Technical Report

UCAM-CL-TR-660
ISSN 1476-2986

Number 660

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Static program analysis based on virtual register renaming

Jeremy Singer

February 2006

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted March 2005 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Christ's College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Abstract

Static single assignment form (SSA) is a popular program intermediate representation (IR) for static analysis. SSA programs differ from equivalent control flow graph (CFG) programs only in the names of virtual registers, which are systematically transformed to comply with the naming convention of SSA. Static single information form (SSI) is a recently proposed extension of SSA that enforces a greater degree of systematic virtual register renaming than SSA. This dissertation develops the principles, properties, and practice of SSI construction and data flow analysis. Further, it shows that SSA and SSI are two members of a larger family of related IRs, which are termed virtual register renaming schemes (VRRSs). SSA and SSI analyses can be generalized to operate on any VRRS family member. Analysis properties such as accuracy and efficiency depend on the underlying VRRS.

This dissertation makes four significant contributions to the field of static analysis research.

First, it develops the SSI representation. Although SSI was introduced five years ago, it has not yet received widespread recognition as an interesting IR in its own right. This dissertation presents a new SSI definition and an optimistic construction algorithm. It also sets SSI in context among the broad range of IRs for static analysis.

Second, it demonstrates how to reformulate existing data flow analyses using new sparse SSI-based techniques. Examples include liveness analysis, sparse type inference and program slicing. It presents algorithms, together with empirical results of these algorithms when implemented within a research compiler framework.

Third, it provides the only major comparative evaluation of the merits of SSI for data flow analysis. Several qualitative and quantitative studies in this dissertation compare SSI with other similar IRs.

Last, it identifies the family of VRRSs, which are all CFGs with different virtual register naming conventions. Many extant IRs are classified as VRRSs. Several new IRs are presented, based on a consideration of previously unspecified members of the VRRS family. General analyses can operate on any family member. The required level of accuracy or efficiency can be selected by working in terms of the appropriate family member.

Contents

Chapter 1

Introduction

Plug-and-play intermediate representations enable greater flexibility in data flow analysis. This *plug-and-play* concept may be implemented for sparse data flow analysis frameworks by varying the degree of virtual register renaming in the intermediate representation.

1.1 About this Chapter

1.1.1 Objectives

This chapter has three primary goals.

1. It introduces the main themes of program analysis and defines relevant terms which will be used throughout this dissertation.

2. It shows that static analysis research is still necessary in the light of recent computer architecture and programming language developments.

3. It sets out the background context in which the thesis is to be developed and understood.

1.1.2 Outline

Section 1.2 defines what is meant by *static analysis*, which is the broad subject of this dissertation. Static and dynamic analysis are briefly contrasted. Section 1.3 argues that static analysis is more relevant than ever before, in the current computational climate. Finally Section 1.4 outlines the contents of this dissertation.

1.1.3 Contributions

Apart from general introductory material, there is one key point in this chapter. Section 1.3.2 gives a balanced and informed discussion of Proebsting's law, which is a contentious issue at present. We reach a different conclusion to Proebsting.

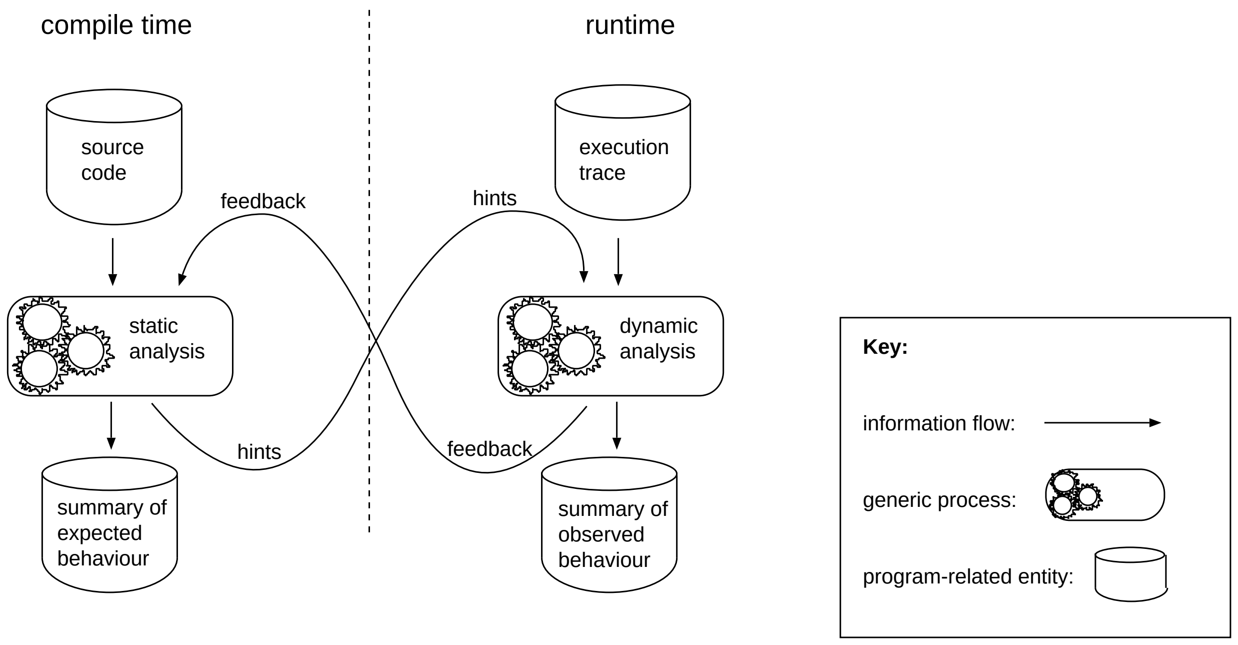


Figure 1.1: An overview of static and dynamic analysis and their interaction

1.2 What is Static Analysis?

The overall aim of program analysis is to identify and inspect the behaviour of subject programs. Program analysis techniques are classified as either *static* or *dynamic*. Static analysis occurs at compile time. Its objective is to predict how the analysed program will behave at runtime. In contrast, dynamic analysis occurs at runtime. Its objective is to report how the analysed program is behaving during runtime. Static analysis inspects a representation derived from the analysed program's source code. In contrast, dynamic analysis inspects a representation derived from the analysed program's execution trace. This dissertation focuses entirely on static analysis, although some of the techniques may also be applicable to dynamic analysis. Figure 1.1 presents a high-level view of the differences between static and dynamic analysis as outlined above. It also depicts the ways in which static and dynamic analysis may interact. Results from static analysis can be provided as 'ahead-of-time calculated data flow facts' (hints) for dynamic analysis. Results from dynamic analysis can be used to anticipate 'likely runtime behaviour' (feedback) for static analysis. These trends are becoming increasingly apparent, as Section 1.3 explains.

Note that many systems for dynamic analysis, such as Valgrind [Net04], have tunable accuracy. The level of detail recorded in the program execution trace can be altered freely but systematically within a uniform framework. The work of this dissertation is directed toward providing the same kind of systematically tunable accuracy within a static analysis framework. This is accomplished by varying the amount of detail expressed in the program representation.

Formally, static analysis is able "to predict safe and computable approximations to the set of values and behaviours arising when the program is executed" [NNH99]. *Safe* results ensure that the analysis always errs on the side of caution, so it possibly overestimates what ‘may’ happen at runtime and underestimates what ‘must’ happen at runtime. *Computable* results ensure that the analysis always terminates eventually.

Aho et al [ASU86] provide the classic presentation of static analysis. More recent treatments are available according to taste. Muchnick [Muc97] is extremely detailed, Appel [App98a] is simple and concise, whereas Nielson et al [NNH99] are particularly formal.

1.3 Motivation

This section discusses the necessity for ongoing static analysis research. Section 1.3.1 examines the urgent need in the present circumstances. Section 1.3.2 refutes the suggestions that static analysis is stagnant and should be abandoned.

1.3.1 Current Trends

There has never been a greater need for accurate and efficient static analysis than for optimizing compilers at the beginning of the 21st century. This section lists some compelling reasons.

1. Rumour suggests that the next generation of the industry standard compiler and systems performance benchmark suite, SPEC CPU 2005, will not permit feedback directed optimization techniques for the production of baseline performance measurements.[1] Feedback directed optimizations are presently permitted for SPEC CPU 2000 baseline measurements [Spe00b]. It appears that next generation compilers will be judged primarily on their static analysis capability, without the aid of additional information gleaned by feedback-directed techniques that incorporate dynamic analysis.

2. Despite the first point, there is an increasing level of synergy between static and dynamic analysis [Ern03]. This is largely fuelled by the growing popularity of just-in-time (JIT) compilation technology, for systems such as Java and .NET. Statically generated annotations (hints) may be inserted into the platform independent bytecode by the source to bytecode compiler. These hints enable the JIT compiler to optimize effectively at runtime. Azevedo et al [ANH99] describe such a system. Information revealed by a hint is normally too time-consuming for the JIT compiler to discover by itself at runtime. In this way, static analysis information may be used to improve the results of dynamic analysis. This is in contrast to feedback-directed optimization (outlined in the first point) in which dynamic analysis information is used to improve the results of static analysis.

3. The greatest incentive for high quality static analysis is the acceleration of hardware and software complexity. As both high-level source languages and low-level target languages become increasingly complicated, compilers must shoulder a growing burden of responsibility to handle the complex high-level source features, and to exploit

[1]This information was personally communicated by Vivek Sarkar, Senior Manager of the Programming Technologies department at the IBM T. J. Watson Research Centre. I pursued the matter with several members of the SPEC CPU committee, but they were unable to confirm or deny the rumour.

the complex low-level target features. These issues are developed further in the next section.

1.3.2 Discussion of Proebsting's Law

The popular interpretation of Moore's law [Moo65] is that microprocessor performance doubles every 18 months. Proebsting's law [Pro98] is a parody of Moore's law. Proebsting postulates that optimizing compiler technology improvements enable typical program performance to double every 18 years. The original basis for this claim was that optimizing compiler technology was approximately 36 years old (in 1998), and a typical C program for Intel's x86 architecture ran four times faster when compiled at maximum optimization level than when compiled without any optimization. Scott [Sco01] gives a more rigorous empirical justification of Proebsting's law. Similar observations hold for Java JIT compilation with the IBM Jikes RVM system [AAB+00]. Empirical data embedded in the source code (the `VM_CompilerDNA` cost/benefit model for adaptive optimization on x86) shows that programs are expected to execute 5.5 times faster when compiled using the aggressive optimizing compiler instead of the baseline non-optimizing compiler.

This potentially depressing observation has led some faint-hearted static analysis researchers (including Proebsting himself!) to suggest that static analysis research has stagnated, and should be abandoned in favour of more fashionable avenues of programming language research, such as programmer productivity. However, Proebsting's conclusion is flawed for the reasons listed below. Actually, research into optimizing compilation and static analysis is more vital than ever before.

Speed is no Longer the Only Goal

Compiler analyses and optimizations are not always intended to improve the speed of output executable code. An increasing trend has been to optimize code for execution in resource-constrained environments [KG03]. Such optimizations may be intended to reduce executable code size, runtime memory footprint, or runtime power consumption, for instance. In these cases, the standard static analysis techniques are applicable, in order to acquire the data flow information necessary for optimization.

Complicated Technology Needs (and Breeds) Complicated Compilers

It can be argued that compiler developers do well to keep up with the rate of microprocessor speed advances, rather than fall behind! In order to advance processor speeds in accordance with Moore's law, major 'under-the-hood' architectural changes have taken place. The processors of today are very different from processors of 36 years ago! For example, compare the Intel Pentium 4 of 2004 with the DEC PDP-10 of 1968. The Pentium 4 manual [Int04] comprises 4 volumes, with 2282 pages in all. In contrast, the DECsystem-10 processor manual [DEC82] has only 514 pages. This comparison hardly begins to indicate the quantum leap of complexity in processor design over 36 years. A Pentium 4 processor has the following features:

- three levels of high speed memory cache,
- deeply pipelined execution system,

- branch prediction,
- hardware register renaming,
- out-of-order speculative execution system,
- superscalar instruction issue,
- hyperthreading capability.

It is obvious that modern compilers are targeting very different kinds of hardware, for which code generation is much more complicated. There are two more issues that increase the requirement for even more complex static analysis.

1. In a bid to increase processor speeds by cutting complexity, recent 'very long instruction word' and 'explicitly parallel instruction computing' architectures offload the responsibility for extracting parallel computations from the hardware at runtime onto the compiler at compile time. In effect, these architectures have the same powerful features as the Pentium class listed above, but none of the inbuilt hardware mechanisms for dependence resolution and conflict avoidance.

2. As processor clock speeds increase, memory latency becomes a greater problem. The optimizing compiler faces increasing pressure to produce code with good memory access performance, which again requires extremely accurate static analysis.

On a different level, increasingly complex high-level programming languages require increasingly complex static analysis support. The trend in software engineering has been to develop code using more high-level programming languages. Recent control flow features include exception handling, virtual method calling and multi-threading. Recent data flow features include objects and genericity. Compilers have to analyse much more complicated high-level programs than 36 years ago! Hence a 2004 C++ compiler is much more complicated than a 1968 BCPL compiler. Despite this increased complexity, users expect compiled code to be more efficient than ever before. Thus static analysis research is necessary for the efficient compilation of new generations of high-level constructs.

Static Analysis is Everywhere

Static analysis techniques that were originally deployed in optimizing compiler technology have now been transferred to other areas of computer systems. For instance, some compiler innovations have been incorporated directly into recent microprocessor hardware. A few examples are listed below.

- The technique of virtual register renaming has been popularized by static single assignment form (SSA) [CFR+91]. However, Cooper and Torczon [CT04] mention that register renaming had been used for static analysis prior to SSA. Register renaming is now performed directly in hardware to avoid unnecessary data dependences in out-of-order and superscalar processors. (Note that the major theme of this dissertation is concerned with virtual register renaming for static analysis!)

- Data dependence analysis has always been an essential element of any static analysis tools. However in addition, dynamic data dependence analysis is now commonly performed directly in hardware to support out-of-order and superscalar execution, which require the extraction of instruction level parallelism on-the-fly.

- The static technique of if-conversion [AKPW83] transforms control dependences into data dependences. This is now supported directly in hardware by many architectures, using predicate registers and conditional execution of all instructions.

In addition, many static analysis techniques invented originally for optimizing compilers are also used in program verification tools. These include the Microsoft SLAM project [BR02] and Metacompilation [HCXE02]. Such systems use static analysis (often combined with other techniques) to detect violations of specified or inferred protocols in program source code. Returning for a moment to the theme of static and dynamic analysis synergy, several verification systems use dynamic analysis to detect likely invariant candidates, then use static analysis to determine whether these candidates are genuine invariants [HL02, ECGN01].

Therefore static analysis is not limited to the compiler as it analyses and generates executable code. Static analysis principles are also used by the hardware on which the code is executed, and the verifiers with which the code is debugged.

1.4 About this Dissertation

1.4.1 Objectives

This dissertation has two primary objectives.

1. It aims to demonstrate that static single information form (SSI) is a viable alternative to SSA and CFG (control flow graph) IRs for static analysis. The key point to demonstrate is the existence of a trade-off between accuracy and efficiency, dependent on the subject IR for static analysis. The analysis client has to decide where the balance lies before choosing the appropriate IR.

2. It aims to show that SSI and SSA are two members of a larger family of related IRs. This family requires exploration and formalization. The final result should be a sliding scale of IRs; with high accuracy but low efficiency analyses at one end of the scale, and low accuracy but high efficiency analyses at the other end. SSA and SSI must be somewhere in the middle of this scale.

1.4.2 Outline

Chapters 1 and 2 provide background material relating to static analysis and intermediate representations. After this, the dissertation falls into two main divisions.

The first part (Chapters 3–6) provides evidence in support of the first objective. Chapter 3 defines SSI, describes how to compute SSI from CFG, and contrasts SSI with both CFG and SSA. Chapter 4 demonstrates how to use SSI for several standard data flow analyses. The results clearly show that SSI enables more accurate analysis results than

SSA. Chapter 5 demonstrates how to use SSI for program slicing. SSI-based slicing gives greater efficiency over CFG-based slicing, and the same level of accuracy. Chapter 6 discusses extending the scope of SSI from single-procedure to whole-program. It shows that SSI can be used in four different ways to perform interprocedural analysis. In some of these approaches, SSI is a straight replacement for CFG or SSA. In other approaches, SSI has special properties that enable different kinds of interprocedural analysis altogether.

The second part of this dissertation (Chapter 7) presents the relevant material to support the second objective. It reviews and classifies various existing SSA extensions, then presents the family of *virtual register renaming schemes* to which SSA and SSI belong, together with many other IRs. Note that Chapter 7 is a natural development of the theme that is first introduced in Chapter 4, regarding the trade-off between accuracy and efficiency for data flow analysis, based on the selected IR for that analysis.

Finally Chapter 8 concludes.

1.4.3 Contributions

This dissertation makes four significant contributions to the field of static analysis research. These are listed below. Chapters 3–6 deal with the first three contributions. Chapter 7 deals with the fourth contribution.

1. It develops the SSI IR. Although SSI was introduced five years ago [Ana99], it has not yet received widespread recognition as an interesting IR in its own right. This dissertation presents a new SSI definition, construction algorithm and a number of applications. It also sets SSI in context among the broad range of compiler IRs.

2. It demonstrates how to perform existing static analyses using new, SSI-based techniques. Examples include live variables analysis, sparse type inference and program slicing.

3. It provides the first large-scale empirical evaluation of the merits of SSI for data flow analysis. Several studies in this dissertation compare SSI with SSA, CFG, and other existing IRs.

4. It identifies a family of virtual register renaming schemes, which are all CFG-like IRs. It shows how generic analyses can operate on any member of the family. The required level of accuracy or efficiency can be selected by working in terms of the appropriate IR family member.

These contributions back up the underlying thesis of this dissertation, which is repeated below for emphasis.

Plug-and-play intermediate representations enable greater flexibility in data flow analysis. This *plug-and-play* concept may be implemented for sparse data flow analysis frameworks by varying the degree of virtual register renaming in the intermediate representation.

A Note on Terminology

A few explanations of terminology are necessary at this stage, in order to clarify the above thesis. A *virtual register* is an abstract location that can hold a single scalar value. Virtual registers are introduced by a compiler when it generates intermediate code. They represent placeholders for physical registers or memory locations, by which they are replaced at code generation time.

When the thesis refers to *plug-and-play*, this is at the abstract level of data flow analysis algorithm rather than at the coal-face of actual implementation code. *Flexibility* mean varying levels of accuracy and efficiency, in this dissertation. Other parameters are possible however.

Many other terms are defined in the glossary at the end of the dissertation. Common acronyms are also expanded in the glossary.

Chapter 2

Background

This chapter presents a brief history of static analysis. It concentrates on the IRs in general rather than any particular analysis or transformation.

2.1 About this Chapter

2.1.1 Objectives

This chapter aims to review the field of IRs for static analysis. Over the last 50 years, many IRs have been developed and deployed in compilers. Since the research area is so large, the study must be a high-level overview. It will focus on a few IRs that are particularly relevant to modern static analysis systems. This chapter will briefly discuss the concept of *virtual register renaming*, which is the main theme of the dissertation.

2.1.2 Outline

Section 2.2 outlines the IR taxonomy. Sections 2.3 to 2.7 go through different taxonomy domains in detail. The order of presentation is from simplest to most complex IR, which (perhaps unsurprisingly) is also chronological. Section 2.8 describes several alternative methods for classifying IRs, including the important concepts of sparseness, live range splitting and virtual register renaming. Section 2.9 lists the different IRs that are covered in this dissertation, principally in their relation to static single information form. Finally Section 2.10 concludes.

2.1.3 Contributions

There are two key contribution in this chapter.

1. It presents a new taxonomic division of IRs. This is necessary in order to specify precisely the family of IRs that this dissertation will investigate.

2. It clarifies the relationship between live range splitting and virtual register renaming.

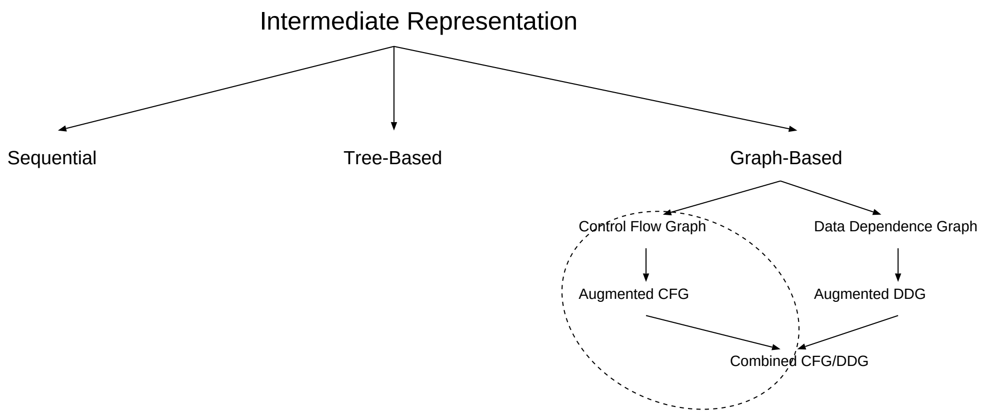


Figure 2.1: Taxonomy of IRs, region of interest is circled

2.2 IR Taxonomy

Figure 2.1 presents the IR taxonomy. The top-level division is based on a measure of the explicitly modelled relationships between program entities. These relationships will probably correspond to pointers in the underlying data structures. So, the top-level divides IRs into three domains (sequential, tree-based and graph-based). Subsequent sections describe each of these domains, in chronological order of their invention. Only the graph-based domain is expanded further in Figure 2.1, since the other domains are irrelevant for this dissertation. The circled region of the taxonomy indicates the taxonomic class of the IRs investigated in this dissertation.

Note that a graph-based IR may have an underlying sequential model of the program. For instance, control flow graphs often consist of basic blocks of sequential machine instructions. Thus, an IR may combine elements from different branches of the taxonomy. Analyses and transformations on this kind of hybrid IR should maintain consistency between these diverse elements at all times.

2.3 Sequential IRs

Assembly language was devised in the 1950's. This is the first sequential IR. In general, there is a one-to-one mapping between assembly language opcodes and machine instructions. This kind of IR is extremely low-level. Control flow is specified by a program counter register. Data flow is specified by value movement between physical machine locations.

Abstract assembly language for virtual machines is used in many low-level analyses. There are several different varieties. The most popular sequential IRs are stack-based and three-address code. Early examples of stack-based IRs include Pascal P-code [Nel79] and BCPL OCode [RWS79]. Java JVM code has sparked a recent revival in this area. Three-address code is epitomized by Muchnick's MIR [Muc97] and Cooper and Torczon's ILOC [CT04]. The GCC RTL IR is another good example.

Code generators perform peephole optimizations on sequential IRs and then macro-

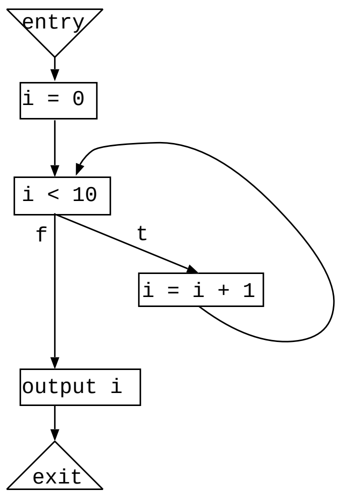


Figure 2.2: Example CFG program

expand each instruction into actual machine code.

2.4 Tree-Based IRs

Most early compilers were extremely simple. These date from the 1960's. The input programs were written in basic high-level languages such as Fortran and COBOL. Such compilers operate in a syntax-directed fashion, parsing source code to produce abstract syntax trees (ASTs). Basic tree-rewriting optimizations are performed, before the ASTs are transformed into the target assembly or machine code. However, ASTs provide no scope for more powerful global data flow analysis [Sch73]. (Note that global analysis is now referred to as *intraprocedural* analysis.) A richer IR is required for such more complex kinds of analysis. Generally, modern compilers generate ASTs from source code, then use this information to construct a graph-based IR, as described in the next section.

2.5 Early Graph-Based IRs

In the 1970's, as computing technology became more powerful, larger memory and faster processing speeds enabled more complex kinds of IRs. A typical compiler from this era transforms ASTs into a graph-based representation, which is then employed as the standard IR for further analysis and transformation, culminating in code generation.

2.5.1 Control Flow Graph

The most widely used graph-based IR is the control flow graph (CFG). CFG is the basis for almost all classical data flow analysis techniques. (In the context of static analysis, the classical age runs from 1970–1990.) The classical description of CFG is given by Aho et al [ASU86]. Nielson et al [NNH99] present a more rigorous overview. Figure 2.2 shows an example CFG program.

Formally, a CFG $G = (N, E, n_{\text{entry}}, n_{\text{exit}})$ with $n_{\text{entry}}, n_{\text{exit}} \in N$ and $E \subseteq N \times N$. N is the set of nodes in G. n_{entry} is the distinguished entry node for G. It has no predecessors. n_{exit} is the distinguished exit node for G. It has no successors. CFG is an intraprocedural representation, so n_{entry} corresponds to the unique procedure entry point and n_{exit} corresponds to the unique procedure exit point. Each node is a basic block. A *basic block* is a sequence of consecutive computational instructions, $i_1, \ldots, i_m$ with the property that every instruction i_j has a unique successor instruction i_{j+1} for $1 \leq j < m$ and every instruction i_k has a unique predecessor instruction i_{k-1} for $1 < k \leq m$. Note that in the simplest case, each instruction can map to a unique basic block. This dissertation assumes that the computational instructions are low-level machine instructions, operating in terms of virtual registers and abstract memory locations. This is similar to the sequential IRs outlined in Section 2.3. E is the set of directed edges in the CFG. An edge $e = (n_i, n_j)$ indicates that control may flow directly from the last instruction of n_i to the first instruction of n_j during program execution.

This dissertation imposes the standard CFG *reachability constraint*: For all nodes $n \in N$, there exists a control flow path from n_{entry} to n; for all nodes $n \in N$, there exists a control flow path from n to n_{exit}. An arbitrary CFG can be rewritten to this standard form using two techniques.

1. Unreachable code elimination removes nodes that are not on a control flow path from n_{entry}.

2. Insertion of 'impossible' loop exit edges breaks infinite loops that are not on a control flow path to n_{exit}. Such impossible edges may be required for data flow analysis, even if the relevant loops will never terminate at runtime [HS02].

There is a vast body of literature dealing with static analysis of CFG. CFG is the standard IR used in monotone data flow frameworks, which for many years has been the prevalent compiler technique for analysis and optimization. Refer to the bibliographic notes at the end of Chapter 10 in the Dragon book [ASU86] for more details.

CFG is an executable representation, since it contains enough information to permit execution by simple interpretation. It follows that code generation from CFG is straightforward. Each node corresponds to a basic block of code, and each edge corresponds to a jump (program counter update) to a new block of code.

2.5.2 Data Dependence Graph

The data dependence graph (DDG) is sometimes also known as the data flow graph. DDG is based on an entirely different model of computation from CFG, in that it is data flow oriented, rather than control flow oriented. DDG has been widely investigated in the static analysis research community [AK02].

DDG represents the flow of values from their creation (definition site) to their consumption (use site). Figure 2.3 shows an example DDG program. This is the DDG version of the CFG program in Figure 2.2.

Formally, a DDG $G = (N, E)$ where $E \subseteq N \times N$. N is the set of nodes in the DDG. A DDG node generally corresponds to a single instruction from a CFG basic block. Sometimes, DDG nodes may represent more primitive operations and operands. E is the set of directed edges in the DDG. Each edge (n_j, n_k) connects a value creation

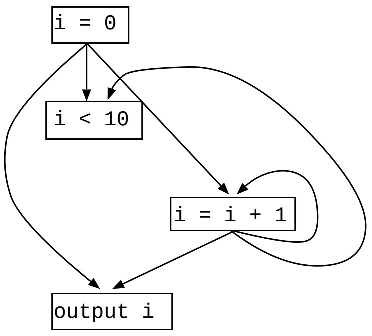


Figure 2.3: Example DDG program

to a value consumption. A DDG edge generally corresponds to a virtual register use in a CFG basic block.

DDG specifies certain constraints on the ordering of operations at runtime as dictated by the data flow behaviour of a program. However DDG does not fully capture the control flow of a program. Thus it is not an executable representation as it stands. It is possible to convert control flow information into data dependence information (using if-conversion [AKPW83]). Nevertheless it is still necessary to linearize the DDG code back into sequential control flow oriented code at the code generation phase.

DDG is commonly used in compilers for instruction scheduling tasks such as loop reordering, auto-parallelization, and code generation for a pipelined architecture or a multiple issue architecture. Actually, most optimizing compiler transformations require dependence information [BGS94].

2.6 Recent Graph-Based IRs

Since the late 1980's, newly developed IRs have had a hybrid form, incorporating elements of both CFG and DDG. This section classifies these hybrid IRs into either augmented CFG or augmented DDG. This is a new division proposed in this dissertation, but it should seem intuitively appealing.

2.6.1 Augmented CFG

Many more recent IRs resemble CFG with varying quantities of supplementary data dependence information. The first such IR was CFG supplemented by def-use chains [ASU86], which are edges that directly connect virtual register definitions to virtual register uses. A *def-use chain* connects a single definition d of virtual register x to all points u_i that use x, such that each u_i is reached by d. A definition d of virtual register x *reaches* a point p if there is a control flow path from d to p, such that x is not redefined along that path. A *use-def chain* is the dual of a def-use chain. A use-def chain connects a single use u of virtual register x to all reaching definition sites d_i that define x.

Dependence Flow Graph

The dependence flow graph (DFG) [PBJ+91, JP93] is an improved version of def-use chaining. DFG has explicit control flow edges, and explicit def-use edges, but def-use edges are factored at control flow split and merge points, which reduces the expense of def-use chaining. Later chapters describe different aspects of DFG in further detail.

WEB

Def-use-use-def-webs form (WEB) is another extension of def-use chaining. Muchnick [Muc97] describes WEB, and shows how it can be useful for register allocation by graph colouring. A *def-use-use-def-web* is the maximal union of def-use chains that share a common use, i.e. def-use chains c_1 and c_2 are in the same web if they both contain a common use, or if there is another chain c_3 that contains a use in common with c_2 and c_3 is in the same web as c_1. Rather than constructing explicit WEB data structures, WEB information can be made implicit in the CFG by *renaming* virtual registers so that they have a common name if and only if they belong to the same web. The Sable Java static analysis suite [QHV00] uses WEB in its Jimple IR.

Static Single Assignment Form

The most popular augmented CFG IR is static single assignment form (SSA) [CFR+91]. SSA encodes data dependence information in the names of virtual registers. Thus, SSA is a CFG with def-use relationships encoded in the virtual register naming convention, like WEB. In a similar way to DFG, def-use chains are factored at control flow merge points (but not split points) to avoid expense [SGW94]. The underlying principles of SSA are developed and extended throughout this dissertation, so it is important to clarify SSA fully in this section.

SSA is CFG with an extra constraint: Each virtual register must have a unique (hence *single*) definition point (hence *assignment*) in the program text (hence *static*). For a SSA program to be valid, each unique definition of a virtual register v must dominate all uses of v. Informally, this means that every virtual register must be defined before it can be used. The concept of dominance is defined formally in Section 3.3.

Two steps are required to transform an arbitrary CFG program so that it satisfies the SSA property:

1. insert pseudo-definitions at control flow merge points in CFG where multiple definitions for a single virtual register will converge, and

2. rename virtual registers so that each definition creates a new name.

Section 3.6.2 provides more information about typical SSA construction algorithms.

SSA pseudo-definitions are called ϕ-functions. They always occur at the start of basic blocks. If a basic block b has n control flow predecessors, then a ϕ-function p belonging to b will have n source operands. Generally, p takes the value of its ith source operand when control flows to basic block b from b's ith predecessor block. Because of this dependence on control flow information, ϕ-functions are not referentially transparent, or directly interpretable. Some additional representation of control flow information is

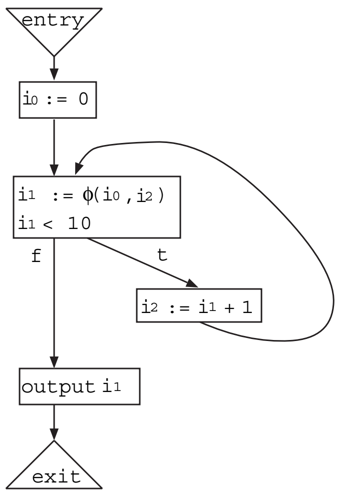


Figure 2.4: Example SSA program

necessary to support redundancy elimination of ϕ-functions [TP95], or SSA interpretation [vWF04].

Figure 2.4 shows an example SSA program. This is the SSA version of the CFG program in Figure 2.2.

The seminal description of SSA is by Cytron et al [CFR+91]. They state that SSA concepts originated in the early work of Shapiro and Saint [SS70]. SSA is the basis for many optimizations, including constant propagation [WZ91], value numbering [RWZ88], and partial redundancy elimination [KCL+99].

Many modern compilers use SSA as their primary IR for static analysis. These include GNU's GCC [Nov03] and Harvard's Machine SUIF [Hol01]. Some JIT compilers also use SSA for heavyweight on-the-fly analysis and optimizations, including Sun's Java HotSpot [Sun99] and IBM's Java Jikes RVM [AAB+00].

SSA is sometimes referred to as 'factored use-def chains' representation [SGW94]. This is because uses can only be reached by one definition in SSA. If a use was reached by more than one definition in the original CFG program, then those multiple reaching definitions are factored into one definition at a ϕ-function in the SSA program. Each SSA use-def chain is a fragment of a use-def chain from the original program. SSA ϕ-functions mark where an original use-def chain has been split. Thus a WEB program can be constructed from a SSA program by renaming virtual registers such that for each ϕ-function $r_0 \leftarrow \phi(r_1, \ldots, r_n)$ in the SSA program, all occurrences of $r_0, r_1, \ldots, r_n$ must be replaced by a fresh virtual register name r in the new WEB program; also all ϕ-functions must be removed.

2.6.2 Augmented DDG

Program Dependence Graph

The most notable augmented DDG representation is the program dependence graph (PDG) [FOW87]. PDG represents a program as a graph in which nodes are instructions, and directed edges represent dependences between instructions. There are two different

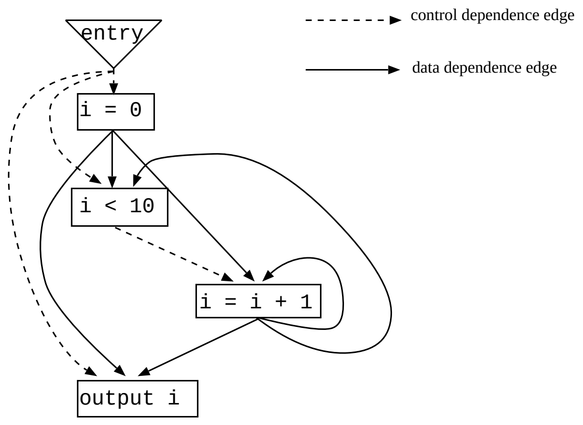


Figure 2.5: Example PDG program

types of edge: data dependence and control dependence. Data dependence edges are the same as edges in the original DDG. Actually, there are three different varieties of data dependence edge, as Section 5.7.2 explains in detail. DDG edges correspond to PDG *flow* dependence edges. Control dependence edges represent essential control flow ordering information. A control dependence edge (n_1, n_2) indicates that the execution of n_2 depends on the outcome of n_1, so n_1 must be executed before a decision can be made about whether to execute n_2. Parallelism is exposed in the PDG, since the only constraints on code ordering are indicated by the dependence edges. Unlike CFG, there are no artificial control flow constraints imposed by the need for a total ordering on instructions.

PDG is generally constructed from a program in CFG form. Data dependence edges are calculated using standard data flow analysis to determine reaching definitions information. Control dependence edges are calculated using dominance information. Figure 2.5 shows an example PDG program. This is the PDG version of the CFG program in Figure 2.2.

Ferrante et al define PDG, give an algorithm for its construction and discuss several applications [FOW87]. Cartwright and Felleisen develop a formal PDG semantics [CF89]. Ramalingam and Reps have a similar semantic formalism [RR89]. There are several PDG-based compilers (for instance, *pdgcc* [NP94]) developed for research purposes, but PDG does not seem to have been adopted by the mainstream compiler community. The primary PDG application at present is program slicing, which Chapter 5 describes in detail.

Program Dependence Web

The program dependence web (PDW) [BMO90] is presented as an extension to PDG that can be directly interpreted. PDW also incorporates single-assignment ideas from SSA, by representing program instructions in gated single assignment form (GSA). GSA is similar to SSA, except that each pseudo-definition function that merges multiple incoming definitions is *gated*, i.e. it has an extra argument that specifies which of the source operands should be assigned to the destination operand. This simplifies the presentation of control dependence, and enables efficient PDW interpretation and code generation. GSA is also developing as a distinct IR from PDW [Hav93, TP95].

Value Dependence Graph

The value dependence graph (VDG) [WCES94] is a functional IR that expresses computation solely as value flow. CFG-based IRs are statement based and name all values. PDG and PDW do the same. In contrast, a VDG program only specifies the flow of values through a computation. There is no superfluous information concerning names, or the order in which values are computed. (In effect, VDG edges correspond to uses of CFG virtual register names.) VDG has a demand-based semantics, so a value is only computed if it is needed by another computation.

The value state dependence graph (VSDG) [JM03] combines VDG and GSA. VSDG also has state dependence edges, to enforce sequentialized computation. These can be used to express store dependencies (ordering of writes to memory) as well as loop termination dependencies (to ensure that a non-terminating loop can cause the program to loop forever, even when the final result is not data dependent on any values computed by the loop). In addition, state dependence edges can be used to model artificial constraints on control flow. Thus it is possible to create a CFG-like total ordering on operations in the program.

2.7 Complete Combination of Augmented CFG and Augmented DDG

The augmented CFG IRs in Section 2.6.1 add data dependence information to CFG. The augmented DDG IRs in Section 2.6.2 add control flow information to DDG. At this point, an obvious question arises: Is there a least upper bound (lub) for these hybrid IRs? If so, what this lub IR? There appear to be three distinct relations between nodes, where nodes are either instructions or values.

1. The *control flow* relation, epitomized by CFG edges.

2. The *data dependence* relation, epitomized by DDG edges.

3. The *control dependence* relation, epitomized by PDG control dependence edges.

Note that control dependence information is distinct from control flow information. While it is true that control dependence can be computed from a knowledge of the control flow relation, this computation is expensive, therefore it may be best to factor this computation into the IR construction costs.

So, the lub IR must encapsulate all three kinds of relation between nodes. *Static single information form* (SSI) is such an IR. Chapter 3 introduces SSI properly. SSI encodes control flow explicitly using edges, in the same way as SSA and CFG. So at first sight it appears that SSI is simply another augmented CFG IR. SSI also encodes data dependence implicitly in its virtual register naming convention, in the same way as SSA. However SSI also encodes control dependence implicitly in its virtual register naming convention. Chapter 5 gives full details of this encoding. So, SSI is one instance of this lub. Most of this dissertation focuses on the the characteristics of SSI, and SSI-based analysis.

2.8 Alternative Classifications

The top-level division for the taxonomy presented in Section 2.2 may seem arbitrary. It is similar to the scheme suggested by Cooper and Torczon [CT04]. However, there are many other classifiers for IRs. This section briefly reviews several alternative classifiers and shows how they relate to the IRs in this dissertation.

2.8.1 Sparseness

In recent years, *sparse* IRs have become popular in the static analysis community. Broadly speaking, a sparse IR connects data flow information creation sites (generally referred to as definitions, since most sparse analyses are forward, virtual register based analyses) directly to data flow information consumption sites (generally referred to as uses). Analysis of sparse IRs is therefore extremely efficient, since analyses only compute data flow information exactly when it is needed and save it exactly where it is needed. This is very different to the dense, or *classical*, data flow analysis techniques described by Aho et al [ASU86].

Ruf [Ruf95b] distinguishes between *analysis-specific* and *general* sparse IRs

- Analysis-specific sparse IRs are specialized to handle only one particular data flow analysis. They do not retain the full semantics of the original program, rather they only represent sufficient information to model the particular data flow properties under consideration. The sparse evaluation graph [CCF91] is an example analysis-specific sparse IR.

- General sparse IRs completely describe a program's behaviour, since they retain the full semantics of a program's execution. Any data flow analysis can be applied to a general sparse IR. WEB SSA and SSI are all examples of general sparse IRs.

This dissertation focuses on general sparse IRs.

2.8.2 Live Range Splitting

A virtual register v is *live* at entry to node n if there is a control flow path from n to some other node which uses v, and there is no definition of v along that path. A virtual register v is live along edge e if v is live at entry to the destination node of e. Let G be the subgraph[1] (V_L, E_L) of CFG nodes and edges where v is live. Then each connected component within G comprises a distinct *live range* of v. The notion of live range does not appear to be clearly defined in the literature. Whenever this dissertation refers to a live range, it conforms to the above definition.

Live range splitting [CH90] enables reasoning about virtual registers at a finer granularity than would be possible if the original high-level names were retained. Both register allocation and data flow analysis can be improved by this method. Briggs' empirical study shows that register allocation can be ameliorated since a greater degree of live range splitting enables better packing of registers during the colouring phase [Bri92]. Disjoint live

[1] Technically G is not a graph since $E_L \not\subseteq V_L \times V_L$, but G becomes a graph by forming the closure: $(V_L \cup \mathit{source}(E_L), E_L)$, where $\mathit{source}(E_L) = \{v_1 | (v_1, v_2) \in E_L\}$. Note that Alan Mycroft helped me to formulate this definition.

ranges of the same variable have different names in the normalized form, and thus each can be allocated a different physical register. This reduces the range over which a single virtual register has to map onto a fixed physical register, easing register pressure. However, reduced register pressure is achieved at the cost of using a larger number of virtual registers during allocation. An increased number of virtual registers degrades the efficiency of the allocation process.

Live range splitting is also beneficial for sparse data flow analysis. Sparse data flow analysis generally associates information with a particular live range, rather than with a particular CFG node. Empirical results in this dissertation show that a greater degree of live range splitting enables a finer granularity of analysis precision. In a sense, this is a generalization of Briggs' observation for register allocation. We generalize 'register colour' to generic data flow information. However, Section 4.6.1 argues that for every data flow analysis, there is 'saturation point' in live range splitting. Beyond this point, further live range splitting cannot improve the accuracy of sparse analysis.

Note that the process of live range splitting is generally carried out by *virtual register renaming*. The next section examines this topic in some detail.

2.8.3 Virtual Register Renaming

The concept of *virtual register renaming* is most important in this dissertation. WEB, SSA and SSI are examples of *virtual register renaming schemes* (VRRSs), with particular constraints to satisfy. However there are many other VRRSs belonging to the same family. These are all augmented CFG IRs, with extra information encoded in the virtual register naming convention. Specific pseudo-definitions (like SSA ϕ-functions) may be required with different properties to handle the idiosyncrasies of each particular VRRS.

If live ranges are modelled so that each live range has a distinct virtual register name, then virtual register renaming enables two kinds of live range splitting. First, virtual registers may be renamed so that each definition is an initial point of a live range. This prevents definitions of the form $v \leftarrow f(v)$, that would extend the duration of the live range of v. (WEB performs only this kind of live range splitting.) Second, an existing live range may be split into two or more subranges by inserting an appropriate pseudo-definition (effectively one or more virtual register clone operations) at a point within the existing live range. (SSA and SSI perform both kinds of live range splitting.) WEB, SSA and SSI IRs all have a one-to-one mapping between virtual register names and live ranges. Thus sparse data flow analysis for these IRs associates information with each virtual register.

2.8.4 Discussion

This section classifies the different IRs in terms of the shapes of live ranges enforced. WEB allows multiple definitions and multiple uses in a single live range. So a WEB live range may have multiple initial points in the CFG and multiple terminal points. SSA restricts live ranges to a single definition, but multiple uses are allowed. So a SSA live range must have a single initial point and may have multiple terminal points. SSI restricts live ranges to a single definition, and a linear sequence of uses such that if the definition is executed, all the uses will be executed as well. So a SSI live range looks like a single-entry-single-exit region [JPP93] in the CFG, with the single initial point being the definition, and the

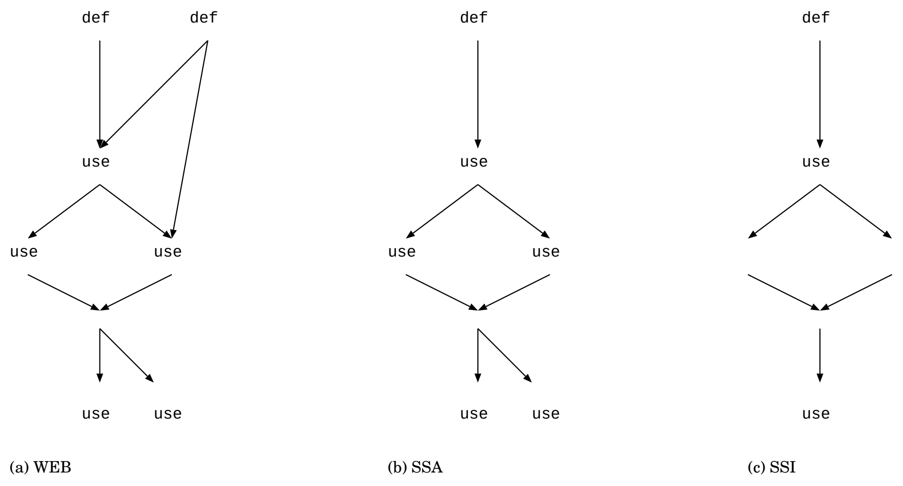


Figure 2.6: Example live ranges in each sparse IR

single terminal point being the last use. Figure 2.6 shows an example live range in each sparse IR.

With reference to the alternative classifiers outlined above, this dissertation concentrates on general sparse IRs that perform live range splitting via virtual register renaming. Data flow analysis on these IRs achieves sparseness by associating data flow information with live ranges (equivalently, virtual register names, since each name encapsulates a static single live range).

2.9 IRs in this Dissertation

This dissertation explores augmented CFG IRs, particularly those that encode additional information by virtual register naming conventions like SSA. The first part of the dissertation (Chapters 3–6) concentrates on *static single information form* (SSI), which is an extension of SSA. The second part of the dissertation (Chapter 7) shows that both SSA and SSI are instances of a more general family of virtual register renaming schemes (VRRSs).

There is no single 'related work' section, comparing SSI with existing IRs. Instead, each chapter compares SSI with relevant IRs at that point. A full list of comparisons is given below.

SSI versus ...	*Chapter(s)*
SSA	3, 4
CFG	3, 4, 5
WEB	4
PDG	5
DFG	3, 5
continuation passing style	6
all VRRSs	7

2.10 Concluding Remarks

To avoid unnecessary complication, this chapter has concentrated on the character of the actual IRs used for static analysis, rather than the details of any particular analysis. Other parts of this dissertation adopt the same policy, most notably Chapter 6. In a sense, each IR determines the style of analysis that can be performed on that IR. The details should be in the IR itself, and the actual analysis simply leverages the information provided by the IR. As Raymond says [Ray99], "Smart data structures and dumb code works a lot better than the other way around."

Nevertheless, a static analysis is characterized by more than the IR in isolation. A *data flow framework* specifies all the factors that might affect the accuracy and efficiency of an analysis. This dissertation seeks to avoid lengthy discussion of the framework details except where absolutely necessary. A crucial point is now stated, which will be most relevant when we consider the family of related IRs in Chapters 4 and 7: For a given analysis (such as constant propagation), apart from changing IR, if the other parameters in the data flow framework are unaltered, then analysis performance depends completely on the expressivity of each particular IR. The analysis is parameterized on the IR alone, so that a single analysis can operate on multiple similar IRs, such as WEB SSA and SSI.

The next chapter reviews SSI in exhaustive detail.

Chapter 3

Static Single Information Form

Static single information form is an augmented CFG IR, which is similar to SSA. Static single information form may be constructed efficiently from CFG.

3.1 About this Chapter

3.1.1 Objectives

Static single information form (SSI) is a recently proposed IR for data flow analysis of imperative programming languages. It was introduced by Ananian in 1999 [Ana99]. This chapter reviews the basic concepts of SSI. However SSI has not been widely adopted so far by the static analysis community. This chapter aims to tackle this issue by providing:

1. a new definition of SSI that is more concise than the original definition, and
2. a new construction algorithm for SSI that is shown to be more efficient for real-world programs than the original algorithm, and
3. a detailed comparison of SSI and its more popular relative, SSA.

3.1.2 Outline

Section 3.2 informally introduces SSI, and reviews its history. Section 3.3 presents a new and concise definition of SSI. Section 3.4 discusses the properties of SSI. Section 3.5 compares two algorithms for SSI construction. Sections 3.6 and 3.7 highlight related work and future work respectively. Finally Section 3.8 concludes.

3.1.3 Contributions

This chapter makes three key contributions.

