ALOCAÇÃO DE REGISTRADORES DESACOPLADA BASEADA EM COLORAÇÃO DE GRAFOS COM COMPARTILHAMENTO HIERÁRQUICO

ANDRÉ LUIZ CAMARGOS TAVARES

ALOCAÇÃO DE REGISTRADORES DESACOPLADA BASEADA EM COLORAÇÃO DE GRAFOS COM COMPARTILHAMENTO HIERÁRQUICO

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Mariza Andrade da Silva Bigonha
Co-Orientador: Fernando Magno Quintão Pereira

Belo Horizonte

26 de janeiro de 2011

ANDRÉ LUIZ CAMARGOS TAVARES

DECOUPLED GRAPH-COLORING REGISTER ALLOCATION WITH HIERARCHICAL ALIASING

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Mariza Andrade da Silva Bigonha
Co-Advisor: Fernando Magno Quintão Pereira

Belo Horizonte
January 26, 2011

Tavares, André Luiz Camargos
T231a Decoupled Graph-Coloring Register Allocation with Hierarchical Aliasing / André Luiz Camargos Tavares. — Belo Horizonte, 2011
xviii, 77 f. : il. ; 29cm

Dissertação (mestrado) — Federal University of Minas Gerais
Orientador: Mariza Andrade da Silva Bigonha

Co-Orientador: Fernando Magno Quintão Pereira

I. Orientador II. Co-Orientador. III. Título.

CDU 519.6*33 (043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Alocação de registradores desacoplada baseada em coloração de grafos com compartilhamento hierárquico

ANDRE LUIZ CAMARGOS TAVARES

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROFA. MARIZA ANDRADE DA SILVA BIGONHA - Orientadora
Departamento de Ciência da Computação - UFMG

PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Co-orientador
Departamento de Ciência da Computação - UFMG

PROF. FABRICE RASTELLO
École Normale Supérieure de Lyon/INRIA

PROF. RENATO ANTÔNIO CELSO FERREIRA
Departamento de Ciência da Computação - UFMG

PROF. ROBERTO DA SILVA BIGONHA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 26 de janeiro de 2011.

Acknowledgments

I would like to thank everyone that participated in this thesis. I am heartily thankful to my co-advisor, Fernando Pereira, whose encouragement, guidance and support from the initial to the final level enabled me to develop an understanding of the subject. I also thank my advisor Mariza Bigonha and Roberto Bigonha for their support and belief in my work. Many thanks go to all my lab colleagues, for many happy moments and fruitful discussions. They are Andrei Rimsa, Rodrigo Sol, Ricardo Terra, César Couto, Giselle Machado, Leonardo Reis, and many others.

Many thanks goes to Fabrice Rastello, who believed in me and gave me an extraordinary opportunity to spend four months at his lab at Lyon, France. I would like to show my gratitude to Christophe Guillon for many fruitful discussions. For my french lab colleagues – Quentin Colombet, Benoit Boissinot and Florian Brandner – I would like to say: *merci pour vos contributions dans ma thèse, pour tous les moments agréables, et pour tous les fromages et du vin français nous avons partagé, spécialement la raclette de Chamonix.*

Last but not least, I would like to show my deepest gratitude to my beloved bride and future wife, Laura Gallo. She has made available her support in a number of ways in another important achievement of my life. Thanks to my mother, my step-father, Eduardo Gribel, my family and friends, which I prefer to not cite by name because they are many, for their unconditional support.

Resumo

Resultados recentes demonstram como fazer alocação de registradores baseada em coloração de grafos que desacopla o derramamento da atribuição de registradores. A abordagem desacoplada tem duas vantagens: primeiro, simplifica os algoritmos de alocação de registradores. Segundo, pode-se manter mais variáveis em registradores, em vez de enviá-las para a memória. Apesar dessas vantagens, o modelo desacoplado usando grafos de interferência, como descrito por trabalhos anteriores, não leva em consideração o compartilhamento de registradores, um fenômeno presente em arquiteturas como x86, ARM e SPARC. Um obstáculo importante é o fato de que os algoritmos desacoplados existentes fazem uma extensiva divisão do tempo de vida das variáveis para tratar de compartilhamento, o que aumenta os grafos de entrada por um fator quadrático. Tais alocadores são ineficientes em termos de consumo de memória, tempo de compilação e qualidade de código produzido.

Esta dissertação introduz técnicas para contornar esse obstáculo. É descrito um teste de derramamento para arquiteturas com compartilhamento melhor que o tradicional teste de simplificação de Kempe. Foi usada uma heurística para fundir – ou evitar a divisão – tempo de vida sempre que possível, e foram adaptados conhecidos algoritmos de fusão de registradores. Para se determinar a melhor representação intermediária para as técnicas descritas, as seguintes opções foram estudadas: *Static Single Assignment* (SSA), *Static Single Information* (SSI), *extended SSA* (e-SSA) e *Elementary Form*. Nesse processo foi desenvolvido um algoritmo para eficientemente criar SSI e e-SSA.

Os resultados foram validados empiricamente ao mostrar que as técnicas descritas melhoram dois conhecidos alocadores de registradores baseados em coloração de grafos para arquiteturas com compartilhamento, são eles: a extensão de Smith *et al.* [SRH04] do *Iterated Register Coalescer* (IRC) [GA96]; e o método de força bruta do Bouchez *et al.* [BDR08]. Ao se executar as técnicas no SPEC CPU 2000, foi possível reduzir o tamanho dos grafos de interferência dos alocadores por um fator de 4, e a qualidade do IRC foi melhorada, em termos de cópias inseridas no programa assembly, de 1.5%

para 0.54%.

Palavras-chave: Compiladores, Alocação de Registradores, Fusão de Registradores, Compartilhamento de Registradores, Desacoplada.

Abstract

Recent results have shown how to do graph-coloring-based register allocation in a way that decouples spilling from register assignment. This decoupled approach has two main advantages: first, it simplifies register allocation algorithms. Second, it might keep more variables in registers, instead of sending them to memory. In spite of these advantages, the decoupled model using the graph coloring approach, as described in previous works, do not handle register aliasing, a phenomenon present in architectures such as x86, ARM and Sparc. An important obstacle is the fact that existing decoupled algorithms have to perform extensive live range splitting to deal with aliasing, increasing the input graphs by a quadratic factor. Such allocators would be inefficient in terms of memory consumption, compilation time and the quality of the code they produce.

In this thesis we introduce a number of techniques that overcome this obstacle. We describe a spill test that deals with aliasing better than Kempe's traditional simplification test. We use heuristics to merge – or rather avoid splitting – live ranges whenever possible, and we adapt well-known coalescing tests to the world of aliased registers. In order to determine the best interference representation for the techniques, we have studied the following options: Single Static Assignment (SSA), Single Static Information (SSI), extended SSA (e-SSA) and Elementary Form. In this process we have developed an algorithm to efficiently create SSI and e-SSA.

We have empirically validated our results by showing how our techniques improve two well known graph coloring based allocators that deal with aliased registers, namely Smith *et al.*'s extension [SRH04] of the Iterated Register Coalescer (IRC) [GA96], and Bouchez *et al.*'s brute force (BF) method [BDR08]. Running our techniques on a subset SPEC CPU 2000, we have been able to reduce the size of the interference graphs that the allocators would require by a factor of 4, and we have improved the quality of IRC, in terms of proportion of copies left in the assembly program, from 1.5% to 0.54%.

Keywords: Compiler, Register Allocation, Register Coalescing, Aliasing, Decoupled.

List of Figures

Contents

Chapter 1

Introduction

Register allocation is the problem of finding storage locations to the values manipulated by a program. Traditional computer architectures provide two storage alternatives: memory or registers. Registers are much faster, yet, they come in very small numbers. For instance, the 32-bit x86 chip contains only seven general purpose registers. Register allocation is an important compiler optimization. In the words of Hennessy and Patterson: "Register Allocation adds the largest single performance improvement to compiled programs" [HP02]. As an example, the difference, in terms of program execution time, between a trivial allocator, that sends every variable to memory, and an optimal, integer linear based approach, can be as high as 250% [NPP07].

Many recent register allocation algorithms follow a *decoupled approach* that separates spilling from register assignment [AG01; HG08; HGG06; PP05; PP08; Ron09; SB07; WF10]. Spilling is a step in the register allocation algorithm, which moves to memory those variables that could not be stored in registers. This model has important advantages. First, the separation between these two phases tends to yield simpler and more modular implementations: different spilling heuristics can easily be combined with different strategies to do register assignment and coalescing. Second, as we will illustrate in Chapter 3, decoupled designs have more flexibility to assign registers to variables; hence, are more successful at avoiding spilling. A key factor behind the decoupled model is the concept of *live range splitting*, which allows allocating a variable to different registers along distinct parts of its live range. Although so fundamental, this very notion of live range splitting makes it difficult to extend decoupled algorithms to computer architectures with aliased register banks.

Quoting Smith *et al.*, "two register names alias when an assignment to one register name can affect the value of the other" [SRH04]. Aliasing characterizes four general purpose registers present in the x86: AX, BX, CX and DX. Each of these 16-bit

registers is divided in two sub-registers, which can be used independently on each other. Aliasing is also present in the floating point registers seen in ARM, PowerPC, and Sparc. In these cases, we can combine two floats into a double precision register. Architectures such as ARM Neon go further, allowing the combination of two doubles into a quad-precision register.

Decoupled register allocation relies on the fact that once we lower the register pressure at any given *program point* to less than K – the number of available registers – then no further spilling will be necessary during the assignment of registers to variables. We define a *program point* as any point between two consecutive program instructions. Thus, a decoupled allocator requires the property which states that the maximum *register pressure* at any program point be equal to the global register pressure. The register pressure of a program point, in our case, is the minimum number of registers necessary to allocate the variables alive in that region. An important result in compiler theory [Bou05; BDJS06; HG06; PP05] is that for architectures with no aliased registers we can satisfy this property by converting the source program into SSA-form [CFR+91]. This conversion is a way of doing live range splitting. However, in face of aliasing, the SSA transformation is not enough: aliased register allocation is NP-complete, even for straight-line programs in SSA form [LPP07].

Aliasing requires a more extensive kind of live range splitting: the conversion to *elementary-form* [PP08], which splits live ranges between each pair of consecutive instructions. There is one decoupled algorithm that deals with aliasing – register allocation by puzzle solving [PP08]. This is a fast allocator, that traverses the dominator tree of a program, converting it to elementary form on the fly, while visiting each instruction. Although good at avoiding spilling, the puzzle solver tends to insert many copies in the assembly program that it produces, because it has a local view of the source code, i.e, it only recognizes the variables alive across a single instruction at each time.

There exist register coalescers that deal with aliasing and see the program as a whole, but they are not decoupled algorithms. Among these allocators we cite integer linear programming formulation [KW98], PBQP approach [SE02] and extensions to graph coloring algorithms [SRH04]. These algorithms demand new techniques to decouple spilling from register assignment, because it is difficult to adapt these methods to handle elementary form programs. The elementary form conversion increases by a quadratic factor the number of variables in the source program, and the traditional approaches do not scale up. This project aims at filling this gap.

The objective of this thesis is to describe a suite of techniques that make decoupled graph coloring register allocation feasible and useful. In other words, we describe

register allocators that:

1. can use live range splitting to avoid spilling;
2. are able to see the whole program, and the many dependencies between copy instructions, to do better register coalescing.

We have adapted two different allocators to follow the decoupled model: the Iterated Register Coalescer [GA96], with extensions by Smith *et al.* [SRH04], and Brute Force coalescer [BDR08]. Relying on the idea of *conservative live range merging*, we mitigate the negative effects of live range splitting, while keeping its advantages.

1.1 Contributions

The main contributions of this thesis are:

1. A modification to the graph coloring register allocators to decouple spilling from register assignment in the presence of aliasing.
2. A pre-processing technique that avoids creating huge graphs during the conversion of a program into elementary form.
3. An adaptation of the Brute Force coalescer of Bouchez *et al.* [BDR08] to deal with aliasing.
4. An adaptation of the Iterated Register Coalescing of Appel and George [AG01] to a decoupled approach.
5. Comparison and evaluation of Brute Force Coalescer and Iterated Register Coalescing in face of aliasing and using a decoupled approach.
6. Developed an algorithm to efficiently create SSI and e-SSA.

1.2 Text Organization

Following we describe how this thesis is organized:

- Chapter 2 presents in details four intermediate representations used in compilers and register allocation. They are: SSA [CFR+91], SSI [Ana99], e-SSA [BGS00] and Elementary Form [PP08].

- Chapter 3 describes the main algorithms present in the literature for register allocation and coalescing.
- Chapter 4 presents the contributions of this thesis.
  - Section 4.1: A modification to the graph coloring register allocators to decouple spilling from register assignment in the presence of aliasing.
  - Section 4.2: An adaptation of the Brute Force coalescer of Bouchez *et al.* [BDR08] to deal with aliasing.
  - Section 4.3: A pre-processing technique that avoids creating huge graphs during the conversion of a program into elementary form.
- Chapter 5 presents the experiments realized to substantiate the evaluation of this work.
- Chapter 6 summarizes our contributions and concludes this work.

Chapter 2

Intermediate Representations

The Intermediate Representation (IR) may have many forms. Aho *et al.* present the most used IR [ALSU06]: syntax tree, pseudo-assembly and three address code. In this chapter we describe three different three address code representation commonly used by compilers. They are Static Single Assignment (SSA) [CFR+91], Static Single Information (SSI) [Ana99] and Elementary Form [AG01].

The intermediate representation at register allocation is relevant to the quality of the allocation. Even more, there are some properties in these representations that are fundamental for some register allocation approaches. While SSA-form programs have chordal interference graphs, the interference graphs of SSI-form programs are interval graphs and the interference graph of an elementary program is an elementary graph. Spill-free register allocation is NP-complete for general programs [CAC+81] because coloring general graphs is NP-complete. However, for architectures without aliasing this problem has a polynomial time solution for SSA-form programs [Bou05; BDMS05; HGG06] because chordal graphs can be colored in polynomial time [Mar06]. For architectures with aliasing it is necessary to have an elementary graph to allocate registers in polynomial time. In this chapter we describe these representations and in Chapter 3 we show how they relate to register allocation.

2.1 Static Single Assignment

Static Single Assignment (SSA) form is a program representation introduced by Alpern *et al.* [AWZ88] and Rosen *et al.*, [RWZ88]. Later, Cytron *et al.* [CFR+91] formally described the intermediate representation and proposed construction algorithms.

The main concept of SSA is that every variable has a single definition. However it is not sufficient only to rename variables. There are cases where two different definitions

of the same variable reaches the same use. For these cases a special instruction is used, the ϕ-functions. The ϕ-functions are an abstraction used to join the live ranges of variables. An assignment such as:

$$(v_1, \ldots, v_n) = \phi[(v_{11}, \ldots, v_{n1}) : L_1, \ldots (v_{1m}, \ldots, v_{nm}) : L_m]$$

contains n ϕ-functions such as $v_i \leftarrow \phi(v_{i1} : L_1, \ldots, v_{im} : L_m)$. The ϕ symbol works as a multiplexer. It will assign to each v_i the value in v_{ij}, where j is determined by L_j, the basic block last visited before reaching the ϕ assignment. Notice that these assignments happen in parallel, that is, all the variables $v_{1i}, \ldots, v_{ni}$ are simultaneously copied into the variables $v_1, \ldots, v_n$. Figure 2.1 shows an example of the SSA representation. The original program presented in Figure 2.1 (a) is not in SSA form, as variables x and y are defined twice in this example. The SSA version of this program, showed in Figure 2.1 (b), renamed both variables and created a ϕ-function for variable y. This ϕ-function is necessary since the live range of the y_1 and y_2 are disjoint.

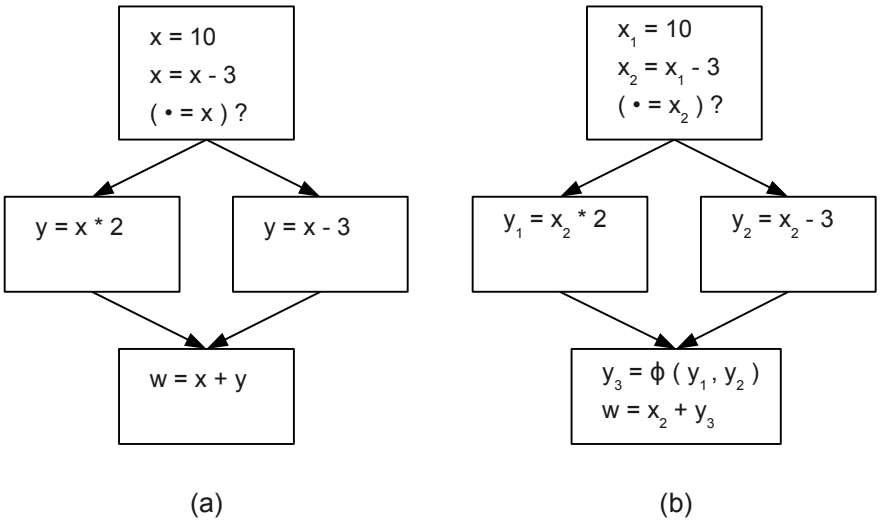

Figure 2.1: Example of the use of SSA on information analysis. (a) Original program. (b) SSA program.

As examples of SSA clients we have a more efficient algorithm to eliminate dead code and global value numbering optimization. Dead code elimination is a compiler optimization that removes code that cannot be reached and also operations related to dead variables, which have no effect on the program. This optimization reduces the code size, which can be important in some contexts, and also can reduce the execution time, by removing unnecessary operations. This optimization was historically performed by data-flow analysis [ALSU06; Muc97], however after SSA conversion approach, the

algorithm was improved and is now based on the control flow graph.

An optimization that depends on SSA is Global Value Numbering (GVN) [Sim96]. GVN evaluates variables assignments and expressions and defines a value range for each variable. The value range of a variable can be later used for other optimizations. For instance, in case the range of a variable is a single value, all uses of this variable can be replaced by this value. The range can also be used to prove that some conditional branches may always take a single decision, allowing to remove the other option of the branch.

2.2 Static Single Information

Static Single Information (SSI) form is a program representation introduced by Ananian [Ana99]; however, program representations with similar properties have been described before by Johnson and Pigali [JP93]. Also, the SSU-form (Static Single Use), described by Plevyak96 [Ple96] in his Ph.D dissertation seems to be equivalent to SSI, although we cannot verify this claim due to the lack of a formal specification of SSU. Benoit *et al.* distinguish two main flavors of the SSI form [BBDR09], which are not equivalent: *strong*, introduced by Ananian [Ana99] and *weak*, described by Singer [Sin06]. According to Boissinot *et al.*, four properties characterize strong SSI form:

- *pseudo-definition*: there exists a definition of each variable at the starting point of the program's control flow graph.
- *single reaching-definition*: each program point is reached by at most one definition of each variable.
- *pseudo-use*: there exists a use of each variable at the ending point of the program's control flow graph.
- *single upward-exposed-use*: from each program point it is possible to reach at most one use of a variable, without passing by a previous use.

The *weak* SSI form differs from the *strong* SSI on the last property presented. While *strong* SSI has the single upward-exposed-use property, *weak* SSI has the post-dominance property. This property defines that each use of a variable must post-dominate its definition. Any strong SSI form program is also a weak SSI form program.

Figure 2.2(a) shows a program written in a C like language, and Figure 2.2(b) gives the control flow graph of this program, in SSI form. The program in Figure 2.2(c) is not in SSI form, as it contains a point exposed to two different uses of a_2.

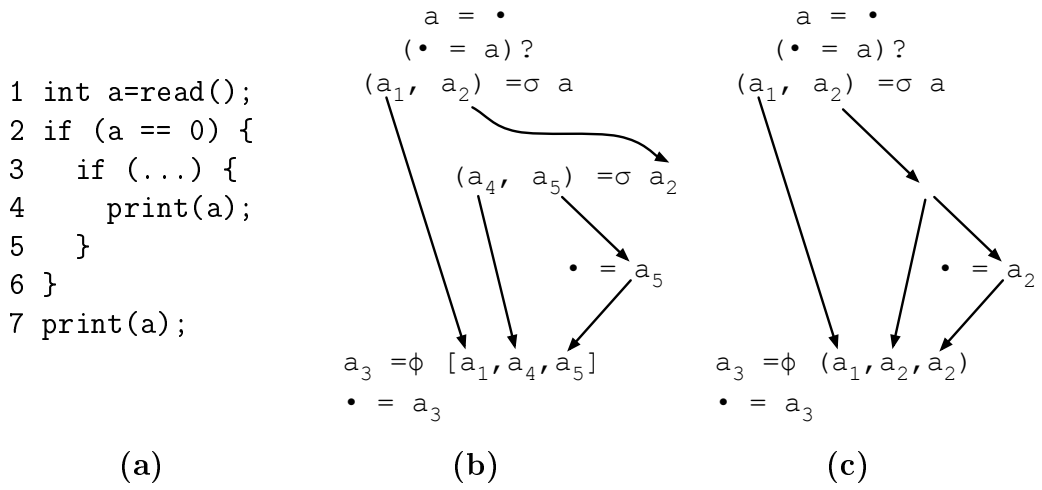

Figure 2.2: Example of the use of SSI and e-SSA on information analysis. (a) Original program. (b) SSI program. (c) e-SSA program.

In order to convert a program into SSI form we need two special types of instructions: ϕ-functions and σ-functions. The ϕ-functions are an abstraction used in the *Static Single Assignment form* (SSA) [CFR+91] to join the live ranges of variables. Any SSI form program is a SSA form program. For instance, the assignment, $v = \phi(v_1, \ldots, v_n)$, at the beginning of a basic block B, works as a multiplexer. It will assign to v the value in v_i, if the program flow reaches block B coming from the i^{th} predecessor of B.

