MSPLS 2002

Proceedings of the Workshop of the Midwest Society for Programming Languages and Systems

Bloomington, Indiana
April 13, 2002

Editors: Jaakko Järvi Andrew Lumsdaine
David S. Wise
{jajarvi|lums|dswise}@cs.indiana.edu

The Midwest Society for Programming Languages and Systems (MSPLS) is a group of people interested in programming languages, programming systems, and system software in general, and who reside in the Midwestern USA. The MSPLS holds one-day workshop meetings at least once each academic year. The aim of the MSPLS Workshops is to allow an exchange of ideas in all areas of programming languages and systems among researchers and practitioners from Midwestern universities and companies.

This report contains the abstracts of the presentations given at the MSPLS Spring workshop, held at the Computer Science department of the Indiana University. The report contains 10 abstracts, including the abstract of the invited talk by Arch Robison. The authors were given the option to submit a full paper in addition to the abstract. Consequently, two full papers describing the work presented in the workshop are included.

The MSPLS maintains a mailing list, which is used to inform about upcoming events. The mailing list is open to anyone, and instructions for joining can be found at: `http://mail.cis.ohio-state.edu/mailman/listinfo/mspls/`.

C# for Complier Gurus

Arch D. Robinson
Intel
arch.robison@intel.com

C# is a new language that is part of Microsoft's .NET. Like Java, it pretends to be a C++ derivative, but is really more like typed Smalltalk dressed in C syntax. This talk compares C# with Java and C++, from the perspective of a compiler writer. Like Java, C# enforces a particular object-oriented style. Unlike Java, however, C# has a more regular type system. For example, "int" is not a special type as it is in Java. C# has far more creature comforts than Java, such as reference parameters, enums, multidimensional arrays, and operator overloading. Like C++, C# has as an "unsafe" subset that permits pointer arithmetic and abusive casting. Unlike C++, it lacks parametric polymorphism and multiple inheritance of implementation. I'll explore these and other differences, and also report on my experience, good and bad, with using C# to prototype compiler infrastructure. Some of the creature comforts that I first dismissed as trivial syntactic sugar have become quite addictive.

Compiler Guaranteed Sequential Consistency

Xing Fang[1,⋆], *Chi-Leung Wong*[2], *Zehra Sura*[2], *Jaejin Lee*[1], *Samuel P. Midkiff*[3] *and David Padua*[2]

[1]*Michigan State University*
[2]*University of Illinois at Urbana Champaign*
[3]*IBM T.J. Watson Research Center*

⋆fangxing@cse.msu.edu

While sequential consistency is arguably the most intuitive and natural memory consistency model for programmers, many shared memory multiprocessors follow a relaxed consistency model. Relaxed consistency models allow reordering of reads and writes, enabling a variety of hardware level optimizations. This boosts system performance, but at the price of difficult programming and porting. In this project, we are building a compiler that achieves the best of both worlds: performance and ease of programming. The compiler provides a sequentially consistent view of the underlying architecture to the programmer by automatically mapping the program with sequentially consistent semantics to hardware supporting relaxed consistency. This is done by inserting memory fence instructions, where necessary, to force the program execution to be sequentially consistent. A simple thread-escape analysis is first performed on the programs, and the result is used to direct fence insertion algorithms in the later passes of the compiler. We present different fence insertion optimization algorithms developed and implemented in Jikes Research Virtual Machine.

The Pensieve Project: A Compiler Infrastructure for Memory Models

Chi-Leung Wong[1], *Zehra Sura*[1], *Xing Fang*[2] *Samuel P. Midkiff*[3], *Jaejin Lee*[2], *David Padua*[1]

[1]*University of Illinois at Urbana Champaign*
[2]*Michigan State University*
[3]*IBM T.J. Watson Research Center*

[⋆]cwong1@uiuc.edu

The design of memory consistency models is difficult since the ideal model should be easy to use yet allow an implementation that provides good performance. Memory model design is even more difficult for programming languages since the audience is much wider. The Java Memory Model (JMM) demonstrates the difficulty of designing a memory consistency model for a language. For example, a common idiom (the "double check idiom") is unsafe with the JMM. In this paper, we describe the design of an optimizing compiler infrastructure in the Pensieve project that allows users to select or customize the memory consistency model for the code to be compiled. The compiler uses the input model to constrain optimizations and code generation. To avoid overly-conservative code generation, the compiler uses rigorous analyses to enable more optimizations to be performed. This includes escape analysis, Shasha and Snir's delay set analysis and our Concurrent Static Single Assignment Form program representation. When completed, the compiler will serve as a testbed to prototype new memory models, and to measure the effect of different memory models on program performance. Moreover, programmers will be able to select suitable memory models for their specific application use.

The Architecture of a World-Wide Distributed Repository to Support Fine-Grained Sharing of Source Code

Jeffrey Mark Siskind
School of Electrical and Computer Engineering
Purdue University
qobi@purdue.edu

There has been an explosion in free software over the past few years. Vehicles like FSF, GNU, CVS, RPM, Linux, and freshmeat.net allow programmers to share hundreds of millions of lines of source code. These vehicles, however, support only coarse-grained source-code sharing. The unit of sharing is a complete package. And packages are monolithic. About the only thing one can easily do with packages obtained from such vehicles is install them. While a package might contain a collection of procedures and type declarations that implement some functionality that a programmer might wish to reuse in a different system, it is difficult to find which package contains that functionality, extract that functionality from its original package, and import it into a new system. In this talk, I present a new vehicle, called October, that is designed to support fine-grained source-code sharing: sharing at the level of individual procedure and type declarations. Unlike CVS, which allows many people to share in the development of the same system, October allows many people to share in the development of different systems.

October is organized like the Web. Instead of Web pages there are top-level definitions. Top-level definitions are stored in a distributed repository implemented by October servers that are analogous to Web servers. They are viewed and manipulated by October browsers that are analogous to Web browsers. October servers and browsers communicate via the October protocol which plays the role of HTTP and allows different server and browser implementations to interoperate. The October protocol is designed from the start with support for searching, caching, mirroring, and load balancing. These play the role of search engines, proxy servers, and Web caches. Search is currently based on string matching, though future plans call for type-based search supported by local and global type inference built into the October protocol and implemented by the October servers. Top-level definitions are stored as abstract syntax trees (ASTs) which play the role of HTML. Instead of URLs there are version locators. Top-level definitions are hyperlinked via embedded version locators. When building an executable, the browser crawls the repository to fetch all top-level definitions referenced directly or indirectly by the top-level definition being built.

October is designed to be programming-language and programmer-preference neutral. It is neither a new programming language nor a new compiler. Rather, it supports existing programming languages and compilers. Just as the Web supports different document styles via DTDs, October supports different programming languages and programmer preferences via programming language definitions (PLDs). A novel aspect of PLDs is that they separate the definition of the abstract syntax for a given language from the mapping between abstract and concrete syntax. This allows users to configure their browsers to dynamically render the code they view in a different concrete syntax

according to their personal preference. PLDs are currently written to support Scheme, C, and Java.

The overall goal of October is to boost world-wide programmer productivity by encouraging an unprecedented degree of source-code sharing, shifting the prevalent mode of programming from implementation to augmentation. In this talk, I will describe how October is designed to support this goal, discuss how this goal motivates and influences the creation of a new infrastructure, present some of the technical problems and design tradeoffs addressed so far while creating this infrastructure, and give a live demo of the prototype implementation of October.

GNUnet - An Anonymous Network

Christian Grothoff
Purdue University
grothoff@cs.purdue.edu

This talk describes the design of a censorship-resistant distributed file sharing protocol which has been implemented on top of GNUnet, an anonymous, reputation-based, network. We describe the semantics of the networking infrastructure and their implications on GNUnet's design. Finally we describe a solution to the problem of achieving accountability in an anonymous network without using any trusted nodes.

Aspect-Oriented Frameworks For Weaving Design Concerns

Atef Bader[1] *, Faisal Akkawi*[2,⋆] *ăand Tzilla Elrad*[2]
[1] *Lucent Technologies*
[I] *Illinois Institute of Technology*
⋆akkawif@iit.edu

Aspect-oriented technology is a programming paradigm that provides the user with the ability to modularize the representation of crosscutting concerns in order to maximize reusability and ensure flexibility of the software system. In this position paper we present the dynamic Weaver Framework (DWF), which is an aspect-oriented framework that supports the dynamic attachment and detachment of aspects to components at run-time, as well as the capability to add and remove aspects and pointcuts during runtime. This capability is the prime factor that enables us to support reconfigurability of the software system. The need to adapt to environmental changes and cope gracefully with the challenges that may have an impact on performance degradation, safety and liveness properties of the running system requires recongfigurability of both the functional and aspectual properties of the system software.

A Robust Algorithm for Partial Redundancy Elimination in Static Single Assignment Form

Thomas VanDrunen
Purdue University
vandrutj@cs.purdue.edu

Partial redundancy elimination (PRE) is a program transformation that optimizes by identifying expressions that are redundant on at least one (but not necessarily all) execution paths. It inserts computations along edges on which such an expression is not redundant, and then eliminates the expression by reloading its value from a temporary. Chow et al. [Chow97, Kennedy99] devised an algorithm for performing partial redundancy elimination on intermediate representation in static single assignment (SSA) form which does not break SSA. Their algorithm, however, makes assumptions about the name space which may not be valid if other optimizations already have been performed on the program. Moreover, a single execution of their algorithm may expose (but not eliminate) new redundancies, so that several passes may be needed to eliminate all of them. We present an algorithm based on Chow's which makes no assumptions about the name space and eliminates all redundant computations in a single path.

Compiling with Code-Size Constraints

Jens Palsberg[⋆] *and Mayur Naik*
Purdue University
[⋆]palsberg@cs.purdue.edu

Most compilers ignore the problems of limited code space in embedded systems. Designers of embedded software often have no better alternative than to manually optimize the source code or even the compiled code. Besides being tedious and error-prone, such manual optimization results in obfuscated code which is difficult to maintain and reuse. In this talk we present a code-size-directed compiler. We phrase register allocation and code generation as an integer linear programming problem where the desired bound on the code size is expressed as an additional constraint. Our experimental results show that our compiler, when applied to two commercial microcontrollers, can generate code that is as compact as carefully crafted code.

Compilation of a High-Level Quantum Chemistry Language into Efficient Parallel Code

Gerald Baumgartner[1,⋆]*, David E Bernholdt*[2]*, Daniel Cociorva*[1]*, Robert Harrison*[3]*, Chi-Chung Lam*[1]*, Marcel Nooijen*[4]*, J. Ramanujam*[5]*, P. Sadayappan*[1]
[1]*Ohio State University*
[2]*Oak Ridge National Laboratory*
[3]*Pacific Northwest National Laboratory*
[4]*Princeton University*
[5]*Louisiana State University*
[⋆]gb@cis.ohio-state.edu

Many computational models of the electronic structure of atoms and molecules, such as the Coupled Cluster approach, involve collections of very computationally intensive tensor contractions. The development an efficient parallel program for such a computational model is very complex and can be the main limiting factor in the rate of progress of the science. We give an overview of an optimizing compiler we are developing, that will translate high-level tensor contraction expressions into high-performance parallel code. We describe the structure of several optimization components and demonstrate their utility using examples.

A Query Language for Feasible Computations over List and Tree Databases

Dirk Van Gucht[1,⋆], *Ed Robertson*[1], *Larry Saxton*[2]

[1] *Indiana University*

[2] *University of Regina, Canada*

⋆vgucht@cs.indiana.edu

We present a language for querying list and tree-structured objects. These objects permit arbitrary-depth sublists or trees. The language is shown to express precisely the feasible, i.e. polynomial-time, generic computations over such structures. The language controls complexity by carefully restricting the replication of values and limiting the nesting of recursion. We also discuss the relationship of our language and XQuery, a query language for XML databases.

Dynamic Weaving for Building Reconfigurable Software Systems

FAISAL AKKAWI
akkawif@iit.edu
Computer Science Dept.
Illinois Institute of Technology
Chicago, IL 60616

ATEF BADER
abader@lucent.com
Lucent Technologies
Naperville, IL 60655

TZILLA ELRAD
elrad@iit.edu
Computer Science Dept.
Illinois Institute of Technology
Chicago, IL 60616

Abstract

Aspect-oriented technology is a programming paradigm that provides the user with the ability to modularize the representation of crosscutting concerns in order to maximize the reusability and ensure the flexibility of software system. In this position paper we present the Dynamic Weaver Framework (DWF), which is an aspect-oriented framework that supports the dynamic attachment of aspects to components at run-time and also the dynamic detachment of aspects from components. As well as the capability to add and remove aspects and pointcuts during runtime. This capability is the prime factor that enables us to support reconfigurability of software systems. The need to adapt to environmental changes and cope gracefully with the challenges that may have an impact on performance degradation, safety and liveness properties of the running system requires recongfigurability of both the functional and aspectual properties of the system software.