1. Section 3.3 gives a new and succinct definition of SSI.
2. Section 3.5.2 presents a new algorithm for the efficient construction of SSI programs from CFG programs.

3. This chapter compares the properties of SSA and SSI in a comprehensive and methodical fashion. Section 3.2 gives a qualitative comparison, and Section 3.6 gives a quantitative comparison. There has been no such previous study.

3.2 Introduction to SSI

SSI requires each virtual register to have a unique definition point in the program text. This is achieved by virtual register renaming, in the same way as SSA. (This dissertation follows the convention that renamed virtual registers have the same name as originally named virtual registers, with an additional integer subscript. This follows the standard presentation given in most SSA research.) However the SSI constraint generally enforces more renaming than the SSA constraint, since virtual registers used in different arms of a conditional branch must have distinct names in SSI. Thus new virtual register names are introduced at the program points given below.

assignment statements: As in SSA, a virtual register can only be assigned a value at one point in the program text. This ensures the desirable analytical property of *referential transparency*, where the value of virtual register v does not depend upon the program point at which v is used. (This is similar to the notion of flow-insensitive data flow information introduced in Chapter 4.)

control flow merge points: As in SSA, the single assignment property means that multiple reaching definitions at a control flow merge point must be factored into a single definition by a ϕ-function.

control flow split points: Unlike SSA, a virtual register that is used in one or more arms of the control flow split is assigned a new name for each arm of the split. Multiple upwardly exposed uses at a control flow split point must be factored into a single use by a σ-function.

The σ-function is the dual of the ϕ-function. Figure 3.1 compares their properties.

Figure 3.2 shows an example SSI program. This is the SSI version of the CFG program in Figure 2.2. Note that there is a σ-function for i, since the value of i is used in the loop body. There are three important features of SSI that are emphasized in this section. These properties follow naturally from the formal definition of SSI in Section 3.3, but it is helpful to build up an intuitive, informal understanding at this early stage.

1. A σ-function is required for a virtual register v used in a conditional context, even when v is not mentioned in the predicate that governs that conditional context. Virtual register y in Figure 3.3 illustrates this point.

2. If a virtual register v is defined before a control flow split point, and v is used after the corresponding control flow merge point, but not used in the conditional context, then the original name can be retained after the control flow merge point, and no ϕ- or σ-function is required. Virtual register y in Figure 3.4 illustrates this point. This property of SSI is known as single-entry-single-exit region bypassing. See Sections 3.5.1 and 5.8.2 for further details.

ϕ-function	σ-function
inserted at control flow merge points	inserted at control flow split points
single destination operand	n destination operands, where n is the number of successors to the basic block that contains this σ-function
n source operands, where n is the number of predecessors to the basic block that contains this ϕ-function.	single source operand
takes the value of one of its source operands (dependent on control flow) and assigns this value to the destination operand	takes the value of its source operand and assigns this value to one of the destination operands (dependent on control flow)

Figure 3.1: Differences between ϕ- and σ-functions

3. If a virtual register v is defined before a control flow split point, and used in the subsequent conditional context, then obviously a σ-function for v is necessary at the control flow split point. However, a σ-function must be the last use of its source operand virtual register name. So the original virtual register name cannot be mentioned after the corresponding control flow merge point. A ϕ-function is required to introduce a new virtual register name. Virtual register x in Figure 3.4 illustrates this point. In effect, σ- and ϕ- functions kill their source operands. This can be useful for some data flow analyses, as Chapter 4 explains in detail.

Ananian [Ana99] introduces SSI by giving a formal definition, a pessimistic construction algorithm and an operational semantics. This chapter presents a more concise definition and an optimistic construction algorithm. The empirical evidence presented in this chapter shows that the optimistic construction algorithm performs better than the pessimistic construction algorithm, and also that SSI is similar to SSA in terms of IR size and construction time. This evidence raises the credibility of SSI as a viable IR for static analysis.

SSI is employed in a wide range of existing analyses and optimizations. Examples are given below.

- sparse conditional constant propagation [Ana99]
- code size reductions for Java bytecode [AR03]
- static object preallocation for Java programs [GSR03]
- model-based debugging of Java programs [MS03]
- automatic synthesis for pipelined asynchronous hardware [TM04]

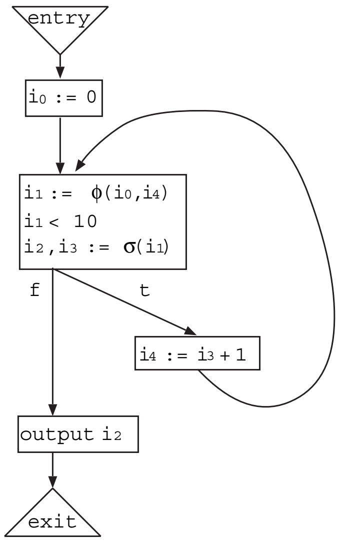


Figure 3.2: Example SSI program

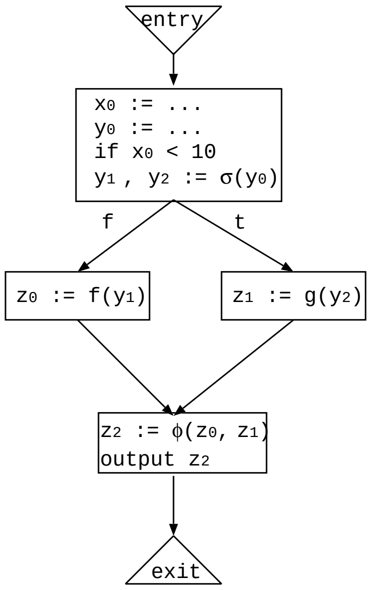


Figure 3.3: When to place σ-functions (1)

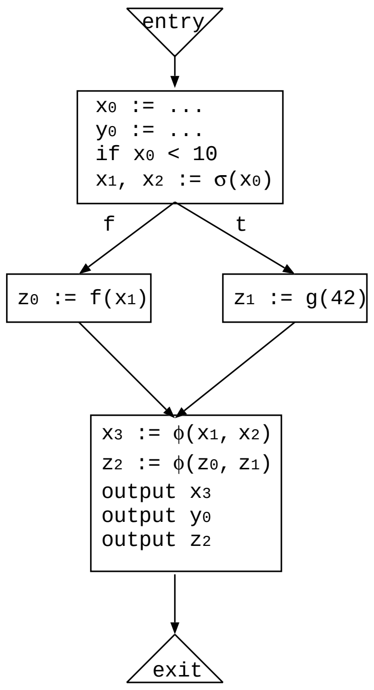


Figure 3.4: When to place σ-functions (2)

In addition, this dissertation discusses five analyses.

- constant propagation (Section 4.3)
- liveness analysis (Section 4.4)
- type inference (Section 4.5)
- slicing (Section 5.5)
- functionalization (Section 6.4.2)

The MIT Flex Java compiler [Fle98] uses SSI as its primary IR. I have added support for SSI to Machine SUIF [Smi96]. This is a low-level interface to SUIF, a flexible compiler infrastructure for imperative languages like C and Fortran [WFW$^+$94]. Most of the algorithms described in this dissertation have been implemented as passes in the Machine SUIF system. These passes are used to obtain the empirical results.

3.3 Definition of SSI

3.3.1 Preliminary Remarks

This presentation of SSI is based on the classical CFG, which was reviewed in Section 2.5.1. The present section defines other concepts that will be relevant in the rest of this dissertation.

Liveness

A virtual register v is *live* at node n if there is a control flow path from n to some other node at which v may be used, and there is no definition of v along that path. Otherwise v is *dead* at node n. Another way of stating that v is live at node n is that there is an *upwardly exposed use* of v at n. Liveness information is important in CFG and all augmented CFG IRs. Liveness information does not make sense in IRs that do not contain explicit control flow edges, such as DDG and augmented DDG IRs. Liveness is implicitly related to control flow, since the notion of liveness is only relevant when there is a total order on operations, as imposed by CFG.

Dominance

Node n_1 *dominates* node n_2 if every control flow path from n_{entry} to n_2 goes through n_1. Note that the notion of dominance can also be applied to CFG edges. This is usually for the presentation of single-entry-single-exit regions [JPP93]. Edge e_1 dominates edge e_2 if every control flow path from n_{entry} to e_2 goes through e_1. Node n_1 *postdominates* node n_2 if every control flow path from n_2 to n_{exit} goes through n_1. Node n_1 *strictly* (post)dominates node n_2 if n_1 (post)dominates n_2 and $n_1 \neq n_2$. The *dominance frontier* of node n, is the set of nodes $\text{DF}(n)$ such that for each $n_i \in \text{DF}(n)$, n strictly dominates an immediate predecessor of n_i but n does not strictly dominate n_i. The *reverse dominance frontier* of node n is the set of nodes N such that for each $n_i \in N$, n strictly postdominates an immediate successor of n_i but n does not strictly postdominate n_i. The reverse dominance frontier is sometimes known as the postdominance frontier. The *iterated dominance frontier* $\text{IDF}(n)$ of node n is the transitive closure of the dominance frontier relation on node n.

$$\text{IDF}(n) = \lim_{i\to\infty}\text{DF}^i(n)$$

where

$$\text{DF}^{i+1}(n) = \text{DF}(x \cup \text{DF}^i(n))$$

and

$$\text{DF}^0(n) = \text{DF}(n)$$

Cytron et al [CFR+91] introduce iterated dominance frontiers for their SSA construction algorithm. Muchnick [Muc97] gives a clear explanation of dominance frontiers and their computation.

Like liveness, dominance information is important in CFG and augmented CFG IRs. However, dominance can also be useful in IRs with other kinds of edges apart from control flow edges. In such cases, dominance is defined in terms of other kinds of paths (rather than control flow paths) through the graph. This dissertation only refers to CFG dominance information.

SSA adopts the convention that a ϕ-function source operand is conceptually treated as a virtual register use at the end of the basic block with which that operand is associated. (This is exactly how the ϕ-function would be translated into executable machine code.) SSI has the same convention for ϕ-functions. Similarly, SSI adopts the convention that a σ-function destination operand is conceptually treated as a virtual register definition at the beginning of the basic block with which that operand is associated.

These conventions have ramifications on the dominance properties of ϕ-function source operands, and σ-function destination operands. In particular, they simplify the SSI definition presented in the next section.

3.3.2 Actual Definition

This section presents the new and concise definition of SSI. A program is in SSI if it satisfies the following three constraints.

S1: Each virtual register has a unique definition point in the program text

S2: Each definition of a virtual register v dominates all uses of v

S3: Each use of a virtual register v postdominates the unique definition of v

The first two constraints are the same as for SSA. In fact, omitting the third constraint would make this definition identical to the SSA definition given by Kelsey [Kel95], and Cooper and Torczon [CT04]. The third constraint ensures that a virtual register must be renamed at a conditional branch if that virtual register's value is used in an arm of that branch.

3.3.3 Comparison with Ananian

Cytron et al [CFR+91] remark, "Static single assignment form may be considered as a property of a single program or as a relation between two programs."

The same distinction is true for definitions of SSI. Some definitions are *declarative*, since they state the properties that a SSI program must satisfy. Other definitions are *prescriptive*, since they specify how a non-SSI program must be altered to transform it into an SSI program.

The new definition presented in Section 3.3.2 is a declarative definition of SSI. In contrast, Ananian's original definition of SSI [Ana99] is a prescriptive definition of SSI, since it gives the conditions that must be satisfied for the correct transformation of a program from CFG to SSI. Declarative definitions avoid all such implementation detail. In a different scenario, imagine trying to prove that Shell sort is equivalent to Quicksort. This would be possible by declarative means without making use of the concept of sorting!

Ananian's Definition

Ananian's original definition is rather more verbose than our new definition. In general, prescriptive definitions are lengthier than declarative definitions. His definition does not enforce the single assignment property directly. He only mentions dominator relations as corollaries of the main definition, rather than stating the definition in terms of dominance, which seems more natural.

The transformation converts an *original* program into a *new* program. The original program is represented as CFG. Note that $\rightarrow^+$ represents a control flow path consisting of at least one edge (a nonnull path). The new program is in SSI. It is an augmented CFG, since it contains additional pseudo-definition functions and its virtual registers have been renamed. The virtual registers in the original program are referred to as the original virtual registers. The virtual registers in the new program are referred to as the new virtual registers.

So, here is Ananian's definition:

A1: If two nonnull paths $x \rightarrow^+ z$ and $y \rightarrow^+ z$ exist having only the node z where they converge in common, and nodes x and y contain either assignments to a virtual register v in the original program or a ϕ- or σ-function for v in the new program, then a ϕ-function for v has been inserted at z in the new program. (Placement of ϕ-functions)

A2: If two nonnull paths $z \rightarrow^+ x$ and $z \rightarrow^+ y$ exist having only the node z where they diverge in common, and nodes x and y contain either uses of a virtual register v in the original program or a ϕ- or σ-function for v in the new program, then a σ-function for v has been inserted at z in the new program. (Placement of σ-functions)

A3: For every node x containing a definition of a virtual register v in the new program and node y containing a use of that virtual register, there exists at least one path $x \rightarrow^+ y$ and no such path contains a definition of v other than at x. (Naming after ϕ-functions)

A4: For every pair of nodes x and y containing uses of a virtual register v defined at node z in the new program, either every path $z \rightarrow^+ x$ must contain y or every path $z \rightarrow^+ y$ must contain x. (Naming after σ-functions)

A5: For the purposes of this definition, n_{entry} is assumed to contain a definition and n_{exit} a use for every virtual register in the original program. (Boundary conditions)

A6: Along any possible control flow path in a program being executed consider any use of a virtual register v in the original program and the corresponding use v_i in the new program. Then, at every occurrence of the use on the path, v and v_i have the same value. The path need not be cycle-free. (Correctness)

Note that Ananian's definition does not prohibit arbitrary program transformations that preserve the original control- and data-flow behaviour. Such transformations include operand reordering for commutative operations. There should be another condition (**A7**) which ensures that differences between an original and a transformed program are restricted to virtual register renaming and pseudo-definition insertion only. This additional constraint is assumed throughout the chapter.

The rest of this section attempts to argue carefully that there is an equivalence between our declarative definition of SSI (referred to as S) and Ananian's prescriptive definition of SSI (referred to as A). The equivalence holds if it can be shown that:

- A implies S, and
- S implies A.

Whether declarative and prescriptive definitions can be proved equivalent in the general case is an open question.

A Implies S

[**A1 and A3 and A6 (almost) imply S1**] A3 states that every definition of virtual register x must reach every use of virtual register x. But there is a contradiction if more than one definition of x reaches a use of x, since A1 states that there would be a ϕ-function for x at the program point where the multiple reaching definitions converge. A3 ensures that such a ϕ-function destination operand would have a fresh name, and A6 ensures that the use site would have been renamed to share this fresh name. Hence this situation would not have arisen in the first place! This means that every definition of a virtual register that is subsequently used defines a fresh virtual register name.

Now, A5 ensures that every virtual register from the original program is used at least once. However, it makes no guarantees about virtual registers introduced by pseudo-definition functions. A3 only holds in the case when ϕ- and σ-function destination operands are subsequently used. If a pseudo-definition function defines a dead virtual register, then A makes no guarantees that this dead register will not be defined by another pseudo-definition function. S is stronger, since S1 provides the static single assignment property for dead virtual registers as well as live ones. The next two implications ignore this defect, and assume that all defined virtual registers are used at least once.

[**A1 and A3 and A5 imply S2**] A3 states that there is at least one control flow path from program point x that defines virtual register v to program point y that uses v, and that there are no definitions of v along any path from x to y. Now A5 states that every original virtual register is defined at n_{entry}. So if there is a control flow path from n_{entry} to y that does not contain x, then A1 guarantees that there must be a ϕ-function for v at the point where the control flow path from n_{entry} merges with the control flow path from x. But this contradicts A3, which states that there are no definitions of v along the path from x to y. Therefore all paths from n_{entry} to y must contain x. This means that x dominates y, which is precisely the condition S2.

[**A2 and A3 and A4 and A5 imply S3**] Consider a definition of virtual register v at node z. A2 and A5 ensure that there is always at least one use of v, at n_{exit}. This is because n_{exit} is reachable from z, and A2 guarantees there must be a ϕ-function for v at n_{exit}, if not earlier. This trivially implies S3, since n_{exit} postdominates all nodes. Consider the case where there are at least two uses of v. Now, from A4, there is an ordering on these uses. So there will be a distinct 'last use' of v, say x. A4 makes it clear that every path from z to x contains all the other uses of v. All paths from z to n_{exit} must go through x. All paths from the uses of v to n_{exit} must go through x. If there were a path from z to n_{exit} that did not contain x, then since n_{exit} has a use of every original virtual register, A2 guarantees that there must be a σ-function for v at the control flow split point, where one path goes through x and the other path goes to n_{exit} without passing through x. But this would be a contradiction, since a σ-function for v be a definition of v along the path from z to x, which contradicts A3. Therefore all paths from z to n_{exit} pass through x. Therefore all paths from z to n_{exit} pass through all uses of v. Therefore all uses of v postdominate z. This is precisely the condition of S3.

S Implies A

Conditions A3 and A4 encapsulate the process of renaming virtual registers in the new program. These conditions do not mention the original program at all. So it should be possible to prove an equivalence between these prescriptive conditions and the declarative conditions. However, the other prescriptive conditions all refer to the original program, and it is not apparent how to handle this in the declarative definition. There is no analogue to A1, A2, A5 and A6.

[**S1 and S2 imply A3**] S1 states that virtual register v has a unique definition point, say x. S2 states that every path from n_{entry} to a use of v, say y, goes through x. Because of the reachability property of CFG nodes, there must be at least one path from n_{entry} to y. Therefore, S1 and S2 imply A3, which states that there is at least one non-null path from x to y, and no path from x to y contains another definition of v.

[**S1 and S3 imply A4**] S1 states that virtual register v has a unique definition point, say z. S3 states that every path from z to n_{exit} goes through all uses of v. When there are less than two uses of v, the implication is trivially satisfied. Consider the case when there are at least two uses. Without loss of generality, select any two distinct uses, say x and y. Note that the reachability property of CFG nodes ensures that there must be at least one path from x to n_{exit}. Now, because of the properties of postdominators, there is a total ordering on all the nodes that postdominate z. So either x postdominates y, or y postdominates x. Without loss of generality, consider the case when x postdominates y. This means that every control flow path from z to x contains y. This is precisely the condition for A4.

Discussion

It appears from the reasoning above that the two definitions are not entirely equivalent. A refers to the original program, which has no equivalent in S. Also, A provides no guarantees about the static single assignment property for dead virtual registers in pseudo-definition functions, as mentioned above.

Hence, the new definition S seems more intuitive and more useful than the old definition A.

3.4 Properties of SSI

This section only discusses a few intrinsic features of SSI, relating to the number of virtual registers and pseudo-definition functions. Section 3.5 compares the properties of two SSI construction algorithms. Section 3.6 compares SSI with other similar IRs.

3.4.1 Bounding the Blow-Up

Consider an arbitrary CFG program p with N nodes. Then, in the worst case, p has $O(N^2)$ edges. Each node in p is the source node for $O(N)$ edges. Generally, the number of virtual registers in p grows as $O(N)$. The number of virtual register definitions in p also grows as $O(N)$. Now consider the transformation from CFG program p into SSI program p'. In the worst case, there will be a ϕ-function and a σ-function for each of the $O(N)$ original virtual registers at each of the $O(N)$ nodes. This means that p' contains $O(N^2)$ pseudo-definition functions.

The most interesting property is the number of virtual registers defined in p'. There are $O(N)$ virtual register definitions in p, which all must be present in p'. p' also has

$O(N^2)$ ϕ-functions that each define 1 virtual register. p' also has $O(N^2)$ σ-functions that each define $O(N)$ virtual registers, since there may be $O(N)$ edges from each CFG node. Therefore the number of virtual registers in p' grows as $O(N^3)$. In contrast, if p'' is the SSA transformation of p, then the number of virtual registers in p'' grows as $O(N^2)$ [CFR+91].

However Cytron et al show that the SSA worst case is never observed in practice. They give comprehensive empirical evidence to suggest that the SSA translation is linear. The same property appears to hold for SSI [Ana99].

3.4.2 Pruning SSI

Both SSI definitions allow for extra pseudo-definition functions to be inserted, that are not strictly necessary. This is the same as for SSA. *Minimal* and *pruned* SSI variants parallel their SSA counterparts. They are clearly defined by Ananian [Ana99]. Minimal SSI contains the minimum number of ϕ- and σ-functions such that the conditions for SSI are satisfied. Pruned SSI is the minimal form with any dead ϕ- and σ-functions removed. Recall that a statement is *dead* if its destination operands are never subsequently used. Liveness information is required for the construction of pruned SSI.

Cooper and Torczon [CT04] discuss *maximal* and *semi-pruned* forms for SSA. There are SSI parallels for these variants too. Maximal SSI inserts, for each virtual register mentioned in the original program, a ϕ-function at each node with multiple predecessors and a σ-function for each node with multiple successors. This is the 'really crude' approach described by Appel [App98b] in the context of SSA construction. Semi-pruned SSI is based on the observation that many virtual registers are local to one CFG node. This is especially true of compiler generated temporary virtual registers. Such virtual registers will never require ϕ- or σ-functions, since they are never live across node boundaries. Because of this, they need not be considered when placing pseudo-definition functions. Semi-pruned SSI avoids many of the redundant pseudo-definitions present in minimal SSI without requiring the expensive data flow analysis necessary for pruned SSI. The virtual register locality information for the semi-pruned construction algorithm must be precomputed. The data flow analysis for locality is straightforward. It is far more efficient to compute than liveness information. Basically, a virtual register v is non-local if there is a use of v before a definition of v in any CFG node. This can be stated formally as: a non-local virtual register v has an upwardly exposed use at the beginning of node n in which there is an instruction that uses v. Semi-pruned form was introduced for SSA by Briggs et al [BCHS98]. Unless stated otherwise, all experiments reported by this dissertation construct and operate on semi-pruned SSI.

In terms of the number of inserted pseudo-definition functions for a program:

$$\text{maximal} \geq \text{minimal} \geq \text{semi-pruned} \geq \text{pruned}$$

3.5 Constructing SSI

This section assumes that the SSI construction algorithm takes an arbitrary CFG program as input, and transforms it into a valid SSI program with the same behaviour. The definition of 'same behaviour' is the same as condition A6 in Ananian's SSI definition above.

Construction of SSI takes place in two phases, in the same manner as for SSA.

1. Identify points where pseudo-definitions (ϕ- and σ-functions) must be inserted.

2. Rename virtual registers in order to respect the SSI property and to retain the original program semantics.

The placement algorithm must be applied first, followed by the renaming algorithm. Ananian presents a simple and efficient renaming algorithm in his thesis [Ana99]. Similarly, the original SSA renaming algorithm [CFR+91] can easily be modified to handle SSI instead. Basically, the renaming algorithm processes nodes in an order given by the dominance relation. For each virtual register from the original program, there is a stack of renamed virtual registers for the new program. As the algorithm processes a virtual register definition, it pushes a new name onto the appropriate stack and renames the defined virtual register accordingly. At each virtual register use, the original virtual register is renamed to the new name on top of the appropriate stack. The rest of this section focuses entirely on the placement algorithm, rather than the renaming algorithm.

There are two distinct styles of SSI construction—optimistic and pessimistic. Indeed, any data flow analysis algorithm can be presented as an optimistic or a pessimistic algorithm [Cli93]. An optimistic algorithm initially assumes the best case. For SSI construction, this means that no pseudo-definitions are required. Then it refines this (possibly incorrect) best case by iterating to a fixpoint that satisfies the data flow property. This is the least fixpoint, if $\perp$ is the best case. A pessimistic algorithm initially assumes the worst case. For SSI construction, this means that pseudo-definitions are required for all virtual registers at all control flow merge and split points. Then it refines this worst case by iterating to a fixpoint that satisfies the data flow property. This is the greatest fixpoint, if $\top$ is the worst case. A pessimistic algorithm can stop before reaching a fixpoint, and still give a valid (though far from optimal) solution. On the other hand, no intermediate states are correct in the optimistic algorithm before the fixpoint is reached.

Note that every SSI construction algorithm also constructs SSA as a byproduct. To recover the SSA, simply elide σ-functions and rename σ-function destination operands to have same name as corresponding σ-function source operands. This is a potentially expensive method of SSA construction. It performs lots of extra calculation and then throws most of the result away. Also, SSI often has more ϕ-functions than SSA, so some ϕ-functions may be redundant after the SSI to SSA conversion step. The DFG inventors advocate this style of SSA construction, but they present no indication of relative performance [JP93].

3.5.1 Pessimistic Construction

Ananian describes his algorithm for calculating SSI in great detail [Ana99]. He states:

> Our algorithm for placing ϕ- and σ-functions in SSI form is *pessimistic*; that is, we at first assume every node in the control flow graph with input arity larger than one requires a ϕ-function for every virtual register and every node with out-arity larger than one requires a σ-function for every virtual

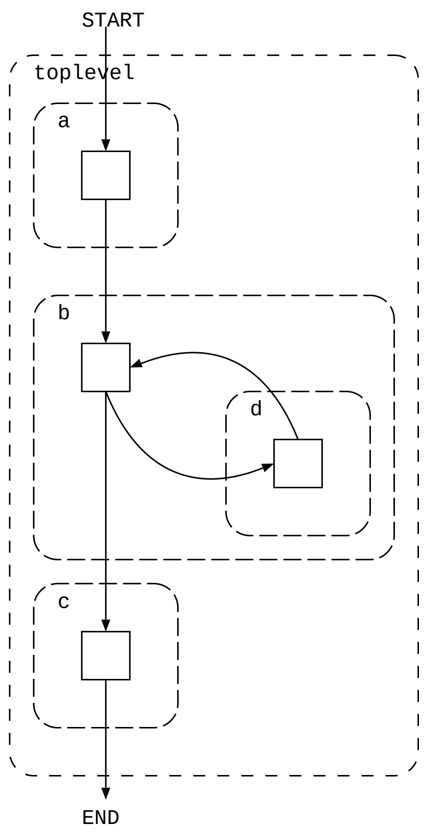


Figure 3.5: A CFG with marked SESE regions (toplevel, a, b, c, d)

register, and then use the program structure tree, liveness information, and unused code elimination to determine safe places to *omit* ϕ- or σ-functions.

Ananian's construction algorithm begins with a program structure tree of single-entry single-exit (SESE) regions, as described by Johnson et al [JPP93, JPP94]. A SESE region in a graph G is an ordered pair (x, y) of distinct control flow edges x and y where:

1. x dominates y, and
2. y postdominates x, and
3. every cycle containing x also contains y and vice-versa.

A program structure tree records the nesting structure of SESE regions in a CFG. Each node in this tree represents a SESE region. The parent of a node is the closest containing region and the children of a node are all the regions immediately contained within it. Figure 3.5 shows a CFG with the SESE regions marked in dashed lines. Figure 3.6 shows the program structure tree for the same CFG.

Ananian's algorithm is presented in Figure 3.7. It performs a post-order traversal of the program structure tree for each virtual register v. That is to say, it visits nested child SESE regions before visiting a parent region. It determines which regions require ϕ- or σ-functions for virtual register v. PlacePhis(r, v) inserts a ϕ-function for virtual register v at each control flow merge point in region r if r contains a definition of virtual register

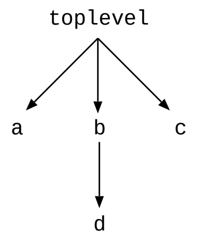


Figure 3.6: A program structure tree of SESE regions

v or if a ϕ-function has already been placed in a child of this region. PlaceSigmas(r, v) inserts a σ-function for virtual register v at each control flow split point in region r if r contains a use of virtual register v (including uses due to previously inserted ϕ-functions) or if a σ-function has already been placed in a child of this region.

MaybeLive(v, n) should return `true` when v may possibly be live at node n. It should give a conservative approximation to liveness, so in the simplest case MaybeLive can be programmed always to return `true`. This causes the placement algorithm to produce minimal SSI. A more pruned SSI may be obtained by a more sophisticated implementation of the MaybeLive function.

Ananian [Ana99] states that his SSI construction algorithm has worst-case linear time complexity with respect to the size of the CFG. Constructing the program structure tree for a CFG also takes linear time with respect to CFG size [JPP93, JPP94]. The placement algorithm in Figure 3.7 makes a single pass through the program structure tree, thus it too is linear.

3.5.2 Optimistic Construction

The new SSI construction method is an optimistic approach to the problem, in contrast to Ananian's pessimistic approach. It initially assumes that no ϕ- or σ-functions are needed, and then analyses the CFG to determine whether any functions should be inserted. It applies a ϕ-function placement pass, followed by a σ-function placement pass, and then iterates to a fixpoint. The algorithm is presented in Figure 3.8. Figures 3.12 and 3.13 give more details.

The iteration is necessary because the insertion of σ-functions may cause extra ϕ-functions to be required, and vice versa. For example, consider the CFG program in Figure 3.9.

As it stands, no σ-functions are required since there are no virtual register uses in either arm of the conditional statement. However, a ϕ-function is required for virtual registers x and z at the control flow merge point after the conditional statement, due to the assignments in the arms of the conditional branch. Figure 3.10 shows the program in this state.

Now each inserted ϕ-function counts as a use of its subject virtual register at the end of both basic blocks that precede that ϕ-function. There is a definition of z in each arm of the conditional branch, however, there is only a definition of x in one arm of the conditional branch. Hence there is an upwardly exposed use of x from the ϕ-function to the corresponding control flow split point. Therefore a σ-function is required for x at

```
Place($G$: CFG) =
        let $r$ be the top-level region for $G$
        for each virtual register $v$ in $G$
            PlacePhis($r$, $v$)
            PlaceSigmas($r$, $v$)


PlacePhis($r$: region, $v$: virtual register): boolean =
        /* post-order traversal */
        $flag \longleftarrow$ false
        for each child region $r'$
            if PlacePhis($r'$, $v$)
                $flag \longleftarrow$ true

        for each node $n$ in region $r$ not contained in a child region
            if $n$ contains a definition of $v$
                $flag \longleftarrow$ true

        /* add $\phi$-functions to merges where $v$ may be live */
        if $flag$ = true
            for each node $n$ in region $r$ not contained in a child region
                if MaybeLive($v$, $n$) = true
                    if the input arity of $n$ exceeds 1
                        place a $\phi$-function for $v$ at $n$

        return $flag$


PlaceSigmas($r$: region, $v$: virtual register): boolean =
        /* post-order traversal */
        $flag \longleftarrow$ false
        for each child region $r'$
            if PlaceSigmas($r'$, $v$)
                $flag \longleftarrow$ true

        for each node $n$ in region $r$ not contained in a child region
            if $n$ contains a use of $v$
                $flag \longleftarrow$ true

        /* add $\sigma$-functions to splits where $v$ may be live */
        if $flag$ = true
            for each node $n$ in region $r$ not contained in a child region
                if MaybeLive($v$, $n$) = true
                    if the output arity of $n$ exceeds 1
                        place a $\sigma$-function for $v$ at $n$

        return $flag$
```

Figure 3.7: Ananian's pessimistic SSI placement algorithm

```
/* initialize arrays */
for each virtual register $v$
    $defsites[v] \longleftarrow \{\}$
    $usesites[v] \longleftarrow \{\}$
    $A_\phi \longleftarrow \{\}$
    $A_\sigma \longleftarrow \{\}$
for each node $n$
    for each virtual register $v \in A_{orig}[n]$
        $defsites[v] \longleftarrow defsites[v] \cup \{n\}$
    for each virtual register $v \in U_{orig}[n]$
        $usesites[v] \longleftarrow usesites[v] \cup \{n\}$
/* perform fixpoint computation */
$change \longleftarrow$ true
while ($change$)
    $change \longleftarrow$ false
    place-$\phi$-functions()
    place-$\sigma$-functions()
```

Figure 3.8: Optimistic SSI placement algorithm

```
$x \longleftarrow$ input()
if (...)
    then $x \longleftarrow 2$
        $z \longleftarrow 0$
    else $z \longleftarrow 2$
$y \longleftarrow x + 1$
```

Figure 3.9: Example CFG program that requires iteration

```
$x \longleftarrow$ input()
if (...)
    then $x \longleftarrow 2$
        $z \longleftarrow 0$
    else $z \longleftarrow 2$
$x \longleftarrow \phi(x, x)$
$z \longleftarrow \phi(z, z)$
$y \longleftarrow x + 1$
```

Figure 3.10: Example program after ϕ-function insertion

```
$x \longleftarrow$ input()
**if** $(\ldots)$
$\langle x, x\rangle \longleftarrow \sigma(x)$
    **then** $x \longleftarrow 2$
         $z \longleftarrow 0$
    **else** $z \longleftarrow 2$
$x \longleftarrow \phi(x, x)$
$z \longleftarrow \phi(z, z)$
$y \longleftarrow x + 1$
```

Figure 3.11: Example program after σ-function insertion

this control flow split point. Figure 3.11 shows the program in this state. Note that a σ-function for z is required as well, to satisfy the requirements of minimal SSI, though not pruned SSI.

The maximum number of iterations required to reach a fixpoint is related to the maximum depth of nested conditional statements and loops in the CFG. Empirical studies have shown this nesting depth to be quite shallow in human-generated code [Knu71].

The new construction algorithm follows the standard method that uses dominance frontiers to discover where ϕ- and σ-functions are needed. Thus our place-ϕ-functions algorithm is identical to the standard SSA ϕ-function placement algorithm [CFR+91]. The place-σ-functions algorithm is the dual of the place-ϕ-functions algorithm. It has the same shape but it is exactly the opposite. It tracks virtual register uses rather than definitions, it uses reverse dominance frontiers rather than standard dominance frontiers, it inserts σ-functions at the end of basic blocks rather than ϕ-functions at the beginning of basic blocks.

We adopt the notational conventions of Appel [App98a] for presenting the ϕ- and σ-function placement algorithms, in Figures 3.12 and 3.13 respectively.

$A_{orig}[n]$ contains the set of virtual registers that are assigned a value at node n, assumed to be precomputed. $\mathit{defsites}[a]$ is initialized to contain the set of nodes that assign a value to virtual register a. W is a work-list of nodes that need to be processed. $DF[n]$ is the set of nodes comprising the dominance frontier of node n. See [CFR+91, App98a] for more details. $A_\phi[a]$ is the set of nodes that contain a ϕ-function for virtual register a.

$U_{orig}[n]$ contains the set of virtual registers that are used at node n, assumed to be precomputed. $\mathit{usesites}[a]$ is initialized to contain the set of nodes that use the value of virtual register a. $RDF[n]$ is the set of nodes in the reverse dominance frontier of node n. See [CFR+91, App98a] for more details. $A_\sigma[a]$ is the set of nodes that contain a σ-function for virtual register a.

Cytron et al claim that their SSA ϕ-function placement algorithm has $O(R)$ time complexity 'in practice' where R is an abstract measure of CFG size. Nevertheless their algorithm has worst case $O(R^3)$ time complexity. There are genuine linear time ϕ-function placement algorithms [BP99, SG95]. Each iteration of the optimistic SSI construction algorithm has the same complexity as the SSA construction algorithm—potentially $O(R^3)$, practically $O(R)$. The number of iterations necessary to reach the fixpoint is related to

```
place-$\phi$-functions() =
        **for each** virtual register $v$
            $W \longleftarrow defsites[v]$
            /* worklist algorithm */
            **while** $W$ not empty
                remove some node $n$ from $W$
                **for each** $y \in DF[n]$
                    **if** $y \notin A_\phi[v]$
                        insert statement $v \longleftarrow \phi(v, v, ..., v)$ at
                            start of node $y$, where $\phi$-function has
                            as many arguments as $y$ has predecessors
                        $A_\phi[v] \longleftarrow A_\phi[v] \cup \{y\}$
                        **for each** $p \in predecessors(y)$
                            $usesites[v] \longleftarrow usesites[v] \cup \{p\}$
                        $change \longleftarrow$ true
                        **if** $y \notin defsites[v]$
                            $defsites[v] \longleftarrow defsites[v] \cup \{y\}$
                            $W \longleftarrow W \cup \{y\}$
```

Figure 3.12: ϕ-function placement algorithm

```
place-$\sigma$-functions() =
        **for each** virtual register $v$
            $W \longleftarrow usesites[v]$
            /* worklist algorithm */
            **while** $W$ not empty
                remove some node $n$ from $W$
                **for each** $y \in RDF[n]$
                    **if** $y \notin A_\sigma[v]$
                        insert statement $v, v, ..., v \longleftarrow \sigma(v)$
                            at end of node $y$, where
                            $\sigma$-function has as many results
                            as $y$ has successors
                        $A_\sigma[v] \longleftarrow A_\sigma[v] \cup \{y\}$
                        **for each** $s \in successors(y)$
                            $defsites[v] \longleftarrow defsites[v] \cup \{s\}$
                        $change \longleftarrow$ true
                        **if** $y \notin usesites[v]$
                            $usesites[v] \longleftarrow usesites[v] \cup \{y\}$
                            $W \longleftarrow W \cup \{y\}$
```

Figure 3.13: σ-function placement algorithm

the maximum nesting depth of conditional statements and loops. In the worst case this grows as $O(R)$, although Knuth [Knu71] observes that the nesting depth is $O(1)$ in most programs. Thus, the SSI construction algorithm has worst case $O(R^4)$ time complexity, but $O(R)$ time complexity in practice. The worst case rarely seems to occur, as the results below indicate.

3.5.3 Empirical Comparison

I implemented both of the above placement algorithms in C++ using Machine SUIF. It is relatively easy to write Machine SUIF passes that operate at the CFG level. I implemented Ananian's algorithm from scratch in just under 4000 lines of code. The new optimistic placement algorithm is based on the existing Machine SUIF SSA pass [Hol01], which was modified and extended to construct SSI. The SSA pass is 5000 lines of code. The SSI pass is 7000 lines of code.

I tested the two algorithms against a selection of programs from the SPEC CINT 2000 benchmark suite [Spe00a, Hen00]. These C programs are compiled into Machine SUIF CFG files, (one CFG for each procedure). Then I ran the placement algorithms on the CFGs. The *pessimistic* algorithm is Ananian's method, as presented in Section 3.5.1. The *optimistic* algorithm is the new method, as presented in section 3.5.2. Note that only non-address-taken local variables and compiler-generated temporaries are transformed into SSI virtual registers. All other variables retain their original names. Chapter 7 briefly discusses the application of SSA renaming to address-taken variables.

I measured the time taken to transform all the CFG files for each benchmark program. Each time recorded is the median value of five tests, on a lightly loaded AMD Athlon 1.4GHz x86 Linux machine. Figure 3.14 presents these results. I also measured the number of ϕ- and σ-functions inserted by each approach, summed over all the procedures in each benchmark program. Figure 3.15 presents these results.

There are several interesting observations from these results.

1. The optimistic approach is often much faster than the pessimistic one. It is never significantly slower. In the best cases, the optimistic approach is twice as fast. The programs in the SPEC CINT 2000 benchmark suite are representative real-world programs that any respectable optimizing compiler should be expected to handle easily. It is noted that no analysed procedure requires more than 5 iterations of the optimistic algorithm placement loop. The average number of iterations is less than 3.

2. The optimistic approach places far fewer ϕ- and σ-functions than the pessimistic one. In the best cases, the optimistic approach places less than half as many pseudo-definitions as the pessimistic approach. The pessimistic algorithm places more pseudo-definitions by its nature, since it computes the greatest fixpoint. This is the main contribution to its sluggish performance. Ananian [Ana99] suggests that a dead code elimination pass should occur after the placement algorithm. This may reduce the number of pseudo-definitions, but it would take even longer to complete the SSI construction.

3. There are approximately twice as many σ-functions as ϕ-functions in each case. This is independent of the placement algorithm used. This is because σ-functions

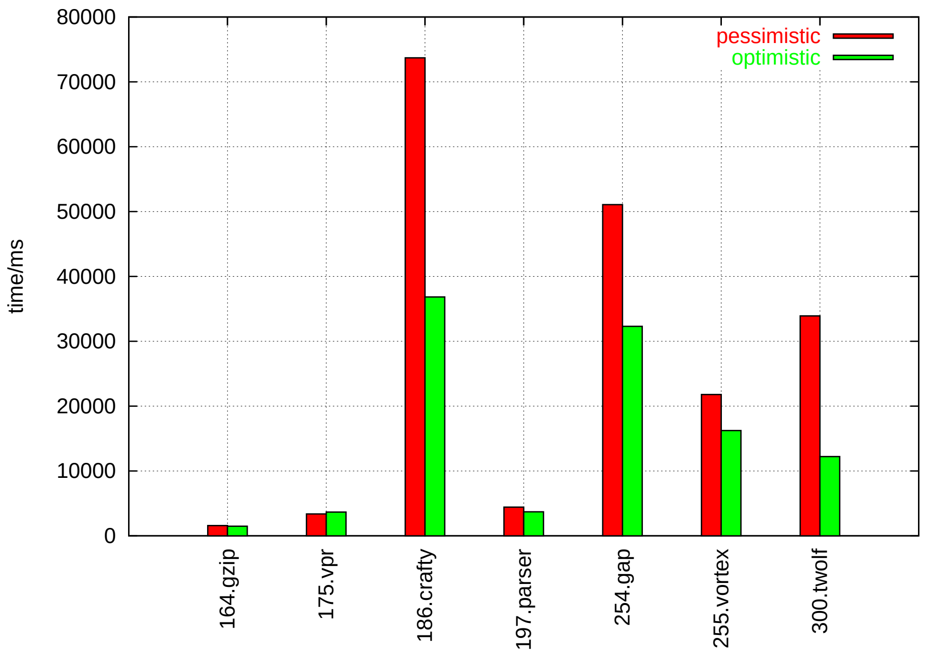


Figure 3.14: Comparing time taken for the two placement algorithms

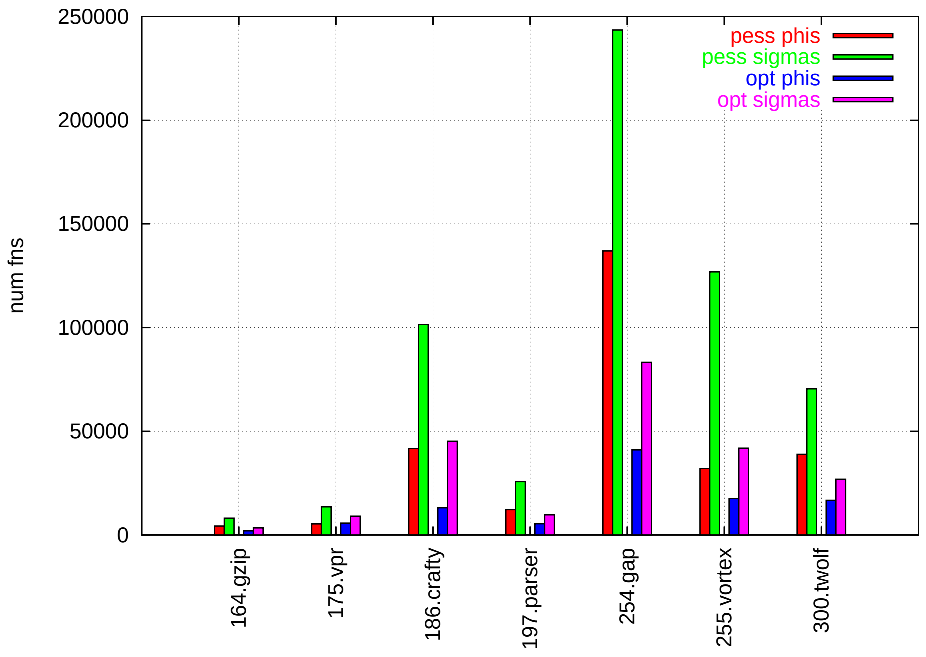


Figure 3.15: Comparing number of inserted pseudo-definition functions for the two placement algorithms

are normally inserted at two-way control flow split points, whereas ϕ-functions are often inserted at n-way control flow merge points ($n > 2$). This is due to the well-structured nature of the benchmark code.

3.6 Related Work

This section reviews several other IRs that have similar properties to SSI. Section 3.6.1 briefly discusses a number of SSA extensions and compares these with SSI. None of these IRs is particularly well known or well documented. Section 3.6.2 provides a detailed comparison of SSA and SSI. It particularly focuses on the algorithms used to construct SSA and SSI. SSA is the only sparse IR that has provoked widespread interest in construction algorithms. This section highlights several popular construction algorithms for SSA. There is a similar relationship between the performance of optimistic and pessimistic algorithms for SSA as for SSI.

3.6.1 Similar IRs

Plevyak [Ple96] describes static single use form (SSU) as a variant of SSA. SSU inserts ϕ-functions at control flow merge points, in the same way as SSA. SSU inserts ψ-functions at control flow split points, in the same way as σ-functions are inserted in SSI. SSU renames virtual registers that are read along different control flow paths. It seems that Plevyak's formulation of SSU is equivalent to SSI, although Plevyak gives neither a formal definition of SSU nor an algorithm for its construction. Plevyak applies SSU to data flow analysis of object-oriented programs.

Predicated SSA (PSSA) [CSC+99] enables effective instruction scheduling for predicated superscalar architectures such as IA-64. PSSA distinguishes between virtual register uses in different conditional contexts by means of full-path predicates (the set of conditions that must hold in order for each basic block to execute). PSSA could be transformed to an IR that resembles SSI if the full-path predicates were incorporated into the naming scheme for virtual registers. PSSA encodes σ-functions by conditional instructions that generate the appropriate values for the full-path predicates. A formal algorithmic description of PSSA construction is not given. An informal description appears to operate in a syntax-directed manner over a graph consisting of hyperblocks of simple predicated abstract machine instructions.

Bodik et al [BGS00] define extended SSA (e-SSA). e-SSA renames virtual registers at the following program points:

- at assignment statements (as in SSA), and
- at control flow merge points (using ϕ-functions as in SSA), and
- at conditional branches (as in SSI, only the pseudo-definition is known as a π-function rather than a σ-function), and
- at array-bounds checks (again using π-functions).

The main difference is that e-SSA only inserts π-functions for the virtual registers mentioned in the conditional branch predicate. In contrast, SSI inserts σ-functions for all virtual registers used in the either arm of the conditional branch. So SSI generally has more pseudo-definitions than e-SSA. e-SSA is used for array-bounds check elimination. The analysis is powerful enough to eliminate many array-bounds checks from Java programs, and efficient enough to be deployed in a state-of-the-art optimizing JIT compiler [AAB+00]. e-SSA is constructed by first inserting π-functions at conditional branches and at array-bounds checks in a syntax-directed manner, then computing SSA using the standard algorithm [CFR+91]. Again, no formal algorithmic description is available.

The dependence flow graph (DFG) [JP93] is very similar to SSI. DFG `merge` nodes correspond to SSA ϕ-nodes (as noted in [JP93]) and DFG `switch` nodes correspond to SSI σ-functions. However, DFG only inserts `switch` nodes for virtual registers that are defined in a conditional context, whereas SSI inserts σ-functions for virtual registers that are mentioned in a conditional context. DFG is constructed using a pessimistic algorithm based on SESE regions [JPP93]. Ananian's pessimistic SSI construction algorithm is an adaptation of the DFG algorithm.

3.6.2 SSA and Construction Algorithms

From a qualitative standpoint, the major difference between SSA and SSI is that SSI enforces more virtual register renaming. In general SSA programs contain fewer pseudo-definition functions, since SSA does not rename virtual registers at control flow split points. This section assesses the impact on SSI construction techniques of the additional pseudo-definitions, by comparing construction algorithms for both SSA and SSI. (As in the majority of this chapter, note that *construction* refers to placement of pseudo-definition functions.)

Optimistic Algorithms

There is a wealth of research concerning SSA construction algorithms. Bilardi and Pingali [BP03] give an overview of *optimistic* SSA construction, with a formal framework for classifying such algorithms. The most popular SSA construction algorithm [CFR+91] is optimistic. It uses dominance frontiers to determine where to place ϕ-functions. Cytron et al claim that this algorithm is 'linear in practice' in the size of the untransformed CFG. They justify this claim with a comprehensive empirical study. However they note that, theoretically, their construction algorithm has a worst case $O(R^3)$ time complexity, where R is an abstract measure of the size of the untransformed CFG. There are other optimistic SSA ϕ-function placement algorithms which guarantee actual linear time complexity [BP99, SG95].

Figure 3.16 shows the time taken for SSA and SSI construction algorithms. The SSA algorithm is an implementation of the work described by Cytron et al [CFR+91]. The SSI algorithm is an implementation of the algorithm described in Section 3.5.2. On average, SSI construction takes 3.7 times longer than SSA construction. 186.crafty is an anomalous result. This is due to the convoluted control flow constructs present in the source code of the chess-playing decision engine. It is common for game-playing programs to have a larger number of potential control flow paths than average programs in a benchmark suite [BL00, SIM+05].