The σ-functions are the dual of ϕ-functions. Whereas the latter has the functionality of a variable multiplexer, the former is analogous to a demultiplexer, that performs an assignment depending on the execution path taken. For instance, the assignment, $(v_1, \ldots, v_n) = \sigma\ v$, at the end of a basic block B, assigns to v_i the value in v if control flows into the i^{th} successor of B. Notice that variables alive in different branches of a basic block are given different names by the σ-function that ends that basic block.

There exists an interesting relationship between the live range of program variables and graphs. Chaitin *et al.* have shown that the intersection graph of the live ranges of a general program can be any type of graph [CAC+81]. In 2005, researchers have shown that the intersection graphs produced from programs in SSA form are chordal [Bou05; PP05]. Recently, Benoit *et al.* showed that the interference graphs of programs in SSI form are interval graphs, a subset of the family of chordal graphs [BBDR09].

Singer gives two examples of SSI clients [Sin03]: very busy expressions, and the dual available expression analysis. An expression e is very busy at program point p if e is computed in any path from p to the end of the program, before any variable that is part of it is redefined. Such analysis, also called *anticipatable expressions analysis* by Johnson and Pigali [JP93], is useful for performing optimizations such as partial redundancy elimination. Conversely, an expression e is *available* at program point p if it is computed in any path from the beginning of the program until p, and none of the variables that are part of e are redefined thereafter.

A sparse analysis associates information to variables, instead of program points. That is, busy expressions associated to variable v are the busy expressions at the definition point of v. Similarly, available expressions associated to v are the expressions available at the program point where v is last used. The SSI form allows us to perform these analyses non-iteratively [Sin03]. As another example, Benoit *et al.* have shown how SSI speeds up the computation of liveness analysis [BBDR09]. This is a dataflow analysis that finds which are the live variables at each program point.

Although the SSI representation suits the needs of many different compilation passes – henceforth called clients, the majority of these clients require only a subset of the SSI properties. This observation is important, because converting a program to SSI form is a time consuming endeavor. For instance, Bodik *et al.*'s ABCD algorithm [BGS00] uses information from conditional branches to put bounds on the value of variables used as array indices. Thus, it requires that only integer variables used in conditionals bear SSI properties. Even less demanding is the sparse conditional constant propagation algorithm described by Ananian [Ana99] and Singer [Sin06], which demands that only variables used in equality comparisons be in SSI form.

Extended SSA (e-SSA) [BGS00], described by Bodik *et al.*, is a subset of SSI. Its clients do not require that variables be fully converted to SSI form. Instead, they need a representation that restricts the *value range* of variables. The value range of a variable is the set of values that the variable may assume during program execution. For instance, variable a in Line 1 of Figure 2.2(a) may assume any value of the integer type in the Java language, thus, its value range is $[-2^{31}, 2^{31} - 1]$. However, the conditional branch in Line 2 restricts the value range of a. Thus, in Line 3 of our example program, this range is $[0, 0]$. There exist two main events that may restrict the value range of a variable v: an assignment to v and a conditional branch that tests v.

2.3 Elementary Form

Elementary form, described by Appel and George [AG01], is an intermediate representation in which the live range of all variables are splitted between every pair of consecutive instructions. Three steps are necessary to convert a program to elementary form. First, we insert a parallel copy between each pair of consecutive instruction. Second, we insert a parallel copy between each consecutive basic block. Parallel copies are necessary, because more than one instruction may be split in a program point. Finally, we rename all variables according to the parallel copies. The second step can be avoided in case the program code is already in SSI form. Appel and George used the idea of inserting parallel copies everywhere in their ILP-based approach to register allocation with optimal spilling [AG01].

Elementary form increases by a quadratic factor the amount of variables, however their relation is simplified. Considering a program with n variables and I instructions, the worst case splitting would produce a program with nI variables, since each variable would be splitted between each pair of instructions. Considering that each instruction defines a new variable, it is possible to say that $I = n$, therefore the worst case would produce n^2 variables, which we call a quadratic factor. Figure 3.1(c) shows a program in elementary form, while the original program is shown in Figure 3.1(a).

An example of client for the elementary form is the Register Allocation by Puzzle Solving [PP08], developed by Pereira and Palsberg. This algorithm allocates each instruction from the source code at a time. In order to be able to separate variables live in each instruction, it is necessary to convert the program to elementary form. More details on this client can be seen in Section 3.2.

2.4 Conclusion

In this chapter we have presented three intermediate representation that can be used in a compiler implementation. SSA is a representation that allows only one definition for each variable. For this purpose they introduced a special instruction called ϕ-function. In order to augment SSA, Ananian proposed SSI. This new representation extends the first one by defining that there should not be two different non-consecutive uses reaching a single instruction. The σ-functions are used with the opposite purpose as ϕ-functions. Finally, we have presented the elementary form, that splits the live range of all variables between every instruction.

In order to achieve our objective, we have studied the described intermediate representations to find out which would be more appropriate for register allocation.

In this process we have developed an algorithm to efficiently create SSI and e-SSA [TPBB10]. Since e-SSA is a subset of SSI, converting from one representation to the other should be done with no collateral damage to the resulting code. We show, in the referenced paper (Appendice A), how this can be achieved and also we propose an approach in which the client is free to decide which representation and which variables to convert.

We conclude from this research that the intermediate representation presented in this chapter have several clients that benefit from their properties. However only SSA and Elementary form has been widely applied on the field of register allocation. Chapter 3 presents advantages of using SSA and Elementary form for register allocation.

Chapter 3

State of Art on Register Allocation Algorithms

Register allocation is an important compiler optimization. In the words of Hennessy and Patterson: “Register Allocation adds the largest single performance improvement to compiled programs" [HP02]. An important step of this optimization is the register coalescing. In this chapter we present the main algorithm in the literature for these optimizations.

There are several different approaches to allocate registers. The classic approach, created by Chaitin *et al.* [CAC+81; Cha82] and later improved by Briggs *et al.* [BCT94] and Appel and George [AG01], models it as a graph coloring problem. Other approaches are Linear Scan Register Allocation [PS99], Register Allocation via Integer Linear Programming [DPV06; FW02; GW96], Register Allocation via Partitioned Quadratic Programming [SE02], Register Allocation via Multi-Flow of Commodities [KG06] and Register Allocation by Puzzle Solving [PP08].

Based on graph coloring register allocation we describe the following register coalescing algorithms: Aggressive Coalescing [Cha82]; Conservative Coalescing [BCT94]; Iterated Register Coalescing [AG01]; Brute-Force Conservative Coalescing [BDR08]; Optimistic Register Coalescing [PM04; PM98]; and Copy Coalescing by Graph Recoloring [HG08]. We also describe Punctual Coalescing [PP10], which is an extension to Register Allocation by Puzzle Solving.

3.1 Register Allocation

The goal of register allocation is to map an unbounded number of variables from the source code to a finite number of registers. Considering that registers are fast and few,

a great challenge of a register allocator is to store as many variables as possible in registers. As an example, Nandivada *et al.* shows that the difference, in terms of program execution time, between a trivial allocator that sends every variable to memory, and an optimal, integer linear programming based approach, can be as high as 250% [NPP07]. For completeness, we present in this section some basic definitions related to register allocation and coalescing. Figure 3.1 is used to describe these definitions.

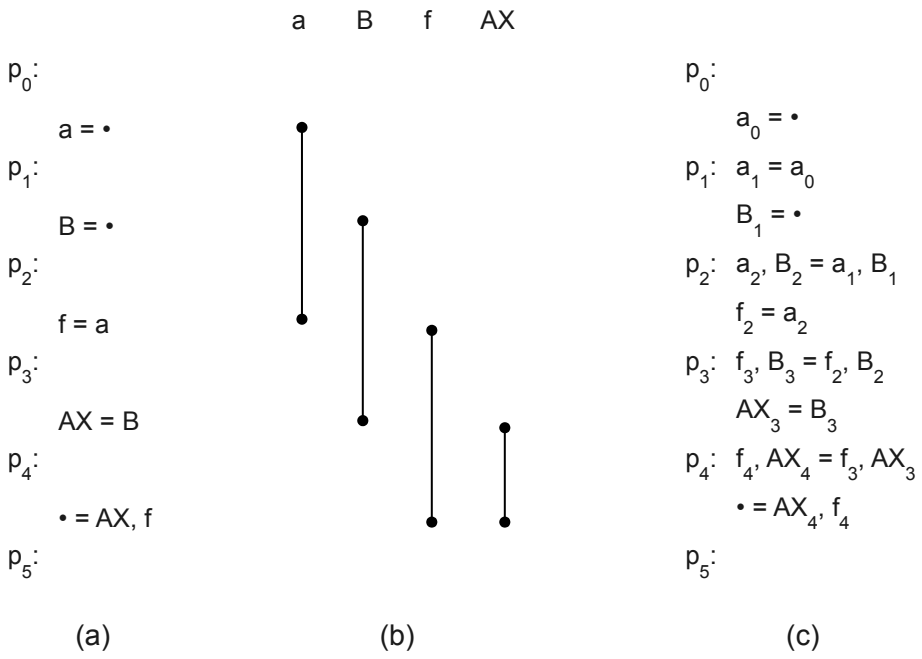

Figure 3.1: An example program. **(a)** Original program. **(b)** Live Range of the variables. **(c)** Program in Elementary Form.

Program Point. Program point is the point between two consecutive instructions. There are also program points in the beggining and end of each basic block. In Figure 3.1(a), there are 6 program points named p_0 to p_5. We say a variable a is live at a program point p if there is a path from p to a use of a without any redefinition of a in the path.

Program Region. A program region is a set of program points.

Live Range. The live range of a variable is the collection of all program points where the variable is live. In Figure 3.1(a), the live range of variable a consists of the program points $\{p_1, p_2\}$. The live range of each variable is represented in Figure 3.1(b).

Pre-Coloring. Pre-Coloring is a phenomenon that forces a variable to be in a certain register. For example, ARM architecture uses registers as arguments to functions, and in x86 the result of a division must be placed in *edx* and *eax*. For example, in Figure 3.1(a), variable AX is pre-colored with register AX.

Spilling. Usually, the number of available registers to store variables from the source program is limited. For this reason there are cases where there is not enough registers to store all variables. If this happens some variables must be stored in memory, in other words *spilled* to memory. In order to spill a variable it is necessary to insert *store* functions to move the variable from a register to the memory, and *load* functions every time it is necessary to rematerialize the variable in a register. These instructions tend to be slow, compared to operations that do not access memory. Therefore, it is desirable that the register allocator minimizes the number of variables spilled.

Coalescing. Coalescing is the act of mapping two non-interfering variables related by a copy instruction to the same register. For instance, in Figure 3.1(a), the instruction $f = a$ can be removed in case variables f and a are assigned to the same register, without changing the program semantic.

Aliasing. An architecture contains aliasing if changing the value of a register affects the value of another register [SRH04]. In Figure 3.2 we show a subset of the registers from x86 architecture. Each 16bits register is subdivided in two 8bit register, which are called *low* address or *high* address. As a convention we will refer in lower case variables that fits in one register and in upper case variables that fit in two registers. Therefore, in Figure 3.1(a), variables a and f fit in one register, while variables B and AX fit in two.

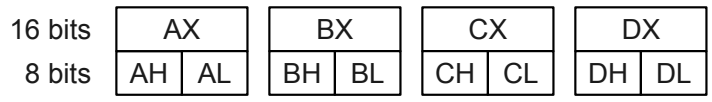

Figure 3.2: Subset of the register bank from the x86 architecture.

Register Pressure. The register pressure of a given program region is the amount of registers required to allocate the live variables within the region. In case of the architecture contains aliasing, the notion of register pressure is associated with each register class. For example, in program point p_2 (see Figure 3.1(a)) the register pressure for registers of 16 bits and 8 bits is different. For the class of 8bit registers the register

pressure is 3, because 3 registers are necessary to hold variables a and B. However, for the class of 16bit registers the register pressure is 4. This is due to the fact that if a register is half occupied, we consider that the whole register is occupied, therefore two 16bit registers are occupied.

Live Range Splitting. The concept of live range splitting is the opposite of coalescing. While coalescing joins the live range of two variables; live range splitting divides the live range of a variable in two new live ranges. The splitting of live ranges tends to lower the number of interferences between variables, which can facilitate the allocation and decrease the number of spilled variables. An example of live range splitting is shown in Figure 3.1(c). In program point p_1 variable a is split. A copy is inserted at this location separating two versions of the variable, a_0 and a_1, each with its own live range.

3.1.1 Graph Coloring Approaches

Chaitin, proposed the approach to solve register allocation modeling as a graph coloring problem [CAC+81; Cha82]. The idea is to represent the program as an interference graph, where each variable present in the source code represents a node in the graph. Two nodes are adjacent if the variables represented by these nodes interfere in the source code, i.e., these variables are live at the same program point and cannot share the same register. The coloring problem is to find a color for each node in the graph, in order to guarantee that adjacent nodes do not share the same color. Each color represents an available physical register from the architecture's register bank.

Additionally, two nodes related by a move instruction are connected by an affinity edge. This special edge does not make two nodes to be neighbors, apart from the case when they interfere and have a normal edge. In case two nodes related by an affinity edge are colored with the same color the move instruction that relates them can be removed from the source code. Figure 3.3 shows the five main steps in Chaitin's graph coloring register allocator:

- **Build graph:** construct the interference graph for the source program. The live range of each variable is created through a dataflow analysis. In case the live range of two registers overlap, they cannot be allocated to the same register and their respective nodes in the interference graph will be adjacent.

- **Aggressive Coalesce:** Any two not adjacent nodes related by an affinity edge are coalesced to form a new node. The original nodes are removed from the graph,

while the new node is included. After any node coalescing the graph must be rebuilt, which could also result in more opportunities for coalescing. Whenever there is no more moves to be coalesced, the coalescing phase is over. More details on aggressive coalescing can be found in Section 3.3.1.

- **Simplify:** simplify the graph using Kempe's heuristics [Kem79]. Consider a graph G, an architecture with k available registers, and that each node has a degree which is represented by the number of its neighbors. In case a node has degree less than k, any possible coloring for its neighbors will leave at least one available color for itself. Based on this assumption the algorithm repeatedly removes nodes with degree less than k from the graph and inserts them in a *simplify stack*. Each node removed from the graph will reduce the degree of its neighbors, leading to more opportunity for simplification. If the graph contains only nodes with degree higher than k, a node is removed from the graph and marked to be spilled. If any node has been marked to be spilled, the process continues to the spill phase, and repeats all the process from the beginning, otherwise, the select phase follows.

- **Spill:** For each spilled node, loads and stores are inserted in the source code. This code will reduce the live range of the spilled variable and will create tiny live ranges in the places where the variable is fetched from memory. These new live ranges must be represented in the graph and the build phase is again executed.

- **Select:** assign color to nodes in the graph. Starting from the top of the *simplify stack*, the original graph is rebuilt. For each node reintroduced in the graph, a color, not yet used by its neighbors in the graph, is selected for it.

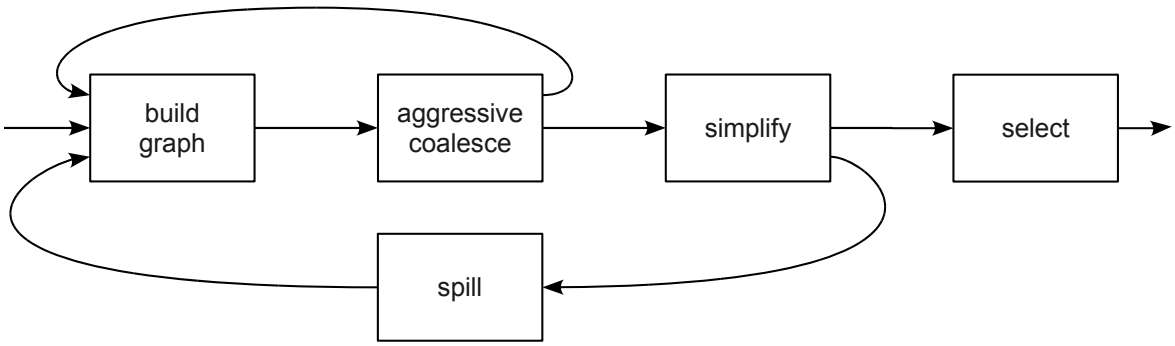

Figure 3.3: Chaitin's graph coloring register allocator.

Figure 3.4 (a) shows an example program, and Figure 3.4 (b) outlines its interference graph. In this example, we assume that lower-case names denote 8-bit variables,

while upper-case names denote 16-bit variables. If we assume an architecture with two 16-bit registers, each having two 8-bit aliases, then the graph in Figure 3.4 (b) is not *colorable*. That is, there is no register assignment that keeps all the variables simultaneously alive in registers. The register allocator normally solves this problem via *spilling*. In Figure 3.4 (c) we have sent variable a to memory; thus, creating two new variables, a_0, at the definition point of a, and a_1 at its use point. The new interference graph, given in Figure 3.4 (d) is now colorable.

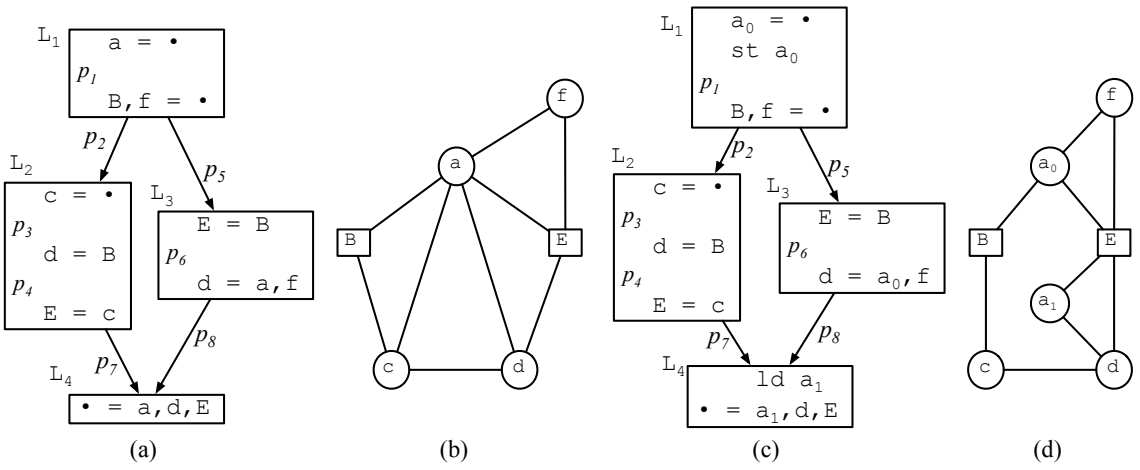

Figure 3.4: Graph-coloring-based register allocation. (a) Example program. (b) Program's interference graph; squares denote double precision values. (c) Program after spilling variable a. (d) New interference graph.

The algorithm described by Chaitin successfully solves the allocation problem using the abstraction of graph coloring. However the spilling decision is made too early in the algorithm, which results in unnecessary spills. To address this issue, Briggs *et al.* developed what they called optimistic coloring [BCT94]. This new approach embodies two modifications on the original algorithm to postpone the decision to spill a node, and the resulting allocator is shown in Figure 3.5:

- **Simplify:** the simplification step removes nodes with degree less than k. Whenever the graph remains with nodes of degree higher than k, a node is removed from the graph and marked as a spill candidate. Instead of spilling the node at this point, the algorithm optimistically stack the node in the *simplify stack* hoping that a color will be available for this node and no spill will be necessary. This process goes until the graph has no more nodes.

- **Select:** in the select step, the allocator may discover there is no color left for a certain node. In this case, it leaves this node uncolored and proceeds coloring the next node. Only nodes marked as spill candidates can be in the situation where no color is left, however for some of them a color will be available and a spill will be saved. Whenever this process ends with a list of uncolored nodes, the allocator insert spill code for each of them, and returns to the build graph phase. In case all nodes are colored, the allocation has succeeded.

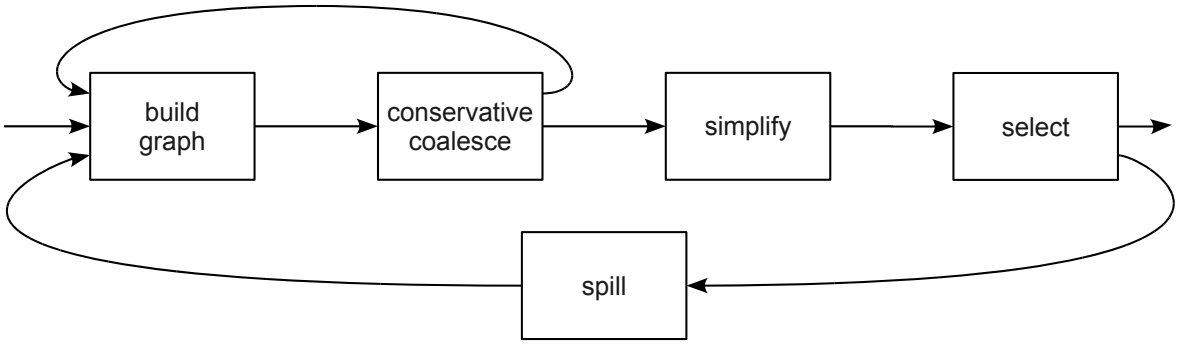

Figure 3.5: Briggs' graph coloring register allocator.