.

Keywords: Dynamic Weaver Framework, reconfigurability, dynamic adaptability.

Introduction

Aspect-oriented software design research [3,4,5,6] has stressed the need to address crosscutting concerns earlier in the design phase in order to avoid the associated code-tangling phenomenon and its undesirable implications.

Software systems go through cycles by which new requirements are introduced that may necessitate changes to their behavioral and structural properties. Some of these changes require invasive modifications. The visitor pattern [7] may reduce the effects of the invasive changes but can't eliminate them. Similarly, few of the structural and behavioral patterns [7], like decorator, adaptor, and proxy patterns, may help reduce the effect of the non-invasive changes but can't eliminate them. In [2] the authors presented the Aspect Moderator Framework which is based on a static proxy that can provide weaving of predefined aspects at runtime, but once compiled we can't change them. DWF enables applications to adapt to changes at run time, because components and aspects are independent from each other's and they are woven at runtime. The component and the aspect in DWF must have a predefined interface, but the users are free to change the class

implementation at runtime. Also, components have no knowledge of the number and type of aspects they are affected by. So we can change the number and the type of aspects associated with a component at runtime by addPointcut() in the aspect class as shown in Figure 2. Other kind of features, like debugging, security, and logging, that may require be activating or deactivating during runtime can benefit from reconfigurability so that different security measures can be introduced or new pointcut added.

Design for change in order to adapt gracefully to the evolving requirements can be farther supported when we consider both the behavioral and structural properties as core elements when addressing dynamic adaptability. While the behavioral property represents the ability to add or alter the behavior of methods in a program, the structural property represents the ability to alter class definitions and the implementation used in the program. There are a number of patterns that are documented in the software pattern literature that show how these patterns can be used to support static and dynamic adaptability of software systems, but these patterns once implemented and compiled will be difficult to alter at runtime

Dynamic Weaver Framework

Recognizing crosscutting concerns when building software systems is crucial to guarantee design and code reuse. And identifying the micro-architectural elements that constitute the skeleton of the crosscutting concerns is even more important for the design and reusability of these concerns. In Figure 1 we illustrate the micro-architectural elements of the dynamic weaver framework.

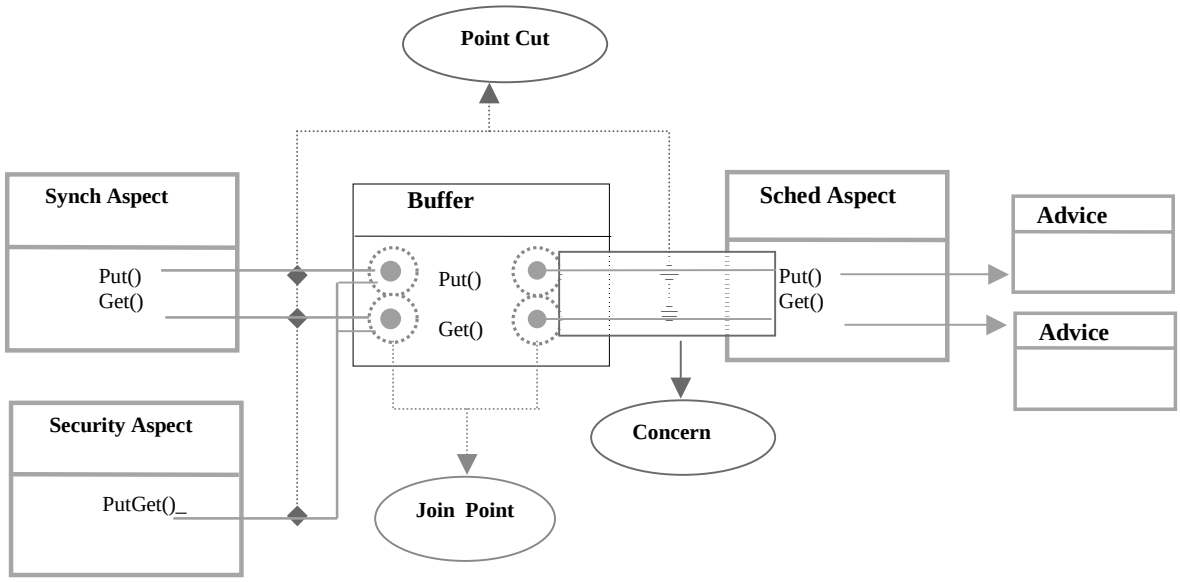


Figure 1: Architectural Elements in the Dynamic Weaver Framework.

Using behavi
since proxies
cases we end up re-implementing each method in the super class or interface and add the

control access code to it, it turns out that proxies are not reusable. Ideally we wish to make aspects generic; but using a proxy for the logging aspect for example would require us to add it the hard way for each method call

Our DWF takes advantage of the dynamic proxy capability in java J2SE[1]. The framework structure is depicted in the class diagram figure 2. Each class uses dynamic proxy class, which represents the aspect weaver class, from the java J2SE. The dynamic proxy class is responsible for creating that proxy object of the initiator object. And each proxy has a number of aspects, and each aspect has a number of pointcuts. The join point in our framework is the method call. Each point cut has an advice class that has two methods beforeAdvice() and afterAdvice() that will be executed when control reaches the join point methods. The semantics of aspect, pointcut, and advice are similar to the ones cited in AspectJ [8].

```
Class Buffer implements BufferIF {

        Put() { ..}
        Get() { ..}
}
```

The Aspect Weaver Framework uses the DynamicProxy class that is part of the J2SE in order to weave classes and their prospective aspects at runtime. The AspectWeaver intercepts the message to the component and redirects it to the AspectRepository. The AspectRepository keeps the information about the aspect(s) (e.g. scheduling, synchronization, security…) to be applied and the order in which they have to be executed. The DWF has a loose coupling between component and aspects, because the component and the aspects no longer have direct reference between them Let us consider an example in which we want to add the debugging concern into the bounded buffer example to show how the framework works.

First we implement the BufferIF

```
Class Buffer implements BufferIF {

        Put() { ..}
        Get() { ..}
}
```

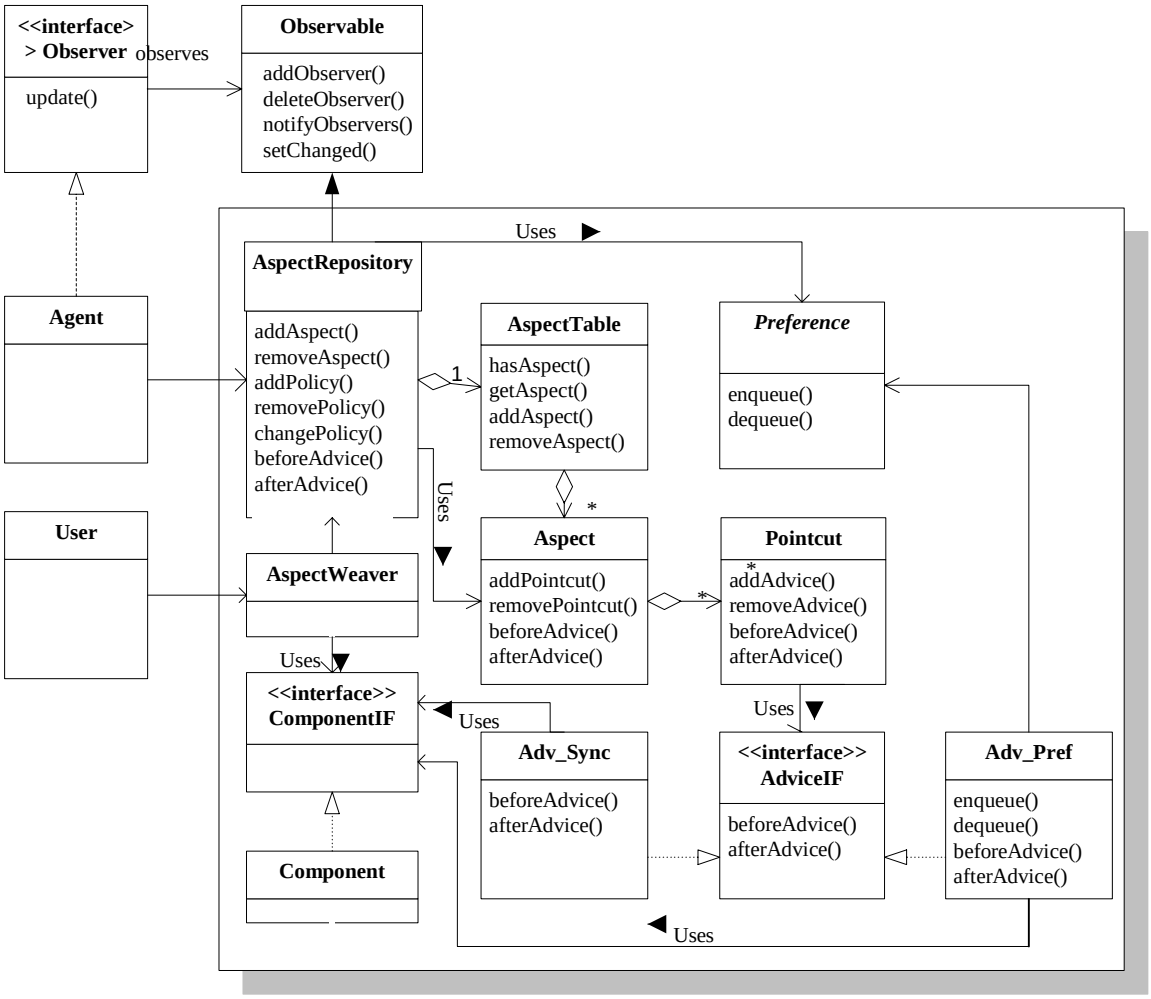


Figure 2. Dynamic Weaver Framework

Then we create the AspectWeaver. The AspectWeaver is the dynamic proxy, which directly interacts with the clients (e.g., producers and consumers using the bounded buffer) and does the actual weaving of aspects at running. All the access from the clients to the bounded buffer will be accomplished through this dynamic proxy. Figure 3 shows the java code for AspectWeaver. Whenever a client calls a method of the bounded buffer, the proxy executes the invoke() method in the AspectWeaver. Inside the invoke(), AspectRepository's beforeAdvice() is called. If the call is successful, the actual operation on buffer is performed by m.invoke() as shown in the code. After this call is completed, AspectRepository's afterAdvice() method is called.

```
public class AspectWeaver implements InvocationHandler {
    private AspectRepository _ar;
    private Object _buf;

    public static Object newInstance(Object buf, AspectRepository ar) {
        return Proxy.newProxyInstance(buf.getClass().getClassLoader(),
                                      buf.getClass().getInterfaces(),
                                      new AspectWeaver(buf, ar));
    }

    private AspectWeaver(Object buf, AspectRepository ar) {
        _ar = ar;
        _buf = buf;
    }

    public Object invoke(Object proxy, Method m, Object[] args)
        throws Throwable {

        Object result = null;
        try {
                _ar.beforeAdvice(_buf, m, args);
                result = m.invoke(_buf, args);
                _ar.afterAdvice(_buf, m, args, result);
        } catch (InvocationTargetException e) {
            throw e.getTargetException();
        }
        return result;
    }
}
Figure3
```