Figure 3.17 shows the number of pseudo-definition functions inserted by these SSA and SSI construction algorithms. The ratios vary with each routine, but on average, the

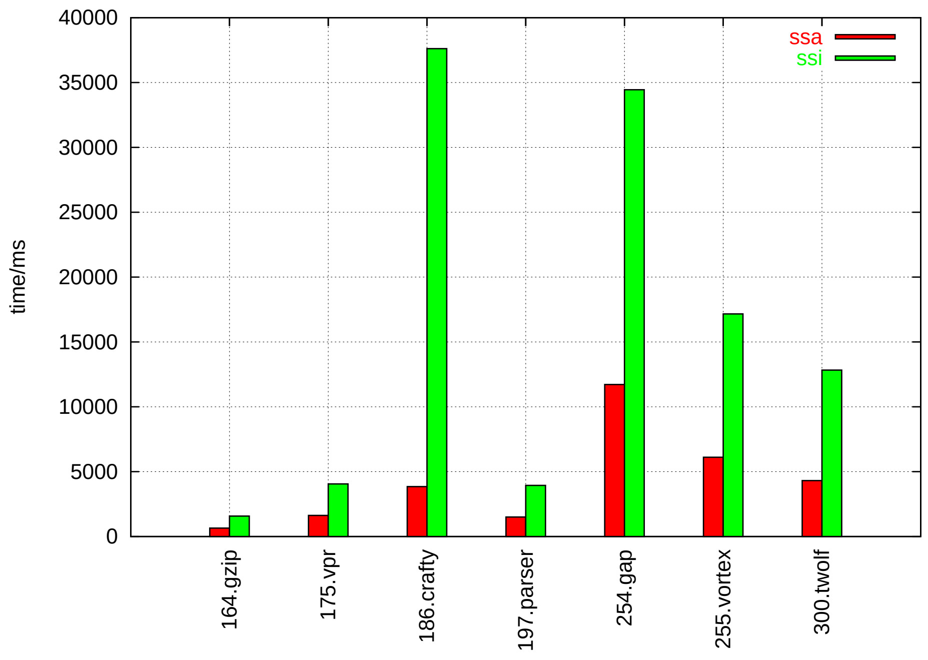


Figure 3.16: Comparing time taken by SSA and SSI construction algorithms

SSI algorithm inserts 6.3 times more pseudo-definition functions than the SSA algorithm.

Pessimistic Algorithms

In contrast to optimistic construction techniques, Aycock and Horspool [AH00] present a *pessimistic* SSA construction algorithm. It assumes that ϕ-functions are needed everywhere and then it removes ϕ-functions that are shown to be redundant. Results show that this pessimistic algorithm is generally two times slower than the standard optimistic algorithm [CFR$^+$91]. The pessimistic algorithm may insert extra ϕ-functions, but actual figures are not recorded in the paper. They argue that their pessimistic algorithm is simpler to implement, since it does not require heavy-weight data flow information such as dominance frontiers. Another point is that SSA construction comprises a very small fraction of the total compilation time. However, to the best of my knowledge, all real-world SSA-based compilers use an optimistic construction algorithm.

The empirical results from Section 3.5.3 show that the pessimistic SSI construction algorithm is on average 1.6 times slower than the optimistic algorithm. These observations are similar to the results of Aycock and Horspool for SSA construction [AH00].

3.7 Future Work

This section outlines three possible alternative approaches for constructing SSI. At the moment, these are only speculative ideas, with little investigation and even less implementation.

1. It may be possible to specify SSA or SSI properties declaratively using the notation of path logic programming [DdS02, SdL04], or some similar formalism. In this way,

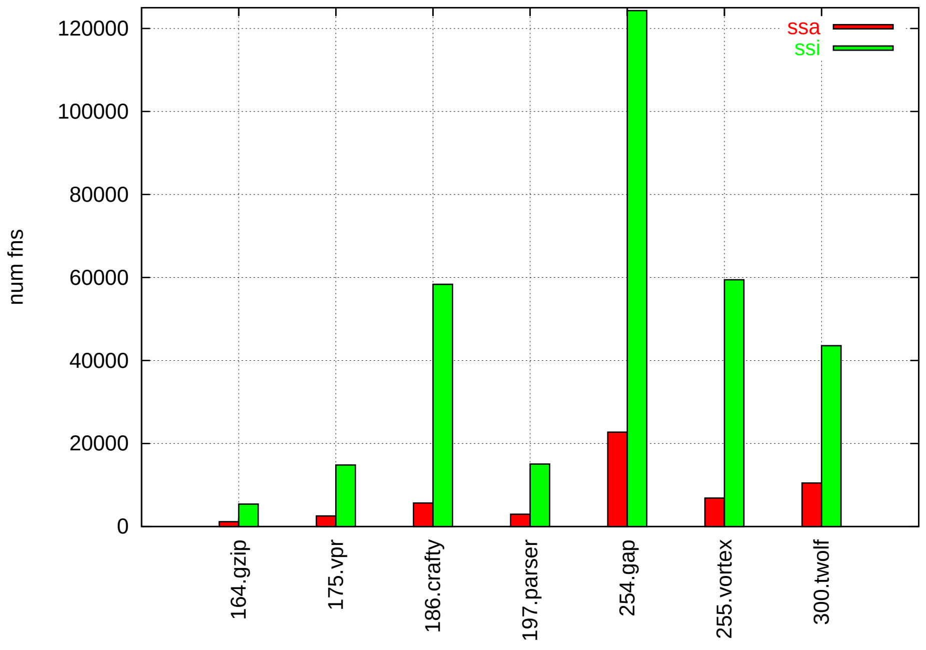


Figure 3.17: Comparing number of pseudo-definition functions inserted by SSA and SSI construction algorithms

the IR specification would be directly executable. Chapter 7 discusses this concept in more detail.

2. Brandis and Mössenbök [BM94] present a syntax-directed SSA construction algorithm for well-structured programs written in high-level programming languages. It should be straightforward to devise a similar syntax-directed SSI construction algorithm. I have commenced the implementation of this approach using the Stratego term rewriting system [Vis01].

3. In the context of procedure unfolding, it should be more efficient to transform small procedures into SSI before inlining them, rather than inlining procedures prior to SSI transformation. This is because the optimistic SSI construction algorithm has superlinear time complexity. This issue becomes relevant when we consider deproceduralization in Chapter 6. Johnson et al [JPP94] advocate such a divide-and-conquer approach for data flow analysis. They use SESE regions in the program structure tree for a single procedure. The same approach may be beneficial for procedures in a call graph that is to be deproceduralized.

3.8 Concluding Remarks

This chapter has comprehensively reviewed SSI. It has proposed a new definition, which is the first declarative definition of SSI. It has presented the first optimistic SSI construction algorithm. Results clearly demonstrate the superiority of this optimistic algorithm over the existing pessimistic algorithm, as is the case for SSA construction. Other SSI-like

IRs have been briefly mentioned. Other approaches to SSI construction have been briefly noted.

There are two main points to take from this chapter.

1. Optimistic construction algorithms for augmented CFG IRs are definitely better than pessimistic algorithms. Whether this result holds true for more general data flow analyses is still an open question.

2. SSI is more expensive to construct than SSA. Whether this extra expense can be justified is the subject of the next chapter.

Chapter 4

Data Flow Analysis

SSI enables the implementation of accurate and efficient data flow analysis algorithms. The performance of these SSI-based algorithms is superior to equivalent algorithms operating on SSA or CFG IRs.

4.1 About this Chapter

4.1.1 Objectives

Whereas Chapter 3 introduced SSI and showed how to construct it, this chapter demonstrates that SSI may be used effectively as an IR for practical static analysis. This chapter concentrates particularly on the distinction between classical and sparse data flow analysis techniques. It aims to provide convincing experimental evidence that sparse analysis algorithms for SSI are more efficient than classical analysis algorithms for CFG, and more accurate than sparse analysis algorithms for SSA. Finally, this chapter explores the reasons why SSI gives superior performance.

4.1.2 Outline

Section 4.2 reviews the main concepts of data flow analysis. It highlights certain intrinsic properties of analyses, and certain properties of algorithms to compute such analyses. This chapter focuses particularly on sparse, flow-insensitive analysis. Subsequent sections describe different data flow analyses implemented using SSI, and contrast them to existing SSA and CFG implementations. In effect these are 'new analyses for old.' Three old data flow analyses are reformulated within a new SSI framework. Section 4.3 discusses constant propagation, for which SSI gives more accurate results than SSA. Section 4.4 discusses liveness analysis, for which SSI gives more accurate results than SSA, and more efficient results than CFG. Section 4.5 discusses type inference, for which SSI gives more efficient results than CFG. Section 4.6 investigates the theory underpinning sparse data flow analysis. It justifies why sparse analysis of SSI can produce more accurate results than SSA, as the empirical evidence clearly shows. It argues that that there is a live range splitting threshold for each particular analysis, beyond which no further improvements in accuracy can be achieved.

4.1.3 Contributions

This chapter has three novel contributions.

1. It provides detailed empirical studies of data flow analysis using SSI. There has been no such previous work. The most significant study in this chapter is an empirical assessment of the improved accuracy of constant propagation achieved by propagating predicate information at conditional branches.

2. It describes a SSI type inference algorithm. This is a sparse version of the recently proposed CFG type inference algorithm. SSI gives more efficient, and as accurate, results as the CFG-based version.

3. It gives insights regarding the accuracy of sparse data flow analysis, with the implicit assumption that data flow values are carried by live ranges (equivalently, by virtual registers). Previously, such information was compiler folklore; this work goes some way toward producing a rigorous result.

4.2 Classification of Analysis Techniques

This section introduces the basic concepts and terminology of data flow analysis. Note that the terms defined below are not used consistently in the literature. This dissertation attempts to use the terms consistently, with the meanings given below.

4.2.1 Fundamentals

A data flow *problem* is a high-level, natural language specification of some aspect of a program's runtime behaviour, which may be estimated at compile time by static analysis. This chapter considers three example data flow problems in detail:

1. constant propagation, and

2. liveness, and

3. type inference.

Data flow *information* is a mathematical abstraction of some aspect of a program's runtime behaviour. Generally, elements from a *property space* L model abstract properties related to specific program entities [NNH99]. A *combination operator* $\bigsqcup : \mathcal{P}(L) \to L$ is defined on L. The combination operator is often known as the *join operator*.[1] Generally, L is a complete lattice, and $\bigsqcup$ is the least upper bound operator for L. There must be some level of abstraction in the property space, to ensure that the data flow analysis is computationally efficient in terms of both time and space. Ultimately, abstraction is necessary in order to guarantee computability

A *transfer function space* is a set of transfer functions F (also known as *flow functions*) where each function $f \in F$ maps data flow information to data flow information ($f : L \to$

[1] The community is entirely polarized about $\bigsqcup$ and $\bigsqcap$. The difference appears to hinge on whether the formalism originates in Europe or America, or perhaps on whether the atmosphere is charged with theory or practice. This dissertation adheres to European conventions.

L). Transfer functions model changes in program state induced by the control flow and data flow paths through the program.

An *intermediate representation* R is a means of describing subject programs for analysis. Chapter 2 gives a broad characterization of IRs. Chapter 3 reviews the SSI IR in detail. R must provide two different kinds of handles. First, R must contain explicit entities (such as CFG program points) with which data flow information may be associated. Second, R must provide explicit information about program behaviour (such as abstract machine instructions) so that the appropriate transfer functions can be derived.

A data flow *framework* is a 3-tuple (L, F, R) where:

- L is the lattice of data flow information, and
- F is the space of transfer functions, and
- R is the IR specification.

Note that most classical definitions of a framework (such as [ASU86]) do not include R. Instead they assume R to be fixed as the CFG IR.

A data flow *analysis* (DFA) is a formal mathematical specification of a data flow problem. It is linked with a suitable data flow framework. The DFA provides a systematic method for deriving transfer functions for a program. A system of simultaneous *data flow equations* is derived from the program. These equations comprise the appropriate set of transfer functions to model the program behaviour. The solution to this system of equations is the result of the analysis. The result represents the compile time approximation to the runtime behaviour, for a given aspect of program behaviour.

A DFA *algorithm* is an effective procedure for setting up and solving a system of data flow equations for a given program. An algorithm is an implementation of an analysis. Conversely, an analysis is a specification of an algorithm. The classical DFA algorithm style [ASU86] is an iterative fixpoint computation. However there are other algorithm styles, such as constraint based analysis [NNH99] and interval analysis [Muc97].

4.2.2 Procedurality

Procedurality is an explicit property of the data flow problem. A problem can be either *intraprocedural* or *interprocedural*. The scope of an intraprocedural problem is a single procedure. Call instructions, if present at all, are treated as atomic instructions. Formal parameters are modelled as instruction source operands, and results as instruction destination operands.

Most intraprocedural problems have natural interprocedural analogues. However interprocedural analysis is more complex, both to formulate and to apply. The extra expense cannot always be justified. For instance, interprocedural constant propagation may provide useful information, but interprocedural available expressions analysis is almost certainly unnecessary.

This chapter only addresses *intraprocedural* problems. Chapter 6 extends SSI to handle *interprocedural* problems, in which data flow information propagates between caller and callee procedures.

4.2.3 Directionality

Directionality is an implicit property of the data flow problem. It refers to the direction in which data flow information travels, with respect to the direction of control flow. A data flow analysis inherits its directionality from the data flow property.

A data flow problem is *forward* if the data flow information at each point depends on the data flow information at control flow predecessors of that point. Forward data flow analysis propagates information in the same direction as the flow of control. Constant propagation (Section 4.3) is a forward problem.

A data flow problem is *backward* if the data flow information at each point depends on the data flow information at control flow successors of that point. Backward data flow analysis propagates information in the opposite direction from the flow of control. Liveness (Section 4.4) is a backward problem.

A data flow problem is *bidirectional* if the data flow information at each point depends on the data flow information at both control flow predecessors and successors of that point. Bidirectional data flow analysis propagates information both with and against the flow of control. Partial redundancy elimination (PRE) was originally formulated as a bidirectional data flow problem [MR79]. PRE attempts to remove equivalent computations that occur more than once along an execution path. PRE has since been decomposed into a fixed sequence of forward and backward (unidirectional) data flow problems [DRZ92, KRS94]. Type inference (Section 4.5) can be expressed as a bidirectional data flow analysis [KDM03]. Unlike PRE, type inference cannot be decomposed into a fixed sequence of unidirectional flows, it is a genuine bidirectional data flow problem [KD99].

4.2.4 Intermediate Representation

The IR in which the subject program is represented is a property of the analysis, since it is a parameter of the data flow framework for that analysis. Note that the data flow problem is entirely IR-agnostic. This chapter considers analyses which operate on the following IRs: CFG, WEB, SSA, and SSI. In each case, the analysis IR will be clearly stated.

4.2.5 Flow-Sensitivity

Flow-sensitivity is a property of the analysis, rather than the data flow problem. A *flow-sensitive* analysis takes account of control flow information present in the IR. Flow-sensitive analysis of CFG takes control flow edge information into account. In contrast, a *flow-insensitive* analysis does not consider control flow information. Flow-insensitive analysis treats a program as an unordered set of operations. An alternative formulation of flow-insensitive analysis rewrites a subject CFG program so there is an edge from every node to every other node. In general, flow-insensitive analyses can be implemented more efficiently, but their results are less accurate than equivalent flow-sensitive analyses. Marlowe et al [MRB95] give a full discussion of flow-sensitivity.

Note that there is no discrepancy between the directionality of a data flow problem and the flow-sensitivity of an analysis for that problem. Problem directionality is inherently represented in an implementing analysis, regardless of whether that analysis is flow-sensitive.

4.2.6 Context-Sensitivity

Context-sensitivity is a property of the analysis, rather than the data flow problem. Context-sensitive analyses are mostly useful for interprocedural problems. A *context-sensitive* analysis differentiates between the different calls of each procedure, so that data flow information that is only relevant in one calling context does not contaminate another calling context. In contrast, a *context-insensitive* analysis merges all information from all calling contexts for a procedure. As with flow-insensitive analyses, context-insensitive analyses are more efficient to implement, but they provide less accurate results. Since this chapter focuses on intraprocedural problems, it does not consider context-sensitivity any further. Chapter 6 discusses how to model context in interprocedural analysis.

4.2.7 Sparseness

Sparseness is a property of the analysis, rather than the data flow problem. A *classical* data flow analysis is *non-sparse* or *dense*. It stores data flow information at each program point. In contrast, a *sparse* analysis only stores data flow information where necessary. The goal of sparse data flow analysis is to avoid storing and propagating irrelevant data flow information. Because of this, sparse data flow analysis has the potential to be more efficient than classical data flow analysis. Admittedly this is a rather imprecise definition, but there is no clearer consensus on the meaning of sparseness in the literature. Sparse data flow analysis operates on subject programs represented in a sparse IR form.

A typical classical implementation of a virtual register based data flow analysis stores one unit of data flow information for each virtual register at each program point. A partially sparse implementation of the same analysis may store data flow information for each virtual register at fewer program points. In the limit, a completely sparse implementation of the analysis stores one unit of data flow information for each virtual register, that holds over the whole program. This contrast is illustrated in Figure 4.1. Classical data flow information is stored in a 2-dimensional array. Sparse data flow information is stored in a 1-dimensional array. This kind of sparse data flow analysis is generally associated with *flow-insensitive* analysis, as outlined in Section 4.2.5. The size of the sparse data flow information may be tuned by changing the number of virtual register names. This process is known as live range splitting, or virtual register renaming. This tuning process is one of the main themes of this dissertation.

This dissertation introduces a new definition of *sparse* analysis:

> An analysis is *sparse* if the data flow information associated with each virtual register has $O(1)$ space complexity.

In contrast, a classical analysis generally stores information that has at least $O(N)$ space complexity with each virtual register. Sparse data flow information grows as $O(N)$ for programs of abstract size N, whereas classical data flow information grows as $O(N^2)$ at least.

4.2.8 Performance

Performance is a property of the algorithm, rather than of the data flow analysis. However, since the algorithm implements the analysis, the algorithm performance is an indication of

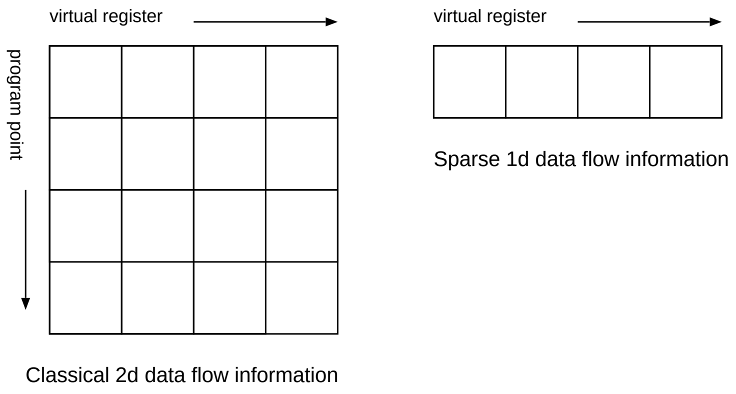

virtual register
program point
Classical 2d data flow information
virtual register
Sparse 1d data flow information

Figure 4.1: Contrast between classical and sparse data flow analysis

the analysis quality. Performance is usually measured in terms of *accuracy* and *efficiency*.

Accuracy is often a difficult metric to quantify. A data flow analysis algorithm computes data flow information results. Most researchers simply measure the code improvements resulting from the optimizations that depend on the data flow information, rather than measuring the information itself. This chapter does not report on optimizations, so a more abstract measure of accuracy is required. In effect, such a measure should indicate the potential scope for optimization opportunities. Section 4.3 introduces such a measure.

In contrast efficiency can be measured easily, in terms of the time taken by the algorithm to compute the data flow information. This chapter considers each algorithm in two distinct phases.

Precomputation: initial pass over the program that generates data flow information.

Query: subsequent lookup of data flow information by analysis clients.

The precomputation time is amortized over all queries. This division is unconventional. Most researchers assume that query time is $O(1)$ and that efficiency should reflect only the performance of the precomputation phase. Section 4.4 considers an example analysis where both precomputation and query times must be considered. Similarly, both phases are important for program slicing, which Chapter 5 describes.

4.3 Constant Propagation

This section uses the example analysis of *constant propagation* to demonstrate that different IRs afford different levels of accuracy for sparse analysis. The constant propagation algorithm is simple, therefore it can be implemented quickly and adapted easily. The analysis gives quantifiable results in terms of number of constants discovered. The trend of comparative results for the different IRs should carry over to alternative sparse analyses.

4.3.1 Problem

Constant propagation is a forward data flow problem, since values flow through the program in the same direction as control flow. It aims to track the flow of constant values

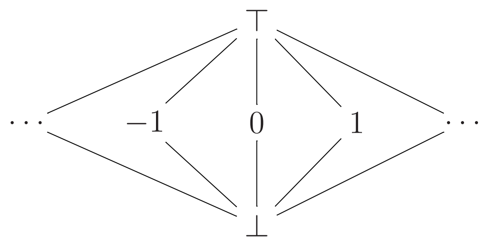


Figure 4.2: Integer constant propagation lattice

through the program. Constant values are introduced by instructions with immediate operands. Constant values are propagated by arithmetic operations and data transfers between virtual registers. Constant propagation information enables *partial evaluation*, which moves computation from runtime to compile time, to make the compiled code more efficient (faster and smaller).

4.3.2 Analysis Techniques

The analysis discovers whether a virtual register contains a constant integer value n (statically determinable), or is undefined (not assigned a value on any executable path through the program) or is overdefined (cannot be statically proved to hold the same constant value on all executable paths through the program). As already noted, this dissertation follows European conventions [NNH99], so $\bot$ denotes undefined and $\top$ denotes overdefined. Figure 4.2 shows the standard lattice ICP for integer constant propagation. The combination operator for ICP is least upper bound $\bigsqcup$. Kildall [Kil73] introduces ICP and describes a simple constant propagation analysis on CFG. Kildall's analysis is non-sparse. It computes a lattice value for each virtual register at each program point.

Wegman and Zadeck [WZ91] give a good survey of sparse techniques for constant propagation analysis. Their best analysis is *sparse conditional constant propagation*, which operates on SSA. The rest of this chapter refers to their analysis as $\mathsf{SCC}_{\text{ssa}}$. It has the following properties.

sparse: Each virtual register has a single associated lattice value from ICP, rather than one value per program point.

conditional: Basic block b that is a successor of a conditional branch c is not analysed if it can be statically determined (perhaps by means of constant propagation) that the outcome of c never allows b to be executed. This is a form of unreachable code elimination, combined with the constant propagation analysis.

partially flow-sensitive: Since the analysis takes into account control flow successor information at conditional branches in order to eliminate unreachable code from the analysis, it cannot be said to be flow-insensitive. However, $\mathsf{SCC}_{\text{ssa}}$ is not entirely flow-sensitive, according to the conventional definition [MRB95]. Control flow information is only used at conditional branches with predicates that can be statically computed as constants.

```
while ($change$)
    $change \longleftarrow$ false
    for each executable definition : $v \longleftarrow expr$
        $newCell \longleftarrow latticeCell[v] \sqcup$ symbolic-execute($expr$)
        if ($newCell \neq latticeCell[v]$)
            $latticeCell[v] \longleftarrow newCell$
            $change \longleftarrow$ true
            update-executability($v$)
```

Figure 4.3: The constant propagation algorithm for WEB, $\mathsf{SCC}_{\text{web}}$

Wegman and Zadeck show that $\mathsf{SCC}_{\text{ssa}}$ is faster than Kildall's analysis since $\mathsf{SCC}_{\text{ssa}}$ is sparse. They also show that $\mathsf{SCC}_{\text{ssa}}$ finds more constants since it is conditional. It is universally accepted that SSA constant propagation is better than CFG constant propagation for these reasons.

This section introduces two other sparse constant propagation analyses ($\mathsf{SCC}_{\text{web}}$ and $\mathsf{SCC}_{\text{ssi}}$) that are similar to $\mathsf{SCC}_{\text{ssa}}$. They share the same lattice, ICP, and transfer function space. The only difference is the IR in which subject programs are expressed. $\mathsf{SCC}_{\text{web}}$ uses WEB and $\mathsf{SCC}_{\text{ssi}}$ uses SSI. This experiment examines the effect that altering the degree of virtual register renaming has on the quality of constant propagation analysis.

Ananian [Ana99] introduces $\mathsf{SCC}_{\text{ssi}}$. It performs *predicated* analysis, since it propagates constants to σ-function destination operands at a conditional branch if the associated predicate forces one or more of the σ-function destination operands to have a constant value in one or more of the branch successors. Wegman and Zadeck hint at this extension in their original paper [WZ91] although they do not develop it formally. Ananian [Ana99] gives the first algorithmic description. However, this current work represents the first major study of the effect of this predicated constant propagation information on the accuracy of the analysis.

$\mathsf{SCC}_{\text{web}}$ is not formally described elsewhere. It applies the same sparse conditional constant propagation analysis as $\mathsf{SCC}_{\text{ssa}}$ to WEB form, with two differences:

1. no support for ϕ-functions, and

2. awareness of possible multiple definitions for a single virtual register.

Figure 4.3 presents the pseudo-code algorithm for $\mathsf{SCC}_{\text{web}}$. Function symbolic-execute() attempts to propagate constants through arithmetic operations, using the same symbolic execution logic as in $\mathsf{SCC}_{\text{ssa}}$. Function update-executability() determines which statements are executable, based on current knowledge about constant variables in conditional branch predicates, as in $\mathsf{SCC}_{\text{ssa}}$. Note that this algorithm sacrifices efficiency for clarity. A decent implementation would employ a worklist-based approach, as initially proposed by Wegman and Zadeck [WZ91].

4.3.3 Implementation Details

$\mathsf{SCC}_{\text{ssa}}$ has an existing implementation for Machine SUIF [Rol03], which conforms to Wegman and Zadeck's algorithm specification [WZ91]. I adapted this code to work on WEB ($\mathsf{SCC}_{\text{web}}$) and SSI ($\mathsf{SCC}_{\text{ssi}}$). $\mathsf{SCC}_{\text{web}}$ is not a genuine WEB-based implementation. It builds WEB on top of SSA (as described in Section 2.6.1). Web w has constant value c if at least one of the data flow values associated with the SSA virtual registers belonging to w is a constant lattice value c and all the others are either c or $\perp$.

In order to compare the properties of each sparse IR, I build WEB, SSA and SSI for a selection of SPEC CPU 2000 benchmarks, using Machine SUIF. Once I have obtained the sparse IRs, I can measure their relative sizes and amenability to analysis, using $\mathsf{SCC}_{\text{ssa}}$, $\mathsf{SCC}_{\text{ssi}}$ and $\mathsf{SCC}_{\text{web}}$.

4.3.4 Empirical Results

This section compares the three constant propagation analyses in terms of their efficiency and accuracy. An inverse relationship between efficiency and accuracy is clear to observe.

Differences in Efficiency

Analysis efficiency is directly related to IR size. Size measures the computational resources required to construct and analyse each IR. Larger representations take more time to compute, more memory to store, and more time to traverse.

Each of the three IRs (WEB, SSA and SSI) has an associated CFG. For an original program p, the CFG will be the same size (in terms of number of nodes) and shape (in terms of edges) when p is transformed into any of the three IRs, since they retain identical control flow structure in common with p, only now with different virtual register names as instruction operands. In addition, SSA and SSI require pseudo-definitions to be stored (ϕ- and σ-functions), however an earlier study [Sin03] has shown these extra operations to be negligible in comparison with the total number of instructions in the CFG.

Space (and time) costs for retaining (and recalculating) data flow information are the main factors that contribute to the differences in analysis efficiency across the three IRs. Recall that the amount of data flow information stored in sparse data flow analysis is directly proportional to the number of virtual registers. For the three IRs under consideration, each virtual register relates directly to a single live range, so the amount of data flow information stored is also directly proportional to the number of live ranges. This observation leads to a useful efficiency metric:

> The *number of virtual registers* mentioned in an IR form of a program is an indication of the efficiency of sparse analysis on that IR. A larger number of virtual registers leads to less efficient analysis.

Figure 4.4 shows the total number of virtual registers mentioned in each IR, for the compiled benchmark programs described in Section 4.3.3. This includes virtual registers defined by ordinary instructions, and virtual registers defined by pseudo-definitions in SSA and SSI.

The results in Figure 4.4 show that SSA defines 14% more virtual registers than WEB, on average. This extra live range splitting is in order to achieve the single assignment

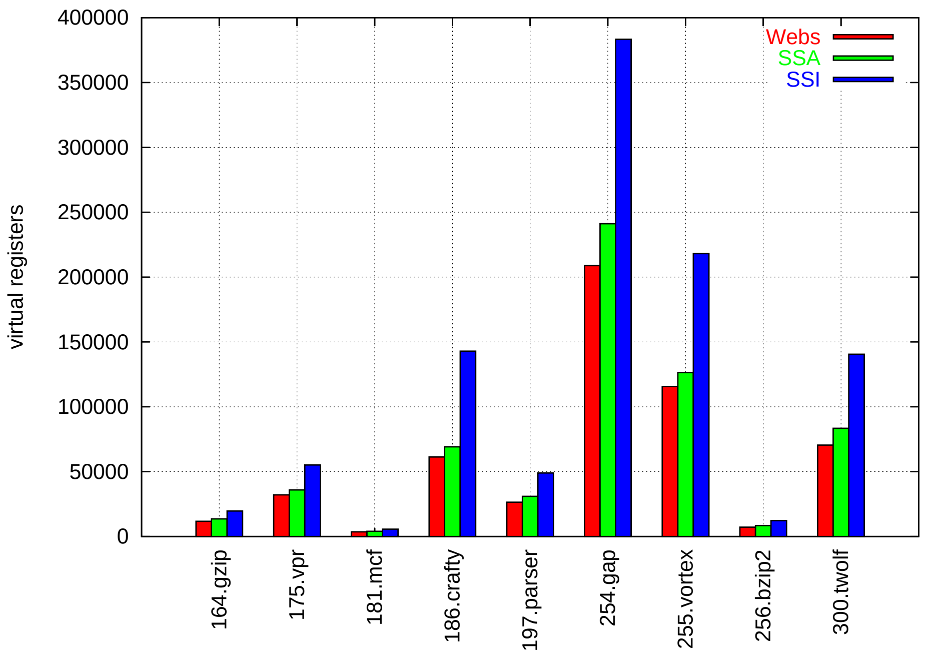


Figure 4.4: Number of virtual registers defined in each IR for several benchmarks

property. It appears that a substantial majority of webs only have a single definition. SSI defines 91% more virtual registers than WEB, on average. SSI performs much more live range splitting, so sparse data flow analysis on SSI will be correspondingly less efficient than on SSA or WEB, since there will be many more data flow values to compute and store.

Differences in Accuracy

Analysis accuracy is directly related to the number of discovered constants. However, it is important to present this information in an IR-invariant way. The results of the three constant propagation analyses are presented in Figure 4.5. The theoretical upper limit of constants that could be found by constant propagation analysis is given by the total number of instruction source operands, for each benchmark. Note that some virtual registers are counted more than once, if they appear as source operands in more than one instruction. The number of instruction source operands is identical for each sparse IR, since only instructions from the underlying CFG are included; source operands of pseudo-definition functions in SSA and SSI are ignored. For each of the three analyses ($\mathsf{SCC}_{\text{web}}$, $\mathsf{SCC}_{\text{ssa}}$, and $\mathsf{SCC}_{\text{ssi}}$) the results show the proportion of instruction source operands (in relation to the total number of instruction source operands) that are virtual registers marked as constant or undefined ($\perp$). This is a useful accuracy metric:

> The *proportion of source operands* that are marked constant or undefined is an IR-independent quantitative indication of the accuracy of a constant propagation analysis.

Note that any difference in the numbers of constants discovered in different IRs is a difference in *real terms*. The constant propagation results are not reported as ‘number of

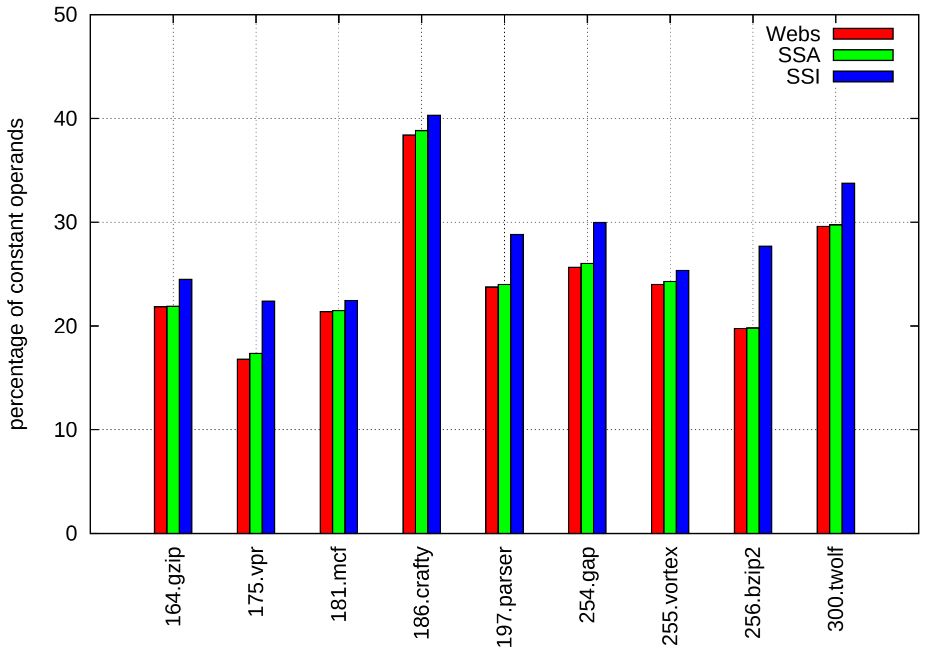


Figure 4.5: Number of calculated constant virtual register instruction source operands in each IR, compared with total number of instruction source operands, for several benchmarks

virtual registers found to be constant' since this is not a fair comparison across the three IRs. One constant-valued virtual register in WEB may map onto three constant-valued virtual registers in SSA, with no actual increase in results accuracy! Instead, results are reported as 'proportion of virtual register instruction source operands found to be constant' which is a fair comparison across the three IRs since the metric is invariant over virtual register renaming.

The results in Figure 4.5 clearly show that $\mathsf{SCC}_{\text{web}}$ and $\mathsf{SCC}_{\text{ssa}}$ give almost identical accuracy. On average, $\mathsf{SCC}_{\text{web}}$ calculates 24.5% of the total number of instruction source operands to be constant, and $\mathsf{SCC}_{\text{ssa}}$ calculates 24.8% to be constant. So $\mathsf{SCC}_{\text{ssa}}$ only discovers 0.3% more than $\mathsf{SCC}_{\text{web}}$, as a proportion of the total number of instruction source operands, averaged over all benchmark tests. $\mathsf{SCC}_{\text{ssi}}$ calculates 28.4% of the total number of instruction source operands to be constant. So $\mathsf{SCC}_{\text{ssi}}$ discovers 3.6% more than $\mathsf{SCC}_{\text{ssa}}$, as a proportion of the total number of instruction source operands, averaged over all benchmark tests.

4.3.5 Discussion

In this study, the constant propagation analysis results are not actually used to perform any optimization. Some virtual registers are more important than others since they have longer live ranges or are used more frequently. Thus replacing some virtual registers by immediate constant values has a greater optimization benefit. This study does not give any insight into which constant virtual registers are most important. It is difficult to make this judgement statically, ideally it requires dynamic analysis. Therefore this study simply presumes that an IR that enables a larger number of constants to be discovered is more effective in general. Note that due to the common data flow framework of $\mathsf{SCC}_{\mathrm{web}}$, $\mathsf{SCC}_{\mathrm{ssa}}$ and $\mathsf{SCC}_{\mathrm{ssi}}$, all the constants discovered by $\mathsf{SCC}_{\mathrm{web}}$ will also be discovered by $\mathsf{SCC}_{\mathrm{ssa}}$ and $\mathsf{SCC}_{\mathrm{ssi}}$, and all the constants discovered by $\mathsf{SCC}_{\mathrm{ssa}}$ will also be discovered by $\mathsf{SCC}_{\mathrm{ssi}}$.

Recall the relationship between analysis efficiency and accuracy. SSI is clearly the least efficient IR since it mentions the most virtual registers, but it is the most accurate IR. However there is an element of diminishing returns here. From the empirical study, SSI programs mention 90% more virtual registers than WEB, but $\mathsf{SCC}_{\mathrm{ssi}}$ can only discover 13% more constants than $\mathsf{SCC}_{\mathrm{web}}$, in relative terms. SSA is certainly the most popular of these three IRs. In this empirical study, SSA loses on accuracy (to SSI) and on efficiency (to WEB). SSA is the median IR in terms of both efficiency and accuracy. Perhaps this compromise is an explanation for SSA's widespread popularity. It seems that WEB would be a better choice of IR than SSA, since it provides almost-as-accurate results, without the overhead of ϕ-functions.

4.3.6 Related Work

Chapter 3 compared SSA and SSI in terms of construction algorithm performance and IR size. However, this earlier work does not assess the relative amenability of each IR to sparse data flow analysis.

Ananian [Ana99] gives a limited empirical assessment of the relative performance of sparse conditional constant propagation on SSA and SSI. His results are measured in terms of dynamic instruction counts before and after optimizations due to constant propagation. He reports that SSI enables a 10% reduction in the number of instructions executed, after applying optimizations due to constant propagation. In contrast, his SSA-based analysis does not reduce the number of instructions executed at all! However, he only tests two programs, one of which is 'Hello world.' He gives few details, but his SSI-based algorithm operates on classes and arrays (in Java) whereas his SSA-based algorithm only seems to handle simple scalar constants (although this is not particularly clear). Our empirical comparison between SSA and SSI is fair since the algorithms share the same data flow framework. The comparison is much more thorough than Ananian's, and also measures the performance of WEB.

At first sight it may appear that the current work only addresses a limited number of bit vector data flow analyses, that calculate properties of individual virtual registers. Constant propagation is the example analysis presented in Section 4.3.4. It is not obvious that the trend of increasing accuracy due to more live range splitting will also hold for other analyses. Recent work on alias analysis [CDC+04] shows empirically that the accuracy of scalar analyses for SSA increases slightly as more precise alias information is provided. One of these analyses is sparse conditional constant propagation, so it seems fair to assume that since constant propagation follows the general trend of increasing accuracy in that study, then constant propagation is an indication of the general trend in this study also.

4.4 Liveness

This section shows that SSI may be used as the basis for efficient sparse *liveness analysis*. The sparse approach is contrasted with classical liveness analysis on CFG.

4.4.1 Problem

Liveness is a property of virtual registers. Recall from Section 3.3.1 that a virtual register is *live* if its present value is required at some point in the future. Thus liveness is intrinsically associated with an ordering of operations. Liveness information is only applicable when there is such an ordering, imposed by control flow or data dependence or some similar notion. Liveness is a *backward* data flow problem, since liveness information depends on future program behaviour. This is the opposite of control flow, which depends on past program behaviour.

4.4.2 Analysis Techniques

As with most data flow analyses, liveness may be computed in a classical (dense) or sparse manner.

Classical Analysis

Classical liveness analysis is a bit vector data flow analysis that operates on CFG. A single bit of information is stored for each virtual register at each program point. This is generally implemented by associating a live_{in} and live_{out} set with each CFG instruction, to represent the virtual registers live immediately before and after execution of this instruction. When virtual register v is live, the appropriate bit is set. When v is dead, the appropriate bit is unset. All the static analysis textbooks describe this classical approach to liveness [ASU86, NNH99, Muc97].

The data flow equations are:

$$\begin{aligned}
\text{live}_{\text{out}}(n) &= \bigcup_{s \in \text{successors}(n)} \text{live}_{\text{in}}(s) \\
\text{live}_{\text{in}}(n) &= \text{live}_{\text{out}}(n) \setminus \text{def}(n) \cup \text{use}(n)
\end{aligned}$$

The classical approach to liveness can be divided into two components, the *precomputation* and the *query*. This division is not normally made, but it is helpful for the contrast between the classical and sparse approaches. The precomputation sets up and solves the data flow equations for liveness, filling in the bit vectors at each program point. This operation has $O(N^4)$ time complexity in the worst case, for a CFG with N nodes. The query returns the result of a single request for liveness information: "Is virtual register v live at point p?" The necessary computation is a single bit vector lookup, which has $O(1)$ time complexity.

Sparse Analysis

Sparse liveness analysis can be considered to transpose the classical liveness analysis matrix of data flow information. Whereas classical liveness stores a bit vector of virtual registers for each program point, sparse liveness stores a vector of program points (at which liveness information changes) for each virtual register. This sparse representation of data flow information seems more appropriate since liveness information does not change frequently in general. Thus the sparse layout may eliminate the problem of duplicated information in the classical layout.

In SSI, each virtual register has a single live range. Each live range has a unique initial point (the single definition of that virtual register) and a unique terminal point (the last use of that virtual register). So, sparse liveness analysis simply stores the points $\mathit{uniquedef}(v)$ and $\mathit{lastuse}(v)$ for each virtual register v. This encapsulates the live range of each virtual register. If a program point p is in the live range of virtual register v, then p will be dominated by $\mathit{uniquedef}(v)$ and postdominated by $\mathit{lastuse}(v)$, due to the properties of SSI.

Again, this sparse approach to liveness can be divided into *precomputation* and *query* components. The precomputation transforms the subject program from CFG to SSI, and computes all dominance and postdominance information. We assume that this cost is amortized over many analyses, since SSI will be used as the primary IR for lots of data flow analysis phases. Now it is necessary to calculate $\mathit{uniquedef}(v)$ and $\mathit{lastuse}(v)$ for each virtual register v. It is straightforward to construct lookup tables for this information at the same time as SSI construction, with no increase in the time complexity of the construction algorithm. Alternatively, the tables can be built from a SSI program in time $O(N^2)$ in the worst case, for a program with N nodes. The query returns a result for a single request for liveness information: "Is v live at p?" It simply determines whether $\mathit{uniquedef}(v)$ dominates p and $\mathit{lastuse}(v)$ postdominates p. If so, then v is live at p. If all dominance information is precomputed, this query takes $O(1)$ time, although it is slightly more complicated than the single bit vector lookup in the classical approach. There are concerns that the liveness information produced from SSI may not be accurate, due to extraneous pseudo-definition functions. Section 4.4.5 gives more details.

4.4.3 Implementation Details

I have implemented both classical and sparse liveness analyses in Machine SUIF. The classical analysis uses the standard bit vector data flow analysis library [HD98]. The sparse analysis uses custom SSI extensions to Machine SUIF presented in Chapter 3. Classical precomputation time is the time to construct the liveness bit vectors given an input CFG program. Sparse precomputation time is the time taken to calculate the *lastuse* information for each virtual register, given an input SSI program with dominance information and *uniquedef* information.

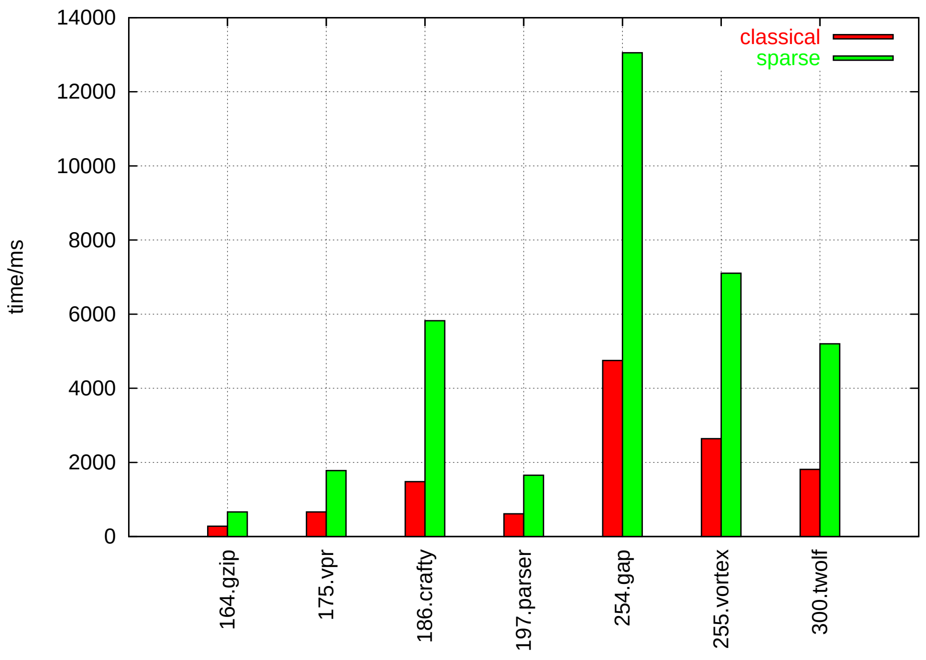


Figure 4.6: Precomputation time for liveness analysis passes

4.4.4 Empirical Results

Precomputation

Figure 4.6 compares the precomputation times for the classical and sparse liveness analysis passes described above, applied to a selection of SPEC benchmark programs. The tests were carried out on a lightly loaded AMD Athlon 1.4GHz machine with x86 Linux. Each time result is the the arithmetic mean of five measurements. Note that on average, the sparse analysis takes three times longer than the classical analysis. Asymptotically, this ratio should decrease as program size increases.

In the limit, the sparse precomputation time could be reduced to 0, if the *lastuse* information is computed at construction time and incorporated into the IR scaffolding. This is feasible if there are other analyses that would find the *lastuse* information helpful.

Query

Section 4.4.2 notes that both classical and sparse queries have $O(1)$ time complexity. In the classical liveness implementation, a single liveness query equates to a function call, which simply performs an array lookup. In the sparse liveness implementation, a single liveness query equates to a function call that executes a total of 9 lines of simple C code, including inlined function calls to `dominates` and `postdominates` functions, that simply perform array lookups.

4.4.5 Discussion

As usual with DFA, there are different trade-offs to make when deciding whether to adopt a classical or a sparse liveness analysis. In general, classical information takes longer to precompute, but shorter to query. On the other hand, sparse information can potentially take less time to precompute (assuming most of the work is factored into the SSI construction pass as Section 4.4.4 mentions) but sparse information takes longer to query. Classical liveness analysis can be considered as a *eager* strategy for DFA, since it precomputes all the necessary information about liveness at each program point for each virtual register. On the other hand, sparse liveness analysis can be considered as a *lazy* strategy for DFA, since it only actually computes liveness information on demand. The correct choice depends on the behaviour of the analysis client that is making the liveness queries. For instance, a client that is constructing a register interference graph requires all liveness information, so would be better served by the eager classical approach. However, for other clients that have few liveness queries, it may be better to adopt the lazy sparse approach.

Note that laziness can be increased, by computing *lastuse* information lazily as well. This can only be justified when clients make extremely few liveness queries.

There are two problems with the sparse liveness analysis outlined above.

1. Sparse liveness information on SSI operates in terms of SSI virtual register names. Recall that an original CFG virtual register name may map onto several SSI names. So the sparse liveness information only makes sense when all of the analysis clients are working in terms of SSI as well. In that sense, it is not really a fair comparison between CFG and SSI, since CFG live ranges can have much more complicated shapes than SSI live ranges. However, it is possible to convert between SSI and CFG namespaces, perhaps analysis clients could perform this conversion automatically when necessary.

2. SSI liveness information can only give the same answers as classical CFG liveness (modulo renaming) on *pruned* SSI. Otherwise, superfluous pseudo-definition functions extend the live ranges of virtual registers for longer than is necessary. *Faint* virtual registers [NNH99] are live, but they are only used to define other faint or dead virtual registers. Pseudo-definition functions provide a common source of faintness, since they often define virtual registers that are never subsequently used. One possibility is to perform dead code elimination to remove such superfluous pseudo-definitions. Chapter 5 deals with this topic. Note that if virtual register v is defined but never used, $lastuse(v)$ does not exist and so it is clear to see that v is globally dead. Also note that overestimation of live ranges is inaccurate, but always *safe* for subsequent transformations that depend on liveness information.

It is commonly acknowledged that SSA cannot be used for sparse liveness analysis, since SSA can only be used for sparse forward data flow analysis [CCF91, JP93]. However, SSA could be used in the sparse scheme outlined above, that tests whether point p is in the live range of v. Recall that a SSA live range must have one initial point (the unique definition of the virtual register) but it can have multiple terminal points (last uses of the virtual register). So the SSI sparse liveness algorithm from Section 4.4.2 can be modified for SSA, by storing *all* the last uses of v with each virtual register v, as well as the unique definition of v. Then when a liveness query is processed, the SSA algorithm checks p against all the last uses of v. If every path from p to n_{exit} contains at least one of the last uses of v, then p is in the live range of v, so v is live at p. However, this is not truly sparse, since the amount of information that is associated with each virtual register v is not $O(1)$. In the worst case, there can be $O(N)$ last uses of v in an SSA program with N nodes, which is as bad as classical analysis that stores $O(N)$ bits of information for each virtual register v. It is possible to use SSA for much more *inaccurate* liveness analysis that is truly sparse. For all virtual registers v, set $\mathit{lastuse}(v)$ to be n_{exit}. This leads to the following formulation of sparse liveness analysis in SSA:

> Virtual register v is live at program point p if $\mathit{uniquedef}(v)$ dominates p and n_{exit} postdominates p.