The algorithm proposed by Briggs also introduced the notion of conservative coalescing. In the coalescing algorithm proposed by Chaitin, any two nodes related by an affinity were coalesced. This approach was very successful in eliminating move instructions, however it constrained the coalesced node, which result in more spilling. To solve this issue Briggs proposes a conservative solution, where two nodes are coalesced when it is safe to be done. More information on conservative coalescing can be found in Section 3.3.2.

The algorithm proposed by Briggs was successful in coalescing conservatively. However it was too conservative, which leads to too many uncoalesced nodes. George and Appel proposed an structural change in the algorithm [GA96], to be able to coalesce more nodes without compromising the colorability of the graph. The main contribution of this approach is the freeze phase, described in sequence. The main steps of the algorithm, shown in Figure 3.6, are:

- **Build Graph:** creates an interference graph, as in Briggs and Chaitin approach. But also categorize each node as being move-related or not. A move-related node is an operator of a move instruction.

- **Simplify:** simplify only those non-move-related with low degree.

- **Coalesce:** perform conservative coalescing on the reduced graph obtained by the simplification phase. Since the degree of many nodes have been reduced, the conservative strategy is likely to find more moves to coalesce than it would possibly find in the initial graph. Any coalesced node that is no longer move-related will be available for the next round of simplification. Simplify and coalesce are iterated until only move-related or significant-degree nodes remain on the graph.

- **Freeze:** in case neither simplify or coalesce applies, they select a low degree move-related node and freeze its moves. To freeze a move means to give up on that move, which will turn the node into a non-move-related node. Simplify is called in case a node is freezed.

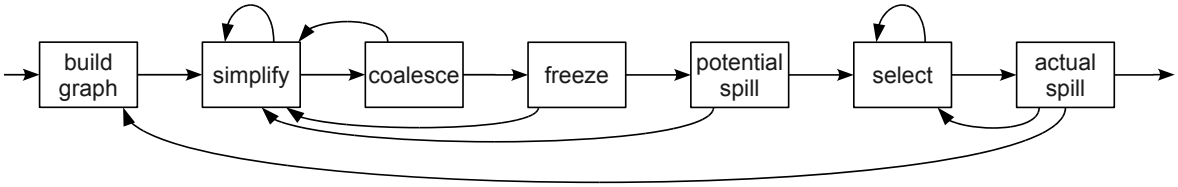

Figure 3.6: Iterated Register Coalescing graph coloring register allocator.

The main contribution of this approach is to iterate the simplify and coalesce steps, with simplify called first. This change leads to more conservative coalesced nodes, since the graph used in the coalesce step is simpler. The authors also proposed another conservative test. In the coalesce step they use Briggs conservative test, in case it fails they use their new test, which is described in Section 3.3.3.

3.1.2 Graph Coloring for Architectures with Aliasing

Traditional graph coloring approaches, described in Section 3.1.1, do not handle an important characteristic commonly found in commercial architectures, aliasing. Multiple register names may be aliases for a single hardware register. Smith *et al.*, propose a generalization of graph-coloring register allocation that handles this characteristic, while preserving the elegance and practicality of traditional graph coloring [SRH04]. The proposed generalization is capable of working on top of any graph coloring algorithm. The authors chose to experiment their algorithm using Appel and George's Iterated Register Allocation [AG01].

In the presence of aliasing, the simple test based on the node degree is not enough to check for greedy K colorability. Smith *et al.* devised a correct test [SRH04], using the idea of *squeeze* introduced by Fabri [Fab79]: a node v can be simplified if the worst case allocation of all neighbors of v (*squeeze* of v) is less than the number of registers available for it.

3.1.3 Linear Scan Register Allocation

Poletto and Sarkar proposed a global algorithm, called Linear Scan [PS99], to solve register allocation that is not based on graph coloring. The linear scan algorithm is fast and easy to implement which makes it a good solution for just-in-time compilers [Ayc03]. Some modern compilers, such as LLVM [Evl04] and Java HotSpot client compiler [WM05], include implementations of this algorithm.

Linear Scan is a greedy algorithm that reduces the allocation problem of finding a coloring to a list of live intervals. Following we present the four steps of this approach:

- **Order instructions:** choose a linear order for the basic blocks in the program. The order in which the basic blocks will be placed does not affect the correctness of the algorithm, but can affect the quality of the resulting code.
- **Create live intervals:** with the program in linear form each variable has a live interval with nonconsecutive program points, i.e., an unconnected live interval. To simplify the process, they consider a continuous live interval that starts at the first definition of the variable and ends at the last use.
- **Allocate intervals:** following the live intervals in order of creation, the algorithm assigns a register to each live interval. In case no register is available, a live interval is selected to be spilled.
- **Rewrite code:** with the assignment of registers or memory positions to live intervals, the code is written. In this step variables are replaced by registers and spill codes are inserted.

3.1.4 Other Register Allocation Approaches

Other register allocation approaches present in the literature are:

- **Register allocation via Integer Linear Programming** [DPV06; FW02; GW96]: The basic idea of this approach is to model the interactions between registers and variables as constraints in a system of integer linear equations. It

produces code of very good quality; however, as integer linear programming is NP-complete [Kar72], it presents a worst case exponential running time, and can take hours to find an optimal solution.

- **Register allocation via Partitioned Quadratic Programming** [SE02]: This approach associates a cost matrix C to each edge of the interference graph of the source program. Each cost matrix C_{uv} describes the tradeoffs of assigning different registers to variables u and v.

- **Register allocation via Multi-Flow of Commodities** [KG06]: In this approach, a program is seem as a collection of K pipes, thru which the allocator must pass a number of indivisible commodities. Each pipe corresponds to a physical location, either register or memory, and each commodity corresponds to a variable. Thus, a flow of a commodity represents the detailed allocation of the variable that the commodity encodes.

3.1.5 Analysis

The literature on graph coloring approaches to solve register allocation have evolved over the last 20 years. Many improvements were proposed. We have described the main flavors on this approach. The algorithm described by Chaitin successfully solves the allocation problem using the abstraction of graph coloring, however the coalescing was aggressive and compromised the spilling. Later, Briggs and Appel improved the original algorithm. The coalescing is now conservative and the spilling decision is postponed which saves many nodes from being stored in memory. The disadvantage of this approach is the graph structure, as its construction and maintenance is time and space consuming.

Smith *et al.* proposed an extension for the graph based algorithms to handle aliasing. With this generalization it is possible to better use the register bank of architectures with aliasing. However the necessity of constructing and maintaining a graph structure is still present, which is time and space consuming.

The main appeal of linear scan is the speed. Which makes it suited for JIT architectures. However this advantage comes with a cost in terms of the quality of code produced. The hard decision of this approach is the order in which the basic blocks are linearized. The perfect order would reduce the number of false interferences[1]. However

[1] An interference is false when the linearization creates an ordering where there is a basic block where a variable is not used between its definition and use.

it is time consuming to find this perfect ordering, therefore the authors use an heuristic to get the best ordering in a reasonable time.

3.2 Decoupled Register Allocation

Key to a decoupled register allocator is the following property:

Property 3.1. *The maximum register pressure at any program point equals the global register pressure.*

Not every program provides this property; however, some representations, such as SSA-form [CFR+91], provide it in the absence of aliasing. In this case, the local register pressure at a given point is simply the cardinality of the set of variables alive at that point. Due to this property, spilling – the lowering of register pressure – can be done directly on the program, without the need of a data structure that gives a global view of the program, such as an interference graph. This property has another advantage, such as simplifying optimizations that impact the register pressure, like partial redundancy elimination [KCL+99]. Decoupled register allocators, in general, follow this four steps:

- **Split Live Range:** use live range splitting to guarantee Property 3.1. In general, live range splitting will divide the program into regions, in such a way that variables in different regions do not interfere.
- **Spill:** lower the register pressure at each program point, via variable spilling, until this local register pressure is less than or equal to K, the number of registers. Any spilling heuristics works, although some might produce code of better quality than others.
- **Select:** assign variables to registers. The register allocator must be able to find a way to assign registers to variables without causing further spills.
- **Insert Code:** insert code between the regions to preserve the semantics of the pre-live range splitting program. This code is necessary because part of a variable might be allocated into different registers across consecutive regions.

In face of aliasing, the SSA form conversion is not extensive enough to guarantee Property 3.1; instead, one solution is to convert to *elementary form* [PP08]. We convert a program to elementary form via the insertion of parallel copies between each pair of consecutive instructions. Figure 3.7 (a) shows our running example in elementary

form. The interference graph of the new program, conveniently called an elementary graph, is given in Figure 3.7 (b). The dotted lines denote *affinity edges*: it is beneficial to assign nodes linked by such edges to the same color, because every time we fail to do it, a copy instruction will make its way into the final assembly program. Elementary graphs have a very simple structure; thus, determining that the local register pressure has a polynomial time solution, even when nodes are allowed to have weights 1, 2 or 4, as in the case of aliased register allocation. The variables in the program given in Figure 3.7 (a) can be allocated into two aliased registers; an improvement on the original program seen in Figure 3.4 (a), which requires three aliased registers. This result is not a coincidence: any program can be transformed into the elementary form, and the elementary form program never requires more registers than the original code [PP08].

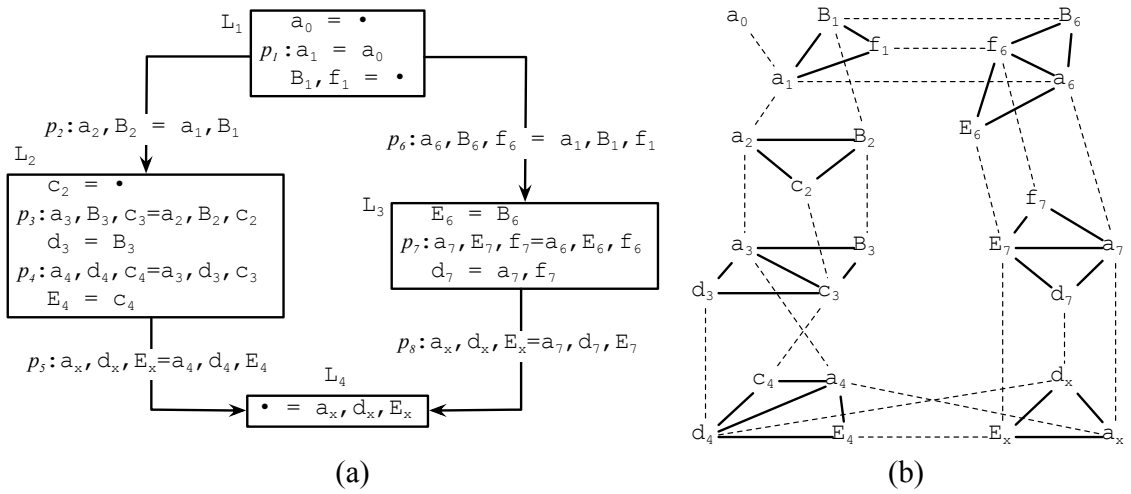

Figure 3.7: (a) The program from Figure 3.4 in elementary form. (b) The interference graph of the elementary program.

The first decoupled register allocator to run on elementary form programs was the puzzle solver [PP08]. This allocator sees the problem of determining the local register pressure as a puzzle. The puzzle pieces correspond to program variables. There are three categories of pieces: K (kill), D (def), and A (live across). Pieces also have a size, or spam, determined by the size of the variable they represent. The challenge is to place all these pieces into a puzzle board, which corresponds to the register file. Figure 3.8 shows the pieces and board that describe an architecture in which a register is divided into two aliases. Other decoupled register allocator approaches are described in [WF10; SB07; AG01].

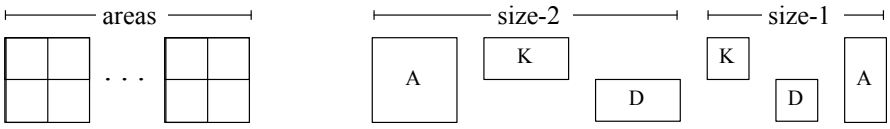

Figure 3.8: Pereira and Palsberg's puzzle notation [PP08]: areas represent the register bank, and pieces represent the program variables.

3.2.1 Analysis

Puzzle solving register allocation is an optimal local solution, which results in perfect allocation for each instruction, but not for the entire program. It is a fast algorithm, that is also suited for JIT machines. However, as in Linear Scan, the drawback is that the quality of code produced is inferior compared to graph coloring approaches. For example, the solution in Figure 3.9 (a) causes the insertion of three instructions to preserve the semantics of the original program – an excess that could be reduced to one by splitting the live range of variable `d`, instead of variable `a` at program point p_4.

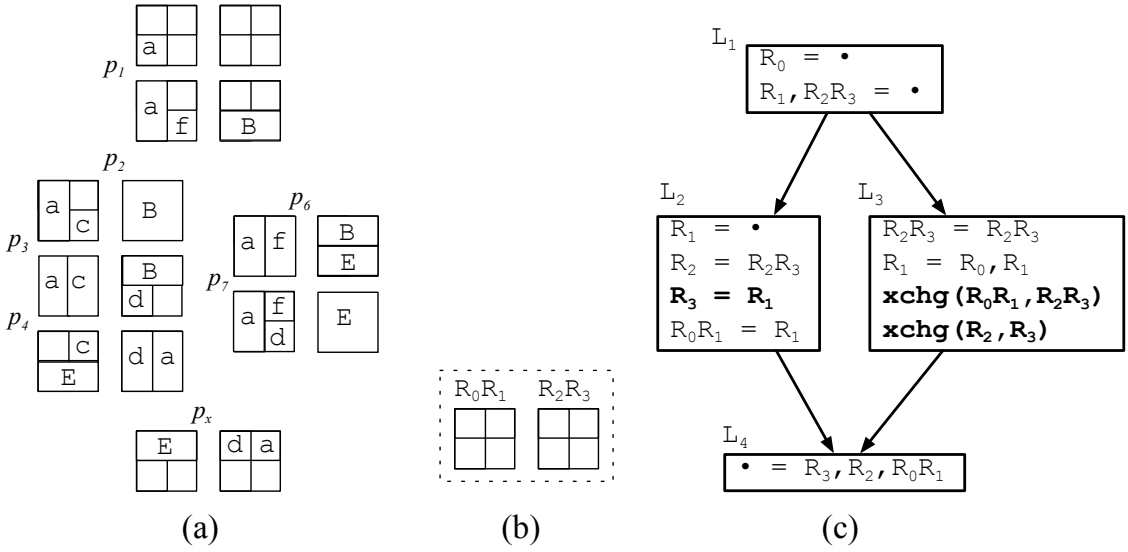

Figure 3.9: Register allocation by puzzle solving. (a) Each unconnected graph from Figure 3.7 (b) is reduced to a puzzle, solvable in polynomial time. (b) Our hypothetical architecture provides two aliased registers. (c) The solution of the puzzles is mapped to assembly code. The bold instructions are code necessary to preserve the semantics of the original program.

3.3 Register Coalescing

The act of coalescing nodes can reduce substantially the number of instructions in the source code, which reduces both the code size and the execution time. Not only original move instructions can be removed from the code, but also move instructions inserted by live range splitting. Bouchez *et al.* have shown that the problem of register coalescing is NP-complete [BDR07]. In this section we describe the main heuristics proposed for this problem.

3.3.1 Aggressive Coalescing

Aggressive register coalescing was described by Chaitin in the paper that first introduced the graph based register allocation [Cha82]. As the focus of his work was to create a new approach to allocate registers, Chaitin used a simple heuristic to solve register coalescing, as follows: every two nodes that do not interfere and are related by a copy instruction are coalesced. This approach coalesces a great number of nodes from the graph, however it can increase the number of register spilling. Considering that the coalescing of two nodes is done with no restrictions, a k-colorable graph can became not k-colorable after the coalescing, which would lead to more registers spilled to memory in order to color the graph.

In general a move instruction from register to memory is more costly then a move between two registers. Therefore coalescing heuristics [Cha82] that increase spilling may also increase the total execution time of the program. For this reason different approaches to coalesce registers were proposed in the literature with the purpose to coalesce only those nodes that would not require any extra spilling.

3.3.2 Conservative Coalescing

In order to address the problem of the large amount of variables spilled to memory, Briggs *et al.* developed an algorithm where only node coalescing considered conservative are performed [BCT94]. A coalescing is considered conservative when the resulting graph is still k-colorable. The resulting graph is k-colorable when the coalesced node has at most k neighbors of significant degree, where significant degree means a degree equal or higher than k.

To understand why this heuristic is conservative it is necessary to recapitulate the simplification heuristic used by Chaitin [Cha82]. Nodes with degree less than k are removed from the graph until the graph becomes empty, or the graph possess only nodes with degree greater than k. In the latter case variables are spilled to memory.

Therefore, only nodes with more than k neighbors with significant degree in the original graph can be spilled to memory.

3.3.3 Iterated Register Coalescing

The coalescing approach described by Briggs *et al.* [BCT94] is capable of coalescing nodes without compromising the process of spilling, however it is too conservative and many nodes are not coalesced. A structural modification is proposed by George and Appel [GA96] in order to have a higher number of nodes being coalesced in a conservative fashion. Instead of coalescing nodes before the graph simplification, as done by Briggs, George and Appel proposes to first simplify the graph and coalesce afterwards, and to iterate these two steps until a fixed point is reached. A fixed point is reached when it is not possible to coalesce any node, neither to simplify the graph. With this modification the coalescing is performed each time in a simpler graph, which allows a higher number of nodes to satisfy Briggs coalescing rule.

George and Appel also developed a new criteria to decide whether a coalesce is conservative. Nodes u and v can be coalesced case all high degree neighbors of u are also neighbors of v, and vice versa. This test is conservative, because when all low degree nodes neighbor to the coalesced node $u + v$ are simplified, the result is a sub-graph of the original graph, therefore it is also k-colorable.

3.3.4 Brute-Force Conservative Coalescing

Briggs *et al.* [BCT94] and George and Appel [GA96] have proposed two conservative coalescing algorithms, that can safely coalesce many move related nodes without changing the colorability of the graph. However many other moves are left uncoalesced when they could be coalesced conservatively. This happens due to the fact that both algorithms are local, depending only on the degree of the neighbors.

To overcome this issue, Bouchez *et al.* proposed Brute-Force Conservative Coalescing [BDR08]. In order to decide whether a move can be coalesced they perform a more expensive test. First the two move related variables are aggressively coalesced. A new graph will be obtained including the new node and excluding the two old nodes. This new graph is tested for colorability. In case it is no longer k-colorable, the coalesce is considered not safe, and the node is de-coalesced returning to the graph just before this operation. A node that is considered unsafe is removed from the list and is never tested again. Since this test is costly they use Briggs' and George's algorithm first, in case they fail, brute force is used. A requirement of this algorithm is to start with a

k-colorable graph.

3.3.5 Optimistic Register Coalescing

The coalescing of nodes can increase their degree and compromise the colorability of a graph. To overcome this issue, [BCT94] and [GA96] developed heuristics that coalesced conservatively and do not compromised the graph colorability. However these heuristics decide whether a coalesce is conservative too early, missing opportunities of coalescing that would turn out to be safe. Further more, they ignore the fact that coalescing may decrease the degree of neighbors and improve the colorability of the graph.

To profit from coalescing benefits without compromising the spilling process, Park and Moon proposed Optimistic Register Coalescing [PM98; PM04], an algorithm which optimistically coalesces all possible move related nodes following Chaitin's aggressive approach. In case a coalesced node is selected to be spilled, this node is split back into the original nodes, which there is a better chance of coloring one of them and spilling only the other.

3.3.6 Copy Coalescing by Graph Recoloring

Copy coalescing by graph recoloring [HG08] was described by Hack and Goos and coalesces nodes of programs in the SSA intermediate representation in a conservative fashion. The first step of the algorithm is to transform the interference graph into a k-colorable graph by spilling some variables. The graph is then colored with an initial coloring. This initial coloring does not considers move related nodes, which could lead them to have different colors. In order to spare some move instructions, the algorithm tries to re-color these nodes with the same color. This could lead to a graph with neighbors having the same color. To solve this issue nodes are recursively re-colored until no neighbors share the same color. This approach could lead to an infinite loop of recoloring. However they limit to one recoloring per variable in each recoloring attempt. In case the graph has neighbor nodes with the same color in the end of a recoloring attempt, this attempt is undone.

Instead of trying to recolor by each affinity, the algorithm creates chunks of affinities. A chunk of affinities is a set of nodes that are related by affinities, but that do not have interference between them, which means they can all have the same color. So instead of having only two related nodes recolored, the algorithm tries to recolor the whole chunk to the same color, and propagates the recoloring if necessary.

3.3.7 Punctual Coalescing

Punctual Coalescing [PP10] was described by Pereira and Palsberg to complete their register allocation algorithm based on puzzle solving [PP08]. As described in Section 3.2, their register allocator traverses the source code and allocates, variables to registers, instruction by instruction without creating an interference graph. A puzzle is created for each instruction, where each live variable represents a piece and the register bank represents the board to be filled with the pieces. In the puzzle solver allocator, each instruction is allocated with no regards to the previous instruction. This could lead to a variable being allocated to different registers between to consecutive instructions, which would require a copy instruction.

In order to reduce the number of copy instructions inserted by the register allocator, the authors developed the Punctual Coalescing. Instead of allocating each instruction independently, the previous instruction is used as a guide to allocate the current instruction. The objective of the algorithm is to keep the greatest number of variables in the same register between the guider instruction and the current instruction.

