demanding it is crucial that as few processors as possible are used, that means that parallelism within processors such as *instruction level or thread level parallelism is maximized.* One of the key questions to be addressed is whether a scientific application like LQCD should be parallelized on a multi-core CPU like Cell BE or accelerated on a GPU. This dissertation presents also our experiences in parallelizing this scientific simulation code from the computational nuclear physics domain. We test the effectiveness of all our optimization techniques on the LQCD application.

1.2 Contribution

The purpose of this thesis is twofold.

1. First, we address the issue of making a program reversible in terms of spatial complexity. Spatial complexity is the amount of memory/register locations required for performing the computation in both forward and backward directions. We present a lower bound of the spatial complexity of a DAG (directed acyclic graph) with reversible operations, as well as a heuristic aimed at finding the minimum number of registers required for a forward and backward execution of a DAG. We define energetic garbage as the additional number of registers needed for the reversible computation with respect to the original computation. We have run experiments that suggest that the garbage size is never more than 50% of the DAG size for DAGs with unary/binary operations.
   This work has been published in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems - CASES'2009 [4].

2. Second, we revisit register allocation issues from the reversible computing angle and we present a new register rematerialization technique based on reverse recomputing. We detail a heuristic algorithm for performing reverse register rematerialization and we use the high memory demanding LQCD (Lattice Quantum ChromoDynamics) application to demonstrate that important gains of up to 33% on register pressure can be obtained. We also show how instruction and thread parallelism may be improved by performing register allocation with reverse recomputing. Applying these optimizations on GPUs increases the number of threads per Streaming Multiprocessor (SM). This is done on the main kernel of Lattice Quantum ChromoDynamics (LQCD) simulation program where we gain a 10.84% speedup.
   These results has been published in two international conferences: in the ACM International Conference on Computing Frontiers- CF'2011 [6] and the International Symposium on Computer Architecture and High Performance Computing - SBAC-PAD'2011 [5]. This work has been awarded Best Paper for the Architecture Track, and Jùlio Salec Aude Award at the SBAC-PAD'2011 conference.

1.3 Organization of the Thesis

The remainder of this thesis is organized as follows: Part 1 is devoted to explain in detail the concepts of reverse computing and information conservation. This part is composed of the following chapters: Chapter 1 reviews previous work related to these concepts and discusses various applications and approaches of reversible computing. In chapter 2 we tackle the problem of reversing a program and we examine the memory cost of the reversibility. Part 2 of this thesis describes different approaches for improving program performance by exploiting information locality through reverse computing. This part is organized as follows: In chapter 3 we discuss previous work on register allocation techniques - to reduce register pressure - related to the work presented in this dissertation. In chapter 4 we present a new rematerialization technique based-register allocation through reverse computing to solve the problem of memory access. Chapters 5 and 6 address issues related to instruction-level parallelism and thread-level parallelism and show how using reverse computing can improve the parallelism in a program. We provide the results of

our optimizations on Cell BE and Nvidia GPU. Finally, chapter 7 contains the conclusion of this work and suggests possible directions for future research.

Part II

Reversible Computing & Information Conservation

Chapter 2

Reversible Computing: Definition and Motivation

Contents

2.1 Power Consumption and Heat Dissipation 28
2.2 Energy Dissipation and Reversibility 29
2.3 Reversible Computing 30
2.3.1 Reversible Operations 31
2.3.2 Reversible Logic 33
2.3.3 Reversible Logic Gates 34
2.4 Reversible Architecture 36
2.5 Reversible Software 37
2.6 Summary 38

One of the most important barriers to the performances of processors is power consumption and heat dissipation. Several works consider how to make architectures and programs cheaper in terms of energy. Thermal models are designed based on underlying electronics for architectures, or based on some software rules for programs, for instance balancing resources usage [35], or minimizing caches misses, or cutting unused devices. There is also a radically different approach that tackles energy issues under the point of view of intrinsics thermodynamics of computation and the basis argument that likewise irreversible thermodynamics transformations, irreversible programs have to dissipate heat. This originates from the Landauer remark [44] that erasing or throwing away a bit information at temperature T must dissipate at least $k_B T ln 2$ of energy, where k_B is Boltzmann's constant. Therefore only reversible programs are likely to be thermodynamically adiabatic. Reversibility means here that no information is ever lost, it can always be retrieved from any point of the program.

This chapter discusses the original motivation of studying reversible computing, as well as its definition and its numerous applications at different levels of a computer system.

2.1 Power Consumption and Heat Dissipation

To better understand why electronic computers dissipate energy, we come back to energy dissipation in conventional circuits. Power dissipation in CMOS circuits arises from two different mechanisms: static power consumption which occurs from resistive devices, and dynamic power consumption, which results from switching capacitive loads between two different voltage states. A boolean variable is implemented as electrical node whose voltage represents the present value of the variable, as shown in figure 2.1.

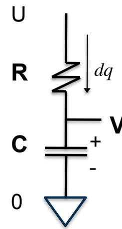


Figure 2.1: Simple model of CMOS system

R is a resistor that varies the power-supply voltage V at the point V. This point is a simple conductor, equipotential, that represents a boolean variable. C is a constant capacitor. The energy dissipated E_d by R when we move charge $Q = CU$ between the conductors in the capacitor considering an incremental change of V from 0 to U is given by:

$$E_d = \int_0^{CU} (U - V) dq$$

$$= U \int_0^{CU} dq - C \int_0^{U} V dV$$

$$= CU^2 - \frac{1}{2} CU^2$$

$$E_d = \frac{1}{2}CU^2$$

The term CU^2 is the total amount of energy provided by the power-supply. $\frac{1}{2}CU^2$ is the amount of the energy stored in the capacitor. Their difference $\frac{1}{2}CU^2$ must have been dissipated as heat. Note that the amount of energy dissipated E_d is always independent of the resistor R [48].

2.2 Energy Dissipation and Reversibility

In the following we will discuss about the thermodynamic view of energy dissipation. The second law of thermodynamics states that any transformation from one state to another involves an increase of global disorder caused by heat dissipation. This disorder is measured by function called `entropy` S. According to this law, the entropy of a system cannot be conserved specially in case of irreversible transformation [23]. The entropy of a system changes at temperature T absorbing an amount of heat δQ is given by

$$dS = \frac{\delta Q}{T}$$

In other words, in any irreversible process where the system gives up energy ΔE, and its entropy falls by ΔS, a quantity at least $T \times \Delta S$ of that energy must give up as waste heat limiting the amount of work a system can do.

However, according to the Clausius equality, the entropy of an isolated system can be constant if the transformation is reversible [46]

$$\oint \frac{\delta Q}{T} = 0$$

Boltzmann [38] gave a statistical definition of entropy in function of Ω; the number of micro-states or the number of configurations defining a macro-state. He showed that his definition was equal to the thermodynamic definition. The statistical entropy is defined as

$$S = K_B \; ln \; \Omega$$

where K_B is a constant number known as the Boltzmann's constant K_B=1.38066$\times 10^{-23}$JK^{-1}.

Based on that Gibbs [38] defined the entropy for a classical system with discrete set of micro-states, using probability theory. If p_i is the probability that micro-state i occurs during the system's fluctuations, then the entropy of the system is

$$S = -K_B \sum_i p_i \ln p_i$$

The statistical and thermodynamic entropy can be interpreted as an application of the information entropy known as Shannon's entropy [58] which quantifies the average information contained in a message. In other words, it is a function to measure the uncertainty of information. The Shannon's entropy of a discrete random variable x consisting of n symbols where a symbol i has a probability $p(i)$ to be appeared is defined as:

$$H(x) = -\sum_{i=1}^{n} p(i) log_2 \; p(i),$$

assuming that x is coded on n bits, where each bit can be in two states 1 or 0. Erasing a bit i is equivalent to lose the information of that bit, which increases the number of states (or level of uncertainty) that i may take, implying an increase of the entropy H.

Landauer [44] started from the fact that information must obey the laws of physics, and proved that information loss which is a thermodynamically irreversible process, is equivalent to energy loss. He showed that erasing or throwing away a bit information at temperature T must dissipate at least

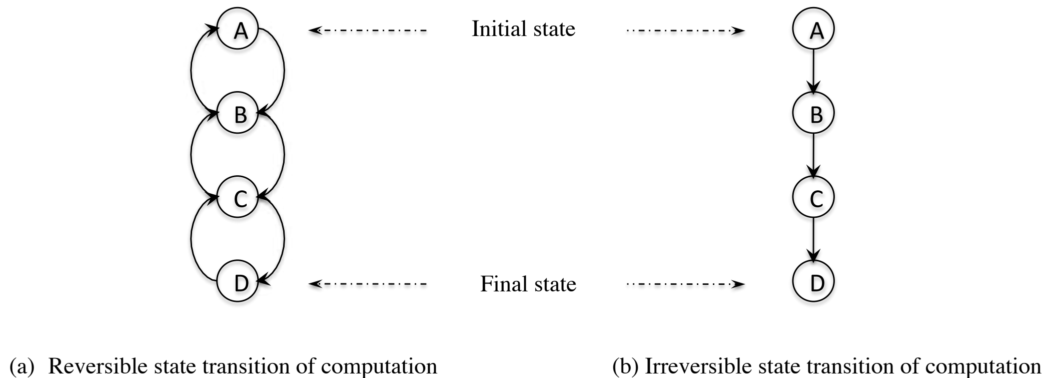
$$K_B T ln2 \quad Joules\ (J)$$

of energy. In general erasing a bit of information increases the number of possible states by factor of 2, therefore the entropy of a system that can be in W states is:

$$K_B\ T\ lnW$$

Therefore, reducing heat dissipation requires that information entropy must be conserved, and conserving the entropy means that the computation process must be reversible. Reversibility of computation means that no information is ever lost, it can always be retrieved from any point of the program.

For example, at room temperature $T = 300$ kelvins, the amount of dissipating heat when erasing one bit is small (i.e., $2.9\times\ 10^{-21}$ joule), but is becoming more and more not negligible compared to the energy cost per logic operation which is already about $10^2\ K_B T ln2$, as shown in Figure 2.3, that continues to fall steadily with the exponential growth of computer performance predicted by Moore's law thanks to the miniaturization of transistors that improved energy efficiency. This is why studying heat dissipation minimization from the thermodynamics view of computation becomes more and more interesting.



(a) Reversible state transition of computation (b) Irreversible state transition of computation

Figure 2.2: State transition: reversible computing vs. irreversible computing

2.3 Reversible Computing

Reversibility of computation has been studied in several works [10, 26, 64] with a first aim to reduce the energy dissipation due to the irreversibility of computing process. A computing is reversible if we can always reconstruct a previous state from the actual state as it is shown in Figure 2.2. The precondition is preservation of all information throughout the entire process of computing.

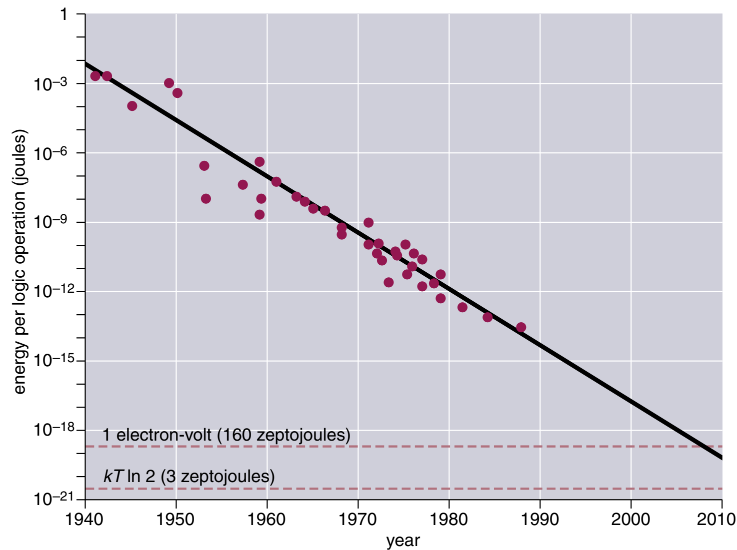


Figure 2.3: Thermodynamic dissipation vs. technological dissipation

2.3.1 Reversible Operations

A boolean function $f(x_1, x_2, ..., x_n) \rightarrow (y_1, y_2, ..., y_k)$ with n input boolean variables and k output boolean variables is called reversible if it is bijective. This means that the number of outputs is equal to the number of inputs and each input pattern maps to a unique output pattern.

Example 1
Consider the XOR $\oplus$ operation. $f(x, y) = \ x \oplus y$ defined by this table:

x	y	x ⊕ y
0	0	0
0	1	1
1	0	1
1	1	0

The operation $\oplus$ is destructive and does not conserve information, it is irreversible operation since we cannot retrieve x and y from $x \oplus y$ because 0 in $f(x, y)$ can be computed from several distinct input pairs (e.g. $< 0; 0 >, < 1; 1 >$). It is necessary that the number of inputs be equal to the number of outputs. One of the ways to make it reversible is adding a second output which can be x in this example, so that the function becomes as shown bellow.

x	y	x ⊕ y	x
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Example 2
Consider the basic arithmetic operations like the increment function, defined from the set of integers Z to Z, that to each integer x associates the integer $y := x + 1$. The inverse function is $x := y - 1$, easily determined from the result uniquely, which means that the increment operation is reversible.

Bennett [10] proved that all operations can be made reversible, however, this goal may not be achieved for free, an additional information, called `garbage`, may be required to make them reversible. A first easy way to make a program reversible is to record the history of intermediate variables along the execution, but then the issue of erasing - forgetting - that information, garbage data, remains. That garbage can be used as an estimation of the intrinsics energy consumption of programs and minimizing it is the objective. Bennett [10] proved that if the input can be computed from the output then there is a reversible way of computing the output from the input while eliminating the garbage. This may be at the price of large space usage for storing all the intermediate states during the computation.

In the following we note a conventional function by:

$$f(x) = y$$

A reversible function f_r that returns the same value as the function f but produces a garbage data g which not useful for the final result but required for reversibility, typically has the form:

$$f_r(x) = (y, g)$$

The associated inverse function f_r^{-1} that uses the garbage data g and the output value y to regenerate the input value x is represented as:

$$f_r^{-1}(y, g) = (x)$$

A simulation of irreversible computing that computes $f(x)$ from x to a reversible computing that compute the same result by using only reversible operations, can be done as is shown in Table 2.1 by: first, computing $f(x)$ and saving the computing history, garbage g, then making a copy of $f(x)$ which is a reversible operation, then recomputing x from the copy of $f(x)$ and g, and erasing in the same time the computing history g. Then, computing x from $f(x)$ and save a new garbage g'. The double x we got can be reduced to one x. Finally undo x and g' by computing $f(x)$. At the end we got $f(x)$. All actions described in Table 2.1 are reversible.

Action	**Work**
Initial configuration	x
Compute $f(x)$ and save garbage g	$f(x), g$
Copy $f(x)$	$f(x), f(x), g$
Reverse computing of x from $f(x)$ and g	$x, f(x)$
Compute x from $f(x)$ and save garbage g'	x, x, g'
Cancel extra x	x, g'
undo computing of x from $f(x)$ and g'	$f(x)$

Table 2.1: Reversible simulation of irreversible computing

2.3.2 Reversible Logic

Let a reversible function $f : \{0,1\}^n \to \{0,1\}^n$ of n boolean variables, such that each output pattern corresponds to one and only one input pattern. The output patterns can be considered as a permutation of 2^n elements of n bits on the input values, so the set of bijective - reversible, functions with 2^n binary inputs of n bits is $2^n!$. In other words, in reversible logic, there are $2^n!$ reversible gates among $(2^n)^m$ of all possible gates with m outputs on n bits.

For example, in the truth table shown bellow, the output $(x_0 \oplus x_1, \neg x_0)$ is just a permutation of the input (x_0, x_1). In total, 24 different 2-bit reversible truth tables can be constructed with two boolean variables.

x_0	x_1	$x_0 \oplus x_1$	$\neg x_0$
0	0	0	1
0	1	1	1
1	0	1	0
1	1	0	0

Hamming distance

The Hamming distance between two strings of bits x and y of equal length n which defined as the number of bit places in which x and y are different, can help to determine if a logic function is reversible or not. The Hamming distance can be found by counting the number of 1 in a bitwise XOR operation between the two strings.

$$d(x,y) = \sum_{i=0}^{n-1} (x_i \oplus y_i)$$

In general, a reversible logic function $f(x) = y$ can be expressed by a truth table of 2^n boolean inputs and 2^n outputs denoted x_i and y_j respectively with $i, j \in [0, 2^n - 1]$.

$$x = (x_i)_{i \in [0,2^n-1]} \quad et \quad y = (y_j)_{j \in [0,2^n-1]}$$

Example

if $x = (100110101)$ and $y = (101100111)$ then $x \oplus y = (001010010)$ and $d(x,y) = 3$

Using Hamming distance we give some results on reversible logic operations:

- if $\exists i, j \in [0, 2^n - 1]$ such that $i \neq j$ $d(y_i, y_j) = 0$ then the function f is not injective and therefore not reversible.
- if $\forall i, j \in [0, 2^n - 1]$ such that $i \neq j$ $d(y_i, y_j) \geq 1$ then the function f is bijective and therefore reversible.
- if $\exists i, \forall j \in [0, 2^n - 1]$ such that $i \neq j$

  $$\sum_{j=0}^{2^n-1} d(y_i, y_j) = C_n^1 + C_n^2 + ... + C_n^n$$

  then the function f is reversible.

2.3.3 Reversible Logic Gates

Most digital gates that implement logical functions are not reversible, such the AND and OR operations. Other ones are reversible like the NOT gate. A design of universal reversible logic circuits requires a set of reversible gates. Several such gates have been proposed. Among them are the Feynman [28] gate also known as controlled-NOT (CNOT) which is 2-bit gate. It can be described by the function $f(x_0, x_1) = (y_0, y_1)$ such that $y_0 = x_0 \oplus x_1$ and $y_1 = x_1$. A 2×2 Feynman gate is shown in Figure 2.4.

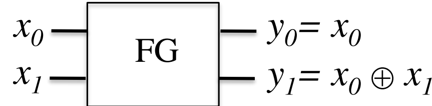


Figure 2.4: Feynman gate

Toffoli [64] invented a universal reversible logic gate called Toffoli gate, also known as controlled-controlled-NOT CCNOT gate, which means that any reversible circuit can be constructed from Toffoli gates. It is a 3-bit gate that implements the logic function $f(x_0, x_1, x_2) = (y_0, y_1, y_2)$ with $y_0 = x_0$, $y_1 = x_1$ and $y_2 = x_0x_1 \oplus x_2$. A 3-input and 3-output Toffoli gate is shown in Figure 2.5.

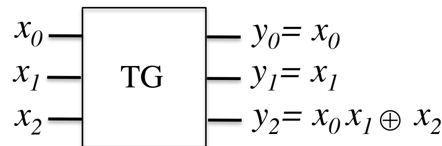


Figure 2.5: Toffoli gate

The Fredkin gates [26] is a 3-bit gate with 3 inputs and 3 outputs, it can be represented as: $f(x_0, x_1, x_2) = (y_0, y_1, y_2)$ with $y_0 = x_0$, $y_1 = \bar{x_0}x_1 \oplus x_0x_2$ and $y_2 = \bar{x_0}x_2 \oplus x_0x_1$, as it is shown in Figure 2.6. The Fredkin gate is a conservative gate, that is, the Hamming weight[1] of its input equals the Hamming weight of its output.

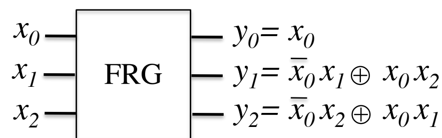


Figure 2.6: Fredkin gate

A reversible logic circuit requires that all gates used be reversible and be interconnected without fan-out[2] in acyclic composition. As with reversible gates, a reversible circuit has

[1]The Hamming weight of a string is the number of 1 in its binary expansion.
[2]The fan-out of a logic gate output is a term that defines the maximum number of gate inputs to which it is connected.

the same number of inputs and outputs. For example, in Figure 2.7 we show an example of reversible circuit composed of four reversible gates.

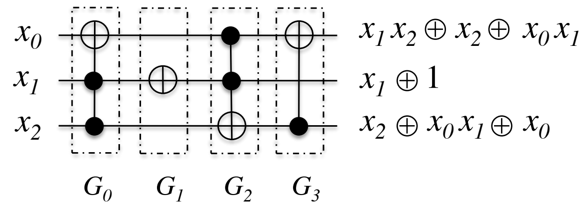


Figure 2.7: Example of reversible logic circuit.

The aim of reversible logic circuit designers is, first, to minimize the gate count, and second, to minimize the garbage outputs and/or constant inputs which is more important than the number of gates used from the point of view of reversible logic [24].

When the number of outputs is smaller than the number of inputs, the garbage outputs is unavoidable, since the reversibility necessitates an equal number of outputs and inputs, and when these numbers are equal and one output pattern occurs more than one time, the constant inputs are required to make the corresponding truth table reversible.

Example

We take back the example given by Maslov in [24]. Consider the AND function between two binary variables denoted by their concatenation $f(x_0, x_1) = x_0x_1$. With respect to the reversibility proprieties, adding one single output to make the number of inputs and output equal does not help to make this function reversible. One of the ways to make it reversible is to add one input and two outputs so that the function becomes as shown in Table 2.2. The desired result, shown in gray lines, can be obtained by setting the value of variable c to 0, (where the word `constant` comes from). The two copies g_0 and g_1 of x_0 and x_1 respectively are called `garbage` because they are not useful for final result.

x_0	x_1	x_0x_1
0	0	0
0	1	0
1	0	0
1	1	1

(a) irreversible AND function

x_0	x_1	c	$g_0 = x_0$	$g_1 = x_1$	$x_0x_1 \oplus c$
0	**0**	**0**	0	0	**0**
0	0	1	0	0	1
0	**1**	**0**	0	1	**0**
0	1	1	0	1	1
1	**0**	**0**	1	0	**0**
1	0	1	1	0	1
1	**1**	**0**	1	1	**1**
1	1	1	1	1	0

(b) reversible function computing the logical AND

Table 2.2: Irreversible vs. Reversible function

Garbage minimization

The term garbage was defined in several works [26, 49, 50], however it does not include constant inputs. Maslov defined the garbage as the set of both garbage outputs and constant inputs, and gave a lower bound for garbage bits for a function
$f : \{0, 1\}^n \rightarrow \{0, 1\}^k$. If M is the number of times an output pattern is repeated in the truth table then the minimum number of garbage bits required to make it reversible is $\lceil log(M) \rceil$. As a result of this theorem, we give a lower bound for input constants which is $\lceil log(M) \rceil - n$ if $\lceil log(M) \rceil > n$ else 0. Hence, the aim of reversible circuit designers is to simulate irreversible functions with only $O(Log\ M)$ garbage bits.

Information distance

For two strings x and y (there is no fixed program here), we see the problem of garbage minimization in Bennett's reversible machine as finding the minimum amount of information g, as well as the result y, to regenerate input values x. This is similarly to the conditional Kolmogorov complexity [42] $K(x|y)$ of a string x relative to a string y. $K(x|y)$ is defined as the size of the shortest program to compute x if y is furnished as an auxiliary input to the computation. In other words, the problem is to find the minimum amount of information required to program a reversible computation from x to y and from y to x. This notion is defined as `information distance for reversible computation` [9] denoted by E_r.

$$E_r(x, y) = K_r(x|y) = K_r(y|x)$$

with K_r is the shortest reversible program that computes y from x and x from y.
For example let take the addition operation $x_1+x_2 = y$. The expression 2+2 corresponding to the first term of the equality contains more information than the expression $y = 4$. All what we know from the second expression is that two numbers have been added which can be from different numbers $1 + 3$, $2 + 2$, $4 + 0$, etc. In this case we say that

$$K(x|y) > K(y|x)$$

with $x = (x_1, x_2)$ which means that more information is required for y to find x than for x to find y.

2.4 Reversible Architecture

Reversible computation requires reversible architectures that must implement a reversible instruction set in a reversible circuit in order that logic operations may be thermodynamically reversible. Because performing exclusively reversible logic operation is necessary but insufficient for a thermodynamic reversibility.

The first reversible processor was `Pendulum` [37] developed at the MIT's reversible computing research group in 1995. Pendulum is a 12-bit fully adiabatic implementation with a complete instruction set and register transfer level datapath based on a RISC processor designed to achieve much lower power.

After Pendulum, and in 1998, the same research group built a fully reversible FPGA chip called `Flattop` [2]. Flattop emulates the Billiard Ball model of computation introduced by Fredkin [26] which is universal and fully reversible.

Other research works on quantum architecture consider various problems of reversible logic circuits synthesis [59, 69], because quantum computers use quantum gates, which are always reversible operators. Recently, researchers in Switzerland at EPFZ led by professor Renato Renner, have laid the theoretical basis for developing a quantum computer that does not dissipate heat, and which will harness the power of atoms and molecules to perform memory and processing tasks. Jeffrey Bokor and his team at UC Berkeley [43] are working to develop magnetic computers that would require no moving electrons and would store and process information using magnets. Their aim is to approach the theoretical energy limit set by Rolf Landauer.

2.5 Reversible Software

Reversibility of computation has several applications at the software level, among which we can mention bidirectional debuggers [57, 61], rollback mechanisms for speculative executions in parallel and distributed systems [19], simulation and error detection techniques [13]. There are also algorithms that require a pass where intermediate results have to be scanned in reverse order. This happen for instance in reservoir simulation [62], or in automatic program differentiation [36].

Two mains approaches have addressed the reversibility at the software level to design reversible applications. Since writing reversible programs by hand is not quite natural, some works are devoted to the design of reversible programming languages [7, 45, 29]. The alternative approach is to convert existing programs written in an irreversible programming language into equivalent reversible programs. An irreversible-to-reversible compiler receives an irreversible program as input and reversibly compiles it to a reversible program [39].

When one converts irreversible programs into reversible ones one has to face the issue of trading time complexity with data storage complexity. One elegant method was proposed by Bennett [8] where he models the former problem with a pebble game. Pebbles represent available data at some point of the program. One can add pebbles on some node when there is a way to compute that node with data identified by a pebble in a previous node. One can remove pebbles if there is an alternative way to recompute data required in this node. This is therefore an abstraction of reversible computations that allows analysis of the space and time complexity for various classes of problems, but this simulation operates only on sequential list of nodes. This sequence is broken hierarchically into sequences ending with checkpoints storing complete instantaneous descriptions of the simulated machine. After a later checkpoint is reached and saved, the simulating machine reversibly undoes its intermediate computation, reversibly erasing the intermediate history and reversibly canceling the previously saved checkpoint. Bennett chose the number of pebbles large enough ($n = O(logT)$) so that m the number of steps become small.

In [68], Vitanyi gives a time-space trade-off for an irreversible computation using time T and space S to be simulated reversibly in time $T' = 3^k 2^{O(T/2^k)} S$ and space $S' = S(1 + O(k))$, where k is a parameter that can be chosen freely $0 \leq k \leq log(T)$ in order to obtain the required trade-off between reversible time T' and space S'. Naumann [51] considers the problem of restoring the intermediate values computed by such a program (the vertices in the DAG) in reverse order for a given upper bound on the available memory, while keeping the computational complexity to a minimum. He shows that the optimal data-flow reversal problem is NP-complete,[51] but he doesn't consider reversible

operations and seeks only reversible behavior for recovering intermediate values in reverse order. Griewank also in [36] presents an optimal time-space trade-off algorithm in the context of automatic differentiation. In [24] it has been showed that the minimum number of garbage bits required to make a boolean function reversible is $\lceil log(M) \rceil$, where M is the maximum of number of times an output pattern is repeated in the truth table. However, traditional techniques for bi-directional execution are not scalable to all classes of problems.

2.6 Summary

As it has been presented in this chapter, several works on reversible computing have been realized with respect to energy, time and space requirements, and various of its applications have been proposed at different levels of computer system, including hardware design, logic synthesis and software conception. Each of them introduces new terms, new designs and new implementations.