In this case, each virtual register has $O(1)$ units of associated data flow information. However live ranges are overestimated greatly. The analysis results will be inaccurate, but safe.

4.4.6 Related Work

There have several recent attempts to formulate a sparse version of liveness analysis. This section reviews these approaches, and contrasts them with the SSI sparse liveness analysis outlined in Section 4.4.2.

Choi et al [CCF91] introduce the sparse evaluation graph (SEG). This is a sparse IR that can be specialized to any particular data flow analysis. SEG is derived from CFG; it only retains nodes for which data flow information may be generated or combined in the CFG. Choi et al show how the SEG may be used for liveness analysis.

Reference: [CCF91]

IR: SEG

Precomputation: Extract SEG from CFG (similar to constructing SSA from CFG). Perform liveness analysis on SEG (more efficient than CFG due to sparse nature of SEG).

Query: Map query point p in CFG onto point p' in SEG. Is v live at p'?

Number of virtual registers: An SEG is specialized to model the liveness properties of one particular virtual register. When a query is made about a different virtual register, a new SEG must be constructed.

Names of virtual registers: SEG retains original CFG names for virtual registers.

Stoltz [Sto95] introduces general reference chains (GRC). These are like def-use chains, in that they are built on top of CFG. Merge operators (similar to SSA ϕ-functions) coalesce data flow information at confluence points. Reference chains link relevant program points that share common data flow information (similar to SSA virtual register renaming). The GRC IR that handles sparse liveness analysis is called the λ-chains IR.

Reference: [Sto95]

IR: λ-chains

Precomputation: Build λ-chains on top of CFG (similar to constructing SSA from CFG). Use λ-chains to calculate classical-style bit vectors for liveness (calculation is more efficient than CFG due to sparse nature of λ-chains).

Query: Classical bit vector lookup.

Number of virtual registers: λ-chains IR models liveness for all virtual registers at once.

Names of virtual registers: Retains original CFG names for virtual registers.

Ananian [Ana99] briefly outlines how SSI may be used to perform backward data flow analyses such as liveness and anticipatability. However, he does not present any algorithmic details.

Reference: [Ana99]

IR: SSI

Precomputation: Transform CFG to SSI.

Query: Not clear from Ananian's short and informal description. He states that the live range for virtual register v can be enumerated by a depth-first search of the dominator tree from the definition of v to all the uses of v. He certainly does not discuss the insight that a unique *lastuse* operation is sufficient to encapsulate an entire live range.

Number of virtual registers: SSI contains information about all virtual registers at once.

Names of virtual registers: Uses SSI virtual register names, which may be mapped back onto original CFG names.

The main disadvantage of SEG and λ-chains is that these IRs are specially designed for liveness analysis. They cannot be used for other data flow analysis. Ruf describes these as *analysis-specific* sparse IRs [Ruf95b]. In contrast, SSI is a general-purpose IR that can be used for other analyses as well as liveness. Thus the construction costs of SSI are amortized over many data flow analyses.

Stoltz's approach [Sto95] is not truly sparse, since he constructs a classical bit vector representation from the λ-chains IR. The SEG liveness algorithm is sparse, since it maps CFG program points onto SEG program points, and computes liveness on SEG. However each SEG only represents the liveness property for a single virtual register. Hence for a CFG with K virtual registers, the approach of Choi et al [CCF91] requires K SEGs, which does not seem to be sparse at all! Ananian's definition of SSI liveness analysis is unclear [Ana99]. His approach could eagerly calculate live ranges for each virtual register, and construct the classical bit vector representation. However, as with Stoltz, this would not be a genuine sparse approach. Alternatively, his approach could lazily determine live ranges on demand, as each liveness query is processed. This is the basis of the sparse liveness analysis presented in Section 4.4.2, which is a truly sparse approach. There is a constant amount of information associated with each virtual register (namely the definition point and the last use point). This information is examined when a liveness query is made.

4.5 Type Inference

Type inference can be expressed as a bidirectional data flow problem. This section demonstrates how to perform type inference using sparse data flow analysis, by representing programs in SSI. Type inference is the most outstanding bidirectional problem that cannot be decomposed into a fixed sequence of unidirectional data flow problems [KDM03]. However it should be possible to devise a sparse data flow analysis for any bidirectional problem, using SSI.

4.5.1 Problem

Values are associated with virtual registers and expressions in a program. Each virtual register and expression is known as a *term*. Each term has a *type*, which determines the maximum set of values that term may hold at runtime. A *type system* is a formal mathematical model that describes how values are partitioned into distinct types, and how such types may be combined. *Type inference* [Car97] is the process of discovering the derivation of types for terms in a subject program, within a given type system. Type inference determines the type of a term t from the contexts in which t is mentioned in the program. This is also known as *type reconstruction*.

This problem is inherently bidirectional. Sometimes type information is propagated forward from the context of a virtual register definition to the context of a virtual register use. However, sometimes precise type information is not available at the definition, so it must be propagated backward from the use to the definition. Both virtual register definitions and uses may generate type information, since the type is important both in the initial definition and in the subsequent uses of a virtual register.

4.5.2 Analysis Techniques

Constraint-Based Analysis

The standard approach to type inference [Mil78] generates constraints on types during a single flow-insensitive pass through the subject program. These type constraints are then solved by unification. The standard Damas-Hindley-Milner algorithm is used in functional languages like ML [MTHM97] and even in some imperative systems [OJ97].

A distinction is generally made between *static type constraints*, which require a virtual register to have the same type throughout the entire program, and *dynamic type constraints*, which allow a virtual register to have different types in different parts of the program. Mycroft [Myc99] shows, in the context of decompilation, that dynamic type constraints are often reduced to static type constraints when the subject program is transformed to SSA. This occurs because of the live range splitting enforced by the SSA transformation. Since SSI is an extension of SSA that performs at least as much live range splitting as SSA, such dynamic type constraints are reduced to static type constraints in SSI as well as in SSA. Indeed, since SSI enforces more live range splitting than SSA, more dynamic type constraints may be reduced to static type constraints. For instance, consider the program below.

```
define $x$ as type $a$ or type $b$
if (...)
    then use $x$ as type $a$
    else use $x$ as type $b$
```

A SSA-based analysis would infer that x has the union type of a or b at the definition point, and then x has type a in the `then` branch and type b in the `else` branch. These are dynamic type constraints. On the other hand, an SSI-based analysis would distinguish between the uses of x in the two conditional contexts as shown below.

```
define $x_0$ as type $a$ or type $b$
if (...)
$x_1, x_2 \longleftarrow \sigma(x_0)$
    then use $x_1$ as type $a$
    else use $x_2$ as type $b$
```

Now there are three different virtual registers for the three different types. x_0 will have the same union type as x above, x_1 has type a and x_2 has type b, based on the contexts in which they are used. Now there are no dynamic type constraints to be satisfied.

Conventional Data Flow Analysis

Type inference can be expressed as classical data flow analysis. The most authoritative recent work on the subject is by Khedker et al [KDM03]. One of their examples is presented later in this section. They define the first formal data flow framework for type inference that relies on bidirectional data flow analysis. They show that the problem is truly bidirectional and cannot be decomposed into a fixed sequence of unidirectional flows, unlike partial redundancy elimination [KD99]. They formulate a flow-sensitive classical CFG-based data flow analysis for type inference. This calculates a type for each virtual register at each program point. It has bidirectional data flow equations. The analysis has the following properties.

extensive information propagation: It uses information from all CFG nodes, not just those directly reachable by forward and reverse control flow paths.

precise information propagation: It encapsulates the influence of data flow information relative to the source of that information. Information that is generated at a point near to node n is more likely to be relevant at node n than information that is generated far from node n.

theoretical characterization of information propagation: It is possible to estimate the complexity of the analysis since it is specified formally.

V ::= virtual registers	
T ::= primitive types	(no type constructors)
C ::= **read** V	(assigns new value to V from input no constraint on type of V)
| **print** V	(uses value of V as output, no constraint on type of V)
| **define** V **as** T	(assigns new value to V, V has type T)
| **use** V **as** T	(uses value of V, V has type T)
| **if** V **then** C **else** C	(V is integer)
| **while** V **do** C	(V is integer)

Figure 4.7: Context-free grammar for simple programming language with dynamic type constraints

Existing DFA-based type inference systems are implemented in the classical style. However type inference can also be performed using *sparse* DFA techniques. This work concentrates on type inference for languages with dynamic type constraints. Recall that dynamic type constraints allow a virtual register to be treated as having different types in different contexts in which it is mentioned. Figure 4.7 gives a simple context-free grammar for an abstract programming language. It distinguishes between **define** statements that assign a new value to a virtual register, and **use** statements that read an existing value from a virtual register. It is unclear whether Khedker et al make this distinction [KDM03]. Their paper makes no mention of **define** statements at all. This section assumes that all their **use** statements are equivalent to **use** statements in this abstract programming language.

Figure 4.8 shows an example program taken from Figure 6 in [KDM03]. This example program has been transformed into SSI. The original version can be recovered simply by eliding ϕ- and σ-functions and numeric virtual register subscripts. Only program statements that mention virtual register a are shown in Figure 4.8, for the sake of simplicity. Of these statements, only **use** statements, and ϕ- and σ-functions are relevant to type inference.

The classical analysis calculates the type of each virtual register at each program point. Khedker et al [KDM03] qualify this data flow information with the degree of certainty (*unknown*, *may*, *must*, *mustnot*) and with the origin of that information in terms of control flow (*current*, *ancestor*, *descendant*, *other*). The sparse analysis presented in this section only qualifies type information with its origin, in order to simplify the presentation. So the degree of certainty is *must*, in terms of the original classical analysis. Uncertainties are represented by a union type. It should be straightforward to extend the sparse analysis to handle degrees of certainty in the same way as the classical analysis.

A control flow *ancestor* of node n is a node a that is passed through on at least one control flow path from n_{entry} to n. A control flow *descendant* of node n is a node d that is passed through on at least one control flow path from n to n_{exit}. Note that such control flow paths may contain cycles and may include n, so an ancestor of n may also be a descendant of n. It is necessary to remember the origin of type information, since the type information propagated to a node from one program point may conflict with the type information propagated from another point. Type information generated at the current node is the most reliable. If this is not available then type information from ancestors should take precedence over type information from descendants because at runtime, control flows from ancestors to descendants. Type information propagated from descendants should only be used where useful information is not available from ancestors. Type information propagated from a node other than an ancestor or a descendant should only be used as a last resort. The Γ operator below selects the most appropriate source of type information available. Note that information generated at the current node about virtual register v corresponds to the *concrete type* of v. The union of all the information about virtual register v generated at all nodes corresponds to the *abstract type* of v.

The primitive types used in this type system are integer i, real r and string s. The set of all types, $\mathcal{T}$, is defined as $\{\mathsf{i}, \mathsf{r}, \mathsf{s}\}$. The component data flow lattice $\hat{\mathcal{L}}$ is the power set of $\mathcal{T}$. The minimal element $\hat{\bot}$ is the empty set. The maximal element $\hat{\top}$ is the set $\mathcal{T}$. The partial order $\hat{\sqsubseteq}$ is the standard set inclusion operator $\subseteq$. The join operator $\hat{\sqcup}$ on lattice elements is the standard set union operator $\cup$. (Note the duality with the classical notation [KDM03], due to their American conventions. This dissertation prefers European style.)

Now the compound lattice $\mathcal{L}$ is defined in terms of the component lattice. Each compound lattice element X has four component elements $\langle X_c, X_a, X_d, X_o \rangle$, each of which is a member of $\hat{\mathcal{L}}$.

X_c represents type information generated at current nodes, that is, at **use** statements. X_a represents type information generated at ancestor nodes, that is, propagated forward through ϕ- and σ-functions. X_d represents type information generated at descendant nodes, that is, propagated backward through ϕ- and σ-functions. X_o represents type information generated at some other node, that is, indirectly in terms of control flow.

The minimal element $\bot$ is $\langle \hat{\bot}, \hat{\bot}, \hat{\bot}, \hat{\bot} \rangle$. The maximal element $\top$ is $\langle \hat{\top}, \hat{\top}, \hat{\top}, \hat{\top} \rangle$. The partial order $\sqsubseteq$ is defined as:

$$X \sqsubseteq Y = (X_c \hat{\sqsubseteq} Y_c) \wedge (X_a \hat{\sqsubseteq} Y_a) \wedge (X_d \hat{\sqsubseteq} Y_d) \wedge (X_o \hat{\sqsubseteq} Y_o)$$

The join operator $\sqcup$ is defined as:

$$X \sqcup Y = \langle X_c \hat{\sqcup} Y_c, X_a \hat{\sqcup} Y_a, X_d \hat{\sqcup} Y_d, X_o \hat{\sqcup} Y_o \rangle$$

It is necessary to define specialized operators $\sqcup_a$ and $\sqcup_d$, which are used to ensure that type information flows to the correct components of the compound lattice elements. $X \sqcup_a Y$ ensures that the data flow information from Y is treated as ancestor information for X. So, Y's current and ancestor information is merged with X's ancestor information. Y's descendant and other information is merged with X's other information. This

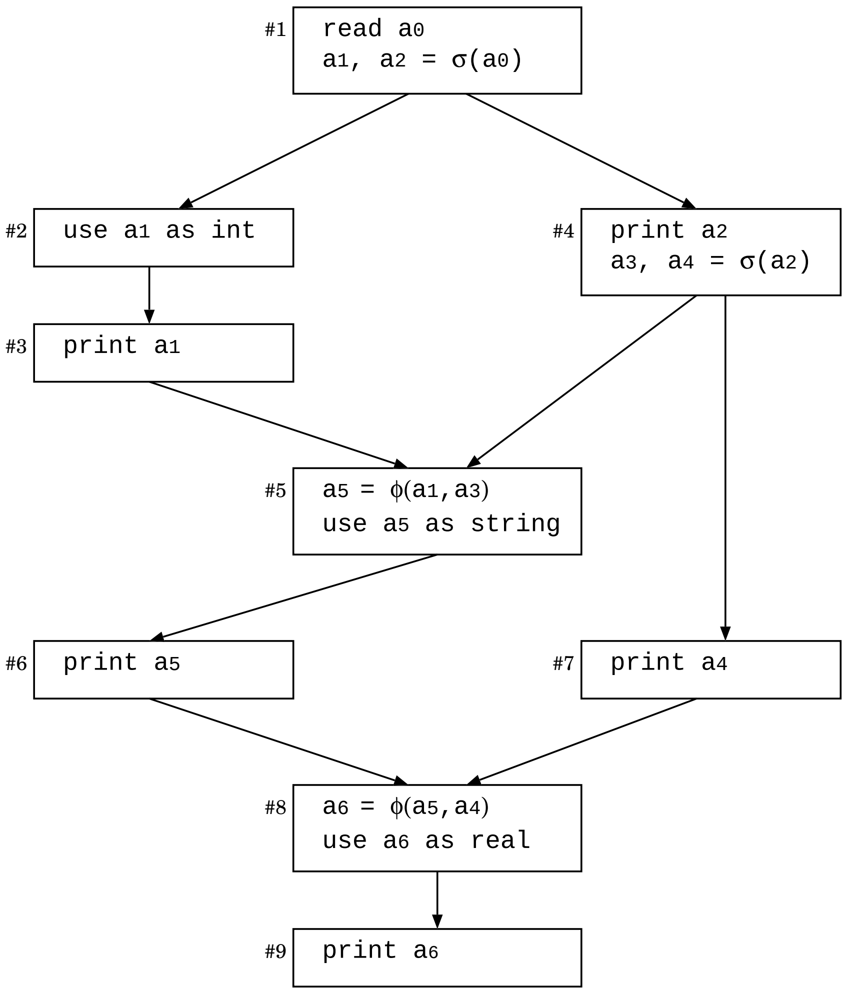


Figure 4.8: Example program from [KDM03]

(def)	**define** x **as** t	$E^x_c := E^x_c \hat{\sqcup} \{\mathsf{t}\}$
(use)	**use** x **as** t	$E^x_c := E^x_c \hat{\sqcup} \{\mathsf{t}\}$
$(sigma)$	$x_1, x_2 \leftarrow \sigma(x_0)$	$E^{x_0} := E^{x_0} \sqcup_d (E^{x_1} \sqcup E^{x_2})$ $E^{x_1} := E^{x_1} \sqcup_a E^{x_0}$ $E^{x_2} := E^{x_2} \sqcup_a E^{x_0}$
(phi)	$x_0 \leftarrow \phi(x_1, x_2)$	$E^{x_0} := E^{x_0} \sqcup_a (E^{x_1} \sqcup E^{x_2})$ $E^{x_1} := E^{x_1} \sqcup_d E^{x_0}$ $E^{x_2} := E^{x_2} \sqcup_d E^{x_0}$

Figure 4.9: Type inference rules for abstract programming language

avoids information from one ancestor contaminating another ancestor through a common descendant, and other similar cases. Note that these operators correspond to the forward ($\sqcup_a$) and backward ($\sqcup_d$) edge flow functions of [KDM03].

$$X \sqcup_a Y = \langle X_c, X_a \hat{\sqcup} (Y_c \hat{\sqcup} Y_a), X_d, X_o \hat{\sqcup} (Y_d \hat{\sqcup} Y_o) \rangle$$

$$X \sqcup_d Y = \langle X_c, X_a, X_d \hat{\sqcup} (Y_c \hat{\sqcup} Y_d), X_o \hat{\sqcup} (Y_a \hat{\sqcup} Y_o) \rangle$$

The Γ operator below selects a precise estimate of the type from a compound lattice element X by giving precedence to information generated at the current node, if any. Otherwise, ancestor information is preferred over descendant information which is preferred over other information, as outlined above.

$$\Gamma(X) = \begin{cases} X_c, & X_c \neq \hat{\perp} \\ X_a, & X_c = \hat{\perp} \wedge X_a \neq \hat{\perp} \\ X_d, & X_c = \hat{\perp} \wedge X_a = \hat{\perp} \wedge X_d \neq \hat{\perp} \\ X_o & \text{in all other cases} \end{cases}$$

E represents the single global vector of data flow information that characterizes sparse data flow analysis. E is indexed by SSI virtual register names. E^v represents the compound lattice element associated with virtual register v.

Figure 4.9 gives the actual inference rules for this type system. Note the bidirectionality for ϕ- and σ-functions, where information is propagated in both directions with respect to control flow.

Initially each element of E is set to $\perp$. A single iteration of the data flow analysis processes the program statement-by-statement and applies the appropriate inference rules. The vector E is updated as specified by the inference rules. This iterative pass is repeated until a fixed point is reached, that is to say, E does not change at all after an entire iterative pass. (The order in which the statements are processed in the iterative pass is irrelevant to the final state of E, although it may have some effect on the number of iterations required to reach a fixed point.)

4.5.3 Implementation Details

Plans for future work include the development of a real-world implementation of this type inference mechanism. At the moment, I have merely produced simple proof-of-concept implementations. I have implemented this type inference algorithm as an ML program using the sparse data flow analysis framework given above.

4.5.4 Empirical Results

The ML proof-of-concept implementation of this type inference system achieves the same results as [KDM03]. It uses a single lattice element for each virtual register, and takes six passes through the program. In summary, the results are:

virtual register v	a_0	a_1	a_2	a_3	a_4	a_5	a_6
$\Gamma(E^v)$	$\{\mathsf{i},\mathsf{s},\mathsf{r}\}$	$\{\mathsf{i}\}$	$\{\mathsf{s},\mathsf{r}\}$	$\{\mathsf{s},\mathsf{r}\}$	$\{\mathsf{r}\}$	$\{\mathsf{s}\}$	$\{\mathsf{r}\}$

4.5.5 Discussion

The origin of data flow information is important, because sometimes there will be conflicting information from different sources, and the most reliable information should be preferred. This requires the representation of flow-sensitive details (about ancestors and descendants) within the data flow information, which may cause problems when the analysis is sparse and flow-insensitive. However this flow-sensitive information is still feasible for SSI-based analysis, since SSI encodes flow-sensitivity in its virtual register renaming scheme. Hence the sparse analysis achieves the same results results as the classical analysis [KDM03] on the example program.

One notable weakness of dynamic type constraints is that they fail to detect *type errors* in programs. This style of type inference assumes that the programmer is always right. So, if a virtual register v is used with type t, then this is always treated as a bona fide fact in the analysis. However, it may be possible to spot anomalies once the types have been inferred, for instance if x has type i everywhere except for one place where it has a different type. Normally type errors are more subtle than this. Aycock [Ayc00] studies Python programs, which are allowed to have dynamic type constraints. He concludes that most programs do not make use of the dynamic typing features of Python. Richards provides anecdotal evidence that this is also true for typeless BCPL programs [RWS79]. If type inference is performed, most BCPL variables hold values of a single type throughout their entire scope.

SSA could have been used as the IR for the sparse bidirectional analysis instead of SSI, however the results would be less accurate. SSA does not split live ranges at conditional branch points, unlike SSI. The data flow equations for SSA-based sparse type inference would be the same as in Figure 4.9, except there would be no *(sigma)* rule.

A key observation made by Khedker et al [KDM03] is that

> Data flow analysis refers to forms of program analysis with no auxiliary store; each node in the program has an attribute. The space required by these attributes is usually tightly bounded by the program whereas the auxiliary store in constraint-based analyses is not tightly bounded.

The sparse data flow analysis presented in Section 4.5.2 also requires no auxiliary store. Each SSI virtual register in the program (rather than each program point) has an attribute. The space required by these attributes is generally smaller for sparse analysis than for classical analysis, since it is only necessary to store a single lattice element for each virtual register, rather than one lattice element per virtual register at each program point. So the classical type inference store is bounded by $O(NV)$, where N measures the number of nodes and V measure the number of virtual registers. The sparse type inference store is bounded by $O(V)$, but the number of SSI virtual registers is generally larger than the number of CFG virtual registers. It is possible to simplify these space complexity metrics to a single variable. In general, V grows as $O(N)$ in CFG programs. For SSI programs, V grows as $O(N)$ on average, but as $O(N^3)$ in the worst case. Thus, the classical type inference store is bounded by $O(N^2)$. The sparse type inference store is bounded by $O(N)$ on average, but by $O(N^3)$ in the worst case.

Data flow information is never killed in sparse data flow analysis, unlike classical data flow analysis. Sparse data flow information has to hold for the entire program, so information must only be generated if it holds globally, rather than locally. Classical data flow information is killed when it is true for some part of the program, but not another part. These local, flow-sensitive effects cannot be modelled by sparse data flow analysis. However they do not need to be, since SSI virtual register renaming effectively encodes flow-sensitivity in the virtual register names [HH98]. Section 4.6.1 discusses this issue at length.

4.5.6 Related Work

Khedker et al [KDM03] build upon much earlier work for classical DFA-based type inference. Aho et al [ASU86] treat type inference as a bidirectional data flow problem. They explicitly state that it requires both forward and backward propagation of information to obtain precise estimates of possible types. Their treatment is based on the work of Tennenbaum [Ten74] for the SETL programming language. Kaplan and Ullman [KU80] present another early example of type inference using DFA. This chapter shows that sparse analysis can achieve the same results for type inference as classical analysis.

A restricted form of type inference is employed for object-oriented languages in order to compute sets of possible concrete classes for variables. This type information enables the replacement of virtual method calls by direct calls, and the elimination of runtime type checks. Chambers et al [CDG96] describe intraprocedural class analysis, which is a standard forward data flow analysis. Diwan et al [DMM01] describe intraprocedural type propagation, which is similar to reaching definitions analysis. Bidirectional data flow analysis is not necessary to solve the problem of concrete type inference in object-oriented languages. Definitions are the only statements that can determine the concrete type. Uses cannot affect the type of variables, unlike in the example analysis presented in Section 4.5.2. So object-oriented type information only flows in the forward direction.

There has been little previous work on type inference for SSA and similar IRs. Mycroft [Myc99] shows how to perform type inference on register transfer language (RTL) code in SSA, for decompilation from assembler programs into C. However he uses constraint-based type inference. Lenart et al [LSG00] describe a system that combines constant propagation and type inference using SSA-based analysis. Their system calculates concrete types for

Java programs, so the type inference is simple forward analysis rather than bidirectional. They argue that combined analyses are more efficient and synergistic.

4.6 Systematic Renaming

This section argues that if virtual registers are renamed at program points where data flow information may change, then sparse analysis can be as accurate as classical analysis.

4.6.1 Achieving Maximum Accuracy

The empirical evidence from the constant propagation study in Section 4.3.4 shows that a greater degree of live range splitting leads to increased accuracy of results. This increased accuracy is definitely not due to any increase in power of expressivity of the underlying IRs. Recall that each IR is a *general sparse IR* [Ruf95b], so the program semantics is fully represented in each case. The difference in accuracy lies only in the fact that the sparse constant propagation analysis associates data flow information directly with each live range. So a smaller live range allow more specialized information to be stored with that live range.

This section argues that, for each data flow analysis, there is a certain amount of live range splitting, a *saturation point*, beyond which further live range splitting cannot increase the accuracy of the analysis. We assume that maximum accuracy is reached when a sparse data flow analysis computes the same data flow values for a live range of virtual register v as classical data flow analysis computes for v at each program point within that live range.

One obvious question arises from Section 4.3.4: Is SSI the necessary amount of live range splitting required for maximum accuracy constant propagation? Is it possible to achieve more precise constant information by performing a greater degree of live range splitting than SSI? Of course, such questions do not only depend on the IR used, but also on the lattice of data flow facts and the space of transfer functions. It is clear that analysis accuracy is a property of the data flow analysis as a whole, rather than just of one particular component.

At this point, it is useful to deal with two potential misconceptions.

1. This section does not attempt to characterize the *style* of data flow analysis used to obtain the data flow information. For instance, many researchers are interested in the relationship between flow-sensitive and flow-insensitive analysis, and how live range splitting improves the accuracy of flow-insensitive analysis [HH98]. Instead this section concentrates entirely on the manner in which the data flow information is stored, once it has been computed. We say that *sparse* analysis associates data flow information with each virtual register, whereas *classical* analysis associates information with each virtual register at each program point (recall Figure 4.1). It is true to say that flow-insensitive analysis generally computes sparse data flow information, but that is because there is no concept of program point in flow-insensitive analysis.

2. This section limits the *scope* of data flow analyses to those which are based on properties of virtual registers, rather than properties of expressions. Note that

there are several ways in which SSA-style analyses may be extended to handle expression-based data flow information [JP93, KCL+99].

4.6.2 Maximum Accuracy Property

In order to devise a formal definition of maximum accuracy, some supporting concepts must be introduced. We assume that sparse analysis is powerful as classical analysis, in that it uses the same data flow lattice and the same space of transfer functions. The only restriction we impose on sparse analysis is that it must store data flow information with live ranges, rather than at specific program points. We assume that data flow information, once computed, is stored in an information array, *info*. Recall Figure 4.1. In classical analysis, *info* is a 2-dimensional array subscripted by virtual register name and by program point. In sparse analysis, $info'$ is a 1-dimensional array subscripted by virtual register name only. (This assumes a 1-to-1 mapping between virtual register names and live ranges in the sparse IR, which is the case for the three IRs in this chapter.)

Now, for maximum accuracy for a given analysis, the computed information for a single live range in the sparse IR must be the same as the classical information for the corresponding virtual register at all points in that live range. Assume $liverange(x)$ is the set of program points within the live range of virtual register x in the sparse IR.

Then, the maximum accuracy property may be stated as:

$$\forall v. \forall n \in liverange(v). info'(v) = info(n, v)$$

4.6.3 Live Range Splitting Limit

This section proposes an upper limit on the degree of live range splitting required to achieve maximum accuracy with the sparse technique, for an abstract data flow analysis AB.

The classical data flow equation for AB is:

$$\mathsf{AB}(n) \quad = \quad \bigsqcup_{s \in flow(n)} \mathsf{AB}(s) \setminus \mathrm{kill}(n) \cup \mathrm{gen}(n)$$

$flow(n)$ is an abstraction for the set of n's data flow predecessors, which are control flow predecessors in a forward analysis, or control flow successors in a backward analysis, or both control flow predecessors and successors in a bidirectional analysis.

In order to achieve maximum accuracy with the sparse technique, new live ranges must be created at each point where the data flow information may change. In general, data flow information changes at gen, kill, and data flow merge and split points.

For analyses where the data flow information deals with virtual registers (a virtual register-based analysis), it is possible to specialize the above equation to V equations for the V virtual registers in the program:

$$\mathsf{AB}(n, v) \quad = \quad \bigsqcup_{s \in flow(n)} \mathsf{AB}(s, v) \setminus \mathrm{kill}(n, v) \cup \mathrm{gen}(n, v)$$

Now, live range splitting to ensure maximum accuracy necessitates renaming v at node n when

1. kill$(n, v) \neq \{\}$, or
2. gen$(n, v) \neq \{\}$, or
3. $|flow(n)| > 1$, and the different data flow predecessors may have different data flow values for v.

The third property is the most difficult to determine. The most pessimistic approach is to rename v whenever $|flow(n)| > 1$, but this introduces far too many names. This is equivalent to Appel's 'really crude' SSA construction algorithm [App98b], which inserts ϕ-functions for every virtual register at every control flow merge point. Aycock and Horspool begin with the same pessimistic assumption in their SSA construction algorithm [AH00]. A better approach is to rename v whenever $|flow(n)| > 1$ and the sets of reaching gens and kills with respect to virtual register v may be different for different elements of flow(n). This is equivalent to the dominance frontier approach to minimal SSA construction by Cytron et al [CFR+91].

Now, a general sparse IR that does this amount of renaming (and insertion of pseudo-definitions as required to preserve value flow through program) is, by construction, able to achieve maximum accuracy for the AB analysis, since a new virtual register name is introduced at every program point where the data flow information for that virtual register may change. Note that we have provided an upper limit on the amount of live range splitting required. It may be possible to achieve the same level of accuracy with less live range splitting, but it certainly will not be possible to achieve improved accuracy with more live range splitting!

4.6.4 Matching IRs to Analyses

This section identifies different IRs that are perfectly suited to different data flow analyses, that is, the IRs perform sufficient live range splitting in order to achieve maximum accuracy for a particular analysis. SSA is a suitable IR for forward data flow analyses where information is generated and killed by definitions, but not generated or killed by uses. Simple constant propagation [WZ91] and value range propagation [Pat95] are examples of appropriate analyses. SSI is a suitable analysis for predicated constant propagation [Ana99]. All these analyses are forward, but SSI can handle backward and bidirectional analyses too [Ana99]. For instance, SSI can handle information that is generated by uses, and killed by definitions.

However, the underlying message is clear: it is not always necessary to perform sufficient live range splitting in order to achieve maximum accuracy. The constant propagation results from Section 4.3.4 show that analysis accuracy gracefully degrades as the amount of live range splitting decreases. The WEB results are almost as good as the SSA results, and not much worse than the SSI results.

The limit to virtual register renaming is imposing a new name for each virtual register at each program point. Sparse data flow analysis of this IR is equivalent to classical data flow analysis, since information for each virtual register is stored at each program point, in both paradigms.

4.7 Concluding Remarks

This chapter has clearly demonstrated that SSI enables accurate and efficient analysis; for forward, backward and bidirectional data flow problems.

A key theme of this dissertation is that SSA and SSI are interchangeable for sparse analysis. However, SSA does not enable as accurate results as SSI. The three example data flow analyses confirmed this trend. There is a trade-off between the efficiency and the accuracy of analysis. This is due to the degree of live range splitting enforced by each IR, which interacts with the intrinsic properties of each particular analysis.

Of course, there are many more IRs in the literature that belong to the same family of related VRRSs as WEB, SSA and SSI. This chapter has only selected these three, because they are fairly common and are suitable for constant propagation analysis. Chapter 7 contains details about other IRs. It reconsiders the relationship between virtual register renaming and analysis accuracy.

The idea that underlies sparse evaluation graphs [CCF91] and general reference chains [Sto95] is the same:

> Given a data flow analysis, it is possible to construct an IR suited to a sparse implementation of that analysis.

Section 4.6.1 appears to advocate exactly this approach. However, the philosophy of the dissertation as a whole is different:

> Given a sparse IR, determine which data flow analyses can be performed with appropriate accuracy and efficiency on that IR.

The idea is that a single IR can be used for many analyses, rather than constantly transforming a program from one IR to another as the program undergoes different analyses.

There is a great deal of current interest in how live range splitting can improve the accuracy of flow-insensitive analysis. However, the only published work in this area [HH98] lacks both theory and empirical results. It describes an algorithm for flow-insensitive points-to analysis, where the accuracy may be improved by conversion to SSA. The authors abandon as an 'open question' the issue of whether flow-insensitive analysis can obtain results as accurate as flow-sensitive analysis. In a sense, this chapter has approached the same problem from a different angle. It has focused on the way the data flow information is *stored*, rather than the way the data flow information is *computed*. The claim is that sparse data flow analysis can be as accurate as classical data flow analysis, given a suitable virtual register renaming transformation on the subject program.

There are two areas for future research.

1. Instantiate the abstract AB analysis with other concrete data flow problems. Collect suitable empirical data for different sparse IRs, as in the case of constant propagation.

2. Formally prove that the same level of accuracy can be achieved by sparse analysis of an appropriate IR as by classical analysis. Apply similar reasoning in order to prove that the same level of accuracy can be achieved by flow-insensitive analysis of an appropriate IR as by flow-sensitive analysis.

The next chapter deals with program slicing, which is a different kind of analysis based on the notion of dependence. Once again, accurate and efficient analysis is possible with SSI.

Chapter 5

Slicing

SSI enables program slicing. Given a small amount of precomputation, SSI slicing queries can be as accurate as, and more efficient than, CFG slicing queries. SSI slicing has the same levels of accuracy and efficiency as program dependence graph slicing.

5.1 About this Chapter

5.1.1 Objectives

The previous chapter showed that data flow analysis can be accomplished much more efficiently by representing programs in SSI rather than the classical CFG. The current chapter provides more evidence to support this significant claim.

Classical CFG data flow analysis generally involves iteration until a fix point is reached. This chapter aims to show that SSI analysis can factor out much of the iterative process from data flow analysis phases by removing the iteration to the initial SSI construction phase. Iterative data flow equations should be computed when SSI is initially constructed. This information is then implicitly encoded into the SSI virtual register renaming scheme. Thus subsequent SSI analysis can reuse the implicit information, rather than having to recompute it many times. This chapter aims to demonstrate the superior efficiency of such SSI analysis over CFG analysis by a case study of *program slicing*, which is a kind of virtual register dependence analysis.

Another aim of this chapter is to explore further the similarities and differences between SSI and other IRs that attempt to encapsulate virtual register dependence. This chapter compares SSI with both program dependence graph (PDG) and dependence flow graph (DFG) IRs, in the context of program slicing.

5.1.2 Outline

Section 5.2 reviews the main concepts of program slicing. It focuses on the most common variant of slicing, for which this chapter develops a SSI analysis. Section 5.3 briefly discusses the different IRs that admit slicing, and their relevant properties. Section 5.4 describes the original CFG slicing algorithm in detail. Section 5.5 describes the new SSI slicing algorithm in detail. The performance of the two algorithms is studied in Section 5.6. Section 5.7 reviews related work in the field, comparing other algorithms with the

SSI slicing algorithm. This uncovers two difficulties with the SSI slicing algorithm. The problems and their solutions are described in Section 5.8. Section 5.9 highlights future areas of development, that would apply the same SSI slicing techniques to other variants of slicing. Finally Section 5.10 concludes.

5.1.3 Contributions

This chapter makes two key contributions.

1. Section 5.5 presents a novel algorithm for slicing SSI programs.
2. The SSI slicing algorithm is compared with CFG and PDG slicing in Sections 5.4 and 5.7.2 respectively. SSI is contrasted with DFG in Section 5.8.2. These comparative studies re-emphasize the advantages of SSI over existing alternative IRs.

5.2 What is Program Slicing?

Program slicing is now a standard program analysis technique. It was devised by Weiser [Wei81] in the late 1970's. His original definition of a slice is

> A slice s of program p is a subset of the statements of p that retains some specified behaviour of p. The desired behaviour is detailed by means of a slicing criterion c. Generally, a slicing criterion c is a set of virtual registers V and a program point l. When the slice s is executed, it must always have the same values as program p for the virtual registers in V at point l.

Program slicing has many applications. It is relevant in any area of software engineering that requires the automatic extraction of reduced programs, focusing only on one particular aspect of the whole computation. The original application area was debugging. Weiser showed that humans mentally apply slicing techniques when finding bugs in programs [Wei82]. This led to his first automatic slicing system. It is easier to identify a bug in a reduced program that only contains the code contributing to the defect. Other disciplines that benefit from slicing include:

- Program comprehension, and
- Automatic parallelization, and
- Component extraction, and
- Software metrics.

Virtual register dependence is the fundamental concept that underlies program slicing. There are two different kinds of dependence, *control* and *data* dependence. Informally, control dependence is concerned with which instructions can cause other instructions to be executed (flow of control through the program), and data dependence is concerned with how values are transferred from one instruction to another (flow of data through the program). Section 2.6 briefly mentioned these issues. A more formal discussion appears in Section 5.4.

```
int i = 0;
int limit = 10;
int sum = 0;
int product = 1;

while (i < limit) {
  i = i + 1;
  sum = sum + i;
  product = product * i;
}
```

```
int i = 0;
int limit = 10;
int sum = 0;


while (i < limit) {
  i = i + 1;
  sum = sum + i;

}
```

Figure 5.1: Example C program (left) and its slice (right)

Figure 5.1 shows a simple C procedure and its slice. The sliced version only retains the program statements that contribute to the final value of variable *sum*. This example C program is used as a subject program to CFG and SSI slicing algorithms later on in this chapter.

The field of program slicing has expanded greatly since Weiser's original formulation. The original slicing style is now known as static, backward slicing, producing executable, syntax-preserving slices. The rest of this section classifies the different parameters that may be altered in the slicing discipline. These are all orthogonal, so they may be combined in any way to produce a distinct slicing variant.

directionality: A *backward* slicing algorithm uses backward data flow analysis to compute program statements that, if executed, may affect the values of the virtual registers in the slicing criterion. In contrast, a *forward* slicing algorithm uses forward data flow analysis to compute program statements that may be affected if the values of virtual registers in the slicing criterion are altered.

knowledge of inputs: A *static* slicing algorithm has no knowledge of the runtime environment of the program. Static slices hold true for all runs of the program. In contrast, a *dynamic* slicing algorithm takes into account some knowledge about the values of input virtual registers to the program being analysed. This means that a dynamic slice is specialized for a particular set of input values, a particular run of the program. It enables the slice to be smaller, using partial evaluation to simplify the dependence information. *Conditioned* slicing is similar to dynamic slicing. It has some knowledge about the relationship between input virtual registers. This symbolic information also allows the simplification of dependence information by partial evaluation.

executability: An *executable* slice is a complete program in itself, so it can be compiled and executed. In contrast, a *closure* slice is just a program fragment, that may not be a syntactically valid program.

relation to original program: A *syntax-preserving* slice is a subset of the original program. It is a transformation of the original program that is obtained merely by deleting statements from the original program. In contrast, an *amorphous* slice does not necessarily preserve the syntax of the original program. Amorphous slicing algorithms may apply other program transformations as well as statement deletion. However an amorphous slice must be semantically equivalent to a syntax-preserving slice for the same slicing criterion.

There are numerous comprehensive surveys and literature reviews relating to program slicing. Tip [Tip95] provides the earliest and clearest such survey. His work is extremely comprehensive, but does not cover the more recent developments (conditioned and amorphous slicing are notably absent from his paper). A later overview of slicing [HH01] deals with both conditioned and amorphous slicing. Another fairly comprehensive recent work [MGM02] reviews the basics and then moves onto concurrent, distributed and object-oriented program slicing. Hoffner [Hof95] provides an early evaluation of different slicing algorithm implementations. His work is useful for providing a classification scheme for different slicing algorithms. (The classification scheme given above is an extension of his work.)

Tip [Tip95] outlines potential complications that arise from high-level constructs such as procedures, pointers, composite datatypes, and concurrency. Chapter 7 discusses the addition of such features to SSI. However this chapter concentrates on simple intraprocedural analysis with scalar virtual register values.

5.3 Intermediate Representations for Slicing

Slices are usually presented in terms of high-level source code. However, the actual slicing operations are performed at a lower level than this. Generally, a source program is transformed into a standard compiler IR to be sliced. Once the low-level slice has been computed, it is used to determine which statements from the source code should comprise the final slice,

The standard IRs for slicing are CFG and PDG, both introduced in Chapter 2. CFG slicing algorithms are usually classical data flow analyses, whereas PDG slicing algorithms are graph-reachability problems. Generally, PDG slicing is much more efficient than CFG slicing. This chapter shows how the CFG-like SSI IR gives much better slicing efficiency than the CFG. In fact, the time complexity is as good as for PDG slicing. This chapter formally relates SSI and PDG IRs.

5.4 CFG Slicing

The discipline of program slicing was originally proposed by Weiser. The earliest available description of this CFG slicing algorithm [Wei81] gives data flow equations and informal descriptions. However, the clearest exposition (with better explanations and superior typesetting) of Weiser's CFG slicing algorithm is presented in Tip's survey [Tip95]. This section outlines the CFG slicing algorithm, based on Tip's description.

Slices are computed by solving sets of data flow equations derived from the input CFG. These equations are solved using the classical iterative technique. The inputs to the slicing algorithm are the CFG of the program to be sliced, p_{CFG} and the slicing criterion $C \equiv \langle V, n\rangle$, where V is a subset of the virtual registers mentioned in p_{CFG} and n is a distinguished node in p_{CFG}.

The slicing algorithm makes use of two notions of virtual register dependence, *data* and *control* dependence. Node m is data dependent on node m' if:

1. virtual register v is defined at m', and

2. v is used at m, and

3. definition of v at m' reaches m.

Control dependence is a more abstract concept. Intuitively, node m is control dependent on node m' if there is a conditional branch at m' that determines whether or not m is to be executed. Formally, node m is control dependent on node m' if:

1. there exists a control flow path from m' to m such that for all intermediate nodes m_i along the path, m_i is postdominated by m, and

2. m' is not postdominated by m.

Weiser's slicing algorithm incrementally computes the set of *relevant virtual registers* at the entry to each node in p_{CFG}, together with the set of *relevant statements*. Informally, a virtual register v is relevant immediately prior to node m if alteration to the value of v at m may affect the values of virtual registers in set V at node n, where $\langle V, n\rangle$ is the slicing criterion. Relevant statements are those that define relevant virtual registers. Branch statements may also be relevant. A branch statement b is relevant if another relevant statement is control dependent on b. The program slice is the set of all relevant statements.

Weiser's algorithm actually operates as follows. First the *directly relevant* virtual registers for each node must be computed. Direct relevance is determined entirely by data dependence, and does not take control dependence into account at all. The set of directly relevant virtual registers at node i is:

$$R_C^0(i) = \begin{cases} V & \text{if } i = n \\ V' & \text{where } v \in V' \text{ if } \quad (j \in \mathit{succ}(i) \wedge v \in R_C^0(j) \wedge v \notin \mathit{def}(i)) \\ & \qquad\qquad\qquad\quad \vee \\ & \qquad\qquad\qquad\quad (j \in \mathit{succ}(i) \wedge v \in \mathit{ref}(i) \wedge \mathit{def}(i) \cap R_C^0(j) \neq \{\}) \end{cases}$$

where:

- $\mathit{def}(m)$ is the set of virtual registers defined at node m, and

- $\mathit{ref}(m)$ is the set of virtual registers used at node m, and

- $\mathit{succ}(m)$ is the set of successor nodes to node m.

The first line of the definition ensures that virtual registers in the slicing criterion are directly relevant at the slicing criterion point n. The second line ensures that virtual registers are relevant at node i if they are relevant at j, a successor to i, and are not defined at i. This is backward propagation of data dependence. The last line ensures that a virtual register v is relevant at node i if i defines a virtual register that is relevant at j, a successor to i, and i uses v in this definition. This backward transitivity encapsulates the actual flow of data through virtual registers.

This direct relevance computation may be reformulated as a classical CFG data flow analysis in terms of monotone data flow equations.

$$\begin{aligned} R_{\text{in}}(i) &= R_{\text{out}}(i) \setminus \mathit{kill}(i) \cup \mathit{gen}(i) \\ R_{\text{out}}(i) &= \bigcup_{j \in \mathit{succ}(i)} R_{\text{in}}(j) \end{aligned}$$

where

$$\begin{aligned} \mathit{kill}(i) &= \mathit{def}(i) \\ \mathit{gen}(i) &= \begin{cases} \mathit{ref}(i) & \text{if } (\mathit{def}(i) \cap R_{\text{out}}(i) \neq \{\}) \wedge i \neq n \\ \mathit{ref}(i) \cup V & \text{if } (\mathit{def}(i) \cap R_{\text{out}}(i) \neq \{\}) \wedge i = n \\ V & \text{if } (\mathit{def}(i) \cap R_{\text{out}}(i) = \{\}) \wedge i = n \\ \{\} & \text{otherwise} \end{cases} \end{aligned}$$

This is slightly different to a classical data flow analysis, such as liveness analysis, due to the non-constant nature of the *gen* sets. In slicing, *gen* sets must be recomputed at each iteration step, rather than precomputed and memo-ized. Nevertheless, standard CFG data flow iterative analysis can be performed to solve the direct relevance equations. The worst-case time complexity for such calculations is $O(N^2)$ for CFGs with N nodes.

The set of directly relevant statements, S_C^0, may be derived immediately from the information about directly relevant virtual registers.

$$S_C^0 \equiv \{i \mid j \in \mathit{succ}(i) \wedge (\mathit{def}(i) \cap R_C^0(j) \neq \{\})\}$$

S_C^0 is the set of all nodes n that define a virtual register that is directly relevant at a successor to n. This set can be calculated by a single linear pass through all the CFG nodes.

If the input program consists entirely of straight line code with no conditional branch statements, then only directly relevant statements are in the slice. Data dependence is sufficient, since there is no control dependence to take into consideration. However, with conditional branch statements, it is necessary to consider control dependence as well. This is referred to by Weiser as *indirect relevance*.

The next step is to compute the set of indirectly relevant branch statements B_C^k. A branch statement b is in the set B_C^k if a relevant statement $s \in S_C^k$ is control dependent on b. The range of influence of a branch statement b, $\mathit{infl}(b)$, is defined as the set of nodes that are control dependent on b. Thus the set of relevant branch statements is defined as:

$$B_C^k \equiv \{b \mid i \in S_C^k, i \in \mathit{infl}(b)\}$$

This set may be computed in a linear pass through the CFG, provided that the control dependence relation for each node has been precomputed. Now the set of indirectly relevant virtual registers, R_C^{k+1}, is determined by considering the virtual registers referenced in the predicates of the indirectly relevant conditional branch statements to be relevant.

$$R_C^{k+1}(i) \equiv R_C^k(i) \cup \bigcup_{b \in B_C^k} R^0_{\langle b, ref(b)\rangle}(i)$$

This is potentially an expensive computation. For each branch statement in B_C^k, it is necessary to do an iterative direct relevance computation over the whole CFG. Recall that a direct relevance computation takes $O(N^2)$ time. The worst case size of B_C^k is $O(N)$, so computation of $R_C^{k+1}(i)$ could take $O(N^3)$ time.

Now, the set of indirectly relevant statements S_C^{k+1} consist of the nodes in B_C^k together with the nodes i that define relevant virtual registers in R_C^{k+1}.

$$S_C^{k+1} \equiv B_C^k \cup \{i \mid def(i) \cap R_C^{k+1}(j) \neq \phi, j \in succ(i)\}$$

The sets R_C^{k+1} and S_C^{k+1} are monotonically increasing subsets of the input program's virtual registers and statements respectively. The fixpoint of the computation of S_C^{k+1} constitutes the desired program slice. The worst case time complexity of this slicing algorithm is $O(N^4)$ for input CFGs with N nodes, although $O(N^2)$ is more likely in practice. Note that Weiser [Wei81] gives the worst case as $O(NE \log E)$ where E (CFG edge count) can grow as $O(N^2)$.

Example

This section considers slicing the simple example program shown in Figure 5.2. This program computes both the sum and the product of the integers 1 to 10 inclusive. The program is shown as a CFG of maximal size basic blocks. However, Weiser's CFG slicing algorithm treats each labelled statement to be a distinct node. The slicing criterion C is $\langle\{\texttt{product}\}, 8\rangle$. Thus only the calculation of the `product` virtual register is deemed relevant. The orthogonal and concurrent computation of the `sum` virtual register is irrelevant for the slice.

The first step is to compute the set of directly relevant virtual registers, R_C^0. This computation iterates over the program, using the classical data flow equations given above. Figure 5.3 shows the fix point that is reached. The directly relevant statements, S_C^0, may be computed from R_C^0. S_C^0 contains the definitions of `i` at nodes 1 and 6, and the definitions of `product` at nodes 4 and 8, i.e. $S_C^0 = \{1, 4, 6, 8\}$.