The aspect repository class is responsible for maintaining a list of aspects that are published and registered by other classes, as shown in Figure 4.

```
public class AspectRepository extends Observable
  implements AspectRepositoryIF {
  public static final int ABORT  = 1, BLOCK  = 2, RESUME = 3;
  private AspectTable aspectTbl = new AspectTable();
  .
  .
  public synchronized void beforeAdvice(Object obj, Method m, Object[] args) {
    Task task = new Task(m.getName());
    int i, rc;
    Aspect a;
    Preference policyAdvice;
    boolean mustWait = false;

    for(i = 0; i < aspectTbl.size(); i++) {
      a = aspectTbl.getAspect(i);
      mustWait = false;
      if (occupied)
        mustWait = true;
      else {
        rc = a.beforeAdvice(obj, m, args);
        if (BLOCK == rc)
          mustWait = true;
      }
      while (mustWait) {
```

```
        policyAdvice = (Preference)
          aspectTbl.getAspect("SCHED").getPointcut("DEFAULT").getAdvice(0);
        policyAdvice.enqueue(task);
        try { this.wait(); }
        catch(InterruptedException iex) { /* Ignore iex */ }
        policyAdvice.dequeue(task);
        if (task.equals(taskToBeAwakened)) {

          mustWait = false; // Exit this while loop and
          i = -1; // Force all the aspects to be checked again.
        }
      }
    }
    .
    .
  }
.
.
}
```

Figur 4

The AspectRepository contains the AspectTable, which has all the registered aspects in it. Initially, the table is empty, i.e. no aspect is registered. Adding and removing advices of certain aspect can be done by calling the corresponding methods of the AspectTable. As shown in Figure 5.

```
public class AspectTable {
  private Vector _aspects = new Vector();

  public int size() { .. }
  public boolean hasAspect(String aspect_name) { .. }
  public Aspect getAspect(String aspect_name) { .. }
  public Aspect getAspect(int index) { .. }
  public Aspect addAspect(String aspect_name) { .. }
  public boolean addAspect(String aspect_name, String pointcut_name,
                           String method_name, AdviceIF advice) {
    .
    .
  }
  public void removeAspect(String aspect_name, String pointcut_name,
                           String method_name) {
    .
    .
  }
  .
  .
}
```

Figure 5

AspectTable can contain multiple Aspect objects. Each Aspect can contain multiple Pointcuts. The method addPointcut() is used for adding new advice. The removePointcut() method removes an advice from a given Aspect. The beforeAdvice() and afterAdvice() methods are invoked inside the AspectRepository's corresponding methods as shown in figure 6, each Aspect object has a number of Pointcut objects and a name for itself. The three arguments passed in the beforeAdvice() method are a reference to the shared object, the Method object, and the array of arguments to the specific shared object's method begin called. All these arguments are to be used inside the current beforeAdvice() method or to be passed on to the Pointcut.beforeAdvice() method. Most of the time, not all of these arguments are used. Nevertheless, they need to be passed to the beforeAdvice() method because some of the advices may utilize those information. For example, a logging advice needs to know all the data passed to the shared object. The afterAdvice() method has all the same parameters as those of beforeAdvice() method. Additionally, it has an argument defined as Object result. This represents the result of a specific shared method's method.

```
public class Aspect {
  private String _aspect_name;
  public Hashtable _pointcuts = new Hashtable();
  .
  .
  public int beforeAdvice(Object obj, Method m, Object[] args){
    if (_pointcuts.containsKey("DEFAULT")) {
      Pointcut p = (Pointcut) _pointcuts.get("DEFAULT");
      return p.beforeAdvice(obj, m, args);
    }
    return AspectRepository.RESUME;
  }

  public Object afterAdvice(Object obj, Method m, Object[] args, Object result){
    if (_pointcuts.containsKey("DEFAULT")) {
      Pointcut p = (Pointcut) _pointcuts.get("DEFAULT");
      return p.afterAdvice(obj, m, args, result);
    }
    return null;
  }
}
```

Figure 6

An Aspect can contain multiple Pointcut objects. And each Pointcut can contain multiple Advices. The method addAdvice() is used for adding a new advice. The removeAdvice() method removes an advice from a given Pointcut. The beforeAdvice() and afterAdvice() methods are invoked by the corresponding Aspect's same-named methods. Class Pointcut is shown in Figure 7. Three arguments passed in the beforeAdvice() method are a reference to the shared object, the Method object, and the array of arguments to the specific shared object's method being called. All these arguments are to be passed on to the advice's beforeAdvice() method. Four arguments passed in the afterAdvice() method will also be passed on to the advice's afterAdvice().

```
public class Pointcut {
  private String _pointcut_name;
  public Hashtable _advices = new Hashtable();
  .
  .
  public int beforeAdvice(Object obj, Method m, Object[] args){
    AdviceIF adv = (AdviceIF) _advices.get(m.getName());
    if (adv == null)
      return AspectRepository.RESUME;
    return adv.beforeAdvice(obj, m, args);
  }

  public Object afterAdvice(Object obj, Method m, Object[] args, Object result){
    AdviceIF adv = (AdviceIF) _advices.get(m.getName());
    if (adv != null)
      return adv.afterAdvice(obj, m, args, result);
    return null;
  }
}
```

Figure7

Actual behavior of each aspect is provided by an object whose interface is defined by AdviceIF. They will be woven at runtime by the dynamic proxy, i.e. AspectWeaver. The interface for Advice is shown in Figure 8.

```
public interface AdviceIF {
    public int beforeAdvice(Object obj, Method m, Object[] args);
    public Object afterAdvice(Object obj, Method m, Object[] args, Object result);
}
```

Figure 8

Shown below is the implementation of the synchronization advice for the bounded buffer's get() method. The beforeAdvice() method returns one of the integer constants defined in the AspectRepository; RESUME, BLOCK, and ABORT

```
public class Adv_SyncGet implements AdviceIF {
  public int beforeAdvice(Object obj, Method m, Object[] args) {
    BoundedBuffer bb = (BoundedBuffer) obj;
    if (bb.getCnt() > 0 ) {
      return AspectRepository.RESUME;
    } else {
      return AspectRepository.BLOCK;
    }
  }

  public Object afterAdvice(Object obj, Method m, Object[] args, Object result) {
    return null;
  }
}
```

Dynamic Weaving and Reconfigurability

The Dynamic Weaver Framework has a number of advantages; it provides the capability to add and remove aspects as well as pointcuts at runtime. This capability is the prime factor that enables us to support reconfigurability in order to adapt to environment changes and cope gracefully with any challenges that may have an impact on performance degradation and safety and liveness properties of the running system. Changes to the software systems may affect the structural or behavioral properties. Although the Visitor pattern [7] may reduce the severity of the invasive changes and the Decorator and Proxy pattern may support the engineering of the non-invasive changes, these patterns offer very little to support the design and implementation of the crosscutting concerns, especially in the cases where we desire to alter, add, or remove these crosscutting concerns at run time from the running system. The DWF represents the solution for all of these problems that are hard to anticipate fully during the design phase.

For instance, we may need to add a new pointcut or add a join point to a list of methods that comprise the point cut. An example of this is the addition of a new method *gget()* to the pointcut of the synchronization aspect, or altering the advice class for a certain joint point, like changing the scheduling point cut for the method get, to service requests based on the priority of the thread.

Conclusion

In this paper we presented an approach by which aspects and components can be weaved, altered or removed dynamically. This approach is a step toward automating the weaving process at runtime. Recognizing the micro-architectural elements of crosscutting concerns during the design phase is essential to ensure the reusability and reconfigurability of the resulting software system, and the DWF is an attempt to meet these desirable properties when crafting software systems.

References

[1] http://ww.java.sun.com/jdk1.3.

[2] Constantinos Constantinides, Atef Bader, and Tzilla Elrad. A Framework to Address a Two-Dimensional Composition of Concerns. Position paper to the OOPSLA ’99 First Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems.

[3] Gregor Kiczales, John Lamping,Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-MarcLoingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP ’97*. LNCS 1241.Springer-Verlag, pp. 220-242. 1997.

[4] Bedir Tekinerdogan and Mehmet Aksit. *Deriving Design Aspects from CanonicalModels*. Position paper in ECOOP ’97 workshop on As-pect- Oriented Programming.

[5] Harold Ossher and Peri Tarr. Multi-Dimensional Separation of Concerns in Hyperspace.Position paper at the ECOOP ’99 workshop on Aspect-Oriented Programming.

[6] Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In Workshop on Advance Separation of Concerns, OOPALA, Minneapoils, USA, 2000.

[7] E. Gamma, R. Helm, R. Johnson & J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[8] The aspectj web site. Http://www/aspectj.org/.

A Robust Algorithm for Partial Redundancy Elimination in Static Single Assignment Form

Thomas VanDrunen

April 4, 2002

Abstract

Partial redundancy elimination (PRE) is a program transformation that optimizes by identifying expressions that are redundant on at least one (but not necessarily all) execution paths of a control flow graph of the program. It inserts computations along control flow edges on which such an expression is not redundant, and then eliminates the expression by reloading its value from a temporary. Chow et al. [5, 11] devised an algorithm for performing partial redundancy elimination on intermediate representation in static single assignment (SSA) form which preserves that form across the transformation. Their algorithm, however, makes assumptions about the name space which may not be valid if other optimizations already have been performed on the program. Furthermore, there are some cases where the algorithm exposes redundancies but does not eliminate them, which requires the optimization to be performed more than once to eliminate all redundant computations. We present an algorithm based on Chow's which makes no assumptions about the namespace and eliminates all redundancies in a single pass.

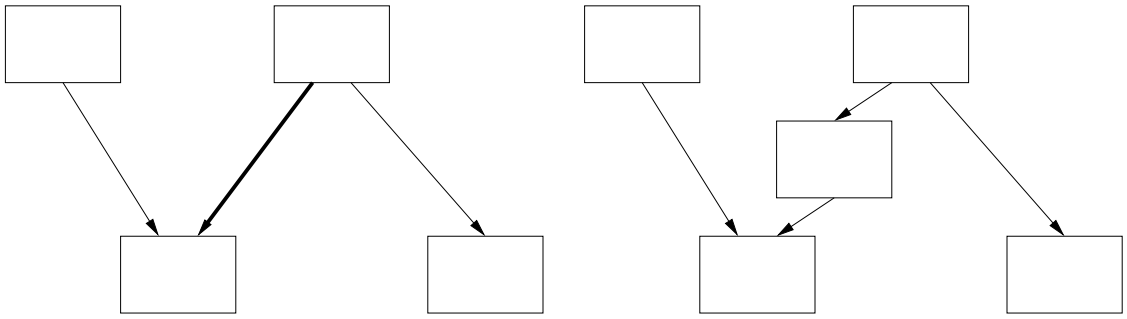
1 Introduction

While machine-level program optimization advances alongside of research in computer architecture, the optimizing of intermediate representation (IR) also enjoys research attention. Besides being of theoretical interest because of their generality and their amenability to proofs of correctness, these optimizations are useful in compiler construction because they apply to compilation from almost any source language to almost any target architecture. Even though system-dependent liabilities exist, causing effectiveness to vary (for example, keeping a value in a register instead of recomputing it or writing to memory may increase register pressure and hurt performance by causing spills – it depends on the number of registers and the cost of memory access), reducing the number of computations on a control flow path will almost certainly improve performance at least modestly.

1.1 Partial redundancy elimination

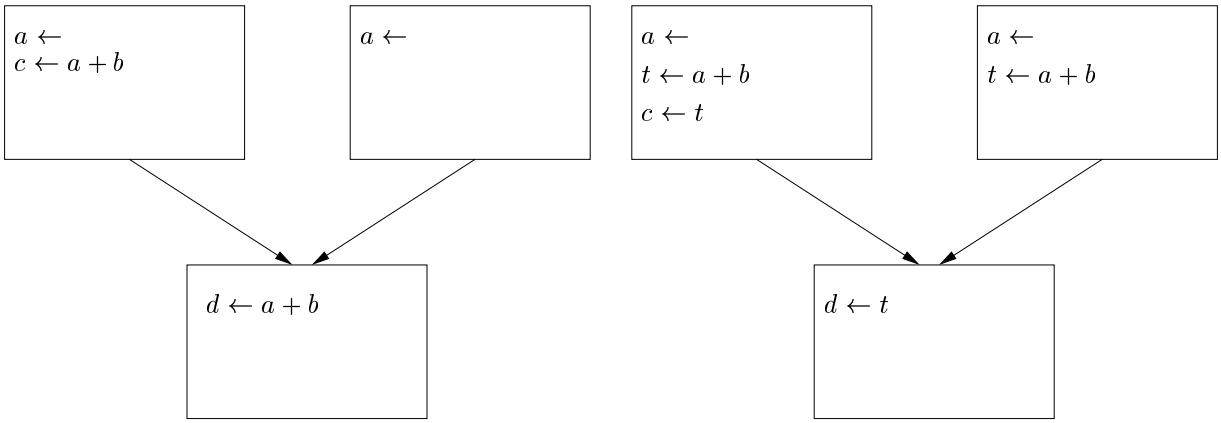
Partial redundancy elimination (PRE) is one such program transformation. Pioneered by Morel and Renvoise [14], it identifies computations that may have been performed at an earlier point in the program and replaces such computations with reloads from temporaries. We say "*may* have been performed" because PRE considers computations that were performed on some, but not necessarily all, control flows paths to that point – in other words, computations that are *partially* redundant. In such cases, a computation must be inserted along paths to that program point that do not already compute that value, and care must be taken so that the length of no path is ever increased and no execution of the optimized program performs a computation which an execution of the unoptimized program would not have done. The net effect is to hoist code to earlier program points. This transformation subsumes common subexpression elimination and loop-invariant code motion.