In the next chapter, we take interest in studying a fundamental factor for improving time, space and energy which is information, and its relationship with storage and recomputing. Since reversible computing allows the conservation of information, one important question we answer in the following is how to make a program reversible.

Chapter 3

Reversible Computing for Information Conservation

Contents

3.1 Cost of Reversibility and Algorithm **40**
3.1.1 Reversible Operations 41
3.1.2 Algorithm 43
3.2 Reversible DAG and Register Reuse DAG - Lower Bound . . **48**
3.3 Reversibility and Values Lifetime **50**
3.4 Experimental Results and Upper-Bound for the Garbage Size **52**
3.5 Summary **54**

In this chapter we study the conservation of information during a computation process. We want that a program does not destroy the information but only create it, and that available information should be always sufficient to permit an exact reconstruction of data.

Since the property of reversible computing is the conservation of information, we want to characterize the intrinsic reversibility of a program or piece of program based on its data dependency graph and not only on some linear sequence of instructions. We consider only DAGs (directed acyclic graphs), this means basic blocks in DDG (data dependency graphs). Compared to other works we consider possible rescheduling of instructions instead of a fixed sequence of already scheduled instructions. And the question that we address is: "given a DAG computation graph with `reversible` operations, what is the minimum amount of garbage necessary to make the whole DAG reversible?". This is equivalent to study `the spatial complexity of reversibly computable DAG`.

Our ambition is therefore modest compared to this major and important issue of understanding precisely the relationship between physics, information, measurement, observers in one hand and information theory, computing in the other hand [30]. We believe however that this work may help optimizing scheduling and data storage in applications mentioned just before this chapter.

The remaining part of the chapter is organized as follows. In Section 3.1 we explain more precisely our basic hypothesis and how we relate the problem of garbage minimization to the problem of register allocation, namely the number of registers required for reversibly executing the computation DAG. Definitions and strategies for register allocation are detailed in Chapter 4. We give a heuristic algorithm for finding the number of additional registers ("garbage") required. In section 3.2 and 3.3 , we propose a lower bound based on the decomposition of DAG into elementary paths and we compare the values lifetime between reversible and irreversible computing. Systematic experiments (section 3.4) with our heuristic algorithm suggests that garbage is never more than $n/2$ where n is the operation count in the DAG. Finally, we summarize our main results in section 3.5, and discuss our following works in the second part on exploiting information locality for improving performance.

3.1 Cost of Reversibility and Algorithm

In this section, we present our approach and algorithm for computing the spatial complexity of reversing a DAG. We consider a DAG of operations, typically the data dependency graph of instructions within a basic block [1], see the part (a) of Figure 3.1. A basic block is a linear sequence of instructions with a single entry point and a single exit, meaning there is no jump instruction into the code except at the last instruction. Once the first instruction in a basic block is executed, the rest of the instructions will for certain be executed exactly once. Basic blocks are represented as directed acyclic data dependency graph with precedence constraints. Nodes of the graph are instructions denoted by the name of the variable carrying the result. The directed edges represent data dependencies between instructions which must be satisfied to ensure correct program semantics. This makes sense as two different nodes need be treated as two different variables. We consider only unary and binary operations and we make the important hypothesis that they can be made `reversible`.

[1]Basic blocks are usually the basic unit to which compiler optimizations are applied.

3.1.1 Reversible Operations

A boolean function $f(x_1, x_2, ..., x_n)$ with n input boolean variables and k output boolean variables is called reversible if it is bijective. This means that the number of outputs is equal to the number of inputs and each input pattern maps to a unique output pattern. Based on that we make the very rough abstract approximation that the operations in the DAG are reversible in the following sense: for unary operations, they are bijective so that the operand is uniquely determined by the result. For example, consider the increment function, defined from the set of integers Z to Z, that to each integer x associates the integer $y := x + 1$. The inverse function is $x := y - 1$, easily determined from the result uniquely. For binary operations, this means that only one additional value beside the result is needed for recovering both operands from this result and this additional value. This is typically the case of the basic arithmetic operations, like addition $c := a + b$, where (a, b) can be retrieved from (a, c) or (b, c) by a simple subtraction. Hence the `'+'` operation is considered as having two operands and two results.

This is only an abstraction and we are aware of a number of flaws underlying the concretization of this assumption. For instance the multiply $'*'$ operation needs at least one additional resulting bit for determining which of both operands was 0 if the result is 0. There are also data precision issues especially with floating point operations and round-off problems but we neglect them and count only the number of data as a measure of memory/register space. We could also consider the semantics of operations and transform the operations in order to minimize the space for storing intermediate results. This is for instance done by Burckel et al. in [56] where they show that the computation transforming n inputs into n outputs can be (reversibly) performed by using a storage space not greater than n. Therefore in our abstracted model, when executing a binary operation we have the choice of memorizing the first or the second operand or both, provided that the reverse operation is possible based on the result and memorized operands.

In Table 3.1, we give some rules to reverse some basic operations.

What we want to evaluate is the maximum size of storage needed to execute the operations of the DAG reversibly in a forward and backward execution, or in other words we want to minimize the history required for performing the DAG reversibly. The additional storage space required compared to a simple forward execution is our criterion of "**energetic garbage**". With this criterion we want to characterize the intrinsic energetic cost of the DAG. Compared to the Bennett strategy for minimizing the storage space with checkpoints we have the degree of freedom to choose the schedule of operations. Finally our problem is *finding a schedule that minimizes the register requirement in a forward and backward execution. The difference between the bidirectional execution and the forward execution is called the energetic garbage.*

As an illustration, consider the code segment shown in Figure 3.1(a) with its corresponding pseudo-assembly code and dependence graph in which each node corresponds to a statement in the code segment. This pseudo code requires only two registers. Figure 3.1(b) shows a reversible code that returns the same result as the code in Figure 3.1(a) and saves also two intermediate values. This code requires three registers for the three outputs g, f and c. Figure 3.1(c) shows the reverse code and its corresponding reverse graph derived from the reversible code in Figure 3.1(b). It shows how a reverse computation could be performed to generate all previous values. Thus, 2 registers are required in the forward computation, 3 in the forward and backward computation. The energetic garbage is 1.

Operation	**Original**	**Reversible**	**Reverse**
Addition	$f(x, y) = x + y = z$	$f_r(x, y) = (x + y, g)$ with $g = y$	$f_r^{-1}(z, g) = (z - g, g)$
Subtraction	$f(x, y) = x - y = z$	$f_r(x, y) = (x - y, g)$ with $g = y$	$f_r^{-1}(z, g) = (z + g, g)$

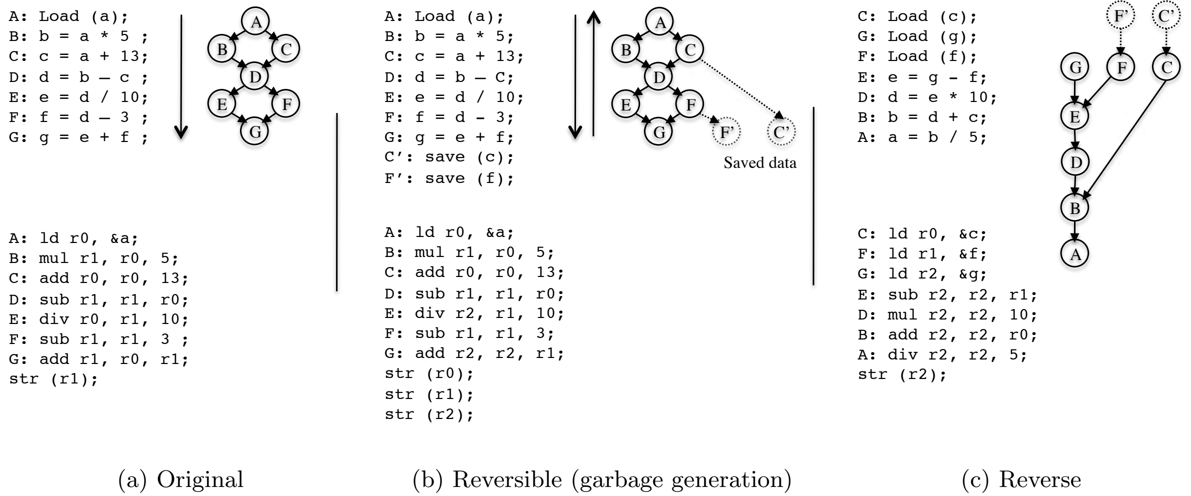
Multiplication	$f(x, y) = x * y = z$	$f_r(x, y) = (x * y, g, bit)$
		s.t.
		if $x \neq 0$ **then**
		$g \leftarrow x$
		$bit \leftarrow 0$
		else
		$g \leftarrow y$
		$bit \leftarrow 1$
			$f_r^{-1}(z, g) = (x, y)$ s.t.
			if $z \neq 0$ **then**
			$x \leftarrow g$
			$y \leftarrow z/g$
			else
			if $bit{=}0$ **then**
			$x \leftarrow g$
			$y \leftarrow 0$
			else
			$y \leftarrow g$
			$x \leftarrow 0$

Division	$f(x, y) = x/y = z$	$f_r(x, y) = (x/y, y)$	$f_r^{-1}(z, g) = (z * g, g)$
Sequence	$f_0;\ f_1;\ \cdots\ f_k$	$f_{0\ r};\ f_{1\ r};\ \ldots\ f_{k\ r};$	$f_{k\ r}^{-1};\ \cdots\ f_{1\ r}^{-1};\ f_{0\ r}^{-1};$
Branch	y: br x;	Saving the address of the last branch.	br y;

Condition	**if** *cond.* **then**
	$statement_0$
	else
	$statement_1$
		if *cond.* **then**
		$statement_{0r}$
		else
		$statement_{1r}$
			if *cond.* **then**
			$statement_{0r}^{-1}$
			else
			$statement_{1r}^{-1}$

Loop	**for**(i=0;i<n;i++)**{** $f_0(x_i)$; $f_1(y_i)$; ... $f_k(z_i)$; **}**	**for**(i=0;i<n;i++)**{** $f_{0\ r}(x_i)$; $f_{1\ r}(y_i)$; ... $f_{k\ r}(z_i)$; **}**	**for**(i=n-1;i≥0;i–)**{** $f_{k\ r}^{-1}(z_i)$; ... $f_{1\ r}^{-1}(y_i)$; $f_{0\ r}^{-1}(x_i)$; **}**

Table 3.1: Original, reversible and reverse function



(a) Original (b) Reversible (garbage generation) (c) Reverse

Figure 3.1: Illustrative example of reversible code and garbage generation.

3.1.2 Algorithm

Like all optimization problems in register allocation it is likely that garbage minimization is a NP-complete problem. But we have not proved it. Here we describe a heuristic that schedules a DAG in the direct order while keeping some variables alive in order to make the backward computation feasible. We call garbage the difference between number of registers in the direct computation and number of registers in the schedule found in this algorithm.

Starting from the input data, the DAG is scheduled first in the forward direction and then in the reverse direction and we are looking for a schedule that uses the minimum number of registers. Since we consider a DAG, the number of registers required by a schedule is also the maximum number of simultaneously live values during direct and reverse computation. One of the main issues of this work comes from the need to deal at the same time with the constraint of the minimum number of registers and with the constraint of saving values for enabling backward scheduling (reversibility constraint). Our algorithm has to arbitrate two kinds of information: first, which intermediate values will be saved in order to make the reverse computing feasible, and second, which successor node will kill the value - and hence will reuse the same register in the actual register assignment of the direct computation.

The algorithm scans the DAG $G = (V, E)$ and schedules instructions in some topological order only in the forward direction. V is the set of nodes and E the set of edges. At each step the instruction with highest priority is elected.

In the following, we give some preliminary definitions that describe the algorithms. Given a graph $G = (V, E)$ we denote by:

$output(v) = \{u \in V \mid (v, u) \in E\}$ the set of direct dependent nodes of v denoted by a direct edge from v in the graph G.

$input(v) = \{u \in V \mid (u, v) \in E\}$ the set of direct connected nodes by outgoing edge to v.

$dependent(v) = \{u \in V \mid u \in output(v) \vee u \in dependent(output(v))\}$, the set of all descendant nodes of v. In other words, it is the set of all nodes such that there is a path from v to these nodes.

independent$(v) = \{u \in V \mid u \notin dependent(v)\}$ the set of all nodes such that there is no path between v and these nodes.

source$(G) = \{u \in V \mid input(v) = \phi\}$ the set of input nodes of the graph G.

A *path* P is a finite totally ordered set of nodes $\{v_0, v_1, ..., v_{k-1}, v_k\}$ such that $(v_0, v_1),\ (v_1, v_2), ...,\ (v_{k-1}, v_k)$ are all edges in E.

A *chain* C is a set of pairwise comparable elements of V defining a total order, but there is not necessarily an edge between each two elements of a chain. Thus, every path is a chain but not vice versa.

The algorithm is presented on the following chart.

Input: DAG computation graph.
Output: number of registers required for reversible computation.

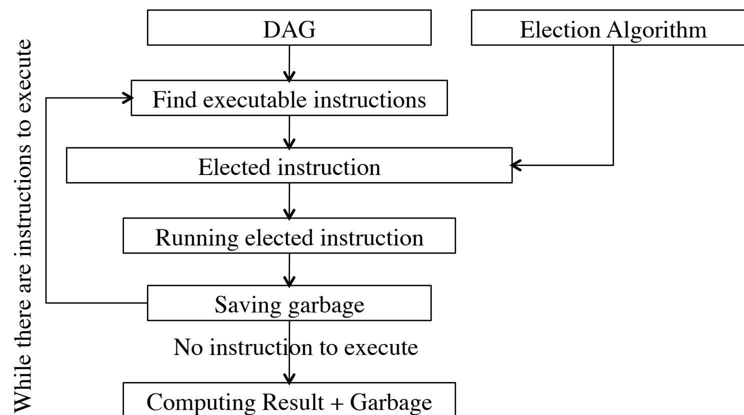


Figure 3.2: Scheduling algorithm diagram.

a. Priority

- We use a heuristic.
- Our scheduling is based on minimizing the number of resources (registers) without time constraint.
- The order in which ready instructions are selected affects the number of registers required and the garbage size.
- We favor instructions that have more predecessors with a minimum of successors in the DAG (this allows to use fewer registers)
- We favor instructions on the critical path. This will increase the number of calls to the scheduler.
- We reuse only registers of the direct predecessors (to preserve information).

b. Labeling
Each node can take one of the following labels:

- *Active*: if the value of a node is already computed and available in one of the registers.
- *Passive*: any already calculated node that will not be used in a future computing.
- *Ready*: if the value of a variable is ready to be calculated and all its predecessors are active.
- *Idle*: a node that is waiting because one of its predecessors is still waiting (waiting for all its predecessors to become active).

Initially all source nodes (nodes without predecessors) are set to *Active*, the others are set to *Idle*.

Labeling rules
Let us consider the functions λ and Ω that define the state of a node u from the set of global nodes V

$$\begin{aligned} \lambda : V &\longrightarrow \{Active, Passive, Ready, Idle\} \\ u &\longrightarrow \lambda(u) \end{aligned}$$

$$\begin{aligned} \Omega : V &\longrightarrow \{Elected, Not\ elected\} \\ u &\longrightarrow \Omega(u) \end{aligned}$$

These rules are applied at each stage of calculation.

$\forall u \in V \wedge |input(u)| = 0 \Rightarrow \lambda(u) = active$
$\forall v \in input(u) \wedge \lambda(v) = active \Rightarrow \lambda(u) = ready$
$\exists u\ \lambda(u) = ready \wedge \Omega(u) = elected \Rightarrow \lambda(u) = active$

Stop condition
$\forall u \in V : \lambda(u) = active \vee \lambda(u) = passive$

The final number of active nodes is the number of registers required.
The active nodes represent also the footprint needed for reversing the DAG computation.

Election Algorithm
As an entry, the list of values ready to be calculated

Rule 1: u is the unique successor of v that remains to be scheduled: u will reuse the register used by v.

```
if ∃u ∈ V  λ(u) = ready  {
        if ∃v ∈ input(u)/|output(v)| = 1  {
                Ω(u) = elected;
                λ(v) = passive;
        }  else
```

```
        if ∃v ∈ input(u)/∀w ∈ output(v) ∧ w ≠ u ∧ λ(w) = active {
                Ω(u) = elected;
                λ(v) = passive;
        }
}
```

Rule 2: Among ready nodes, one node with most successors is elected. For each pair of ready nodes $u,v \in V$

$\exists u\ \forall v\ (u,v) \in V\ \ \lambda(u) = \lambda(v) = ready$
 if $|output(u)| \geq |output(v)|$
 $\Omega(u) = elected;$

The second rule is applied if the first rule fails to elect an instruction (node). Once we find the elected instruction, we go the labeling rules.

c. Guarantees

This algorithm ensures that all nodes will be covered. In other words, all values will be calculated, and at the end, the number of active nodes is the number of register requirements and values of active nodes is a sufficient information for a backward generation of all values.

d. Analysis

We can show that the following algorithm allows electing a high priority instruction, which consumes fewer resources, at least at each computation step, this is a local optimization. At each computation step only one instruction is elected. The first rule allows the node who has a predecessor with one successor or predecessor of degree 1, to run first, this will not influence any other possible decision, and this node will reuse the register of this predecessor. The second rule is applied if the first rule fails to elect an instruction. It consists of choosing the variable at the greatest distance from the result and which has more successors, the idea behind this is to increase the number of calls to the scheduler and the set of candidates from which the elected will be chosen. Therefore if we consider the number of candidates for the election at step i is $NC_i = k$ then we want that at step $i{+}1 \quad NC_{i+1} \geq k$. Labeling each calculated node as active will reduce the degree of successor nodes and increase the priority of neighboring nodes. An execution of this algorithm on the DAG of Figure 3.3(b) is shown in Table 3.2. In the following, we show that for every DAG G, values of active nodes after running the algorithm allow the computation of all intermediate results back to input nodes.

e. Proof

We prove that from the nodes with active state we can find back all previous values.
At the end of the direct computation, we have only active or passive nodes, knowing that an active node changes its state to passive only during the call of the election procedure and in the back track, an idle node becomes active iff:

$\exists v \in input(u)\ \lambda(v) = active\ \wedge\ \forall w \in output(v)\ v \neq u\ \lambda(w) = active$

Nodes	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**
Initialization	A	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I
Labeling	A	R	R	I	I	I	I	I	I	I	I	I	I	I	I	I
Election	-	-	E	-	-	-	-	-	-	-	-	-	-	-	-	-
....																
Labeling	P	P	P	P	A	P	A	P	P	P	A	P	A	A	A	R
Election	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	E
Labeling	P	P	P	P	A	P	A	P	P	P	A	P	A	P	A	A

A: Active   P: Passive   R: Ready   I: Idle   E: Elected

Table 3.2: An execution of the algorithm described by relabeling and electing rules (corresponding to the code of Example 2 Figure 3.3)

Proof by contradiction:

Assuming there is always a node with idle state:
$\forall$ *instant* $t\ \exists u \in V\ \ \lambda(u) = idle\ \implies$
$\forall v \in input(u)\ \lambda(v) = idle \vee (\exists v \in input(u)\ \lambda(v) = active \wedge \exists w \in output(v)\ \ v \neq u\ \ \lambda(w) = idle)$

1. If $\forall v \in input(u)\ \ \lambda(v) = idle$ by recursion we find that $\exists u \in V/|input(u)| = 0\ \ \lambda(u) = idle$ which is contradiction, because we know that : $\forall u \in V/|input(u)| = 0 \implies \lambda(u) = Active$ (stop condition of the algorithm)

2. If one of its successor node is active and has a different predecessor in idle state: $\exists v \in input(u)\ \ \lambda(v) = active \wedge \exists w \in output(v)\ \ v \neq u\ \ \lambda(w) = idle$

This means there are two neighbor nodes in the idle state and each one of them is waiting for the other to become active, in other words both of them were passives at the end of the direct computation. A node becomes passive only during the call of the election procedure; in this case we have either:

$|input(u)| = 1 \wedge |input(w)| = 1 \wedge input(u) = input(w) = \{v\} \wedge \lambda(u) = \lambda(w) = passive.$

Which is impossible because v could not make two active nodes passive.

Let: $|input(u)| = 1 \wedge |input(w)| > 1$ This means that the state of w was modified from active to passive by another successor and not v, but as w is always waiting $\lambda(w) = idle$, it implies there is another idle neighbor which cannot modify its state and that there is another successor thanks to it, it took passive state previously.

Let take u_i the neighbor of u such that $|input(u_i)| > 1 \wedge \lambda(u_i) = passive \implies \exists u_{i+1}$ neighbor of u_i and $|input(u_{i+1})| > 1 \wedge \lambda(u_{i+1}) = passive \implies \exists u_{i+2} |input(u_{i+2})| > 1 \wedge \lambda(u_{i+2}) = passive \implies ....$

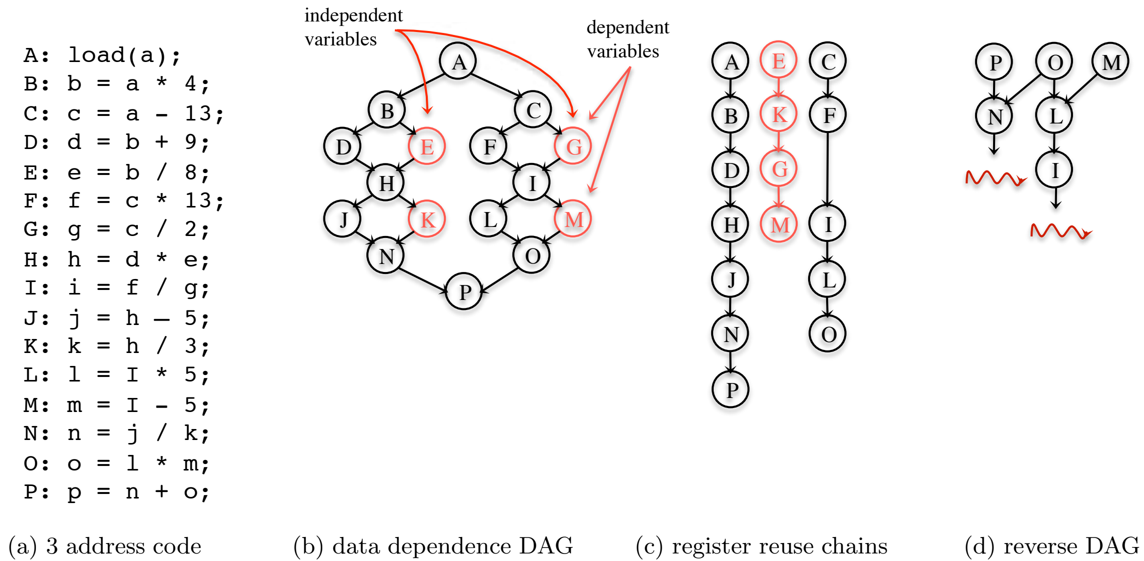
That leads us to the infinity, knowing that our graph is bounded (number of nodes is limited), which is contradictory.

We have shown that our algorithm at the end of a direct computation allows to re-browse all of the graph in the reverse direction, that means recomputing all intermediate values. The table 3.2 shows the execution of the algorithm on the graph of Figure 3.3.

3.2 Reversible DAG and Register Reuse DAG - Lower Bound

In this section we are seeking for a lower bound on the number of registers required for a DAG reversible computation. For that purpose we investigate the degree of register reuse in the reversible computation and we study its limitations in relation to the degree of dependency among variables.

Since the aim of reversible computing is the regeneration of data, some variables - contrary to the conventional computing, have to be saved even if they are not useful for the final result of (direct) computation. Therefore, register reuse is limited either between (a) independent values or (b) indirectly dependent values.



(a) 3 address code (b) data dependence DAG (c) register reuse chains (d) reverse DAG

Figure 3.3: Example of register reuse limitation for reversing a DAG

Figure 3.3 shows a 3 address code for a basic block and its corresponding data dependence DAG using statement labels. In this particular example, the minimum register requirement to compute 'P' is three. Figure 3.3(c) gives an example of register allocation that uses only three registers. However, as shown in Figure 3.3(d), three registers are not enough for saving certain intermediate variables during a direct computation for reversibly computing 'A' after computing 'P'.

Register reuse between independent values. Independent variables correspond to variables for which no dependency chain between both exists. They correspond to independent information. Since we want to be able to recover all intermediate values, this implies that for each chain in the graph we need at least one live variable at each step of the computation. This means that the number of chains in a minimal decomposition is a lower bound for register requirement in reversible computing. This is actually also a consequence of the Dilworth theorem.

Theorem 1 *The maximum number of independent elements in a partial order is equal to the number of chains in a minimal decomposition [25].*

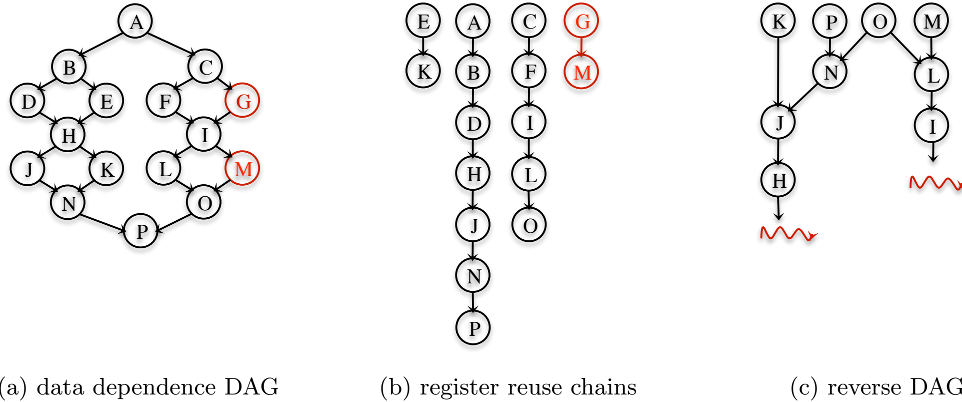
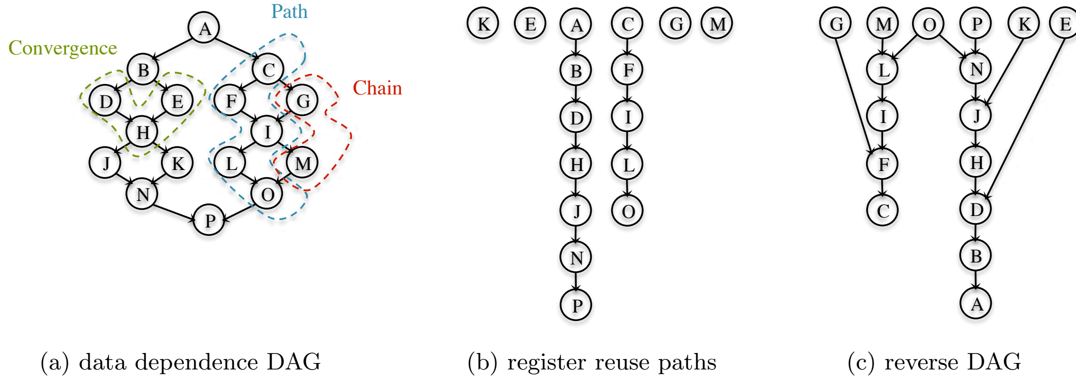


Figure 3.4: Example of register reuse limitation between dependent values for reversing a DAG

In the same example shown in Figure 3.3, at most four instructions can be executed in parallel because the number of chains in the minimal decomposition is four. An example of minimal chain decomposition is shown in Figure 3.4(b). The values computed by independent statements D, E, F and G can all be live at the same time and thus cannot share registers. Likewise, the values computed by J, K, L and M could not share the same registers since they are in separate chains, so there is no register reuse between K and M or K and G -contrary to the direct computation, nevertheless they are independent and they have not the same live range. Hence, the register requirement with respect to this condition is four but still not enough for reversibly computing this DAG, as shown in Figure 3.4(c).