3.3.8 Analysis

Aggressive Coalescing can coalesce a great number of nodes from the graph, however it can increase the number of register spilling. Since instructions from register to memory are often costly, the aggressive coalescing can in thesis slow down the execution time. To address this problem Briggs *et al.* developed an approach which conservative coalesces move instructions, without increasing the number of spilled variables, which results in better performance most of the time. However it is too conservative and many nodes are not coalesced. For this reason George and Appel proposed, Iterated Register Coalescing, a new conservative heuristic and also a structural change in the allocator design. These modifications increased the number of nodes coalesced in a conservative fashion, however the complexity of the algorithm is increased, and may compromise the interaction with other algorithms. Brute-Force Conservative Coalescing uses a global view of the interference graph to decide whether a coalesce is conservative. Many uncoalesced nodes by Briggs *et al.* and George and Appel are now coalesced, however the algorithm covers the whole graph, which makes it very slow.

Optimistic Register Coalescing profits from aggressive coalescing without compromising the colorability of the graph. However the author has concluded that it is necessary further experiments to evaluate if the results can be better than the early described conservative algorithms. Copy Coalescing by Graph Recoloring does not

compromises the spilling process and has the advantage of having a valid coloring during all steps of allocation. Therefore the algorithm can be stopped at any time that a valid result will be returned. This can be useful for JIT compilers, where execution time is a restriction, and the algorithm can be stopped after a certain number of recoloring was performed or a certain amount of time was spent. While this has a direct impact on the code size, it does not seem to have a dramatic impact to the performance of the compiled programs. Punctual Coalescing executes in linear time and is an optimal local solution, which is a good option for environments with dynamic compilation, where compilation time is important. Since the solution is not global, many unnecessary copies are inserted, specially between basic blocks.

3.4 Conclusion

In this chapter we presented the main algorithms present in the literature for register allocation and coalescing. Chaitin *et al.* introduced the notion of graph coloring register allocation in 1981, [CAC+81]. Later, Briggs *et al.* improved this idea with a better coalescing algorithm, in 1981 [BCT94]. In 2001, Appel and George changed the structure of the algorithm in order to coalesce more nodes conservatively, creating the Iterated Register Coalescing (IRC) [AG01]. On top of IRC, Smith *et al.* proposed, in 2004, an algorithm to deal with architectures with aliasing [SRH04]. In parallel, other researchers have proposed to solve the problem without using the graph coloring abstraction: Poletto and Sarkar proposed Linear Scan Register Allocation [PS99], which linearizes the live range of variables and allocates them; and Pereira and Palsberg proposed Puzzle Solving Register Allocation [PP08], which allocates each instruction at a time. The graph based approaches have a global view of the program, which results in better code, however it is necessary to maintaining a graph structure. The other two approaches have only local view of the problem, and therefore they do not have to deal with a heavy structure, as an interference graph, which results in a faster algorithm.

The main coalescing approaches also came in the graph coloring paradigm. First, Chaitin *et al.* proposed an aggressive coalescer [CAC+81], which lead to more spilling. Later Briggs *et al.* [BCT94] and Appel and George [AG01] proposed two conservative algorithms that coalesce nodes without compromising the colorability of the graph. Finally, Bouchez *et al.* proposed a brute-force algorithm for conservative coalescing [BDR08]. This algorithm has the advantage of coalescing a great number of nodes, and also the structure of the algorithm is very simple, however it is slow. In parallel, Park and Moon created an optimistic approach [PM98; PM04] and Hack and Goos created

an algorithm with recoloring [HG08]. The first is based on aggressive coalescing, but de-coalesces a node in case the colorability is affected. The second finds a initial coloring and tries to recolor the graph to improve the number of nodes coalesced.

In Chapter 4 we describe how the main aspects of the algorithms presented in this Chapter, are combined to describe a suite of techniques that make decoupled graph coloring register allocation, in the presence of aliasing, feasible and useful.

Chapter 4

Live Range Merging

In this chapter we describe the techniques that we use to design decoupled graph coloring register allocators that handle aliasing in a practical way. In order to have a decoupled approach, it is necessary to transform the interference graph into a k-colorable graph. For architectures with aliased registers, one solution is to convert the program into elementary form to make a k-colorable graph. As we show in Chapter 5, the conversion to this intermediate representation makes the interference graph, in average, 8x bigger, which is impracticable for an graph coloring approach. In order to maintain the elementary form properties without increasing too much the graph size, we propose two live range merging techniques: simple and punctual. Both of them are conservative and will never transform a k-colorable graph into a non-k-colorable graph.

4.1 Decoupling Spilling from Register Assignment in Face of Aliasing

Figure 4.1 shows our version of the iterated register coalescer, whose original design appears in Figure 3.6. A cursory comparison of the figures reveals that the decoupled version has less iterations between its phases. The *simplify*, *coalesce*, *freeze* and *select* phases are the same detailed by George and Appel [GA96]. The *split* phase implements the conversion of the source program into elementary-form and is described by Pereira and Palsberg [PP08]. The decoupled version uses a phase called *patch*, related to the implementation of parallel copies, not present in the original IRC algorithm. The polynomial time coloring of SSA-based allocators is possible only if the compiler can emulate the behavior of parallel copies [BCD+10; PP06]. After register allocation, the compiler must implement these parallel copies, using the instructions available in the

target architecture. Parallel copy patching has been thoroughly described by Pereira and Palsberg [PP09]. Therefore, the only new step of our implementation is the *spill* phase, which we describe in this section.

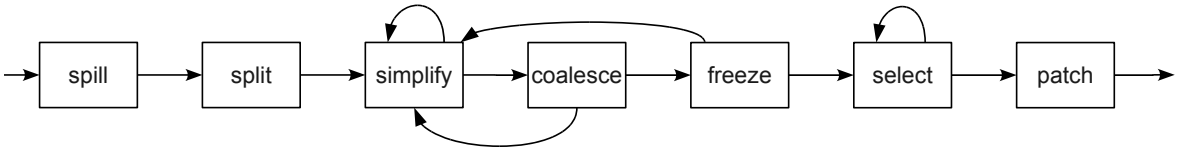

Figure 4.1: A decoupled re-implementation of the Iterated Register Coalescer, which we use in the experiments of Chapter 5.

Decoupled register allocation is interesting as long as it does not cause more spilling than traditional graph-based register allocators. The elementary form is an easy way to provide this guarantee. Given that the conversion to elementary form divides the source program in regions that are very small and simple, the problem of determining the local register pressure for each region has polynomial time solution [PP08], at least for architectures with quad, double and single registers, such as x86, ARM, PowerPC and SPARC. The polynomial time solution still hold in face of pre-allocation, a phenomenon caused by architectural constraints that force variables to be assigned particular registers [PP08].

Therefore, to bring the decoupled approach to Smith-style allocators, we convert the source program into elementary form, and analyze each program point independently, spilling variables until all the remaining variables are K-colorable. However, not every variable can be removed from a given program point. In order to guarantee that the "post-spilling" program will contain registers to re-load spilled variables, we cannot remove variables defined or used at that point. Figure 4.2 (a) shows the graph that corresponds to the instruction $\mathtt{E}_6 = \mathrm{B}_6$ in Figure 3.7 (b). Variables $\mathtt{a}_6$ or $\mathtt{f}_6$ can be spilled, but the compiler must assign the other variables a register in order to produce valid assembly code to that instruction. Notice that we do not ever have to build the program interference graph during the spilling phase: we can work on the program itself. We have experimented with two ways to perform the local register pressure test: the original Smith simplification test and a new version of this test, which we augment with the possibility of doing live range merging.

1. Checking Colorability Via Smith's Simplification Test.

Graph-based algorithms normally rely on Kempe's technique [Kem79] to remove nodes with degree less than K – the number of registers – until either the graph is empty, or all the nodes have higher degree. If a graph can be completely simplified

via Kempe's method, then the graph is called *greedy K colorable* [BDR07]. In the case of both the Iterated Register Coalescer, and the Brute Force coalescer, the spilling phase must guarantee that the program that passes forward to the other phases of the register allocator has an interference graph that is greedy K colorable. In the presence of aliasing, the simple test based on the node degree is not enough to check for greedy K colorability. A correct test has been devised by Smith *et al.* [SRH04], using the idea of *squeeze factor* introduced by Fabri [Fab79]: a node v can be simplified if the worst case allocation of all neighbors of v is less than v's *squeeze factor*. Figure 4.2 illustrates this idea. Figure 4.2 (a) shows a subgraph of the graph given in Figure 3.7 (b). Each vertex has been augmented with the *squeeze* factor of the variable that it represents, as determined by Smith *et al.*'s simplification criterion. The squeeze of a variable is the maximum number of registers that could be denied to it, given a worst case allocation of its neighbors. For instance, variable $\mathtt{B}_6$ has two neighbors, which could be assigned different double-precision registers; thus, its squeeze factor is 4. Using Smith's test, we remove variables, until all the remaining variables have a squeeze factor less than K. Notice that, in an elementary-form program the region in which variables interfere is so simple that we do not need to build a graph to perform Smith's test: we simply count the number of variables simultaneously alive.

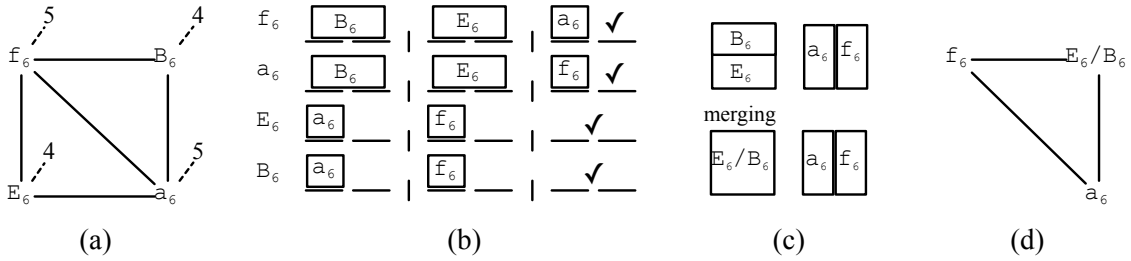

Figure 4.2: Smith *et al.* Simplification test. (a) A connected component of the graph in Figure 3.7 (b). (b) Worst case allocation for each variable. (c) A tidy allocation produced by a puzzle solver. (d) Variable merging guided by the puzzle solver.

2. Improving Smith's Test With Live Range Merging.

Assuming only two double-precision registers, the squeeze-based simplification test would fail to simplify any node in Figure 4.2 (a), and some variable would have to be spilled. On the other hand, there exists a register assignment that accommodates all the variables, as Figure 4.2 (c) shows. We say that a graph is *colorable* if there exists a valid assignment of registers to nodes such that every node receives a register. Any

greedy K-colorable graph is colorable; however, the opposite is not true, as the example in Figure 4.2 demonstrates.

In order to improve Smith *et al.*'s simplification test, we do live range merging whenever we are unable to simplify any variable. We use the Algorithm 4.1.

Algorithm 4.1: Improving Smith's test with live range merging

```
1  Let P be the collection of pieces that form a puzzle in an elementary-form
   program. Assume that we group the pieces of P into three sets: A, K and D,
   according to the notation in Figure 3.8
2  while P ≠ ∅ do
3      if ∃v ∈ P : v is simplifiable then
4          simplify(v)
5          P = P \ {v}
6      else if ∃d ∈ D and k ∈ K : d.size= k.size then
7          Let d, k be the largest pieces that fulfilled the condition
8          a = merge(d, k)
9          P = P \ {d, k} ∪ {a}
10     else
11         Let v ∈ A : v not used in the related instruction
12         spill(v)
13         P = P \ {v}
```

When merging pieces, we choose the pair with pieces of largest spam because this strategy reduces more drastically the squeeze factor of the other variables alive in that program point. Another important detail of our algorithm is the fact that we use live range merging with discretion. If we are stuck in the simplification process, then we choose only one pair of pieces, merge them, and re-try the simplification test. We proceed in this careful fashion because merged variables will be assigned the same register. This restriction might have the undesirable side effect of constraining too much the register coalescer that will run after spilling takes place.

We do no apply live range merging at program points that contain pre-allocated variables. Pre-allocation might prohibit the merging of live ranges, and, in face of this phenomenon we fall back to Smith *et al.*'s simplification test.

4.2 Augmenting the Brute-Force Register Allocator to Handle Aliasing

Smith et al's extensions have been designed to fit any graph coloring based algorithm. However, to the best of our knowledge, the Iterated Register Coalescer is the only

documented algorithm that has been adapted to use such extensions. We have added one more member to this family: the brute force register coalescer of Bouchez *et al.*. Confirming Smith *et al.*'s expectations, this adaptation was fairly straightforward.

The brute-force algorithm, outlined in Figure 4.3, has a more modular design than the iterated register coalescer. After spilling is performed, we order the copies in the source program according to their *profitability*, and try to coalesce them following this ordering. The profitability of a copy is a measure of how much improvement its deletion can bring to the target code, independent of how this elimination impacts the other copies. Copies inside deeply nested loops tend to be more profitable than copies outside loops. We say that the coalescing of vertices a and b is *conservative* if the interference graph that we obtain after collapsing these nodes into a single node ab is greedy K-colorable. We use one of the following three tests, in order, to guarantee that it is conservative to coalesce copy $a = b$:

1. `Briggs`(a, b) [BCT94]: the merging of a and b will create a node ab with fewer than K neighbors with squeeze greater than K.

2. `George`(a, b) [GA96]: assuming that a is a pre-allocated variable, then every neighbor of a already interferes with b, or has squeeze less than K. Notice that we must also try `George`(b, a), as this rule is asymmetric.

3. `Brute`(a, b) [BDR08]: the graph that results from merging a and b is greedy K-colorable.

The only difference from our tests to the original tests [BDR08; BCT94; GA96] is the use of the node's squeeze factor instead of its degree.

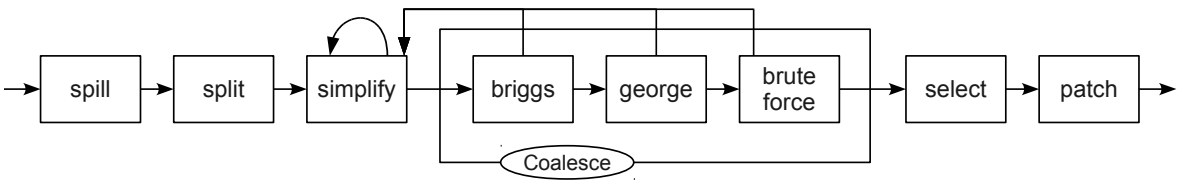

Figure 4.3: Bouchez *et al.*'s brute-force register coalescer.

4.3 Conservative Live Range Merging to Reduce the Problem Size

The heavy price incurred by the conversion into elementary form is the growth in the program size. As an example, whereas the interference graph in Figure 3.4 (b) has six nodes, the corresponding elementary graph seen in Figure 3.7 (b) has 26. This explosion is observed in actual benchmarks. Figure 4.4 compares the size of program traces taken from SPEC CPU 2000 before and after the conversion into elementary form. The puzzle solver avoids dealing with such big programs exactly because it only sees one instruction at a time: the live ranges around an instruction are split on the fly, during a traversal of the dominator tree of the source program. This is a luxury that a Chaitin style graph coloring algorithm has not been designed to afford.

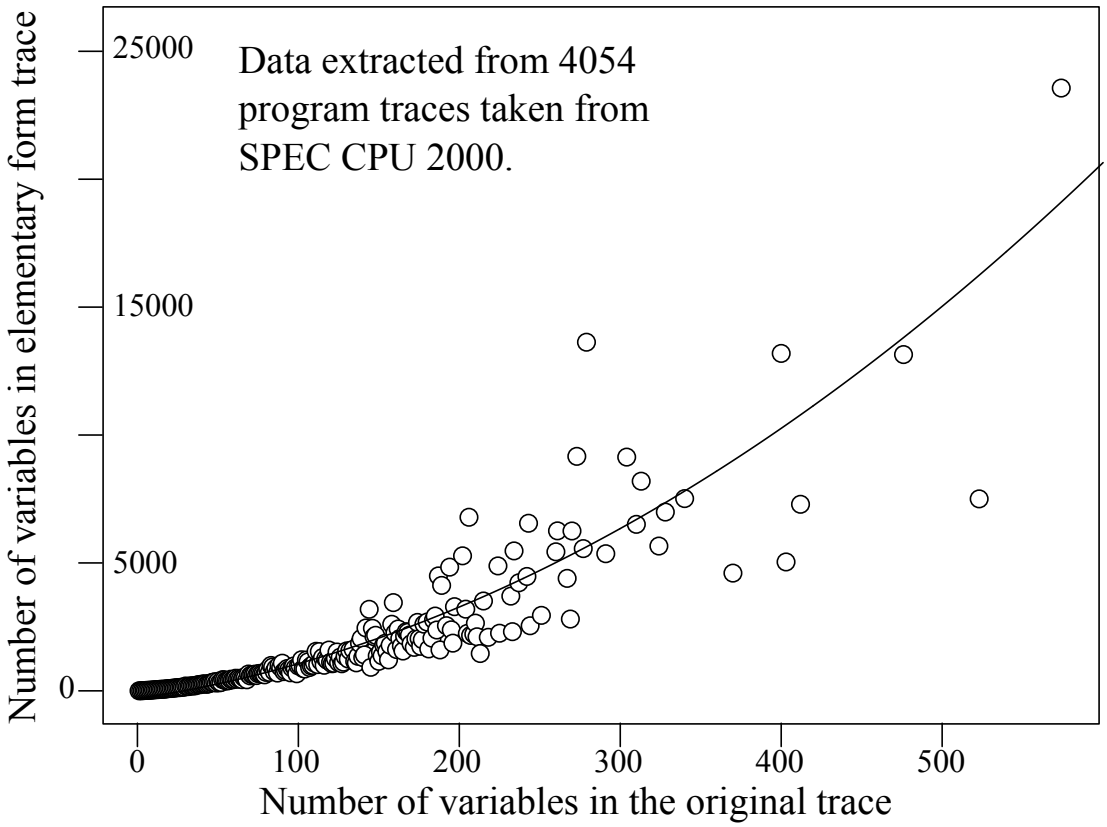

Figure 4.4: The growth in the number of program variables due to the conversion to elementary-form.

A simple way to shrink the size of elementary programs is to apply some fast coalescing criterion, i.e, Briggs's or George's to each affinity edge of the elementary

graph. However, even this approach is too slow: each of these tests is $O(K)$, and the elementary graph contain $O(I \times K + B^2 \times K)$ affinities, where B is the number of basic blocks in the program. Thus, this trivial elimination of affinities is $O(K^2 \times (B^2 + I))$. We want to perform this merging in batches, simultaneously eliminating many affinities at the expenses of a single and efficient conservative test. In this section we describe a way of doing it.

In order to perform live range merging, we traverse the dominator tree of the source program, eliminating all the affinities between an instruction, the *guider*, and its successor, *the follower*, whenever we can prove that it is conservative to do so. Otherwise, we keep all the affinities between the guider and the follower. We have tested two different criteria to ensure a conservative merging of live ranges. We call these tests *simple* and *punctual*.

4.3.1 The Simple Test

An elementary graph is formed by many unconnected components, which represent the live ranges of variables at some particular program point. Therefore, we expect a lot of redundancies between graphs formed from consecutive instructions. Given two instructions, the guider and the follower, all the vertices that correspond to variables live-in at the follower are connected through affinity edges to the vertices in the guider. A variable is live-in (out) at some instruction i if its live range contains any program point before (after) i. The only vertices in the follower's graph which have no affinities for vertices in the guider's are those nodes that represent variables defined in the follower instruction. We call them *critical nodes*. Normally an instruction defines at most one variable; hence, we expect to find at most one critical node in the follower. In light of these observations, the simple merging test is presented in Algorithm 4.2

Algorithm 4.2: Simple Test

```
1 if every critical vertex s in the follower's graph has a squeeze factor less than
    K then
2     for any vertex v in the follower's graph that has an affinity for a vertex u
      in the guider's graph do
3         collapse v and u into a node uv
```

Theorem 4.1. *Let* guider *(g) and* follower *(f) be two consecutive instructions. Let G_g be the interference graph of the variables live-in and live-out at g, and G_f be the interference graph of the variables live-in and live-out at f. If G_g is greedy K colorable,*

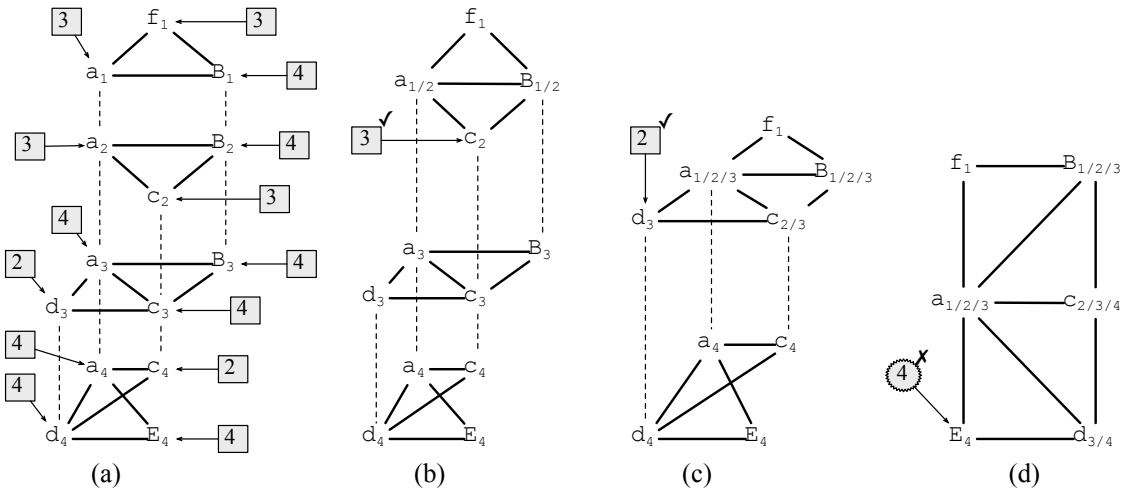

Figure 4.5: Simple live range merging applied on part of the graph of Figure 3.7. Considering two double precision registers available, we can merge all the subgraphs but the last, because the squeeze of $\mathtt{E}_4$ is 4.

then the graph that results from merging G_g and G_f via the simple test is greedy K colorable.