Throughout this paper, we will discuss transformations as they would be performed on an IR that uses a *control flow graph* (CFG) over *basic blocks*. A basic block is a code segment that has no jump or branch statements except for possibly the last statement, and none of its statements, except possibly the first, is a target of any jump or branch statement. A CFG is a graphical representation of a procedure that has basic blocks for nodes and whose edges



(a) Critical edge highlighted (b) With the critical edge having been removed

Figure 1: Critical edge example



(a) Before PRE (b) After PRE

Figure 2: PRE example

represent the possible execution paths determined by jump and branch statements. We assume that all critical edges have been removed from the CFG, that is, edges from blocks with more than one successor to blocks with more than one predecessor. Such an edge can be eliminated by inserting an empty block between the two blocks connected by the critical edge, as illustrated in Figure 1. We further assume that the instructions of the intermediate representation that interest us are in three-address form, saving their result to a variable or temporary.

Another useful abstraction for control flow is the *dominator tree*. The nodes in the dominator tree are the same as the nodes of the control flow graph, but the children of a given node are the nodes that are immediately dominated by that node, that is, nodes that are dominated by it and are not dominated by any other node also dominated by it (note that this excludes the node itself). The *dominance frontier* of a basic block is the set of blocks that are not dominated by that block but have a predecessor that is.

Having described the basics of program representation, we now consider an example of PRE performed on a code fragment in that representation. In the unoptimized version of this fragment in Figure 2(a), the expression $a + b$ is redundant when the left control flow path is taken, computed twice without any change to the operands. However, it is not redundant along the right edge; even if it were computed sometime earlier in the program, the predecessor block along that edge modifies one of the operands. Thus the expression is *partially redundant* in the bottom basic block.

To remove the partially redundant computation, we insert a copy of it in the right predecessor, since the value of the computation is unavailable along the incoming edge from it. This insertion makes the expression in the bottom basic block fully redundant. If we allocate a fresh temporary t and save each early computation to it, the later computation

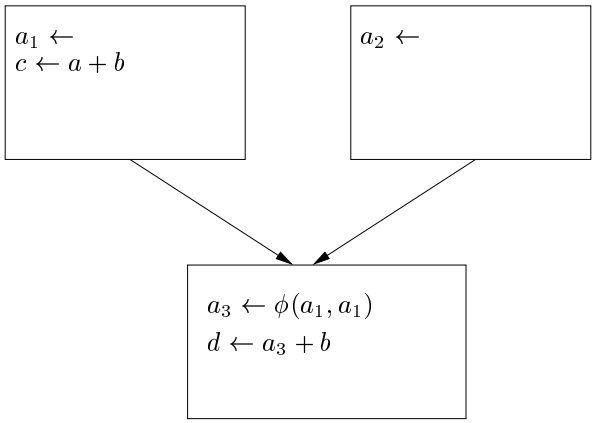


Figure 3: SSA form

can be replaced by that temporary. In this way we produce the optimized code found in Figure 2(b).

Notice that PRE does not reduce code size. Both the optimized and unoptimized versions in this example contain two occurrences of the computation. Moving the code, however, decreases the length of the left execution path. The length of the right execution path is unchanged.

1.2 Static single assignment form

Static single assignment (SSA) form is an intermediate representation property where each variable – whether representing source-level variables chosen by the programmer or temporaries generated by the compiler – has exactly one definition point. Even though definitions in reentrant code may be visited many times in the course of execution, statically each SSA variable is assigned exactly once. If several assignments to a variable occur in the source program, the building of SSA form splits that variable into several versions, one for each definition. Basic blocks where execution paths merge (such as the bottom block in our example) represent the consequent merging of variables by a *phi function*. Phi functions occur only in instructions at the beginning of a basic block (before all non-phi instructions) and have the same number of operands as the block has predecessors in the CFG, each operand corresponding to one of the predecessors. The result of a phi function is the value of the operand associated with the predecessor from which control has come. A phi function is an abstraction for moves among temporaries which would occur at the end of each predecessor – SSA form does not allow such moves since each would have the same variable as its target. For simplicity we will refer to phi functions and the instructions that contain them simply as ϕs. Figure 3 shows the SSA form of the program considered above.

SSA is desirable for many transformations and analyses because it implicitly provides def-use information: For any given use of a variable, the definition relevant to that use is known immediately because a variable (by which we mean, an SSA version of a variable) has only one definition; a given definition's set of uses is simply all places where the target of the definition is used. SSA also relieves the optimizer from depending on the programmer's choice of variable names and the way variables are reused in the program, since a transformation may regard each version of a variable as a completely different variable.

1.3 Related work

As already mentioned, PRE has its origin in the work of Morel and Renvoise [14]. They presented PRE as a generalization of two operations previously regarded as distinct: the elimination of redundant computations and the hoisting of invariant computations out of loops. Drechsler and Stadel extended this work so that subexpressions can be cleanly removed if they are redundant but contained in larger expressions that are not [8]. Thus the transformation generalizes common subexpression elimination as well as invariant code motion.

Knoop et al synthesized several previously known algorithms into a code motion technique that was computationally optimal for eliminating redundant computations [12]. Under the hard constraint of control flow path optimality, their algorithm placed computations as late as possible to reduce register pressure. This algorithm was made more practical by Drechsler and Stadel in [9] and by the original authors in [13].

PRE was also given attention for its usefulness in concert with other program transformations. Briggs and Cooper proposed reassociation and global value numbering as enabling optimizations to make PRE effective [4]. At the same time, Click presented an alternative to PRE: a heuristic code motion technique that did not remove redundant code, followed by a global value numbering phase [6]. This combination, he argued, was more effective than traditional PRE. Value numbering was also used for redundancy elimination in Taylor Simpson's Ph.D. thesis [16]. However, although they have similar effects, value numbering is not completely comparable with PRE, as noted in section 12.4 of [15]. The recasting of the algorithm presented in this paper as a value numbering problem or the use of value numbering in conjunction with it is an area of future investigation (see section 5).

One challenge to combining any set of optimizations is finding a suitable, common program representation. If all optimizations use and preserve the same IR properties, then the task of compiler construction is simplified, optimization phases easily can be reordered or reentered, and expensive IR-rebuilding is avoided. Despite SSA's usefulness as an IR, prior work on PRE not only had not used SSA, but, in the case of [4], explicitly broke SSA before performing PRE. Chow et al presented a PRE algorithm which assumed and preserved SSA form, SSAPRE [5, 11]. It is on that algorithm that the present work is principally based.

Another advancement was made by Bodík et al by considering path-based value numbering as a form of PRE and noticing that more redundancies could be eliminated if changes were made to the CFG [2, 3]. These will be described later in the paper.

The contribution of the present work is a simplification of Chow's algorithm which relaxes restrictions made on the input code and incorporates some of the techniques found in Bodík.

Rest of the paper. Our algorithm is presented as an advancement over the algorithm in [5, 11]. Section 2 summarizes Chow's SSAPRE algorithm and walks through examples on which it fails. Section 3 explains our robust algorithm. Section 4 reports and comments on measurements of the performance of the optimization. Section 5 concludes by considering future work, particularly plans to coordinate this algorithm with other optimizations.

2 Chow's algorithm

2.1 Summary

Chow's algorithm for SSAPRE associates expressions that are lexically equivalent when SSA versions are ignored. For example, $a_1 + b_3$ is lexically equivalent to $a_2 + b_7$. We consider $a + b$ as the canonical expression and expressions like $a_1 + b_3$ as versions of that canonical expression; lexically equivalent expressions are assigned version numbers analogous to the version numbers of SSA variables. A *chi statement*[1] (or simply χ) merges versions of expressions at CFG merge points just as ϕs merge variables. The chis can be thought of as potential phis for a hypothetical temporary that may be used to save the result of the computation if an opportunity for performing PRE is found.

As a ϕ stores its result in a new version of the variable, so χs are considered to represent a new version of the expression. The operands of a χ (which correspond to the incoming edges just as ϕ operands do) are given the version of the expression that is available most recently on the path they represent. If the expression has never been computed along that path, then there is no version available, and the χ operand is denoted by $\perp$. χs and χ operands are also considered occurrences of the expression; expressions that appear in the code are differentiated from them by the term *real occurrences*.

For a code-moving transformation to be correct, it must never insert a computation that will be performed on a path in the optimized version along which it was never performed in the unoptimized version. (Since this optimization does not alter the structure of the CFG, control flow paths in the optimized program correspond to paths in the unoptimized program). In other words, it is incorrect to insert a computation if there exists a path from it to exit along which it is

[1] In [5, 11], χs are called Phis (distinguished from phis by the capitalization) and denoted in code by the capital Greek letter Φ.

Properties for χs

downsafe	The value of the χ is used on all paths to program exit.
canBeAvail	None of the χs operands are $\perp$ or the χ is downsafe (i.e., the value has been computed on all paths leading to this point or insertions for it can be made safely).
later	Insertions for the χ can be postponed because they will not be used until a later program point.
willBeAvail	The value of the χ will be available at this point after the transformation has been made; it can be available and cannot be postponed.

Property for χ operands

insert	An insertion needs to be made so that this χ operand's value will be available.

Table 1: Chi and chi operand properties

	Occurrence being inspected		
Definition	real occurrence	χ	χ operand
real occurrence	Assign the old version if all corresponding variable have the same SSA version; otherwise assign a new version and push the item on the stack.	Assign a new version and push the item on the stack.	Assign the old version if the defining occurrence's variables have SSA versions current at the point of the χ operand; otherwise assign $\perp$.
chi	Assign the old version if all the definitions of the variables dominate the defining χ; otherwise assign a new version and push the item on the stack.	Assign a new version and push the item on the stack	Assign the old version if the definitions of the current versions of all relevant variables dominate the defining χ; otherwise assign $\perp$.

Table 2: How to determine what version to assign to an occurrence of an expression

not used. Not only would this lengthen an execution path (defying optimality), but it could also cause the optimized code to throw an exception that the unoptimized code would not. This property is called *downsafety.* A chi is downsafe if its value is used on all paths to procedure exit. This and other properties of χs and χ operands are summarized in Table 1. Details about their use and how to compute them are discussed later.

SSAPRE has six phases.

χ **Insertion** For any real occurrence of a canonical expression on which we desire to perform SSAPRE, insert χs at blocks on the dominance frontier of the block containing the real occurrence and at blocks dominated by that block that have a ϕ for a variable in the canonical expression. (These insertions are made if a χ for that canonical expression is not there already.) These χs represent places that a version of an expression reaches but where it may not be valid on all incoming edges and hence should be merged with the values from the other edges.

Rename Assign version numbers to all real expressions, χs, and χ operands. This algorithm is similar to that for renaming SSA variables given in [7]. While traversing the dominator tree of the CFG in preorder, maintain a renaming stack for each canonical expression. The item at the top of the stack is the defining occurrence of the current version of the expression. For each block in the traversal, inspect its χs, its real instructions, and its corresponding χ operands in the χs of its successors, assigning a version number to each. If an occurrence is given a new version number, push it on the stack as the defining occurrence of the new version. When the processing of a block is finished, pop the defining occurrences that were added while processing that block. Table 2 explains how to assign version numbers for various types of occurrences depending on the defining occurrence of the current version (this table is expanded from Table 1 of [11]). Note that χ operands cannot be defining occurrences.