Register reuse between dependent values. For dependent values the main difference in reversible computation compared to direct ordinary computation is that we can not simply reuse the register of a killed value because of **convergences** in the graph. A convergence is a binary operation which has two operands and one result. This corresponds to loss of information that has to be stored for backward computation. This means for instance that stronger constraints for reuse have to be met for reuse chains than just ordinary chains. At least there must be a dependency edge between both operations in order that the second one can reuse the register of the first one. Therefore, we need to consider decomposition of the graph into paths instead of chains. We prevent that a path contains a sub-path for the same reason (presence of a convergence).

The measurement of register requirement uses a Reuse DAG, which indicates which instructions can reuse a register used by a previous instruction. We use the same algorithm of the construction of the Reuse DAG for registers, proposed in [11], but the relation R that allows a value to reuse the register of a killed value requires that the defining instruction of the new value be the killing instruction of the previous one, in other words, it should be the last instruction that uses it. Therefore, all allocation chains of the reuse DAG are paths in the original DAG.



(a) data dependence DAG (b) register reuse paths (c) reverse DAG

Figure 3.5: Example of register reuse for reversing a DAG

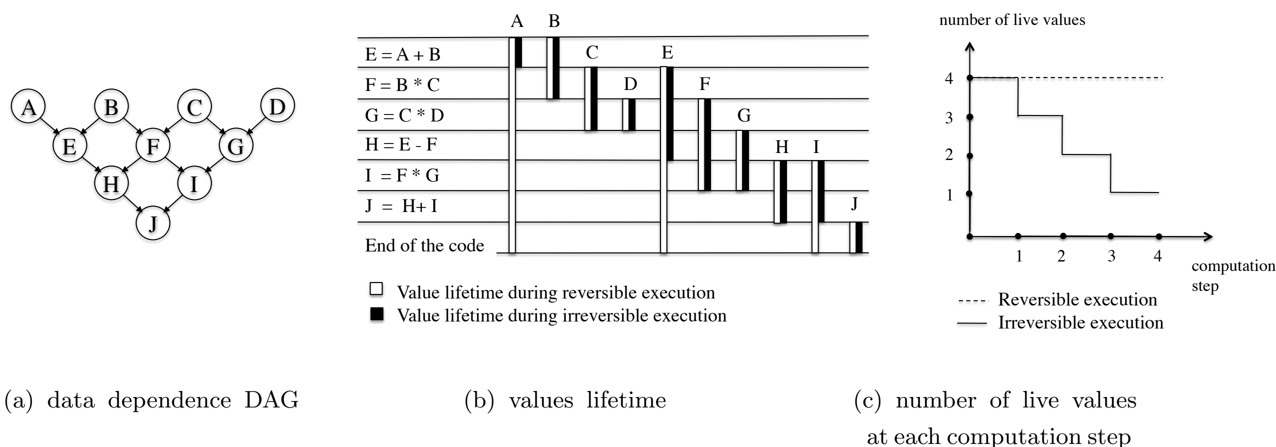
We come back to our example. G and M are indirectly dependent, at the point of use of M, G is already dead and could normally share the same register. But we prevent this, because we need G and I for recomputing F. In this example there are five convergences in the graph, which means a loss in information at five points during the computation, so we have to save additional information which is one of the inputs at each convergence. We can determine a lower bound of the size of this information by finding the minimal number of paths in the minimal decomposition of the reuse DAG. As shown in Figure 3.5(b), the sets of nodes $\{A, B, D, H, J, N, P\}, \{C, F, I, L, O\}, \{E\}, \{K\}, \{G\}$ and $\{M\}$ are all paths in the graph, and each end of each path can define an information that should be saved for the backward computation. The set of these paths is called register-reuse DAG because all nodes of a path are assigned to the same register. The reuse DAG chains those values that are not simultaneously live and can thus share a register, without any violation of information. Different paths are assigned to different registers, and therefore the number of paths is the number of registers. Thus, the register reuse DAG in Figure 3.5(b) requires at least six registers. Therefore, by Theorem 2, a minimum decomposition of a DAG into elementary paths that do not contain any sub-path gives a lower bound to the register requirement for a reversible execution.

Theorem 2 *The minimum number of registers required for a reversible execution of a program is bigger than or equal to the number of elementary paths in a minimal decomposition of the corresponding DAG.*

The reverse DAG of the original DDG, shown in Figure 3.5(c), shows how we can reconstruct all previous computed data.

3.3 Reversibility and Values Lifetime

There are cases when conventional and reversible computations consume the same number of registers, but values lifetime is not the same. As an example, consider the DAG in Figure 3.6(a). Both of the executions -reversible and non-reversible, require 4 registers for the computation. A register is used to hold a value from the time that the defining instruction executes until the value is killed by the last instruction that uses it. Figure 3.6(b) shows how a value that should be killed and frees a register, stays for the whole computation.



(a) data dependence DAG (b) values lifetime (c) number of live values at each computation step

Figure 3.6: Example comparing number of live values and their lifetime between reversible and irreversible execution

We compute the number of live values at every computation step. Since we allow only one instruction to be executed at each computation step, the curve is either constant or increasing in the reversible computing, as shown in Figures 3.6(c) and Figure 3.7; because we should not get rid of a value if it is not directly replaced by another otherwise we will lose unrecoverable information.

This can define another problem in the reversible computing: how to keep the minimum garbage with a shortest lifetime possible? We can deduce the minimum number of registers required for a reversible execution of a program from a direct execution by computing the number of live values at each step. The idea is to add the positive difference of the number of live values between two successive iterations in each computing step.
If we take Ψ_i the number of live values at the iteration i with i $\in$ [1, N - Ψ_0] and N is the graph size or the number of nodes and Ψ_0 is the number of sources in the graph, q is an integer. This simple code can deduct the number of registers required for the reversible execution.

$$q = \Psi_0;$$
$$if(\Psi_{i+1} - \Psi_i > 0)$$
$$q+ = \Psi_{i+1} - \Psi_i;$$

In the reversible computing, we should not kill a datum if we have no other way to recompute it unlike in conventional computing. However, each increase in the number of live registers means that new values are generated and no one was killed. The possibility to clean garbage data (garbage-free) to free registers, can be done by a backtrack to recompute intermediate values or the input data.

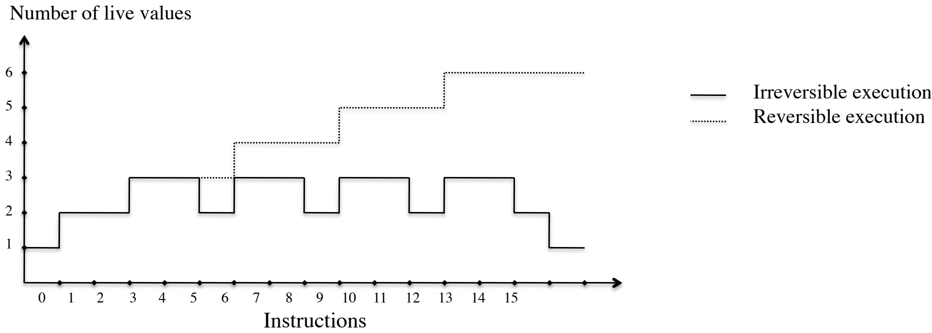


Figure 3.7: Example comparing the number of live values at each computation step for the same code in reversible and irreversible execution (corresponding to the code of Example 2 Figure 3.3)

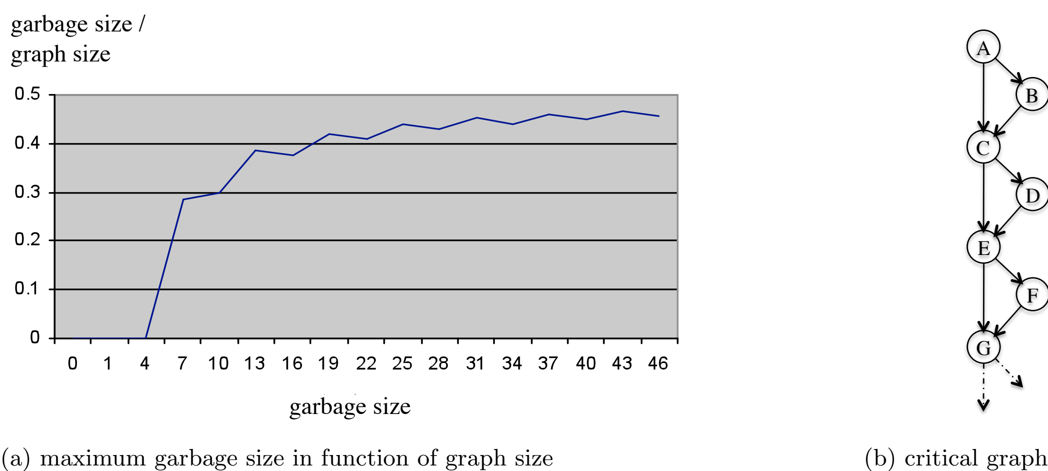
3.4 Experimental Results and Upper-Bound for the Garbage Size

We come back to our algorithm defined in section 3.1. In order to understand more about what this criterion of garbage is, we made two kinds of experiments. First we generated all graphs of some (small) fixed given size (≤ 10) and second we used a randomly generated set of larger graphs (up to 46). For exhaustive or random generation of DAGs we represent DAGs by their adjacency matrix, namely a boolean matrix with rows and columns labeled by graph vertices, with a 1 or 0 in position (v_i, v_j) according to whether v_i and v_j is an edge in the DAG or not. In both cases we computed the garbage as explained in previous sections.

From our experiments of small graphs we exhibited critical graphs for which garbage is maximal. These are the graphs of Figure 3.8(b), where indeed according to Section 3.2 garbage is at least $n/2$ because a partition into elementary paths results in at least $n/2$ paths if the DAG has n nodes.

In [47] where reversibility of automata instead of DAG is considered, they argue that energetic garbage is related to convergences in automata - that result in loss of information. In this case our critical graphs do not have the maximal number of convergences. Maximal number of convergences correspond to DAG with only binary operations. In our case if we want to maximize the garbage size we have to maximize the number of convergences without increasing the number of register requirements in the direct computation. At least two registers are needed to create a convergence in a graph, so we fix the register requirement to two, and we try to create a maximum of convergences. We can observe that each increase of the number of convergence of one corresponds to an increase of two in the graph size, which explains the upper-bound of the garbage size (50% of the graph size).

This $n/2$ upper-bound is actually corroborated by experiments on randomly generated larger graphs. Figure 3.8(a) reports the maximum garbage obtained in percentage of the number of nodes. One can see that this number is never more than 50%. Figure 3.9 reports also an histogram of garbage size for different size of graphs.



(a) maximum garbage size in function of graph size (b) critical graph

Figure 3.8: Upper-bound to the garbage size

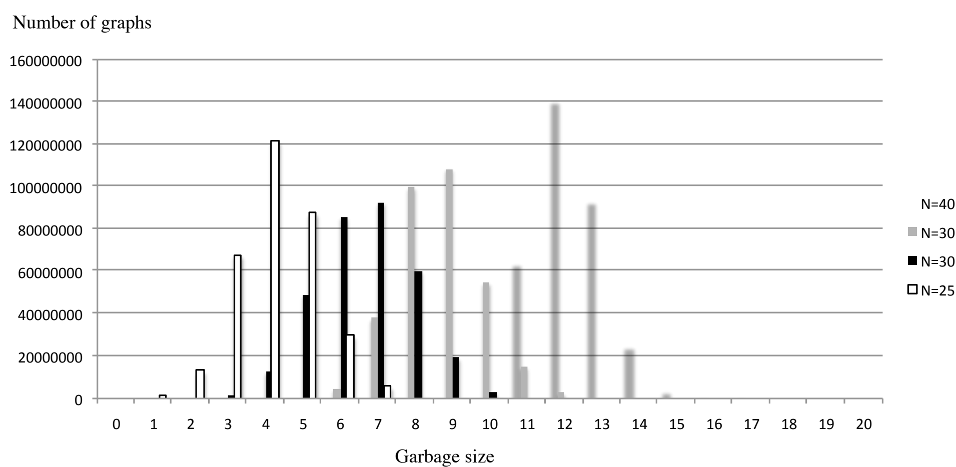


Figure 3.9: Number of graphs randomly generated according to the garbage size

Figure 3.10 shows the percentage number of graphs per garbage size for graph size equals to 40. It shows that more than 90% of generated graphs have a garbage size less than 35% of the graph size.

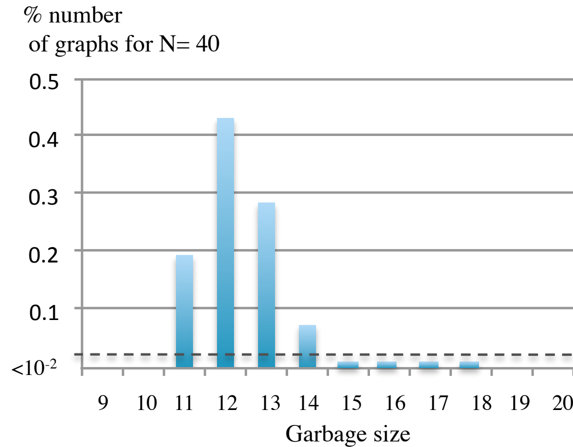


Figure 3.10: upper-bound to the garbage size

3.5 Summary

We have presented an analysis of the number of registers required to make a DAG computing graph reversible. We defined the energetic garbage as the additional number of registers required for computing a forward and backward execution of the graph with respect to the simple forward execution. We gave a lower bound as the size of the decomposition of the graphs into elementary paths and through our experiments, we found that the garbage size does not exceed 50% of the program size - for DAG of unary/binary operations. However, values lifetime is shorter in a direct computation.

It is amazing to observe that in code optimization on current processors, minimizing power consumption amounts most of the time to minimizing the number of memory accesses, cache misses, etc. Therefore it was quite expected that this thermodynamics view of computation leads to trade-off between storage and recomputing, and helps to understand more thoroughly the relationship between variables of a computation graph in terms of mutual information between them. Our next step in this dissertation, is to show how exploiting this information may make the compiler problem of data rematerialization easier to effectively minimize the number of memory access. We will show that recomputing a value in reverse path from output operands is more beneficial than recomputing it from its original input operands which allows to reduce the register demands efficiently. This, is totally contradictory with what we have seen in this chapter where making a program reversible may require more registers.

Part III

Using Reverse Computing to Improve Performance

Chapter 4

Register Allocation Overview

Contents

4.1 Register Allocation Architecture 58
4.1.1 Data Dependency 58
4.1.2 Data Dependency Graph 59
4.1.3 Basic Block 59
4.1.4 Interference Graph 59
4.1.5 Meeting Graph 60
4.1.6 Register Requirements 60
4.1.7 Register Saturation 60
4.1.8 Register Pressure 60
4.1.9 Live Range Splitting 61
4.1.10 Coalescing 61
4.1.11 Register Spilling 61
4.1.12 Register Rematerialization 62
4.2 Different Register Allocation Approaches 62
4.2.1 Register Allocation via Graph Coloring 62
4.2.2 Linear Scan Register Allocation 63
4.2.3 Register Allocation based on Register Reuse Chains 63
4.2.4 Register Allocation via Integer Linear Programming 64
4.3 Register Allocation and Instruction Scheduling 64

While being a very old computer science problem, register allocation is always an important issue in architectures where memory access time and communication time are ever and ever increasing with respect to computing time.

In this chapter, we give some preliminary definitions related to the register allocation and we summarize previous works and various approaches to decrease register pressure in compilers.

4.1 Register Allocation Architecture

4.1.1 Data Dependency

A data dependence in instruction sequence is a situation in which the execution order of two instructions or more must be preserved otherwise the final result will be wrong. This happens in three situations:

- *Read After Write (RAW),* refers to a situation where a write is followed by a read of the same variable. In other words, it occurs when an instruction uses (reads) the resulting value of a previous instruction (write). Such dependence is known as a *true dependence.*

  Example
  I_1. $\mathbf{a} = b + c$
  I_2. $d = \mathbf{a} + 3$
  Here the output value d of instruction I_2 is calculated from the output value a of the instruction I_1. A wrong scheduling that fetches the value of a before being calculated and saved by instruction I_1 will give a wrong result of the value d

- *Write After Read (WAR),* also know as an *antidependence.* This occurs when an instruction writes to a destination that is read by a previous instruction.

  Example
  I_1. $a = \mathbf{b} + c$
  I_2. $\mathbf{b} = d + 3$
  Here I_1 must have read the b variable before I_2 writes it else the result of I_1 would be wrong.

- *Write After Write(WAW)*, it is the situation when two instructions write the same destination, also called *output dependence.*

  Example
  I_1. $\mathbf{a} = b + c$
  I_2. $\mathbf{a} = d + 3$
  Both I_1 and I_2 have the same output. To avoid a dependence violation, the instruction I_1 should be finished before the instruction I_2, otherwise a will have a wrong value after both operations are performed.

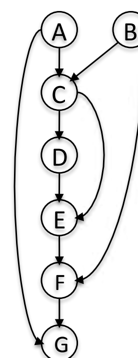
Register renaming eliminates false dependencies (WAR and WAW) between instructions to facilitate program transformation and out-of-order execution to exploit the potential parallelism available in applications. This technique was first introduced by Tomasulo [65] in 1967 for floating point instruction in the IBM 360/91. Tjaden and Flynn [63] were the first to suggest the use of this technique to remove false dependencies. Optimizing compilers do that by using a program representation called Single Static Assignment (SSA) where each value is assigned exactly once.

4.1.2 Data Dependency Graph

Data Dependency Graph (DDG) is a directed graph representing dependencies between instructions which must be satisfied to ensure correct program semantics, where nodes denote results carried by instructions and edges represent data dependencies. In formal term, it is a graph $G = (V, E)$ of set of nodes V and an order relation $E = V \times V$ with $(u,v) \in V$ represents and edge from u to v. For example, given the instruction sequence shown in Figure 4.1(a), the corresponding data dependency graph is shown in Figure 4.1(b)

```
A:  load(a);
B:  load(b);
C:  c = a + b;
D:  d = c * 3;
E:  e = d – c;
F:  f = e / b;
G:  g = f + a;
```

(a)



(b)

Figure 4.1: (a) 3 address code and (b) its corresponding data dependency graph

4.1.3 Basic Block

A basic block is a linear sequence of instruction with a single entry point and a single exit, meaning there is no jump instruction into the code except at the last instruction. Once the first instruction in a basic block is executed, the rest of the instructions will for certain be executed exactly once. Basic blocks are represented as directed acyclic data dependency graph with precedence constraints, and they are usually the basic unit to which compiler optimizations are applied. Each basic block gives rise to partial order on its nodes.

Register allocation can be performed within a basic block, which is called `local register allocation`, or across basic blocks, also called `global register allocation`. In both cases, compilers uses intermediate representations for resolving the problem of register allocation. In this thesis we focus on the local register allocation problem.

4.1.4 Interference Graph

In an interference graph [20] nodes represent variables and an edge connects two nodes if the corresponding variables interfere, so they cannot be assigned to the same register.

Consider the example in Figure 4.2, a is used to compute c so it is live from the load instruction of b to this line. Also a is used to compute g so it still live till computing g. Hence the node a in the interference graph is connected to all other nodes except g. The lifetime of a is shown in the table of Figure 4.2, where the symbol 'x' denotes whether a interferes with variables presented in the rows. For example a is live from the instruction B to the instruction G. b is live from the instruction C to the instruction F.

	a	b	c	d	e	f	g
`A: load(a);`							
`B: load(b);`	x						
`C: c = a + b;`	x	x					
`D: d = c * 3;`	x	x	x				
`E: e = d – c;`	x	x	x	x			
`F: f = e / b;`	x	x			x		
`G: g = f + a;`	x					x	

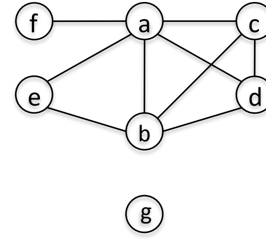


Figure 4.2: Example of interference graph

4.1.5 Meeting Graph

The meeting graph was introduced by Eisenbeis et al. [27] as an alternative to the interference graph for loop register allocation when variables span more than one iteration. It models loop unrolling and register allocation together to keep a pipeline full without spilling. The graph is denoted by $G = (V, E)$ where V is the set of variables or intervals, and E are directed edges such that an edge between two nodes u and v means that the interval of u ends when the interval of v begins. Therefore, their model takes into account the notion of time contrary to the interference graph where time is not present. They presented an optimal algorithm based on graph coloring for allocating variables to a rotating register file, as well as a heuristic for loop variable spilling if the coloring fails, and gave a bound for the unrolling factor which gives an optimal coloring. In this work authors assumed that loop instructions schedule is fixed.

4.1.6 Register Requirements

Register requirements, also called register demands, corresponds to the number of simultaneously live values with respect to the program specification like instruction's sequence order.

4.1.7 Register Saturation

Touati [66] gave an upper-bound of the register requirements for any schedule of the instructions inside a basic block, independently of the functional unit constraints and called it `register saturation (RS)`. The aim is to guarantee a maximum degree of parallelism.

4.1.8 Register Pressure

Register pressure concerns the number of registers required compared to the number of available registers. `High register pressure` is the situation when the number of registers required exceeds the number of available registers which means that more spills and reload are needed. `Low register pressure` usually means that there are enough registers available.

4.1.9 Live Range Splitting

Live-range splitting is a technique to split the live range of variables into smaller subranges by adding copies and renaming variables, each of which can be assigned to a different register. The idea behind is to minimize the interferences between variables to improve results of coloring the interference graph. An example of live-range splitting is shown in Figure 4.3. a interferes with both b and c, but in part 1 and 2 of its live-range a interferes only with one variable. We can make a copy of a (called a' in this example) and rename it in its last use by the name of the copy as shown in Figure 4.3(d). After register renaming a interferes only with b, c interferes with a'. The new interference graph shown in Figure 4.3(d) has a complexity of coloring less than that in Figure 4.3

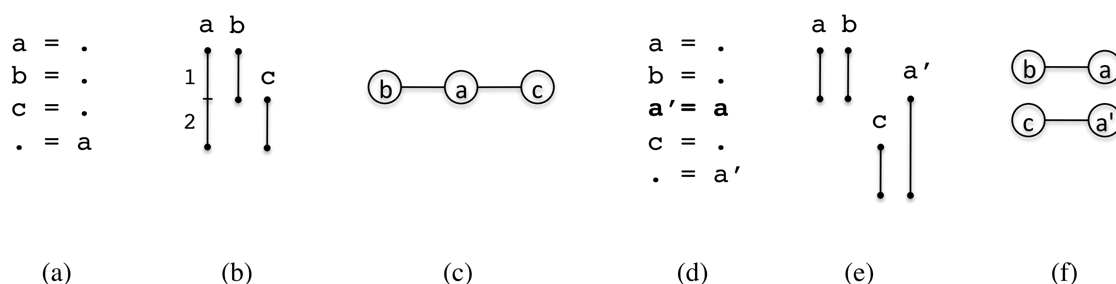


Figure 4.3: (a) Pseudo code. (b) Live-ranges of variables. (c) Interference graph before splitting live-range of a. (d) Pseudo code after splitting. (e) New presentation of live-ranges. (f) New interference graph.

4.1.10 Coalescing

This technique is the inverse of live-range splitting. It aims at eliminating copies if the source and target operands of a copy operation do not interfere. Hence, the variable and its copy can be assigned to the same register. This can be done by, first, renaming copy's occurrences as the variable's name, and then removing the copy operation. In the context of the interference graph, the coalesced node will ensure that the register allocator assigns a variable and its copy to the same register. However the coalesced node will have all interference edges of the source and target nodes being coalesced, which may affect the coloring of the interference graph. Figure 4.3 shows an example of coalescing if we take it in the reverse order of live-range splitting. Consider the pseudo code after splitting a as the original code where a' is the copy of a. The graph coloring algorithm can give different colors to a and a', though the aim is that a and a' share the same register. On the other hand, in the resulting interference graph of coalescing, shown in Figure 4.3(c), a and a' are fused in one node which mean one color. Different approach of coalescing have been proposed [33, 17, 21, 53]. Bouchez et al. [14] studied the NP-completeness of different various of these approaches.

4.1.11 Register Spilling

When there are not enough registers available, the register allocator decides which variables are held in registers and which should be spilled to memory. The register allocator inserts store and load operations to move values between registers and memory since the number

of registers required is larger than the number of available registers. These memory access can slow down the program performance because accessing memory is much slower than accessing a register. Therefore, the aim is to reduce the spill code. Register allocation and spill minimization are NP-complete problems in general [21, 14], therefore, many heuristics have been proposed to solve these problems like graph coloring [20, 17] and linear scan [67, 54]

4.1.12 Register Rematerialization

Rematerialization in register allocation amounts to recomputing values instead of spilling them in memory when registers run out.

One of the first who have efficiently conceived rematerialization were Briggs et al. in [16], their approach focuses on rematerialization in the context of Chaitin's allocator [20], where the problem was already discussed briefly. Punjani in [55] has implemented rematerialization in GCC, the experimental results indicated a gain of 1-6% in code size and 1-4% improvement in execution performance. Zhang in [71] proposed an aggressive rematerialization algorithm to reduce security overhead that uses multiple instruction to recompute a value and extends the live-ranges of depending values deliberately to make the values alive through the point of rematerialization.

Simpson in [60] proposes a register rematerialization pass before the register allocator when a register pressure estimate is considered high. However, it is difficult to correctly approximate register pressure without resolving register allocation pass.

Most of these previous works target rematerialization across basic blocks and ignore it inside basic blocks, and many of rematerialization algorithms are invoked before register allocation [60], which make rematerialization decision less efficient because of the lack of information concerning register requirements, excessive registers and rematerializable values, and which can create extra dependencies to extend live-range of inputs of the rematerialized value, and this can increase register pressure.

In this thesis, we consider using reverse computing to reduce both register pressure and spill code and we argue that there are more opportunities for rematerialization through reverse operations than direct operations only. In the next Chapter, we will give all details of effective rematerialization through reverse computing.

4.2 Different Register Allocation Approaches

Many register allocation algorithms have been proposed in the literature. In this section we briefly introduce the main approaches to solve this problem.

4.2.1 Register Allocation via Graph Coloring

Graph coloring [21, 20] is the most common solution that has been proposed to solve the problem of register allocation. Most works that came later to solve this problem were all focused on optimizing the coloring algorithms [15, 22]. The basic idea of this technique is, given an interference graph of a program, the algorithm attempts to color the resulting interference graph with a number of colors equal to the number of registers available in such a way that two connected nodes may not have the same color, this makes sure that two variables that interfere are not assigned to the same register. If the number of colors is not sufficient to color the graph with respect to the previous propriety, this means

that the number of interfering variables is possibly larger than that of available registers, hence the content of at least one register must be spilled out to memory. The algorithm attempts to re-color the graph from the beginning. The coloring of the graph is called optimal if a minimum of colors is used to color the graph. For example, the interference graph in Figure 4.2 for the 3-address code shown in Figure 4.1(a) requires four colors to be colored. The smallest number of colors required to color a graph is called in graph theory `chromatic number`, this number represents the minimum register requirements in register allocation. However a graph can be colored in different ways using at least k colors where k is the chromatic number.

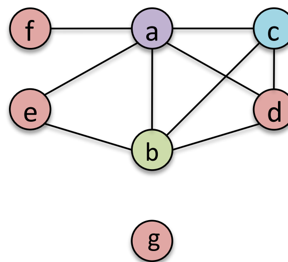


Figure 4.4: Example of graph coloring

Register allocation via coloring of chromatic graph G given a number of available registers k is NP-complete except the case where $k = 1$ and $k = 2$, it is equivalent to decide if G admits a proper node coloring with k colors [32]. However, it is NP-hard to find a lower bound of register requirements which is equivalent to compute the chromatic number [40].