Proof. The proof is straightforward: if the merging is done, the resulting graph is formed by all the nodes from G_g plus the critical nodes in the follower. Because of the test in line 1 Algorithm 4.2, we know that every critical node can be simplified. Once they are simplified, we fall back into G_g, which, by hypothesis, is greedy K colorable. □

Figure 4.5 illustrates the simple live range merging technique applied to the sequence of instructions from program point p_1 to p_5 in Figure 3.7 (a). We have augmented the graphs in Figure 4.5 (a) with the squeeze factor of each node, and we have highlighted the squeeze factor of each critical node in the next figures.

4.3.2 The Punctual Test

The simple test only merges the live ranges of variables that are connected by affinity edges. However, in order to further reduce the size of the interference graph, we can also merge non-affinity related variables, using a technique based on punctual coalescing [PP10], which we call *punctual merging*. Punctual coalescing is a strategy used in conjunction with puzzle-based register allocation to remove copies in the target code. The punctual coalescer tries to fit variables related by affinities into the same

columns of two consecutive puzzle boards, and it is able to find the largest number of matches that do not compromise the solvability of these two puzzles. We confine punctual merging to the boundaries of basic blocks, because, by merging non-affinity related variables, we may eliminate coalescing opportunities. As we show in Chapter 5, punctual merging decreases the capacity of both, the Iterated Register Coalescer and the Brute Force Coalescer to eliminate copies in the final assembly code. Our live range merging technique based on punctual coalescing is presented in Algorithm 4.3.

Algorithm 4.3: Punctual Test

```
1  for each instruction guider in a top-to-bottom visit of the basic block do
2      Let follower be the instruction after guider
3      Let merges be an empty list
4      puzzleBoard = punctualCoalesce(guider, follower)
5      for each column r in puzzleBoard do
6          Let prevNode be undefined
7          for currentNode in that order in guider.r.{K, A, D},
           follower.r.{K, A, D} do
8              if prevNode is defined and prevNode.size = currentNode.size then
9                  prevNode = merge(prevNode, currentNode)
10             else if currentNode is defined then
11                 prevNode = currentNode
12     for each node s that contains follower's critical vertex do
13         if squeeze(s) > |s.register_class.registers| then
14             undo merges
```

Figure 4.6 illustrates punctual merging. We have used a different example this time, because our running example from Figure 3.7 is not complex enough to exercise the interesting aspects of punctual merging. Notice that the simple merging technique, when applied on Figure 4.6 (b) would only merge the variables `C`'s and `f`'s. However, assuming a solution of punctual coalescing that places variables $\mathtt{f}_0$, $\mathtt{f}_1$ and $\mathtt{g}_1$ into the same column, we can also merge these pieces. The same happen with – non-contiguous – variables $\mathtt{d}_0$ and $\mathtt{e}_1$. On the other hand, we cannot merge variables $\mathtt{a}_0$ and $\mathtt{b}_1$, because there are pieces in the puzzle rows between these two variables, i.e, $\mathtt{C}_0$ and $\mathtt{C}_1$.

4.4 Conclusion

In this chapter we described the techniques that we use to design decoupled graph coloring register allocators that handle aliasing in a practical way. We propose two live

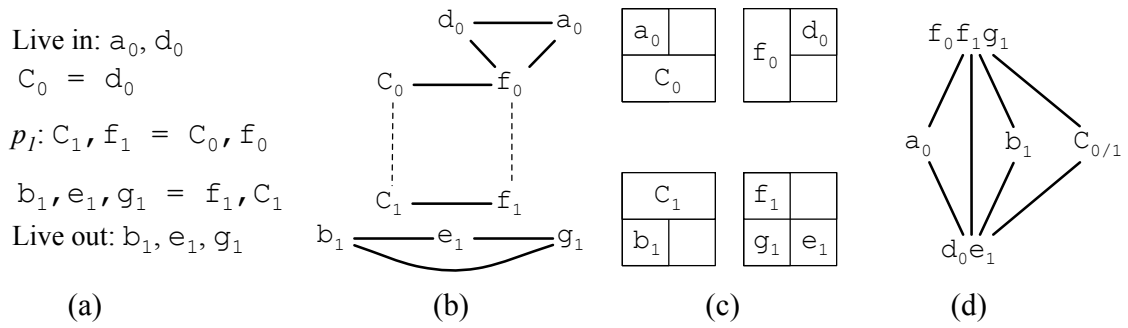

Figure 4.6: A constructed example showing punctual merging. (a) The elementary-form program. (b) The interference graph. (c) The solution of punctual coalescing. (d) The solution of punctual merging.

range merging algorithms that can reduce the graph size in a conservative fashion. It is important to have such algorithms to reduce the huge graphs created by elementary form.

We have adapted Bouchez *et al.*'s Brute Force coalescer [BDR08] to deal with aliasing, using Smith *et al.*'s. Confirming Smith *et al.*'s expectations, this adaptation was fairly straightforward.

We have also described an approach to improve Smith's spilling test. Using his squeeze-based simplification test, the graph for an instruction could be considered k-colorable, and still be colorable. In this case a variable would be spilled unnecessarily. To prevent this from happening, we now check if a graph is colorable, which means that there exists a valid assignment of registers to nodes such that every node receives a register. In case a graph is colorable but not k-colorable, we repeatedly merge nodes until the graph becomes k-colorable.

Chapter 5

Experiments

We chose to run our experiments on SPEC CPU 2000, which we have compiled into MinIR using LLVM 2.7 [LA04]. Most of the functions that we found in SPEC do not possess enough aliasing to exercise our algorithms; therefore, in these tests we only show results for the functions with at least 10% of instructions with live variables of different sizes, i.e, 8 and 16 bits.

For the sake of simplicity, we have executed our experiments in an artificial architecture, MinIR, described in Section 5.1; thus, we will not show run-time numbers. Nevertheless, we have checked the validity of each register allocation using the type-system of Nandivada *et al.* [NPP07].

5.1 MINimal IR Architecture

In this section we describe a prototype architecture called MINimal IR (MinIR)[1], which is based on the YAML[2] serialization format. YAML is used by STMicroelectronics *Inc* to quickly prototype hardware. The algorithms proposed in this chapter were implemented in Python, producing code to MinIR. MinIR provides a minimalist textual machine level intermediate representation to be used for experimental tools.

We have chosen to work with MinIR to reduce development time. The MinIR architecture provides simple structures and methods to access information on the source code. Furthermore, like any other dynamic language, Python can be used for scripting which facilitates the job of implementing the algorithms.

From the perspective of a compiler's writer, a scripting language permits rapid prototyping of compiler algorithms. With the use of MinIR and Python it was possible

[1] `http://www.assembla.com/wiki/show/minir-dev`
[2] `http://www.yaml.org/`

to implement the different forms of live range merging presented in this dissertion, as well as all the register allocation algorithms used to support the work. In this chapter we present experiments realized to substantiate this work.

Our target has the same instructions as x86; however, it uses a different register bank, which we outline in Figure 3.2. This register bank, presented in Algorithm 5.1 is a subset of the registers found in x86, showing two classes of registers. The first class, called $G8$, contains eight 8-bit registers, while the other class, called $G16$, contains four 16-bit register, as illustrated in Figure 3.2.

Algorithm 5.1: Register Bank from Figure 3.2 written for MinIR

```
1 regclasses:
2     - name: G8
3       registers: [AH, AL, BH, BL, CH, CL, DH, DL]
4     - name: G16
5       registers: [AX, BX, CX, DX]
6       elements: {AX: [AH, AL], BX: [BH, BL], CX: [CH, CL], DX: [DH,
      DL]}
```

YAML provides two main structures to store information. The first is the hash table, which is represented as {key1: value1, key2: value2, ...}. The second structure is an array, which is represented as [value1, value2, ...]. The elements of arrays/hashes can also be listed one element per line way using "-" for arrays.

We will be using an example to describe the structure of a MinIR program. The algorithm for a non-recursive factorial is presented in Figure 5.1(a), while its control flow graph is shown in Figure 5.1(b). Algorithm 5.2 shows the code for the non-recursive factorial written for MinIR, which computes the factorial of a positive interger n as the product of all positive integers less than or equal to n.

Following we present the main elements of a program written for MinIR:

- **Function:** Each function must have a label (line 1), a list of entry (Line 2) and exit (Line 3) points, and also a list of basic blocks (Lines 4-21). A function can have one or more entry and exit points.

- **Basic Block:** Each basic block is constituted of a label (Line 5) and a list of operations (Lines 6-9).

- **Operation:** The only mandatory information for an operation is its name, which is represented by *op*. The list of defined variables ($defs$) and the list of used variables ($uses$) are optional. There are also other optional information that

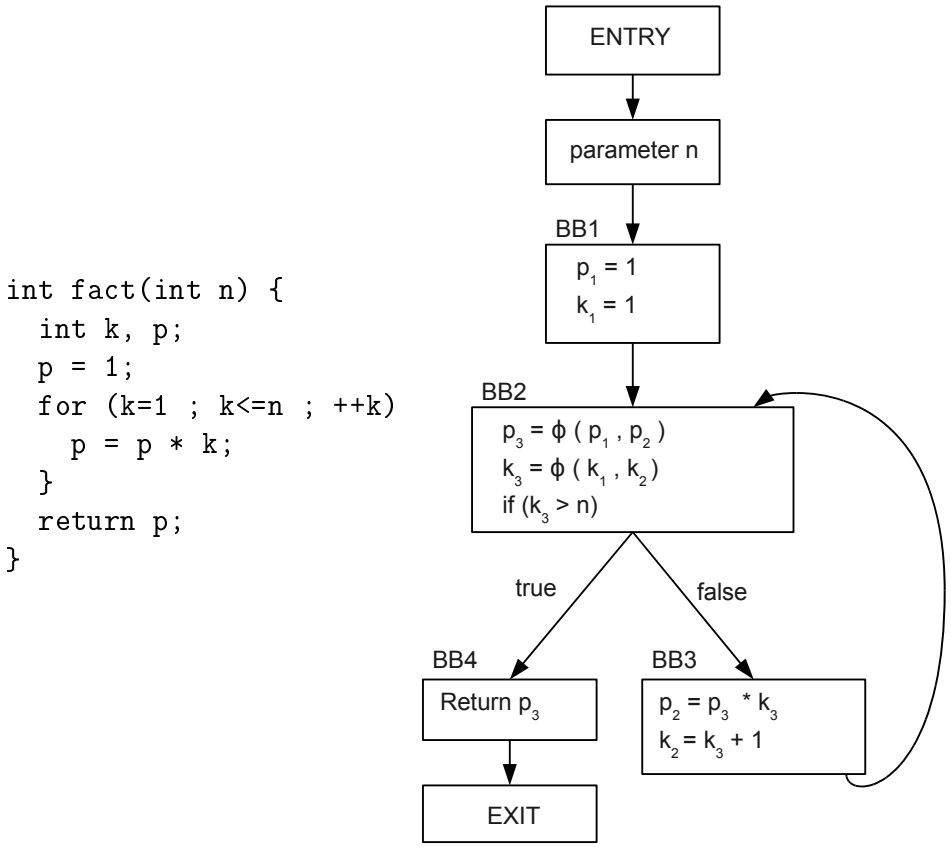

Figure 5.1: Example of non-recursive factorial. (a) Factorial in C. (b) Control flow graph of the program.

are used for specific types of operations. Below we describe the main types of operations:

- **ϕ-function:** The ϕ-functions must always be the first operations in a basic block, as seen in Lines 12-13 for $BB2$. It is mandatory to present the list of used and defined variables. As described in Section 2.1, depending on the control flow, one of the variables in the use list will be assigned to the defined variable. Therefore each used variable must be followed by the basic block it came from. For example, in Line 12, variable $p3$ will be attributed $p1$ case the control flow comes from $BB1$, and $p2$ case it comes from $BB3$.
- **Branch:** A branch instruction has the power to alter the control flow of the program. Two information are important for these instructions: the basic block to go in case the branch is taken (*target*), and the basic block to continue in case the branch is not taken ($fallthru$). For example, in Line 14,

we have a *gbt* operation. Case $k3$ is greater than n the control flow will go to $BB4$, in any other case the control flow will continue to $BB3$. A branch operation is always the last instruction of a basic block, and case it is not present, the control flow will continue to the next basic block of the list.

- **General:** These are all operations that do not fit in any of the above. An example is the addition operation in Line 18. Variable $k2$ will receive the result of the addition between variable $k3$ and 1.

- **Operand:** There are three types of operands:

  - **Variable:** A variable is represented by its name followed by its register class. For example, in Line 8, variable $p1$ is defined and its register class is $G16$.

  - **Register:** In case a variable is precolored by a register, we only show the register. If $p1$ was precolored to AX, we would replace $p1.G16$ by AX everywhere.

  - **Immediate:** An immediate is represented surrounded by single quotes, as the use operand in Line 8.

5.2 Experimental Results

Live Range Merging

The chart in Figure 5.2 illustrates the power of our live range merging techniques. The conversion to elementary form, on average, creates graphs 800% bigger than the original graphs. Simple live range merging reduces this difference to about 200%. Punctual merging is more aggressive, generating graphs that are even smaller than the graphs manipulated by the original implementation of iterated register coalescing.

Measuring the Effectiveness of Live Range Merging on the Register Coalescers

Live range merging, be it simple or punctual, not only reduces the size of the interference graph, but also improves the effectiveness of copy coalescing, as we show in Figure 5.3. This figure compares the brute force coalescer [BDR08], the iterated register coalescer [GA96] and the punctual coalescer [PP10]. The first two algorithms were implemented using Smith *et al.*'s extensions, as we have described in Section 4.2. Whereas punctual coalescing runs directly on elementary-form programs, we have tested the

Algorithm 5.2: Non-recursive Factorial written for MinIR

```
1  - label: factorial
2    entries: [BB1]
3    exit: [BB4]
4    bbs:
5      - label: BB1
6        ops:
7          - { defs: [n.G16], op: entry }
8          - { defs: [p1.G16], op: mov, uses: ['1'] }
9          - { defs: [k1.G16], op: mov, uses: ['1'] }
10     - label: BB2
11       ops:
12         - { defs: [p3.G16], op: phi, uses: [p1.G16<BB1>, p2.G16<BB3>] }
13         - { defs: [k3.G16], op: phi, uses: [k1.G16<BB1>, k2.G16<BB3>] }
14         - { op: bgt, fallthru: BB3, target: BB4, uses: [k3.G16, n.GR16] }
15     - label: BB3
16       ops:
17         - { defs: [p2.G16], op: mult, uses: [p3.G16, k3.G16] }
18         - { defs: [k2.G16], op: add, uses: [k3.G16, '1'] }
19     - label: BB4
20       ops:
21         - { op: return, uses: [p3.G16] }
```

other algorithms with three different inputs: elementary graph, graph after simple live range merging, and graph after punctual live range merging.

Figure 5.3 shows only the result of IRC using Smith *et al.*'s extensions. The algorithm executing over elementary graphs increased the size of the original program code, due to the insertion of register copies, by about 1,5%. Using simple live range merging this number shrinks down to about 0,54%. Graphs after punctual live range merging also yield better results than pure elementary graphs: 0,7%. Notice that in all three cases, this increase is the result of doing live range splitting: there are situations, like in the example in Figure 3.4, when splitting the live ranges of variables via copy instructions is necessary to avoid spills. Simple live range merging tends to produce better result than punctual live range merging. We speculate that this happens because the punctual merging constrains too much the interference graph, creating nodes with larger squeeze factors.

Figure 5.4 shows the effectiveness of the brute force coalescer implemented using the modifications from Section 4.2. Running the algorithm directly on elementary-form programs increased the code size by 0,34%. If we precede the execution of the algorithm with simple live range merging, than the number of copy instructions decreased to

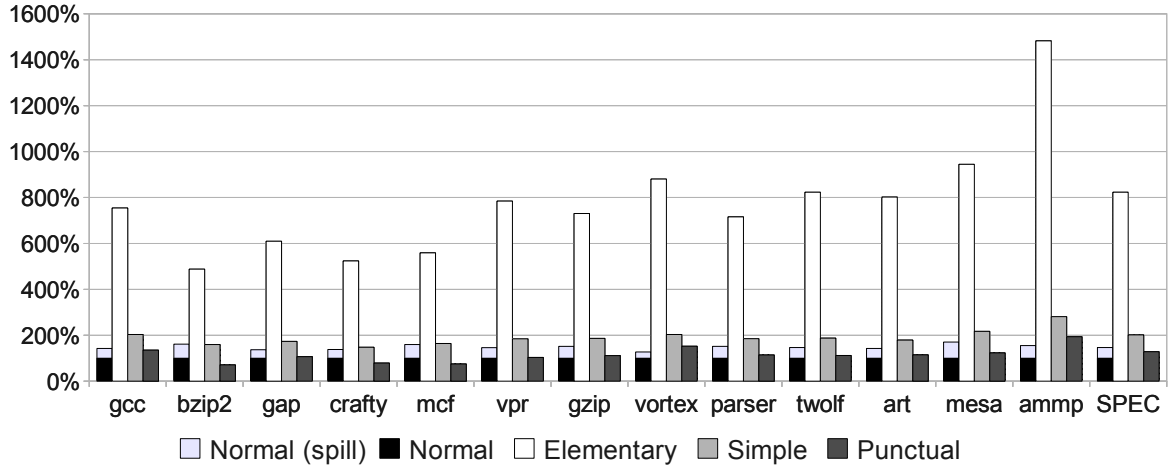

Figure 5.2: Comparison of the live range merging techniques. The bars represents the number of nodes in the interference graph (values are normalized to *normal*). ***Normal:*** input graph passed to the normal iterated register coalescing algorithm – the number of nodes is the number of variables in the source program. ***Normal (spill):*** the largest graph that the iterated register coalescer had to manipulate due to new variables created after spilling. ***Elementary:*** graph in elementary form. ***Simple:*** graph after the simple live range merging algorithm. ***Punctual:*** graph after punctual live range merging algorithm.

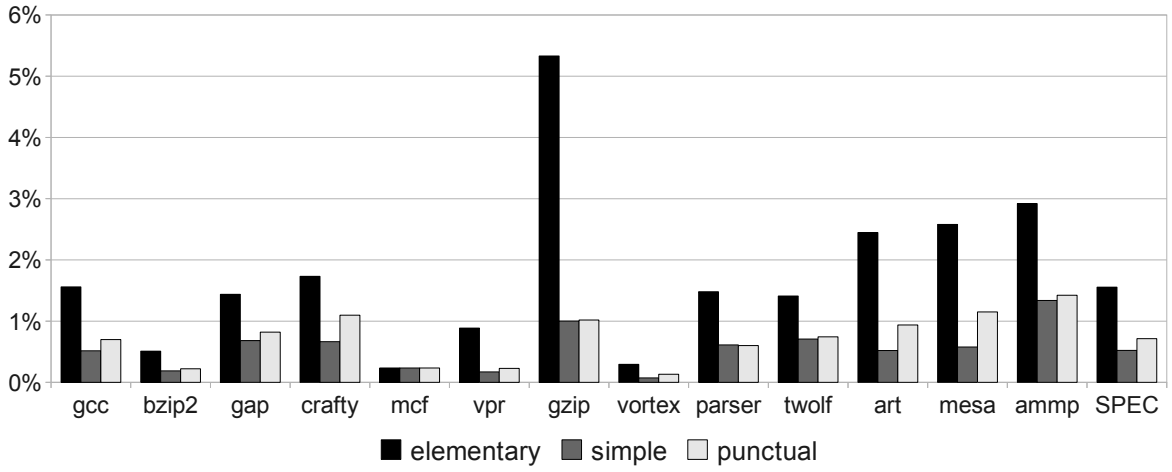

Figure 5.3: The effect of live range merging on the coalescing power of the iterated register coalescer (IRC). The graph shows the percentage of copies, used to convert the input program into elementary form, that were not coalesced by IRC.

0,33%. Contrary to IRC, in the case of Brute Force, punctual live range merging degrades the ability of the coalescer to eliminate copies: it increases the final code size in 0,58% – 0.24% more than running the coalescer on the original elementary-form

programs.

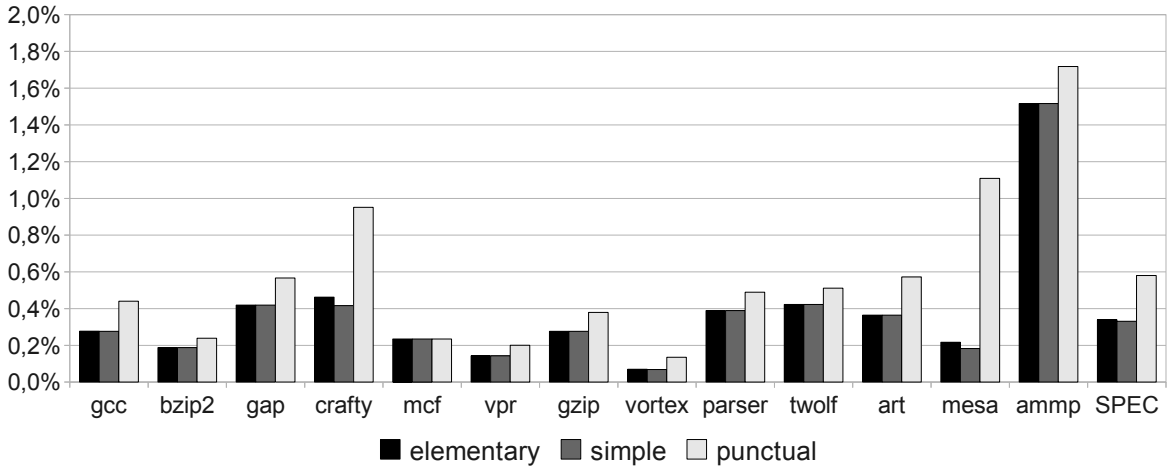

Figure 5.4: The effect of live range merging on the coalescing power of the brute force coalescer (BF). The graph shows the percentage of copies, used to convert the input program into elementary form, that were not coalesced by BF.