Downsafety Compute downsafety with another dominator tree preorder traversal, maintaining a list of χs that have not been used on the current path. When program exit is reached, mark the χs which are still unused (or used

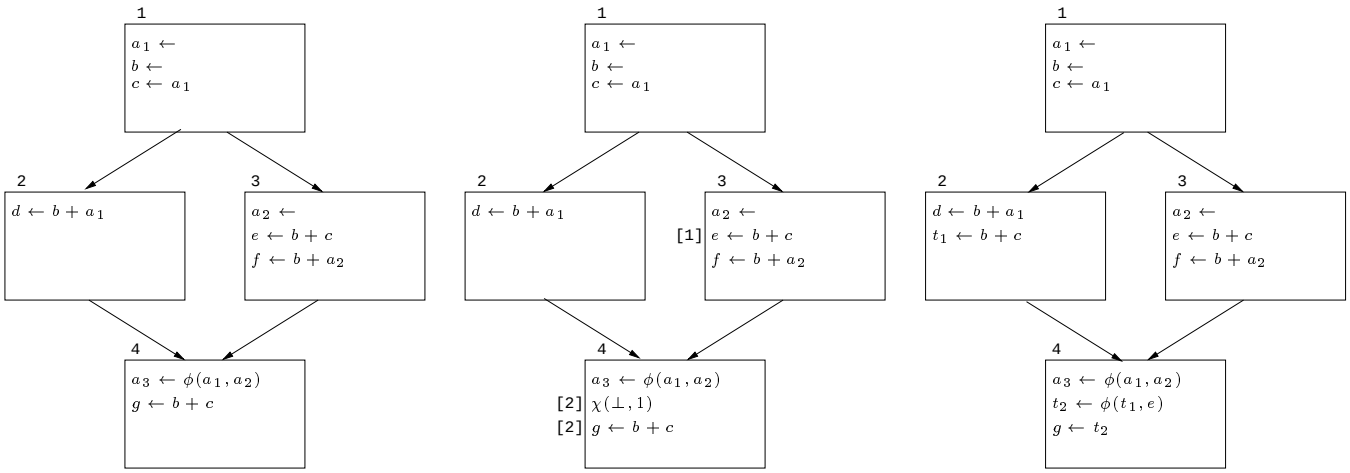


Figure 4: PRE example

only by operands to other χs that are not *downsafe*) as not *downsafe*.

WillBeAvail Compute *canBeAvail* for each χ by considering whether it is *downsafe* and, if it is not, whether its operands have versions that will be available at that point. Compute *later* by setting it to false for all χs with an operand that has a real use for its version or is defined by a χ that is not *later*. Compute *willBeAvail* by setting it true for all χs for which *canBeAvail* is true and *later* is false. Compute *insert* for each χ operand by setting it to true for any operand in a χ for which *willBeAvail* is true and either is $\perp$ or is defined by a χ for which *willBeAvail* is false.

Finalize Using a table to record what use, if any, is available for versions of canonical expressions, insert computations for χ operands for which *insert* is true (allocating new temporary variables to store their value), insert ϕs in place of χs for which *willBeAvail* is true, and mark real expressions for reloading that have their value available at that point in a temporary.

Code Motion Replace all real expressions marked for reloading with a move from the temporary available for its version.

2.2 Example

Consider the unoptimized program in Figure 4 (a). The expression $b + c$ in block 4 is partially redundant because it is available along the right incoming edge. If we perform SSAPRE on this canonical expression, (b) shows the program after χ Insertion and Rename. The occurrence in block 3 is given the version 1. Since block 4 is on its dominance frontier, a χ for this expression is placed there to merge the versions reaching that point, and the right χ operand is given version 1. Since the expression is unavailable from the left path, the corresponding χ operand is $\perp$. The χ itself is assigned version 2. Since that χ will be on top of the renaming stack when $b + c$ in block for is inspected and since the definitions of its variables dominate the χ, it is also assigned version 2.

The χ is clearly *downsafe*, so it can be available. Since its right operand has a version defined by a real occurrence, it cannot be postponed. Therefore, *willBeAvail* is true for it. The Finalize and Code Motion steps insert a computation for its $\perp$ χ operand and a phi in the place of the χ to preserve the value in a new temporary. A move from that temporary can then replace the real occurrence in block 4, as (c) displays.

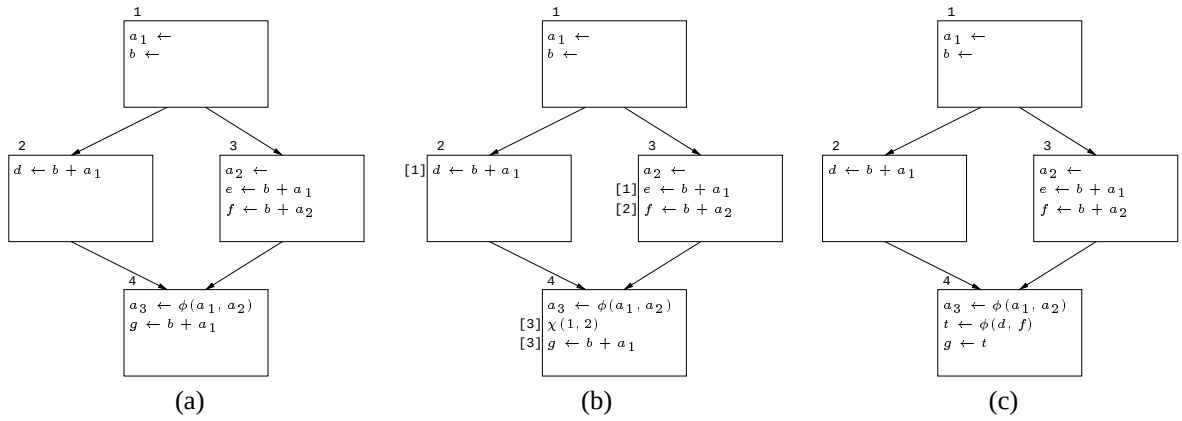


Figure 5: PRE after Constant Copy Propagation

2.3 Counterexamples

The preceding example illustrates how SSAPRE's dependence on lexical equivalence weakens its ability to find all redundancy. Since the definition of c is a move from a_1, $b + c$ has the same value as $b + a_1$ in block 2. This means inserting $b + c$ in that block is not optimal – it may not be redundant lexically, but it clearly computes the same value. Situations like this motivate research for making (general) PRE more effective, such as in [4]. In this case, all we need is a simple constant copy propagation to replace all occurrences of c with a_1, as shown in Figure 5(a). Now the expression in block 4 is fully redundant even in the lexical sense.

Chow's SSAPRE is fragile when used with a transformation like this. It assumes a stronger condition on the code than SSA form, namely that no more than one version of a variable may be live simultaneously. Constant copy propagation breaks this condition here, as both a_1 and a_2 are live at the exit of block 3, and, assuming a_3 is used later in the program, both a_1 and a_3 are live in block 4. In this case, $b + a_1$ is given the same version in blocks 2 and 3; $b + a_2$ in block 3, since it does not match the current version $b + a_2$, is given its own version. See Figure 5(b).

What version should be given to the right operand of the χ in block 4? Version 2 ($b + a_2$) is the current defining occurrence, and Table 2 suggests that its version should be used if its operands are the current versions of the variables. Since a_2 is the corresponding operand to the ϕ at that block, it can be considered current, and we give the χ operand version 2. However, this produces incorrect code – in the optimized program in Figure 5(c), g has value $b + a_2$ on the right path, not $b + a_1$ as it should. [5, 11] give an alternative Rename algorithm called Delayed Renaming which might avoid producing wrong code. In that algorithm, the "current" variable versions for a χ operand are deduced from a later real occurrence that has the same version as the χ to which the operand belongs, adjusted with respect to the ϕs at that block. In this case, the χ has the same version as $g \leftarrow b + a_1$. Since neither b nor a_1 are assigned by ϕs, they are the current versions for the χ operand, and are found to mismatch $b + a_2$; thus the χ operand should be $\perp$. This would still, however, miss the redundancy between $e \leftarrow b + a_1$ and $g \leftarrow b + a_1$.

The entry in Table 2 for processing real occurrences when a χ is on the stack is also fragile. Figure 6 displays another program before, during, and after SSAPRE. Since two canonical expressions ($b + a$ and $b + c$) are being processed, we distinguish their χs and version labels with α and β, respectively. The same program, before SSAPRE but after constant copy propagation, is shown in Figure 7 (a). There is now only one canonical expression. χ Insertion and Rename produce the version in (b). The right χ operand is $\perp$ because a_1 in $b + a_1$ is not "current" on that edge, not being the corresponding operand to the ϕ. Such a χ represents the correct value for $f \leftarrow b + a_3$. However, $g \leftarrow b + a_1$ is also assigned the same version: the χ is the definition of the current version, and the definitions of all the operands of $g \leftarrow b + a_1$ dominate it, which are the conditions prescribed by Table 2. Consequently, the optimized version assigns an incorrect value to g in (c).

In both cases, the problem comes from the algorithm assuming that only one version of a variable can be current at a given time. Simple fixes are conceivable, but not without liabilities. Renaming variables to make this assumption valid will merely undo the optimizations intended to make PRE more effective. The rules in Table 2 could be made