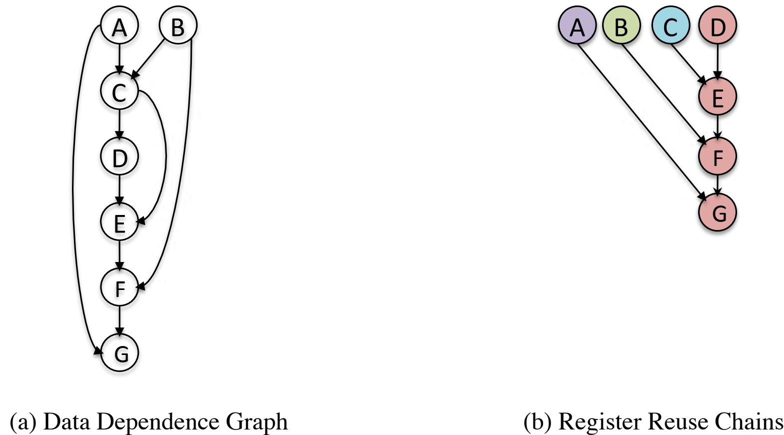
4.2.2 Linear Scan Register Allocation

Poletto [54] proposed a new approach to accelerate the graph coloring algorithm in a single linear-time scan of the variables live-range. The algorithm replaces the live range of each variable by a contiguous interval with long lifetime, and proceed to color these intervals, which makes the algorithm faster than coloring an interference graph, but not optimal. The linear scan algorithm is mostly used in Just in Time (JIT) compilers like Java and LLVM.

4.2.3 Register Allocation based on Register Reuse Chains

Berson [12] proposed registers allocation technique based on register reuse chains where each chain contains values that can share the same register. In this approach, both scheduling and register allocation are solved simultaneously. To avoid additional dependencies, the algorithm starts first by assigning nodes that are not simultaneously live in any possible schedule to the same chain, then reduces the number of chains by assigning dependent nodes which do not overlap their live ranges to the same chain. Only at the end, and if the number of chains is still larger than the number of available registers, the algorithm merges independent chains. Zhang [72] showed that register allocation based on register reuse chains approach requires fewer register on average than the traditional register allocation based on graph-coloring algorithm.

The register reuse chains in Figure 4.5 corresponding to the DDG of the pseudo code in Figure 4.1(a), can be decomposed exactly into four chains. The number of chains presents the register requirement. Detailed description of the algorithm used for the construction of register reuse chains is given in Chapter 5.



(a) Data Dependence Graph (b) Register Reuse Chains

Figure 4.5: Example of register reuse chains

4.2.4 Register Allocation via Integer Linear Programming

Goodwin and Wilken [34] were one of the first who used Integer Linear Programming (ILP) to solve the problem of register allocation and instruction scheduling for regular architectures. The basic idea of this approach is to express the interaction between registers and variables as a sequence of binary decisions, such as, to assign or not assign a given variable to a register at a given time. However, ILP is an attractive technique only for small code segments because compilation time increases exponentially with code size and can take hours to find an optimal solution. This is why most of works that came later [3, 31] were all focused on optimizing the solution time while keeping the result optimal. Fu and Wilken [31] proposed a faster solution for optimal register allocation. Their approach reduces the ILP model complexity by removing unnecessary decision variables from the integer program like spill and deallocation decisions.

4.3 Register Allocation and Instruction Scheduling

If the number of simultaneously live values exceeds the number of available physical registers, selected values must be spilled and reloaded from memory, which can reduce overall performance. On the other hand, an aggressive optimization in order to limit register pressure may lead to underutilized registers, resulting in suboptimal performance. This is why, instruction scheduling and register allocation are important phases in a high-performance compiler, and the ordering of these two phases can affect the program's performance.

In most research efforts, instruction scheduling and register allocation are studied separately. However, these optimizations influence each other significantly. The usefulness of register reuse chains comes from that. Berson and all [12] make a promising contribution by proposing registers allocation based on register reuse chains where scheduling and register allocation are solved simultaneously. Register allocation and instruction scheduling can be also solved simultaneously using integer linear programming. Many formulations have been proposed for the problem.

Chapter 5

Using Reverse Computing to Decrease Spill Code

Contents

5.1 Problem Statement: Register Allocation 67
5.1.1 Recomputing vs. Storage 67
5.1.2 Aggressive Register Reuse 69
5.2 Rematerialization rules and guidelines 69
5.2.1 Building Register Reuse Chains 70
5.2.2 Detecting Excessive Registers 81
5.2.3 Discovering Rematerializable Values 81
5.2.4 Graph Transformation 84
5.2.5 More Opportunities for Reverse Computing than for Direct Computing 85
5.3 Experimental Results 85
5.3.1 Lattice QCD Computation 86
5.3.2 Register Requirements 87
5.3.3 Spill Costs 88
5.3.4 Run-Time Performance 89
5.3.5 Inverse Precision 90
5.4 Summary 90

Over the years, processor cycle time is decreasing much faster than memory access times, and the architectural design of processors has improved with the development of pipelining and multiple instruction issue. These factors have influenced the increasing technological gap between processor speed and the speed of the memory hierarchy. Figure 5.1 shows processor and memory speedup during the years, and the processor-memory performance gap.

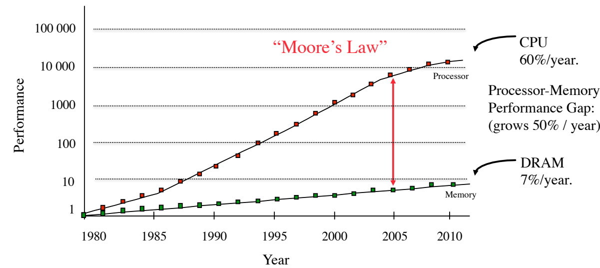


Figure 5.1: Memory Access vs CPU Speed

But, no matter how fast a CPU is, it has to request data from different levels of some memory hierarchy. Hence CPU speed is hindered by the slow read/write memory speed. A computer with insufficient storage space will have to rely on a higher level of memory for this data, which would make the processor speed essentially irrelevant, forcing programs to run at the speed of this memory level. This is why memory together with communication optimizations are today the most important room for performance.

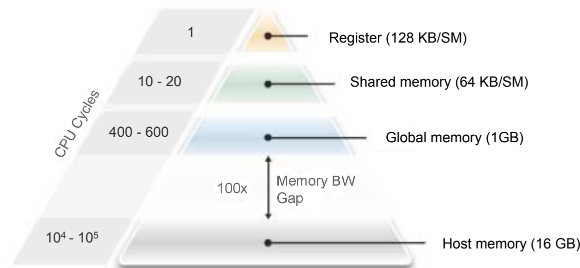


Figure 5.2: GPU architecture: Memory Access vs CPU Speed

The purpose of this chapter is to describe how to maximize (increase) the use of the processing units to overlap memory transfer by computation, and how to best set up data items to use the memory effectively, by using as much fast memory and as little slow-access memory as possible.

We test the effectiveness of the solution we propose on the LQCD application. LQCD program typically reads sequentially through a large 4D lattice of data, modifies the data, and writes out to another large lattice, which will be memory bound, so anything that optimizes the flow of data is highly desirable, typically load data once, and need not touch them again. To ensure this, we exploit *recomputing* instead of spilling in the kernel's code and since a value can be recomputed locally no need to re-load it from memory.

5.1 Problem Statement: Register Allocation

We revisit register allocation issues from the reversible computing angle.

Since LQCD program is memory demanding it is important that the least number of data are stored in memory or equivalently the largest number of useful data. That means first that intermediate values be kept in lowest levels of memory - registers if possible - and therefore that they have short lifetimes or can be *recomputed from other values.* Second that the vertical memory hierarchy is stressed as little as possible: we have to avoid spill operations that store intermediate value in the memory and reload it on demand. This means that we have to *minimize spill code.* Hence, in this program, we can see that register allocation is a still delicate and critical issue that must in no ways be left aside as it is the basic bottleneck that conditions the whole performance.

Reversible computing has a lot to do with the classical issue of trade-off between data storage and data recomputing. In reversible computing no information is ever lost, every data value can always be retrieved from any point in the program.

In register allocation one can use rematerialization instead of spilling, meaning that we recompute some value v instead of keeping it live. Recomputation is performed from values still stored in registers, recomputation is done in the same way as specified in the program. But there is a part of information on v carried by other values w that *have been computed* from v. Hence this gives new opportunities for recovering the v value: undoing the computation from the w values, or in other words, reversely computing v.

Therefore the question that we address in this chapter is *whether rematerialization by reverse computing – reverse rematerialization – can help improving register allocation.*

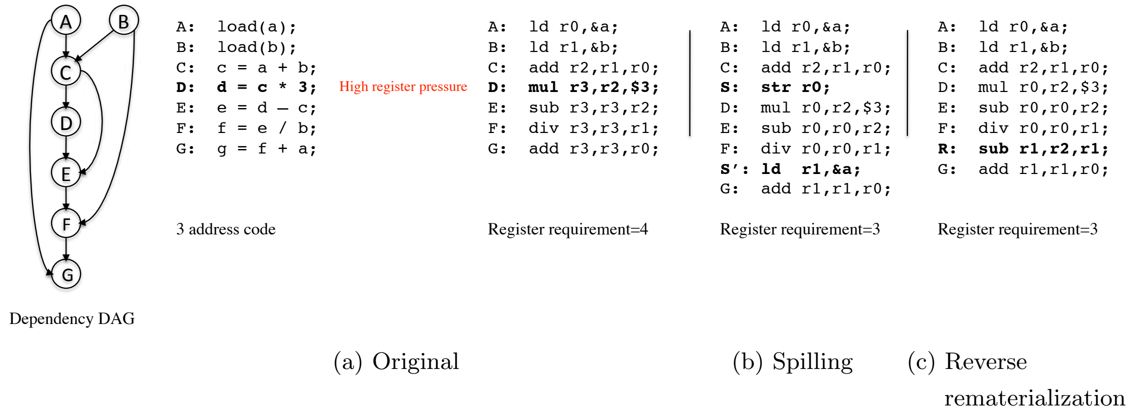
We develop a heuristic for rematerialization-based register allocation through reverse computing and demonstrate important gains over a kernel of the LQCD computation.

As an illustration, consider the code segment shown in Figure 5.3(a) with its corresponding pseudo-assembly code and dependence graph in which each node corresponds to a statement in the code segment. This original pseudo code requires four registers. Figure 5.3(b) shows the same code that returns the same result as the code in Figure 5.3(a) but with an additional load/store operation. This code requires three registers. Figure 5.3(c) shows always the same code that returns the same result but with an additional operation. It shows how a reverse computation of A from B and C could minimize register pressure and avoid load/store operations. Thus, four registers are required in the forward computation, three with an additional reverse operation without any additional load/store operation.

5.1.1 Recomputing vs. Storage

In this section, we present our approach about recomputing to minimize load/store operations when we achieve high register pressure. We consider a DAG of operations, typically the Data Dependency Graph of instructions within a basic block, see part (a) of Figure 5.3. Nodes of the graph are instructions denoted by the name of the variables carrying the result. This makes sense as two different nodes need to be treated as two different variables. We make the important hypothesis that they can be made `reversible`.

Most problems of spill code minimization are NP-complete [14]. Here we have another degree of freedom. We can recompute a value no matter the number of instructions required to recompute it, and as we make the condition that all operations can be made reversible, a value can be recomputed either from source or result operands.

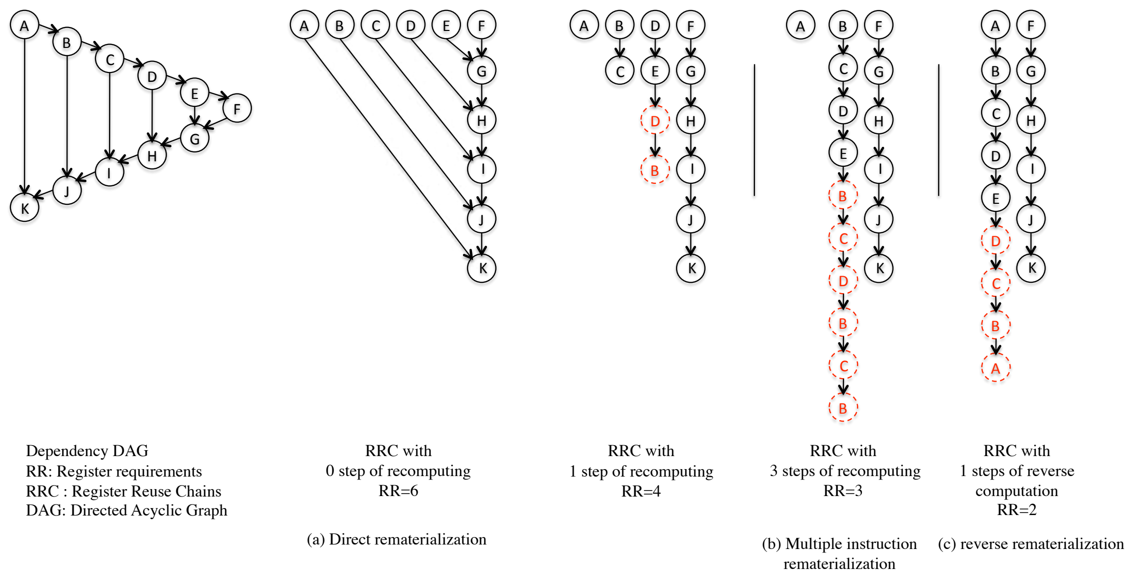


(a) Original (b) Spilling (c) Reverse rematerialization

Figure 5.3: Reducing register pressure using reverse rematerialization.

An example of direct (forward) rematerialization is shown in Figure 5.4(a). In the example six registers are required to perform the DAG according to the initial reuse DAG drawn in (a). Live ranges of A, B, C, D, E and F overlap. But since B and D are alive during the computation of all outputs of C and E respectively, and since C can be directly computed from B, and E from D, we can choose to let C reuse the register of B, and let E reuse the register of D, and recompute later D from C and B from A before computing H and J respectively. This is drawn in the right part of figure 5.4(a).

We can even more increase register reuse by considering sequences with *more than one instruction* for rematerialization (figure 5.4(b)). A is alive during all the computation. Thus we can rematerialize B, C, D from A. In this example register requirement is only 3 with 6 additional operations and this could remove 3 spill operations if we had only 3 available registers. As a matter of fact it makes sense to use multiple instructions only as far as the execution of the sequence remains negligible with respect to the latency of memory access. In this work, we first consider that computation is free so we do not consider this trade-off.



(a) Direct rematerialization (b) Multiple instruction rematerialization (c) reverse rematerialization

Figure 5.4: Recomputing Vs Storage.

Direct rematerialization aims at avoiding spilling by recomputation. However direct rematerialization is limited because values needed for recomputation have to stay alive and recomputation may take multiple instructions. In contrast considering reverse computing makes rematerialization more attractive because it can reduce both register pressure and number of rematerialization instructions. In Figure 5.4(c) only 2 registers are required and each recomputed value rematerialized by one instruction which would avoid 4 spills for the whole DAG in the case of 2 available registers. The register pressure is high after computing C, D, E and F causing A, B, C and D to be spilled. A simple way to avoid inserting four load operations before computing H, I, J and K is to rematerialize them from their outputs with one instruction by recomputing D from E, C from D, B from C, and A from B. For minimizing spill code we increase register reuse by reducing values lifetime.

We suggest a hybrid algorithm that both considers direct and reverse computing. The idea is to check at each point when a extra register is needed if a previous value is rematerializable either in a direct or reverse way in order to consider reusing the register where it is stored.

5.1.2 Aggressive Register Reuse

The idea of aggressive register reuse is to enforce register reuse between direct dependent values. Based on the generation of register reuse chains [12] we propose to consider register rematerialization after register allocation. Register requirement is determined from the reuse DAG which indicates which instruction can use a register used by a previous instruction. For the reuse DAG we use the algorithm proposed by Yukong in [72]. But we use a more aggressive reuse relation by allowing reuse not only for killed variables but also for possibly rematerializable values either directly from the source operands or reversibly from the result operands. This is likely to promote register reuse between dependent values and hence reduce register pressure.

5.2 Rematerialization rules and guidelines

Rematerialization should be done after register allocation for two reasons:

- Before register allocation there is no information about actual register pressure, register requirement and excessive register demands, which makes rematerialization decision very difficult to take.
- Rematerialization decision before register allocation would create extra dependencies and possibly extend live-range of inputs of the rematerialized value, and this could increase register pressure.

All information regarding excessive registers, rematerializable values, etc. can be extracted from register reuse chains. Berson [12] showed that excessive register demands can be better determined using register reuse DAG, where instruction scheduling and register allocation can be solved simultaneously. Because instruction scheduling and register allocation are mutually dependent and it is better to manage them in a common framework, which is the aim of the register reuse chain. Zhang [72] has also shown that register allocation based on a register reuse chains approach requires fewer registers on average than traditional register allocation algorithms based on graph-coloring algorithms.

The register rematerialization can be decomposed into four passes:

- a) Building register reuse chains.
- b) Detecting excessive registers.
- c) Discovering rematerializable values.
- d) Graph transformation.

5.2.1 Building Register Reuse Chains

The first phase decomposes the input data dependency graph $G = (V, E)$ where V is the set of nodes and E the set of edges in G into reuse chains. Each chain contains values that can share the same register. We start by considering register reuse between dependent values and then between independent values. For each node we identify possible reuse nodes through the relation "*can reuse*". If the live-range of a variable does not overlap the range of another variable they can share the same register. Formally:

$$(u, v) \in V^2,\ u \text{ can reuse } v \text{ iff } \mathit{output}(v) \cap \mathit{dependent}(u) = \phi$$

where $\mathit{output}(v)$ is the set of direct dependent nodes of v denoted by a direct edge from v in the graph G, and $\mathit{dependent}(v)$ is the set of all descendant nodes of v. In other words, it is the set of all nodes such that there is a path from v to these nodes. An example of the relation "*can reuse*" is shown in Figure 5.5.

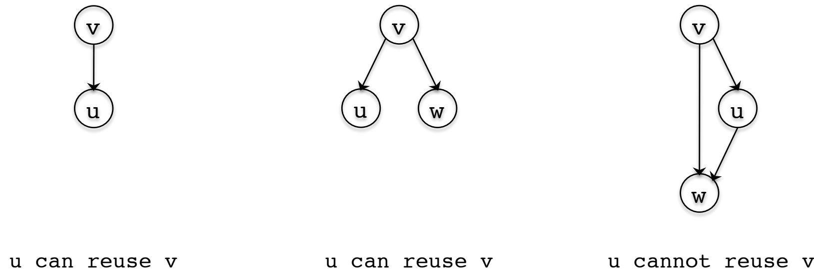


Figure 5.5: Example of the relation "*can reuse*"

In the following we denote also $\mathit{input}(v)$ the set of direct connected nodes by outgoing edge to v and $\mathit{independent}(v)$ the set all nodes such that there is no path between v and these nodes.

5.2.1.1 Register Reuse between Dependent Values

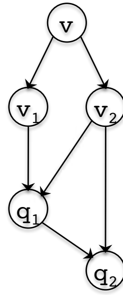
This is an iterative algorithm. In order to create as few as possible additional dependencies we start by first defining register reuse between dependent values. The initial reuse node has to be one Earliest Ultimate Killing *(EUK)* node. *EUK* of a node is the earliest node where the lifetime of the values residing in the node is guaranteed to be over [72].

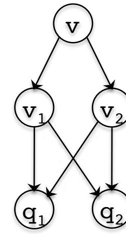
k is killer of v iff $\forall u \in output(v),\ k$ depends on u

and

q is EUK node of v if q is killer of $v\ \wedge\ \forall k$ killer of $v, k \neq q\ k$ depends on $q \vee q$ is independent of k

means that when k or q are executed we are sure that all their inputs which correspond to dependent nodes of v are already computed and as a result v is dead. Figure 5.6 illustrates killing and earliest ultimate killing nodes.





q_1 and q_2 are two killing nodes of v
q_1 is the EUK node of v

q_1 and q_2 are two EUK nodes of v

Figure 5.6: Example of killing and earliest ultimate killing nodes

Initial Register Reuse Chains
As it is shown in Algorithm 1, to build the initial register reuse chains we associate to each node its EUK node (determined by the Algorithm 2 and 3) if it exists. This allows generating register reuse chains without any additional dependency. A node v can have more than one EUK node if $EUK(v) \notin output(v)$. If two EUK nodes of v are independent we choose one of them and this does not effect the initial register reuse chains. A node k can be the EUK of several nodes. If it is the case we choose the nearest node to the killer following the function *NearestToKiller* defined by the Algorithm 6, and we set the other nodes as free (without reuse nodes).

Algorithm 1: InitialRegisterReuseChain(G)

```
/*The function generates the initial register reuse chains based on the EUK nodes*/
forall the i ∈ G do
    GetEUK(i,G)
forall the i ∈ G do
    if EUK(i)≠ φ then
        Reuse(i)←EUK(i)
    else
        Reuse(i)←i
forall the (i, j) ∈ G^2 do
    if Reuse(i) = Reuse(j) && NearestToKiller(i,j)=j then
        Reuse(i)←i
        forall the k ∈ G do
            if IsKiller(k,i)=1 && k≠Reuse(j) then
                if Reuse(i)=i || Reuse(i) depends on k then
                    Reuse(i)←k
return Reuse
```

Algorithm 2: GetEUK(i, G)

```
/* i is a node in the Graph G */
/* The function returns the earliest ultimately killing node of i */
forall the j ∈ G do
    if IsKiller(j,i,G)=1 then
        if j ∈ output(i) then
            EUK(i)←j
        else
            if EUK(i) = φ || EUK(i) depends on j then
                EUK(i)←j
return EUK(i)
```

Algorithm 3: IsKiller(i, j, G)

```
/* i and j are two nodes in the Graph G */
/* The function checks if node i is a killer of node j */

forall the k ∈ G do
    if k ∈ output(j) && i ∉ dependent(k) then
        return 0
if output(j)= φ then
    return 0
else
    return 1
```

Example Consider the dependence graph shown in Figure 5.7. Node A has three direct dependent nodes, B, C and D. Node K is the only node that depends on all three nodes. Thus, K is a killer of A, and as K is the only killing node of A, it is also its *EUK* node. Similarly, D has two outputs, G and H. J and K are descendant of both G and H. Thus J and K are killers of D. Since K depends on J, node J is the *EUK* node of D. Node I has a single dependent node K. Thus K is also the *EUK* node of I. Table 5.1 shows killing and ultimately earliest killing nodes of all nodes of the graph shown in Figure 5.7.

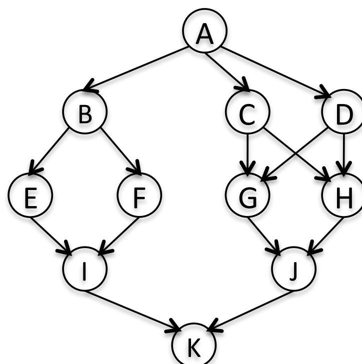


Figure 5.7: Initial Data Dependency Graph

Node	A	B	C	D	E	F	G	H	I	J
Killer nodes	K	I, K	J, K	J, K	I, K	I, K	J, K	J, K	K	K
EUK node	K	I	J	J	I	I	J	J	K	K

Table 5.1: List of killer and EUK nodes

To build the initial register reuse chains and once killing and earliest ultimate killing nodes are found, we associate each *EUK* node to only one node. Figure 5.8 shows the initial register reuse chains derived from the dependence graph by the algorithms 1, 2 and 3. If two or more nodes have the same *EUK* node like A, I and J, we call the function *NearestToKiller* to choose the closest node among them to their common *EUK* node, which is K in this example. I and J are at the same distance from K but closer than A. Hence, we choose K as a reuse node of I, just because I was visited before J. More details about the function *NearestToKiller* to choose a schedule that might require fewer registers are given in the subsection `Distance between Nodes`.

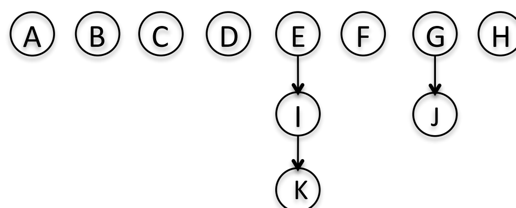


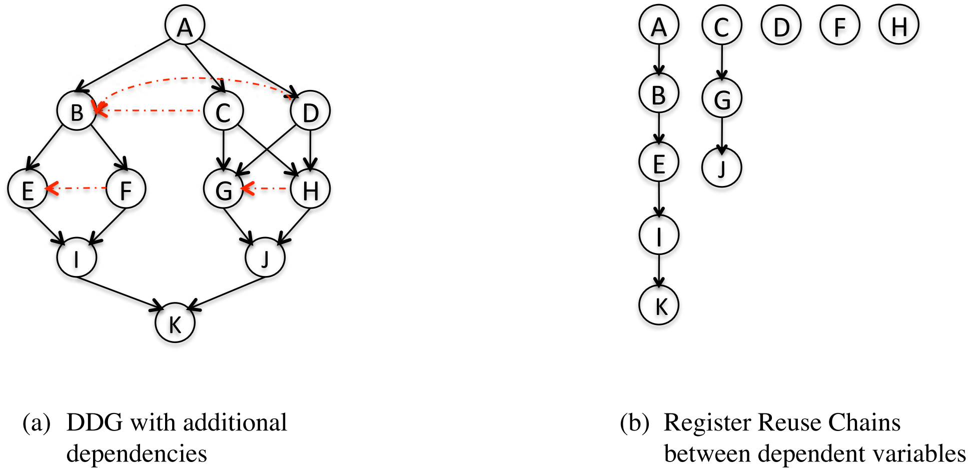
Figure 5.8: Initial Register Reuse Chains

The register reuse algorithm continues visiting all nodes that can reuse a node v. If v has no reuse node we choose the first visited node otherwise we compare the visited

node with the current reuse node. The criteria of comparison are: the live-range and dependencies between reuse nodes. Hence, if the *EUK* of node v does not depend directly on v, we choose a reuse node from the set of outputs of v. If more than one node can reuse v, we compare their live-range ends by using the Algorithm 5, and we choose the latest one. If all outputs of v already use other registers, we search a closer reuse node from the set of dependent nodes of v, and if two reuse nodes have the same distance from v and are independent, we compare their live-range ends and we choose the first done.

Starting graph reduction by choosing reuse nodes from dependent nodes creates fewer dependencies because reuse nodes require always dependence from reused nodes.

Example Consider the same data dependency graph as in the previous example. The algorithm starts with the input node of the graph which is the node A, and chooses B as reuse node. It creates new dependencies from the other output nodes of A, C and D to B. Then, it visits the next node B and calls the function *CanReuse* to checks if E and F can reuse B. As E and F are at the same distance from their common *EUK* node, the algorithm chooses the first visited node E and it adds an additional dependency from F to E. The node I has always a reuse node K found from the previous step of the algorithm. Similarly for C, the algorithm is called and chooses G as a reuse node and adds a new dependency from H to G by following rules. Therefore, the algorithm stops by visiting all nodes to assign them reuse nodes, which are not assigned, from their dependent nodes. The new data dependency graph and its corresponding register reuse chains are shown in Figure 5.9. The additional dependencies are shown by disconnected red edges.



(a) DDG with additional dependencies

(b) Register Reuse Chains between dependent variables

Figure 5.9: Building register reuse chains

If the number of register reuse chains is still greater than the number of available registers we proceed to reduce the number of chains by defining new reuse nodes from the set of independent nodes.

5.2.1.2 Register Reuse between Independent Values

Independent values correspond to nodes for which no dependency path between both exists. In certain cases, they correspond to sequences of nodes that can share the same register. The function *CannotReuse* checks if there is no constraint that prevents reusing a node. If a node is dead and has no reuse node, it can be reused by an independent node if the latter does not reuse any register and if there is no dependency violation. If there

are two or more independent nodes of v that can reuse it, we compare the live range ends of these nodes with the function *GetLastLiveRange* and we choose the node which finishes first.