Figure 5.5 presents the comparison between the three algorithms: brute force (BF), iterated register coalescing (IRC) and punctual coalescing [PP10] in terms of the number of copies that each algorithm can eliminate via register coalescing. We have chosen the best configuration for each algorithm: IRC and BF run after simple live range merging, while punctual coalescing can only run on elementary-form programs. Confirming previous results [BDR08; PP10], the brute force coalescer is the most effective algorithm, adding only 0,33% new instructions into the final assembly code. IRC comes next, with an increase of 0,55%. The punctual coalescer increases the target code by almost 2.0%. This relatively bad result of the punctual algorithm is due to the fact that it is a local approach, which does not attempt to eliminate copies between basic blocks.

Spilling

The objective of this section is to show the effect of the technique discussed in Section 4.1 on spilling. Figure 5.6 presents the number of variables spilled with and without merging. We have set to 100% the number of variables spilled without merging. From the graph we can conclude that less than 0,5% of the variables spilled were saved using node merging. We attribute this low effectiveness of node merging to two factors. First there is a small number of instructions with aliasing in SPEC 2000. Second, the spilling algorithm used (spill everywhere) reduces too much the register pressure at each program point shadowing the need of a more advanced spilling heuristics. A

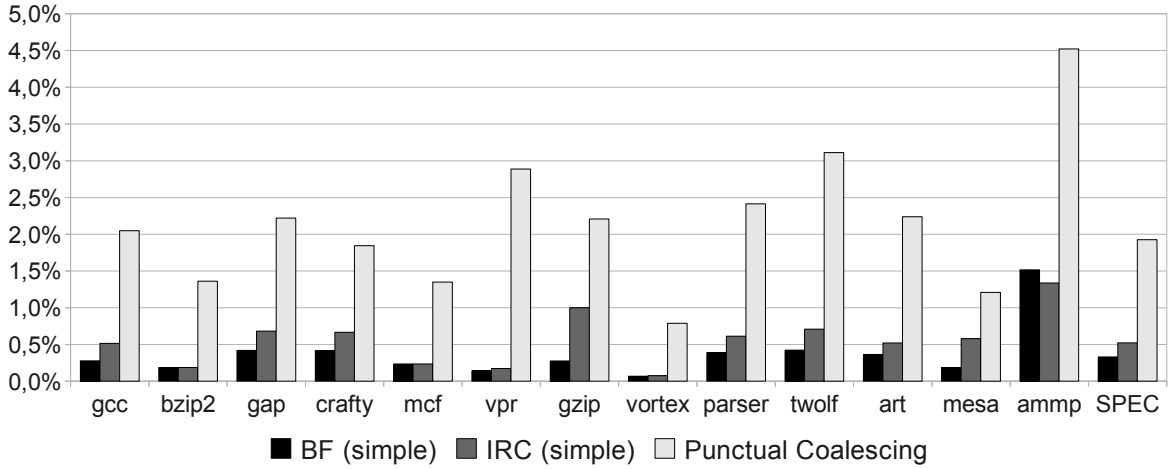

Figure 5.5: Comparing the effectiveness of three different register coalescing approaches. The graph shows the percentage of non-coalesced copies used to convert the input program into elementary form.

better algorithm to insert loads and stores would substantially increase the register pressure [AG01]. We believe that in such scenario many graphs would not be K-colorable, while still being colorable, boosting up the need of node merging.

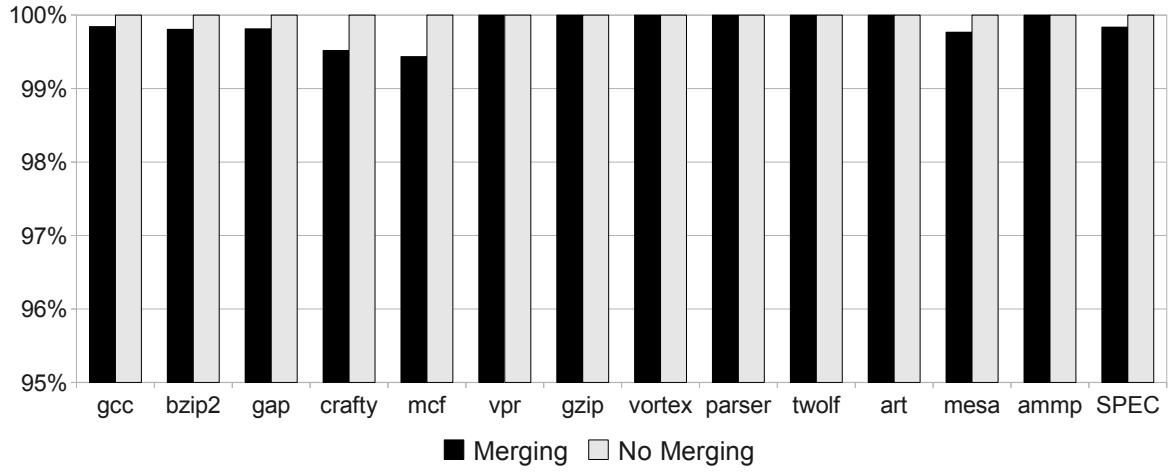

Figure 5.6: Number of variables spilled with and without node merging.

5.3 Evaluation

Many register allocation algorithms follow a decoupled approach that separates spilling from register assignment. This model has important advantages. First, the separation

between these two phases tends to yield simpler and more modular implementations: different spilling heuristics can easily be combined with different strategies to do register assignment and coalescing. Second, as we have illustrated in Chapter 3, decoupled designs have more flexibility to assign registers to variables; hence, are more successful at avoiding spilling. Although so fundamental, this very notion of live range splitting makes it difficult to extend decoupled algorithms to computer architectures with aliased register banks.

In order to implement a decoupled approach for architectures with aliasing, it is necessary to convert the program to elementary form. As shown in Figure 5.2, an interference graph of a program in elementary form is in average 8x bigger than the graph of the original program. This impressive increase in graph size is unfeasible for graph coloring approaches.

The solution found to keep a decoupled approach and a representation with the same properties as an elementary form was to conservatively merge live ranges. Simple live range merging produced graphs only 2x bigger than the original program's graph. Even better was the result of punctual live range merging, that produced graphs smaller than the largest graph the iterated register coalescer had to manipulate due to new variables created after spilling. Reduced graph size shows that our live range merging techniques are effective, however we still have to evaluate whether they are efficient.

To evaluate the efficiency of our techniques, we have executed them with different register allocation algorithms. Since spilling is now decoupled and executed before live range splitting, the number of spilled variables is not a good measure for our efficiency test. Therefore, we have decided to measure the number of variables coalesced in each case.

For both register allocation algorithms, IRC and Brute Force, the use of simple live range merging resulted in more variables coalesced than in an elementary form program, as showed in Figures 5.3 and 5.4. However we cannot say the same for punctual live range merging. For iterated register coalescing, it resulted in more coalescing than in elementary form, but for brute force, it coalesced less than in elementary form. And for both register allocation algorithms, simple live range merging coalesced more than punctual live range merging.

With this evaluation it is possible to say that these techniques should be used in different situation. Punctual live range merging produced the smallest graphs, while simple live range merging resulted in a code with more instructions coalesced and consequently a better final code. Therefore, we can say that the first technique is better suited for environments where the compilation time is an issue. And the second technique is indicated in case the best final code is sought.

We also proposed a merging technique to improve spilling. The motivation for this technique is that with Smith's squeeze test some graphs are considered not k-colorable, while they can be colored using other algorithms, for example Punctual Coalescing. This often happens in instructions where the register pressure is close to the maximum register pressure for the architecture. To combine both ideas, we presented a technique that merges variables in an instruction to make all instructions colorable by Punctual Coalescing, also k-colorable by Smith's squeeze test. However, as shown in Figure 5.6, the number of spilled variables saved by this new idea was not satisfactory. Since, the spilling algorithm used (spill everywhere) reduces substantially the register pressure of each instruction, there were few opportunities to apply the merging technique presented. Even though, we consider that this technique can be useful when used with an efficient spilling heuristic.

Chapter 6

Conclusion

This thesis has introduced a number of techniques that make graph coloring-based register allocation more practical and effective in the presence of live range splitting. Live range splitting helps to decrease the number of variables spilled during register allocation. However, in order to produce code to architectures with aliased register banks, register allocation algorithms demand a very aggressive form of live range splitting – the elementary format – which would increase too much the size of the program's interference graph, in addition of potentially causing the insertion of too many copies into the final assembly code. Our new techniques allows the register allocators to use all the power of the elementary format, while at the same time avoiding the size explosion, and decreasing the amount of copies into the assembly program.

6.1 Contributions

The main contributions of this thesis are:

- **A modification to the graph coloring register allocators to decouple spilling from register assignment in the presence of aliasing.**

  We describe, in Section 4.1, two ways to check that the register pressure at a given program point is low enough that no further spilling will be required once register assignment starts. The first test is based on Smith *et al.* extensions [SRH04] of Kempe's simplification test [Kem79]. The second test does *live range merging* to increase the ability of the spiller to keep more variables in registers.

- **A pre-processing technique that avoids creating huge graphs during the conversion of a program into elementary form.**

We call this method *conservative live range merging*, and we provide two versions, described in Section 4.3, which we call *simple* and *punctual*.

- **Simple merging:** Merges nodes from consecutive instructions, connected by affinity edges, in case it is considered conservative.
- **Punctual merging:** Merges nodes as in simple merging, and also merge non-affinity related variables, using a technique based on punctual coalescing [PP10].

One point that must be made clear is that we do not convert a program to elementary form, build its interference graph, and then merge nodes. Rather, we avoid splitting nodes when producing the elementary-form program.

- **An adaptation of the Brute Force coalescer of Bouchez *et al.* [BDR08] to deal with aliasing.**

  This is a straightforward use of Smith *et al.* notion of *variable squeeze* on Briggs *et al.* and George and Appel coalescing rules. We believe that Smith *et al.*'s implementation of the Iterated Register Coalescer already uses this rules; however, given that they are not described in the literature, we have explained them in Chapter 4 for the sake of completeness.

- **An adaptation of the Iterated Register Coalescing [AG01] to a decoupled approach.**

  Smith *et al.* described an algorithm for architectures with aliasing and implemented it on top of IRC. We have adapted this version to a decoupled approach, described in Chapter 4.

- **Comparison and evaluation of Brute Force coalescer and Iterated Register Coalescing in face of aliasing and using a decoupled approach.**

  We have implemented both coalescer algorithms using Smith's notion of variable squeeze and in a decoupled fashion. We tested these algorithms using SPEC 2000 on MinIR architecture. The results and evaluation of these experiments are described in Chapter 5.

- **Developed an algorithm to efficiently create SSI and e-SSA.**

  During the research process of finding the best representation to use in register allocation, described in Chapter 2, we have developed an algorithm to efficiently create SSI and e-SSA [TPBB10]. We show, in the referenced paper, how this can

be achieved and also we propose an approach where the client is free to decide which representation and which variables to convert.

In addition to the concrete contribuitions listed above, this project had the positive side-effect of creating an environment favorable of cooperation between Brazilian and French Institutes, once the subject of this thesis is part of the activities of the cooperation project between Fapemig and Inria, whose project team includes a brazilian and french researchers and students. Yet from the academic point of view, besides these contribuitions, this thesis resulted in 2 papers:

- Efficient SSI conversion, in Appendix A [TPBB10] which was published in SBLP 2010 and awarded the second best paper in the conference.
- Decoupled Graph-Coloring Register Allocation with Hierarchical Aliasing (to be submitted to SCOPES 2011).

6.2 Future Work

Following we present the list of possible future work:

- **Implement a better spilling algorithm.** As we described in Section 5.3, one possible reason for the spilling merging technique have performed so badly, could be that the spilling algorithm was not good enough. With a good algorithm that can keep the register pressure as high as possible, we expect that our technique will be able to merge more nodes and spare more spilling.
- **Implement the proposed algorithms in an industrial compiler.** We have implemented the algorithms we described Chapter 4, in an experimental architecture, called MinIR. For this reason we do not have execution time results. If implemented in an industrial compiler, as GCC [Gou05] or LLVM [LA04], we could have experimental results, and also compare it with the current register allocator for these compilers.
- **Study the reason why simple merging produced better code, while punctual merging produced smaller graphs.** We would like to study the results presented in Chapter 5 for the two proposed merging algorithms, to develop a new merging technique that could combine the qualities of both.

Bibliography

[AG01] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. *SIGPLAN Not.*, 36(5):243–253, 2001.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[Ana99] Scott Ananian. The static single information form. Master's thesis, MIT, September 1999.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, New York, NY, USA, 1988. ACM.

[Ayc03] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.

[BBDR09] Benoit Boissinot, Philip Brisk, Alain Darte, and Fabrice Rastello. SSI properties revisited. Technical Report 00404236, LIP Research Report, 2009.

[BCD+10] Florent Bouchez, Quentin Colombet, Alain Darte, Fabrice Rastello, and Christophe Guillon. Parallel copy motion. In *SCOPES*, pages 1–10. ACM, 2010.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994.

[BDJS06] Philip Brisk, Foad Dabiri, Roozbeh Jafari, and Majid Sarrafzadeh. Optimal register sharing for high-level synthesis of SSA form programs. *TCAD*, 25(5):772–779, 2006.

[BDMS05] Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomial-time graph coloring register allocation. In *14th International Workshop on Logic and Synthesis*. ACM Press, 2005.

[BDR07] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of register coalescing. In *CGO*, pages 102 – 104. IEEE, 2007.

[BDR08] Florent Bouchez, Alain Darte, and Fabrice Rastello. Advanced conservative and optimistic register coalescing. In *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 147–156, New York, NY, USA, 2008. ACM.

[BGS00] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM, 2000.

[Bou05] Florent Bouchez. Allocation de registres et vidage en mémoire. Master's thesis, ENS Lyon, October 2005.

[CAC+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. *SIGPLAN Not.*, 17(6):98–101, 1982.

[DPV06] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill Science/Engineering/Math, 2006.

[Evl04] Alkis Evlogimenos. Improvements to linear scan register allocation. Technical report, University of Illinois, Urbana-Champaign, 2004.

[Fab79] Janet Fabri. Automatic storage optimization. In *CC*, pages 83–91. ACM, 1979.

[FW02] 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.

[GA96] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.

[Gou05] Brian J. Gough. *An Introduction to GCC*. Network Theory Ltd, 1st edition, 2005.

[GW96] 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.

[HG06] Sebastian Hack and Gerhard Goos. Optimal register allocation for SSA-form programs in polynomial time. *Information Processing Letters*, 98(4):150–155, 2006.

[HG08] Sebastian Hack and Gerhard Goos. Copy coalescing by graph recoloring. *SIGPLAN Not.*, 43(6):227–237, 2008.

[HGG06] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *CC*, pages 247–262. Springer-Verlag, 2006.

[HP02] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, 2002.

[JP93] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *PLDI*, pages 78–89. ACM, 1993.

[Kar72] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[KCL+99] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred C. Chow. Partial redundancy elimination in SSA form. *ACM Trans. Program. Lang. Syst.*, 21(3):627–676, 1999.

[Kem79] A. B. Kempe. On the geographical problem of the four colors. *Amer. J. Mathematics*, 2:193–200, 1879.

[KG06] David Ryan Koes and Seth Copen Goldstein. A global progressive register allocator. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 204–215, New York, NY, USA, 2006. ACM.

[KW98] Timothy Kong and Kent D Wilken. Precise register allocation for irregular architectures. In *MICRO*, pages 297–307. IEEE, 1998.

[LA04] Chris Lattner and Vikram S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.

[LPP07] Jonathan K. Lee, Jens Palsberg, and Fernando M. Q. Pereira. Aliased register allocation. In *ICALP*, 2007.

[Mar06] Dániel Marx. Precoloring extension on unit interval graphs. *Discrete Appl. Math.*, 154(6):995–1002, 2006.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[NPP07] V. Krishna Nandivada, Fernando Pereira, and Jens Palsberg. A framework for end-to-end verification and evaluation of register allocators. In *SAS*, pages 153–169. Springer, Kongens Lyngby, Denmark, August 2007.

[Ple96] John Bradley Plevyak. *Optimization of Object-Oriented and Concurrent Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[PM98] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. In *In Proceedings of the 1998 International Conference on Parallel Architecture and Compilation Techniques*, pages 196–204, 1998.

[PM04] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. *ACM Trans. Program. Lang. Syst.*, 26(4):735–765, 2004.

[PP05] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *APLAS*, pages 315–329. Springer, 2005.

[PP06] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation after classic SSA elimination is np-complete. In *Foundations of Software Science and Computation Structures*. Springer, 2006.

[PP08] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation by puzzle solving. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 216–226, New York, NY, USA, 2008. ACM.

[PP09] Fernando Magno Quintao Pereira and Jens Palsberg. SSA elimination after register allocation. In *CC*, pages 158 – 173, 2009.

[PP10] Fernando Magno Quintão Pereira and Jens Palsberg. Punctual coalescing. In *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 165–184. Springer Berlin / Heidelberg, 2010.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *TOPLAS*, 21(5):895–913, 1999.

[Ron09] Hongbo Rong. Tree register allocation. In *MICRO*, pages 67–77. ACM, 2009.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, New York, NY, USA, 1988. ACM.

[SB07] Vivek Sarkar and Rajkishore Barik. Extended linear scan: an alternate foundation for global register allocation. In *LCTES/CC*, pages 141–155. ACM, 2007.

[SE02] Bernhard Scholz and Erik Eckstein. Register allocation for irregular architectures. In *LCTES/SCOPES*, pages 139–148. ACM, 2002.

[Sim96] Loren Taylor Simpson. *Value-driven redundancy elimination.* PhD thesis, Rice University, Houston, TX, USA, 1996. Chair-Cooper, Keith D.

[Sin03] Jeremy Singer. SSI extends SSA. In *PACT (Work in Progress Session)*, pages XX–YY, 2003.

[Sin06] Jeremy Singer. *Static Program Analysis Based on Virtual Register Renaming.* PhD thesis, University of Cambridge, 2006.

[SRH04] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 277–288, New York, NY, USA, 2004. ACM.

[TPBB10] André Tavares, Fernando Magno Pereira, Mariza Bigonha, and Roberto Bigonha. Efficient ssi conversion. In *SBLP 2010*, sep 2010.

[WF10] Christian Wimmer and Michael Franz. Linear scan register allocation on ssa form. In *CGO*, pages 170–179. ACM, 2010.

[WM05] Christian Wimmer and Hanspeter Mossenbock. Optimized interval splitting in a linear scan register allocator. In *VEE*, pages 132–141. ACM, 2005.

Appendix A: Efficient SSI Conversion

Efficient SSI Conversion

André Luiz C. Tavares[1], **Fernando M. Q. Pereira**[1],
Mariza A. S. Bigonha[1], **Roberto S. Bigonha**[1]

[1]Departamento de Ciência da Computação – UFMG
Av. Antônio Carlos, 6627 – 31.270-010 – Belo Horizonte – MG – Brazil

{andrelct,fpereira,mariza,bigonha}@dcc.ufmg.br

Abstract. *Static Single Information form (SSI) is a program representation that enables optimizations such as array bound checking elimination and conditional constant propagation. Transforming a program into SSI form has a non-negligible impact on compilation time; but, only a few SSI clients, that is, optimizations that use SSI, require a full conversion. This paper describes the SSI framework we have implemented for the LLVM compiler, and that is now part of this compiler's standard distribution. In our design, optimizing passes inform the compiler a list of variables of interest, which are then transformed to present, fully or partially, the SSI properties. It is provided to each client only the subset of SSI that the client needs. Our implementation orchestrates the execution of clients in sequence, avoiding redundant work when two clients request the conversion of the same variable. As empirically demonstrated, in the context of an industrial strength compiler, our approach saves compilation time and keeps the program representation small, while enabling a vast array of code optimizations.*

1. Introduction

Static Single Information (SSI) form is a program representation introduced by Scott Ananian [Ananian 1999]. This program representation redefines some variables at *program split points*, which are basic blocks with two or more successors. SSI form enables many compiler optimizations, because it allows an analyzer to augment variables with information inferred from the result of conditional branches. A non-exhaustive list of potential SSI clients includes array bounds check elimination [Bodik et al. 2000], bitwidth analysis [Stephenson et al. 2000], flow sensitive range interval analysis [Su and Wagner 2005], conditional constant propagation [Ananian 1999, Wegman and Zadeck 1991], partial redundancy elimination [Johnson and Pingali 1993], faster liveness analysis [Boissinot et al. 2009, Singer 2006], and busy expression elimination [Singer 2003].