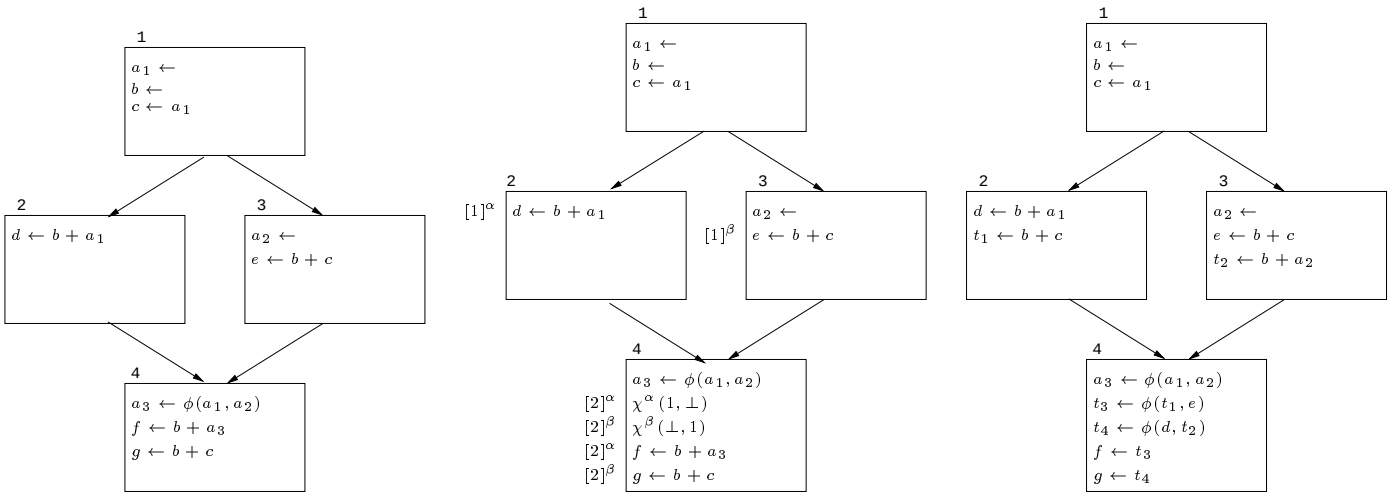


Figure 6: PRE example

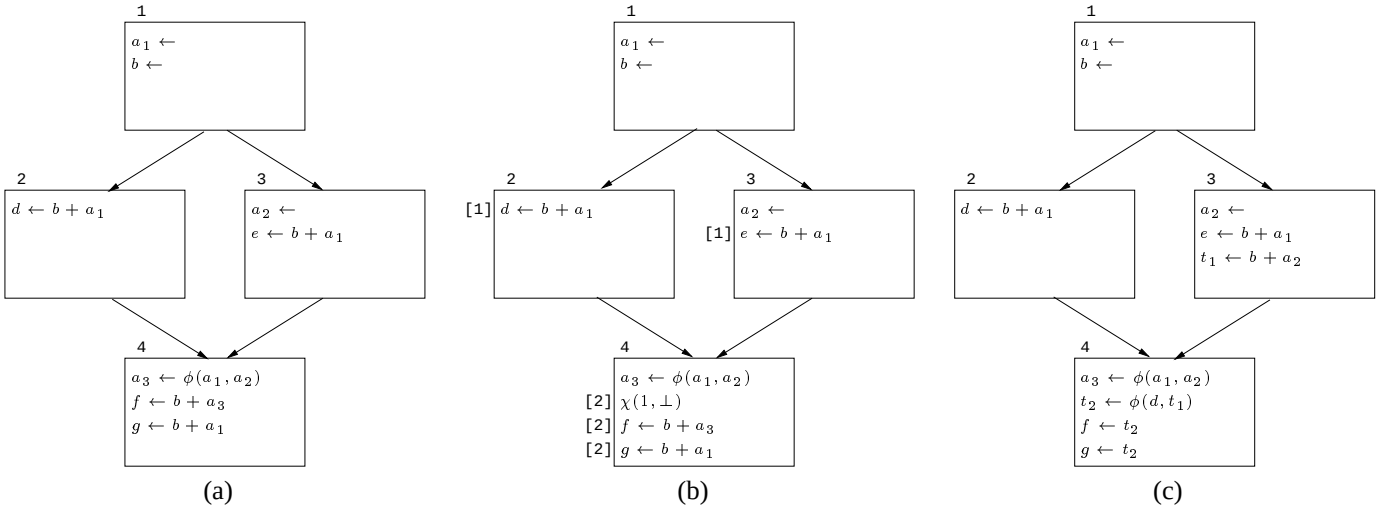


Figure 7: PRE after Constant Copy Propagation

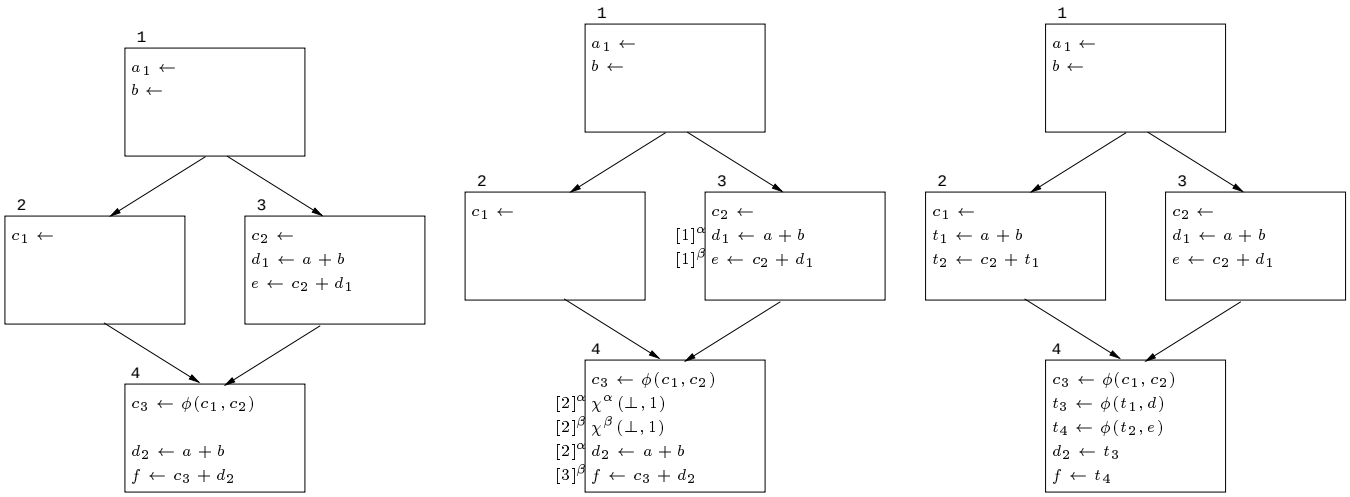


Figure 8: Nested redundancy

more strict; for example, we could associate versions of variables with each χ and make sure all versions match before assigning a real occurrence the same version as a χ. This would prevent the incorrect result in Figure 7, but since $b + a_1$ in the original is indeed partially redundant, ignoring it would not be optimal.

Another case when SSAPRE misses redundancy is when a redundant computation has subexpressions. Consider the program in Figure 8(a). The expression $a + b$ in block 4 is clearly partially redundant; $c_3 + d_2$ is partially redundant also, since d_1 and d_2 have the same value on that path. Essentially, it is the complex expression $c + a + b$ that is redundant.

Figure 8(b) shows the program after χ Insertion and Rename. As earlier, version numbers and χs for the two expressions are distinguished with Greek letters. $d_2 \leftarrow a + b$ is given the same version as the corresponding χ and will be removed by later SSAPRE phases. However, $f \leftarrow c_3 + d_2$ contains d_2, whose definition does not dominate the χ^{β}. According to Table 2, we must assign the real occurrence a new version, even though the χ represents the correct value on the right path. The optimality claim of [5, 11] assumes that a redundancy exists only if there is no non-ϕ assignment to a variable in the expression between the two occurrences; this example frustrates that assumption, and comparing this with the running example in [4] demonstrates that this situation is not unrealistic when PRE is used with other optimizations. The optimized form we desire is in Figure 8(c), which would result if Chow's SSAPRE were run twice. (This situation is not just a product of our three-address representation, since it is conceivable that d has independent uses in other expressions.)

3 Algorithm

The problems with Chow's algorithm stem from assumptions about lexical equivalence. From this point on, we will abandon all notions of lexical equivalence between distinct SSA variables. We will ignore the fact that some SSA variables represent the same source-level variables and consider any variable to be an independent temporary whose scope is all instructions dominated by the its definition.

Our algorithm searches backward from an occurrence of an expression for an earlier computation of that expression's value or for a merge point, at which it will place a χ. A key difference between this and Chow's algorithm is that it searches backwards from expressions to be eliminated rather than forwards from expressions that provide values. If the algorithm discovers the definition of a variable in the expression for which it is searching, it will alter the form of that expression accordingly; for example, if it is searching for $t_3 + t_7$ and discovers the instruction $t_7 \leftarrow t_1$, then from that point on it will look for $t_3 + t_1$. When a χ is created, it is allocated a fresh temporary, which will be the target

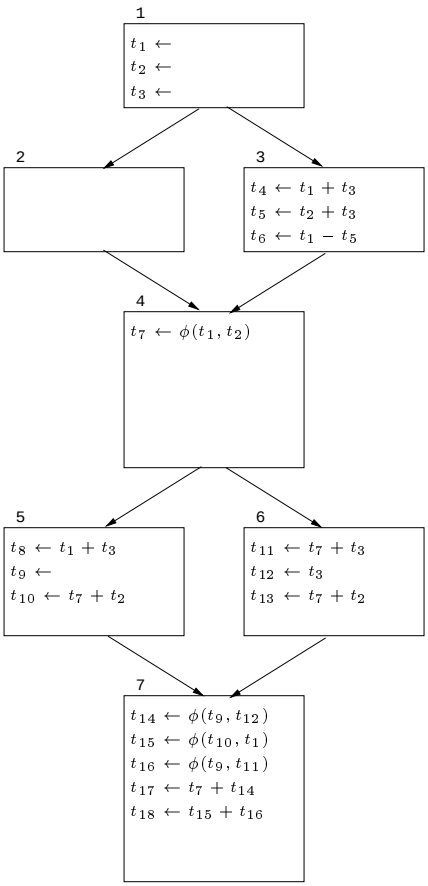


Figure 9: Unoptimized

of the ϕ that will be placed there if the χ is used in a transformation. The result of this phase is that instructions are assigned temporaries that potentially store earlier computations of the instructions' values. The instructions may be replaced by moves from those temporaries.

Since all variables are distinct, so are all expressions. This relieves the need of the notion of canonical expressions and the need for version numbers. Thus our algorithm has no renaming phase. Not only do the problems in Chow's algorithm come from ambiguities in the specification of Rename, but also our experience with implementation has found Rename to be particularly prone to bugs.

The algorithm then analyzes the χ operands that are $\perp$ to determine which ones represent *downsafe* insertions and analyzes the χs to determine which ones can be made available, which ones can be postponed, and, from these, which ones should be made into ϕs. This happens in the two phases Downsafety and WillBeAvail.

Finally, where appropriate, insertions are made for χ operands, ϕs are put in the place of χs, and redundant computations are replaced with moves from temporaries. This phase, Code Motion, subsumes the Finalize phase in Chow's algorithm.

The next subsections describe the four phases of Robust SSAPRE (RSSAPRE) in detail. We clarify each step by observing what it does on a running example. The unoptimized version is in Figure 9. Assignments with no right hand side are assumed to come from sources we cannot use in RSSAPRE. For example, t_1, t_2, and t_3 in block 1 may be parameters to the procedure, and t_9 in block 5 could be the result of a function call.

3.1 χ Insertion

χ Insertion is the most complicated of the phases, and it differs greatly from the equivalent phase in Chow's SSAPRE. A χ is a potential ϕ or merge point for a hypothetical temporary, and it has an expression associated with it, for which the hypothetical temporary holds a pre-computed value. The hypothetical temporary is allocated at the time the χ is created, so it is in fact a real temporary from the temporary pool of the IR; it is hypothetical only in the sense that we do not know at the time it is allocated whether or not it will be used in the output program. The hypothetical temporary also provides a convenient way of referring to a χs, since the χ is the hypothetical definition point of that temporary. In the examples, a χ will be displayed by the Greek letter followed by its operands in parentheses and its hypothetical temporary in parentheses. The expression it represents is written in the margin to the left of its basic block.

A χ operand is an object that is associated with a CFG edge and either points to a temporary or is $\perp$. They are represented in the examples accordingly, listed from left to right in the same order as their respective CFG edges appear. However, an important feature of our algorithm is that χs at the same block can share χ operands. Assume that if two χ operands are in the same block, are associated with the same CFG edge, and point to the same temporary, then they are the same object. χ operands that are $\perp$ are numbered to show which ones represent distinct objects. The temporary to which a χ operand points is called the *definition* of that χ operand, and if the χ operand is $\perp$, we say that its definition is *null*. χ operands also have expressions associated with them, as explained below, but to reduce clutter, they are not show in the examples.

Real occurrences of an expression (ones that appear in the code and are saved to a temporary) also have definitions associated with them, just as χ operands do. This is the nearest temporary (possibly hypothetical) found that contains the value computed by the instruction. If no such temporary exists, then the definition is $\perp$. In the examples, this temporary is denoted in brackets in the margin to the left of its instruction. This notion of definition creates a relation among real instructions, χs, and χ operands and is equivalent to the *factored redundancy graph* of [5, 11].

χ insertion performs a depth-first, preorder traversal over the basic blocks of the program. In each basic block, it iterates forward over the instructions. For each instruction on which we desire to perform PRE (in our case, binary arithmetic operations), the algorithm begins a search for a definition. During the search, it maintains a search expression e, initially the expression computed by the instruction where it starts. It also maintains a reference to the original instruction, i. Until it reaches the preamble of the basic block (which we assume includes the ϕs and χs), the algorithm inspects an instruction at each step.

The current instruction being inspected is either a computation of e, the definition point of one of the operands of e, or an instruction orthogonal to e. This last case will probably cover most of the instructions inspected, and such instructions are simply ignored. If the instruction is a computation of e (that is, the expression computed exactly matches e), then the search is successful, and the temporary in which the current expression stores its result is given as the definition of i. When the current expression defines one of e's operands, then what happens depends on the type of instruction. If it is not itself a candidate for PRE or a simple move (for example, a function call), then nothing can be done; the search fails, and i's definition is null. If the current instruction is a move, then we emend e accordingly: the occurrences in e of the the target of the move are replaced by the move's source. The search then continues with the new form of e. If a constant copy propagation has been performed immediately before RSSAPRE, then this should not be necessary, since simple moves will be eliminated. However, we make no assumptions about the order in which optimizations are applied, and in our experience, such moves proliferate under many IR transformations.

If the current instruction stores the result of a PRE-candidate computation to an operand of e, then we can conjecturally emend e. Since blocks are processed in depth-first preorder and instructions processed by forward iteration, we know that the instruction being inspected has already been given a definition. If that definition is not null, then we can conjecture that the instruction will be replaced by a move from the temporary of its definition to the temporary being written to, and we emend e as if that move were already present. From a software engineering standpoint, such conjectural emendations may be skipped during early development stages, and Section 4 shows results both with and without them. However, they are necessary for optimality, since they take care of the situation in Figure 8. Without them, RSSAPRE would require multiple runs to achieve optimality, especially in concert with the enabling optimizations described in [4]. If the instruction's definition is null (or if conjectural emendations are turned off), such an instruction should be treated the same way as non-PRE-candidates, and the search fails.