Example Consider the same previous example. After the first phase of register reuse algorithm which considers only dependent nodes, H still has no reuse node and F is not assigned to any node, as it is shown in the part (b) of Figure 5.9. Since H and F are independent nodes, the function *CannotReuse* is called to check if F can reuse H without any dependence violation. Thus, F is assigned to H and a new dependency is added in the DDG from J, the output of H, to F. The additional dependency is shown by a green disconnected edge. Finally, all nodes are visited and the reduction algorithm is stopped. The final register reuse chains are shown in Figure 5.10(b).

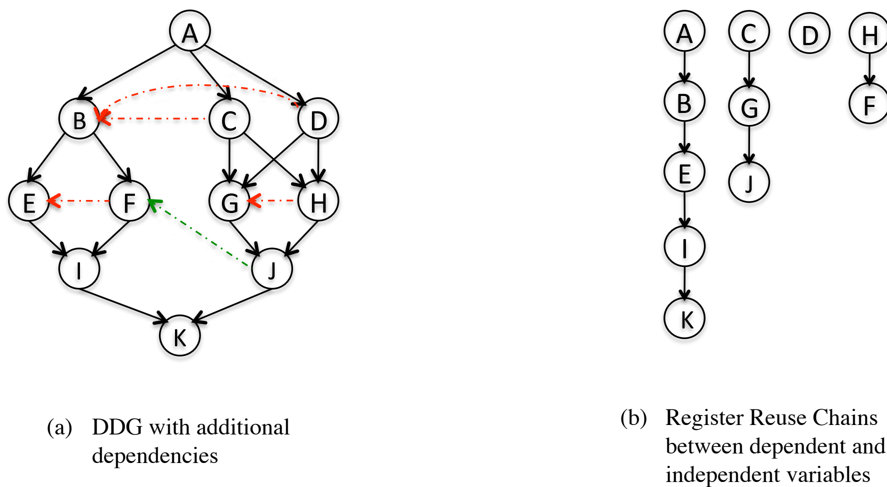


Figure 5.10: Building Register Reuse Chains

During the reduction process some dependencies have been added to minimize the register pressure and they now have to be considered for the scheduling. Once the number of chains is less than the number of available registers we stop the reduction phase. The register requirements is the number of chains. Between dependent nodes the relation *CanReuse* - defined in Algorithm 11, checks if a node can reuse an other node without any dependency violation. Between independent nodes, the relation *CannotReuse*, defined in Algorithm 12, checks if there is a constraint preventing register reuse. This iteratively adds dependencies one by one every time we reduce the initial register reuse chains. The register reuse chains are built thanks to the general algorithm of register reuse shown in Algorithm 4.

Algorithm 4: Algorithm of register reuse

```
notation: liverange(p) ≻ liverange(q) if p stays live after the last use of q
if EUK(v) ∈ output(v) then
    Reuse(v) ← EUK(v)
if (EUK(v) ∉ output(v)) ∨ (EUK(v) = φ) then
    get P | ∀p ∈ P, p ∈ output(v) ∧ p can reuse v
    if |P| ≥ 1 then
        Reuse(v) ← GetLastLiverange(P)
    else
        get Q/∀q ∈ Q, q ∈ dependent(v) ∧ q can reuse v
        if (|Q| ≥ 1) then
            ∀(a, b) ∈ Q^2
            if (a depends on b) then
                Reuse(v) ← b
            else
                if (b depends on a) then
                    Reuse(v) ← a
                else
                    if liverange(a) ≻ liverange(b) then
                        Reuse(v) ← b
                    else
                        Reuse(v) ← a
        else
            get R/∀r ∈ R, r ∈ independent(v) ∧ r can reuse v
            if (|R| ≥ 1) then
                ∀(a, b) ∈ R^2
                if (a depends on b) then
                    Reuse(v) ← b
                else
                    if (b depends on a) then
                        Reuse(v) ← a
                    else
                        if (liverange(a) ≻ liverange(b)) then
                            Reuse(v) ← b
                        else
                            Reuse(v) ← a
```

Distance between Nodes

The notion of distance between nodes is used to favor schedules that are likely to fewer less registers. The function *GetLastLiveRange* defined in Algorithm 5 is used to compare the live range ends of variables, by computing the distance from nodes and their common ultimate killing node if it exists (with the function *NearestToKiller* defined in Algorithm 6), otherwise by computing the distance from their last common dependent node (with the function *NearestToLastCommonNode* defined in Algorithm 10). The aim is to reduce the lifetime of each value, which helps prevent that too many live ranges overlap. The value lifetime is defined as the length of time for which a value is held on a register from its defining instruction to killing instruction.

In the set of nodes that can reuse v, if these nodes are independent or indirectly dependent of v, we favor the node with a longest distance from their *EUK* common node if it exists, otherwise we favor the node with the longest distance from their last common dependent node. This corresponds (with high probability) to the node whose live range finishes first. If these nodes are directly dependent of v, we favor the node with the shortest distance from the common *EUK* node if it exists, otherwise from their last common node. The value corresponding to this node will be the last killed, so we schedule its defining instruction after the defining instructions of all other outputs of v to reduce its lifetime.

Example 'A' has three outputs, 'B', 'C' and 'D'. All of them can reuse it. Choosing 'D' as a reuse node of 'A' creates dependencies from 'B' and 'C' to 'D' forcing them to be computed before 'D'. As Figure 5.11(b) shows, 'B' and 'C' stay live during the computing of 'D', 'E' and 'F'. Thus, the maximum number of simultaneously live values is four.

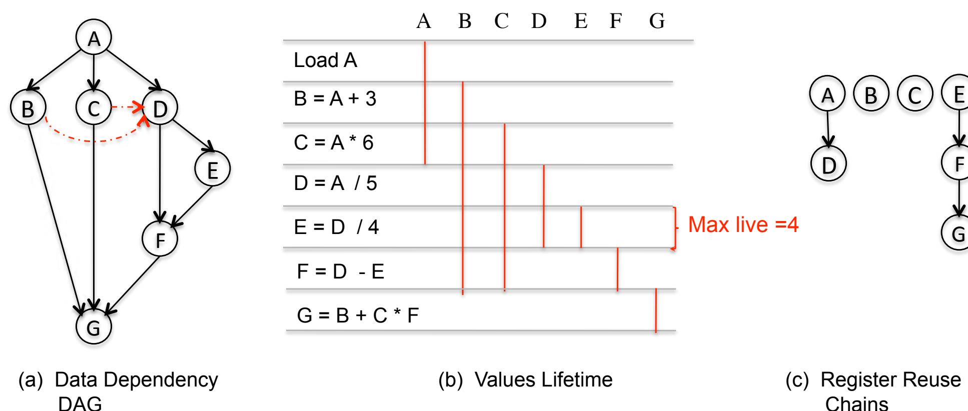


Figure 5.11: Live ranges and distance between nodes

However, the earliest common killer of the three nodes is 'G'. The distance between 'B' and 'G', or 'C and 'G' is 1. The shortest distance between 'D' and 'G' is 2. So, we choose 'C' to reuse 'A', knowing that 'D' dies just after computing 'F', making its register free, so that 'B' can reuse it. Hence, the number of simultaneously live values is three.

Some detailed algorithms of functions used to build our tool of generation of register reuse chains are shown below.

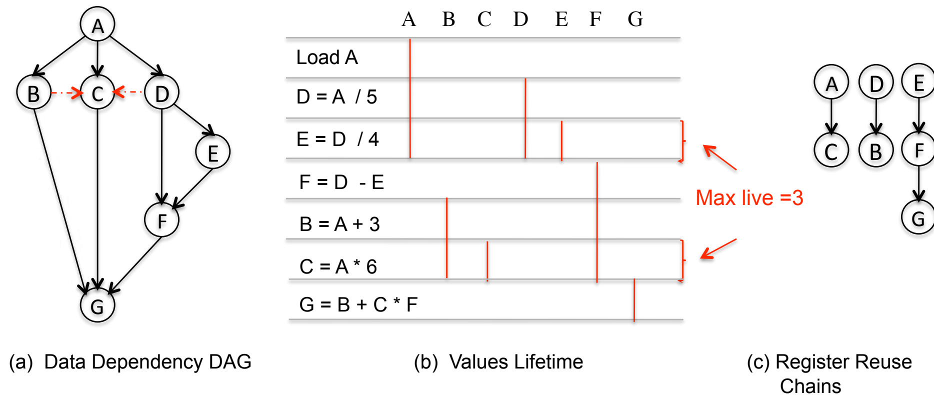


Figure 5.12: Live ranges and distance between nodes

Algorithm 5: GetLastLiveRange(node i, node j, Graph G)

```
/* i and j are two nodes in the Graph G                                    */
/* The function returns the value with highest probability that it will stay live
after the last use of the other value.                                     */
if IsKiller(i,j)=i then
    return i
else
    if IsKiller(j,i)=j then
        return j
    else
        if GetCommonKiller(i,j)≠ φ then
            return NearestToKiller(i,j)
        else
            if GetLastCommonNode(i,j)≠ φ then
                return NearestToLastCommonNode(i,j)
```

Algorithm 6: NearestToKiller(node i, node j)

```
/* i, j and k are three nodes in the Graph G                               */
/* The function compares the shortest distance from the common killing node of i
and j, and i, and the shortest distance from the common killing node of i and j, and
j, and returns the smallest one                                            */
node k;
int d1,d2;
k=GetCommonKiller(i,j, Graph G);
if k≠ φ then
    d1=GetShortestDistance(k,i);
    d2=GetShortestDistance(k,j);
    if d1<d2 then
        return i;
    else
        return j;
```

Algorithm 7: GetCommonKiller(i, j, Graph G)

```
/* i and j are two nodes in the Graph G                                   */
/* The function returns the closest common killer of i and j              */
for k ∈ G do
    if (Iskiller(k,i)=1) && (Iskiller(k,j)=1) then
        if depends(killer,k)=1 || (killer=ϕ) then
            killer←k
return killer
```

Algorithm 8: GetShortestDistance(node i, node j, Graph G, int ldist)

```
/* i and j are two nodes in the Graph G                                   */
/* The function returns the shortest distance between i and j if i depends on j and
-1 if they i is independent of j                                          */
int gdist=size(G);
for k ∈ G do
    if i ∈ output(k) then
        ldist++;
        if k == j then
            if ldist < gdist then
                gdist=ldist;
        else
            if k ∉ source(G) then
                GetShortestDistance(k, j, ldist);
        ldist−−;
if gdist == size(G) then
    return -1;
else
    return gdist;
```

Algorithm 9: GetLastCommonNode(i, j, Graph G)

```
/* i and j are two nodes in the Graph G                                   */
/* The function returns the closest common node of i and j                */
for k ∈ G do
    if (depends(k,i)=1) && (depends(k,j)=1) then
        if (depends(k,common)=1) || (common=ϕ) then
            common←k;
return common;
```

Algorithm 10: NearestToLastCommonNode(node i, node j, Graph G)

```
/* i, j and k are three nodes in the Graph G */
/* The function compares the shortest distance from the common node of i and k,
and k, and the shortest distance from the common node of j and k, and k, and
returns the smallest one */
node c;
int d1,d2;
c=GetCommonNode(i,j);
d1=GetShortestDistance(c,i);
d2=GetShortestDistance(c,j);
if d1<d2 then
    return i;
else
    return j;
```

Algorithm 11: CanReuse(node i, node j, Graph G)

```
/* i and j are two nodes in the Graph G */
/* Between dependent nodes, the function checks if a node can reuse an other node
without any dependence violation. */
for k ∈ G do
    if k ∈ output(j) && k∈ dependent(i) then
        return 0;
if i ∈ dependent(j) then
    return 1;
else
    return 0;
```

Algorithm 12: CannotReuse(node i, node j, Graph G)

```
/* i and j are two nodes in the Graph G */
/* Between independent nodes, the function checks if a node can reuse an other
node without any dependence violation. */
if j ∈ dependent(i) || i ∈ dependent(j) then
    return 1;
else
    for k ∈ G do
        if k ∈ output(j) && k∈ dependent(i) then
            return 1;
return 0;
```

5.2.2 Detecting Excessive Registers

Excessive register demands arise when the number of values simultaneously live exceeds the number of available registers or the target number that we are seeking for. The register reuse chains identify excessive sets that represent values whose scheduling requires more resources than available. Since scheduling and register allocation are solved simultaneously, the order in which values are computed is known and as a result excessive demands for registers can be determined. The excessive sets are then used to drive reduction of the excessive demands for registers and rematerialization is used to reduce register demands.

5.2.3 Discovering Rematerializable Values

In general, a value stays live after being used a first time because it is used more than once. While it is not used by all direct dependent operations it is live and the register of this value cannot be reused. A value v might be rematerializable by a direct operation from source operands, or by a reverse operation from the result and the rest of operands of the operation.

v is directly rematerializable iff $\forall p \in \mathit{input}(v)$ p is live.

v is reversibly rematerializable iff $\exists q \in \mathit{output}(v)$ $\forall p \in \mathit{input}(q)$ s.t. $p \neq v \Rightarrow p$ and q are live.

We call $\mathit{R\text{-}input}(v)$ the set of sets of operands from which v can likely be recomputed either by a direct or reverse operation.

v is rematerializable by multiple instructions iff $\exists S \subset \mathit{R\text{-}input}(v)$ $\forall p \in S \;\Rightarrow\; p$ is live or p is rematerializable

5.2.3.1 Rematerialization Decision

In general, an early rematerialization decision (which variable must be used and rematerialized after) before register allocation is definitive and will not be undone later. It might increase register pressure by extending lifetime of inputs of the rematerialized value, which is the length of the longest path from its definition to its last use. The rematerialization decision is efficiently controlled in our approach. It is performed after register allocation and it manages at the same time both rematerializable and excessive nodes of the graph. We first find all rematerializable values and compare their live-ranges with the ones of excessive variables. For each rematerializable value we choose the appropriate value that can reuse it and does not prevent its recomputing. Based on register reuse chains we give conditions for a value u to reuse the register of value v in the case of high register pressure and recompute v later when we achieve low register pressure, knowing that live-ranges of u and v overlap.

$$\exists x \in ExcessiveRegisters(G), v \text{ is rematerializable so } x \text{ can reuse } v \text{ iff}$$
$$\exists S \subset R\text{-}input(v),\ \forall p \in S,\ reuse(p) \text{ is defined after } x \text{ or } reuse(p) = \phi$$

If at the next use of v we still have high register pressure, we add the condition

$$output(x) \cap output(v) = \phi$$

In technical term, we compare the reuse node of p if it exists with x. If $reuse(p)$ is defined after the defining instruction of x, v is considered as a rematerializable value. This happens when x does not depend on $reuse(p)$, so we schedule instructions to compute $reuse(p)$ after x. Otherwise, if p has no reuse node and its last use is before the next use of v, we extend live-range of p up to the next *output(x)*.

In Figure 5.13 and 5.14, we show values lifetime, just before and after rematerialization decision of the DDG shown in Figure 5.7. H is the excessive node, A is rematerializable as D is live and *reuse*(D)=ϕ. Thus, H can reuse A which reduces the number of simultaneously live values from four to three. Once we reach a low register pressure after computing J we rematerialize A from D to recompute B. The final register allocation chains are shown in Figure 5.15(a). The chain 'A - H - F' share the same register, and the chain 'B - E - I - K' still share the same register but with the new A after rematerialization.

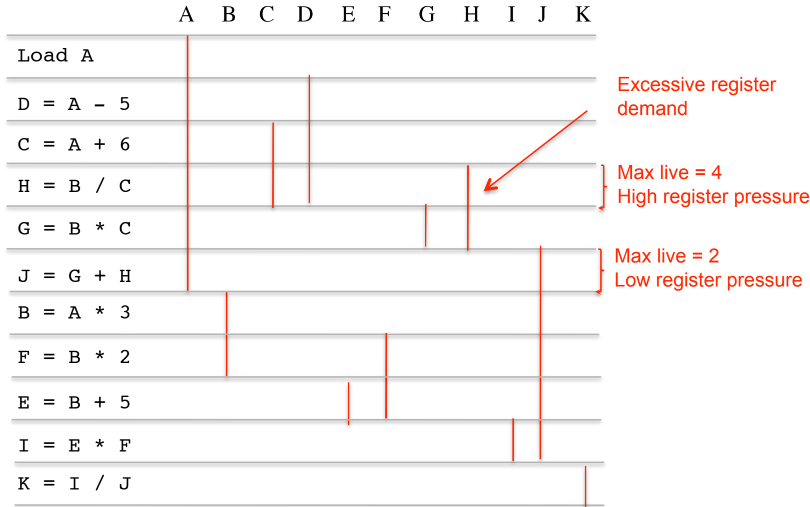


Figure 5.13: Values lifetime before rematerialization

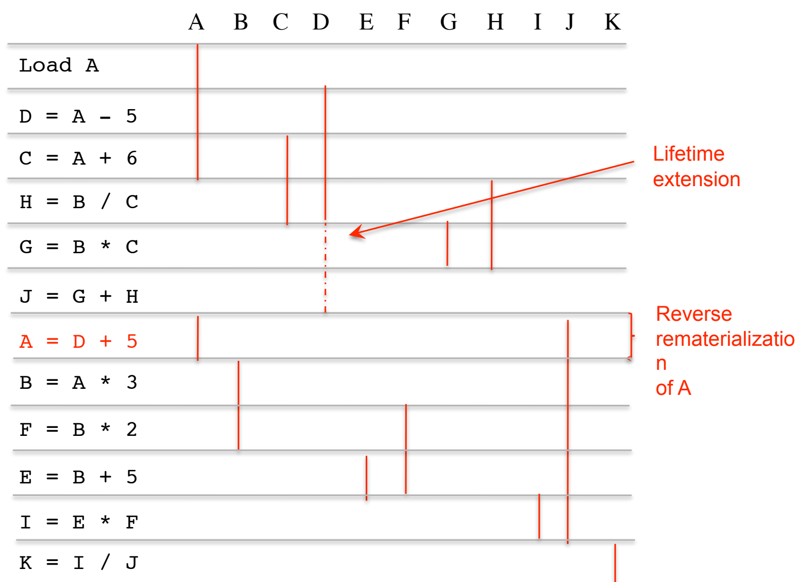
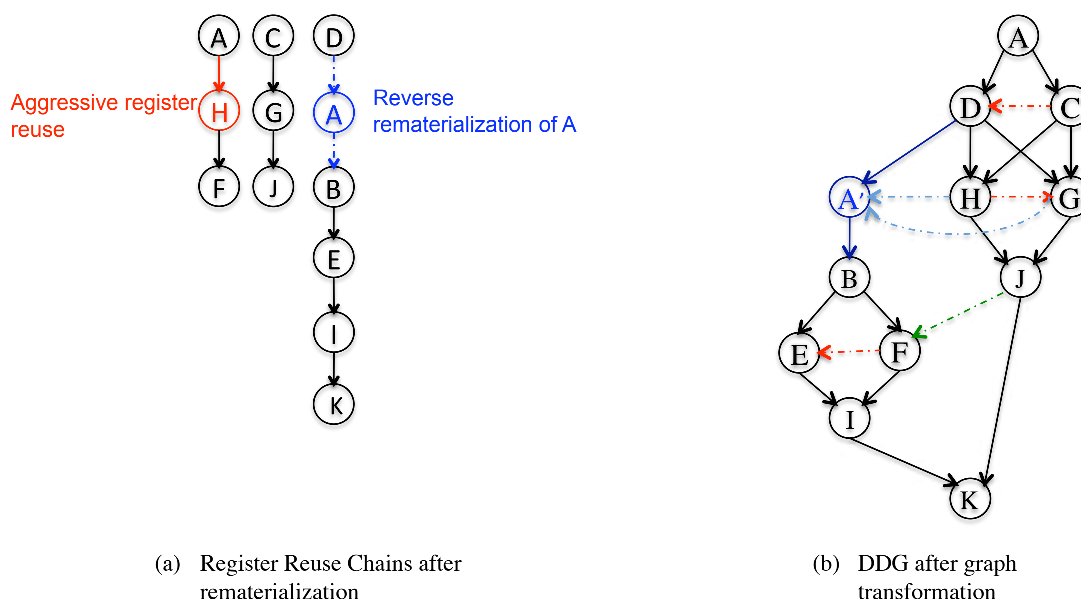


Figure 5.14: Values lifetime after rematerialization.



(a) Register Reuse Chains after rematerialization

(b) DDG after graph transformation

Figure 5.15: Data dependency graph after register rematerialization.

5.2.4 Graph Transformation

Once rematerialization decision is made we proceed to the graph transformation of the original DDG. Graph transformation, or graph rewriting, consists in rules for creating a new graph out of an original graph: we insert a copy of the rematerializable value with all edges from its new inputs, and move all edges from the rematerializable value to nodes calculated after the excessive node (new reusing node) to the new copy node.

The algorithm guarantees that the graph transformation does not create any cycle in the new graph. No cycle exists between the new inserted value and the excessive value as all inserted edges from the new node are directed to values computed after the excessive value. Hence there is no path between the inserted node and the excessive node. In Figure 5.16 we show an example of graph transformation for the 3 address code shown in Figure 5.3, in this example D is the excessive node, A the rematerializable value and A' is the copy value of A rematerialized from B and C. Because we know that A' is computed after C and B all output edges from A' are to nodes computed after C and B. Therefore there is no path between output nodes of A' and C or B. As a result there is no path between A' and C or A' and B hence no cycle between A' and C or A' and B.

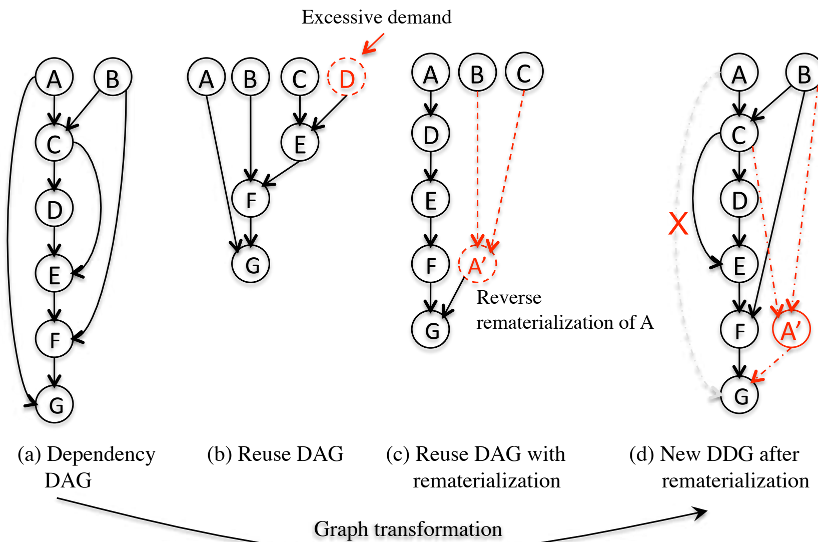


Figure 5.16: Graph transformation.

The rematerialization algorithm we propose is iterative, after each graph transformation we re-call the algorithm till there will be no rematerializable value.

It is important to note that it may happen that sometimes rematerialization reduces only locally the register pressure and not globally. In Figure 5.17, rematerialization has no effect on the global register requirements. But, if we assume there are two registers available, rematerialization helps to avoid a first spill. Even though this does not seem to improve a lot since the register count is fixed, this can in fact be exploited for running more computations (parts of DAG, iterations, threads) in parallel.

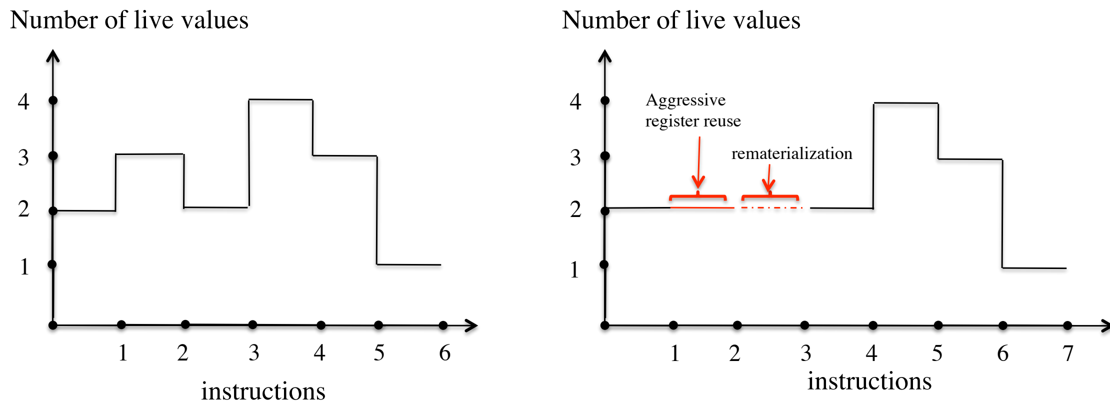


Figure 5.17: Local register pressure reduction

5.2.5 More Opportunities for Reverse Computing than for Direct Computing

Based on the abstract definition of reversible function where the number of inputs equals the number of outputs, any operation is reversible since the register reuse is limited between direct dependent operands, which means that values of reused registers can be retrieved easily from output operands, contrary to the direct rematerialization where the necessary condition is to keep all input values of the operation live.

Also, the direct rematerialization is limited in the fact that program's inputs are not rematerializable, and there is only one way to recompute intermediate values through direct operations, unlike the reverse rematerialization where inputs like intermediate values are always rematerializable and from different instructions since register reuse is limited between direct dependent values. Taking back the example in Figure 5.4. B can be recomputed in reverse way from two different operations, from C or from I and J. However there is only one way to recompute B with a direct operation (from A). This effect has to be more precisely measured experimentally and theoretically analyzed with respect to DAG properties.

5.3 Experimental Results

In this section, we evaluate the effectiveness of our compiler optimization. In order to understand more about the recomputing opportunities, we developed a profiling tool that is able to measure the degree of recomputing in an acyclic code. We perform two separate evaluations: **direct rematerialization** does rematerialize a value with a direct operation from its source operands. **Reverse rematerialization** does rematerialize a value in the case of high register pressure with a reverse operation. In both cases rematerializing a value

can be done by multiple instructions. We apply our tool on a set of Data Dependency Graphs extracted from the most critical kernel in the LQCD simulation program [18]. The kernel of the code *(Hopping_Matrix)* contains two separate synchronized loops k and l, which we will call them also *Hopping_Matrix_k* and *Hopping_Matrix_l* respectively.

One iteration of loops k and l contains 768 and 840 operations respectively. Their corresponding data dependency graph contains 872 and 1016 nodes respectively. Our tool uses a register allocator based on register reuse chains. We used this kernel to show that opportunities exist for both direct and reverse rematerialization to significantly reduce spill costs. We performed many experiments with different values for the number of available registers.

5.3.1 Lattice QCD Computation

This application performs complex operations on 4-dimensional space-time lattice that describes the strong force which binds protons and neutrons forming the atomic nucleus. The main kernel routine, called *Hopping_Matrix*, is contributing about 90% of the total execution time [70]. This routine is responsible for computing the action of the Wilson-Dirac operator, as mentioned in the equation bellow. The actions of Dirac operator involve a sum over quark field ($\psi_{i+\hat{\mu}}$) multiplied by a gluon gauge link ($U_{i,\mu}$) through the spin projector ($I \pm \gamma_\mu$)

$$\chi_i = \sum_{\mu=x,y,z,t} \kappa_\mu \{U_{i,\mu}(I - \gamma_\mu)\psi_{i+\hat{\mu}} + U^\dagger_{i-\hat{\mu},\mu}(I + \gamma_\mu)\psi_{i-\hat{\mu}}\}$$

In technical term, the core kernel involves *O(n)* computation over two sequential and synchronized loops, single or double precision floating point latices, where each loop iteration itself involves complex-vector multiplications, vector-matrix multiplications, vector additions and vector subtractions. In the first loop, for every lattice point, as inputs, there are four complex constants, one input *spinor* contains that four *SU(3)* vectors, each of these *SU(3)* vectors consists of three complex numbers. Four inputs three-by-three complex matrix U that refer to *gauge* links. As output, there are eight *halfspinor phi* consist of two *SU(3)* vectors, leading to a total data usage of 104 input complex numbers and 96 output complex numbers. Inputs of the second loop are the same four U matrices and four complex constant of the first loop with its eight halfspinor outputs. The second loop returns one spinor. The loop body consists of the same operations as in the first loop.