Although the SSI representation suits the needs of many different compilation passes – henceforth called clients, the majority of these clients require only a subset of the SSI properties instead of a full conversion. This observation is important, because converting a program to full SSI form is a time consuming endeavor. For instance, Bodik *et al.* [Bodik et al. 2000]'s ABCD algorithm uses information from conditional branches to put bounds on the value of variables used as array indices. Thus, it requires that only integer variables used in conditionals bear SSI properties. Even less demanding is the sparse conditional constant propagation algorithm described by Ananian [Ananian 1999]

and Singer [Singer 2006], which demands that variables used in equality comparisons be in SSI form. On the other hand, the partial redundancy elimination algorithm described by Johnson *et al.* [Johnson and Mycroft 2003] uses an analysis called *anticipability*. A non-iterative computation of the anticipatable variables requires that all program variables be in SSI form.

In this paper we describe an *on-demand SSI conversion* framework, which saves compilation time and space in three different ways. First, it converts only a subset of variables in the source program to SSI form. Clients provide to our module a list of variables that must have the SSI properties, and only these variables are transformed. Second, we provide two conversion modes for each variable: *full* and *partial*. If a variable is fully converted into SSI form, then it presents the SSI properties traditionally described in the literature [Ananian 1999, Boissinot et al. 2009, Singer 2006]. On the other hand, if a variable is partially transformed, then it presents a restricted set of properties, that we describe in this paper. The partial conversion fits the needs of many SSI clients [Ananian 1999, Bodik et al. 2000, Singer 2006, Stephenson et al. 2000, Su and Wagner 2005, Wegman and Zadeck 1991], and, contrary to the full conversion, it uses a non-iterative algorithm, which is faster, as we empirically demonstrate. Third, our SSI conversion algorithm is a state-full black-box. Because we allow different clients invoking our converter in sequence, we log the SSI conversions that we perform, so that subsequent requests on the same variable do not lead to redundant work being performed.

Our SSI framework is now part of the default distribution of the Low Level Virtual Machine [Lattner and Adve 2004] (LLVM), version 2.6. LLVM is an industrial strength compiler, used by companies like Cray [1] and Apple [2]. We have implemented two SSI clients: the ABCD algorithm of Bodik *et al.* [Bodik et al. 2000], and a sparse conditional constant propagation (CCP) algorithm, similar to the one described by Singer [Singer 2006, p.59]. When compiling the SPEC CPU 2000 benchmark suite, the partial transformations that ABCD and CCP request are about 15 and 24 times faster than fully converting a program to SSI. The SSI conversion is based on the insertion of special instructions – σ-functions and ϕ-functions – in the source program. ABCD and CCP generate approximately 6.5 and 10 times less special instructions than the full conversion. We emphasize that the same infrastructure is used in the three transformations that we have compared: CCP, ABCD and full; however, because we build the SSI representation on demand, we give to each client only the program properties that it requires.

In Section 2 we review the SSI representation. In Section 3 we analyze the properties that many compiler optimizations previously described in the literature require from a program representation, and we show how different subsets of SSI suit these needs. In Section 4 we discuss our approach to build the SSI representation on demand. Section 5 validates our work with a series of experiments, and Section 6 concludes this paper.

2. Background on SSI

The term Static Single Information form seems to have been coined by Scott Ananian in his master thesis [Ananian 1999]; however, program representations with similar properties have been described before [Johnson and Pingali 1993]. For instance, the SSU-form

[1] `http://blogs.rapidmind.com/2009/05/27/why-we-chose-llvm/`

[2] `http://arstechnica.com/apple/news/2007/03/apple-putting-llvm-to-good-use.ars`

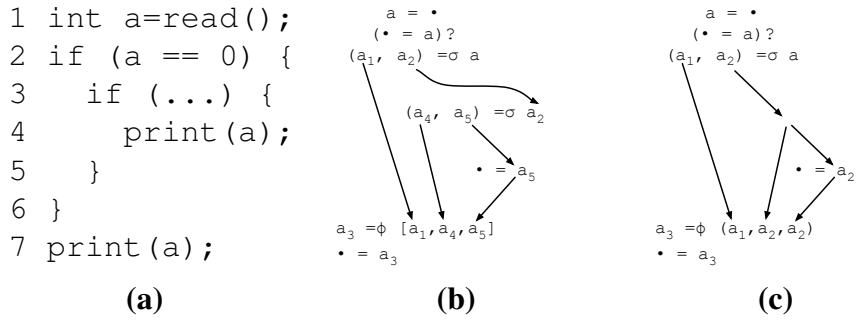

Figure 1. Example of the use of SSI on information analysis.

(Single Static Use), described by Plevyak in his Ph.D dissertation, [Plevyak 1996], seems to be equivalent to SSI, although we cannot verify this claim due to the lack of a formal specification of SSU. Boissinot *et al.* [Boissinot et al. 2009] distinguish two main flavors of the SSI form, which are not equivalent: *strong*, introduced by Ananian [Ananian 1999] and *weak*, described by Singer [Singer 2006]. Any strong SSI form program is also a weak SSI form program; thus, we will be using SSI as a synonym for Strong SSI. According to Boissinot *et al.*, four properties characterize strong SSI form:

- *pseudo-definition*: there exists a definition of each variable at the starting point of the program's control flow graph.
- *single reaching-definition*: each program point is reached by at most one definition of each variable.
- *pseudo-use*: there exists a use of each variable at the ending point of the program's control flow graph.
- *single upward-exposed-use*: from each program point it is possible to reach at most one use of a variable, without passing by a previous use.

Figure 1(a) shows a program written in a C like language, and Figure 1(b) gives the control flow graph of this program, in SSI form. The program in Figure 1(c) is not in SSI form, as it contains a point exposed to two different uses of a_2.

In order to convert a program into SSI form we need two special types of instructions: ϕ-functions and σ-functions. ϕ-functions are an abstraction used in the *Static Single Assignment form* (SSA) [Cytron et al. 1991] to join the live ranges of variables. Any SSI form program is a SSA form program. For instance, the assignment, $v = \phi(v_1, \ldots, v_n)$, at the beginning of a basic block B, works as a multiplexer. It will assign to v the value in v_i, if the program flow reaches block B coming from the i^{th} predecessor of B.

The σ-functions are the dual of ϕ-functions. Whereas the latter has the functionality of a variable multiplexer, the former is analogous to a demultiplexer, that performs an assignment depending on the execution path taken. For instance, the assignment, $(v_1, \ldots, v_n) = \sigma\ v$, at the end of a basic block B, assigns to v_i the value in v if control flows into the i^{th} successor of B. Notice that variables alive in different branches of a basic block are given different names by the σ-function that ends that basic block.

The insertion of ϕ and σ functions is a form of *live range splitting*. The live range of a variable is the set of program points where that variable is alive. Variable v is said to be alive at program point p if there is a path from p to a use of v that does not go through

any definition of v. Two algorithms for converting a program into SSI form have been described in the literature: we have Ananian's [Ananian 1999] pessimistic algorithm, and Singer's [Singer 2006] optimistic approach. We use the latter, as it subsumes the former.

There exists an interesting relationship between the live range of program variables and graphs. Chaitin *et al.* [Chaitin et al. 1981] have shown the intersection graph of the live ranges of a general program can be any type of graph. In 2005, researchers have shown that the intersection graphs produced from programs in SSA form are chordal [Bouchez 2005, Pereira and Palsberg 2005]. Recently, Boissinot *et al.* [Boissinot et al. 2009] showed that the interference graphs of programs in SSI form are interval graphs, a subset of the family of chordal graphs.

3. Examples of SSI Clients

This section shows examples of compiler optimizations that use the SSI representation, giving emphasis on the subset of SSI that each client needs. The SSI facilitates two types of program analyses. First, it helps analyses that extract information from conditional statements, such as constant propagation and array bound checks elimination. Second, it facilitates sparse backwards analyses that associate information with the uses of variables. Section 3.1 discusses examples in the former class, and Section 3.2 goes over the latter.

3.1. Information Analyses

Information analyses are among the main reasons behind the design of the SSI representation. These analyses use information from conditional branches to enable compiler optimizations, such as removing redundancies inserted to ensure language safety. For instance, Figure 2(a-c) shows three common Java idioms where exceptional cases are identified by the programmer via conditional tests. However, similar tests will be implicitly created by the java compiler to enforce the strongly typed nature of the language [Arnold et al. 2005]. In the figures, these tests appear in bold face. In another example, Figure 2(d) shows a Ruby program where a runtime test is used to handle integer overflows. The code in Lines 3-4 is implicitly performed, at runtime, by the Ruby interpreter; but, given the loop boundaries, this test will never be true.

Among the examples of redundant code elimination based on information analyses we cite Bodik *et al.*'s ABCD algorithm [Bodik et al. 2000] and the sparse conditional constant propagation method of Wegman and Zadeck [Wegman and Zadeck 1991]. Ananian describes a long list of information analyses when introducing the SSI representation [Ananian 1999]. Furthermore, many compilers already perform simple forms of redundant check elimination. For instance, LLVM is able to eliminate simple boundary checks inserted by the GNAT front-end used in the compilation of ADA programs.

In addition to removing redundant code, information analyses are also useful to detect security vulnerabilities in programs [Rimsa et al. 2010], and to discover the range of values that variables might assume. For instance, in Figure 2(a) we know that any value of variable `v` used in the true branch of the conditional is less than `v.length`. Examples of range analyses are the bitwidth inference engine of Stephenson *et al.* [Stephenson et al. 2000], the range propagation algorithm of Su and Wagner [Su and Wagner 2005], and the range analyzis used by Patterson to predict the outcome of branches [Patterson 1995].

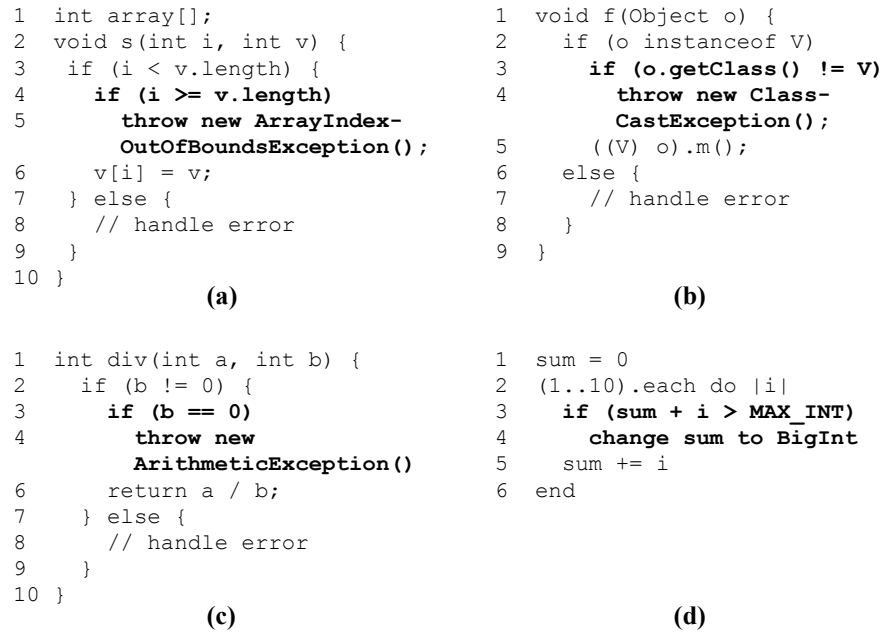

Figure 2. Examples of defensive programming idioms.

Although well known in the literature, these analysis and heuristics are described using different program representations, whereas they all can be elegantly modeled as constraint systems built on top of some *subset* of SSI form. In this sense, different clients have different needs, and not every variable in the source program needs to be transformed to meet the SSI properties. For instance:

- ABCD algorithm that removes redundant array bound checks [Bodik et al. 2000], as in Figure 2(a), requires only that variables used in conditionals, and that represent either array indices or array lengths have SSI properties;
- in order to remove redundant type casts, as in Figure 2(b), a client must require that variables used as operands of the `instanceof` function be in SSI form;
- in order to remove redundant divide-by-zero tests, as in Figure 2(c), we need that numeric variables used in conditionals be in SSI form;
- the ABCD version we implemented requires any variable used in branches to be in SSI form. So, the algorithm is used as a more general *redundant test elimination*, that allows, for instance, to remove the test in Line 3 in Figure 2(d).

In general, information analyses require that only variables used inside conditionals have the SSI properties. Furthermore, these variables do not have to show the SSI properties in all the program points where they are alive. Consider, for example, the program in Figure 1(a). It is possible to infer that the value of variable a, inside the innermost if statement, is always 0. An SSI based constant propagation analysis would produce the program in Figure 1(b), and would derive the constraints $\langle a_2 = 0 \rangle$ and $\langle a_4 = a_2 \rangle$. But the second constraint only exists because variable a_4 was created to guarantee the SSI property that no program point is reached by two different uses of the same variable. Thus, in order to avoid redundancies, we allow clients to specify the representation in Figure 1(c), which yields only the constraint $\langle a_2 = 0 \rangle$.

3.2. Backward Analyses

In addition to being useful for information analyses, the SSI representation also facilitates sparse backward analyses. Singer [Singer 2003] gives two examples of such analyses: very busy expressions, and the dual available expression analysis. An expression e is very busy at program point p if e is computed in any path from p to the end of the program, before any variable that is part of it is redefined. Such analysis, also called *anticipatable expressions analysis* by Johnson and Pingali [Johnson and Pingali 1993], is useful for performing optimizations such as partial redundancy elimination. Conversely, an expression e is *available* at program point p if it is computed in any path from the beginning of the program until p, and none of the variables that are part of e are redefined thereafter.

A sparse analysis associates information to variables, instead of program points. That is, busy expressions associated to variable v are the busy expressions at the definition point of v. Similarly, available expressions associated to v are the expressions available at the program point where v is last used. The SSI form allows us to perform these analyses non-iteratively [Singer 2003]. As another example, Boissinot *et al.* [Boissinot et al. 2009] have shown how SSI speeds up the computation of liveness analysis. This is a dataflow analysis that finds which are the live variables at each program point.

Contrary to the analyses described in Section 3.1, the backward dataflow analyses demand the full SSI representation. That is, every program variable must present the SSI properties discussed in Section 2. We show that the same infrastructure that supports information analyses also supports the backward analyses described in this section.

4. Building Partial SSI

We convert a program into SSI form on demand. This means that a client gives to our transformation pass a list of variables of interest, and we modify only these variables. There are two modes of conversion, *partial* and *full*. We convert a variable to SSI form via live range splitting and renaming. The difference between partial and full conversion is the amount of live range splitting required.

4.1. Converting a Program to Full SSI Form

If a variable is fully converted to SSI form, then it meets the definition of strong SSI form. This type of conversion is useful for the backward analyses described in Section 3.2. To perform the full conversion, we use the algorithm designed by Singer [Singer 2006, p.46]. This method, shown in Figure 3(a), combines Cytron *et al.*'s algorithm to insert ϕ-functions [Cytron et al. 1991], and Singer's algorithm to insert σ-functions [Singer 2006].

The insertion of σ-functions guarantees the single upward-exposed-use property. This phase happens as follows: for each use of a variable v, Singer inserts a σ-function in each basic block in the post dominance frontier of v. A basic block B_2 post-dominates a basic block B_1 if every path, from the exit of the source program to B_1 contains B_2. If B_2 post-dominates a predecessor of basic block B_0, but does not post-dominates B_0, then B_0 is in the post-dominance frontier of B_2. The σ-functions produce new uses of v, which cause the insertion of more σ-functions. This process iterates until a fix-point is reached.

The insertion of ϕ-functions, necessary to guarantee the single reaching-definition property, is the dual of the insertion of σ-functions, and it follows Cytron's algo-

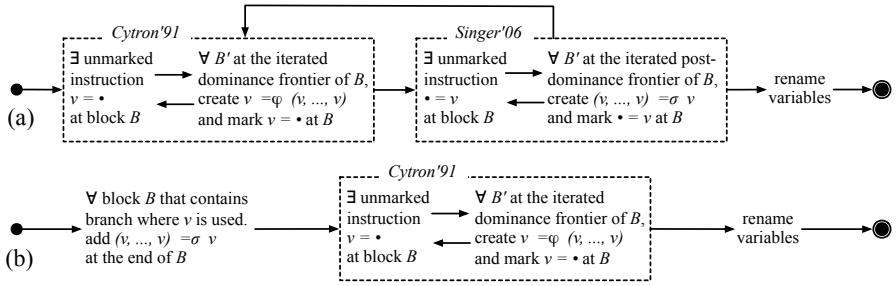

Figure 3. (a) Singer optimistic algorithm to convert a program into SSI form (b) Our algorithm to produce partial SSI form.

rithm [Cytron et al. 1991]. Whereas the insertion of σ's requires post-dominance frontiers and tracks uses of variables, the insertion of ϕ's uses dominance frontiers and tracks variable definitions. Iterations between the two boxes in Figure 3(a) happen because the insertion of σ-functions create new definitions of variables, and force a new round of placement of ϕ-functions. Additionally, the insertion of ϕ-functions also leads to the insertion of σ-functions, because it creates new uses of variables. Once a fix-point is reached, meaning that the properties stated in Section 2 have been attained, a renaming pass converts the program into SSI form.

4.2. Converting a Program to Partial SSI Form

The information analyses described in Section 3.1 do not require that variables be fully converted to SSI form. Instead, they need a representation that restricts the *value range* of variables. The value range of a variable is the set of values that the variable may assume during program execution. For instance, variable a in Line 1 of Figure 1(a) may assume any value of the integer type in the Java language, thus, its value range is $[-2^{31}, 2^{31} - 1]$. However, the conditional branch in Line 2 restricts the value range of a. Thus, in Line 3 of our example program, this range is $[0, 0]$. There exist two main events that may restrict the value range of a variable v: an assignment to v and a conditional branch that tests v.

In order to be in partial SSI form, a variable must meet four properties. Three of them were seen in Section 2: pseudo-definition, single reaching-definition and pseudo-use. We call the fourth property the single upward-exposed-conditional. This property, which is less general than Section 2's single upward-exposed-use, is stated as follows:

- *single upward-exposed-conditional*: if v is used at a branch instruction i, then from i it is possible to reach only one use of v without passing across another use.

Thus, in order to partially convert a variable v to SSI form we can add σ-functions at the boundaries of basic blocks that end with a conditional branch where v is used. Returning to our first example, variable a is fully converted to SSI form in Figure 1(b). On the other hand, the same variable is only partially converted to SSI form in Figure 1(c). The branch instruction $(\bullet = a)?$ is post-dominated by the use $(a_1, a_2) = \sigma\ a$.

The algorithm that we use to convert a program to partial SSI form is shown in Figure 3(b). This algorithm has lower complexity than Singer's, because the placement of σ-functions is simpler. In order to convert a variable v to partial SSI form, we loop over the uses of v, and for each use that is a conditional instruction, we create a σ-function in

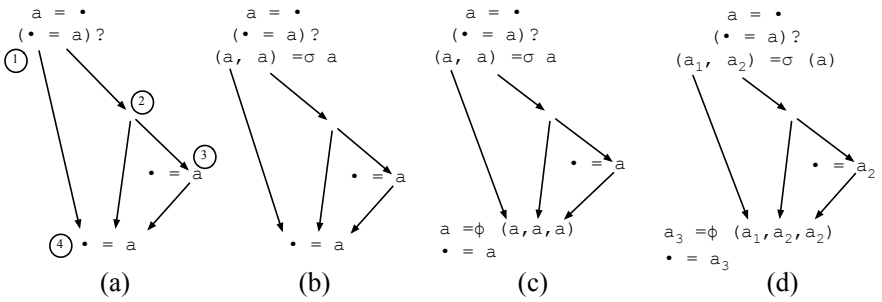

Figure 4. Partially converting a program for information analysis. Full is given in Figure 1(b).

the basic block that contains that use. Once all the uses of v have been visited, we proceed to the insertion of ϕ-functions. The placement of ϕ-functions is the same as in Singer's method, but in the partial transformation this phase happens only once.

Figure 4 illustrates these concepts. Figure 4(a) shows the control flow graph of the program used in Figure 1(a). We are interested in partially converting variable a into SSI form. Variable a is used in Blocks 1, 3 and 4. Only the first use is a branch, so we insert a σ-function after Block 1, (see Figure 4(b)). This σ-function defines two new instances of variable a, because Block 1 has two successors. After σ-functions have been inserted, we move on to the insertion of ϕ-functions. Since we now have two definitions of variable a reaching Block 4, we insert a ϕ-function in the beginning of this block, as shown in Figure 4(c). Finally, a renaming step will produce the program in Figure 4(d).

The algorithm in Figure 3(b) might insert more σ-functions than the minimal number necessary to guarantee the single-upward-exposed-conditional property. For instance, we would insert a σ-function after the conditional in Figure 4(a), even if there were no uses of variable a inside Block 3. In this case, variable a already has the single upward-exposed-conditional property, but our algorithm is unable to see this fact. Tracing an analogy with the SSA conversion algorithm, our method produces what we would call the "maximal" [Briggs et al. 1998, p.7] partial-SSI representation. We opted to build this simple representation, instead of the pruned form because the simpler approach is faster [Singer 2006]. Whereas the latter construction requires an analysis to identify which uses of variables reach branching points, the former simply inserts σ-functions after conditionals.

4.3. Complexity Analysis

The complexity of converting a single variable to SSI form, using the algorithm in Figure 3(a) is computed as follows. The complexity of a round of insertion of σ-functions, or ϕ-functions, is $O(B^2)$ [Cytron et al. 1991], where B is the number of basic blocks in the source program. But as empirically demonstrated [Singer 2006], this algorithm is $O(B)$ in practice. There are, indeed, true $O(B)$ algorithms for the placement of ϕ and σ-functions (see Sreedhar *et al* [Sreedhar and Gao 1995]). The maximum number of alternations between the insertion of ϕ and σ-functions is $O(B)$; so, the total complexity of the algorithm is $O(B^3)$.