When the search reaches the preamble of a block, there are three cases, depending on how many predecessors the

block has: zero (in the case of the entry block), one (for a non-merge point), or many (for a merge point). At the entry block, the search fails. At a non-merge point, the search can continue starting at the last instruction of the predecessor block with no change to i or e. At a merge point, the algorithm inspects the expressions represented by the χs already place in that block. If one is found to match e, then that χ – or more properly, that χ's hypothetical temporary – is given as i's definition. If no suitable χ is found, the algorithm creates one; a new temporary is allocated (which becomes i's definition), and the χ is appended to the preamble and to a list of χs whose operands need to be processed, as will be described below. In either case, the search is successful.

When all basic blocks in the program have been visited, the algorithm creates χ operands for the χs that have been made. Since χ operands have definitions just like real instructions do, this involves a search routine identical to the one above. The only complications are χ-operand sharing among χs at the same block and emendations with respect to ϕs and other χs. For each χ c in the list generated by the search routine and for each in-coming edge η at that block, the algorithm determines an expression for the χ operand for that edge. To do this, we begin with the χ's own expression, e, and inspect the ϕs at the block and the χs that have already been processed. If any write to an operand of e (that is, if one of e's operands is the target of a ϕ or the hypothetical temporary of a χ), then that operand must be replaced by the operand of the ϕ that corresponds to η or the definition of the χ operand that corresponds to η. (If such a χ operand is $\perp$, then we can stop there and know that the χ operand we wish to create will be $\perp$, and it is impossible to make an expression for it.) For example, if $e = t_4 + t_6$ and that block contains $t_6 \leftarrow \phi(t_1, t_2)$, the left χ operand will have the expression $t_4 + t_1$ and the righe will have $t_4 + t_2$. We call the revised expression e'. Once an expression has been determined, the algorithm inspects the χ operands corresponding to η of all the χs at that block that have already been processed; if any such χ operand has an expression matching e', then that χ operand becomes an operand also of c. This sharing of χ operands is necessary to discover as many redundancies as [2], as the examples will show. If no such χ operand is found, the algorithm creates a new one with e' as its expression, and it searches for a definition for it in the same manner as it did for real instructions, in this case i referring to the χ operand. Note that this may also generate new χs, and the list of χs needing to be processed may lengthen.

The search for a definition for a χ operand has one qualification. Extra caution needs to be made when doing a conjectural emendation. If the current instruction being inspected writes to an operand of e' but is defined by a χ that depends on i (assuming i is a χ operand), then the emendation should not be made. A χ c depends on a χ operand i if i is an operand of c or if i is an operand to a χ that in turn has a χ operand upon which c depends. This situation happens in loops, especially instructions that increment a loop counter. Making an emendation in such a case would cause χs to be generated infinitely.

Consider the program in Figure 9. From the instruction $i = t_4 \leftarrow t_1 + t_3$ in block 3, we search for an occurrence of $e = t_1 + t_3$ and immediately hit the preamble of the block. Since it is not a merge point, the search continues in block 1. The write to t_3 is relevant to e, but since that instruction is not a move or a PRE candidate, the search fails and i is assigned $\perp$. Similarly, the searches from $t_5 \leftarrow t_2 + t_3$ and $t_6 \leftarrow t_1 - t_5$ fail.

For $i = t_8 \leftarrow t_1 + t_3, e = t_1 + t_3$, the search takes us to the preamble of block 4. Since it is a merge point, we place a χ there with expression $t_1 + t_3$ and allocate t_{19} as its hypothetical temporary, which also become the definition of $t_8 \leftarrow t_1 + t_3$. Similarly for $i = t_{10} \leftarrow t_7 + t_3$, we place a χ, allocating the temporary t_{20}; and for $i = t_{11} \leftarrow t_7 + t_3$, a χ with temporary t_{21}. For $i = t_{13} \leftarrow t_7 + t_2, e = t_7 + t_2$, when the preamble of block 4 is reached, we discover that a χ whose expression matches e is already present, and so its temporary, t_{20}, becomes i's definition. Finally, searches from the two real instructions in block 7 produces the χs with temporaries t_{22} and t_{23}, which serves as the real instructions' definitions. Figure 10(a) displays the program at this point.

So far, this phase has generated five χs. The search for definitions for their operands is more interesting. The t_{19} χ has expression $e = t_1 + t_3$. Since none of e's operands are defined by ϕs, e also serves as the expression for the χ's operands. Search from the left operand fails when it hits the write to t_3 in block 1. Search from the right operand, however, discovers $t_4 \leftarrow t_1 + t_3$, which matches e; so t_4 becomes its definition. The t_{20} χ has expression $e = t_7 + t_2$. Since $t_7 \leftarrow \phi(t_1, t_2)$ writes to one of e's operands, the two χ operands will have the expressions $t_1 + t_2$ and $t_2 + t_2$. Since neither have occurred in the program, two $\perp$ χ operands are created. The t_{21} χ has expression $e = t_7 + t_3$. The ϕ changes this to $t_1 + t_3$ for the left edge and to $t_2 + t_3$ for the right edge. The former is identical to the expression for the left operand of the t_{19} χ, so that operand is reused. Searching from the right χ operand discovers $t_5 \leftarrow t_2 + t_3$.

Turning to block 6, the t_{22} χ has expression $t_7 + t_{14}$, which the ϕs change to $t_7 + t_9$ for the left edge and $t_7 + t_{12}$ for the right. On the left side, a search for a definition for the χ operand halts at the non-PRE-candidate write to t_9 in

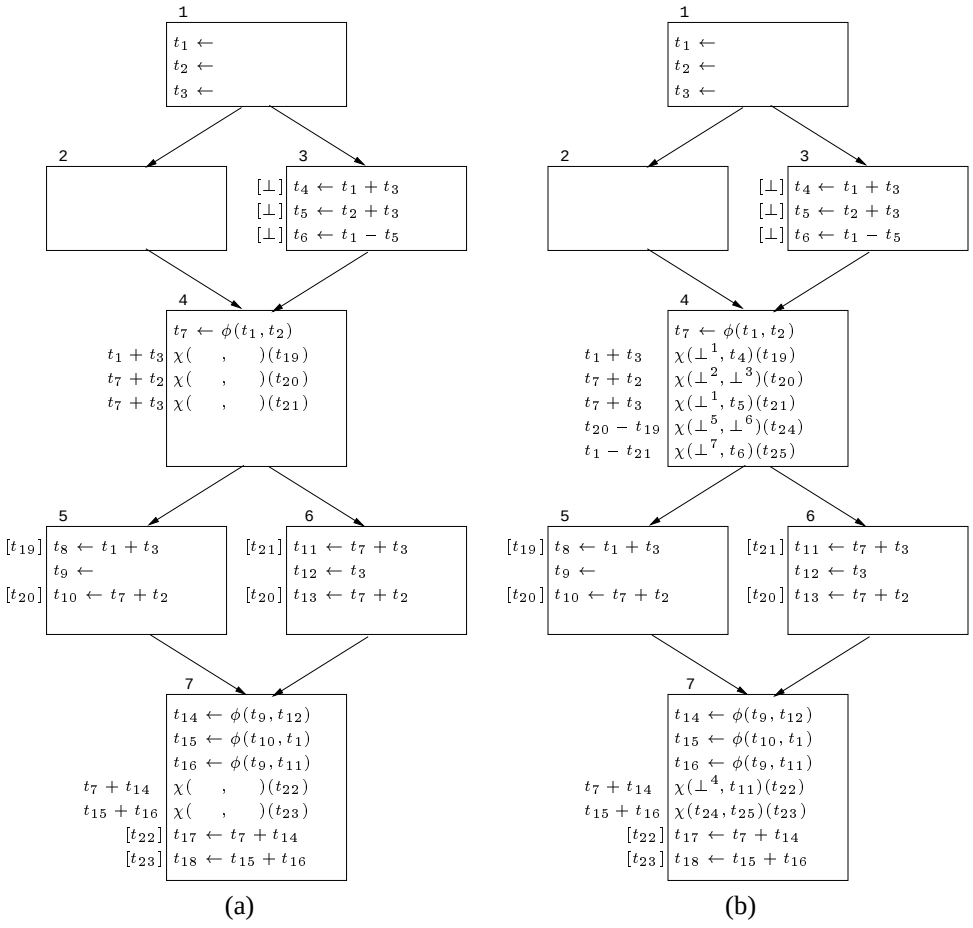


Figure 10: During chi insertion

block 5. The right hand search discovers $t_{11} \leftarrow t_7 + t_3$. The left and right χ operands for the t_{23} χ ($e = t_{15} - t_{16}$) have expressions $t_{10} - t_8$ and $t_1 - t_{11}$, respectively. On the left, $t_{10} \leftarrow t_7 + t_2$ and $t_8 \leftarrow t_1 + t_3$ in block 5 affect the search expression, which we conjecturally emend to $t_{20} - t_{19}$. When the preamble of block 4 is reached, a new χ must be placed, allocating the temporary t_{24}. On the right, $t_{11} \leftarrow t_7 + t_3$ causes the search expression to be emended to $t_1 - t_{21}$, which also requires a new χ at block 4. Finally, we search for definitions for the χ operands of the two χ recently placed in block 4. The program at the end of the χ Insertion phase is shown in Figure 10(b).

3.2 Downsafety

Recall that an inserted computation is *downsafe* only if the same computation is performed on all paths to exit. Avoiding inserting computations that are not *downsafe* prevents lengthening computation paths and causing exceptions that would not be thrown in the unoptimized program. Since χ operands represent potential insertions, we consider downsafety to be a property of χ operands rather than of χs as in [5, 11] and Table 1. The second phase of RSSAPRE is a static analysis that determines which χ operands are *downsafe*. We are interested only in χ operands that are $\perp$ or defined by a χ; if a χ operand's definition is the result of a real instruction, no insertion would ever need to be made for it.

The question of whether a χ operand's value is used on all paths to program exit is akin to liveness analysis for variables and can be determined by a fixed-point iteration. At the entry to a block that is a merge-point, the χ operands of its χs become "live." A use of their value – which happens if a real instruction has the hypothetical temporary of the χ as a definition – "kills" them. All χ operands that are not killed in that block are "live out", and must be considered "live in" for the block's successors.

The fact that χs share χ operands complicates downsafety, since if a χ operand is an operand to several χs, a use of any of their hypothetical temporaries will kill it. Moreover, the use of a χ's hypothetical temporary will kill all of that χ's χ operands. To handle this, in addition to a set of live χ operands, we must maintain on entry and exit of every block a mapping from temporaries to sets of χ operands which a use of them will kill.

What if a χ operand's value is used as the definition to another χ operand? We could consider that χ operand to be a use occurring at the end of the corresponding predecessor block (not in the block that contains its χ). That would complicate things because if the χ operand was the only killer for the first χ operand on a path to exit, then the downsafety of the first would depend on the downsafety of the second. Chow's algorithm handled this by propagating a false value for downsafety from χs already found to be false to χs that depended on them. We can handle this with our map structure: If a use of t_1 will kill χ operand ω_1, χ operand ω_2 has t_1 as its definition, and a use of t_2 will kill ω_2, then we say that t_2 will also kill ω_1. The effect this has is that the use of a temporary as the definition of a χ operand cannot itself kill another χ operand, but that the fate of both χ operands are linked on subsequent paths to exit.

Suppose $live\_in(b)$, $live\_out(b)$, $map\_in(b)$, and $map\_out(b)$ are the live in and out sets and mappings in and out, respectively, for a block b, and that $\Delta(i)$ finds the definition of instruction i. Then to compute downsafety, we must solve the following data flow equations:

$$
\begin{array}{rcl}
live\_in(b) & = & \bigcup_{b' \in pred(b)} live\_out(b') \\
 & = & \cup\text{set of } \chi \text{ operands at } b \\
map\_in(b) & = & \bigcup_{b' \in pred(b)} map\_out(b') \\
 & & \{\bigcup t \mapsto \sigma \mid \chi \text{ at } b \text{ with temporary } t \text{ and set of } \chi \text{ operands } \sigma \\
live\_out(b) & = & live\_in(b) \\
 & & -\{\omega \mid \Delta(i) = t \text{ and } map\_in(t) = \omega \text{for some instruction } i \in b \\
 & & \quad \text{or for some corresponding } chi \text{ operand } i \text{ in a successor}\} \\
map\_out(b) & = & map\_in(b)
\end{array}
$$

Table 3 lists what χ operands are *downsafe* in our example and why. As in the Figures, χ operands are identified by their definition. Of particular interest is $\perp^1$ because it is *downsafe* only because it is shared: it is killed on the left path because it belongs to the t_{19} χ and on the right because it belongs to the t_{21} χ. If these were considered separate χ operands, then a redundancy would be missed.