The next step is to determine the indirectly relevant branch statements. Nodes 1 and 4 are not control dependent on any branch statements. However, nodes 6 and 8 are both control dependent on the branch statement at node 5. Thus $B_C^0 = \{5\}$.

In order to compute the set of relevant virtual registers R_C^1, it is necessary to compute the indirectly relevant virtual registers from the branch statement, $R^0_{\langle\{\texttt{i,limit}\},5\rangle}$. This relevance set is given in Figure 5.4.

The set of indirectly relevant virtual registers R_C^1 is now obtained by the union of R_C^0 and $R^0_{\langle\{\texttt{i,limit}\},5\rangle}$. Figure 5.5. shows this relevance set. In fact, this is the final relevance set computation. Further computation confirms that this is the fix point for the computation. The set of relevant statements S_C^1 is $\{1, 2, 4, 5, 6, 8\}$. (There are no further control dependences, so we have reached the fix point.) This is the slice of the program for criterion C, depicted graphically in Figure 5.6.

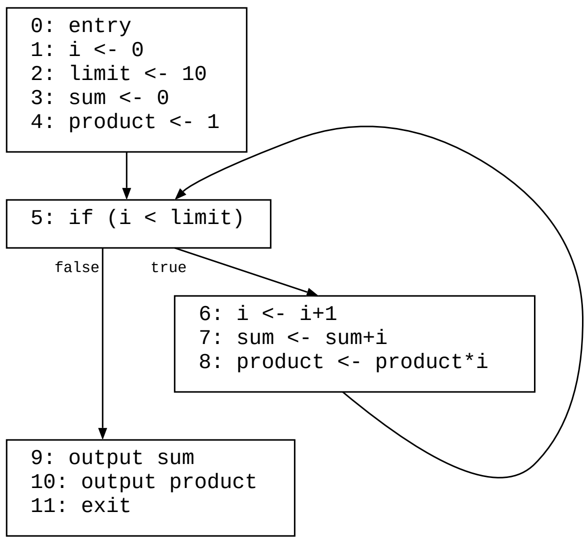


Figure 5.2: Example CFG program to be sliced

node	relevant vars
0	
1	
2	`i`
3	`i`
4	`i`
5	`i`, `product`
6	`i`, `product`
7	`i`, `product`
8	`i`, `product`
9	
10	
11	

Figure 5.3: Calculation of $R^0_{\langle\{\texttt{product}\},8\rangle}$

node	relevant vars
0	
1	
2	`i`
3	`i`, `limit`
4	`i`, `limit`
5	`i`, `limit`
6	`i`, `limit`
7	`i`, `limit`
8	`i`, `limit`
9	
10	
11	

Figure 5.4: Calculation of $R^0_{\langle\{\texttt{i},\texttt{limit}\},5\rangle}$

node	relevant vars
0	
1	
2	`i`
3	`i`, `limit`
4	`i`, `limit`
5	`i`, `limit`, `product`
6	`i`, `limit`, `product`
7	`i`, `limit`, `product`
8	`i`, `limit`, `product`
9	
10	
11	

Figure 5.5: Calculation of $R^1_{\langle\{\texttt{product}\},8\rangle}$

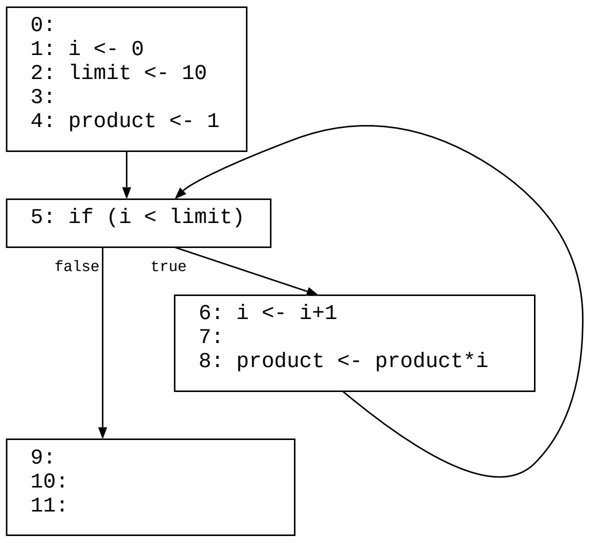


Figure 5.6: Slice of example CFG program

5.5 SSI Slicing

For Weiser's CFG slicing algorithm, the slicing criterion is specified as a set of virtual registers V and a program point n, for program p_{CFG}. In the new SSI slicing algorithm, the slicing criterion is merely a set of virtual registers V, for program p_{SSI}. There is no longer any need to specify a program point, since SSI virtual register dependence does not vary with program point. Indeed, the SSI slicing criterion can be more flexible, since the set of virtual registers can come from any program points, whereas Weiser restricted the virtual register set to a single point. This is a key property of IRs like SSA and SSI:

> Virtual register renaming transforms flow-sensitive properties (for which a program point must be specified for the property to be valid) into flow-insensitive properties (for which no program point is necessary).

In effect, the virtual register renaming scheme gives a handle on a point, or a small set of contiguous points, over which a property holds. Hasti and Horwitz are the first to notice this powerful feature of SSA-like IRs [HH98]. They show that expressing a program in SSA can transform certain flow-sensitive data flow properties into flow-insensitive properties.

There are the two ways in which a virtual register can belong to a virtual register dependence set, data dependence and control dependence. SSI permits trivial calculation of data dependence, since each virtual register has a unique definition site. Any IR that has the single-assignment property, such as SSA, also allows data dependence calculations to be performed cheaply. To find the virtual register dependence set for v_{ssi}, one simply has to locate the unique definition of v_{ssi}, add all the virtual registers referenced in this definition to the virtual register dependence set, then trace back transitively all their definitions, adding referenced virtual registers to the virtual register dependence set. This gives us the set of directly relevant virtual registers, to use Weiser's parlance, in a single linear pass through the program.

However, SSI also simplifies calculation of control dependence. (SSA does not do this!) A data dependence on a virtual register defined by a σ-function indicates that there should

```
vardep($v$ : virtual register, $p$ : SSI program) =
    **if** $alreadyseen(v)$
        **return** $\{\}$
    **else if** $v \in inputvars(p)$
        **return** $\{v\}$
    **else return** $ref(defsite(v)) \cup (\bigcup_{x \in ref(defsite(v))} vardep(x, p))$
```

Figure 5.7: The virtual register dependence algorithm for SSI

also be a control dependence on the associated conditional branch instruction. This is because the conditional branch outcome determines which of the σ-function destination operands is assigned a value, depending on control flow. Thus whenever a σ-function source operand is added to the virtual register dependence set, the referenced virtual registers in the associated conditional branch predicate should also be added, since these virtual registers are indirectly relevant (due to control dependence). Once these indirectly relevant virtual registers are in the virtual register dependence set, their dependences must be traced backed as well.

Effectively, SSI transforms control dependence into data dependence by creating σ-functions for all virtual registers used in a conditional context, and treating these σ-functions as if they reference not only their source virtual registers, but also all the virtual registers referenced in the predicate of the appropriate conditional branch instruction. This is the huge advantage of using SSI instead of SSA. Although SSA also has the unique definition site feature, SSA does not represent control dependence in its virtual register renaming scheme. SSI incorporates control dependence in an elegant and efficient manner. (Note that SSI does not transform *all* control dependence into data dependence. Section 5.8 discusses this problem. For the moment, this issue is conveniently ignored.)

Once the virtual register dependence set has been fully computed (when there are no more statements to analyse, or no more outstanding dependences to trace) the SSI slicing algorithm has almost finished. A single sweep through the program marks as 'relevant' all instructions that define a virtual register in the computed dependence set. For relevant σ-function definitions, the associated conditional branch instructions should also be marked as relevant. Although this relevant statement marking could actually be done at the same time as the virtual register dependence calculation itself, this presentation leaves the marking till the end to keep algorithm simple.

Figure 5.7 gives the algorithm for the variable dependence computation. At a first inspection, it appears to be tracking data dependence only. However, the auxiliary function definitions in Figure 5.8 reveal that the *ref* function treats control dependence in exactly the same manner as data dependence.

Each SSI virtual register can only be considered once by the *vardep*() routine. Hence the time complexity is bounded by the number of SSI virtual registers in the program. Ananian [Ana99] claims that the number of SSI virtual registers grows linearly with the number of nodes in the program's original CFG. Hence the SSI virtual register dependence analysis algorithm is expected to exhibit linear behaviour. Note that this claim is based on empirical evidence. Theoretically, the worst case growth in the SSI virtual register

- *alreadyseen*(v) returns true if we have already placed virtual register v in the dependence set that is currently being built, otherwise false.
- *inputvars*(p) returns the set of program p's input virtual registers. These virtual registers are not defined within p, so we cannot get a handle on their definitions.
- *ref*(s) returns the set of virtual registers that may be referenced (used) when statement s is executed. If s is a σ-function, then *ref*(s) also includes all of the virtual registers referenced in the conditional test associated with this σ-function.
- *defsite*(v) returns the statement in program p that is the unique definition of virtual register v.

Figure 5.8: Auxiliary routines for the virtual register dependence algorithm

namespace is $O(N^3)$ for a program with N nodes, which would make the SSI algorithm as computationally expensive as Weiser's algorithm.

However, Section 5.6 reports on experiments that compare the two algorithms. The results tend to support the fact that the time complexity of the SSI slicing algorithm is effectively linear in program size, and Weiser's algorithm is superlinear.

Example

This section presents an example of the SSI slicing algorithm. It computes the same slice as the earlier CFG-based example. Figure 5.2. gives the original CFG program. Figure 5.9 shows the SSI version of this program. The slicing criterion is therefore {`product4`}. Recall that SSI slicing criteria do not need to specify a program point.

The virtual register dependence of `product4` is computed as follows. Its definition refers directly to `product2`, but this definition is a σ-function, therefore we must also include the virtual registers referenced in the conditional branch predicate, which are `i2` and `limit0`. Now we trace back these virtual registers to their unique definitions, and add the referenced virtual registers (if any) at each definition point. `product2` depends on `product1` and `product0`. `i2` depends on `i1` and `i0`. `i1` depends on `i4`. Once the the complete set of relevant virtual registers has been computed, we just extract all the definitions of these virtual registers from the original program. This is the sliced version of the program, shown in Figure 5.10.

5.6 Empirical Comparison

This section compares the performance of the two slicing algorithms described above. The algorithms were implemented in the Machine SUIF framework. The Weiser CFG slicing algorithm uses the standard Machine SUIF CFG representation. The SSI slicing algorithm uses the Machine SUIF SSI representation described in Chapter 3. Weiser's algorithm was implemented in 1400 lines of C code, whereas the new SSI algorithm only takes 1000 lines. This is already an indication that the SSI algorithm is conceptually simpler.

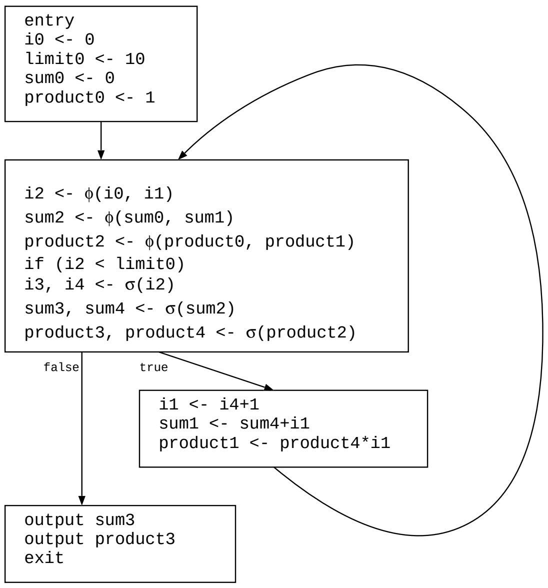


Figure 5.9: Example SSI program to be sliced

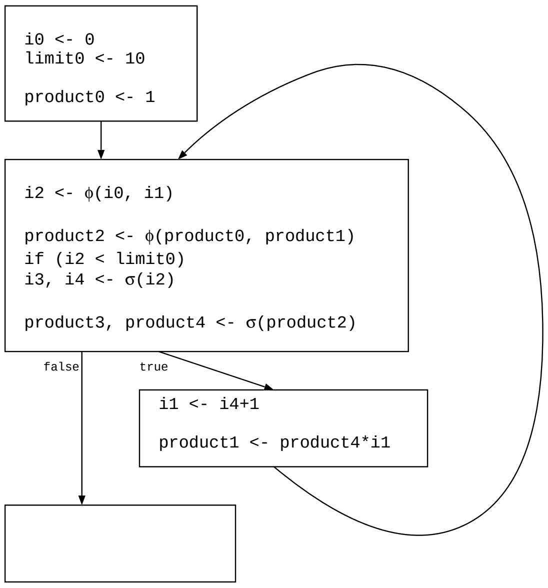


Figure 5.10: Slice of example SSI program

```
int f(int a, int b, int c) {
  a = b + c + 1;
  b = a + c + 1;
  c = b + a + 1;

  return c;
}
```

Figure 5.11: C source code for the datadep1 test

```
int f(int a, int b, int c) {
  if (a) {
    if (b) {
      if (c) {
        a = b + c + 1;
        b = a + c + 1;
        c = b + a + 1;
      }
    }
  }

  return c;
}
```

Figure 5.12: C source code for the ctrldep1 test

Two sets of tests were used to compare the slicing algorithms. The datadep test set features only data dependences, whereas the ctrldep test set features both control and data dependences.

The simplest datadep test, datadep1, is shown in Figure 5.11. datadep$N$ is derived from datadep1 by repeating the sequence of three assignment statements N times. We tested with $N \in \{1, 100, 200, 500\}$. This means that the final value of virtual register `c` depends on all previous assignments to `a`, `b` and `c`. Slices are always taken on the return value at the return statement.

The simplest ctrldep test, ctrldep1, is shown in Figure 5.12. ctrldep$N$ is derived from ctrldep1 by repeating the block of nested if statements N times. We tested with $N \in \{1, 10, 20, 50, 100, 200, 500\}$. This means that the final value of virtual register `c` depends on all previous assignments and conditional statements. Again, slices are always taken on the return value at the return statement.

The results are summarized in graph form. Figure 5.13 shows the times taken for the datadep test set. It can be seen that both CFG and SSI slicing have linear asymptotic time complexity, in relation to the size of the program being sliced. However, the SSI slicer is about 5 times more efficient than the CFG slicer, on simple data dependence cases. Figure 5.14 shows the times taken for the ctrldep test set. Actually, the SSI slicer times are multiplied by 100 in order to show them clearly on the graph. Again, it can be seen that the SSI slicer has linear time complexity in the size of the program. However, the CFG slicer has noticeably superlinear complexity. Recall from Section 5.4 that the worst case time complexity for CFG slicing is $O(M^4)$ for a CFG with M nodes. Measurements could not be obtained for the ctrldep500 test case, since it took too long to complete. Note that all times are the arithmetic mean of three tests on a lightly loaded machine (AMD Athlon 1.4GHz, x86 Linux).

Weiser's CFG slicing algorithm is remarkably inefficient. Each R^k and S^k equation recomputes slices from scratch. No information from previous solutions of R^k and S^k equations is reused. Some form of memo-ization may improve this inefficiency, however our implementation did not implement this. Basically, the CFG slicing algorithm is inefficient, since so much information has to be recomputed each time. No dependence information is encoded in the IR, unlike SSI. For instance, ctrldep100 makes over 2000 direct relevance computations for virtual registers at different points. Many of these are duplicated. Look at the times (in seconds) for ctrldep100 below:

algorithm	ctrldep100	datadep100
Weiser	260	0.1
SSI	0.1	0.02

It can be seen that slices which require control dependence information are much more expensive to compute with Weiser's algorithm than simple data dependence slicing. On the other hand, there is negligible difference between control dependence and data dependence with the SSI algorithm. The above table shows that Weiser's algorithm takes takes over 2000 times longer for ctrldep100 than for datadep100. On the other hand, the SSI algorithm only takes 5 times longer for ctrldep100 than for datadep100. The SSI increase in time is largely due to the increased number of virtual registers in the program, and the larger dependence set. Note that the comparatively small times for SSI are not at all to do with the fact that SSI sometimes computes smaller slices. In all these test cases, the slices computed by the two algorithms are identical.

The timings given above are only for the slice calculations. They do not include any computation time for IR construction. It may be argued that for the comparison between CFG slicing and SSI slicing to be fair, the SSI slicing times should include the time taken to construct SSI from CFG. However, this is not necessary for two reasons:

1. This dissertation advocates the use of SSI as the standard compiler IR. It is envisaged that a program will be in SSI for all analyses in the intermediate stages of the compilation cycle. Thus the construction time of SSI will be amortized over multiple analyses.

2. The actual SSI construction time for the most complicated test program (ctrldep500) actually only takes 7.2 seconds, which is insignificant in relation to the CFG slicing time for on this test.

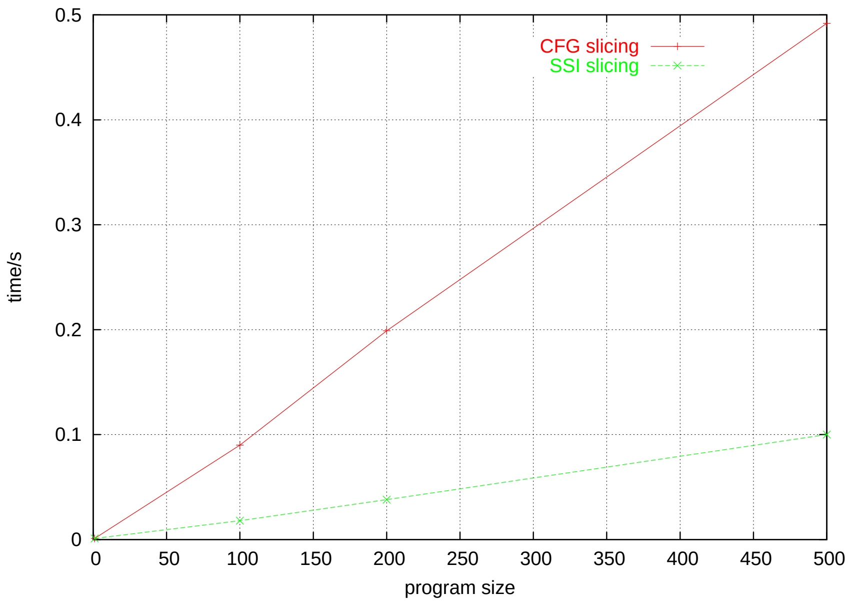


Figure 5.13: Slice times for data dependence test set

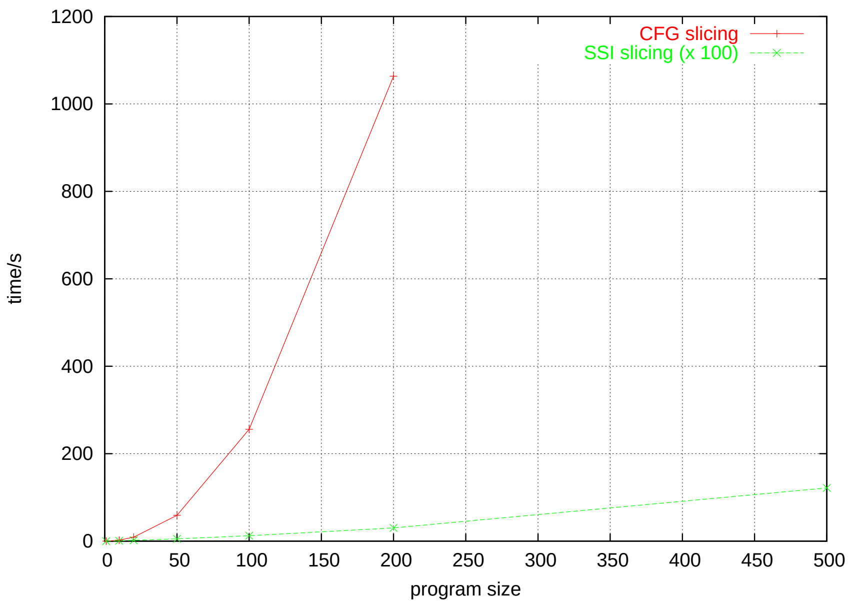


Figure 5.14: Slice times for control dependence test set

5.7 Related Work

5.7.1 Dead Code Elimination and Slicing

This section discusses SSA-based dead code elimination (DCE), and its similarities with the SSI slicing algorithm presented in Section 5.5. Dead code is also known as *unused* code or *unnecessary* code. (Note that dead code should not be confused with either redundant code or unreachable code. Section 4.3 incorporates unreachable code elimination into SSI-based data flow analysis.) Dead code comprises dead instructions, which compute values that are not used anywhere in the program. Such values may be stored in dead virtual registers. The opposite of a dead virtual register (or instruction) is a *live* virtual register (or instruction). Section 3.3.1 defines the property of liveness.

Classical DCE merely eliminates virtual register definitions that do not reach any uses, and do not have any other side-effects. On the other hand, *aggressive* DCE assumes an instruction is dead unless there is evidence that it contributes to the final result of the program. Appel [App98a] describes this aggressive DCE as a marking algorithm, where an instruction must be marked as live if it:

1. has a side-effect (e.g. writes to memory), returns from a function or calls another function that may have side-effects; or

2. defines some virtual register v that is used by another live instruction; or

3. is a conditional branch on which some other live instruction is control dependent.

The set of live instructions defined by these properties may be computed using a worklist algorithm. All unmarked instructions are dead, and may be removed from the program.

This aggressive DCE algorithm bears great resemblance to our SSI slicing algorithm in Section 5.5. The first reason for marking a live instruction corresponds to the slicing criterion. The second reason for marking a live instruction corresponds to a data dependence (Weiser's direct relevance). The third reason for marking a live instruction corresponds to a control dependence (Weiser's indirect relevance).

Of course, both DCE and slicing are easier to do on SSI than CFG, because of the implicit encoding of data dependence and control dependence in virtual register naming scheme. Note that the common SSA-based DCE algorithm (as presented by Cytron et al [CFR+91], Appel [App98a], and Cooper and Torczon [CT04]) requires the additional calculation of control dependences, since this information is not incorporated directly into SSA.

Ananian [Ana99] does not define this aggressive DCE algorithm for SSI. His 'unused code elimination' algorithm makes no mention of control dependence. Thus it can only be as powerful as classical DCE, which simply eliminates assignments to dead virtual registers, and cannot eliminate unnecessary conditional branches. Ananian never seems to recognize the implicit control dependence information encoded by σ-functions. Perhaps his caution is because SSI control dependence information is not always complete, but the techniques presented in Section 5.8 overcome this difficulty.

Liao et al [LDB+99] develop an SSA slicing algorithm. However, their algorithm inserts explicit control dependence edges. Data dependence is represented implicitly using SSA virtual register names. Control dependence is represented explicitly using control dependence edges. The slicing algorithm traces both types of dependence when calculating slices, similar to the PDG graph-reachability algorithm. However, this approach also requires the explicit computation and representation of control dependence.

5.7.2 PDG versus SSI

The program dependence graph (PDG) is by far the most common IR used for slicing algorithms. Section 2.6.2 introduces PDG in some detail. Ottenstein and Ottenstein [OO84] introduce the PDG-based version of slicing. They say that "explicit data and control dependence make the PDG ideal for constructing program slices." The slicing criterion is restated as a node (or nodes) in the PDG, and then slicing is reduced to a graph-reachability problem, to determine which nodes are reachable from the criterion node (or nodes) in the PDG with all edge directions reversed. These reachable nodes constitute the program slice. Edges may be either control dependence or data dependence edges. This slicing algorithm has linear time complexity in the size of the PDG, which should be proportional to the size of the original CFG.

Many other papers describe PDG slicing. The most notable extension to the original formulation is *interprocedural* PDG slicing using system dependence graphs (SDGs) [HRB90] This is the basis of the CodeSurfer [Gra04] system, which is a popular commercial slicing tool for industrial-size C programs.

PDG slicing effectively computes the transitive closure of the dependence relation for the slicing criterion node (or nodes). This is the same concept as the SSI slicing algorithm presented in Section 5.5. The SSI algorithm simply follows (backward with respect to control flow) the implicit data and control dependences through the program. This observation raises a fundamental question. *What exactly is the difference between PDG and SSI?* Both representations encode control dependence and data dependence information. PDG uses data dependence edges to encode not only data dependence, but also a partial ordering on program statements. This information is not required for slicing analyses. In contrast, SSI exactly encodes the right data dependence information for slicing. With regard to control dependence information, PDG exactly encodes the right amount. On the other hand, SSI encodes something slightly different to control dependence. SSI encodes virtual register usage in conditional contexts. This is often exactly equivalent to control dependence, but not always. Section 5.8 discusses how to represent control dependence precisely in SSI. The remainder of this section contrasts PDG and SSI notions of dependence in more detail.

Data Dependence

PDG encodes data dependence explicitly as data dependence edges between nodes. There are three different kinds of data dependence edges.

1. Flow dependence edges: from a virtual register definition to a use that is reached by that definition. This is a read-after-write dependence. This kind of dependence is familiar from the description of def-use chains in Section 2.6.1.

2. Output dependence edges: from a definition d_1 of virtual register v to another definition d_2 of v, where d_1 occurs before d_2 in terms of control flow. This is a

write-after-write dependence. d_1 must be executed before d_2 in order to preserve the semantics of the program.

3. Anti dependence edges: from a use of virtual register x to a definition of x, where the use occurs before the definition in terms of control flow. This is a write-after-read dependence. The use must be executed before the definition in order to preserve the semantics of the program.

Generally, only flow dependence edges are required for slicing. Only flow dependence needs to be traced back in order to compute the PDG nodes comprising the slice. Output and anti dependence edges do not represent true data dependence. Instead they encode a partial order on program statements, which is necessary since there is no explicit control flow relation between PDG nodes. However, PDG slices are normally mapped back to high-level source code, where control flow is explicitly represented. Thus there is no need for any such control flow information to be present in the computed PDG slice.

SSI encodes data dependence implicitly in the virtual register naming scheme for the program. Recall that SSI is a transformed CFG, with renamed virtual registers and pseudo-definitions at control flow split and merge points. There is effectively a dependence edge from the unique definition of a virtual register to all of the uses of that virtual register. This is equivalent to the PDG flow dependence edge. SSI has no equivalent of output or anti dependence edges. Since each SSI definition creates a fresh virtual register name, SSI does not suffer from the PDG problems of enforcing an order for multiple definitions of the same virtual register. SSI eliminates output and anti dependence edges by virtual register renaming. Note that SSI retains an explicit total control flow ordering on nodes, which is absent from the PDG.

The program dependence web [BMO90] and Click's simple graph-based IR [CP95] are PDG variants that enforce the SSA virtual register renaming convention. Thus these IRs only require flow dependence edges to represent data dependence.

Control Dependence

PDG encodes control dependence explicitly as control dependence edges between nodes. Top-level unconditionally executed code is control dependent on the `entry` node. Code in a conditionally executed node is control dependent on the predicate governing that node. Code within a loop body is control dependent on the predicate governing that loop. (The formal definition of control dependence was given in Section 5.4.) In effect, PDG encodes control dependence on a per-statement level, since each PDG node roughly corresponds to a high-level program statement.

SSI encodes control dependence implicitly by the use of σ-functions. Since dependence information is encoded directly as virtual register names, extra graph edges are not required. Every virtual register that is used in a conditional context but defined outside that conditional context requires a σ-function at the program point where the controlling predicate is evaluated. In effect, SSI encodes control dependence at the virtual register level. However, this virtual register dependence does not correspond exactly with control dependence. For example, consider the situation shown in Figure 5.15, where virtual register y_2 is either assigned constant 0 or 42, depending on condition x.

Note that although the value of y_2 at the end of the program is dependent on the value of condition virtual register x, yet because there are no σ-functions associated with `if`

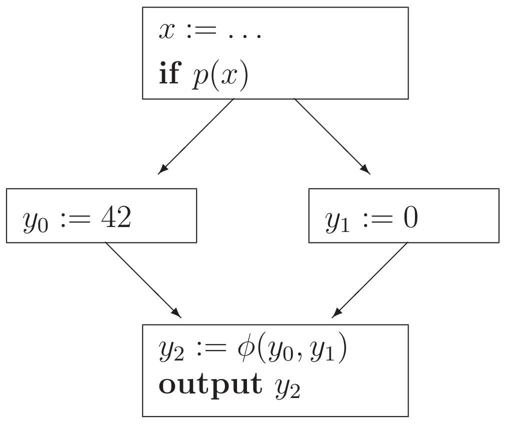


Figure 5.15: Example SSI program with missing control dependence

statement (since no virtual registers are used in either arm of the conditional branch) then this control dependence information escapes unnoticed in SSI. Whenever virtual registers are assigned constant values in conditional contexts, SSI does not automatically represent that control dependence information. The PDG version of this example program has control dependence edges from the predicate to the two conditionally executed statements. This may appear to be a significant flaw inherent in SSI, but Section 5.8.1 present several possible solutions.

The key point of this section is that *SSI is fundamentally different to PDG*. The key point of the following section is that with minor changes, *SSI can be used to model exact (PDG-like) control dependence.*

5.8 Some Problems with SSI Slicing

This section explores two problems with the SSI slicing algorithm presented in Section 5.5. Both problems relate to the issue of control dependence. In effect, SSI represents control dependence as data dependence, via σ-function pseudo-definitions. However σ-functions are only present for virtual registers that are used in a conditional context but defined outside that conditional context. There is not an exact correspondence between σ-functions and control dependence. Sometimes there are not enough σ-functions, so control dependence information is missing in the SSI version of a program. This is the first problem, which is discussed in Section 5.8.1. Sometimes there are too many σ-functions, so extra control dependence information is present in the SSI version of a program. This is the second problem, which is discussed in Section 5.8.2. Both problems are illustrated by examples. Possible solutions are presented.

These problems did not become apparent until after I had completed the initial implementation of the SSI slicing algorithm. I did not anticipate them initially. In order to simplify the presentation of SSI slicing algorithm, Section 5.5 omits any description of these difficulties. As it stands, the chapter structure charts the journey of discovery I made. This seems to be an intuitive order of presentation. Note that the solutions proposed do not involve major alterations to the SSI slicing algorithm. Most of the differences are precomputations on the input SSI program.

5.8.1 Too Little Control Dependence

Figure 5.15 shows an SSI program that exhibits the problem of too little control dependence, which is mentioned in Section 5.7.2. The definitions of y_0 and y_1 are control dependent on the conditional branch, but this information is not represented by the SSI virtual register names since the assignments only use constant values, rather than virtual registers. SSI converts control dependence information into data dependence information, which works for virtual registers, but not for constants. Since constant values do not have any data dependence, it is not possible to associate control dependence information with them either. The issue is that SSI only encodes dependence in virtual register names, and constant values do not have virtual register names.

Thus a slice for virtual register y_2 would contain the assignments to y_0 and y_1, but not the conditional branch that determines which of the two assignments is actually executed. The remainder of this section describes three possible solutions to this problem of missing control dependence information.

Explicit Control Dependence Edges

One solution is to add control dependence edges to SSI. The original definition of SSI only has one kind of edge between nodes, the control flow edge. It would be possible to define an additional set of edges, $CD \subseteq N \times N$, such that $(n_0, n_1) \in CD$ if n_1 is control dependent on n_0. This necessitates a change in the SSI slicing algorithm to handle the new kind of edge. The virtual registers referenced by an instruction include not only the virtual registers used in the definition at that instruction, but also the virtual registers referenced in the predicate upon which that instruction is control dependent. A slice includes all definitions of virtual registers in the virtual register dependence set, supplemented by all the predicates upon which those definitions are control dependent.

There are three disadvantages with this approach.

1. The SSI definition and construction algorithm must be modified. In addition to placing pseudo-definition functions and renaming virtual registers, control dependence edges must be computed for each node. This uses reverse dominance frontiers [CFR+91] which are already required by SSI construction, so there is no change in the time complexity of the algorithm. The number of control dependence edges grows as $O(N^2)$ in the worst case.

2. This approach effectively transforms SSI into PDG. Data dependence is represented by virtual register naming. Control dependence is represented by explicit edges. The modified SSI slicing algorithm is identical to the PDG slicing algorithm [OO84], and also to the SSA slicing algorithm [LDB+99] which similarly relies on explicit control dependence edges. SSI and PDG IRs are now indistinguishable.

3. SSI σ-functions become irrelevant for slicing in this scheme.

Gating Pseudo-Definition Functions

A SSI ϕ-function represents the merge of multiple reaching definitions, but it does not specify the condition that determines which value is selected by the ϕ-function. Gating

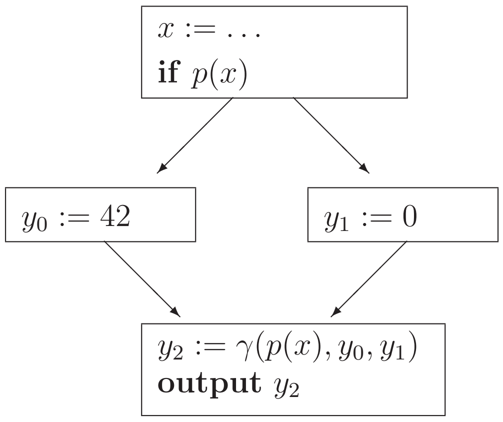


Figure 5.16: GSA version of example program

functions are defined as part of the gated single assignment form (GSA) component of the program dependence web (PDW) [BMO90]. Figure 5.16 shows the GSA version of the SSI program in Figure 5.15. A gating function captures the condition that specifies which definition to choose at a control flow merge point. The simplest gating function is the γ-function. For instance, $x_2 = \gamma(B, x_0, x_1)$ means that x_2 takes the value of x_0 if B is true, and x_1 if B is false. There are other gating functions for loop entry and loop exit nodes.

Note that a gating function is data dependent on the condition that controls which of its source operands it chooses. This effectively transforms control dependence into data dependence. Now the slicing algorithm simply needs to track data dependence, and control dependence is incorporated implicitly.

However, there are three disadvantages with this approach.

1. Additional precomputation is necessary before slicing can take place. SSI ϕ-functions must be replaced by appropriate gating functions. There are several algorithms that convert ϕ-functions to gating functions [BMO90, Hav93, TP95]. Some of these algorithms claim to be 'almost linear' in the size of the input CFG, but in the worst case, the time taken is $O(N \times E)$, where N is the number of nodes and E is the number of control flow edges.

2. This approach effectively transforms SSI into GSA. The new slicing algorithm is identical to GSA slicing, which is used for slicing the value dependence graph (VDG) [WCES94, Ern95].

3. Again, SSI σ-functions become irrelevant for slicing in this scheme.

Pseudo-Virtual Register References

The simplest solution, though the least elegant of the three, is to rewrite the SSI program so each definition that only involves constant source operands also uses a pseudo-virtual register. Figure 5.17 shows a rewritten version of the example program from Figure 5.15. The semantics of the program are unchanged. However, there is an extra virtual register, *zero*, which is used by every instruction that previously only used constant operands. There are pseudo-definitions for *zero* at the appropriate control flow split and merge

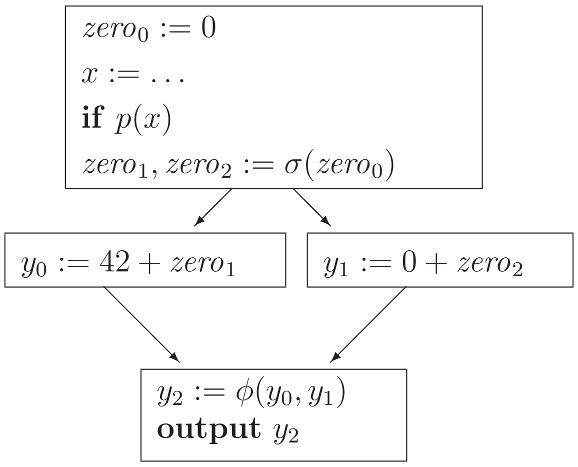


Figure 5.17: Rewrite of example program with pseudo-virtual register

points, in order for *zero* to satisfy the SSI virtual register naming conventions. Now, all control dependences can be represented using σ-functions. This process adds just enough extra dependence information to capture the control dependences of constant operands via σ-functions.

There are three reasons why this is the preferred approach.

1. A small amount of precomputation is required, but this is minimal. Constant assignments need to be replaced by assignments that reference the pseudo-virtual register. Then ϕ- and σ-functions must be inserted for the pseudo-virtual register, using the standard SSI construction algorithm.

2. The IR to be sliced is standard SSI. This means that the standard algorithms for construction and slicing can be employed. The alternative approaches described above require adding new edges or new pseudo-definition functions to the SSI semantics, which would entail significant modifications to both algorithms for construction and slicing.

3. SSI σ-functions are required to represent control dependence information in this scheme.

Note that the dependence flow graph (DFG) IR uses a similar approach to represent control dependence [JP93].

5.8.2 Too Much Control Dependence

Consider the CFG program in Figure 5.18. Figure 5.19 shows the SSI version of this program, which contains too much control dependence information.

The uses of x (SSI virtual registers x_1 and x_2) in the two arms of the conditional branch mean that a σ-function is required for x (SSI virtual register x_0) at the control flow split point. A ϕ-function is required to merge the virtual registers defined by the σ-function. Then the final use of x (SSI virtual register x_3) uses the value defined by the ϕ-function. This scenario is ideal for data flow analysis, where data flow information

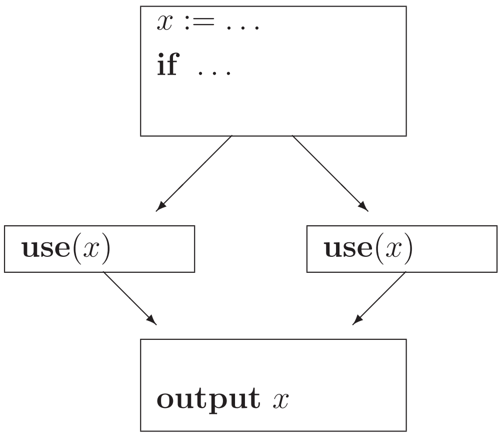


Figure 5.18: Example CFG program

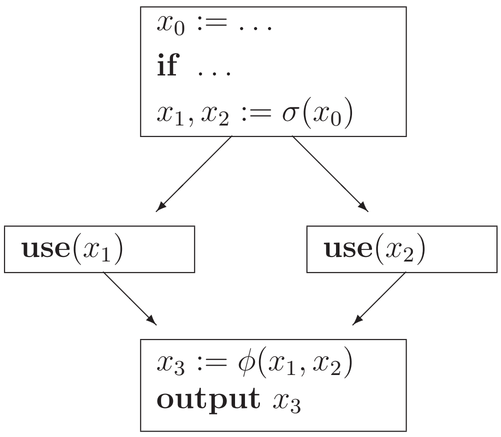


Figure 5.19: Example SSI program with too much control dependence

may be generated at the virtual register use sites in the conditional branch arms, and has to be merged together at the subsequent control flow merge point. However, in the context of program slicing, it makes more sense for x after the control flow merge (SSI virtual register x_3) to have the same name as the x before the control flow split (SSI virtual register x_0). As things stand, a SSI slice on x_3 would contain the ϕ-function that defines x_3, and therefore also the σ-function that defines the source virtual registers of the ϕ-function, and therefore also the predicate associated with that σ-function. However, this slice is based on spurious dependence relations. The predicate should not be in the slice, because the value of x_3 is independent of the outcome of the conditional branch. In the original CFG program, x is only used in conditional contexts, and not redefined. This is apparent in the SSI program since the source operands of the ϕ-function are all defined by the same σ-function.

SSI sometimes enforces more virtual register renaming than is required for slicing. Virtual register dependence information is only influenced by definitions and enclosing conditional context. So, if the source operands of a ϕ-function are all defined by the same σ-function, the dependence calculation should not incorporate the virtual registers used in the predicate associated with this σ-function, when tracing back dependences from the ϕ-function. After the ϕ-function, this conditional context is no longer relevant. Note that a ϕ-function's source operands can all originate from the same σ-function in a transitive manner, through several nested pseudo-definitions, as illustrated by Figure 5.24.

The simplest solution is to accept that the SSI slicing algorithm overestimates control dependence, so the slices returned are too large. Any computed slice is *safe*, since it includes all instructions that may affect the value of the virtual registers in the slicing criterion. However the slice may contain extra conditional branch instructions that do not affect the slicing criterion at all. Thus the slice is unlikely to be minimal. A *minimal* slice is the smallest slice that satisfies the criterion [Dan99].

A better solution is to disregard control dependence information in certain cases. For instance, when a ϕ-function's source operands all derive their value from a common σ-function, then there is no need to incorporate the control dependence information associated with this σ-function when computing slices for the ϕ-function destination operand. The dependence calculation should *bypass* the σ-function, since its dependences do not affect the value of the virtual register below the subsequent control flow merge point.

There are three stages at which it is possible to calculate this *bypass information*:

1. at SSI construction time, or
2. after SSI construction time but before slice time, or
3. at slice time.

The first two methods are precomputation of bypass information. The third method is on-the-fly computation of bypass information. Method 1 is straightforward to implement. A σ-function s should be bypassed by the ϕ-function p that is inserted to merge all the virtual registers defined by s. This avoids incorporating false dependence information. In the optimistic SSI construction algorithm from Section 3.5, this corresponds to all ϕ-functions inserted after the first iteration (i.e. all ϕ-functions that would not be present in SSA). Such ϕ-functions could be marked at SSI construction time. Method 2 is described at length in the remainder of this section. Method 3 complicates the SSI slicing algorithm

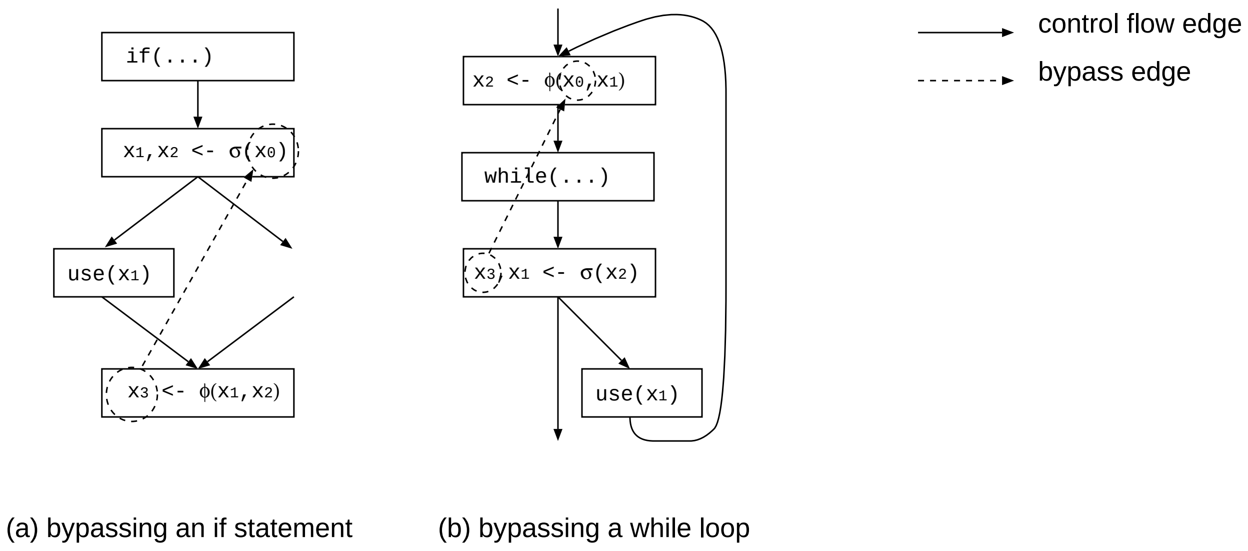


Figure 5.20: Bypass situations in rSSI

considerably. The algorithm has to compute bypass information on-the-fly, combined with tracking virtual register dependence. In principle this could be done, but it is contrary to the general philosophy developed in this dissertation:

Precompute as much information as possible, to make the final data flow analysis as simple and as efficient as possible.

Methods 1 and 2 precompute bypass information, which means that the SSI slicing algorithm from Section 5.5 requires minimal changes.

Precomputation of Bypass Information

For the sake of simplicity, this presentation concentrates on a restricted form of SSI, which this section refers to as rSSI. The bypassing approach can be generalized to full SSI, although it may be easier to transform more general SSI programs into rSSI.

In rSSI, ϕ-functions must have exactly two source operands, and σ-functions must have exactly two destination operands. N-operand pseudo-definition functions can be simulated by cascading two-operand pseudo-definition functions. This is how an N-operand ϕ-function is replaced by γ-functions in GSA [BMO90]. In rSSI, each ϕ-function can be paired with a single σ-function. Figure 5.20 shows the two cases when bypassing may be required in rSSI. A ϕ-function may bypass a σ-function, and vice versa. This is represented by *bypass edges* between virtual registers in the SSI program.

The rest of this section presents an algorithm that computes bypass information for each virtual register in an arbitrary rSSI program. The bypass algorithm associates a single unit of bypass information with each virtual register in the program, although in fact, not all virtual registers require bypass information. The only virtual registers that may need bypass information are destination operands of ϕ-functions and σ-functions. This information is stored in the *binfo* array, which is subscripted by virtual registers. Each entry *binfo*$[v]$ may be one of the following data flow values.

```
bypass() =
        while $change$ = true
            $change \longleftarrow$ false
            for each virtual register $v$
                if $binfo[v] = \bot$
                    analyse-def($v$)
```

Figure 5.21: top-level bypassing algorithm

- $\bot$: This indicates that the no bypass information has been computed for v.
- $\top$: This indicates that bypassing is not possible for v.
- virtual register x: This indicates that bypassing may take place, since virtual register v always gets its value from x.

All entries in *binfo* are initialized to $\bot$.

Figure 5.21 shows the bypass() function. It iterates over all virtual registers in the program until it reaches a fix point. For each virtual register, the algorithm checks if it can be bypassed to another virtual register, avoiding spurious control dependences. Figure 5.22 shows the analyse-def() function, which determines the sources of values that flow into each virtual register. It updates the *binfo* array and global variable *change* if required. Figure 5.23 shows the find-src() function, which discovers whether the operand supplied as a parameter obtains its value via a bypass edge.

The time complexity of this bypass algorithm is $O(N^2)$ on average, but $O(N^7)$ in the worst case, where N is the number of nodes in the SSI program. Function find-src() is $O(1)$ generally, and $O(N)$ in the worst case. Function analyse-def() has the same time complexity as find-src(). Cytron et al [CFR+91] show that the number of virtual registers in SSA grows as $O(N)$ generally, and Ananian [Ana99] confirms this for SSI. However the number of virtual registers for SSI grows as $O(N^3)$ in the worst case. Therefore *binfo* has $O(N^3)$ entries in the worst case. Therefore the top-level bypassing algorithm may perform $O(N^3)$ iterations of the while loop, since it is monotonic and at least one *binfo* entry must change each time. Each iteration of the while loop checks all *binfo* entries. So the top-level algorithm has $O(N^7)$ time complexity in the worst case. In general, time complexity is more like $O(N^2)$, when function find-src takes time $O(1)$ and *binfo* has size $O(N)$.

Using Bypass Information for Slicing

Once the bypassing precomputation is complete, SSI slicing proceeds as normal. The SSI virtual register dependence equation is unchanged from Figure 5.7. However the behaviour of the auxiliary functions must be modified. If $binfo[v] \neq \top \text{or} \bot$, then $ref(defsite(v))$ should return $binfo[v]$. Otherwise the algorithmic details are unchanged. This will ensure that the appropriate pseudo-definitions are bypassed properly.

After the virtual register dependence information has been computed, the slice must be constructed. As in Section 5.5, the slice should include all instructions that define

```
analyse-def($v$: virtual register) =
        $def \longleftarrow$ get-unique-def($v$)
        **if** $def$ is an ordinary definition
            $binfo[v] \longleftarrow \top$
            $change \longleftarrow$ true
        **else if** $def$ is a $\phi$-function, $v \longleftarrow \phi(v_1, v_2)$
            $x_1 \longleftarrow$ find-src($v_1$)
            $x_2 \longleftarrow$ find-src($v_2$)
            **if** $x_1$ and $x_2$ are defined by same $\sigma$-function, $x_1, x_2 \longleftarrow \sigma(x_0)$
                $binfo[v] \longleftarrow x_0$
                $change \longleftarrow$ true
            **else if** either $x_1$ or $x_2$ defined by ordinary definition
                $binfo[v] \longleftarrow \top$
                $change \longleftarrow$ true
        **else if** $def$ is a $\sigma$-function, $v, w_0 \longleftarrow \sigma(w_1)$
            $x \longleftarrow$ find-src($w_1$)
            $xdef \longleftarrow$ get-unique-def($x$)
            **if** $xdef$ is a $\phi$-function, $x \longleftarrow \phi(w_0, z)$
                $binfo[v] \longleftarrow z$
                $change \longleftarrow$ true
            **else if** $xdef$ is an ordinary definition
                $binfo[v] \longleftarrow \top$
                $change \longleftarrow$ true
        **return**
```

Figure 5.22: analyse-def, auxiliary function for bypassing algorithm

```
find-src($v$: virtual register) : virtual register =
        $v_0 \longleftarrow v$
        $v_1 \longleftarrow binfo[v_0]$
        **while** $v_1$ is a virtual register (not $\top$ or $\bot$)
            $v_0 \longleftarrow v_1$
            $v_1 \longleftarrow binfo[v_0]$
        **return** $v_0$
```

Figure 5.23: find-src, auxiliary function for bypassing algorithm that recursively traverses bypass edges for virtual registers

virtual registers in the virtual register dependence set, together with conditional branches associated with (non-bypassed) σ-functions in the slice. However, pseudo-definition functions in the slice may now need to be rewritten. If a pseudo-definition function p that defines virtual register v has $binfo[v] = x$, then p must be rewritten as a simple assignment $v \leftarrow x$. This effectively bypasses the whole of the single-entry-single-exit (SESE) region in between a pair of corresponding pseudo-definition functions.