The partial conversion has lower complexity. Inserting σ-functions is $O(U)$,

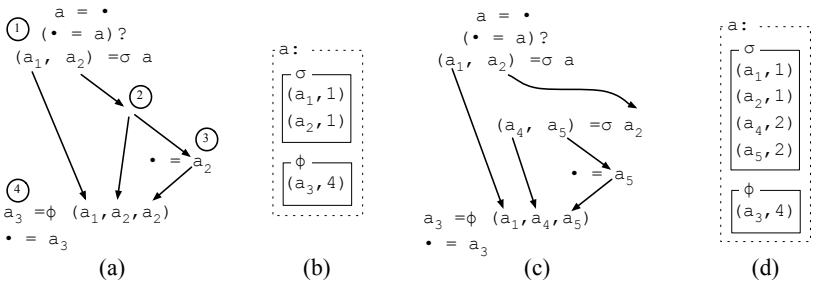

Figure 5. Preventing clients that run in sequence from performing redundant work.

where U is the number of conditional instructions using v. The complexity of inserting ϕ-functions is $O(B^2)$. As in the full conversion, it is $O(B)$ in practice. There is no alternation between the insertion of ϕ and σ-functions; thus, the total complexity of the partial conversion algorithm is $O(U) + O(B^2)$.

4.4. Orchestrating the Execution of Different Clients

A compiler might perform several passes on the same code, in order to carry out different optimizations. This includes the possibility of separate clients of our SSI transformation framework running on the same program. Hence, one of the objectives of our design is to allow clients to execute in sequence, without having to perform redundant work.

Our implementation guarantees that SSI clients running in sequence will never insert redundant σ or ϕ-functions into the source program. That is, let c_1 and c_2 be two SSI clients running in sequence. Lets assume that c_1 causes the insertion of σ_1 or ϕ_1 at program Point p_1 to transform a variable v. If c_2 also requests the conversion of v, leading to the insertion of another σ instruction at p_1, then nothing will happen, because our SSI converter knows that instruction σ_1 is already breaking the live range of v. To avoid redundancy, our SSI implementation keeps an internal state: it maps each variable to a table of pairs. Each pair consists of the identifier of either a σ or a ϕ-function, plus a program point. Figure 5 shows these concepts. Figure 5(b) shows the table created for variable a after some client requests the partial conversion of this variable, yielding the program in Figure 5(a). Once a second client requests the full conversion of a, we already know that no σ-function must be inserted at program Point 1. But the insertion of a σ-function at program Point 2 would lead to the creation of a ϕ-function at program Point 4. Again, we check a's table to avoid inserting a new ϕ-function. Upon discovering the instruction $a_3 = \phi(a_1, a_2, a_2)$, we change the two occurrences of a_2 to a_4 and a_5, as seen in Figure 5(c). Figure 5(d) shows the new table of variable a. Notice that this data structure is not essential to avoid inserting redundant σ and ϕ instructions; we could avoid it by looking at the current state of the intermediate representation. But it speeds up redundancy checks: if not for the data structure we would have to go through the parameters of ϕ and σ functions looking for occurrences of a variable before changing it.

5. Experimental Results

This section describes experiments that we have performed to validate our SSI framework. Our experiments were conducted on a dual core `Intel Pentium D` of 2.80GHz of

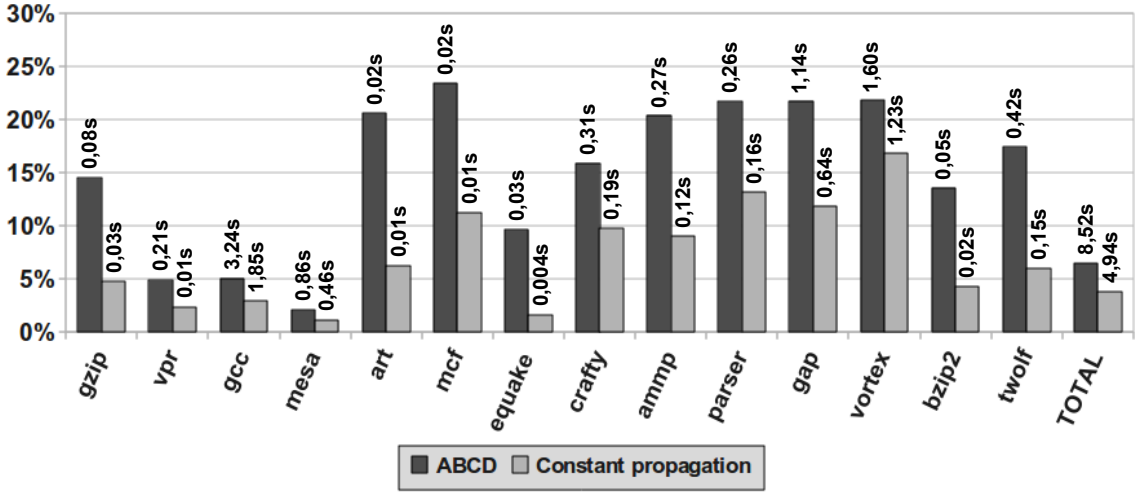

Figure 6. Execution time of partial compared with full SSI conversion. 100% is the time of doing the full SSI transformation. The shorter the bar, the faster the partial conversion when compared to the full conversion.

clock, 1GB of memory, running `Linux Gentoo`, version 2.6.27. Our framework runs in LLVM 2.5 [Lattner and Adve 2004], and it passes all the tests that LLVM does. The LLVM test suite consists of over 1.3 million lines of C code. In this paper we will be showing only the results of compiling SPEC CPU 2000. We will use three different clients of our SSI framework:

1. *Full*: converts a program to strong SSI form with the algorithm of Figure 3(a).
2. *ABCD*: generalizes ABCD algorithm for array bound checking elimination [Bodik et al. 2000]. We eliminate conditional branches on numeric inequalities that can prove redundant, such as the redundant tests in Figures 2(a) and 2(d).
3. *CCP*: does conditional constant propagation, that is, it replaces the use of variables that have a value range equal to a zero length interval $[c, c]$ by the constant c. As an example, this optimization replaces the use of variable a in Line 4 of Figure 1(a) by the constant 0. This client requires that only variables used in equality tests, e.g, ==, be converted to SSI.

When reporting the time of ABCD or CCP we show the time of running the algorithm in Figure 3(b). The time of performing redundant branch elimination or conditional constant propagation is not shown. Similarly, time reports for the full conversion include only the time to run the algorithm in Figure 3(a).

The chart in Figure 6 compares the execution time of the three SSI clients. The bars are normalized to the running time of the full SSI conversion. On the average, the ABCD client runs in 6.8% and the CCP client runs in 4.1% of the time of the full conversion. The numbers on top of the bars are absolute running times. The partial conversions tends to run faster in clients with sparse control flow graphs, which present fewer conditional branches, and therefore fewer opportunities to restrict the value ranges of variables.

Figure 7 compares the running time of our partial conversion algorithm with the running time of the `opt` tool. This tool is part of the LLVM framework, and it performs target independent code optimizations. `Opt` receives a LLVM bytecode file, optimizes it, and outputs the modified file, still in LLVM bytecode format. The SSI clients are `opt`

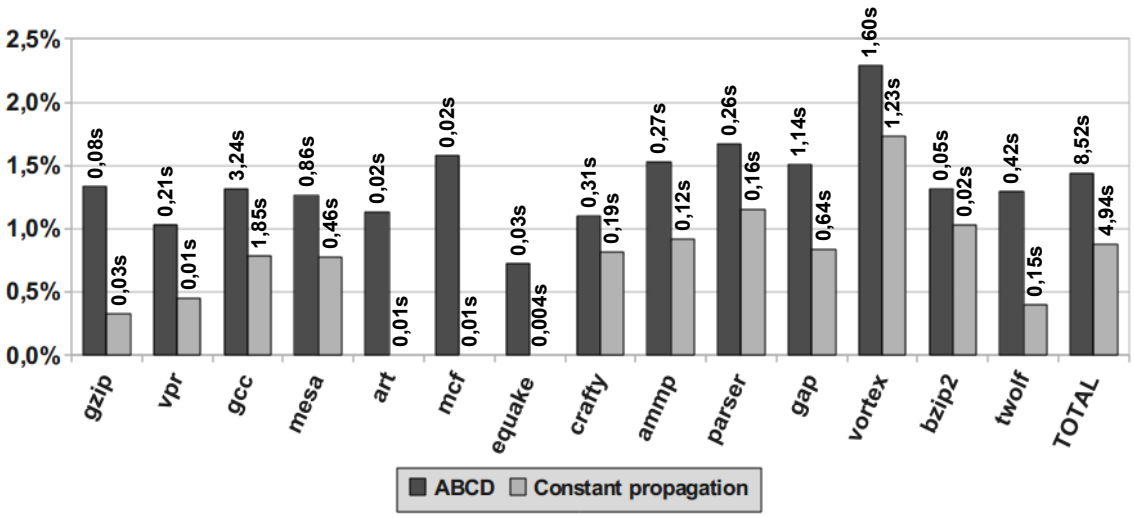

Figure 7. Execution time of partial SSI conversion compared to the total time taken by machine independent LLVM optimization passes (`opt`). 100% is the total time taken by `opt`. The shorter the bar, the faster the partial conversion.

passes. The bars are normalized to the `opt` time, which consists on the time taken by machine independent optimizations plus the time taken by one of the SSI clients, e.g, ABCD or CCP. Among the optimizations performed by `opt` we list partial redundancy elimination, unreachable basic block elimination and loop invariant code motion. The ABCD client takes 1.48% of `opt`'s time, and the CCP client takes 0.9%. To emphasize the speed of these passes, we notice that the bars do not include the time of doing machine dependent optimizations such as register allocation.

Figure 8 compares the number of σ and ϕ-functions inserted by the SSI clients. The bars are the sum of these instructions, as inserted by each partial conversion, divided by the number of σ and ϕ-functions inserted by the full SSI transformation. The numbers on top of the bars are the absolute quantity of σ and ϕ-functions inserted. The CCP client created 67.3K σ-functions, and 28.4K ϕ-functions. The ABCD client created 98.8K σ-functions, and 42.0K ϕ-functions. The full conversion inserted 697.6K σ-functions, and 220.6K σ-functions. There is an apparent mismatch between the number of instructions inserted and the time to do it. That is, in Figure 8 we see that about 10% to 15% of the nodes are inserted for ABCD and CCP when compared to full SSI. But in Figure 6 we see that this only takes 4% to 6% of the computation time. This fact happens because full SSI requires iterations between the insertion of σ and ϕ nodes, whereas the partial construction does not.

The chart in Figure 9 shows the number of σ and ϕ-functions that each SSI client inserts per variable. The figure emphasizes the difference between the partial conversion required by the two information analyses and the full SSI transformation. On the average, for each variable whose conversion is requested by either the ABCD or the CCP client, we will create 0.6 ϕ-functions, and 1.3 σ-functions. On the other hand, the full SSI conversion will insert 6.1 σ-functions and 2.7 ϕ-functions per variable.

Figure 10 shows the number of variables that have been transformed by each client. In two benchmarks, `gcc` and `vortex`, ABCD client has transformed more vari-

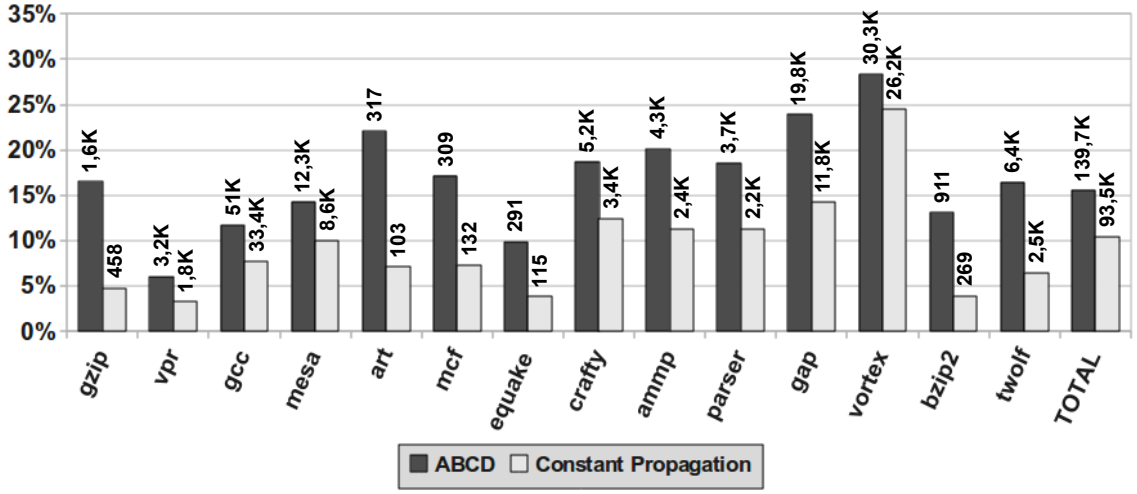

Figure 8. Number of ϕ and σ-functions produced by partial SSI conversion compared with full conversion. Values on top of bars denote absolute number of instructions. 100% is the number of instructions inserted by the full conversion.

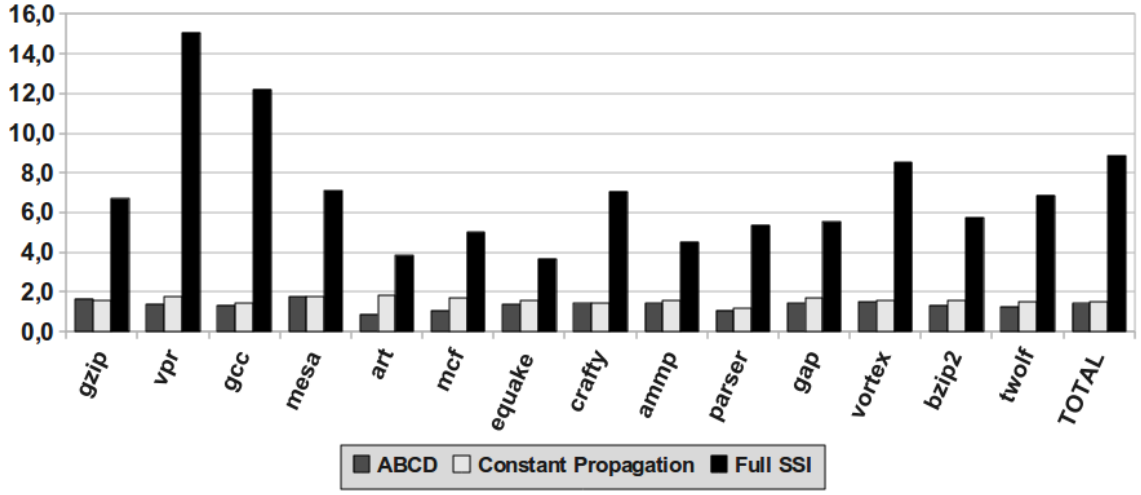

Figure 9. Average number of ϕ and σ-functions produced per variable.

	gzip	vpr	gcc	mes	art	mcf	eqk	crfty	ammp	par	gap	vor	bzp2	twolf	**Total**
ABCD	968	2K	38K	7K	361	292	211	4K	3K	3K	14K	20K	686	5K	**100K**
CCP	296	1K	24K	5K	56	78	74	2K	2K	2K	7K	17K	169	2K	**62K**
Full	1K	4K	36K	12K	376	360	807	4K	5K	4K	15K	13K	1K	6K	**101K**

Figure 10. Number of variables converted to SSI. We use shorter names.

ables than the full client. This fact happens because the full client transforms only variables that are alive across different basic blocks. ABCD and CCP clients, on the other hand, use the partial conversion algorithm from Figure 3(b), which converts variables used in conditionals, even when those variables are not alive outside the basic block where they are used. Notice that, in this case, we will not have any use of the variable after the conditional, and no σ or ϕ-functions will be inserted by the algorithm of Figure 3.

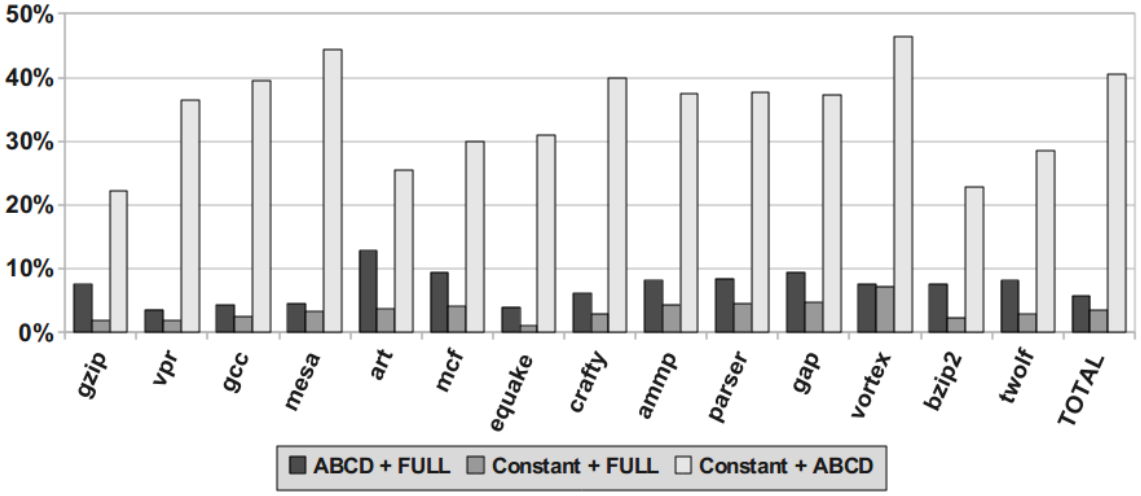

Figure 11. Percentage of σ and ϕ-functions saved by running clients in sequence.

The chart in Figure 11 compares the number of σ and ϕ-functions that we save by running different SSI clients in sequence. We compute the bars as follows. Let c_1 and c_2 be two SSI clients, such that c_1 inserts n_1 special instructions (ϕ or σ functions) into the source program, and c_2 inserts n_2. Let $n_{1,2}$ be the number of special instructions generated when both clients run in sequence. The bars represent the formula $1 - (n_{1,2}/(n_1 + n_2))$. This measure denotes the number of repeated instructions that are inserted by both clients running independently, and that are saved when these clients run in sequence. Our framework avoids the insertion of redundant instructions by keeping a record of variables that each client transforms, as described in Section 4.

6. Conclusion

This paper has presented the design and implementation of a SSI conversion framework, which is useful to several analysis and optimizations present in compiler back-ends. Our implementation differs from previous works because it allows the client to specify which variables should be converted into SSI. Furthermore, it allows a variable to be converted into SSI partially, that is, only at those program points where SSI properties are required. These two capacities of our framework allows it to be one order of magnitude faster than traditional approaches to SSI generation. Our implementation has been deployed on the LLVM compiler, and now is part of its official distribution.

Acknowledgement

This research has been supported by FAPEMIG, Edital 11/2009.

References

Ananian, S. (1999). The static single information form. Master's thesis, MIT.

Arnold, K., Gosling, J., and Holmes, D. (2005). *Java(TM) Programming Language, The (4th Edition) (Java Series)*. Addison-Wesley Professional.

Bodik, R., Gupta, R., and Sarkar, V. (2000). ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM.

Boissinot, B., Brisk, P., Darte, A., and Rastello, F. (2009). SSI properties revisited. Technical Report 00404236, LIP Research Report.

Bouchez, F. (2005). Allocation de registres et vidage en mémoire. Master's thesis, ENS Lyon.

Briggs, P., Cooper, K. D., Harvey, T. J., and Simpson, L. T. (1998). Practical improvements to the construction and destruction of static single assignment form. *Software Practice and Experience*, 28(8):859–881.

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

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490.

Johnson, N. and Mycroft, A. (2003). Combined code motion and register allocation using the value state dependence graph. In *CC*, pages 1–16. Springer-Verlag.

Johnson, R. and Pingali, K. (1993). Dependence-based program analysis. In *PLDI*, pages 78–89. ACM.

Lattner, C. and Adve, V. S. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE.

Patterson, J. R. C. (1995). Accurate static branch prediction by value range propagation. In *PLDI*, pages 67–78. ACM.

Pereira, F. M. Q. and Palsberg, J. (2005). Register allocation via coloring of chordal graphs. In *APLAS*, pages 315–329. Springer.

Plevyak, J. B. (1996). *Optimization of Object-Oriented and Concurrent Programs*. PhD thesis, University of Illinois at Urbana-Champaign.

Rimsa, A. A., D'Amorim, M., and Pereira, F. M. Q. (2010). Efficient static checker for tainted variable attacks. In *SBLP*. SBC.

Singer, J. (2003). SSI extends SSA. In *PACT (Work in Progress Session)*, pages XX–YY.

Singer, J. (2006). *Static Program Analysis Based on Virtual Register Renaming*. PhD thesis, University of Cambridge.

Sreedhar, V. C. and Gao, G. R. (1995). A linear time algorithm for placing ϕ-nodes. In *POPL*, pages 62–73. ACM.

Stephenson, M., Babb, J., and Amarasinghe, S. (2000). Bidwidth analysis with application to silicon compilation. In *PLDI*, pages 108–120. ACM.

Su, Z. and Wagner, D. (2005). A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computeter Science*, 345(1):122–138.

Wegman, M. N. and Zadeck, F. K. (1991). Constant propagation with conditional branches. *TOPLAS*, 13(2).