Considering that an input value should be loaded at most once, and once a final output is computed it is directly stored in the memory, thus:

- One complex-complex multiplication requires 6 floating point operations and 6 registers.
- One complex-vector multiplication requires 18 floating point operations and 10 registers at least.
- One vector-matrix multiplication amounts to 66 floating point operations and 14 registers at least by performing three independent vector-vector multiplications.

The result accumulation is 768 flops for the first loop and 840 flops for the second one. An optimized use of data lets reduce the memory usage to 48 registers per iteration in

both first and second loop.
For a maximum instruction level parallelism, the register requirements are:

- 6 registers for complex-complex multiplication.
- 14 registers for complex-vector multiplication.
- 18 registers for vector-vector multiplication.
- 42 registers for vector-matrix multiplication.

Thus, the memory size required for the kernel scales linearly with the loop size, but register requirements and the number of floating point operations are fixed *O(1)*.

Figures 5.19, 5.20, 5.22 and 5.21 show data dependency graphs of $Hopping_Matrix_k$, $Hopping_Matrix_l$, $_complex_times_vector$, and $_su3_multiply$.

5.3.2 Register Requirements

Both reverse and direct rematerialization reduce the overall register requirements of *Hopping_Matrix* but with different reduction rates and different costs. Table 5.2 shows the number of register requirements for each benchmark before and after rematerialization by using reverse and direct computing with one and multiple instructions. The percentage gain is shown in table 5.3, followed by the cost of reversibility given by the number of additional operations. There is no positive impact of direct rematerialization on *_su3_multiply* and *_complex_times_vector* in *Hopping_Matrix (loop k)* or *_su3_inverse_multiply* and *_complexcjg_times_vector* in *Hopping_Matrix (loop l)*, unlike when using reverse rematerialization with one or multiple instructions that helps by reducing the number of register requirements by 16.2% and 33.3% respectively in *_complex_times_vector*, and 7.1% and 14.3% in *_su3_multiply.*

benchmark	**without. remat.**	**direct remat.**	**reverse remat.**	**reverse remat. multiple. inst.**
_complex_times_vector	6	6	5	4
_complexcjg_times_vector	6	6	5	4
_su3_multiply	14	14	13	12
_su3_inverse_multiply	14	14	13	12
Hopping_Matrix (loop k)	48	39	35	33
Hopping_Matrix (loop l)	48	48	47	45

Table 5.2: Contribution of reverse rematerialization to the minimization of register requirements

Even in $Hopping_Matrix$, the direct rematerialization is less successful than reverse rematerialization and with highest cost; 54 operations added which represent 7% of the total number of operations in *Hopping_Matrix (loop k)* for a reduction of register requirements of 18.7%. The same amount of register requirement reduction is given by using reverse rematerialization with lowest cost, only 33 operations added from total of 768 operations that represents 4.3%. With reverse rematerialization with multiple instructions we achieve up to 31.2% of reduction. The direct rematerialization is limited due to the multi usage of inputs data, which are not rematerializable in a direct way.

By considering all macros and data structures in the l loop of *Hopping_Matrix* as elementary elements, the data dependency graph corresponding to one iteration of the loop l is a tree where each node v (except inputs nodes) depends at least on one node u with one output edge, that means v is the only reuse node of u. As the rematerialization depends on the number of reuse of variables, in a graph if there are few cases of node reuse then there are few opportunities of rematerialization. In the loop l, only reverse rematerialization in functions *_su3_inverse_multiply* and *_complexcjg_times_vector* has a positive impact. Table 5.2 shows a register pressure reduction of 1 register by using reverse rematerialization with one instruction and 2 registers with multiple instructions.

	register requirements	**gain**	**direct remat.**	**reverse remat.**	**reverse. remat. multiple inst.**
Hopping_Matrix (loop k) 768 operations 872 nodes	39/48	18.7%	54	33	33
	35/48	27%	-	45	45
	33/48	31.2%	-	-	153
Hopping_Matrix (loop l) 840 op 1016 nodes	47/48	2.1%	-	8	8
	45/48	6.2%	-	-	94
_complex_times_vector 18 op 26 nodes	5/6	16.7%	-	3	3
	4/6	33.3%	-	-	9
_su3_multiply 66 op 90 nodes	13/14	7.1%	-	3	3
	12/14	14.3%	-	-	21

Table 5.3: Cost of the reversibility: number of additional operations

Finally, note that even in the absence of register pressure reduction, reverse rematerialization can reduce the number of variables that are alive simultaneously at some computing step, thus register pressure is reduced locally and therefore load/store operations could be avoided.

5.3.3 Spill Costs

We measure spill costs statically. For that we compute the register requirements which is the maximum number of variables that are simultaneously alive. When the number of available registers is less than the number of simultaneously alive variables, the register allocator decides which variables should not be stored in registers and load/store instructions are introduced. The number of spill operations depends on the number of variables that exceed the number of available registers, hence reducing the maximum number of simultaneously live variables N by S with R is the number of available registers and $(N - S) \geq R$ means that spill cost is reduced at least by S. For all rematerialization techniques, the number of available register is assumed to be one. Thus, the rematerialization algorithm extracts recomputing and applies rematerialization to reduce register requirements as small as possible.

Table 5.4 shows the number of static spills using different techniques of rematerialization. The table compares all techniques and shows how reverse rematerialization is more beneficial. For example, it indicates 51 load/store operations for *Hopping_Matrix (loop k)* without rematerialization; compared to 45 load/store operations with direct rematerialization and no spill instruction using reverse rematerialization, for 35 available registers.

benchmark	number of available registers	number of spill operations			
		without remat.	direct remat.	reverse remat.	reverse remat. multiple inst.
_complex_times_vector	5	3	3	0	0
	4	6	6	3	0
_complexcjg_times_vector	5	3	3	0	0
	4	6	6	3	0
_su3_multiply	13	3	3	0	0
	12	6	6	3	0
_su3_inverse_multiply	13	3	3	0	0
	12	6	6	3	0
Hopping_Matrix (loop k)	39	45	0	0	0
	35	51	45	0	0
	33	57	51	9	0
Hopping_Matrix (loop l)	47	4	4	0	0
	45	12	12	8	0

Table 5.4: Contribution of reverse rematerialization to minimize spill operations

5.3.4 Run-Time Performance

The performance improvement from the reduction in explicit spills cannot be determined exactly, but running the two equivalent codes shown in Figure 5.18 - a simulation of the above example in Figure 5.4 - shows a difference in performance up to 40% for a sequence size equals to $100x2^{10}$. This is because for the first code, the maximum number of simultaneously live values is equal to the sequence size which is larger than the number of registers, creating more spills to memory. Inversely for the second code where the number of simultaneously live values is constant and independent of the sequence size.

```
for(i=0;i<SIZE-1;i++)
     A[i+1]=A[i]+ i;
B=A[SIZE-1];
for(i=SIZE-1;i>=0;i--)
     B=B+A[i];
return B;
```

(a) without rematerialization

```
for(i=0;i<SIZE-1;i++)
          A=A+i;
B=A;
B=B+A;
for(i=SIZE-2;i>=0;i--){
          A=A-i;
          B=B+A;
}
return B;
```

(b) with reverse rematerialization

Sequence's size	5120	10240	102400	1024000
%Performance (double)	+25%	+37%	+40%	+45.5%
%Performance (simple)	-6%	+10%	+26%	+30%

Figure 5.18: Contribution of reverse rematerialization to execution time

5.3.5 Inverse Precision

In a static analysis of FORTRAN programs, Knuth [41] reports that 39% of arithmetic operators were additions, 22% subtractions, 27% multiplications, 10% division, and 2% exponentiations.

All numbers expressed in floating point format are rational numbers with a terminating expansion in the relevant base. The number of bits of precision limits the set of rational numbers that can be represented exactly, the error can be related to the decimal place of the right-most significant digit, specifically for multiplication and division where results in general are rounded, so in turn the result of the inverse operation is not exact if the result it self is not, though small errors may accumulate as operations are performed repeatedly. In case of rematerialization with one instruction, at most one operand is inexact, for the multiplication $y * z = x + error(x)$, the error of the inverse operation $\frac{x+error(x)}{y}$ is $\frac{error(x)}{y}$. For the division, $\frac{y}{z} = x + error(x)$, the error of the inverse operation $y = x * z$ is $error(y) = z * error(x)$

But in general, the error propagation given dependent variables each with an error is: for the addition and the subtraction, the precision error in the result is given by: $error(x) = error(y) + error(z)$ for the operation $x = y + z$. For the multiplication and the division, the maximum error in the result is given by: $error(x) = error(y) * z + error(z) * y + error(y) * error(z)$. Usually $error(y) << y$ and $error(z) << z$ so that the last therm is much smaller than the other terms and can be neglected. Formally we write more compactly by forming the relative error, that is the ratio of $error(x)/x$, namely $\frac{error(x)}{x} = \frac{error(y)}{y} + \frac{error(z)}{z} + ...$

Even though we did not observe differences between both original and optimized version of one LQCD kernel run, we are aware that this application is very sensitive to data precision. We have not yet run the whole code that intensively iterates on data and calls this kernel. This will be a meaningful test.

5.4 Summary

We have presented reverse rematerialization, a novel method for reducing register pressure. Reverse rematerialization takes advantage of the relative cost of computing versus memory access. It recomputes data instead of spilling them. We have found that there may be more opportunities for recomputing a value in reverse direction from output operands than recomputing it from its original input operands. In this context reverse rematerialization seems more beneficial. It provides a mechanism to reduce register pressure with a lowest cost than classical rematerialization techniques. Our rematerialization algorithm targets the basic blocks with higher register pressure, it is intended to work after register allocation based on register reuse chains that can provide all necessary information to extract opportunities for recomputation in a graph.

Reverse rematerialization is also an alternative to spilling. Spilling is just storing intermediate values in memory. Conversely we want to see if rematerialization and especially reverse rematerialization could be an alternative to storing unneeded arrays of intermediate values in the memory. This could help exploiting at most as possible the available memory which is one of the bottlenecks of LQCD. The next step would be then to apply it also to communication - recompute rather than communicate.

We also have to extensively check whether precision issues can be overcome, this will be done on the LQCD application that is a specially well adapted benchmark for that purpose as it requires very high precision at least in some parts.

In the next chapters, we will see how reverse rematerialization may improve the parallelism. Improving register reuse by using this method can increase instruction and thread-level parallelism typically available in the GP-GPU - general purpose graphical process units.

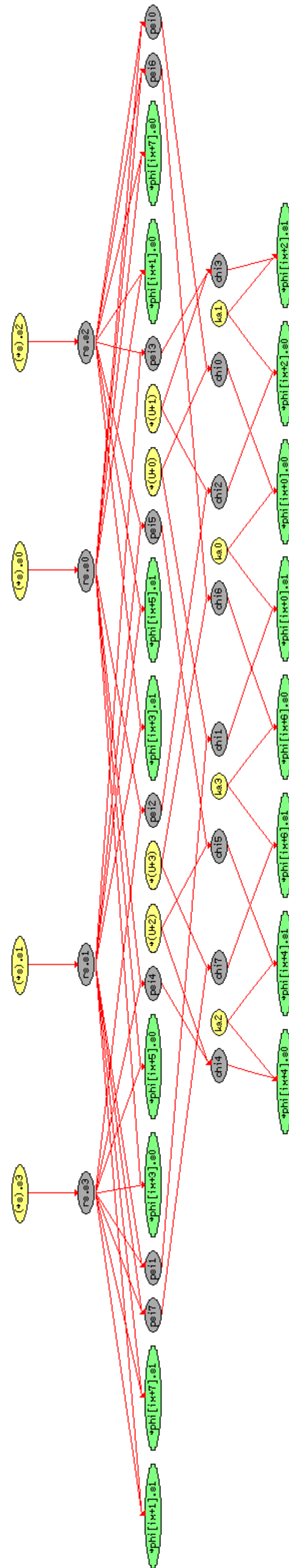


Figure 5.19: Hopping_Matrix_k: data dependency graph

Figure 5.20: Hopping_Matrix_l: data dependency graph

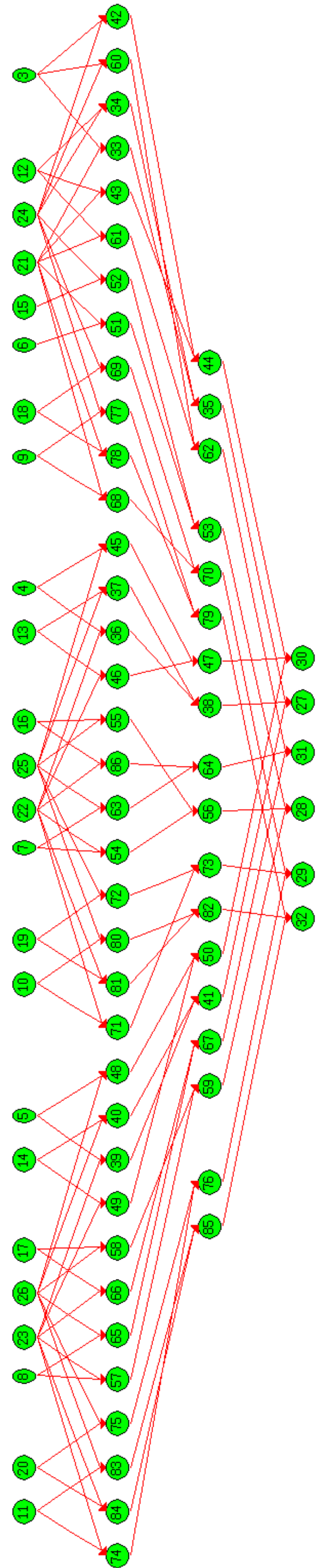


Figure 5.21: su3_multiply: data dependency graph

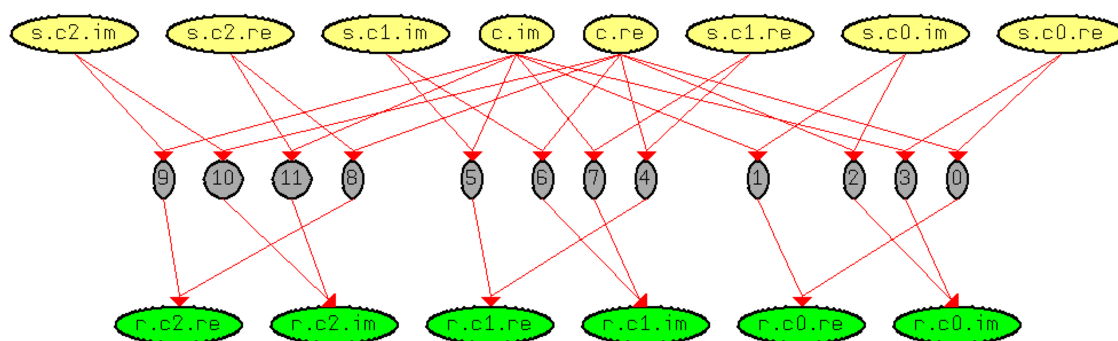


Figure 5.22: complex_times_vector: data dependency graph

Chapter 6

Using Reverse Computing to Increase Intruction Level Parallelism

Contents

6.1 What is Instruction-Level Parallelism? 98
6.1.1 Instruction-Level Parallelism Challenges 98
6.2 Cell BE Implementation 101
6.2.1 Cell BE Architecture Overview 101
6.2.2 Programming Cell BE 102
6.2.3 Performance Measurement 105
6.3 Summary 106

In this chapter, we show how we can improve instruction-level parallelism on Cell BE (IBM) by increasing register reuse through recomputing, to allow more independent instructions to be performed simultaneously. Even with enough available registers, rematerialization through reverse operations may help to reduce register usage per loop iteration by increasing pipeline usage through unrolling loops. We demonstrate a 16.8% (statically timed) gain over a basic LQCD computation on Cell BE.

6.1 What is Instruction-Level Parallelism?

A program is, in essence, a set of instructions, that can be grouped together to accomplish a task. These instructions can be scheduled according to the data and resources dependencies among them, and then executed in parallel without changing the result of the program, to speed up the execution. This is called Instruction-Level Parallelism (ILP). This is achieved by performing different stages of the pipeline[1] through a number of execution units within the processor.

Example:

1.	A	=	B	+	C		1.	A	=	B	+	C
2.	E	=	A	*	D	Scheduling	3.	G	=	F	+	C
3.	G	=	F	+	C	⟹	2.	E	=	A	*	D

IF	ID	ADD	...	...	...	IF	ID	ADD	...	...	...
	IF	ID	stall	stall	...		IF	ID	MUL	...	...
		IF	stall	...	...			IF	ID	stall	...

The above code fragment consists of three instructions. The second instruction depends on the result of the first instruction. So that, the processor stalls till computing of the value A. The third instruction can be executed at the same time as the first instruction because they do not depend on each other's result. The compiler or the run-time system - for instance superscalar processor - separate dependent instructions, 1 and 2, and regroup independent instructions, 1 and 3, to be executed in parallel. In both cases, before and after scheduling, the parallelism is the number of instructions divided by the number of cycles required. Architectures that have been proposed to take advantage of this kind of parallelism are superscalar processors [1] and VLIW architectures [52].

6.1.1 Instruction-Level Parallelism Challenges

Parallelism is limited among instructions. First, by data dependencies between pairs of instructions. Second, by resource dependencies when an instruction requires a hardware resource which is still being used by a previously issued instruction.

In the following, we apply our reverse computing-based technique in order to increase the amount of ILP in the kernel of *Hopping_Matrix* according to resource constraints. Remember that the usefulness of reverse rematerialization is not only decreasing register

[1] Instruction pipeline is a technique used to increase the number of instructions that can be executed in a unit of time. The fundamental idea is to split the processing of a computer instruction into a series of independent steps, with storage at the final stage.

pressure, but also increasing register reuse to allow more instructions to be performed simultaneously.

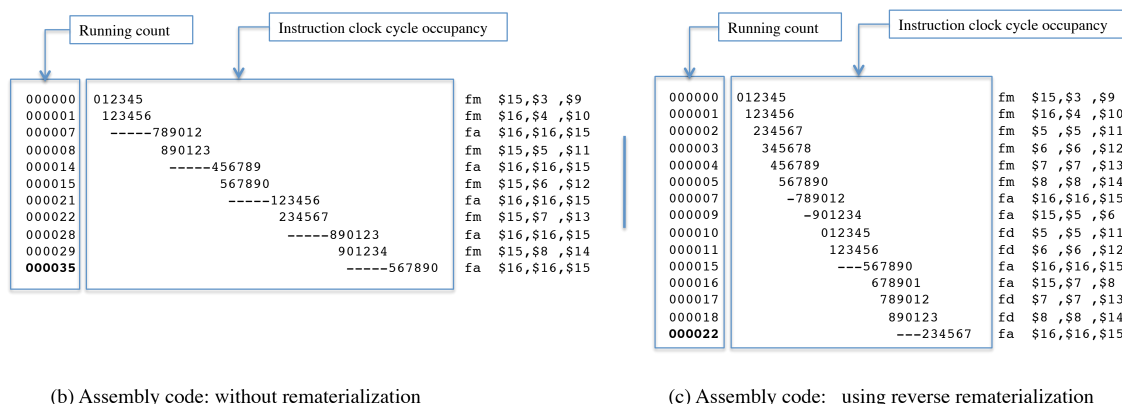
Two different kinds of techniques for increasing the instruction-level parallelism can be exploited; exploiting it within a basic block and exploiting it across basic blocks.

6.1.1.1 Instruction-Level Parallelism within Basic Blocks

Consider the following code that corresponds to a part of the *_su3_multiply* macro. As the assembly code in Figure 6.1(a) shows, the added operations caused by reverse rematerialization do not increase the cycle time since they are overlapped with synchronized operations and replace stall cycles. In this example, we consider that a reverse operation has the same cycle time of its original operation. This is only an abstraction specially for the multiplication and the division.

```
0  C[0]  = A[0]*B[0]
1  C[1]  = A[1]*B[1]
2  C[2]  = C[1]+C[0]
3  C[3]  = A[2]*B[2]
4  C[4]  = C[3]+C[2]
5  C[5]  = A[3]*B[3]
6  C[6]  = C[5]+C[4]
7  C[7]  = A[4]*B[4]
8  C[8]  = C[7]+C[6]
9  C[9]  = A[5]*B[5]
10 C[10] = C[9]+C[8]
   ...
```

(a) 3 address code



(b) Assembly code: without rematerialization

(c) Assembly code: using reverse rematerialization

Figure 6.1: Using reverse computing to increase instruction level parallelism

Operation 2 depends on the results of operations 0 and 1, so it cannot be calculated until both of them are completed. Knowing that input data stay alive during the whole computation, so with only two available registers, the processor stalls for six cycles before computing the result of operation 2. However operations 0, 1, 3, 5, 7, 9 do not depend on any other operation, so they can be calculated in parallel if there are enough available registers. As can be seen in Figure 6.1(c), by using reverse rematerialization we can use registers of input data to perform the six operations and replace most of stall cycles. We rematerialize input data once intermediate values are used. The performance gain in this example is up to 31.7%

Table 6.1 shows the number of clock cycles - statically timed - and the speedup due to reverse computation for some basic computations of LQCD program. We set the number of available registers for small codes like *_vec_times_vec*.

Benchmark	Available registers	Cycles -without remat.-	Cycles -rev remat.-	Improvement
_vec_times_vec	14	49	44	10.2%
Hopping_Matrix (loop k)	-	4815	4004	16.84%

Table 6.1: Contribution of reverse rematerialization to improve performance

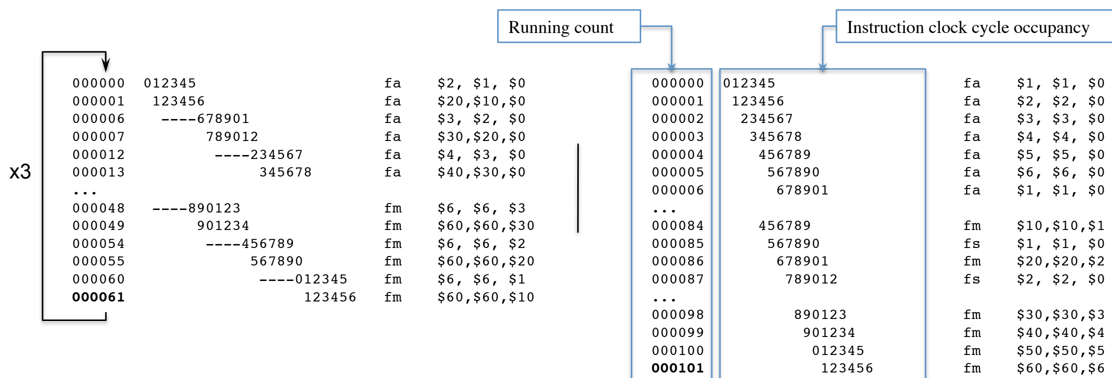
6.1.1.2 Instruction-Level Parallelism across Basic Blocks

Sometimes, even if the register file can hold all intermediate values of one iteration, the processor might occasionally stall as a result of data dependencies and branch instructions. To avoid stalls, a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction. Rematerialization through reverse operations helps to reduce register usage per iteration to keep pipeline full by unrolling loops without any spill operations. It can exploit parallelism among instructions by finding sequences of unrelated instructions from different iterations that can be overlapped in the pipeline.

Consider the C code in Figure 6.2(a), where the body of the outer loop is a simulation of the example of Figure 5.4(a). The first inner loop generates a sequence of numbers where each term is found by adding the previous one with a fixed number S. The second inner loop returns the product of all previous elements and stores the result in array B in memory.

```
for(j=0;j<N;j++){
     A[0]=B[j];
     for(i=1;i<M;i++)
          A[i] = A[i-1] + S;
     B[j] = A[M-1] * A[M-2];
     for(i=M-3;i>=0;i--)
          B[j] = B[j] * A[i];
}
```

(a) Pseudo C code



(b) Assembly codes - without reverse rematerialization

(c) Assembly codes - using reverse rematerialization

Figure 6.2: Using reverse computing to increase instruction level parallelism

Here, the compiler can exploit a minimum of M registers, the sequence's size, which is the number of simultaneously live values. As shown in Figure 5.4(c), using reverse rematerialization, we can reduce the register requirement to only two registers.

By assuming that the number of the first inner loop's iterations is 6, and the number of available registers is 12, the compiler can select the unroll factor that fits the register requirements into the available registers. The outer loop can be unrolled twice, without using rematerialization, and 6 times using reverse rematerialization. The aim is to avoid the stalls and extra tests and branches. After unrolling, there are 2 copies of the original loop body and 6 copies of the modified loop with additional operations. If we assume that the number of j-loop iterations is 6, the estimated running time of this loop is the running time of the body loop after unrolling, times the new number of iterations, as shown in Figure 6.2(b). In this example it is estimated to 201 cycles for the original code without using rematerialization, compared to only 107 cycles using rematerialization.

The assembly code in Figure 6.2(c) shows this optimization and the number of cycles, statically timed using the spu-timing tool of IBM on Cell BE. As can be seen, with reverse rematerialization, we can use available registers to perform six independent operations and replace all stall cycles. The performance gain in this example is x2.

However, this optimization is traded off on Cell BE due to the small size of SPE's local store (LS) where the code has to be loaded, against the potential penalty caused by increased code size on the larger loop body to fit both code and data.

6.2 Cell BE Implementation

6.2.1 Cell BE Architecture Overview

The cell Broadband Engine (CBE), shown in Figure 6.3, is a single chip heterogeneous multi-core processor that can provide a huge computational power with high efficiency for a wide range of applications due to a 64-bit Power Processor Element (PPE) and eight Synergistic Processor Elements (SPEs).

Each SPE is a Single Instruction Multiple Data (SIMD) engine with 128-bit vector registers for single and double precision instructions, and 256 KB of non-coherent local memory. It communicates with other SPEs and main memory through its DMA controller. The communication between the PPE, the SPEs, main memory, and external devices is realized through an Element Interconnect Bus (EIB), which has a 204.8 GB/s bandwidth peak performance at half the system clock rate. The Memory Interface Controller (MIC) provides 25.6 GB/s to system memory. The I/O controller (IO) provides peak bandwidth of 25 GB/s inbound and 35 GB/s outbound.

The PPE and each SPE can complete eight single precision operations per clock cycle by using a vector fused-multiply-add instruction ("SIMD" instructions), which translates to 25.6 GFLOPS at 3.2 GHz. In double precision, at the same processor clock rate, the theoretical peak performance is 6.4 GFLOPS for PPE and 1.8 GFLOPS per SPE. As a result, the CBE is capable of achieving 230.4 GFLOPS in single precision and 20.08 GFLOPS in double precision at 3.2 GHz.