Example

This section presents an example of the bypassing algorithm, and the modified slicing algorithm. Figure 5.24 shows the example program.

The *binfo* array computed by bypass() is shown in Figure 5.25.

Now consider calculating a slice for x_9. The virtual register dependence set will contain x_9, but not x_8 or x_2. This is because $binfo[x_9]$ is x_0, so the SESE region between $9 : \phi$ and $2 : \sigma$ is completely bypassed. So the virtual register dependence algorithm terminates with dependence set $\{x_0, x_9\}$. The constructed slice contains statements 1 and 9. However, statement 9 is transformed to $x_9 := x_0$ rather than being a ϕ-function. Such copy operations can be eliminated from slices by standard copy propagation analysis.

Related Work

The dependence flow graph (DFG) [JP93] is briefly mentioned in Section 2.6.1. DFG has a number of similarities with SSI. DFG edges connect virtual register definitions to virtual register uses, but are intercepted by *switch* nodes (like SSI σ-functions) at control flow split points, and by *merge* nodes (like SSI ϕ-functions) at control flow merge points. The key difference between DFG and SSI is bypass behaviour. A DFG edge for virtual register x can bypass a single-entry-single-exit (SESE) region that does not contain a *definition* of x [JP93]. SESE regions are defined in Section 3.5.1. In contrast to DFG, SSI can bypass a region (i.e. refrain from inserting pseudo-definitions for x) only if this region does not contain a *mention* of x. This may cause overestimates of control dependence information, as explained above. The solution to this problem is to introduce bypassing information in SSI, similar to DFG bypassing edges. SSI bypassing edges connect virtual register uses directly to definitions without being intercepted by intervening extraneous pseudo-definitions due to matching pairs of control flow split and merge points.

Unlike SSI bypassing information, DFG bypassing edges are built at DFG construction time. DFG is constructed using a pessimistic algorithm. [JP93]. Ananian's SSI construction algorithm in Section 3.5.1 is based on the DFG construction algorithm. To the best of our knowledge, no other augmented CFG IR incorporates explicit bypassing information.

Discussion

Bypassing information has some effect on the performance of the SSI slicing algorithm. The precomputation time will increase. Programs to be sliced must be converted to SSI, and then the SSI code must be analysed to determine bypassing information. However, it is assumed that this precomputation time is amortized over many slicing queries on the subject program. Bypassing should reduce the time taken for each slicing query. The

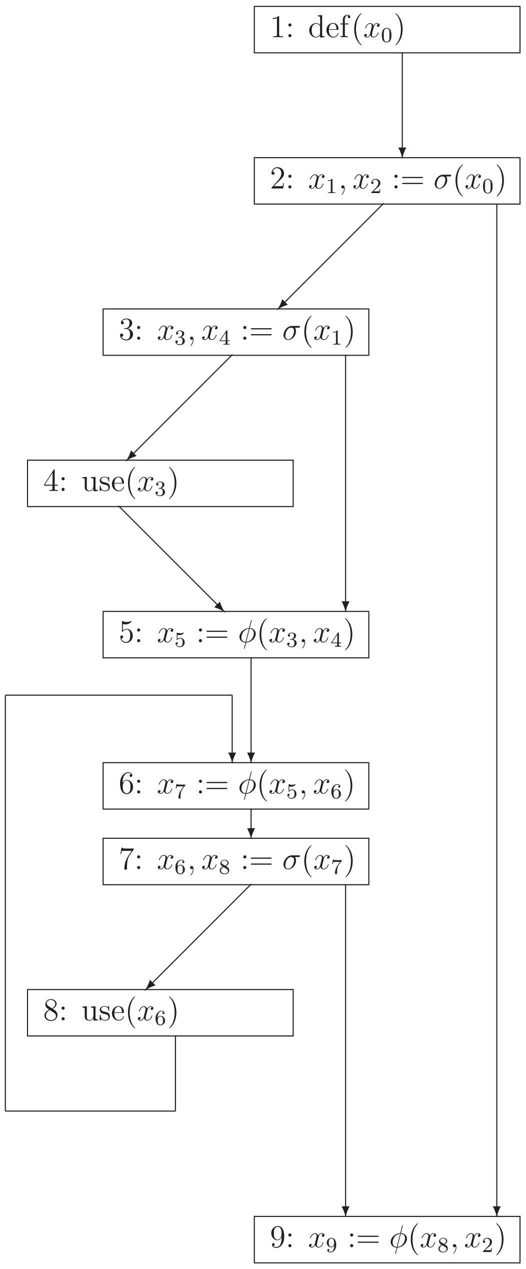

1: def(x_0)
2: $x_1, x_2 := \sigma(x_0)$
3: $x_3, x_4 := \sigma(x_1)$
4: use(x_3)
5: $x_5 := \phi(x_3, x_4)$
6: $x_7 := \phi(x_5, x_6)$
7: $x_6, x_8 := \sigma(x_7)$
8: use(x_6)
9: $x_9 := \phi(x_8, x_2)$

Figure 5.24: Example SSI program for bypassing

virtual register v	$\mathit{binfo}(v)$
x_0	$\top$
x_1	$\top$
x_2	$\top$
x_3	$\bot$
x_4	$\bot$
x_5	x_1
x_6	$\bot$
x_7	$\bot$
x_8	x_5
x_9	x_0

Figure 5.25: *binfo* array for example program

virtual register dependence analysis does slightly more work for ϕ-functions, since it has to look up the bypass information. However, since virtual register dependence sets should be smaller with bypassing information, the overall query time should decrease.

This proposed solution to the problem of 'too much control dependence' is acceptable because it does not necessitate serious changes to the SSI slicing algorithm. All the details of computing bypass information are factored out into the precomputation stage.

5.9 Possible Extensions

The slicing discipline considered so far in this chapter has been syntax-preserving, static, backward slicing. Recall that Section 5.2 defined each of these terms. This section examines how the SSI slicing algorithm could be modified to handle other slicing variants.

Backward slicing could easily be converted to *forward* slicing, simply by tracing dependences in the other direction, from virtual register definitions to uses. Forward slices show the parts of the program that are affected by values in the slicing criterion. This dependence computation is easy to perform in SSI. A lookup table must be constructed that relates definitions to uses (a form of def-use chaining). This could be stored in a sparse manner, by associating uses with each SSI virtual register.

Static slicing could easily be converted to *dynamic* slicing, where the values of input virtual registers are taken into account. In SSI, this simply corresponds to performing a limited amount of constant propagation, and then dead code elimination, prior to computing a static slice as before.

Amorphous slicing is equally straightforward. The algorithm would compute the static slice as usual, then apply aggressive compiler optimizations. There are many SSI-based optimizations available, such as constant propagation, expression simplification, dead code elimination, and partial redundancy elimination. Note that transformations may enable further slicing, so this process could be applied iteratively.

Type-directed slicing is a new discipline proposed by this dissertation. It would take into account the runtime types as well as the data flow, when calculating program slices. It would be most useful for dynamically typed languages like LISP, or modern object-oriented languages like Java. SSI could form the ideal basis for such an algorithm since it already supports DFA-based type inference (Section 4.5), as well as efficient slicing techniques.

There are many possible uses for type-based slicing information:

- debugging. This is the original, classical use for slicing information. It should be easier to locate bugs in program with some knowledge of the types involved. The specified type information ought to restrict the possible search area. (Perhaps this information may be known by the programmer, but cannot be inferred statically by type inference algorithm.)

- visualization. Program comprehension is another standard use for slicing information. In an integrated software engineering environment, interactive type-based slices may help programmers to visualize and understand programs in a more informative manner.

- optimization. For instance, type-based slicing information may aid the calculation of optimal locations to insert runtime type checks, so as to minimize overhead.

Steindl [Ste98] describes a slicer for Oberon-2, which is a modular object-oriented language. He allows manual user intervention to restrict the types of objects in order to reduce the number of methods to consider at dynamic dispatch sites. To the best of our knowledge, there is no other related work dealing with slicing using types in this manner.

5.10 Concluding Remarks

This chapter has demonstrated that SSI moves complexity from the data flow analysis phase to the earlier IR construction phase. The effect of this factoring should make analyses faster, which is a particular advantage for debugging and program comprehension. The slicing algorithm presented shows clearly that SSI factors out the computation of virtual register dependence information since it is encoded implicitly as part of the virtual register renaming scheme.

In the earlier consideration of SSI construction techniques, Chapter 3 rubbished the linear time algorithm (by Ananian), and preferred the potentially quadratic algorithm. In contrast, this chapter prefers the linear algorithm for slicing as opposed to the potentially cubic algorithm (by Weiser). These conclusions may seem inconsistent, but there is a subtle difference. The linear SSI construction algorithm inserts too many ϕ- and σ-functions, then subsequently prunes them out of the transformed program which takes extra time. Thus it performs poorly due to *information overload*. On the other hand, Weiser's superlinear slicing algorithm performs poorly due to *information shortage*. It needs to recompute dependence information, whereas the SSI slicing algorithm never needs to recompute dependence information, since this information is latent in the IR itself.

One key issue is whether SSI gives any significant advantage over all other IRs for slicing. The honest answer is probably that SSI can only be 'as good' as other IRs in the best case. However, this chapter has shown that SSI enables simple specification and efficient implementation of the slicing algorithm. Also, it is advantageous to have a single IR that can be used for all compiler analyses. One area for future work is the implementation of new slicing variants in this SSI framework, as suggested by Section 5.9. Type-directed slicing seems particularly novel.

Another key issue is whether the SSI encoding of control dependence is better than either the PDG encoding or the association of control dependence information with SSA. This chapter has shown that σ-functions transform some, but not all, control dependence into data dependence. There is a fundamental difference between σ-functions and control dependence edges. Further work is required to clarify this difference formally. Note that SSI, with the appropriate fixes from Section 5.8, is effectively both CFG and PDG. This is an interesting combination. There is redundancy in this IR, but it may enable efficient traversal, analysis and transformation. Again, more work is needed here.

So far, all the data flow analyses presented in this dissertation have assumed a simple intraprocedural model for subject programs. The next chapter extends SSI to deal with multiple procedures.

Chapter 6

Interprocedural Extensions

There are several ways to accommodate SSI in an interprocedural program representation. This chapter explores four approaches.

6.1 About this Chapter

6.1.1 Objectives

Procedures complicate program analysis. A more sophisticated IR structure is required to handle programs consisting of multiple procedures. However procedures are the cornerstone of every programming language, so it is essential to support them properly. This chapter aims to scale SSI beyond the scope of a single procedure. It will show how existing techniques for interprocedural analysis can be adapted to incorporate SSI. It will argue that SSI enables each technique to become more effective. Note that this chapter concentrates on the IRs for interprocedural analysis, rather than any particular analysis or transformation.

6.1.2 Outline

Section 6.2 reviews the notion of a procedure. It indicates why interprocedural analysis is intrinsically more complicated than intraprocedural analysis. Section 6.3 describes two interprocedural IRs based on the call graph concept, then it shows how these IRs may be extended to use SSI. Section 6.4 describes two interprocedural IRs that refactor the control flow behaviour of subject programs, then it shows how these IRs may be extended to use SSI. Finally Section 6.5 concludes.

6.1.3 Contributions

This chapter makes three significant contributions.

1. It provides a taxonomy of interprocedural IRs. The distinction between call graph and refactoring approaches seems intuitive, but it has not been drawn before.
2. It shows how SSI may be extended to interprocedural scope. There has been no previous formulation of interprocedural SSI. The same techniques should apply to SSA and other similar IRs.

3. Section 6.4.2 presents a new algorithm that converts SSI into a functional notation, similar to continuation passing style.

6.2 Procedures

So far, this dissertation has only dealt with intraprocedural analysis. Programs have had a monolithic structure, represented by single-procedure IRs such as CFG. However the procedural paradigm is one of the fundamental abstractions that underlies most high-level programming languages. A procedure is a self-contained unit of code, generalized over formal parameters. A procedure is called (also known as executed or invoked) when that procedure is supplied with actual parameters (arguments) at a call site. A call site is a program point at which a procedure call takes place. Each call site has an associated context, which is related to the program state at that point. At every call site, two procedures interact. The caller procedure issues the call to the callee procedure, and supplies actual parameters. At some later stage, the callee procedure hands control back to the caller procedure, and may supply return values.

Procedures are beneficial to programmers in a couple of ways:

1. They separate interface and implementation, which is advantageous for all the usual software engineering reasons (modularity, maintainability, modelling and mathematical reasoning).

2. They facilitate code reuse. A procedure's computation may be repeated by issuing several calls. A procedure may be applied to alternative data by supplying different actual parameters.

However, procedures are detrimental to program analysis. This is due to the difficulty of propagating precise data flow information across procedure boundaries. The obstacles are numerous:

1. Since a procedure may be called from many different contexts, the only data flow information that is guaranteed always to be valid at the entry to that procedure is the intersection of the information from each context. More specific information requires context-sensitive analysis (see Section 4.2.6) which is expensive to implement [WL95].

2. The namespace restrictions enforced by procedure definitions allied with the aliasing of names due to parameter passing ensure that the same value has different names in different procedures, which complicates analysis. More generally, at procedure boundaries, some names are created, others are destroyed, others cease to be visible and others change their names!

3. Some languages permit procedure-valued variables, and transfer of procedures as arguments and results. In such languages, it is not always possible to determine statically which procedures are actually called at each call site. This adds further imprecision to the analysis due to necessary overestimation of potentially called procedures.

```
int f(int a, int b)
{
  return a+b;
}

int g(c, d)
{
  if (c>0) return d;
  else return f(d, d);
}

int main()
{
  int x = f(0,0);
  return g(x,1);
}
```

Figure 6.1: Example program with procedure calls

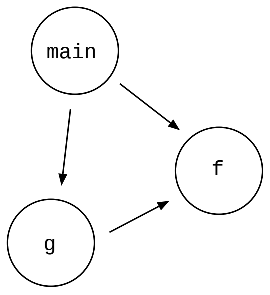


Figure 6.2: Example call graph

Interprocedural analysis acquires information from the whole program. This contrasts with intraprocedural data flow analysis, which only gathers information from within the scope of a single procedure at a time. The remainder of this chapter explores IRs that represent the whole program in order to support interprocedural analysis.

6.3 Call Graph Approaches

The two most common interprocedural IRs are *supergraphs* and *summary graphs*, which are both reviewed in this section. Both supergraphs and summary graphs are derived from the program *call graph*. A call graph is a directed multigraph. Each node represents a procedure, and a directed edge $p_1 \rightarrow p_2$ represents a potential call from p_1 to p_2. There may be more than one edge $p_1 \rightarrow p_2$ since there may be more than one distinct call site in p_1 that may call p_2.

For example, consider the program source code given in Figure 6.1. The corresponding call graph is shown in Figure 6.2.

```
for each procedure $f$
    for each CFG node $n$ : call $g(\ldots)$
        replace $n$ by a new node $n'$ : callpoint $g(\ldots)$, with $\text{pred}(n') = \text{pred}(n)$
        add a new node $n''$ : returnpoint $g(\ldots)$, with $\text{succ}(n'') = \text{succ}(n)$
        add a new edge from $n'$ to entry node of $g$
        add a new edge from exit node of $g$ to $n''$
    end
end
```

Figure 6.3: Supergraph construction algorithm

In the supergraph representation, each atomic procedure node from the call graph is replaced by a CFG-like subgraph representing that same procedure. Supergraph edges connect specific program points within procedures, modelling control flow in a much more detailed way. Section 6.3.1 describes the supergraph IR in detail. In the summary graph representation, each atomic procedure node from the call graph is replaced by a set of nodes that model parameter passing. Edges encapsulate data flow within and between procedures. Section 6.3.2 describes the summary graph IR in detail.

6.3.1 Supergraphs

Myers introduces the program supergraph [Mye81]. Each procedure is modelled as a CFG, as presented in Section 2.5.1. These (intraprocedural) CFGs are combined to create an (interprocedural) supergraph as follows:

Previously, a procedure call was modelled as a primitive instruction. Now, a procedure call should consist of two CFG nodes, a callpoint and a returnpoint. The callpoint specifies actual parameters, the returnpoint specifies location of return value. Each callpoint connects to one or more n_{entry} nodes of procedures that may be called at that point. Each n_{exit} node connects to one or more returnpoint nodes of procedures that may have called this procedure. Figure 6.3 shows the algorithm that builds a supergraph from a collection of CFGs.

Any program execution path can be modelled as a path in the supergraph. However, not every path in the supergraph is a valid execution path, since each edge from a `callpoint` node has exactly one corresponding edge to a `returnpoint` node. (The program execution sequence should always return from a callee procedure to the appropriate caller procedure, rather than an arbitrary procedure that may call the callee at some other point in the execution sequence.) This is the problem of context-sensitivity, matching `returnpoint` nodes with `callpoint` nodes.

Data flow problems may be formulated and solved on the supergraph using standard monotone data flow analysis frameworks and iterative solution techniques [Mye81]. Simple intraprocedural analysis techniques are extended to handle callpoint and returnpoint nodes.

However the supergraph is a large graph, although not as large as a deproceduralized version of the program, see Section 6.4.1. There is one CFG for each procedure p, with

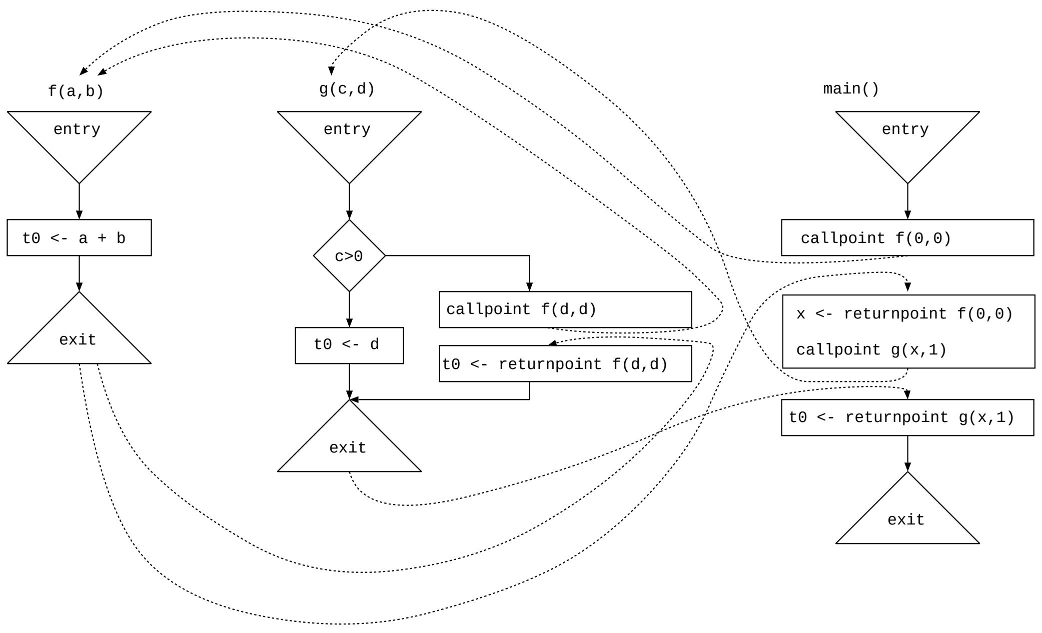


Figure 6.4: Example supergraph

N_p incoming call edges for the N_p possible call sites, and N_p outgoing return edges to the corresponding call sites. This is a large data structure to compute, analyse and store and maintain, which may cause concerns regarding scalability in terms of computational time and space. Figure 6.4 gives an example supergraph, based on the example program from Figure 6.1.

The supergraph approach to interprocedural analysis is used in several systems. It has been instantiated with other intraprocedural graphs apart from CFGs. Figure 6.5 tabulates a number of IRs based on the supergraph concept.

Many kinds of data flow analyses have been extended from intraprocedural scope to interprocedural, using the supergraph representation. Some examples are given below.

Supergraph variety	Canonical reference	Procedure representation	Additional Notes
Supergraph	[Mye81]	CFG	original formulation
Interprocedural CFG	[LR91]	CFG	-
System Dependence Graph	[HRB90]	PDG	with context-sensitive extensions
Interprocedural Value Dependence Graph	[Ern95]	VDG	-
Interprocedural SSA	[LDB+99]	SSA	uses ϕ-functions to model parameter passing

Figure 6.5: Different kinds of supergraphs

- Liveness analysis, for interprocedural register allocation [Wal86].
- Constant propagation, for interprocedural partial evaluation [MS93].
- Slicing of complete programs [HRB90].
- Conditional branch elimination [BGS97].
- Alias analysis [LR91].

Incorporating SSI

It is straightforward to see that interprocedural SSI can be formulated via the supergraph representation. The supergraph form should be instantiated with each procedure represented as an SSI graph, rather than a CFG. Interprocedural SSA [LDB+99] reuses the standard SSA ϕ-functions for interprocedural data flow merges. Interprocedural SSI could also use standard SSI σ-functions for interprocedural data flow splits. The details are given below.

- ϕ-functions should be used for formal parameter definitions in callee procedures that have multiple callers.
- ϕ-functions should be used for procedure return values in caller procedures at indirect call sites.
- σ-functions should be used for actual parameter definitions at indirect call sites.
- σ-functions should be used for procedure return values in callee procedures that have multiple callers.

Chapter 4 remarked that virtual register renaming alleviated the problems of flow-insensitivity for intraprocedural analysis. At this point, it is clear to see that interprocedural SSI may alleviate some of the problems of context-insensitivity for interprocedural analysis. The ϕ- and σ-functions for parameters and return values enable explicit data flow merges and splits at procedure boundaries. If parameters and return values can be matched up correctly (using techniques such as CFL-reachability [Rep98]) then there should be less imprecision due to context contamination.

6.3.2 Summary Graphs

The program summary graph [Cal88] is a directed graph derived from the call graph and supplementary information about parameter passing. The summary graph is more detailed than than the call graph, but it is more concise than the supergraph. The summary graph records all interprocedural data flow that occurs as a result of parameter passing at procedure boundaries, and brief summaries of intraprocedural data flow for each procedure.

A summary graph consists of four different kinds of nodes: `entry`, `exit`, `call` and `return`. For each procedure f with n formal parameters there are n `entry` nodes ($\texttt{entry}_1^f$ $\ldots\texttt{entry}_n^f$) and n `exit` nodes ($\texttt{exit}_1^f$ $\ldots\texttt{exit}_n^f$) all directly associated with procedure f. For each call site $f(arg_1, \ldots, arg_n)$ in procedure f', there are n `call` nodes ($\texttt{call}_1^f$

... $\texttt{call}_n^f$) and n `return` nodes ($\texttt{return}_1^f$ ... $\texttt{return}_n^f$) all associated directly with procedure f'.

Some edges model interprocedural data flow, binding actual parameters to formal parameters. These edges depend solely on parameter passing information, not at all on intraprocedural data flow information. Such edges go from callers' `call` nodes to corresponding callees' `entry` nodes. Edges are also inserted from callees' `exit` nodes to callers' `return` nodes for call-by-reference parameter passing, but not for call-by-value.

Other edges model intraprocedural data flow, based on reaching definitions information. Such edges go from $\texttt{entry}_i^f$ to $\texttt{call}_j^g$ if the value of the ith formal parameter at the entry point of procedure f reaches the jth actual parameter at the call site to procedure g within the body of f, and from $\texttt{entry}_i^f$ to $\texttt{exit}_i^f$ if the value of the ith formal parameter at the entry point of procedure f reaches the return statement at the exit point. Edges are also required (for call-by-reference parameter passing) from $\texttt{return}_i^f$ to $\texttt{call}_j^g$ if the value of the ith actual parameter, subsequent to return from procedure f, reaches the jth actual parameter of a call to procedure g, all within the body of another procedure h.

Figure 6.6 shows an example summary graph, based on the example program from Figure 6.1. Note that since this example C program uses call-by-value parameter passing, there are no edges to or from `return` nodes in this summary graph.

Often the summary graph representation is supplemented with extra intraprocedural nodes for specific analyses. In Callahan's original paper [Cal88], he introduces `use` nodes for variables, in order to solve interprocedural liveness analysis. The interprocedural flow graph [HS94] extends the summary graph so as to compute interprocedural def-use chains. Agrawal [Agr00] adds `(new className)` nodes to enable interprocedural type inference for object-oriented languages.

The summary graph is a very popular approach to whole program analysis. It is deployed in the FIAT system [HMCCR94], a generic compiler framework for interprocedural analysis and transformation. It is generally efficient and gives accurate results.

Incorporating SSI

Since the summary graph is primarily concerned with interprocedural data flow resulting from parameter passing, it does not naturally accommodate the SSI concept. It would be possible to use SSI to compute individual intraprocedural summaries. Previous chapters have established that SSI is good for intraprocedural data flow analysis. So standard intraprocedural analysis is performed on each procedure (maybe using SSI) to generate accurate summaries (which appear as intraprocedural edges in the summary graph). Then interprocedural data flow analysis takes place using the summary graph directly.

6.4 Control Flow Refactoring Approaches

Refactoring is defined as "a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior" [Fow05]. This section considers two radical control flow transformations on programs to make them amenable to interprocedural analysis.

One of the themes of this dissertation is that the programmer's decisions (at source code level) are not set in stone, and can be overruled at intermediate stages of compila-

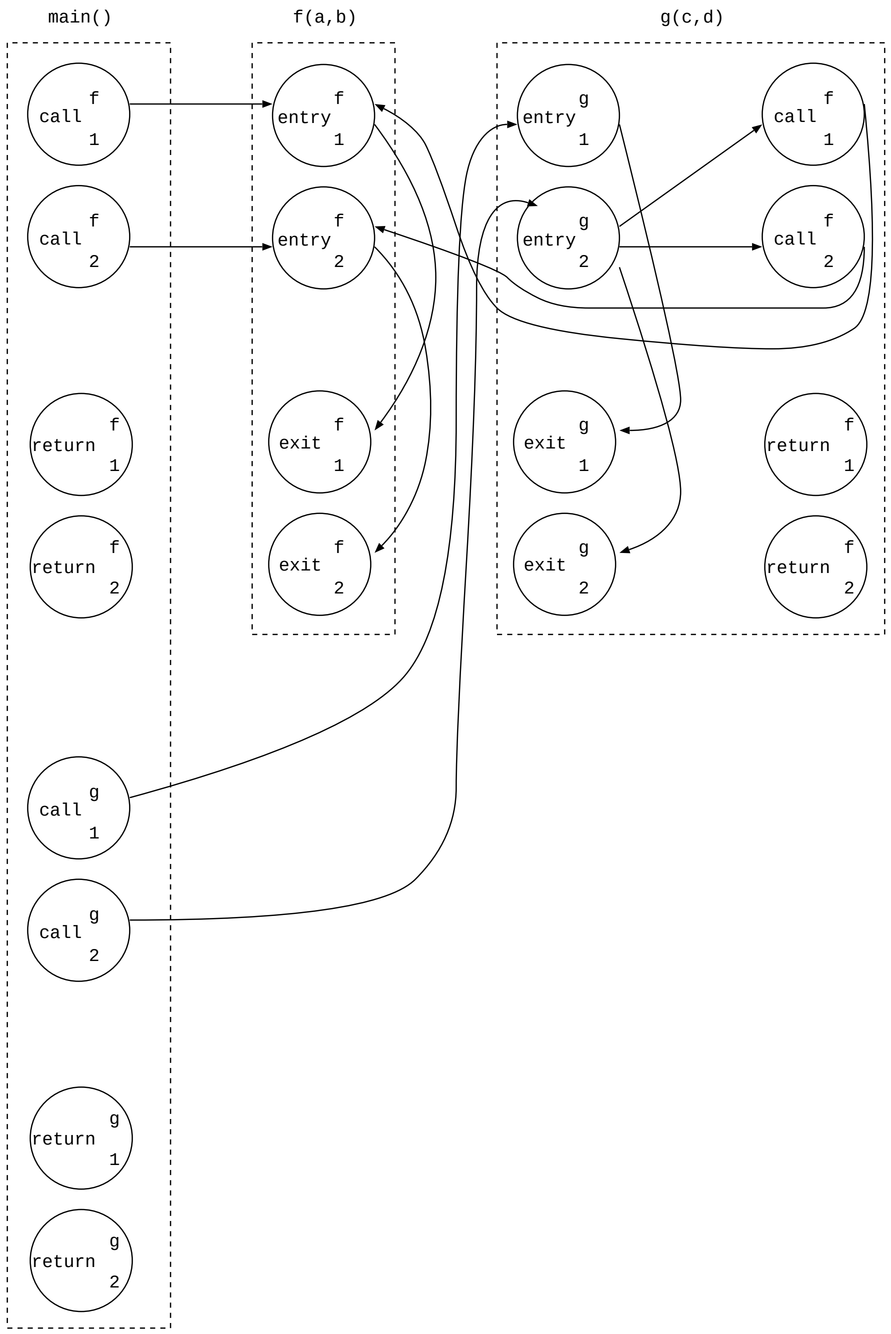


Figure 6.6: Example summary graph

```
int main()
{
  int x = 0+0;           /* inline call of f(0,0) */
  return (x>0)?1:(1+1);  /* inline call of g(x,1) */
}
```

Figure 6.7: Deproceduralized example program

tion. So far, this dissertation has advocated the normalization of virtual register names, overriding any naming decisions made by the programmer. This chapter shows how interprocedural control flow can be normalized. It presents two opposite extremes. Section 6.4.1 encodes all control flow without using procedures, by inlining procedure bodies at each call site (deproceduralization). Section 6.4.2 encodes encodes all control flow as procedure calls (functionalization). Then Section 6.4.3 compares these two approaches empirically, using a standard benchmark test.

6.4.1 Deproceduralization

The simplest approach to interprocedural analysis is to eliminate procedure calls altogether, by a process known variously as deproceduralization, or procedure unfolding, or inlining or procedure integration. This process converts a program with multiple procedures and procedure call sites into a monolithic single-procedure program. Thus interprocedural data flow problems reduce to simpler intraprocedural data flow problems. Richardson and Ganapathi [RG89a] observe that deproceduralization can be viewed as an upper limit on the improvement available through use of interprocedural data flow information.

Methodology

A program may be deproceduralized as follows:

1. start with entry procedure `main`
2. for every call site $f(arg_1, \ldots, arg_n)$ in the body of procedure `main`, replace this call site by the body of the deproceduralized version of procedure $f(par_1, \ldots, par_n)$, modelling the effects of parameter passing by substituting the actual parameters $arg_1, \ldots, arg_n$ in place of the formal parameters $par_1, \ldots, par_n$.

Figure 6.7 shows the deproceduralized version of the example program from Figure 6.1. Note that parameters are restricted to constants and virtual registers. All dereferencing must be performed explicitly, with results saved in virtual registers. This avoids unintended semantic changes that may occur in the presence of compound parameters, reminiscent of the confusion over call-by-name parameter passing.

Of course, this transformation is only valid for non-recursive programs, which have an acyclic call graph. Otherwise deproceduralization leads to infinite expansion; the algorithm given above would never terminate. Sometimes recursive procedures may be replaced by iterative looping constructs, but these program rewrites are difficult to achieve automatically. Indirect procedure calls can also cause problems. In these cases it is not always possible to determine statically which procedure will be called at runtime, so it is not known which procedure body to insert at the call site. One solution is to inline all potential callee procedures and insert a conditional branch at the call site to determine which inlined procedure body should be executed.

Disadvantages

If every procedure is inlined at every one of its call sites, then the code size can potentially increase exponentially in relation to the size of the original program. This causes problems at various stages of the compilation cycle.

data flow analysis: Compilers are often optimized for dealing with human-generated source code, which has very different properties to deproceduralized code. Knuth describes an empirical study of human-generated programs, which are all "surprisingly simple" [Knu71]. A deproceduralized program consists of one long procedure, which has an extremely large number of variables in scope simultaneously. This may cause the data flow analysis to perform slowly, leading to longer compile times. Cooper et al [CHT92] reveal that some Fortran compilers define an upper limit on the number of variables allowed in data flow analysis, variables above this limit are not subject to data flow analysis.

code generation: A large number of simultaneously live variables causes pressure on the register allocator. Many registers will be spilled to memory. The procedure calling convention (in a non-deproceduralized program) forces the spillage of variables from caller procedures while the callee procedure is running. In a deproceduralized version of the program, the register allocator has to rely on heuristics to determine which registers to spill at each program point, and where live ranges should be split. The heuristics are often unreliable [Bri92].

execution: A deproceduralized program has a larger executable image size than a non-deproceduralized version of the same program. The extra amount of code can cause many instruction cache misses at runtime, which can degrade computational performance in comparison with non-deproceduralized version of program [CCCH93].

Advantages

However, deproceduralization can clarify data flow information, which should result in the following advantages at compile time.

1. It is straightforward to apply standard intraprocedural analysis to the complete (albeit very large) monolithic program, provided the compiler is able to cope with such large programs. No complex interprocedural analysis is required at all.

2. The analysis of deproceduralized code does not suffer from the imprecision of context-insensitivity [Ruf95a]. Each procedure body is inlined at each call site, so no procedure body has to be analysed in multiple contexts.

3. It is the simplest way to perform procedure specialization. The body of each inlined procedure can be specialized at each call site. Constant parameters can be propagated into the procedure body, for instance.

4. Similarly, the code surrounding the inlined procedure can also be specialized. For instance loop-invariant code can be hoisted from within a loop, even if the code motion crosses a procedure boundary.

There are also advantages at runtime. Inlined procedure bodies eliminate most of the stack frame overhead associated with standard calling conventions, for instance.

Previous Work

Deproceduralization has a long history. It is described as “in-line expansion” in the Dragon book [ASU86], where it is presented as a conceptually related technique to call-by-name parameter passing, and as thoroughly common knowledge to the compiler community.

Wegman and Zadeck [WZ91] discuss how constant propagation can be more effective on deproceduralized code. They describe a constant propagation algorithm that is combined with deproceduralization. Time and space savings are effected by combining these optimizations. If deproceduralization is done first, then some procedure calls are inlined, yet these calls may be shown by constant propagation to be not executable, in which case the irrelevant code is eliminated. More recently, the Vada [HFH$^+$02] and Nova [GB03] systems both employ deproceduralization to extend their intraprocedural analyses to whole program scope.

Cooper et al [CHT91] report that deproceduralization does not result in noticeably improved runtime performance. They study a series of small FORTRAN benchmarks compiled with (1980’s) industry-standard FORTRAN compilers. They conclude that language constraints and compiler engineering frailties caused the disappointing results, rather than any intrinsic problem with deproceduralization in general. More recent empirical studies show that deproceduralization improves runtime performance for Pascal [RG89b] and C [HC89, ASG97] programs.

Incorporating SSI

The original formulation of SSI is intraprocedural in scope. Thus, a deproceduralized monolithic program can obviously be expressed in SSI, with all the attendant benefits for analysis and transformation. However, it is interesting to consider whether the original program should be deproceduralized prior to SSI conversion, or instead whether each procedure from the original program should be converted to SSI prior to deproceduralization and then further transformed to reinstate the SSI property. The latter approach may be more efficient because, to restore SSI after inlining lots of SSI procedures, it is only necessary to deal with assignments due to parameter passing at call sites. Since the optimistic SSI construction algorithm is superlinear, it may be more efficient to transform many small procedures rather than one large procedure. Furthermore, if a procedure has several call sites, it is better to transform that procedure once (before deproceduralization), and perform systematic renaming to prevent virtual register name clashes in same scope, rather than reconstruct SSI for the inlined procedure at each of its call sites.

```
$r \leftarrow 1$
$x \leftarrow 5$
**while** $(x > 0)$ **do**
    $r \leftarrow r * x$
    $x \leftarrow x - 1$
**done**
**return** $r$
```

Figure 6.8: Factorial example program

This piece-wise, incremental SSI construction seems appealing. No-one has done any research in this area to date. It appears to be a promising avenue for future research.

6.4.2 Functionalization

SSA and SSI programs have the desirable property of *referential transparency*. This means that the value of an expression depends only on the value of its subexpressions and not on the order of evaluation or side-effects of other expressions. Referentially transparent programs are easier to analyse and transform. Referential transparency is automatically enforced by pure functional programming languages.

Since SSI is such a straightforward extension of SSA, it follows that algorithms for SSA can be quickly and naturally modified to handle SSI. Recall that Chapter 3 extended the standard SSA construction algorithm [CFR+91] to construct SSI instead. Similarly, Chapter 4 shows that the SSA conditional constant propagation algorithm [WZ91] has a natural analogue in SSI [Ana99], which produces more accurate results.

It is a well-known fact that SSA can be seen as a form of functional programming [App98b]. Inside every SSA program, there is a functional program waiting to be released. Therefore, one should not be surprised to discover that SSI can also be considered as a form of functional programming. Consider the program shown in Figure 6.8 which calculates the factorial of 5.

Figure 6.9 gives the CFG for this example program. Figure 6.10 gives the SSI program that is constructed from the CFG. This SSI program can be simply transformed into the functional program shown in Figure 6.11. The rest of this section describes this new transformation algorithm.

In the conversion from SSA to functional notation, a basic block `#n` that begins with one or more ϕ-functions is transformed into a function f_n. Jumps to such a basic block become tail calls to the corresponding function. The actual parameters of the tail calls are the source operands of the ϕ-functions. The formal parameters of the corresponding functions are the destination operands of the ϕ-functions.

In the conversion from SSI to functional notation, in addition to the above transformation, whenever a basic block ends with one or more σ-functions, then successor blocks `#p` and `#q` are transformed into functions f_p and f_q. Jumps to such successor blocks become tail calls to the corresponding functions. The actual parameters of the tail calls are the source operands of the σ-functions. The formal parameters of the corresponding functions are the relevant destination operands of the σ-functions. (Note the continued

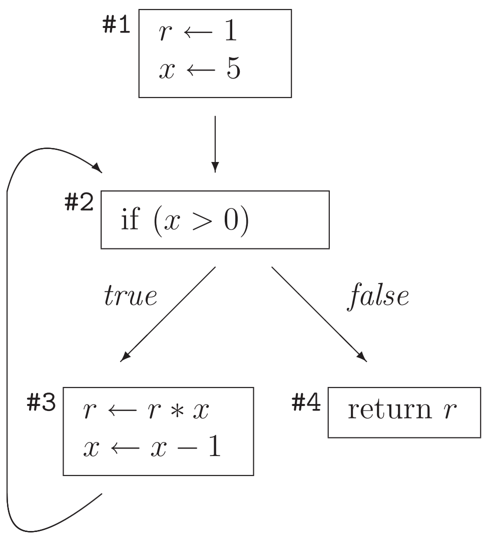
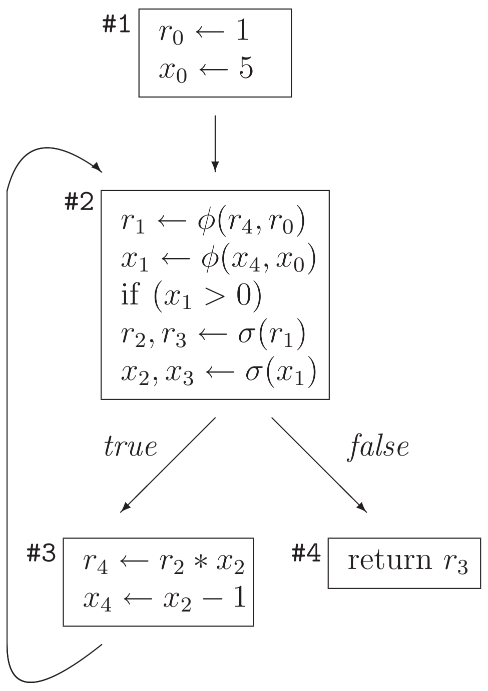


Figure 6.9: CFG for factorial program



#1
$r_0 \leftarrow 1$
$x_0 \leftarrow 5$
#2
$r_1 \leftarrow \phi(r_4, r_0)$
$x_1 \leftarrow \phi(x_4, x_0)$
if $(x_1 > 0)$
$r_2, r_3 \leftarrow \sigma(r_1)$
$x_2, x_3 \leftarrow \sigma(x_1)$
true
false
#3
$r_4 \leftarrow r_2 * x_2$
$x_4 \leftarrow x_2 - 1$
#4
return r_3

Figure 6.10: SSI for factorial program

```
let $r_0 = 1,\ x_0 = 5$
in
  let function $f_2(r_1, x_1)$ =
    let
      function $f_3(r_2, x_2)$ =
        let $r_4 = r_2 * x_2,\ x_4 = x_2 - 1$
        in
          $f_2(r_4, x_4)$
      function $f_4(r_3, x_3)$ =
        return $r_3$
    in
      if $(x_1 > 0)$
        then $f_3(r_1, x_1)$
        else $f_4(r_1, x_1)$
  in
    $f_2(r_0, x_0)$
```

Figure 6.11: Functional code for SSI factorial program

duality between ϕ- and σ-functions.)

Previous Work

To the best of our knowledge, no-one has attempted to transform SSI into a functional notation. Ananian [Ana99] gives an executable representation for SSI, but this is defined in terms of demand-driven operational semantics, and seems rather complicated.

Several people have noted a correspondence between programs in SSA and λ-calculus. Kelsey [Kel95] shows how to convert continuation passing style [App92] into SSA and vice versa. Appel [App98b] informally shows the correspondence between SSA and functional programming. He gives an algorithm [App98a] for translating SSA to functional intermediate representation. (Appel's algorithm is extended later in this section.)

Chakravarty et al [CKZ03] formalize a mapping from programs in SSA form to administrative normal form (ANF) [FSDF93]. ANF is a restricted form of λ-calculus. They also show how the standard SSA conditional constant propagation algorithm [WZ91] can be rephrased in terms of ANF programs.

More generally, the VDG representation [WCES94] has popularized the concept of representing all control flow as function calls.

Transformation

This section presents the algorithm that transforms SSI into functional notation.

It adopts a cut-down version of Appel's functional intermediate representation (FIR) [App98a]. The abstract syntax of FIR is given in Figure 6.12. FIR has the same expressive power as ANF [FSDF93]. FIR can easily be transformed to continuation passing style (CPS) [App92]. Expressions are broken down into primitive operations whose order of evaluation is specified. Every intermediate result is an explicitly named temporary. Every

atom	$\rightarrow$	c	constant integer
atom	$\rightarrow$	v	variable
exp	$\rightarrow$	**let** *fundefs* **in** *exp*	function declaration
exp	$\rightarrow$	**let** $\underline{v}$ = *atom* **in** *exp*	copy
exp	$\rightarrow$	**let** $\underline{v}$ = *binop*(*atom*,*atom*) **in** *exp*	arithmetic operator
exp	$\rightarrow$	**if** *atom relop atom* **then** *exp* **else** *exp*	conditional branch
exp	$\rightarrow$	*atom*(*args*)	tail call
exp	$\rightarrow$	**let** $\underline{v}$ = *atom*(*args*) **in** *exp*	non-tail call
exp	$\rightarrow$	**return** *atom*	return
args	$\rightarrow$		
args	$\rightarrow$	*atom args*	
fundefs	$\rightarrow$		
fundefs	$\rightarrow$	*fundefs* **function** $\underline{v}$(*formals*) = *exp*	
formals	$\rightarrow$		
formals	$\rightarrow$	$\underline{v}$ *formals*	
binop	$\rightarrow$	**plus** | **minus** | **mul** | . . .	
relop	$\rightarrow$	**eq** | **ne** | **lt** | . . .	

Figure 6.12: Functional intermediate representation

argument of an operator or function is an atom (variable or constant). As in SSA, SSI and λ-calculus, every variable has a single assignment (binding), and every use of that variable is within the scope of the binding. (In Figure 6.12, binding occurrences of variables are underlined.) No variable name can be used in more than one binding. Every binding of a variable has a scope within which all the uses of that variable must occur.

- For a variable bound by **let** $v = \ldots$ **in** *exp*, the scope of v is just *exp*.
- The scope of a function variable f_i bound in

  $$\begin{array}{l}\textbf{let function } f_1(\ldots) = exp_1 \ldots \\ \quad\textbf{function } f_k(\ldots) = exp_k \\ \textbf{in } exp\end{array}$$

  includes all the exp_j (to allow for mutually recursive functions) as well as *exp*.
- For a variable bound as the formal parameter of a function, the scope is the body of that function.

Any SSI program can be translated into FIR. Each basic block with more than one predecessor is transformed into a function. The formal parameters of that function are the destination operands of the ϕ-functions in that basic block. (If the block has no ϕ-functions then it is transformed into a parameterless function.) Similarly, each basic block that is the target of a conditional branch instruction is transformed into a function. The formal parameters of that function are the appropriate destination operands of the σ-functions in the preceding basic block (that is to say, the σ-functions that are associated with the conditional branch). We assume that the SSI program is in edge-split form—no basic block with multiple successors has an edge to a basic block with multiple predecessors. In particular this means that basic blocks that are the targets of a conditional branch can only have a single predecessor. (It should always be possible to transform an SSI program into edge-split form.)

If block f dominates block g, then the function for g will be nested inside the body of the function for f. Instead of jumping to a block which has been transformed into a function, a tail call replaces the jump. The actual parameters of the tail call will be the appropriate source operands of corresponding σ- or ϕ-functions. (Every conditional branch will dominate both its `then` and `else` blocks, in edge-split SSI.)

The algorithm for transforming SSI into FIR is given in Figure 6.13. It is based on Algorithm 19.20 from Appel's book [App98a]. `Translate()` ensures function definitions are correctly nested. `Statements()` outputs FIR code for each basic block. Appel's algorithm handles SSA, so we extend it to deal with SSI instead. In our algorithm lines of code that have been altered from Appel's original SSA-based algorithm are marked with a `!` and entirely new lines of code (to handle SSI-specific cases) are marked with a `+`. In the code for the `Statements()` function, $\oplus$ represents the general case for binary arithmetic operators and $<$ represents the general case for binary relational operators.

Optimistic versus Pessimistic

Recall from Chapter 3 that there are two different approaches to SSI construction—optimistic and pessimistic. Similarly, there appear to be optimistic and pessimistic approaches to the transformation into FIR. The pessimistic approach takes the original

```
  1: Translate($node$) =
  2:     let $C$ be the children of $node$ in the dominator tree
  3:     let $p_1, \ldots, p_n$ be the nodes of $C$ that have more than one predecessor
  4:     for $i \leftarrow 1$ to $n$
  5:         let $a_1, \ldots, a_k$ be the targets of $\phi$-functions in $p_i$ (possibly $k = 0$)
  6:         let $S_i$ = Translate($p_i$)
  7:         let $F_i$ = "function $f_{p_i}(a_1, \ldots, a_k) = S_i$"
+ 8:     let $s_1, \ldots, s_m$ be the nodes of $C$ that are the target of a conditional branch
+ 9:     for $i \leftarrow 1$ to $m$
+ 10:        let $q_i$ be the (unique) predecessor of $s_i$
+ 11:        let $a_1, \ldots, a_k$ be the targets (associated with $s_i$) of $\sigma$-functions in $q_i$
+                (possibly $k = 0$)
+ 12:        let $T_i$ = Translate($s_i$)
+ 13:        let $G_i$ = "function $f_{s_i}(a_1, \ldots, a_k) = T_i$"
! 14:    let $F = F_1 F_2 \ldots F_n G_1 G_2 \ldots G_m$
  15:    return Statements($node, 1, F$)

  16: Statements($node, j, F$) =
  17:    if there are $< j$ statements in $node$
  18:    then let $s$ be the successor of $node$
  19:        if $s$ has only one predecessor
  20:        then return Statements($s, 1, F$)
  21:        else $s$ has $m$ predecessors
  22:            suppose $node$ is the $i$th predecessor of $s$
  23:            suppose the $\phi$-functions in $s$ are
                     $a_1 \leftarrow \phi(a_1 1, \ldots, a_1 m)$, ...
                     $a_k \leftarrow \phi(a_k 1, \ldots, a_k m)$
  24:            return "let $F$ in $f_s(a_1 i, \ldots, a_k i)$"
  25:    else if the $j$th statement of $node$ is a $\phi$-function
  26:        then return Statements($node, j + 1, F$)
+ 27:    else if the $j$th statement of $node$ is a $\sigma$-function
+ 28:        then return Statements($node, j + 1, F$)
  29:    else if the $j$th statement of $node$ is "return $a$"
  30:        then return "let $F$ in return $a$"
  31:    else if the $j$th statement of $node$ is $a \leftarrow b \oplus c$
  32:        then let $S$ = Statements($node, j + 1, F$)
  33:            return "let $a = b \oplus c$ in $S$"
  34:    else if the $j$th statement of $node$ is $a \leftarrow b$
  35:        then let $S$ = Statements($node, j + 1, F$)
  36:            return "let $a = b$ in $S$"
  37:    else if the $j$th statement of $node$ is "if $a < b$ then goto $s_1$ else goto $s_2$"
  38:        then since this is edge-split SSI form
  39:        assume $s_1$ and $s_2$ each has only one predecessor
! 40:        let $a_1, \ldots, a_k$ be
!                the source operands of $\sigma$-functions in $node$ (possibly $k = 0$)
! 41:        return "let $F$ in if $a < b$ then $f_{s_1}(a_1, \ldots, a_k)$ else $f_{s_2}(a_1, \ldots, a_k)$"
```

Figure 6.13: Algorithm that transforms SSI to functional intermediate representation

CFG and converts each basic block into a top-level function, with tail calls to appropriate successor functions. Each generated top-level function has a formal parameter for every program variable, and each function call site has an actual parameter for every program variable. Appel [App98b] refers to this as the “really crude approach.” Useless parameters may be identified and eliminated by means of liveness and other data flow information. The necessary parameters for each functional block should be those variables which are live at each corresponding basic block boundary in the original program. This makes sense since SSI implicitly encodes liveness information (Section 4.4).

The optimistic approach is exactly as given in the algorithm above. It can be explained in the following manner. It uses the CFG dominance relations to determine how the functional blocks should be nested. (Nesting is required in order for functional blocks to use variables declared in outer scope.) Then it applies standard lambda lifting techniques [Joh85] to generate the appropriate parameters for each functional block.

Converting Functional Programs back to SSI

It is possible to transform an arbitrary program p expressed in FIR into SSI, simply by treating p as an imperative program. (Let-bound atomic variables become mutable virtual registers and function applications become procedure calls.) Standard SSI construction techniques can then be applied to the imperative program.

However, suppose that a program p_{SSI} in SSI has been transformed into a program p_{func} in FIR. This section addresses the concept of recovering p_{SSI} from p_{func}.

p_{func} is in SSA, since each let-bound variable is only assigned a value at one program point. However p_{func} is not in SSI, since the same parameters are supplied to the tail calls on either side of an `if` statement. (Recall that these parameters correspond to the source parameters of the σ-functions associated with this conditional branch in p_{SSI}.) The simplest way to transform p_{func} into a valid SSI program, say p'_{SSI}, is to add σ-functions at each `if` statement, and rename the parameters of the tail calls accordingly. There is a drawback with this approach however. Now imagine converting p'_{SSI} into FIR using our algorithm. There would be an additional layer of function calls at the `if` statements, because of the extra σ-functions. Admittedly these extra function calls could be removed by limited β-contraction, but it is embarrassing to admit that converting from SSI to FIR and back to SSI (ad infinitum) does not reach a fixed point. In fact this is a diverging computation.

The problem is that the σ-functions are already encoded as function calls in p_{func} but we do not recover this information. We insert extra σ-functions instead. One way to avoid this would be to inline (β-contract) all functions in p_{func} that are only called from one call site (this includes all functions that originated from σ-functions). If this transformation is performed prior to the insertion of σ-functions, then the problem of an extra layer of function call indirection does not arise.

Kelsey [Kel95] gives a method for recovering ϕ-functions from functional programs. We should be able to apply similar techniques to p_{func}. Thus it should be possible to recover (something resembling) p_{SSI} from p_{func}.

Motivation

This section briefly considers why the transformation from SSI into FIR may be of value.

It effectively makes SSI interprocedural in scope, by abstracting all control flow into function calls. This functionalization transformation entirely eliminates the artificial distinction between intraprocedural and interprocedural control flow.

Typed functional languages may be useful as compiler IRs for imperative languages. There has recently been a great deal of research effort in this area, with systems such as typed assembly language [MWCG99], proof carrying code [Nec97, App01] and the value dependence graph [WCES94]. SSA and SSI fit neatly into this category, since they can be seen from a functional perspective, and they are most amenable to high-level type inference techniques [Myc99]. The creators of similar typed functional IRs for Java bytecode, such as λJVM [LTS01] and GRAIL [BMS03], comment that a functional representation makes both verification and analysis straightforward. It is useful for reasoning about program properties, such as security and resource consumption guarantees. Functional notations are also well-suited for translation into lower-level program representations.

Algorithms on functional IRs can often be more rigorously defined [CKZ03] and proved correct. It would be interesting to compare existing SSA or SSI data flow analyses with the equivalent analyses in the functional paradigm, perhaps to discover similarities and differences. Such cross-community experience is often instructive to one of the parties, if not both.

Finally we note that standard SSI is not an executable representation per se. It is restricted in the same manner as original SSA, since ϕ- and σ-functions require some kind of runtime support to determine which source operand value to assign to which destination operand. Ananian has concocted an operational semantics for an extended version of SSI [Ana99], however this is quite complex and unwieldy to use. On the other hand, functional programs are intuitive and have a familiar execution semantics. Some simple SSI programs have been successfully translated into Haskell and ML code, using the transformation algorithm of section 6.4.2. For instance, Figure 6.14 shows the dynamic data flow graph [NM03] of three Haskell factorial functions that each compute the factorial of 5 (the answer is 120). The three values are then added together (the sum total is 360). The left portion of the graph represents a standard Haskell iterative definition of the factorial function:

```
faci 0 acc = acc
faci n acc = fac1 (n-1) (acc*n)
```

The middle portion of the graph represents a standard Haskell recursive definition of the factorial function:

```
facr 0 = 1
facr n = n * facr (n-1)
```

The right portion of the graph represents the Haskell version of the functional program from Figure 6.11 which is the transformation of the SSI program from Figure 6.10. Note that the right portion of the dynamic data flow graph has exactly the same shape as the left portion, which reveals that both are computing factorials iteratively, so the transformation from imperative to functional style does not alter the data flow behaviour of the program at all.

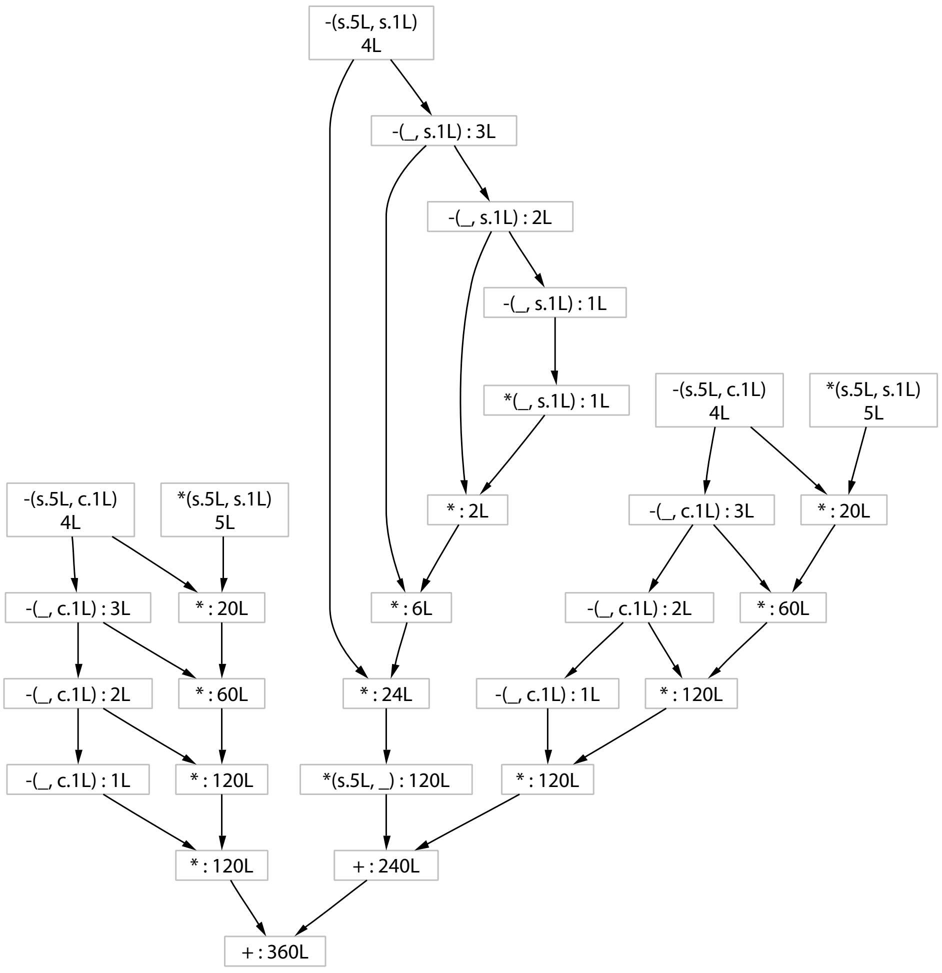


Figure 6.14: Dynamic data flow graph for three factorial(5) functions

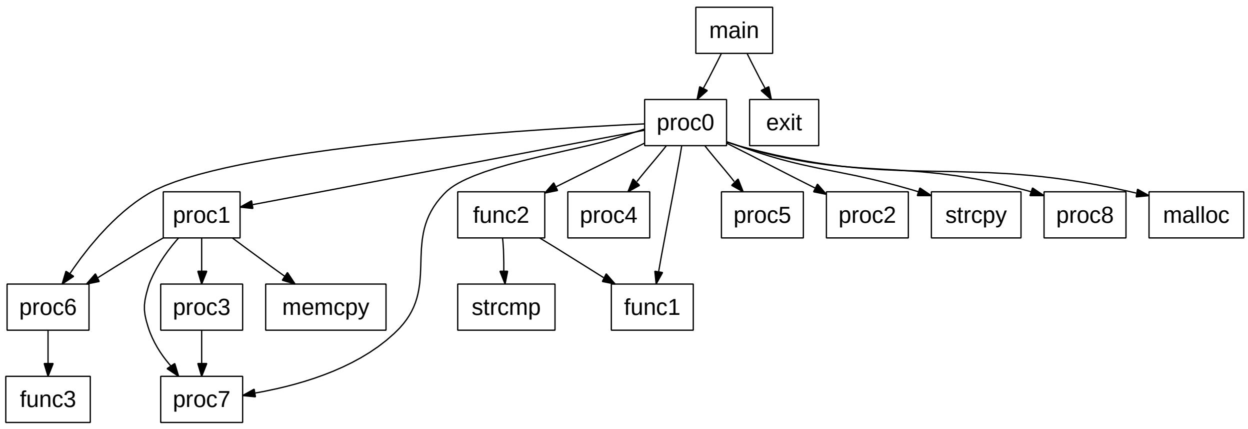


Figure 6.15: Dhrystone call graph (courtesy of AbsInt)

Further Remarks

Compilers for functional programming languages (such as the Glasgow Haskell compiler) often translate their intermediate form into an imperative language (such as C) which is then compiled to machine code. This seems rather wasteful, since the C compiler (if it uses a functional representation as its intermediate form) will attempt to reconstruct the functional program which has been carelessly thrown away by the functional compiler backend. Perhaps this functional information should be preserved throughout the compilation cycle.

The transformation algorithm presented above could formalized, in the same manner as Appel's original work on SSA [App98b, App98a] has been formalized [CKZ03]. The next step is to translate existing SSI analysis algorithms to this new functional framework. Also, it is necessary to consider how to take advantage of FIR in order to devise new analyses and optimizations. On a different note, SSA and SSI are just two members of a large family of virtual register renaming schemes, as the next chapter explains. It would be interesting to see if every scheme in the family could be converted to a functional notation like FIR, using the same general techniques outlined in this section.

6.4.3 Empirical Comparison

In order to compare these two control flow refactoring approaches to interprocedural analysis, I decided to test both transformations (functionalization and deproceduralization) on a benchmark test. The Dhrystone benchmark [Wei84, Wei88] is a simple integer and string processing benchmark. The original version is written in C. I ported this to the functional programming language OCaml, preserving as much as possible of the original semantic style. Dhrystone is a suitable program for comparing these two transformations, since it has a computationally intensive inner loop. Also it has an acyclic call graph (Figure 6.15), which means that all procedures may be inlined at all call sites.

The OCaml code was functionalized, using the transformation described in Section 6.4.2. The C code was deproceduralized, using the transformation described in Section 6.4.1. Figure 6.16 shows the sizes of the original and transformed source code, in lines

Language	Original/loc	Functionalized/loc	Deproceduralized/loc
OCaml	481	704	-
C	568	-	572

Figure 6.16: Sizes for Dhrystone benchmark source code

Language	Original/s	Functionalized/s	Deproceduralized/s
OCaml	44	95	-
C	33	-	22

Figure 6.17: Times for Dhrystone benchmark runs

of code (loc). Figure 6.17 summarizes the results obtained. OCaml code was compiled using the optimizing `ocamlopt` x86 native code compiler (v3.07+2). C code was compiled using GCC (v3.2.2) with the -O2 optimization level. The times given are in seconds, and they measure the time taken for 10^8 iterations of the Dhrystone loop. All times are the arithmetic mean of three runs on a lightly loaded AMD Athlon 1.4GHz x86 Linux machine.

The results show that `ocamlopt` and GCC produce optimized code of comparable quality. The original version of the OCaml Dhrystone benchmark takes 33% more time than the C version. However, the functionalized version takes more than twice as long as the original version. The functionalization transformation (which demands quality interprocedural optimization) seems to hinder the OCaml optimizer from working. Procedures really do hinder data flow analysis, as Section 6.1 suggested. The deproceduralized version takes 33% less time than the C standard code. Thus the deproceduralization transformation enables further optimizations.

6.5 Concluding Remarks

There is no definitive technique for performing interprocedural analysis. As Chapter 4 concluded, the appropriate IR must be chosen by taking issues like accuracy and efficiency into consideration. This chapter has reviewed the four main interprocedural IRs. It has shown that SSI can be accommodated in each of these IRs. A new algorithm for functionalization has been presented.

The next chapter reviews how SSI may be extended to handle other high-level features as well as procedures. Also, it argues that SSI is a particular instance from a general family of IRs.

Chapter 7

Beyond SSI

SSI and SSA are two members of a family of IRs known as virtual register renaming schemes. There are many other IRs that belong to the same family.

7.1 About this Chapter

7.1.1 Objectives

SSI can be seen as an extension of SSA. However, since the introduction of SSA in the late 1980's, there have been a remarkable number of independently proposed extensions. This chapter aims to place SSI in context, by reviewing many of the other SSA extensions and setting them in an orderly framework. Gaps in the framework may lead to the proposal of new SSA extensions.

7.1.2 Outline

Section 7.2 reviews many SSA extensions proposed in the literature. It divides these extensions into two categories: general and feature-specific. Section 7.2.1 goes through many general extensions in detail. Section 7.2.2 briefly reviews a selection of feature-specific extensions. Section 7.3 re-explores the general extensions, and shows how these may be characterized formally as *virtual register renaming schemes* (VRRSs), using a specification language. Section 7.4 presents an empirical investigation into how the number of virtual register names varies with VRRSs. Section 7.5 reviews the small amount of previous work in this area. Section 7.6 discusses possible future research directions.

7.1.3 Contributions

This chapter makes four key contributions.

1. Section 7.2 provides a systematic classification of IRs that extend the original version of SSA. To the best of our knowledge, there is no existing classification.

2. Section 7.2.1 formulates static single mention form (SSM), which is a new general SSA extension arising from the consideration of gaps in the systematic classification space.

3. This chapter develops the notion of *virtual register renaming schemes*, They are characterized both informally and formally, and examined empirically. The VRRS concept is a significant generalization of the virtual register renaming idea introduced by SSA.

4. Since this is a new area of research, the chapter points out possible future directions.

7.2 SSA Extensions

Since the introduction of SSA, various extensions have been proposed. This chapter distinguishes between *general* extensions and *feature-specific* extensions. General extensions modify the SSA renaming convention to enable different kinds of sparse data flow analysis. For instance, SSI is a general extension of SSA. Section 7.2.1 reviews a number of general SSA extensions. Feature-specific extensions modify the SSA representation to model high-level language or low-level architectural features. Section 7.2.2 briefly reviews a number of feature-specific SSA extensions.

There are two main reasons for representing a program in an extended version of SSA.

1. Ease of analysis: SSA-based virtual register renaming enables sparse data flow analysis for a certain class of forward data flow problems. Sparse data flow analysis is often more efficient than classical data flow analysis [CCF91]. General SSA extensions that perform different degrees of renaming may enable sparse solutions to other data flow problems. Chapter 4 explored this area in detail. Also, features-specific SSA extensions that incorporate high-level features permit more powerful analyses that leverage the extra information.

2. Ease of code generation: Virtual register renaming is equivalent to live range splitting, which has a major effect on the performance of register allocation. General SSA extensions that perform more renaming are likely to enable better register allocation. Also, feature-specific SSA extensions that incorporate low-level features permit more appropriate code generation for specialized target architectures.

7.2.1 General Extensions

The original version of SSA [CFR+91] assumes that programs are represented as intraprocedural CFGs. Primitive instructions evaluate arithmetic expressions (containing only scalar virtual register values and constants) and use the results either to determine conditional branch outcomes or to assign values to some virtual registers. This section continues to assume the same simple, scalar, intraprocedural model. Support for high-level features such as pointers, arrays, objects and procedures will be considered in Section 7.2.2. The general SSA extensions should be entirely orthogonal to the feature-specific extensions, so any general extension could potentially be combined with any feature-specific extension, in the same way that original SSA is extended to create a feature-specific SSA extension.

Figure 7.1 summarizes the general SSA extensions to be considered in this section. The remainder of this section goes through each IR in greater detail. Note that this

IR	Canonical reference	Pseudo-function	Naming convention
Static single assignment	[CFR+91]	ϕ	so each virtual register is the target of exactly one assignment statement in the program source code
Static single information	[Ana99]	ϕ, σ	so each virtual register is the target of exactly one assignment statement, and so each virtual register is used in exactly one conditional context, in the program source code
Static single use	[LCK+98]	Λ	so each virtual register is used exactly once in the program source code
Linear naming	[Baw93]	clone	so each virtual register is used at most once during program execution
Dynamic single assignment	[Fea91]	add array dimensions	so each virtual register is defined at most once during program execution
Static single mention	n/a	clone	so each virtual register is mentioned exactly once in the program source code, excluding mentions in clone functions
Def-use-use-def webs	[Muc97]	n/a	so each virtual register has exactly one live range in the program source code
Register allocated code	[Cha82]	spill code	so there is a fixed, platform-dependent upper limit on the number of virtual registers that are live at any point in the program source code and during program execution

Figure 7.1: Table summarizing some general SSA extensions

section treats original SSA as the 'base case' general SSA extension, for the purposes of comparison with other general extensions.

Static Single Assignment Form

SSA was thoroughly reviewed in Chapter 2. Recall that a program is defined to be in SSA if each virtual register is the target of exactly one assignment statement. From this property it is possible to derive the virtual register renaming criterion: a virtual register v should be renamed at each program point where v is the target of an assignment statement. The renaming criterion may be used to convert CFG programs with unconstrained naming conventions into SSA programs.

Prior to SSA conversion, a program may have multiple definitions of a virtual register that reach a control flow merge point via different branches. When converting to SSA, such values have to be merged into a new virtual register at the control flow merge point. These merges are called ϕ-functions. A ϕ-function has n source operands if it is placed at an n-way control flow merge, and one destination operand. A ϕ-function assigns the value of one of the source operands to the destination operand, dependent on control flow. This is not an executable semantics. Extra 'runtime support' is needed, as [CFR+91] suggests, in order to specify explicitly how the ϕ-function chooses the correct source operand. Several executable versions of SSA have been proposed, such as the program dependence web [BMO90] and gated single assignment form [TP95]. However execution is not a major application of SSA. SSA is generally used for static analysis. ϕ-functions are transformed into register move instructions at the code generation phase. Often sophisticated register allocation techniques eliminate many of these move instructions.

SSA is a popular IR for data flow analysis. Examples of SSA-based analysis include:

- constant propagation [WZ91], and
- global value numbering [RWZ88, AWZ88], and
- partial redundancy elimination [KCL+99], and
- type-based decompilation [Myc99].

SSA also benefits register allocation. SSA renaming ensures that each virtual register has only one live range, and there is only one definition within each live range. This amount of live range splitting allows great flexibility during register allocation. Empirical studies show that both classical register allocation via graph colouring [Bri92] and the more recent linear scan register allocation [SS03] can be improved using SSA.

Static Single Information Form

SSI [Ana99] has been the main focus of this dissertation so far. It was defined at length in Chapter 3. SSI renames virtual registers at assignments and also at certain branch points. This is exactly the same as Plevyak's SSU [Ple96] and similar to predicated SSA [CSC+99] and extended SSA [BGS00].

SSI has two different types of pseudo-definition function: ϕ and σ. The ϕ-functions have identical semantics as those in SSA. The σ-functions have dual semantics to ϕ-functions. A σ-function has n destination operands if it is placed at an n-way control flow split point, and one source operand. It assigns the value of its source operand to one of its destination operands, dependent on control flow.

SSI is useful for both forward, backward and bidirectional sparse predicated data flow analysis, as Chapter 4 demonstrated. Other common SSI analyses include:

- sparse predicated conditional constant propagation [Ana99], and
- symbolic bounds analysis [RR00], and
- bitwidth analysis [SBA00], and
- data size optimizations [AR03].

There has been no recent work on using SSI for register allocation. However Briggs [Bri92] assesses the effect of live range splitting (effectively virtual register renaming) at both forward and reverse dominance frontiers, using ϕ- and 'reverse' ϕ-functions. This combination seems similar to SSI, although Briggs conducted his investigations before SSI was invented. Unfortunately Briggs gives neither a formal nor an algorithmic characterization of his live range splitting transformation. Briggs reports that the effect of this live range splitting on register allocation is "...mediocre. Out of 70 routines, there were difference in 42 cases: 13 improvements and 29 degradations." Briggs' degradations are due to excess register copy and spill operations, which his code generator is unable to remove.

Static Single Use Form

Confusingly, there are three independent (and inconsistent) IRs known as static single use form (SSU) [Ple96, LCK+98, GB03].

Plevyak's SSU [Ple96] creates a unique virtual register for each assignment "under each set of local control flow conditions," which amounts to renaming virtual registers when they are defined and when they are used in different conditional contexts. This is the same as SSI, which is described above. The remainder of this section ignores Plevyak's formulation of SSU.

Lo et al [LCK+98] develop SSU in the context of partial redundancy elimination for memory write operations, also known as partial dead store elimination. They say that SSU is the dual of SSA. However they restrict SSU renaming to virtual registers that are the operands of memory operations (load and store instructions). Each load and store instruction has an associated virtual register that holds the data to be transferred. Only these virtual registers are candidates for SSU renaming. The SSU construction algorithm assigns a new name to each virtual register associated with a load instruction. This new name is also applied to all the virtual registers of all store instructions that reach that load instruction. This is the basis of the SSUPRE algorithm, which is the dual of the well-known SSAPRE algorithm [KCL+99].

The following discussion considers the more general case, where all virtual registers are candidates for SSU renaming. However, it follows the SSU notational conventions established by Lo et al [LCK+98]. In SSU, each use of a virtual register establishes a new name. Thus no virtual register appears as a source operand of more than one instruction. Each use of virtual register v postdominates all the definitions of v. No definition of a

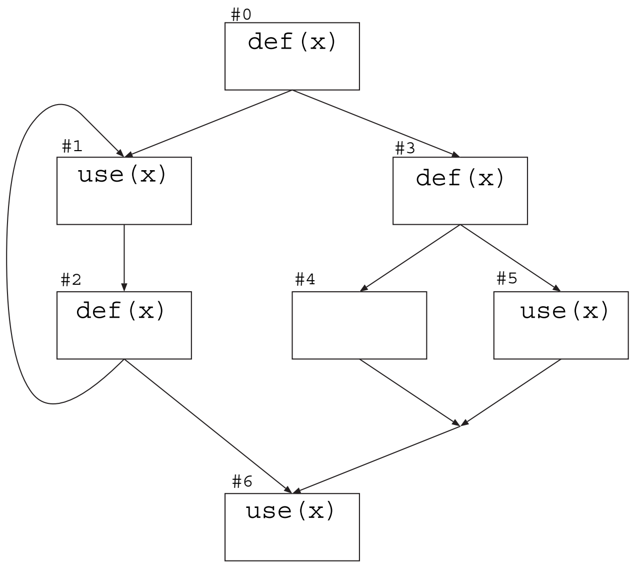


Figure 7.2: Example program before SSU renaming

virtual register reaches more than one use. It may be necessary to insert extra definitions into the program in order to achieve this single-reaching-definition property, when converting from an arbitrary CFG program into SSU.

The SSU factoring operator is the Λ-function, which is the dual of the SSA ϕ-function. SSU Λ-functions share the same semantics as SSI σ-functions. A Λ-function is treated as a use of its source virtual register v, so it always establishes a new name for v as well. Λ-functions are inserted at the iterated reverse dominance frontiers of uses. This is exactly the dual of ϕ-function placement in SSA.

So the transformation from CFG to SSU takes place in two steps.

1. Λ-function placement

2. virtual register renaming

Virtual registers must be renamed to satisfy the SSU constraint that each virtual register is only used once, and the transformed program retains the same observable runtime behaviour as the original program. Note that this SSU variant (renaming applied to all virtual registers) is not used as an IR for any extant data flow analysis. Generally, SSU renaming is only applied to certain virtual registers with specific properties, often associated with memory store operations.

Figure 7.2 shows an example CFG program. It only gives abstract definitions and uses of virtual register `x`, in order to simplify the presentation. Figure 7.3 shows the same program after conversion to SSU. Note that Λ-functions have been inserted at the reverse dominance frontiers of the use instructions. Also note that an extra definition of virtual register `x0` is required at the end of node `#5`, in order to reach the use of `x0` at node `#6`.

George and Blume [GB03] describe an independently formulated version of SSU. Their SSU property is that any virtual register use occurring as a source operand in a store operation is the only use of that virtual register in the entire program. Again, this variant of SSU only renames virtual registers that participate in store instructions. SSU can be

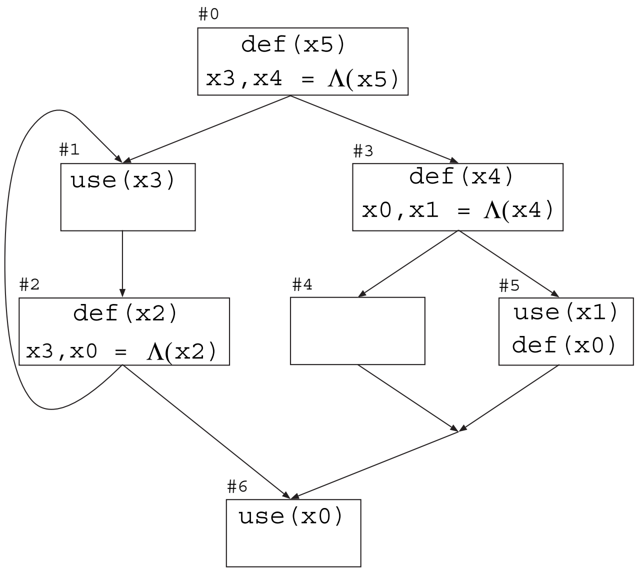


Figure 7.3: Example program after SSU renaming

generated simply by making sufficiently many copies (clones) of each virtual register. The program is assumed to be in SSA form already, so no virtual register is ever redefined after its creation. In this way originals and clones are guaranteed to be consistent. SSU is used to solve the combined register and bank assignment problem [GB03] for code generation on Intel IXP network processors [Int03]. IXP processors have distinct banks of register files. Different banks have different capabilities for accessing memory. IXP processors have restricted data paths, and a register whose value is to be stored out into memory cannot subsequently participate in general purpose instructions until the store instruction has completed. These restrictions ensure that bank assignment is a non-trivial problem.

Stoltz's λ-graph for backward data flow analysis [Sto95] has much in common with SSU. Stoltz inserts λ-functions (sic) at control flow split points, and renames virtual registers at both definitions and uses. The λ-graph representation is used for computing virtual register liveness and expression anticipatibility, both in a sparse manner.

Linear Naming Form

A virtual register is *linear* if it is used at most once dynamically. Note that linearity is concerned with the number of times each name must be resolved in its scope during execution, rather than any static property of the program. A program is in linear naming form (LN) if each virtual register is used no more than once during execution. So we could refer to linear naming as *dynamic single use*. Explicit copies are required to manage nonlinear naming. These copy operations correspond to clone operations in other renaming schemes. Bawden [Baw93] introduces LN. He compiles Scheme programs to a linear graph reduction model. LN appears to be mostly applied to functional programming languages. However Baker [Bak95] argues that mainstream programming languages should have 'use-once' variables.

The linearity constraint has been shown to be useful at a higher level of abstraction than virtual registers. Bawden [Baw93] uses linearity to support cheap cross-network

```
for (i = 0; i < 10; i++)
  a = i * i;
```

Figure 7.4: Example program (not in DSA)

```
for (i = 0; i < 10; i++)
  a[i] = i * i;
```

Figure 7.5: Example program after conversion to DSA

references and portable data structures in distributed systems. Baker [Bak95] shows that linear references to objects improve garbage collection performance and avoid costly synchronization mechanisms. More recent work [ESM04] uses linearity to defer decisions about parameter passing conventions (by value or by reference) from design time (as in conventional imperative languages) until compile time.

Whether linearity is a sensible constraint to impose at the level of virtual register names remains an open question. In general, many variable reuses will lead to explicit copy operations to preserve linearity. This makes linearity an expensive feature to support, in terms of construction and analysis costs in both computational time and space. Standard imperative loop constructs breaks the linear naming convention. Such code has to be replaced by tail recursive function calls. Weise et al [WCES94] claim this functional style of code is much easier to analyse. Johnson [Joh04] claims entirely the opposite. It appears that such judgements are largely subjective.

Dynamic Single Assignment Form

A program is in dynamic single assignment form (DSA) [Fea91] when each virtual register is assigned at most one value during execution. Generally, DSA is applied to programs containing both scalar and array computations, but this section only considers the cut-down scalar version of DSA for simplicity. Virtual registers may not be defined more than once at runtime in DSA. These are eliminated by increasing the dimension of each virtual register as required, effectively expanding it into an array with one element for each potential definition. Rau refers to such arrays as "expanded virtual registers" [Rau96]. He describes them as "a fiction that cannot be translated into hardware." Expanded virtual registers are the *clone operations* required to support DSA. Figure 7.4 shows a typical program where virtual register `a` is assigned more than one value at runtime. Figure 7.5 shows that same program after DSA conversion. Note that virtual register `a` has been transformed into an array, so each array element has a single definition at runtime.

DSA prevents multiple assignments since they often obscure the data flow behaviour of the program. DSA is used in parallelization optimizations [Fea91] and in memory hierarchy organization for embedded systems [VJB$^+$03].

There is no DSA construction algorithm for arbitrary CFGs. Vanbroekhoven et al [VJB$^+$03] give an algorithm that transforms a restricted class of SSA programs into DSA. Offner and Knobe [OK03] describe weak DSA, in which a virtual register can be defined more than once so long as the same value is assigned each time. They develop a conversion algorithm from CFG to weak DSA via Array SSA (a feature-specific SSA extension

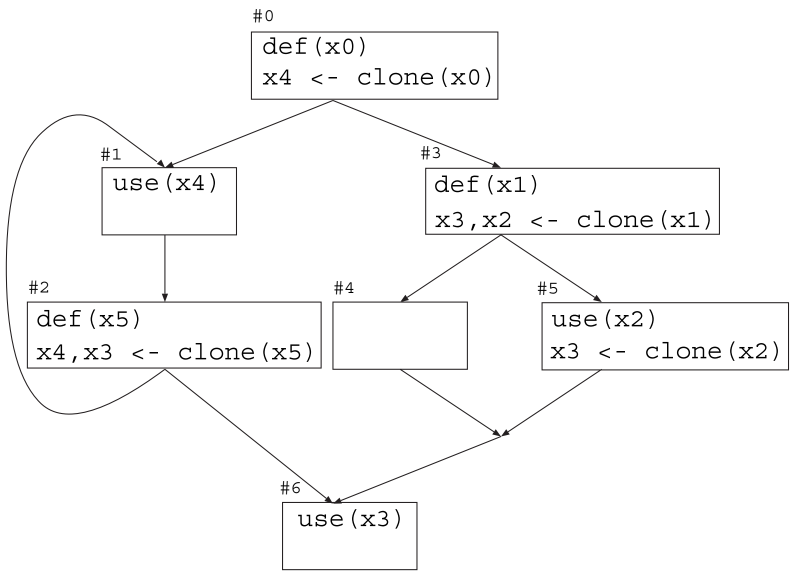


Figure 7.6: Example program after SSM renaming

mentioned in Section 7.2.2).

Static Single Mention Form

Static single mention form (SSM) is a new IR proposed in this dissertation. It has emerged as a result of considering conceptual gaps in the family of general SSA extensions.

Recall that an instruction i *mentions* a virtual register v if i may define or use the value of v. The SSM property is that each virtual register mention has a distinct name. This requires explicit transfer operations to handle the flow of data between static single mentions. (Such transfer functions can be provided by clone operations, which do not necessarily count as virtual register mentions.) Figure 7.6 shows an SSM version of the example program from Figure 7.2. Note that each `def` and `use` has a different subscript in the SSM version, and clone operations are inserted at appropriate points so that the values that flow through the program are unchanged from the original.

There are a number of ways to transform a program into SSM. The differences hinge on the clone operations, which may be inserted early (immediately after virtual register definitions) or late (just before virtual register uses). For late cloning, the clone operation source operand may be the closest reaching mention (definition or use), or the closest reaching definition. Early cloning SSM effectively assigns a unique name to each link in a def-use chain from the original program. With late cloning SSM when closest reaching mentions are used as clone operation source operands, live ranges are split at each mention so as to avoid overlap. Of course, other cloning policies are possible. The examples given above are extremal.

Extended array SSA form [FKS00] is effectively SSM for array variables. An array is renamed each time an element is defined (using a $d\phi$-function). An array is renamed each time an element is read (using a $u\phi$-function). These array names are used during data flow analysis for redundant load identification. In classical flow-sensitive analysis, a bit vector of virtual register properties is associated with each program point. In SSM, virtual register properties can be related directly to each virtual register mention (recall definition of sparseness from Section 4.2.7) since each virtual register has a unique program point where it is mentioned.

SSM can be seen as a pessimistic approach to register allocation. The SSM construction algorithm operates like a register allocation algorithm where every virtual register clashes with every other virtual register, so every virtual register mention is re-coloured to have its own unique colour. The clone operations ensure that the correct values flow between virtual registers. These are the minimal constraints on the code. Conventional register allocation may then coalesce virtual registers as guided by clone operations, liveness information and suitable heuristics.

SSM could be useful as a normalized form for solving the register bank assignment problem. In some architectures, registers are assigned to different register banks, and instructions can only take their operands from certain banks. Thus a value may have to be copied from one bank to another. If each mention of a value has a different virtual register then, at register allocation time, the program is in a normalized form that makes no assumptions about register bank assignments. Thus it is possible to generate bank assignments at the same time as register allocation and code generation. Existing techniques to resolve register bank conflicts [ZP03, GB03] for Intel network processors seem less elegant. This topic requires more investigation.

WEB

Recall from Section 2.6.1 that a def-use-use-def-web is the maximal union of def-use chains that share a common use. Each web is treated as an atomic unit in the register allocation phase. Generally all virtual register mentions within the web will be allocated to the same physical location. The WEB IR encapsulates existing live ranges for virtual registers in the program. Strictly, each web includes only those instructions that define or use the subject virtual register of that web, whereas the live range includes all intermediate instructions on control flow paths between definitions and uses in the web. One interesting feature of WEB is that no clone operations are required. WEB construction is one of the least intrusive transformations that renames virtual registers.

Register Allocated Code

So far, most of the general SSA extensions presented have increased the number of virtual registers when compared with equivalent CFG programs. However it is sometimes desirable to reduce the number of virtual registers, using the same mechanism of renaming. For instance, consider register allocation, which attempts to map M virtual registers onto N physical registers, usually when $M >> N$. The constraint to be satisfied is that no more than N virtual registers are live at any point in the program. The semantic machinery required to achieve this property is *spill code*, which transfers values out to memory so that they do not need to be held in physical registers. When spilled values are required, they must be loaded back from memory. Spill operations are modelled by two pseudo-functions.

1. $mem \leftarrow \text{spill}(reg)$
2. $reg \leftarrow \text{unspill}(mem)$

Spilled virtual registers are not taken into consideration when calculating the N live virtual registers at each point. Thus the transformation to register allocated code consists of inserting sufficient spill operations to satisfy the N-live-registers constraint. The intricate details of the transformation concern how to avoid inserting superfluous spill operations that cause the generated code to be inefficient. There are many different register allocation techniques. Briggs [Bri92] provides a good overview. Code c that has been register allocated should be dynamically equivalent to that same code c' prior to register allocation. This means that physical registers (or memory locations) in c and the corresponding virtual registers in c' should contain same values at any point during program execution.

Register allocated code is of little use for data flow analysis, since unrelated values are artificially connected by sharing the same physical register. Therefore, in order to analyse register allocated code, the first step is to normalize the naming conventions. Mycroft [Myc99] advocates transforming register allocated code into SSA, which is more amenable to data flow analysis. Mycroft states that SSA and register allocated code are inverse to each other, since register allocated code reduces the number of virtual registers as much as possible by combining non-overlapping live ranges whereas SSA has a distinct live ranges for each definition, with one virtual register name per live range. In fact, the situation is more general than Mycroft's statement: Renaming schemes that increase the number of names are, in effect, performing live range splitting to some degree. Renaming schemes that reduce the number of names are, in effect, combining non-overlapping live ranges of virtual registers.

Preliminary Classification of Renaming Schemes

At first inspection, it may seem that the IRs presented above are a haphazard selection of vaguely similar forms. Section 7.3 introduces a more formal classification. However at this point, Figure 7.7 shows how some of these general SSA extensions are related, by means of a lattice diagram. Note that this lattice diagram only includes general SSA extensions whose renaming criterion may be determined entirely statically. DSA and LN have renaming criteria which specify dynamic properties, which makes them more difficult to compare in the same framework. Also register allocated code does not seem to fit neatly into this scheme. $x \sqsubset y$ (for instance, SSI $\sqsubset$ SSA) means that a program in form x also satisfies all the properties of y, although it may have more than sufficient clone operations. Note that CFG is the standard control flow graph representation. It is assumed that all general SSA extensions are conforming CFGs.

7.2.2 Feature-Specific Extensions

So far this dissertation has mostly considered intraprocedural representations of simple monolithic programs. This allows a clear presentation of the main ideas, without the distractions of other programming language features. However, there are a number of SSA extensions that elegantly handle both low-level architectural features and high-level programming constructs. These *feature-specific SSA extensions* have flourished rapidly since the original formulation of monolithic SSA. In the same way that SSA can be extended to handle any of these features, so too any general SSA extension can be similarly extended. This is possible if each feature-specific extension is somehow adaptable to the alternative renaming criteria of other general SSA extensions. Feature-specific extensions

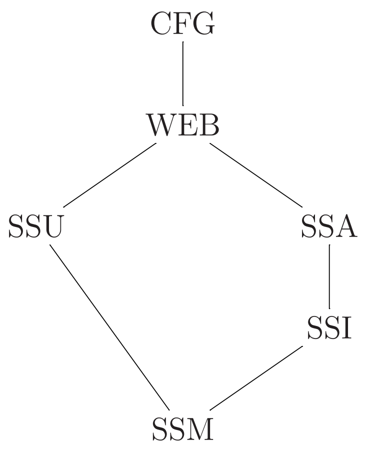


Figure 7.7: Lattice diagram of renaming schemes

may incorporate extra pseudo-definitions. If these are related to ϕ-functions, it may be possible to create general versions that correspond to alternative clone operations. On the other hand, if feature-specific pseudo-definitions are entirely independent of ϕ-functions, then it should be possible to just retain an unchanged semantics in an alternative general SSA extension. The remainder of this section briefly examines a selection of these feature-specific extensions.

Low-Level Architectural Features

Leung and George [LG99] present a simple scheme for converting between native machine code and SSA, such that references to actual physical registers are preserved. This allows standard SSA algorithms to perform analysis and optimization of native machine code. Von Ronne et al [vWF04] introduce an interpreted variant of SSA. They argue that SSA-based interpretive bytecode could be used in a hybrid virtual machine that supports both compilation and interpretation. Their scheme handles variable renaming at each definition, correct selection of ϕ-function source operands, and non-scalar variables.

There have been a great many advances in processor design in the last twenty years. Some notable improvements are listed below.

Predicated instructions: These reduce the number of branch instructions, thereby increasing instruction level parallelism. Multiple control flow paths may be executed concurrently or interleaved, however only the path with the correct conditions is actually allowed to change the machine state.

Speculative execution: Processors may execute instructions without the certain knowledge that instruction source operands are correct (data speculation), or that this path of execution should be taken (control speculation). The resulting computation is possibly (but not certainly) valid. If the computation is found to be incorrect, the processor has to roll back and start again. Generally, speculations are accurate, and provide an effective way of increasing instruction level parallelism.

Multiple processors: Shared memory multiprocessor machines are widespread. Indeed, chip multiprocessors are becoming commodity items due to advances in fabrication technology. Parallel programming is now a fact of life, and must be supported by all compilers.

SSA extensions have been proposed for each of these architectural advances. Predicated SSA [CSC+99] handles predicate definitions and allows multiple control flow paths to be merged into a single predicated region.

Speculative SSA [LCH+03] introduces the notion of likeliness for instructions. Compilers use profiling information and heuristic rules to specify degrees of likeliness. This technique exposes opportunities for data speculation. Speculative SSA is used for speculative partial redundancy elimination.

Srinivasan et al [SHW93] introduce a parallel SSA, allowing compilers to apply traditional scalar optimizations to explicitly parallel programs. Their scheme supports parallel updates and guarantees to preserves the SSA property. Concurrent SSA [LMP97, LPM99] works for a less restrictive parallel programming paradigm. CSSAME [NUS98] extends concurrent SSA to incorporate explicit mutual exclusion.

High-Level Intraprocedural Constructs

There are several SSA extensions that deal with pointers and aliasing. Cytron and Gershbein present the simplest of these extensions [CG93] which is easy to implement, although it is not as powerful as more complicated approaches [CCL+96, LH98].

The original SSA inventors [CFR+91] proposed treating arrays as single scalar variables, which are accessed using the `Access` and `Update` functions. Array SSA [KS98] is probably the most popular SSA extension to handle arrays in a sophisticated manner. Extended array SSA [FKS00] also deals with objects in a strongly typed language such as Java.

Chapter 6 discusses extending SSA and similar IRs to support multiple procedures, and enable interprocedural analysis. For instance, interprocedural SSA [LDB+99] may be extended easily to another general SSA extension.

7.3 Family of Virtual Register Renaming Schemes

The general SSA extensions outlined in Section 7.2.1 are all derived from the CFG IR. They differ from CFG in two ways:

1. convention for virtual register names, and

2. additional clone operations, also known as pseudo-definitions.

This dissertation refers to such IRs as *virtual register renaming schemes* (VRRSs). Note that the term 'register renaming scheme' is commonly used [GGV98, JRB+98] when referring to dynamic register renaming in hardware. In contrast, this dissertation deals with static register renaming in software; hence it introduces the term 'virtual register renaming scheme.' This section shows how these VRRSs fit into a general framework. It groups them together and shows their relationship to each other, in a more formal manner than in Section 7.2.1.

7.3.1 Attributes

A VRRS program is still a CFG program. It encodes data flow as movement of values between virtual registers. It encodes control flow as edges between basic blocks of instructions. A particular VRRS s has two distinguishing attributes.

1. Naming property $\mathfrak{P}_s(v, p)$: Program p conforms to s if and only if each virtual register $v \in \text{VRegs}(p)$ satisfies the naming property throughout p. (Note that $\text{VRegs}(p)$ denotes the set of all virtual registers mentioned in program p.)

2. Additional semantic machinery: Some pseudo-definition functions may be necessary to achieve property $\mathfrak{P}_s(v, p)$ throughout program p.

This section discusses these two attributes in detail.

Naming Property

The first attribute is the characteristic property, $\mathfrak{P}$, of the VRRS. Often $\mathfrak{P}$ can be used to derive a criterion for deciding when a virtual register should be renamed. In the past, whenever a new VRRS has been proposed, the naming property has been specified informally in natural language. For instance, the original SSA definition [CFR+91] says that "a single program is defined to be in SSA form if each variable is a target of exactly one assignment statement in the program text." Note that Section 7.2.1 specified all VRRS properties in this way.

It would be far more satisfactory to have a *specification language* to define VRRS naming properties in a formal manner. A sufficiently general specification language can be used to describe all VRRSs. It provides the following beneficial features.

- New VRRSs can be specified concisely. This enables rapid comprehension and adoption by the community.

- A common framework clarifies relationships between VRRSs. For instance, the formal specifications given below clearly demonstrate that SSI is a simple extension of SSA, and that SSU is the dual of SSA.

- Formal specifications are precise. They avoid ambiguity, which is a common fault with informal specifications.

- Formal specifications enable mathematical reasoning about issues such as correctness. Automated tools can verify that a program conforms to a particular VRRS, or that a program transformation creates or maintains VRRS properties. Section 7.3.2 provides a detailed discussion of the possibilities for such automatic tools.

The remainder of this section discusses necessary features for a VRRS specification language. This should be considered as an initial guide, rather than a definitive formulation. The specification language adopts a set-theoretic notation. Properties are phrased in terms of *entities*, *entity sets* and *mathematical operators*.

There are five different kinds of *entity*: program; virtual register; node; edge; and path. A VRRS property specifies some constraint on these entities that should be enforceable by virtual register renaming.

program: This is generally a CFG with a set of nodes N, a set of edges $E \subseteq N \times N$, distinguished entry node $n_{\text{entry}} \in N$ and exit node $n_{\text{exit}} \in N$. See Section 2.5.1 for more details, such as the reachability constraint.

virtual register: This is a named unit of atomic storage. The set VRegs(p) is the set of all virtual registers mentioned in program p.

node: This is a labelled unit of atomic execution with a single associated instruction. Instructions perform local actions on virtual registers. The set Nodes(p) is the set of all nodes belonging to program p.

edge: This represents a unit of control flow transfer through the program. The set Edges(p) is the set of all edges belonging to program p.

path: A path is a sequence of nodes linked by edges in the CFG. For instance, $(n_0, n_1, \ldots, n_k)$ is a path in program p if, for $0 \leq i \leq k$, $n_i \in \text{Nodes}(p)$ and, for $0 \leq j \leq (k-1)$, $(n_j, n_{j+1}) \in \text{Edges}(p)$. Paths are not necessarily acyclic. The set Paths(p) is the set of all paths belonging to program p. This set will not be finite if there are loops in the control flow graph (which cause cyclic paths). Note that $\text{Paths}(p) \subseteq \bigcup_{i=1}^{\infty} N^i$ where $N^1 = \text{Nodes}(p)$, $N^2 = \text{Nodes}(p) \times \text{Nodes}(p)$, etc. Paths specify ranges of influence for virtual registers. Dominance is also based on paths. This information is often used in VRRS specifications.

There are many different *entity sets* that express relationships between entities. Set operations on these entity sets are used to query the relationships. In order for the specification language to be extensible, it should be possible to add new entity sets to this initial collection.

For a node n, there are three virtual register entity sets.

1. defs(n) is the set of virtual registers defined by the instruction associated with n.
2. uses(n) is the set of virtual registers used by the instruction associated with n.
3. mentions(n) is the set of virtual registers defined and/or used by the instruction associated with n.

For a program p, we have already described some of the entity sets.

1. VRegs(p) is the set of all virtual registers in p.
2. Nodes(p) is the set of all nodes in p.
3. Edges(p) is the set of all edges in p.
4. Paths(p) is the set of all paths in p.

One other node entity set is required for the moment. The set Clones(p) is the set of all nodes that are associated with clone functions (pseudo-definitions).

For arbitrary nodes n and n' belonging to the same program p, it is necessary to characterize some useful sets of paths that go through these nodes. Such path entity sets may be infinite, since they are subsets of the potentially infinite set Paths(p).

1. pathsFromTo(n, n') is the set containing all paths of the form $(n, \ldots, n') \in \text{Paths}(p)$, where ... represents zero or more nodes belonging to p. This is the set of all paths that start at n and end at n'.

2. pathsThrough(n) is the set containing all paths of the form $(\ldots, n, \ldots) \in \text{Paths}(p)$, where ... represents zero or more nodes belonging to p. This is the set of all paths that go through n.