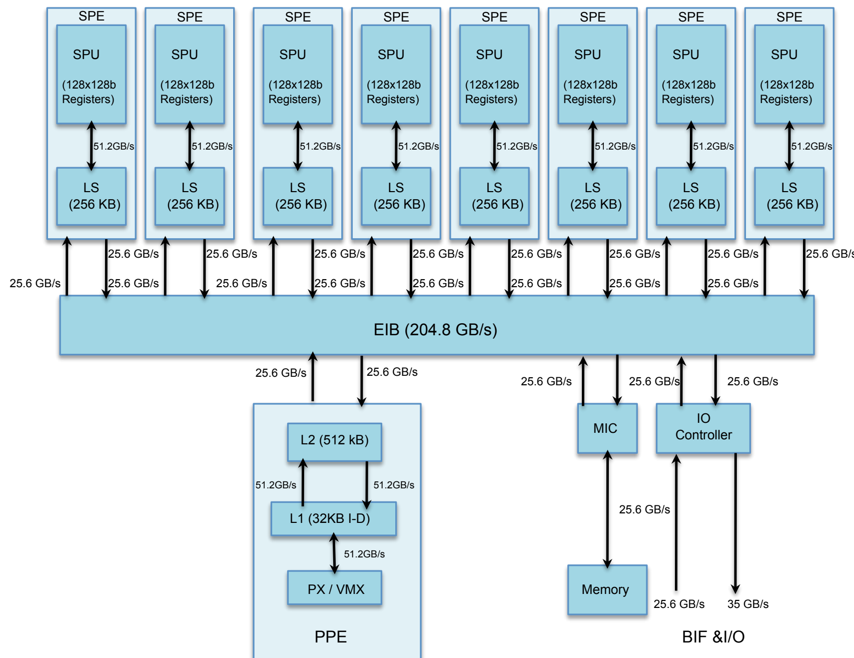


Figure 6.3: Cell Broadband Engine Architecture

6.2.2 Programming Cell BE

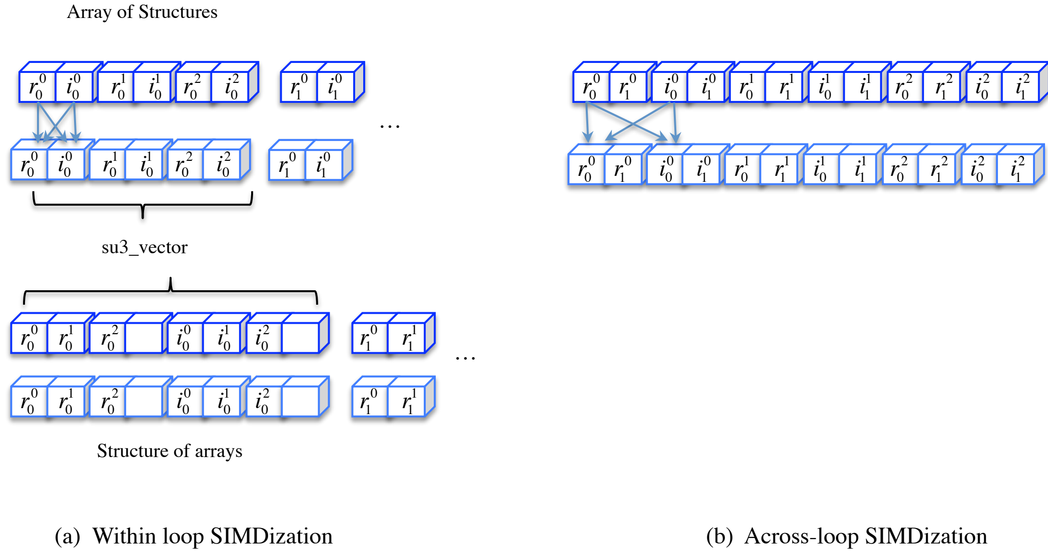
6.2.2.1 Code SIMDization

The first important step in developing Cell BE application, after having determined kernels, is to use Vector/SIMD functional units; Multimedia Extension (VMX unit of the PPE) and Synergistic Processor units (SPUs of the eight SPEs). Nearly all instructions provided by the SPE operate in a SIMD fashion on 128 bits of data. SIMDization is a form of fine grained parallelism that obeys certain constraints with respect to accessed data. Data have to be aligned and contiguous.

Therefore, to achieve high rates of computation at moderate costs in area, data organization is very important. Two organizations are possible: array of structures (aos) and structure of arrays (soa). In our code, the first organization helps to optimize space and use the whole 128-bit registers, because the elementary data in the code is structure of three complex numbers, each complex variable representing two 64-bit double floats, so three registers are required to hold three complex numbers, contrary to the second organization that requires four registers to hold one *SU(3) vector*. Data organization is shown in Figure 6.4.

Recall that two complex numbers $x = a + bi$ and $y = c + di$ are multiplied as follows:
$Real\ part = a * c - b * d$
$Imaginary\ part = a * d + b * c$
In the array of structure organization, one vector register can hold one complex double precision. So, within-loop SIMDizable, a cost model is employed to shuffle SIMD vectors elements.

A best way to enable an efficient parallel execution is extracting SIMD parallelism



(a) Within loop SIMDization (b) Across-loop SIMDization

Figure 6.4: Data organization for SIMD operations (SM)

across-loop iterations and not within-loop, as loop-level SIMDization satisfies dependence conditions as a traditionally vectorizable loop. This makes easier and faster developing and avoids all shuffling operations.

Data in this case is structure of very long arrays. Figure 6.5 shows different data structures used in our implementation of LQCD on Cell BE.

```
struct complex {
   double re,im;
};

struct su3 {
   complex c00,c01,c02,c10,c11,c12,c20,c21,c22;
};

struct su3_vector {
   complex c0,c1,c2;
};

struct spinor {
   su3_vector s0,s1,s2,s3;
};

struct halfspinor {
   su3_vector s0,s1;
};

spinor *k;
halfspinor *phi;
su3 *gauge;
```

Data structure: within loop SIMDization

```
struct complex {
   double *re,*im;
};

struct su3 {
   complex c00,c01,c02,c10,c11,c12,c20,c21,c22;
};

struct su3_vector {
   complex c0,c1,c2;
};

struct spinor {
   su3_vector s0,s1,s2,s3;
};

struct halfspinor {
   su3_vector s0,s1;
};

spinor k;
halfspinor phi;
su3 gauge;
```

Data structure: across loops SIMDization

Figure 6.5: Different data structures used in LQCD

6.2.2.2 Code Partitioning

The next step, is to determine for each kernel the number and mapping of SPE threads used in the computation. The *Hopping_Matrix* algorithm exhibits two synchronized loops containing synchronized operations. We wrote two kernels for these two loops and we called them *Hopping_Matrix_k* and *Hopping_Matrix_l*, and each thread of each kernel would perform a different amount of work. The master thread runs on the PPE processor.

6.2.2.3 Communication and Data Transfer

A more significant step, is to determine the optimal memory layout and the access to memory. The PPE accesses main storage with load and store instructions that move data between main storage and a private register file, the contents of which may be cached. The PPU can perform one 16 Byte access to the 32KB L1 cache per cycle. The L1 cache has one read bus and one write bus to the 512 KB L2 cache performing up to two 16 Byte accesses per cycle (one read, one write). The SPEs, in contrast, access main storage with direct memory access (DMA) commands that move data and instructions between main storage and local store (LS). The SPE's 256KB local memory fully pipelined 16-byte accesses for memory instructions and 128-byte accesses for instruction fetch and DMA transfers. To fit code and data into the small SPE memory and keep the computation fluid, we used double buffering on the input and output data buffers where possible. Essentially, during current computation we had ongoing DMAs that transfers out the previous results and transfers in the next input data from/to global memory. See Figure 6.6 for an illustration of how double buffering works.

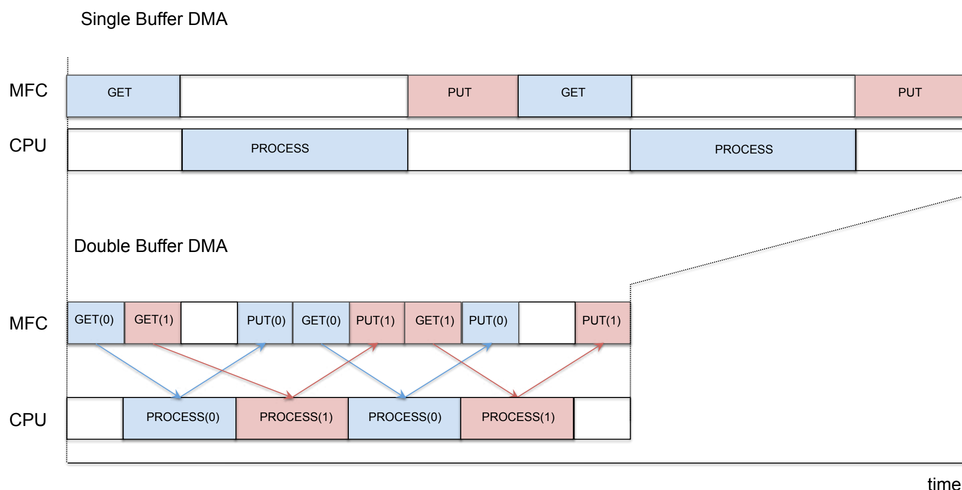


Figure 6.6: Double buffering

Register Allocation Register allocation is very important in Cell BE in terms of performance, because, first the memory is slow and second the local store of SPE is small (256KB) to fit both code and data. Reducing lifetime of intermediate local variables allows to use fewer registers and less local store and avoids each time 6 cycles of load/store latency in case of high register pressure.

By programming the SPE in assembler, we can select which registers to use and schedule the code manually. We perform aggressive register allocation to increase register reuse

using recomputing for all local computations, to make good use of the 128-entry register file. As a result, most local variables are rematerialized from live data, and thus memory storage and associated load/store instructions are reduced. This provides a high degree of flexibility and, when used judiciously, will yield highest performance.

6.2.3 Performance Measurement

The experiments were done on the IBM BladeCenter QS22. We first evaluate the across-loops SIMDization for double and single precision computation by measuring SPE time elapsed without DMA transfer. The across loops SIMDization delivers better SPE performance of up to x2 compared with within-loop SIMDization. For single precision computation, we do not expect better results with within-loop SIMDization because more shuffling operations are required than in double precision computation.

Tables 6.2 and 6.3 show the SPU time spent by both *Hopping_Matrix_k* and *Hopping_Matrix_l*, without data transfer, for the maximum buffer size, for one iteration. Data size concerns only input and output data of the program. The number of operations in the tables is the total number of operations executed for the whole buffer size.

To measure time elapsed in DMA transfer we initialize the decrementer register just before mfc_put, which writes back data in the main memory, and we read the new register's value after having got input data from main memory with mfc_get. We did not implement a single precision version for SIMDization within-loop because we believe that performance will be penalized by the increased number of shuffling operations that will be required.

SIMDization	across-loops		within-loop
	single precision	double precision	double precision
Code size (KB)	16.58	16.43	10.11
Data size (KB)	216.125	222.125	243
Number of iterations	36	74	162
Number of operation	110592	113664	124416
SPU time (μs)	12.94	18.08	38.14

Table 6.2: Hopping_Matrix_k

SIMDization	across-loops		within-loop
	single precision	double precision	double precision
Code size (KB)	18.85	18.63	12.95
Data size (KB)	216.125	216.125	243
Number of iterations	36	72	156
Number of operation	124320	124320	131040
SPU time (μs)	11.89	15.83	42.34

Table 6.3: Hopping_Matrix_l

The number of iterations with SIMDization across-loops is divided by two in double precision version and by four in single precision version, because we perform two and four spinors data per iteration respectively, contrary to the SIMDization within-loop where one spinor data is performed per iteration. However, the loop body in the SIMDization across-loops version is larger, which means that the code size is larger too. The different

data size in different versions is explained by the limited LS space which is being used for both code and data. An increased code size limits space to buffer data.

For the global Cell BE performance, we vary the used SPE count from 1 to 16. Across loops SIMDization still returns better results but not with the same rate comparing with within-loop SIMDization. As shown in Tables 6.2 and 6.3, the code size in within-loop SIMDization implementation is smaller than that in across-loop SIMDization implementation which allows to increase the buffer size and reduce the number of DMA transfers.

Figures 6.7 shows global performance on the QS22 and how performance scales with the count of SPEs. The peak performance is reached with 8 SPEs most of time. So few SPEs can be used to save power.

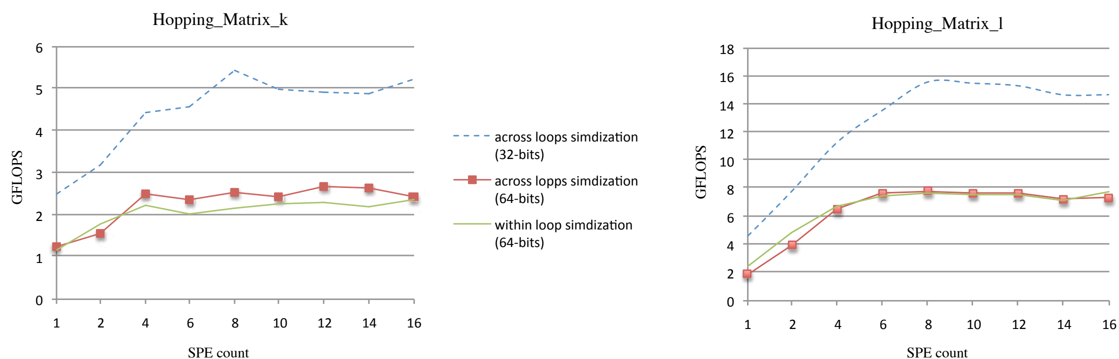


Figure 6.7: Scalability of performance with the count of the used SPEs for different SIMDization techniques

6.3 Summary

We have seen that reverse rematerialization may improve instruction parallelism, within and across basic blocks.

Within a basic block, when stall cycles due to data dependencies are caused by the limited number of registers, reverse rematerialization can increase register reuse to allow more independent instructions to be performed simultaneously and replace stall cycles.

Across multiple basic blocks and multiple iterations of a loop, reverse rematerialization can help to reduce register demands per iteration to increase pipeline usage through unrolling loops to exploit parallelism among instructions by finding sequences of unrelated instructions from different iterations that can be overlapped in the pipeline.

In the next Chapter, we will show how reverse rematerialization can increase thread-level parallelism typically available in the GP-GPU - general purpose graphical process units, by improving register pressure.

Chapter 7

Using Reverse Computing to Increase Thread Level Parallelism

Contents

7.1 GPU Architecture and Programming Model 108
7.1.1 Memory Hierarchy 109
7.1.2 Thread-Level Parallelism 110
7.1.3 Register Usage, Rematerialization and Performance 110
7.2 Experimental Results 113
7.3 Analysis of Results 116
7.3.1 Limitations 117
7.4 Summary 117

Single core processor has apparently hit physical limits of clock frequencies. To sustain the Moore's law, chip-makers now increase the number of processing units on a single chip, instead of increasing clock rate. These processing units are called cores; the chip is called multi-core processor.

However, an increased number of cores per chip at the same clock rate does not always speed up linearly applications that have already reached the limits of their parallel efficiency. Only applications that have excellent parallel efficiency can further exploit the power of multi-core processors. In this case, multiple independent data buffers can be processed simultaneously. However the parallelism in an application is not limited only by data dependencies but also by resource availability.

The aim of this chapter is to study the exploitation of inter-core parallelism or thread level parallelism, through recomputing in general and reverse computing in particular.

We study the limits of parallelism available in applications due to the memory space availability. We target multi-core processors with shared memory or shared register file structure like GPUs, and we show how we can increase thread level parallelism by increasing resources availability through recomputing to minimize memory space requirements. Our recomputing approach is more generic, it is also based on reverse computing with one or multiple instructions.

7.1 GPU Architecture and Programming Model

GPU is a massively parallel architecture initially intended for media application such 3D games or high-end 3D rendering. But modern GPUs have been designed to enable fundamental advances in processor performance and their highly parallel structure makes them more effective than general-purpose CPUs for a range of complex algorithms in both commercial and scientific fields. It is made possible by the addition of programmable stages and higher precision arithmetic making it, an ideal processor to accelerate a variety of data parallel applications. Figure 7.1 shows a block diagram of the NVIDIA Fermi Streaming Multiprocessor (SM). The NVIDIA, first who coined the terms graphics processing unit and GPU, Fermi architecture has been designed to support a broad range of application by featuring up to 448 CUDA cores. A CUDA core executes a floating point or integer instruction per clock for a thread. The 448 CUDA cores are organized in 14 Streaming Multiprocessors (SMs) of 32 cores each. Each CUDA processor has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU). Each SM has 16 load/store units, allowing source and destination addresses to be calculated for sixteen threads per clock. Four Special Function Units (SFUs) execute transcendental instructions such as sine, cosine, reciprocal, and square root. Each SFU executes one instruction per thread, per clock. Up to 16 double precision fused multiply-add operations can be performed per SM, per clock. Each SM has a 128 KB of shared register file and a 48 KB of shared memory. The SM schedules threads in groups of 32 parallel threads called warps. Each SM features two warp schedulers and two instruction dispatch units, allowing two warps to be issued and executed concurrently.

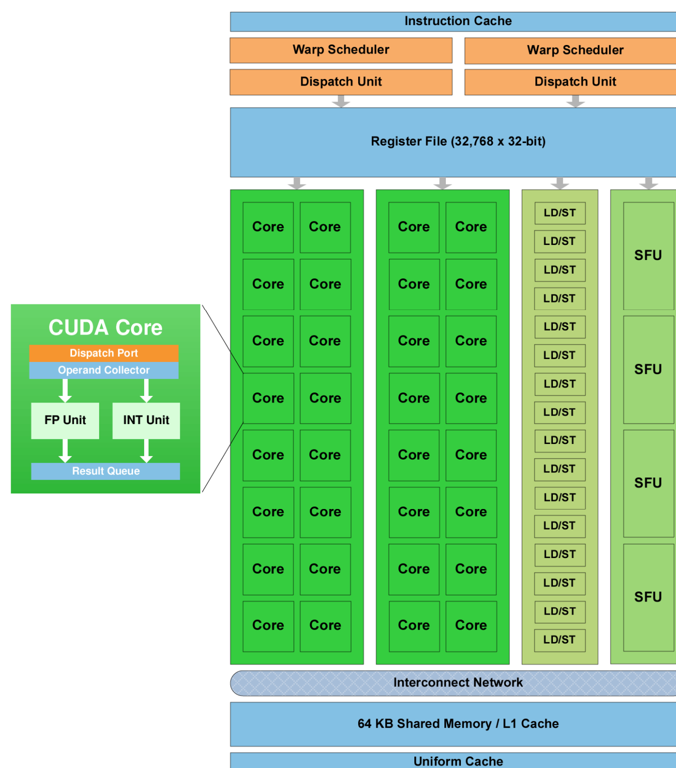


Figure 7.1: Fermi Streaming Multiprocessor (SM)

7.1.1 Memory Hierarchy

NVIDIA Fermi GTX 470 has three physical programmable memory spaces, which have different characteristics that affect the performance of applications depending of their usage.

Shared Register Files At the top level of the memory hierarchy are the shared register files. Each SM has 32 768-entry, 32-bit shared register file providing the fastest access to data possible at 1 clock cycle. This register file stores all data types - integer, floating-point, etc.

Shared Memory Fermi introduced 64 KB of on-chip memory that can be configured as 48 KB of shared memory used to facilitate reuse of data within the same thread block and 16 KB that can be requested as L1 cache memory. Latency of shared memory is 10 to 20 clock cycles . There is also the 768 KB L2 data cache shared by all SMs services for all load, store and Texture requests.

Global Memory Global memory has the greatest access latency, because it is off-chip. Its latency is roughly 40x higher than shared memory. There is a latency of 400 to 600 clock cycles to read data from global memory.

Local, Texture and Constant memory are all off-chip memories implemented in the global memory.

For existing applications that are bandwidth constrained, reducing the shared memory and register usage yields significant performance improvements. First, by avoiding higher-level memory usage, to automatically benefit from fast on-chip registers. Second, by increasing the multiprocessor warp occupancy to hide latency of the global memory by issuing independent instruction while waiting for the global memory access to complete.

We study the effects of register requirement via direct measurement of time.

7.1.2 Thread-Level Parallelism

The SM's 32 768-entry, 32-bit register file allows running 1024 threads simultaneously which is the maximum number of threads per block running on a single streaming multiprocessor. Several thread blocks may run on the same SM. If one thread block stalls while waiting for global memory, etc., then another thread block may run. This can be used to hide the latency. The number of registers and shared memory required by the kernel affects the number of threads per block and the number of thread blocks per streaming multiprocessor. If a thread uses more "variables" than registers allocated for each thread, the extra variables will automatically be spilled to a special region of global memory, called local memory, which may decrease performance. The optimization we found to be important, as Fermi runs most efficiently with a large numbers of threads, was a concept `"recomputing or rematerialization"`, that helps to minimize register requirements per thread. We mean that all registers of values that can be recomputed, they can be re-used. This increases register reuse and reduces register demands per thread that should increase the number of active threads per SM.

7.1.3 Register Usage, Rematerialization and Performance

Register pressure is a big issue for GPUs, and one way of limiting register pressure is to make sure we efficiently use data presented in different levels of memory hierarchy. Some times several data carry the same information, so no need to keep all of them in memory. Example $c{:=}a{+}b$, the information in *(a,b,c), (a,b), (a,c)* or *(b,c)* is the same, so no need to keep the three values live at the same time, because we can always compute one value from the two others. a can be retrieved from *(b,c)* and b from *(a,c)* by a simple subtraction. Hence, values that carry the same information can share the same register.

To achieve this goal, we use our technique of register rematerialization based on register reuse chains after register allocation, proposed in Chapter 3. Figure 7.2 shows an overview of the algorithm. It takes a GPU kernel as an input and constructs data dependence graphs (DDG) for the basic blocks. The measurement of register requirements uses a reuse DAG indicating which instruction can reuse a register used by a previous instruction. Once rematerialization decision is made, we proceed to the graph transformation of the original DDG. The rematerialization algorithm we propose is iterative, after each graph transformation we re-call the algorithm till there will be no rematerializable value.

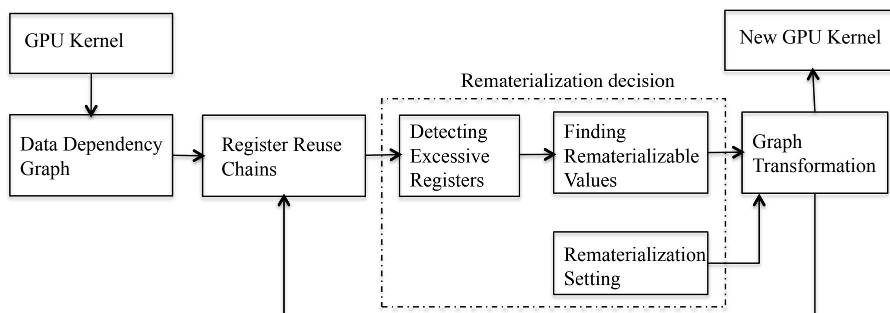


Figure 7.2: Overview of the rematerialization algorithm

We aim to study the impact of reverse rematerialization on performance and how additional operations could affect the execution time. We try also to find a trade-off between register usage, number of rematerialization instructions and performance. We have implemented the code of Figure 6.2(a) on NVIDIA GTX 470 as both opportunities of direct and reverse rematerialization exist. The body of the outer loop is a simulation of the example in Figure 5.4(a). We apply direct and reverse rematerialization till there will be no rematerializable value. In this example the register requirements using reverse rematerialization is lowered to two as shown in Figure 5.4(c). Figure 7.3 shows execution time as we vary the number of simultaneously live values. By comparing the two plots of Figure 7.3 we notice that the positive impact of reducing register usage per thread by using reverse rematerialization and rematerialization in general is not immediate. This can be explained by, first, the limited number of concurrently running threads per SM, 1024 on GTX 470 compared to the large number of registers, 32 768 32-bit registers, which allows assigning up to 32 registers per threads. So no speedup is expected with a number of live values less than 32. Second, the number of additional operations increases by increasing the number of simultaneously live values when the number of available registers is fixed, as shown in figure 7.4.

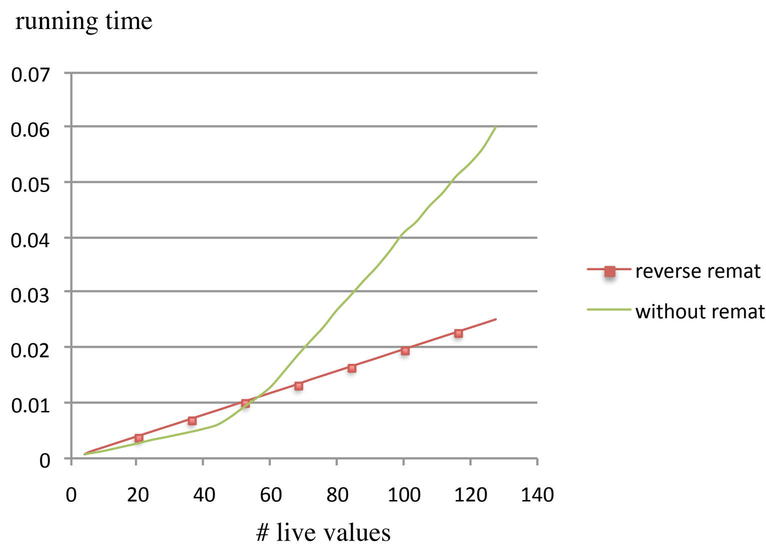


Figure 7.3: Reverse rematerialization : performance vs. register requirements

After the crossing of the two curves, we start getting speedup and when the number of live values is getting larger, the performance gap between the two versions is getting larger too. In this simulation, the register requirements per thread using rematerialization is constant, only the number of operations is increasing. The plot of reverse rematerialization is almost linear with the number of simultaneously live values because the register requirement and the number of active threads are constant and only the number of operations is increasing. Thus, there is no extra spill of variables or use of local memory compared to the original code.

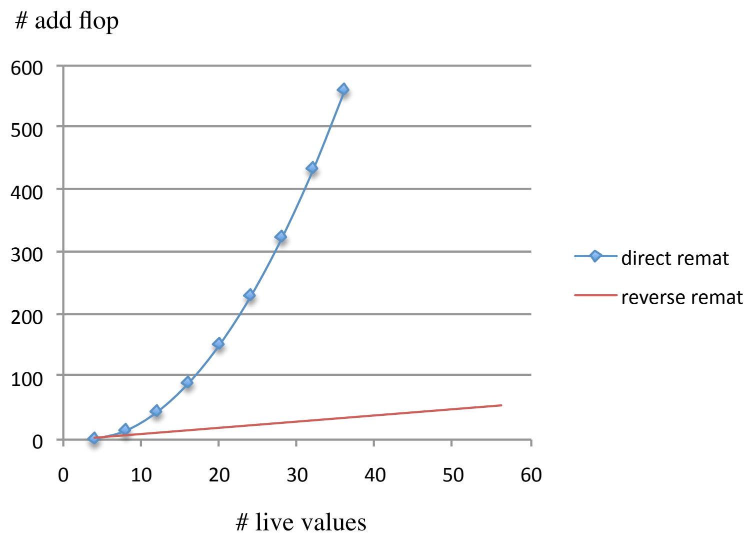


Figure 7.4: Cost of reversibility: additional operations

Now we choose a configuration with 48 simultaneously live values that requires 48 registers. We apply again reverse rematerialization and we vary the number of available registers in the algorithm. Knowing that the number of additional operations decreases by increasing the number of available registers and vice versa. Figure 7.5 shows that the execution time decreases gradually by increasing the number of available registers. However the optimal running time is reached when the number of registers is less than that required for the original version.

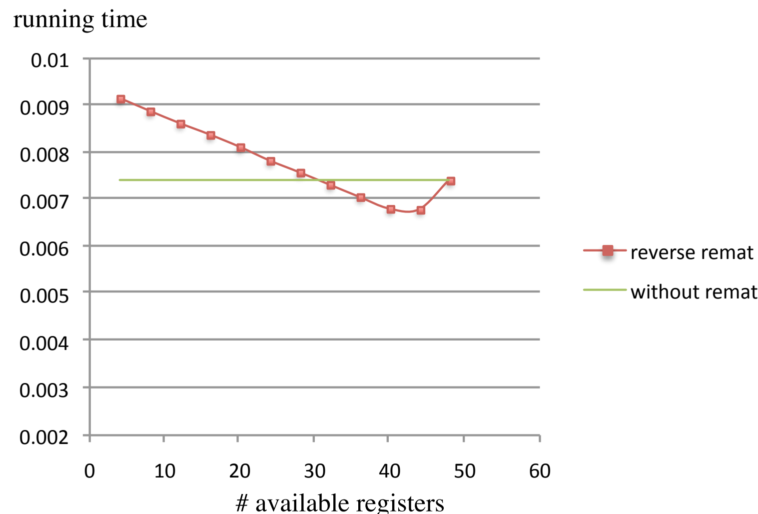


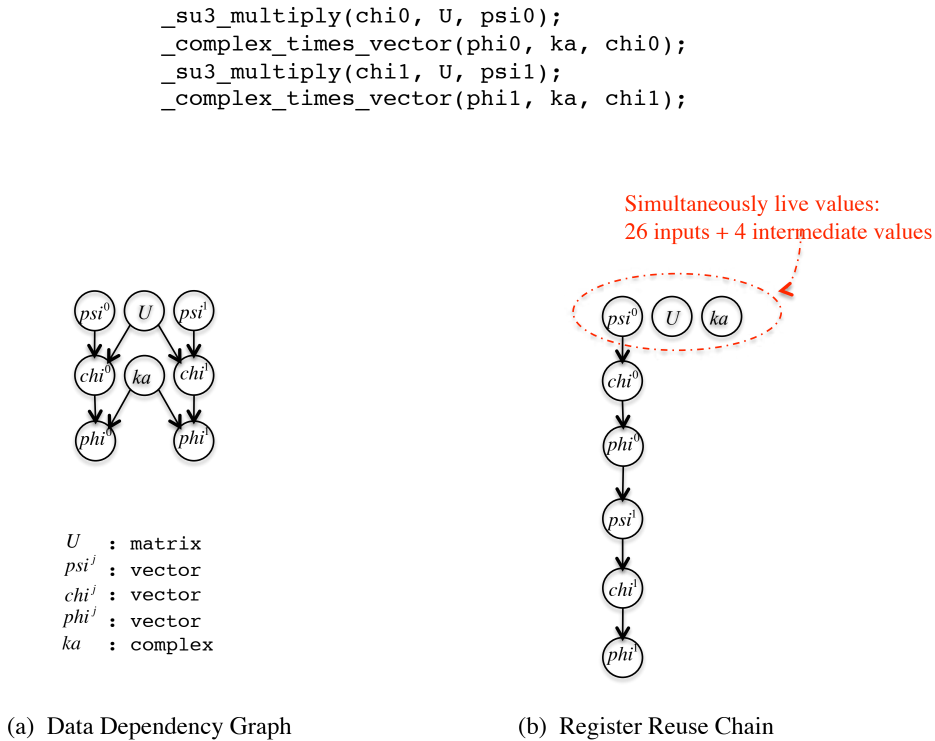
Figure 7.5: Reverse rematerialization : performance vs. available registers

In the future, in order to take advantage of this approach systematically, we would like to find ways in which a compiler can automatically predict with which rate reducing register demands per thread, using recomputing, will provide higher performance. This allows high performance CUDA programs to be built with minimum time and effort.

7.2 Experimental Results

This section presents our experiments in parallelizing LQCD application on NVIDIA GPU. We have taken the simplest approach in parallelizing across multiple cores of GPU, by assigning one thread to each lattice site. In other words, we have assigned one thread per loop iteration *Hopping_Matrix*.

We wrote a parser for the kernel that extracts basic blocks and builds their data dependency graphs. We use our tool to study and optimize this application. We give priority to reduce register requirements per thread without inserting any spill instruction. Figure 7.6 shows a code fragment of *Hopping_Matrix* and its corresponding data dependency graph.



(a) Data Dependency Graph (b) Register Reuse Chain

Figure 7.6: HMC code fragment: original implementation

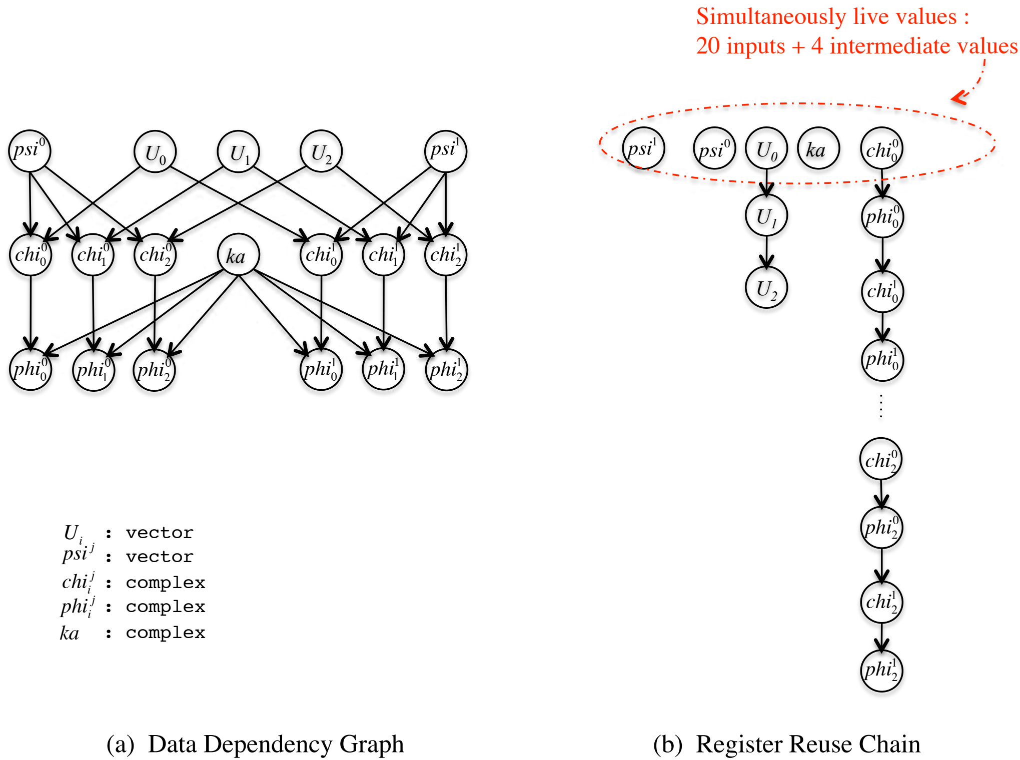
We divide the instructions into sub-instructions. The major improvement is reordering the computational instructions, so that, we decrease the number of simultaneously live values. A vector of matrix U is used for the computation of two complex of two different vectors. We compute the *i-th* complex of each vector instead of computing one vector after another, as shown in figure 7.7. Thus, a matrix U is never loaded entirely, but only the six values on each line that will be replaced by the next six values of the next line. Hence, we can save up to 6 registers in this code fragment.

As a result, we got a new version of *Hopping_Matrix* called *Hopping_Matrix_opt* that requires fewer registers than the original code. We call also the optimized versions of *Hopping_Matrix_k* and *Hopping_Matrix_l*, *Hopping_Matrix_k_opt* and *Hopping_Matrix_l_opt* respectively.

```
_vector_times_vector(chi00, U0, psi0);
_complex_times_complex(phi00, ka, chi00);
_vector_times_vector(chi01, U0, psi1);
_complex_times_complex(phi01, ka, chi01);

_vector_times_vector(chi10, U1, psi0);
_complex_times_complex(phi10, ka, chi10);
_vector_times_vector(chi11, U1, psi1);
_complex_times_complex(phi11, ka, chi11);

_vector_times_vector(chi20, U2, psi0);
_complex_times_complex(phi02, ka, chi20);
_vector_times_vector(chi21, U2, psi1);
_complex_times_complex(phi21, ka, chi21);
```



(a) Data Dependency Graph (b) Register Reuse Chain

Figure 7.7: HMC code fragment: code splitting and reordering - optimized implementation

In Chapter 3, we showed that rematerialization through reverse computing can reduce the register usage at a significant rate, so we performed this optimization for both original and optimized versions as described in Table 7.1 that shows a gain of 6% and 14% in register requirement for *Hopping_Matrix_k* and *Hopping_Matrix_k_opt* respectively, and an increase of 20% of the maximum number of threads per block in the optimized implementation using recomputing. This optimization increases the number of instructions of the GPU kernel by 7%, but recomputing using `reverse operations` from output operands amortizes the cost of recomputing to 4.68%. Even if recomputing increases the number of instructions, this should not penalize performance since threads of one warp are running in parallel comparing to the gain from reducing register demands that frees more space for more threads. The total space gain equals to register gain per thread times the number of threads per SM.

	min reg. req.	max thread/block
Hopping_Matrix_k	50	640
Hopping_Matrix_k_remat	47	640
Hopping_Matrix_k_opt	43	704
Hopping_Matrix_k_opt_remat	37	768
Hopping_Matrix_l	54	576
Hopping_Matrix_l_opt	50	640

Table 7.1: Contribution of reverse rematerialization to increase thread level parallelism

We measure the performance of the original and optimized code without and with using recomputing. Tables 7.2 and 7.3 present results for: (1) original code without any kernel optimization (2) code optimized using our register allocation tool. This result demonstrated the efficacy of recomputing to improve performance by increasing memory space availability and thread level parallelism. It provides improvements of 5.75% and 10.83% over original code for single and double precision respectively, and -1.33% and 10.60% over the optimized implementation for single and double precision versions respectively.

		without remat	with remat	%Perf
Hopping_Matrix_k	float	19.12	20.22	5.75%
	double	9.69	10.74	10.83%

Table 7.2: Contribution of reverse rematerialization to increase performance on NVIDIA GPU (gflops) - original implementation -

		without remat	with remat	%Perf
Hopping_Matrix_k_opt	float	20.19	19.92	-1.33%
	double	11.88	13.14	10.60%

Table 7.3: Contribution of reverse rematerialization to increase performance on NVIDIA GPU (gflops) - optimized implementation -

Our results in table 7.4 illustrate that on a NVIDIA GTX 470 GPU, recomputing can even improve single-thread performance. Among the four different single precision versions, applying recomputing on the original code takes the least amount of time which increases performance up to 22.9% to 0.14 gflops. When converting to double precision only an improvement of 6.4% can be observed by using recomputing on the original code. Our optimization has a negative impact on the optimized versions.

		without remat	with remat	% Perf
Hopping_Matrix_k	float	0.1163	0.1429	22.9%
	double	0.0565	0.0601	6.4%
Hopping_Matrix_k_opt	float	0.1115	0.1097	-1.6%
	double	0.0747	0.0737	-1.3 %

Table 7.4: Single thread performance on NVIDIA GPU (gflops)

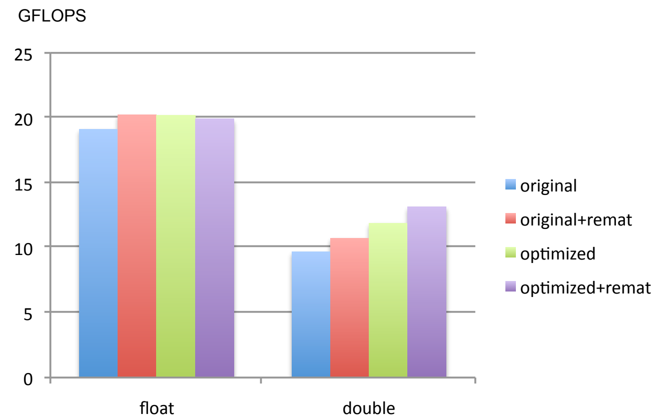


Figure 7.8: Reverse rematerialization: global performance on NVIDIA GPU (gflops)

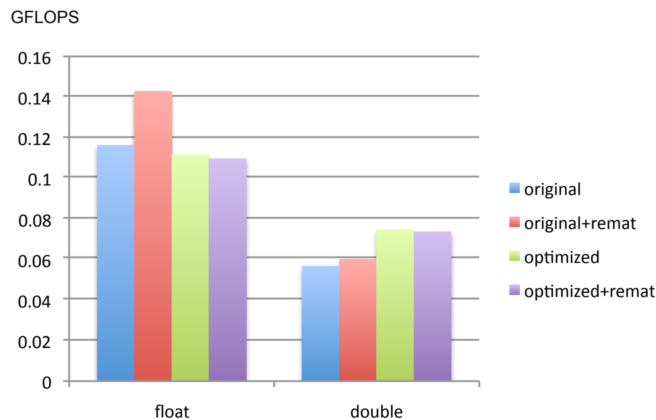


Figure 7.9: Reverse Rematerialization: single thread performance on NVIDIA GPU (gflops)

7.3 Analysis of Results

Table 7.5 shows a weak single-thread performance on NVIDIA GPU compared to Cell BE. This is justified by the slower memory, lower clock frequency and short vector data parallelism. The power of the GPU comes from that it can keep thousands of threads running concurrently. Constantly swapping different threads, with no context switch overhead, hides global memory latency and stall time due to dependencies among instructions, which is approximately 24 cycles. Thus, increasing the number of active threads per SM makes GPU able to run a lot of threads in essentially the same time as a single thread would execute as it is shown in Table 7.6.

NVIDIA GPU today demonstrates approximately twice the performance for single precision execution when compared to double precision. We know double precision rules in High Performance Computing for problems that require higher precision and to minimize the accumulation of round-off error.Because of lack of 64-bit registers, recomputing to reduce register usage has greater advantages for 64-bit floating point execution.

		Cell BE	NVIDIA GTX 470
Hopping_Matrix_k	float	2.82	0.14
	double	1.07	0.07
Hopping_Matrix_l	float	4.62	0.12
	double	1.87	0.03

Table 7.5: Cell BE Vs NVIDIA GPU: single thread performance (gflops)

		Cell BE	NVIDIA GTX 470
Hopping_Matrix_k	float	5.44	20.22
	double	2.68	13.14
Hopping_Matrix_l	float	15.61	25.92
	double	7.79	8.52

Table 7.6: Cell BE Vs NVIDIA GPU: global performance (gflops)

7.3.1 Limitations

Currently, our reverse rematerialization has several limitations. First, there are data precision issues especially with floating point operations and round-off problems. Next, division operation, the inverse of multiplication, when implemented in software, will require more non pipelined stages, and each will require temporary storage of a few intermediate results, so we only apply reverse rematerialization when we do not exceed the number of registers of the original operation.

7.4 Summary

We have shown experimental results for our approach using reverse computing based register rematerialization. A number of previous works have addressed the interaction between instruction scheduling and register allocation, but using reverse computing gives news degrees of freedom and we have shown that it can still improve performance despite an increase in instruction count. In our algorithm the maximum number of steps required for recomputing a value is a parameter hence in the future, the user could flexibly play with this parameter to find the best trade-off between number of threads, register count and additional instruction count

We have also shown that register allocation is still an important issue on the recent GPU processors as different threads share a common register file and the number of simultaneously concurrent threads depends on the number of registers of each individual thread. In this case too reverse computing helps gaining performance as it provides more opportunities for register rematerialization.

We will have to extensively check whether precision issues can be overcome, this will be done on the LQCD application that is a specially well adapted benchmark for that purpose as it requires very high precision at least in some parts.

Part IV

Conclusion

Chapter 8

Conclusion and Future Work

Contents

8.1 Conclusion 122
8.2 Future Work 123

8.1 Conclusion

The reversibility view of computation can help to improve time and space efficiency of programs. The goal of this thesis was to study the effects of information on time and space from the point of view of reversible computing. In particular, we have tackled three significant problems:

1. the trade-off between conservation of information and memory space.
2. the relationship between information and recomputing.
3. the effect of recomputing on the execution time and memory space requirements.

This work addresses these problems in the context of reversible computing. The basic idea of using reverse computing for improving performance is motivated by two observations. First, the relative cost of computing versus the memory access. Second, reversible computing favors the regeneration of data.

It is quite interesting to note that when trying to break today's barriers we have to rely on new hypotheses, for instance considering that computation is almost for free, specially when the gap between processor and memory speeds is increasing.

The work described in this thesis addresses three topics:

- studying the spatial complexity of reversible programs by designing a new technique for register allocation and instruction scheduling.
- using reverse recomputing to improve time and space efficiency of programs.
- practical works, including a variety of experiments on multi-core processor architectures.

In the first part, we have presented an analysis of the number of registers required to make a DAG computing graph reversible. We have defined the energetic garbage as the additional number of registers required for computing a forward and backward execution of the graph with respect to the simple forward execution. We gave a lower bound as the size of the decomposition of the graphs into elementary paths and through our experiments, we found that the garbage size does not exceed 50% of the program size - for DAG of unary/binary operations. However, values' lifetime is shorter in a direct computation.

In the second part, we have presented reverse rematerialization, a novel method for reducing register pressure which takes advantage of the relative cost of computing versus memory access. We have found that there may be more opportunities for recomputing a value in reverse path from output operands than recomputing it from its original input operands. We also showed that reverse rematerialization may increase instruction-level parallelism and thread-level parallelism by increasing resource availability.

For the experiments, we first implemented a graph generator for generating connected and directed acyclic graphs. The graph generator can generate all graphs of a given size or random graphs of different sizes. The graphs have been used to test the effectiveness of our reverse register allocation algorithm. We have developed a register allocation technique for reversible execution of programs. The allocator computes also the energetic garbage defined as the difference between the number of registers required for direct execution and those required for reversible execution. We have developed a technique based on register reuse chains for measuring the register requirements in basic blocks. We developed a new

rematerialization technique based on register reuse chains through reverse computing. We ported the LQCD simulation code on Cell BE and NVDIA GPU and we applied our optimization techniques on this code. The results demonstrated the benefit of the optimizations for decreasing spill code and improving both instruction-level parallelism and thread-level parallelism on both architectures, which are critical for achieving good performance. Finally, experiments showed that the performance depends on information locality rather than data locality.

8.2 Future Work

Despite our efforts, we are aware that more systematic experiments should be done and more benchmarks should be used in order to prove that rematerialization based reverse computing is effective in the general case. But we believe that as the gap between communication/memory latency and CPU latency increases we possibly will have to consider that computation is for free and only communication matters. Recomputing together with reverse recomputing may then be an alternative to communication/memory access, but at the price of opening the precision issue, a new trade-off that we believe is worth being a topic for future research in high performance computing.

The other next step would be to apply recomputing in general and reverse recomputing in particular to communication - recompute rather than communicate since the inter-processor communication seriously affects the performance of parallel processing.

We would also like to find ways in which a compiler can automatically predict the good parameters of reverse rematerialization in order to take advantage of this approach systematically. This allows high performance programs to be built with minimum time and effort.

In addition, it is quite interesting to integrate our register allocation algorithm for reversible programs presented in Section 3.1 in a reversible compiler like the RCC -Georgia Tech's reversible C code compiler [39] in order to compare the size of generated codes with existing reversible programs and memory usage.

Finally, we aim also to continue studying the thermodynamics aspect of computation and try to understand more deeply the program entropy.

Bibliography

[1] Tilak Agarwala and John Cocke. High performance reduced instruction set processors. *Technical Report #55845*, March 31 1987.

[2] M. Josephine Ammer, Michael Frank, Tom Knight, Nicole Love, Norm Margolus, Carlin Vieri, and Margolus Carlin Vieri. A scalable reversible computer in silicon, 1998.

[3] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 243–253, New York, NY, USA, 2001. ACM.

[4] Mouad Bahi and Christine Eisenbeis. Spatial complexity of reversibly computable dag. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems - CASES*. ACM, 2009.

[5] Mouad Bahi and Christine Eisenbeis. High performance by exploiting information locality through reverse computing. In *International Symposium on Computer Architecture and High Performance Computing - SBAC-PAD*. IEEE Computer Society, 2011.

[6] Mouad Bahi and Christine Eisenbeis. Rematerialization-based register allocation through reverse computing. In *International Conference on Computing Frontiers - CF*. ACM, 2011.

[7] H. G. Baker. *NREVERSAL of fortune - the thermodynamics of garbage collection.* Int'l Workshop on Memory Mgmt, 1992.

[8] Charles H. Bennett. Time/space trade-offs for reversible computation. *SIAM J. Comput.*, 18:766–776, August 1989.

[9] Charles H. Bennett, Péter Gács, Ming Li, Paul M. B. Vitányi, and Wojciech H. Zurek. Information distance. *IEEE Transactions on Information Theory*, 44(4):1407–1423, 1998.

[10] D. H. Bennett. *Logical reversibility of a computation.* IBM J. Res. Dev., 1973.

[11] David Berson, Rajiv Gupta, and Mary Lou Soffa. Ursa: A unified resource allocator for registers and functional units in vliw architectures. In *In Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 243–254, 1992.

[12] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. Ursa: A unified resource allocator for registers and functional units in vliw architectures. In *PACT '93: Proceedings of the IFIP WG10.3. Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 243–254, Amsterdam, The Netherlands, The Netherlands, 1993. North-Holland Publishing Co.

[13] P. G. Bishop. Using reversible computing to achieve fail-safety. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, ISSRE '97, pages 182–, Washington, DC, USA, 1997. IEEE Computer Society.

[14] Florent Bouchez. *A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases.* PhD thesis, ENS Lyon, 2009.

[15] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. *SIGPLAN Not.*, 39:283–294, April 2004.

[16] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 311–321, New York, NY, USA, 1992. ACM.

[17] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16:428–455, May 1994.

[18] A. Shindlerb C. Urbacha, K. Jansenb and U. Wenger. Hmc algorithm with multiple time scale integration and mass preconditioning. *Computer Physics Communications*, 2006.

[19] Christopher D. Carothers, Kaylan S. Perumalla, and Richard M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. In *Proceedings of the thirteenth workshop on Parallel and distributed simulation*, PADS '99, pages 126–135, Washington, DC, USA, 1999. IEEE Computer Society.

[20] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–105, New York, NY, USA, 1982. ACM.

[21] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

[22] Frederick Chow and John Hennessy. Register allocation by priority-based coloring. *SIGPLAN Not.*, 19:222–232, June 1984.

[23] John Collier. Two faces of Maxwell's demon reveal the nature of irreversibility. *Studies In History and Philosophy of Science Part A*, 21(2), 1990.

[24] G.W. Dueck D. Maslov. *Reversible cascades with minimal garbage.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2004.

[25] R. P. Dilworth. *A decomposition theorem for partially ordered sets.* Ann. of Math., 1950.

[26] Fredkin Edward and Toffoli Tommaso. Conservative logic. *International Journal of Theoretical Physics*, 21:219–253, 1982.

[27] Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol. The meeting graph: a new model for loop cyclic register allocation. In *Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*, PACT '95, pages 264–267, Manchester, UK, 1995. IFIP Working Group on Algol.

[28] Richard Feynman. Quantum mechanical computers. *Optics News*, 11:11–20, 1985.

[29] M. P. Frank. *The R programming language and compiler*. MIT RC Proj. Memo #M8, 1997.

[30] Michael P. Frank. The physical limits of computing. *Computing in Science and Engg.*, 2002.

[31] Changqing Fu and Kent Wilken. A faster optimal register allocator. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, pages 245–256, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[32] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[33] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18:300–324, May 1996.

[34] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocations using 01 integer programming. *Softw. Pract. Exper.*, 26:929–965, August 1996.

[35] C. Leung H. Yang, G. R. Gao. On achieving balanced power consumption in software pipelined loops. In *CASES*, 2002.

[36] M. Iri, K. Tanabe, K. Academic, and A. Griewank. On automatic differentiation. In *in Mathematical Programming: Recent Developments and Applications*, 1989.

[37] R. M. Frederic J. V. Carlin, F. K. Thomas. *Pendulum: A Reversible Computer Architecture*. Master's thesis, MIT Artificial Intelligence Laboratory, 1995.

[38] E. T. Jaynes. Gibbs vs Boltzmann Entropies. *American Journal of Physics*, 33(5):391–398, 1965.

[39] k. Perumalla and R. Fujimoto. *Source-code transformations for efficient reversibility*. Technical Report GIT-CC-99-21, College of Computing, Georgia Tech., 1999.

[40] Richard M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[41] D. E. Knuth. An empirical study of fortran programs. In *Software: Practice and Experience, Volume 1, Issue 2*, pages 105–133, 1971.

[42] A. N. Kolmogorov. Three approaches to the definition of the concept "quantity of information". In *Problemy Peredaci Informacii*, volume 1, pages 3–11, 1965.

[43] Brian Lambson, David Carlton, and Jeffrey Bokor. Exploring the thermodynamic limits of computation in integrated systems: Magnetic memory, nanomagnetic logic, and the landauer limit. *Phys. Rev. Lett.*, 107:010604, Jul 2011.

[44] R. Landauer. *Irreversibility and heat generation in the computing process.* IBM Journal of Research and Development, 1961.

[45] C. Lutz and H. Derby. *Janus: a time-reversible language.* Caltech class project, 1982.

[46] Richard H. Dittman Mark Waldo Zemansky. *Heat and thermodynamics: an intermediate textbook, 7th edition.* McGraw-Hill, New York, USA, 1997.

[47] P. Matherat and M. T. Jaekel. *Logical Dissipation of Automata Implements - Dissipation of Computation.* Technique et Science Informatiques, 1996.

[48] P. Matherat and M. T. Jaekel. What about the "dissipation of computation" question ? a return to bennett. pages 690–713, 2007.

[49] M.A. Perkowski M.H.A. Khan. Logic synthesis with cascades of new reversible gate families. *6th International Symposium on Representation and Methodology of Future Computing Technology*, pages 43–55, March 2003.

[50] Alan Mishchenko and Marek Perkowski. Logic synthesis of reversible wave cascades, June 2002.

[51] U. Naumann. *On optimal DAG reversal.* Technical Report AIB-2007-05, 2007.

[52] A. Nicolau and J. A. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE Trans. Comput.*, 33:968–976, November 1984.

[53] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. *ACM Trans. Program. Lang. Syst.*, 26:735–765, July 2004.

[54] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21:895–913, September 1999.

[55] Mukta Punjani. Register rematerialization in gcc. In *GCC Developers' Summit 2004*, 2004.

[56] E. Gioan S. Burckel. *In Situ Design of Register Operations.* ISVLSI, 2008.

[57] J. Y. Chung S. K. Chen, W. K. Fuchs. Reversible debugging using program instrumentation. *IEEE Trans. Softw. Eng.*, 2001.

[58] Claude E. Shannon and Warren Weaver. *A Mathematical Theory of Communication.* University of Illinois Press, Champaign, IL, USA, 1963.

[59] Vivek V. Shende, Aditya K. Prasad, Igor L. Markov, and John P. Hayes. Reversible logic circuit synthesis. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, ICCAD '02, pages 353–360, New York, NY, USA, 2002. ACM.

[60] Loren Taylor Simpson. *Value-driven redundancy elimination.* PhD thesis, Rice University, Houston, TX, USA, 1996.

[61] N. Doi T. Koju, S. Takada. An efficient and generic reversible debugger using the virtual machine based approach. In *VEE*, 2005.

[62] L. Kent Thomas, Thomas N. Dixon, and Ray G. Pierson. Fractured reservoir simulation. *SPE Journal*, 23(1):42–54, 1983.

[63] G.S. Tjaden and M.J. Flynn. Detection and parallel execution of independent instructions. *IEEE Transactions on Computers*, 19:889–895, 1970.

[64] Tommaso Toffoli. Reversible computing. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 632–644, London, UK, 1980. Springer-Verlag.

[65] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.*, 11:25–33, January 1967.

[66] Sid-Ahmed-Ali Touati. *Register Pressure in Instruction Level Parallelism.* PhD thesis, Université de Versailles, France, June 2002.

[67] Omri Traub, Glenn H. Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *PLDI*, pages 142–151, 1998.

[68] Paul M. B. Vitanyi. *Time, space, and energy in reversible computing.* Conf. Computing Frontiers, 2005.

[69] Alexis De Vos and Stijn De Baerdemacker. Symmetry groups for the decomposition of reversible computers, quantum computers, and computers in between. *Symmetry*, 3(2):305–324, 2011.

[70] P. Vranas, M. A. Blumrich, D. Chen, A. Gara, M. E. Giampapa, P. Heidelberger, V. Salapura, J. C. Sexton, R. Soltz, and G. Bhanot. Massively parallel quantum chromodynamics. *IBM J. Res. Dev.*, 52:189–197, January 2008.

[71] Tao Zhang, Xiaotong Zhuang, and Santosh Pande. Compiler optimizations to reduce security overhead. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 346–357, Washington, DC, USA, 2006. IEEE Computer Society.

[72] Yukong Zhang, Young-Jun Kwon, and Hyuk Jae Lee. A systematic generation of initial register-reuse chains for dependence minimization. *SIGPLAN Not.*, 36(2):47–54, 2001.