The specification language requires *mathematical operators* for dealing with *sets* and *logic*. The set operators are used to query entity sets. They are $\in$ (inclusion), *subseteq* (subset), and $=$ (equality). The logic operators include standard propositional logic connectives such as $\wedge$ (and), $\vee$ (or), and $\neg$ (not). Quantifiers from first order logic are also required. The universal quantifier $\forall$ and the existential quantifier $\exists$ operate over entities. The unique existential quantifier $\exists!$ can be defined in terms of $\forall$ and $\exists$, but it is helpful to define it in-place since it is often used to specify properties such as single assignment. Note that the $|$ (such that) operator can be understood in terms of entity set intersection. Finally the $\triangleq$ operator binds a property specification to a property symbol.

Let us call this specification language VRegSpecLang. Figure 7.8 shows the VRegSpecLang specifications for $\mathfrak{P}_{\text{SSA}}$, $\mathfrak{P}_{\text{SSU}}$, and $\mathfrak{P}_{\text{SSI}}$. Basically, $\mathfrak{P}_{\text{SSA}}$ ensures that each virtual register v has a unique definition that dominates all uses of v. It is simply a transliteration of the informal SSA property into the formal notation of VRegSpecLang. The VRegSpecLang specification for $\mathfrak{P}_{\text{SSU}}$ exhibits duality with $\mathfrak{P}_{\text{SSA}}$, since 'defines' is dual with 'uses' and 'paths from entry' is dual with 'paths to exit.' The VRegSpecLang specification for $\mathfrak{P}_{\text{SSI}}$ clearly reveals that SSI programs satisfy the SSA property, since $\mathfrak{P}_{\text{SSI}}$ has just one extra line than $\mathfrak{P}_{\text{SSA}}$, which specifies the additional constraint on uses in conditional contexts. The VRegSpecLang specification for $\mathfrak{P}_{\text{SSM}}$ transliterates the informal description from Section 7.2.1. A virtual register may only mentioned once, apart from as an operand in a clone operation. The SSM transformation is allowed to introduce entirely new virtual registers, but these must be confined to clone operation mentions.

Semantic Machinery

The second VRRS attribute encapsulates the mechanics of introducing sufficient new virtual register names to maintain the naming property $\mathfrak{P}$. Generally, each VRRS provides customized virtual register clone operations. These are notational abstractions that allow circumvention of non-VRRS-conforming behaviour. Appel [App98b] refers to SSA clone operations (ϕ-functions) as a "notational trick." Sometimes these clone operations are known as *pseudo-definition* functions. Clone operations are required if it is impossible to satisfy $\mathfrak{P}$ using only classical CFG semantics. Clone operations are only useful for analysis rather than execution, so they do not need to be supported by real hardware. All clone operations may be removed (generally invalidating $\mathfrak{P}$) immediately prior to code generation, either by coalescing virtual registers or by inserting standard move instructions.

A clone operation acts as a live range splitting device. Recall from Section 2.8.2 that a live range for virtual register v extends from one or more definitions of v to one or more uses of v. A clone operation for v at some point within a live range of v effectively splits that live range into two or more sub-ranges. Clone operations may be classified by their policies regarding position, operand selection, and arity.

```
$\mathfrak{P}_{\text{SSA}}(x : \text{virtreg}, p : \text{program}) \triangleq$
    $\exists!\ n \in \text{Nodes}(p) \mid x \in \text{defs}(n)$
        $\forall\ m \in \text{Nodes}(p) \mid x \in \text{uses}(m)$
            $\forall\ q \in \text{Paths}(p) \mid q \in \text{pathsFromTo}(n_{\text{entry}}, m)$
                $q \in \text{pathsThrough}(n)$
```

```
$\mathfrak{P}_{\text{SSU}}(x : \text{virtreg}, p : \text{program}) \triangleq$
    $\exists!\ n \in \text{Nodes}(p) \mid x \in \text{uses}(n)$
        $\forall\ m \in \text{Nodes}(p) \mid x \in \text{defs}(m)$
            $\forall\ q \in \text{Paths}(p) \mid q \in \text{pathsFromTo}(m, n_{\text{exit}})$
                $q \in \text{pathsThrough}(n)$
```

```
$\mathfrak{P}_{\text{SSI}}(x : \text{virtreg}, p : \text{program}) \triangleq$
    $\exists!\ n \in \text{Nodes}(p) \mid x \in \text{defs}(n)$
        $\forall\ m \in \text{Nodes}(p) \mid x \in \text{uses}(m)$
            $\forall\ q \in \text{Paths}(p) \mid q \in \text{pathsFromTo}(n_{\text{entry}}, m)$
                $q \in \text{pathsThrough}(n)$
            $\wedge$
            $\forall\ q' \in \text{Paths}(p) \mid q' \in \text{pathsFromTo}(n, n_{\text{exit}})$
                $q' \in \text{pathsThrough}(m)$
```

```
$\mathfrak{P}_{\text{SSM}}(x : \text{virtreg}, p : \text{program}) \triangleq$
    $\exists!\ n \in \text{Nodes}(p) \mid x \in \text{defs}(n)$
        $\forall\ m \in \text{Nodes}(p) \mid x \in \text{uses}(m)$
            $m \in \text{Clones}(p)$
    $\vee$
    $\exists!\ n \in \text{Nodes}(p) \mid x \in \text{uses}(n)$
        $\forall\ m \in \text{Nodes}(p) \mid x \in \text{defs}(m)$
            $m \in \text{Clones}(p)$
    $\vee$
    $\forall\ n \in \text{Nodes}(p) \mid x \in \text{mentions}(n)$
            $n \in \text{Clones}(p)$
```

Figure 7.8: Formal specifications of several VRRS naming properties using VRegSpecLang

position: This may be *early* (minimize number of CFG edges between definition of clone function's source virtual register and clone operation itself), *late* (minimize number of CFG edges between clone function and use of clone function's destination virtual register), or somewhere *in between* these two extremes. For instance, the standard SSA construction algorithm inserts clone operations at dominance frontiers of virtual register definitions in order to minimize the number of clone operations.

operand selection: For source operands, this may be either *closest reaching definition* (including genuine definitions and definitions in clone operations), or *closest reaching actual definition* (not including definitions in clone operations). For destination operands, this may be either *closest upwardly exposed use* (including genuine uses and uses in clone operations), or *closest upwardly exposed actual use* (not including uses in clone operations). In all cases, *closeness* is measured by the number of CFG edges between the two nodes in question. The operand selection decision alters the shapes of def-use chains, which may affect the results of the sparse data flow analysis.

arity: The most basic clone operation takes a single source operand and a single destination operand. However, more advanced clone operations may be constructed by combining basic clone operations. For instance, SSI uses different clone operations depending on their positions: ϕ-functions at control flow merge points and σ-functions at control flow split points. Again, the actual operands defined and used by a clone operation may be dependent on control flow. This is the case with SSI.

Often, clone operations in a particular VRRS have a fixed setting for some of the policies outlined above, but are flexible for others. However, the framework given above should be useful for comparing the semantics of different clone operations.

7.3.2 Transformation Process

This section discusses the transformation of an arbitrary input CFG program into an equivalent program that conforms to a particular VRRS. Generally, this transformation requires a VRRS construction algorithm specialized for a particular VRRS. For instance, Chapter 3 described two construction algorithms for SSI.

All such VRRS construction algorithms must respect the *dynamic equivalence property* for input CFG program p_{CFG} and output VRRS program p_{s}, where s is some particular VRRS:

> Along any control flow path, consider any use of a virtual register v in p_{CFG} and the corresponding use v' in p_{VRRS}. Then v and v' have same value. Note that the path need not be cycle free.

This dynamic equivalence property is independent of any particular VRRS. It presupposes the existence of a relation, r : virtreg $\times$ label $\rightarrow$ virtreg, that maps virtual registers at labelled program points in p_{CFG} to corresponding virtual registers in p_{VRRS}. The dynamic equivalence is in some sense an observational equivalence. Although the transformed program is phrased in terms of different names, the dynamic data flow of values through the program is unchanged. A VRRS specification is not required to give any details as to how the transformation should be achieved. The specification simply has to provide the conditions that a transformed program must satisfy. In the language of Section 3.3.3, a VRRS specification is not required to be *prescriptive*.

Given a formal specification language like VRegSpecLang, it may be possible to generate automatically a conformance checker from a VRRS specification, that decides whether or not an input program satisfies the given VRRS property $\mathfrak{P}$. Model checking techniques may be useful here, since they operate over infinite paths through a state space. Automatic generation of conformance checkers for program transformations seems a much more difficult problem. Such checkers would verify that a program transformation pass either creates or maintains VRRS properties on programs it transforms.

The most difficult problem of all is the general VRRS transformation algorithm, which takes an input CFG program p_{CFG} and a VRRS specification s, and transforms p_{CFG} so that it conforms to s. The transformation process consists of a sequence of virtual register renamings and insertions of clone operations. However there will be infinitely many transformations of p_{CFG} that conform to s. Usually the *minimal* transformation is preferred, since it contains as little virtual register cloning as possible, while still satisfying s. One potential approach is as follows: Input programs could be converted to some more general renaming scheme and then specialized to the desired VRRS by removing superfluous clone operations and merging virtual register live ranges. This is how SSA is computed using the dependence flow graph (DFG) [JP93]. Effectively, the DFG construction algorithm inserts *switch* nodes (clone operations) at control flow split points and *merge* nodes (clone operations) at control flow merge points. SSA is recovered from DFG by removing the switch nodes, and converting the merge nodes into ϕ-functions. It should be possible to adopt a similar approach for other VRRSs. However, this is an inherently pessimistic approach to the problem, and Chapter 3 concluded that an optimistic approach is better.

7.4 Empirical Study

This section presents results obtained from the application of some VRRS transformations on CFG programs.

7.4.1 Methodology

As in previous empirical measurements, the analysed programs belong to the SPEC CPU 2000 benchmark suite [Spe00a]. A selection of the C integer benchmark programs were compiled to to CFG using the Machine SUIF compiler infrastructure [Smi96]. Figure 7.9 shows the number of virtual register names mentioned in several VRRSs. The CFG figures are taken from the CFG dump files created by the Machine SUIF compiler. The SSA figures are taken from the standard Machine SUIF SSA construction pass [Hol01]. The SSI figures are taken from the optimistic SSI construction pass described in Chapter 3. Note that both SSA and SSI passes construct semi-pruned form. Also, they only consider compiler-generated temporaries and non-address-taken automatic variables, all of which may safely reside in registers. The SSU and SSM figures are approximations, derived from the SSI results. At present, there is no SSU or SSM construction pass available for Machine SUIF. The SSU approximation estimates the number of SSU virtual registers from the number of genuine virtual register uses and the number of uses in SSI

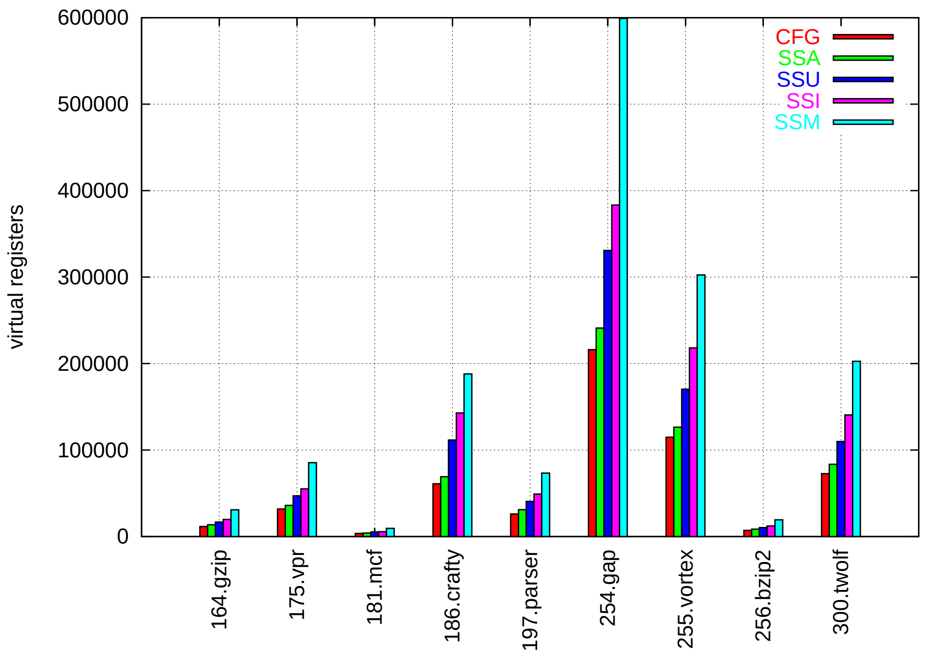


Figure 7.9: Comparison of number of virtual register names mentioned in several VRRS transforms of SPEC benchmark programs

σ-functions. Note that *genuine* uses do not include pseudo-definition functions, hence this number is invariant across all VRRSs. Note that the SSU approximation assumes that SSI σ-functions and SSU Λ-functions are equivalent. In general, there will be more σ-functions than Λ-functions, so this is an overestimate. The SSM approximation estimates the number of SSM virtual registers from the number of genuine virtual register mentions and the number of ϕ- and σ-functions in SSI. Each SSI ϕ- and σ-function is assumed to establish one new virtual register name. Note that this approximation is likely to be inaccurate since the actual clone operations for SSM are not rigorously defined, so it is not possible to say how many virtual registers would be defined by clone operations. This approximation can be treated as a lower bound on the number of virtual registers in SSM.

7.4.2 Namespace Explosion

Figure 7.9 clearly shows the effect of the virtual register naming convention on the number of virtual registers mentioned in a program. In general, more virtual register names cause smaller live ranges, which can potentially produce more accurate results in sparse data flow analysis. Chapter 4 described sparse constant propagation analysis, which is a practical example of this phenomenon. In effect, the VRRS family provides a sliding scale of related IRs from which clients can trade-off efficiency and accuracy of sparse data flow analysis. This chapter presents a larger number of VRRSs than Chapter 4, thus providing greater scope for more substantial trade-offs.

From the selection of VRRSs presented, SSI or SSM enable the most accurate sparse analysis at the cost of reduced efficiency. However recall from Section 4.6.1 that some sparse analyses do not require the power of SSI or SSM, and maximum accuracy can be achieved from a VRRS that performs less virtual register renaming.

Note that the WEB figures are so similar to SSA on the scale of Figure 7.9 that they are omitted. Figure 4.4 gives WEB vital statistics.

There are fairly fixed ratios between VRRSs in Figure 7.9. For instance, SSU consistently mentions around 1.4 times as many names as SSA. This implies that the majority of virtual registers are only used once. This is due to Machine SUIF compiler temporaries, which have been generated liberally in the compilation from C to CFG. Note that no optimizations have been applied to this code. Common subexpression elimination should increase the SSU:SSA ratio, since virtual registers may be reused rather than values recomputed. However, the different ratios between VRRSs should still be invariant across the different benchmarks.

7.5 Related Work

7.5.1 VRRS Frameworks

There has been no previous attempt to define a general framework for VRRSs. Indeed, there has been little previous categorization of any of the VRRSs presented in this chapter. This section examines existing frameworks that group several VRRSs together, although no such framework has sufficient generality to incorporate all the VRRSs mentioned in Section 7.2.1.

Johnson and Mycroft [JM03] claim that the value state dependence graph (VSDG) is a sparse generalization of CFG. However VSDG eliminates virtual register names altogether. Most data dependence graph frameworks do not have the semantic capability to express virtual register naming conventions, since they are name-agnostic. Therefore it is not possible to model different VRRSs effectively in such frameworks.

Hendren et al [HGS92] describe ALPHA, which is a family of three structured IRs: SIMPLE, α-tree and σ-α-graph. SIMPLE has the same expressive power as CFG, α-tree as SSA, and σ-α-graph as SSI. However, this is an ad-hoc collection of IRs which does not have any formal basis for the inclusion of IRs within the framework.

Stoltz's general reference chaining framework (GRC) [Sto95] extends the concept of SSA. Whereas SSA links virtual register definitions directly to virtual register uses, GRC links virtual register mentions directly to virtual register mentions. (Note a difference in terminology: this dissertation prefers the term *mention* whereas Stoltz uses *reference*.) Recall that a *mention* may be either a definition or a use. A *link* may be either forward or backward with respect to control flow. A *chain* is a set of links between mentions. If links are modelled implicitly by virtual register names, then two virtual register mentions that are linked should have the same virtual register name. Stoltz shows how GRC may be specialized into SSA and λ-chains IR, which has the same expressive power as SSU. There are algorithms to transform programs from CFG into GRC, including the insertion of merge operations at program points where information converges from two or more different control flow paths. However, this framework is still not general enough to cater for all VRRSs. Notably, SSI cannot be expressed in GRC. This is because GRC algorithms only allow unidirectional data flow information. Thus only one kind of merge operator is permitted, either for 'downward-exposed references' in forward data flow problems (like SSA ϕ-functions) or for 'upward-exposed references' in backward problems (like SSU Λ-

functions). So GRC is used for sparse forward and backward data flow problems, but it cannot handle bidirectional problems.

Thus there is no existing IR framework within which all the VRRSs described in Section 7.2.1 can be properly accommodated.

7.5.2 VRRS Specification Languages

The specification language VRegSpecLang, described in Section 7.3.1, is a custom language for specifying VRRS properties. There are a number of existing languages that describe CFG properties. However none of these is able to specify VRRS properties concisely.

Bernhard Steffen [Ste93] shows how to represent data flow analysis algorithms using modal logic. This enables the data flow analysis algorithms to be automatically generated using model checking and partial evaluation. Steffen and Schmidt [SS98] describe this approach as "encoding a bit vector's bits as boolean propositions which are decided by model checking." Steffen [Ste93] works through the example of partial redundancy elimination for expressions in CFG programs. This framework is sufficiently powerful to support standard unidirectional bit vector data flow analysis algorithms, but VRRS transformation is more complicated. (For instance, consider the SSI construction algorithm in Section 3.5.2.) Steffen's modal specifications are generally universally satisfied, which means they hold at all program points. VRRS specifications often need to be restricted to certain paths through the CFG. In fact, Steffen does not operate directly in terms of CFG. Instead he uses an abstract model, with abstract propositions derived from the original program. VRRS specifications are more intuitive when expressed directly in terms of CFG entities like nodes and paths. Steffen's modal specifications cannot express uniqueness succinctly. This appears to be a general problem with path logics. However uniqueness is essential for many VRRS specifications.

Sittampalam et al [SdL04, DdS02] specify program transformations in a declarative style known as path logic programming, which uses regular expressions on paths instead of modal logic. They generate program transformers from these specifications. They specify situations that are wrong, then specify the rewrite required to rectify the problem. In contrast, the aim of VRegSpecLang is to specify the correct situation alone, without providing details of any transformation algorithm. Again, path logic programming seems to have problems with uniqueness, and with non-universal applicability of rules. David Lacey [Lac03] describes a similar system. He develops TRANS, which is a language for CFG rewrites governed by predicates that determine when to apply the rewrite rules. Predicates are described using CTL, which is a style of temporal logic. Lacey's system has the same drawbacks as path logic programming.

7.6 Concluding Remarks

This chapter has reviewed an extensive selection of SSA extensions, and classified these extensions as either general or feature-specific. It has shown that the general SSA extensions form a family of virtual register renaming schemes. It has related several existing virtual register renaming schemes in this family (such as SSA, SSU and SSI) together with a new scheme (SSM). It has presented a simple specification language, VRegSpecLang, for describing VRRS properties.

The primary future research goal is to devise a generic algorithm which, given a desired VRRS property $\mathfrak{P}$, will transform an arbitrary CFG program into an equivalent program conforming to that VRRS. It is not yet clear whether this kind of automatic program generation is feasible.

In general, opinion is divided with regard to the merits of virtual register renaming. However several respected researchers suggest that an increased degree of virtual register renaming improves the accuracy of data flow analysis. Zadeck [Zad04], one of the founding fathers of SSA, states:

> Lots of names in general means better optimization. At the admitted cost of extra space and time, the smaller granularity of treating the SSA variables as first class variables will in general lead to better packing of registers and a smaller number of copy statements.

Cooper and Torczon [CT04], longstanding compiler researchers from Rice University, relate the lesson learned from a Fortran compiler project in the late 1980's:

> Unfortunately, associating multiple expressions with a single temporary name obscured the flow of data and degraded the quality of optimization. The decline in code quality overshadowed any compile time benefits.

The most significant contribution of this chapter is to provide a formal framework, that enables a systematic choice about the appropriate level of virtual register renaming. It does not advocate any particular VRRS; it simply points out that there is a wide variety of available VRRSs.

Chapter 8

Conclusions

This dissertation has developed SSI as an extension of SSA. Further, it has shown that systematic virtual register renaming is a useful mechanism that generates IRs for flexible sparse data flow analysis.

8.1 Summary

This section summarizes the ideas presented throughout the dissertation.

Chapter 1 argued that static analysis is still necessary, despite the doubt and dissent that Proebsting's law has stirred up [Pro98].

Chapter 2 explained that there are many existing IRs for static analysis. It asserted (and Chapter 5 confirmed) that SSI embeds control flow, data dependence and control dependence relations. This means that SSI is at least as expressive as any of the recently proposed augmented CFG and augmented DDG IRs.

Chapter 3 comprehensively reviewed SSI, with a concise new definition and optimistic construction algorithm. It compared SSI and SSA.

Chapter 4 showed that SSI enables more accurate, though less efficient, analysis than SSA and WEB which are similar IRs. The improved accuracy of SSI was shown, both empirically and formally, to be due to its extra live range splitting as a result of the greater degree of virtual register renaming.

Chapter 5 showed that SSI encodes dependence information implicitly in its virtual register naming convention. This means that SSI slicing is as accurate as, and more efficient than, CFG slicing.

Chapter 6 showed that SSI can easily be extended to whole program scope, using existing or novel techniques.

Chapter 7 showed that SSI is one of many SSA extensions. This chapter classified these extensions, and identified a family of virtual register renaming schemes (VRRSs). It formulated a new IR called static single mention form (SSM). It introduced a formal specification language, VRegSpecLang, for these VRRSs, and specified several VRRSs using this new notation.

8.2 Future Work

Much work remains to be done. Individual chapters pointed out future research opportunities. This section highlights the most outstanding issues.

1. Newly proposed SSI analyses should be added to existing compiler infrastructures.
2. In order to be credible, SSM requires a construction algorithm and some real-world applications. Otherwise it will remain nothing more than a theoretical nicety.
3. Alternative VRRSs should be used for static analysis. Either new analyses should be devised for existing VRRSs, or entirely new VRRSs should be created. Again, the key issue is the trade-off between accuracy and efficiency for sparse analysis.
4. Given the foundation provided by VRegSpecLang, it should be possible to produce automated tools for VRRS verification and generation. A VRRS compiler compiler would be most desirable.
5. VRRSs should be linked with other research areas. For instance, aspect-oriented programming (AOP) [KLM+97] may provide some inspiration. Is virtual register renaming an AOP cross-cutting concern? Could virtual register renaming points be specified as AOP join points?

8.3 Final Remarks

Over the last 18 years, SSA has become virtually ubiquitous within mainstream optimizing compilers. Perhaps over the next 18 years, more generalized notions of virtual register renaming will be incorporated into optimizing compilers, thus enabling greater flexibility for analysis. VRRSs may provide the opportunity for optimizing compiler research to overcome Proebsting's pessimism.

Appendix A

Glossary

This glossary defines some of the terminology and acronyms that are used throughout the dissertation. Some of the concepts are defined in detail during the dissertation. Other concepts are assumed to be common knowledge. The terms are listed in alphabetical order. Terms that have been introduced or redefined by this dissertation are given in italics.

assignment: see **definition**.

augmented CFG: **CFG** with additional encoded information.

augmented DDG: **DDG** with additional encoded information.

basic block: a sequence of consecutive **instructions** such that if control reaches the first **instruction**, then every **instruction** in the block must be executed.

CFG: control flow graph.

classical: data flow analysis research between 1970 and 1990, generally relating to **CFG**.

client: an analysis or transformation that relies on data flow information computed by another analysis.

clone operation: an **assignment** that copies the value of one **virtual register** into another, thus introducing a new **virtual register** name at the next **program point**.

control flow path: a sequence of successive nodes and edges that lead from the path start point to the path end point in a **CFG** with a single **instruction** at each node.

DDG: data dependence graph.

DFA: data flow analysis.

DFG: dependence flow graph, an **augmented CFG**.

definition: an action that updates the value stored in a **virtual register**.

dense: the data flow information stored with each register cannot be bounded by $O(1)$.

Dragon book: **classical** compiler textbook [ASU86] with a red dragon of 'complexity of compiler design' depicted on the front cover.

execution path: a **control flow path** that may be executed. Some control flow paths cannot be executed.

***FIR*:** functional intermediate representation.

flow-sensitive: analysis results depend not only on the **instructions**, but also on the control flow relations between those **instructions**.

flow-insensitive: analysis results depend on **instructions** only, not on control flow relations between those **instructions**.

GSA: gated single assignment form, an **augmented DDG**.

high-level language: a conventional programming language that provides some level of abstraction above the primitive **instructions**.

instruction: a primitive computational operation.

load: an **instruction** that reads a value from **memory**.

memory: data storage area, distinct from any **physical registers**.

mention: a **definition** or a **use**.

PDG: program dependence graph, an **augmented DDG**.

PDW: program dependence web, an **augmented DDG**.

physical register: a real storage location that holds a single scalar value, and can be accessed at high speed.

program: a specification of a computation, in some formal notation such as a **high-level language** or a **CFG**.

program point: either immediately before or immediately after any single **instruction** in the **program**.

program text: the code of a program.

pseudo-definition: a **clone operation**.

reach: **instruction** i reaches **instruction** j if there is a control flow path from i to j along which the effect of **instruction** i is not killed.

reaching definition: a **definition** of **virtual register** v at **instruction** i is a reaching definition to **instruction** j if there is a **control flow path** from i to j along which **virtual register** v is not defined except at i.

reference: see **use**.

SEG: sparse evaluation graph, an analysis-specific **sparse** intermediate representation.

***sparse*:** the data flow information stored with each **virtual register** is $O(1)$.

SSA: static single assignment form, an **augmented CFG** and a **VRRS**.

SSI: static single information form, an **augmented CFG** and a **VRRS**.

***SSM*:** static single mention form, an **augmented CFG** and a **VRRS**.

SSU: static single use form, an **augmented CFG** and a **VRRS**.

store: an **instruction** that writes a value to **memory**.

tail call: the last action of a procedure is to call another procedure, which can be implemented more efficiently than a standard procedure call.

upwardly exposed use: a **use** of **virtual register** v at **instruction** i is an upwardly exposed use to **instruction** j if there is a **control flow path** from j to i along which **virtual register** v is not defined.

use: an action that reads the value stored in a **virtual register**.

VDG: value dependence graph, an **augmented DDG**.

virtual register: an abstract storage location that holds a single scalar value.

***VRRS*:** **virtual register** renaming scheme.

VSDG: value state dependence graph, an **augmented DDG**.

WEB: def-use-use-def webs form, an **augmented CFG** and a **VRRS**.

Bibliography

[AAB$^+$00] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), Feb 2000.

[Agr00] Gagan Agrawal. Demand-driven construction of call graphs. In *Proceedings of the 9th International Conference on Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 125–140. Springer, 2000.

[AH00] John Aycock and Nigel Horspool. Simple generation of static single assignment form. In *Proceedings of the 9th International Conference in Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 110–125. Springer, 2000.

[AK02] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach.* Morgan Kaufmann, 2002.

[AKPW83] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 177–189, 1983.

[Ana99] C. Scott Ananian. The static single information form. Technical Report MIT-LCS-TR-801, Laboratory for Computer Science, Massachusetts Institute of Technology, Sep 1999.

[ANH99] Ana Azevedo, Alex Nicolau, and Joe Hummel. Java annotation-aware just-in-time (AJIT) compilation system. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 142–151, 1999.

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[App98a] Andrew W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[App98b] Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, Apr 1998.

[App01] Andrew W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–256, 2001.

[AR03] C. Scott Ananian and Martin Rinard. Data size optimizations for Java programs. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems*, pages 59–68, 2003.

[ASG97] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 134–145, 1997.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–11, 1988.

[Ayc00] John Aycock. Aggressive type inference. In *Proceedings of the 8th International Python Conference*, pages 11–20, 2000.

[Bak95] Henry G. Baker. 'Use-once' variables and linear objects—storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1):45–52, Jan 1995.

[Baw93] Alan Bawden. Implementing distributed systems using linear naming. Technical Report 1627, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Mar 1993.

[BCHS98] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software—Practice and Experience*, 28(8):859–881, Jul 1998.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, Dec 1994.

[BGS97] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Interprocedural conditional branch elimination. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 146–158, 1997.

[BGS00] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 321–333, 2000.

[BL00] Thomas Ball and James R. Larus. Using paths to measure, explain, and enhance program behavior. *IEEE Computer*, 33(7):57–65, 2000.

[BM94] Marc M. Brandis and Hanspeter Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 16(6):1684–1698, Nov 1994.

[BMO90] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 257–271, 1990.

[BMS03] Lennart Beringer, Kenneth MacKenzie, and Ian Stark. Grail: a functional form for imperative mobile code. *Electronic Notes in Theoretical Computer Science*, 85(1), Jun 2003.

[BP99] Gianfranco Bilardi and Keshav Pingali. The static single assignment form and its computation. Technical report, Department of Computer Science, Cornell University, Jul 1999.

[BP03] Gianfranco Bilardi and Keshav Pingali. Algorithms for computing the static single assignment form. *Journal of the ACM*, 50(3):375–425, May 2003.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, 2002.

[Bri92] Preston Briggs. *Register Allocation via Graph Colouring*. PhD thesis, Rice University, Apr 1992.

[Cal88] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN conference on Programming Language Design and Implementation*, 1988.

[Car97] Luca Cardelli. *Type Systems*, chapter 103, pages 2208–2236. CRC Press, 1997.

[CCCH93] W. Y. Chen, P. P. Chang, T. M. Conte, and W. W. Hwu. The effect of code expanding optimizations on instruction cache design. *IEEE Transactions on Computers*, 42(9):1045–1057, Sep 1993.

[CCF91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 55–66, 1991.

[CCL$^+$96] Fred C. Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in SSA form. In *Proceedings of the International Conference on Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 1996.

[CDC$^+$04] Rezaul A. Chowdhury, Peter Djeu, Brendon Cahoon, James H. Burrill, and Kathryn S. McKinley. The limits of alias analysis for scalar optimizations. In *Proceedings of the 13th International Conference on Compiler Construction*, volume 2985 of *Lecture Notes in Computer Science*, pages 24–38. Springer, 2004.

[CDG96] Craig Chambers, Jeffrey Dean, and David Grove. Whole-program optimization of object-oriented languages. Technical Report 96-06-02, Department of Computer Science and Engineering, University of Washington, Jun 1996.

[CF89] E. Cartwright and M. Felleisen. The semantics of program dependence. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 13–27, 1989.

[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. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[CG93] Ron Cytron and Reid Gershbein. Efficient accommodation of may-alias information in SSA form. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 36–45, 1993.

[CH90] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, Oct 1990.

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 98–101, 1982.

[CHT91] Keith D. Cooper, Mary W. Hall, and Linda Torczon. An experiment with inline substitution. *Software—Practice and Experience*, 21(6):581–601, Jun 1991.

[CHT92] Keith D. Cooper, Mary W. Hall, and Linda Torczon. Unexpected side effects of inline substitution: A case study. *ACM Letters on Programming Languages and Systems*, 1(1):22–32, Mar 1992.

[CKZ03] Manuel M.T. Chatravarty, Gabriele Keller, and Patryk Zadarnowski. A functional perspective on SSA optimisation algorithms. In *Proceedings of the 2nd International Workshop on Compiler Optimization Meets Compiler Verification*, 2003.

[Cli93] Cliff Click. From quads to graphs: An intermediate representation's journey. Technical Report CRPC-TR93366-S, Center for Research on Parallel Computation, Rice University, Oct 1993.

[CP95] Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, pages 35–49, 1995.

[CSC$^+$99] Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Jeanne Ferrante. Predicated static single assignment. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 245–255, 1999.

[CT04] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004.

[Dan99] Sebastian Danicic. *Dataflow Minimal Slicing*. PhD thesis, University of North London, 1999.

[DdS02] Stephen Drape, Oege de Moor, and Ganesh Sittampalam. Transforming the .NET intermediate language using path logic programming. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 133–144, 2002.

[DEC82] DEC. *DECsystem-10 / DECSYSTEM-20 Processor Reference Manual*. 1982. http://www.36bit.org/dec/manual/.

[DMM01] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Using types to analyze and optimize object-oriented programs. *ACM Transactions on Programming Languages and Systems*, 23(1):30–72, Jan 2001.

[DRZ92] Dhananjay M. Dhamdhere, Barry K. Rosen, and F. Kenneth Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pages 212–223, 1992.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, Feb 2001.

[Ern95] Michael D. Ernst. Slicing pointers and procedures (abstract). Technical Report MSR-TR-95-23, Microsoft Research, Jan 1995.

[Ern03] Michael D. Ernst. Static and dynamic analysis: synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[ESM04] Robert Ennals, Richard Sharp, and Alan Mycroft. Linear types for packet processing. In *Proceedings of the 13th European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2004.

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–51, Feb 1991.

[FKS00] Stephen Fink, Kathleen Knobe, and Vivek Sarkar. Unified analysis of array and object references in strongly typed languages. In *Proceedings of the 7th International Static Analysis Symposium*, volume 1824 of *Lecture Notes in Computer Science*, pages 155–174. Springer, 2000.

[Fle98] The Flex compiler infrastructure, 1998. http://www.flex-compiler.lcs.mit.edu/Harpoon/.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, Jul 1987.

[Fow05] Martin Fowler. Refactoring home page, 2005. http://www.refactoring.com.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 237–247, 1993.

[GB03] Lal George and Matthias Blume. Taming the IXP network processor. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 26–37, 2003.

[GGV98] A. Gonzlez, J. Gonzlez, and M. Valero. Virtual-physical registers. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 175–184, 1998.

[Gra04] GrammaTech. CodeSurfer, 2004. http://www.grammatech.com/products/codesurfer/.

[GSR03] Ovidiu Gheorghioiu, Alexandru Salcianu, and Martin Rinard. Interprocedural compatibility analysis for static object preallocation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages*, pages 273–284, 2003.

[Hav93] Paul Havlak. Construction of thinned gated single-assignment form. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 477–499. Springer, 1993.

[HC89] Wen-mei W. Hwu and Pohua P. Chang. Inline function expansion for compiling C programs. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 246–257, 1989.

[HCXE02] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 69–82, 2002.

[HD98] Glenn H. Holloway and Allyn Dimock. The Machine SUIF bit-vector data-flow-analysis library, 1998.

[Hen00] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, Jul 2000.

[HFH$^+$02] Mark Harman, Chris Fox, Rob Hierons, Lin Hu, Sebastian Danicic, and Joachim Wegener. Vada: A transformation-based system for variable dependence analysis. In *Proceedings of the 2nd IEEE Workshop on Source Code Analysis and Manipulation*, 2002.

[HGS92] L. Hendren, G. Gao, and V. Sreedhar. ALPHA: A family of structured intermediate representations for a parallelizing C compiler, 1992. ACAPS Technical Memo 49, School of Computer Science, McGill University, Montreal, Quebec.

[HH98] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 97–105, 1998.

[HH01] M. Harman and R. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[HL02] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, 2002.

[HMCCR94] Mary W. Hall, John M. Mellor-Crummey, Alan Carle, and René G. Rodríguez. FIAT: A framework for interprocedural analysis and transformation. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 522–545. Springer, 1994.

[Hof95] Tommy Hoffner. Evaluation and comparison of program slicing tools. Technical Report LiTH-IDA-R-95-01, Department of Computer and Information Science, Linköping University, Sweden, 1995.

[Hol01] Glenn Holloway. The Machine-SUIF static single assignment library, 2001.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, Jan 1990.

[HS94] Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, Mar 1994.

[HS02] Glenn Holloway and Michael D. Smith. The Machine-SUIF control flow graph library, 2002.

[Int03] Intel Corporation. Intel IXP2400 network processor: Flexible, high-performance solution for access and edge applications, 2003.

[Int04] Intel. *IA32 Intel Architecture Software Developer's Manual.* 2004. `http://www.intel.com/design/pentium4/manuals/index_new.htm`.

[JM03] Neil Johnson and Alan Mycroft. Combined code motion and register allocation using the value state dependence graph. In *Proceedings of the 12th International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2003.

[Joh85] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science.* Springer, 1985.

[Joh04] Neil E. Johnson. Code size optimization for embedded processors. Technical Report UCAM-CL-TR-607, University of Cambridge Computer Laboratory, Nov 2004.

[JP93] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 78–89, 1993.

[JPP93] Richard Johnson, David Pearson, and Keshav Pingali. Finding regions fast: Single entry single exit and control regions in linear time. Technical Report CTC93TR141, Department of Computer Science, Cornell University, Jul 1993.

[JPP94] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994.

[JRB+98] Stephen Jourdan, Ronny Ronen, Michael Bekerman, Bishara Shomar, and Adi Yoaz. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *Proceedings of the 31st ACM/IEEE International Symposium on Microarchitecture*, pages 216–225, 1998.

[KCL+99] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, May 1999.

[KD99] Uday P. Khedker and Dhananjay M. Dhamdhere. Bidirectional data flow analysis: myths and reality. *ACM SIGPLAN Notices*, 34(6):47–57, Jun 1999.

[KDM03] Uday P. Khedker, Dhananjay M. Dhamdhere, and Alan Mycroft. Bidirectional data flow analysis for type inferencing. *Computer Languages, Systems and Structures*, 29(1–2):15–44, 2003.

[Kel95] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices*, 30(3):13–22, Mar 1995.

[KG03] Uday Khedker and R. Govindarajan. Compiler analysis and optimizations : What is new? In *Proceedings of the Workshop on Cutting Edge Computing (New Frontiers in High Performance Computing)*, pages 59–69, 2003.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[Knu71] Donald E. Knuth. An empirical study of FORTRAN programs. *Software—Practice and Experience*, 1(2):105–133, Apr 1971.

[KRS94] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, Jul 1994.

[KS98] Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in parallelization. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 107–120, 1998.

[KU80] Marc A. Kaplan and Jeffrey D. Ullman. A scheme for the automatic inference of variable types. *Journal of the ACM*, 27(1):128–145, Jan 1980.

[Lac03] David Lacey. *Program transformation using temporal logic specifications.* PhD thesis, University of Oxford, Aug 2003.

[LCH$^+$03] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. A compiler framework for speculative analysis and optimizations. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 289–299, 2003.

[LCK$^+$98] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 26–37, 1998.

[LDB$^+$99] Shih-Wei Liao, Amer Diwan, Robert P. Bosch, Jr., Anwar Ghuloum, and Monica S. Lam. Suif explorer: an interactive and interprocedural parallelizer. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 37–48, 1999.

[LG99] Allen Leung and Lal George. Static single assignment form for machine code. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 204–214, 1999.

[LH98] Christopher Lapkowski and Laurie J. Hendren. Extended SSA numbering: Introducing SSA properties to language with multi-level pointers. In *Proceedings of the 7th International Conference on Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 128–143. Springer, 1998.

[LMP97] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing*, volume 1366 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 1997.

[LPM99] Jaejin Lee, David A. Padua, and Samuel P. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1999.

[LR91] William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem taxonomy. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 93–103, 1991.

[LSG00] Alexandre Lenart, Christopher Sadler, and Sandeep K. S. Gupta. SSA-based flow-sensitive type analysis: combining constant and type propagation. In *Proceedings of the ACM Symposium on Applied Computing*, pages 813–817, 2000.

[LTS01] Christopher League, Valery Trifonov, and Zhong Shao. Functional Java bytecode. In *Proceedings of the 5th World Conference on Systemics, Cybernetics, and Informatics—Workshop on Intermediate Representation Engineering for the Java Virtual Machine*, 2001.

[MGM02] G.B. Mund, D. Goswami, and Rajib Mall. *Program Slicing*, chapter 8, pages 269–294. CRC Press, 2002.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, Dec 1978.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, Apr 1965.

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, Feb 1979.

[MRB95] T. J. Marlowe, B. G. Ryder, and M. Burke. Defining flow sensitivity for data flow problems. Technical Report LCSR-TR-249, Laboratory of Computer Science, Rutgers University, Jul 1995.

[MS93] Robert Metzger and Sean Stroud. Interprocedural constant propagation: an empirical study. *ACM Letters on Programming Languages and Systems*, 2(1-4):213–232, Mar–Dec 1993.

[MS03] Wolfgang Mayer and Markus Stumptner. Debugging program exceptions. In *Proceedings of the 14th International Workshop on Principles of Diagnosis*, pages 119–124, 2003.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (revised)*. MIT Press, 1997.

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

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[Myc99] Alan Mycroft. Type-based decompilation. In *Proceedings of the 8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 208–223. Springer, 1999.

[Mye81] Eugene M. Myers. A precise inter-procedural data flow algorithm. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 219–230, 1981.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[Nel79] Philip A. Nelson. A comparison of PASCAL intermediate languages. *ACM SIGPLAN Notices*, 14(8):208–213, Aug 1979.

[Net04] Nicholas Nethercote. Dynamic binary analysis and instrumentation. Technical Report 606, Computer Laboratory, University of Cambridge, Nov 2004. PhD dissertation.

[NM03] Nicholas Nethercote and Alan Mycroft. Redux: A dynamic dataflow tracer. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[Nov03] Diego Novillo. TreeSSA a new optimization infrastructure for GCC. In *Proceedings of the 2003 GCC Developers' Summit*, pages 181–193, 2003.

[NP94] Cindy Norris and Lori L. Pollock. Register allocation over the program dependence graph. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 266–277, 1994.

[NUS98] Diego Novillo, Ronald C. Unrau, and Jonathan Schaeffer. Concurrent SSA form in the presence of mutual exclusion. In *Proceedings of the International Conference on Parallel Processing*, pages 356–364, 1998.

[OJ97] Robert O'Callahan and Daniel Jackson. Lackwit: a program understanding tool based on type inference. In *Proceedings of the 19th International Conference on Software Engineering*, pages 338–348, 1997.

[OK03] Carl Offner and Kathleen Knobe. Weak dynamic single assignment form. Technical Report TR-HPL-2003-169, HP Labs, Nov 2003.

[OO84] K. Ottenstein and L. Ottenstein. The program dependence graph in software development environments. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, 1984.

[Pat95] Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 67–78, 1995.

[PBJ$^+$91] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence flow graphs: an algebraic approach to program dependencies. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 67–78, 1991.

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

[Pro98] Todd Proebsting. Proebsting's law: Compiler advances double computing power every 18 years, 1998. http://research.microsoft.com/~toddpro/papers/law.htm.

[QHV00] Feng Qian, Laurie Hendren, and Clark Verbrugge. A comprehensive approach to array bounds check elimination for Java. Technical Report Sable TR 2000-4, School of Computer Science, McGill University, Nov 2000.

[Rau96] B. Ramakrishna Rau. Iterative modulo scheduling. *International Journal of Parallel Processing*, 24(1):3–64, Feb 1996.

[Ray99] Eric S. Raymond. *The Cathedral and the Bazaar*. O'Reilly, 1999.

[Rep98] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11–12):701–726, Nov–Dec 1998.

[RG89a] Stephen Richardson and Mahadevan Ganapathi. Interprocedural analysis versus procedure integration. *Information Processing Letters*, 32(3):137–142, Aug 1989.

[RG89b] Stephen Richardson and Mahadevan Ganapathi. Interprocedural optimization: experimental results. *Software—Practice and Experience*, 19(2):149–168, Feb 1989.

[Rol03] Laurent Rolaz. An implementation of sparse conditional constant propagation for Machine SUIF, 2003.

[RR89] G. Ramalingam and Thomas Reps. Semantics of program representation graphs. Technical Report TR-900, Computer Sciences Department, University of Wisconsin-Madison, Dec 1989.

[RR00] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices and accessed memory regions. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 182–195, 2000.

[Ruf95a] Erik Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 13–22, 1995.

[Ruf95b] Erik Ruf. Optimizing sparse representations for dataflow analysis. *ACM SIGPLAN Notices*, 30(3):50–61, Mar 1995.

[RWS79] Martin Richards and Colin Whitby-Strevens. *BCPL: the Language and its Compiler*. Cambridge University Press, 1979.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–27, 1988.

[SBA00] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 108–120, 2000.

[Sch73] Marvin Schaeffer. *A Mathematical Theory of Global Program Optimization*. Prentice-Hall, 1973.

[Sco01] Kevin Scott. On Proebsting's law. Technical Report CS-2001-12, Department of Computer Science, University of Virginia, Mar 2001.

[SdL04] Ganesh Sittampalam, Oege de Moor, and Ken Friis Larsen. Incremental execution of transformation specifications. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 26–38, 2004.

[SG95] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 62–73, 1995.

[SGW94] Eric Stoltz, Michael P. Gerlek, and Michael Wolfe. Extended SSA with factored use-def chains to support optimization and parallelism. In *Proceedings of the Hawaii International Conference on Systems Sciences*, pages 43–52, 1994.

[SHW93] Harini Srinivasan, James Hook, and Michael Wolfe. Static single assignment for explicitly parallel programs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 260–272, 1993.

[SIM$^+$05] Alex Shye, Matthew Iyer, Tipp Moseley, David Hodgdon, Dan Fay, Vijay Janapa Reddi, and Daniel A. Connors. Analysis of path profiling information generated with performance monitoring hardware. In *Proceedings of the 9th Workshop on Interaction between Compilers and Computer Architecture*, pages 34–43, 2005.

[Sin03] Jeremy Singer. SSI extends SSA. In *Work in Progress Session Proceedings of the Twelfth International Conference on Parallel Architectures and Compilation Techniques*, Sep 2003.

[Smi96] Michael D. Smith. Extending SUIF for machine-dependent optimizations. In *Proceedings of the First SUIF Compiler Workshop*, pages 14–25, Jan 1996.

[Spe00a] SPEC CPU2000 benchmark suite, 2000. http://www.spec.org.

[Spe00b] SPEC CPU2000 run and reporting rules, 2000. http://www.spec.org/cpu2000/docs/runrules.html.

[SS70] R. M. Shapiro and H. Saint. The representation of algorithms. Technical Report CA-7002-1432, Massachusetts Computer Associates, Feb 1970.

[SS98] David Schmidt and Bernhard Steffen. Program analysis as model checking of abstract interpretations. In *Proceedings of the 5th International Symposium on Static Analysis*, volume 1503 of *Lecture Notes in Computer Science*, pages 351–380. Springer, 1998.

[SS03] Konstantinos Sagonas and Erik Stenman. Experimental evaluation and improvements to linear scan register allocation. *Software—Practice and Experience*, 33(11):1003–1034, Sep 2003.

[Ste93] Bernhard Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming*, 21(2):115–139, Oct 1993.

[Ste98] Christoph Steindl. Intermodular slicing of object-oriented programs. In *Proceedings of the 7th International Conference on Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 1998.

[Sto95] Eric James Stoltz. *Intermediate Compiler Analysis via Reference Chaining*. PhD thesis, Oregon Graduate Institute of Science and Technology, Jan 1995.

[Sun99] Java HotSpot, 1999. http://java.sun.com/products/hotspot/.

[Ten74] A. Tennenbaum. *Type determination for very high level languages*. PhD thesis, Courant Institute, New York University, Oct 1974. As cited by [ASU86].

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[TM04] John Teifel and Rajit Manohar. Static tokens: Using dataflow to automate concurrent pipeline synthesis. In *Proceedings of the 10th International Symposium on Asynchronous Circuits and Systems*, pages 17–27, 2004.

[TP95] Peng Tu and David Padua. Efficient building and placing of gating functions. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 47–55, 1995.

[Vis01] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In *Rewriting Techniques and Applications*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer, May 2001.

[VJB+03] Peter Vanbroekhoven, Gerda Janssens, Maurice Bruynooghe, Henk Corporaal, and Francky Catthoor. A step towards a scalable dynamic single assignment conversion. Technical Report CW 360, Department of Computer Science, Katholieke Universiteit Leuven, Apr 2003.

[vWF04] Jeffery von Ronne, Ning Wang, and Michael Franz. Interpreting programs in static single assignment form. In *Proceedings of the ACM SIGPLAN 2004 Workshop on Interpreters, Virtual Machines and Emulators*, pages 23–30, 2004.

[Wal86] David W. Wall. Global register allocation at link time. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, pages 264–275, 1986.

[WCES94] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: representation without taxation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 297–310, 1994.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, Jul 1982.

[Wei84] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, Oct 1984.

[Wei88] Reinhold P. Weicker. Dhrystone benchmark: rationale for version 2 and measurement rules. *ACM SIGPLAN Notices*, 23(8):49–62, Aug 1988.

[WFW+94] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers, 1994.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 1–12, 1995.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr 1991.

[Zad04] F. Kenneth Zadeck. Posting to GCC mailing list, Mar 2004. `http://gcc.gnu.org/ml/2004-03/msg01005.html`.

[ZP03] Xiaotong Zhuang and Santosh Pande. Resolving register bank conflicts for a network processor. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 269–278, 2003.