$\perp^1$	*downsafe*: killed by $t_8 \leftarrow t_1 + t_3$ on the left path and $t_{11} \leftarrow t_7 + t_3$ on the right.
t_4	Irrelevant: defined by real instruction.
$\perp^2$	*downsafe*: killed by $t_{10} \leftarrow t_7 + t_2$ on the left path and $t_{13} \leftarrow t_2 + t_2$ on the right.
$\perp^3$	*downsafe*: killed by $t_{10} \leftarrow t_7 + t_2$ on the left path and $t_{13} \leftarrow t_2 + t_2$ on the right.
t_5	Irrelevant: defined by real instruction.
$\perp^5$	Not *downsafe*.
$\perp^6$	Not *downsafe*.
$\perp^7$	Not *downsafe*.
t_6	Irrelevant: defined by real instruction.
$\perp^4$	*downsafe*: killed by $t_{17} \leftarrow t_7 + t_{14}$.
t_{11}	Irrelevant: defined by real instruction.
t_{24}	*downsafe*: killed by $t_{18} \leftarrow t_{15} - t_{16}$.
t_{25}	*downsafe*: killed by $t_{18} \leftarrow t_{15} - t_{16}$.

Table 3: Downsafety for the running example.

3.3 WillBeAvail

The WillBeAvail stage computes the remaining properties for χs and χ operands, namely, *canBeAvail*, *later*, and *willBeAvail* for χs and *insert* for χ operands. The most important of these is *willBeAvail* because it characterizes χs that will be turned into ϕs for the optimized version.

We first determine whether it is feasible and safe for a χ to be made into a ϕ. If all of a χ's operands either have a real use or are *downsafe*, then that χ is *canBeAvail*. A χ is also *canBeAvail* if all of its χ operands that are not *downsafe* are defined by χs which also *canBeAvail*, since, even though an insertion for that χ operand would not be safe, no insertion is needed if the value will be available from the defining χ.

To this end, after initializing *canBeAvail* to true for all χs in the program, we iterate through all χs. If a χ c has a χ operand that is not *downsafe* and is $\perp$ (and if *canBeAvail* has not already been proven false for c), we set *canBeAvail* to false for it. Then we make an inner iteration over all χs; for any χ that has an operand defined by c, if has a non-*downsafe* χ operand but is still marked *canBeAvail*, then its *canBeAvail* should be cleared in the same manner. In our example, all χs are *canBeAvail* except for the ones with temporaries t_{24} and t_{25}. Since they each have at least one $\perp$ χ operand that is not *downsafe*.

Next, we compute *later*, which determines if a χ can be postponed. This will prevent us from making insertions that do no benefit and would only increase register pressure. *later* is assumed true for all χs that *canBeAvail*. Then we iterate through all χs, and if a χ c is found for which *later* has not been proved false and which has an operand defined by a real occurrence, we reset *later* for it. To do this, we not only set *later* to false, but, similarly to how *canBeAvail* was propagated, we also iterate through all χs; if any is found that has an operand with c as a definition, that χ's *later* is reset recursively. The idea is that if the definitions of any of a χ's variables are available (either because they are real occurrences or because they are χs that cannot be postponed), the χ itself cannot be postponed. In our example the t_{20} χ is *later* because both of its operands are $\perp$. The t_{23} χ is also *later* because both of its operands come from χs for which *canBeAvail* is false. These computations can be postponed.

At this point, computing *willBeAvail* is straightforward. A χ will be available if it can be made available and there is not reason for it not to be – that is, if it is *canBeAvail* and not *later*. In our example, all χs are *willBeAvail* except for the ones associated with t_{20}, t_{24}, and t_{25}. From this we also can compute *insert*, which characterizes χ operands that require a computation to be inserted. If a χ operand is *insert* if it belongs to a *willBeAvail* χ and is either $\perp$ or defined by a χ for which *willBeAvail* is false. Being in a *willBeAvail* χ implies that such a χ operand is *downsafe*. In our example, χ operands $\perp^1$ and $\perp^4$ are *insert*. *willBeAvail* and *insert* can be computed on demand when they are needed in the next phase.

3.4 CodeMotion

The earlier phases having gathered information, the final stage, CodeMotion, transforms the code by inserting ϕs and anticipated computations and eliminating redundant computations. The net effect is to hoist code to earlier program points.

If *willBeAvail* is true for a χ, then the value it represents should be available in a temporary in the optimized program; a ϕ needs to be put in its place to merge the values on the incoming paths. The operands to this new ϕ will be the temporaries that hold the values from the various predecessors. If *insert* is true for any of its operands (indicating that the value it represents is not available, that is, has not been computed and stored in a temporary), then a computation for that value must be inserted at the end of the predecessor block it represents. Any real occurrence whose definition is another real occurrence or a *willBeAvail* χ is redundant, and can be replaced with a move from the temporary holding its value – if it is defined by a χ, the temporary is that which is the target of the ϕ put in place of the χ; if it is defined by a real occurrence, the temporary is the one that stores the result of that occurrence.

Four steps complete the changes to the code: Inserting appropriate computations, creating new ϕs, and eliminating fully redundant computations.

To do the insertions, we iterate over all χ operands. If any is marked *insert*, then we allocate a new temporary, manufacture an instruction which computes the χ operand's expression and stores the result in the fresh temporary, append that instruction at the end of the corresponding basic block, and set the fresh temporary to be the χ operand's definition. In our example, $\perp^1$ requires us to insert $t_{26} \leftarrow t_1 + t_3$ in block 2, where t_{26} is fresh. Similarly, we insert $t_{28} \leftarrow t_1 + t_3$ at block 5 for $\perp^4$.

We then iterate over all χs. For a χ c that *willBeAvail*, we insert a ϕ at the end of the list of ϕs already present at the block. That ϕ merges the temporaries that define c's χ operands into c's hypothetical temporary. Because insertions have been made, all valid χ operands will have temporaries for definitions by this point. In our example, we create $t_{19} \leftarrow \phi(t_{26}, t_4)$ and $t_{21} \leftarrow \phi(t_{26}, t_5)$ in block 4 and $t_{22} \leftarrow \phi(t_{28}, t_{11})$.

Finally, we iterate over all instructions. If any is defined by the target of another real instruction or of a *willBeAvail* χ (which by this time has been made into a ϕ), it is replaced with a move instruction from its definition to its target. In our example, $t_{11} \leftarrow t_7 + t_3$ in block 6 is replaced with $t_{11} \leftarrow t_{21}$ and $t_{17} \leftarrow t_7 + t_{14}$ in block 7 is replaced with $t_{17} \leftarrow t_{22}$.

Figure 11 displays the final program. The improvement can be seen by comparing the number of computations on each possible execution path: left-left, left-right, right-left, and right-right. In the unoptimized program, the number of computations are four, four, seven, seven, respectively; in the optimized program, they are four, three, six, five. The benefit varies among the possible paths, but never is a path made worse.

4 Performance

We have implemented this algorithm for the optimizing compiler of JikesRVM [1], a virtual machine that executes Java classfiles. Our implementation has the option to perform or not to perform conjectural emendations. Table 4 displays the results for running seven benchmarks from the SPECjvm2000 suite. The columns give the results for executing the benchmark without RSSAPRE, RSSAPRE without conjectural emendations, and full RSSAPRE. The numbers are in milliseconds as reported by Java's `System.currentTimeMillis()` call. In each case, the benchmark was run once so that it would be compiled in full by the dynamic compiler, and then (without restarting the virtual machine, avoiding recompiling the benchmarks) run ten times with execution time measured for each run. The large cells give the average time over ten runs, with the range in parentheses. The smaller columns for the two PRE versions measure the number of times the optimization was applied by counting the number of static computations eliminated.

The RSSAPRE runs are consistently better in all cases except for db, where it makes no impact. RSSAPRE with conjectural emendations found only a modest number of optimization opportunities that were not found when conjectural emendations were turned off. Only two cases show it to have a definite impact on performance; while the performance of mpegaudio is better with conjectural emendations, it appears to cause the improvement in compress over the unoptimized version to retreat. This may be because the more aggressive version puts particular pressure on the registers. We speculate that if RSSAPRE were used with the enabling optimizations suggested by [4] that the

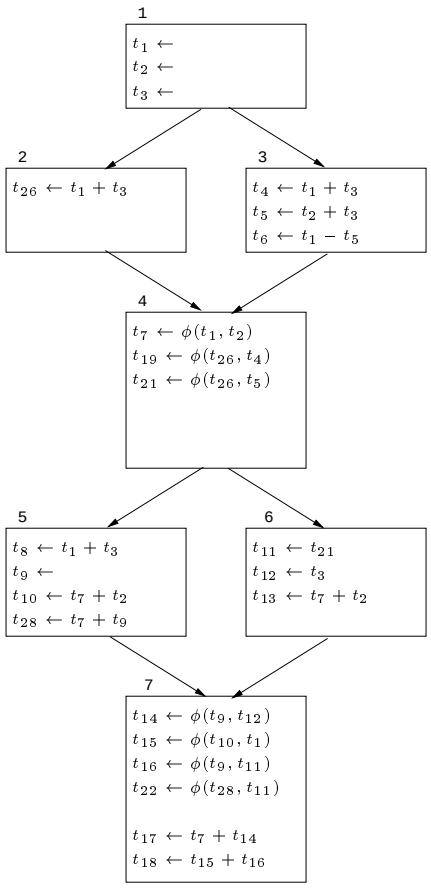


Figure 11: Unoptimized

	nop	pre		pre+	
201 compress	14553 (14449-14798)	13945 (13829-14195)	11	13971 (13853-14208)	12
202 jess	8054 (7653-8610)	8021 (7598-8629)	15	8003 (7596-8613)	15
205 raytrace	5977 (5953-6030)	5830 (5805-5879)	15	5783 (5757-5874)	18
209 db	29484 (25949-33525)	29165 (25896-33610)	7	30059 (25679-33884)	7
222 mpegaudio	12679 (12550-13424)	11771 (11757-11815)	71	11724 (11709-11777)	79
227 mtrt	6413 (6267-6658)	6305 (6135-6583)	15	6243 (6140-6537)	18
228 jack	10557 (10409-10778)	9259 (9094-9493)	16	9243 (9081-9474)	17

Table 4: Performance results

number of transformations, especially with conjectural emendations, would rise greatly, and the true impact would be shown. This is an area of future work, to which we turn now.

5 Future work

As already mentioned, we are eager to implement the reassociation and value-numbering optimizations described in [4] to see how effect they make our algorithm. To our knowledge, these optimizations have not been used with so rich a version of PRE, nor while maintaining SSA form.

In addition to this, the algorithm should be extended to handle array and object references. Throughout this work, we have assumed that the only expressions we are concerned with are arithmetic. Object references can also benefit from PRE, although alias analysis will make it more complicated. Array SSA form [10], used in JikesRVM, will be of benefit here.

The contribution of this work is a simpler, more robust version of Chow's SSAPRE. The error-prone Rename phase has been eliminated, Downsafety has been recast as a version of a standard data flow problem, the algorithm no longer makes assumptions about the namespace, and a wider range of redundancies are eliminated. It is now fit to be used in conjuction with other optimizations.

References

[1] 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 jalapeo virtual machine. *IBM System Journal*, 39(1), February 2000.

[2] Rastislav Bodík and Sadun Anik. Path-sensitive value-flow analysis. San Diego, CA, january 1998.

[3] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant expressions. In *Proceedings of the conference on programming language design and implemenation*, pages 1–14, Montreal, Quebec, June 1998.

[4] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. *SIGPLAN*, 29(6):159–170, June 1994.

[5] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on SSA form. In *Proceedings of the conference on programming language design and implemenation*, volume 32, pages 273–286, June 1997.

[6] Cliff Click. Global code motion, global value numbering. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 246–257, La Jolla, CA, June 1995. *SIGPLAN Notices* 30(6), June 1995.

[7] 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, October 1991.

[8] Karl-Heinz Drechsler and Manfred P. Stadel. A solution to a problem with Morel and Renvoise's "global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, October 1988.

[9] Karl-Heinz Drechsler and Manfred P. Stadel. A variation of Knoop, Rüthing, and Steffen's "lazy code motion". *ACM SIGPLAN Notices*, 28(5):29–38, May 1993.

[10] Stephen Fink, Kathleen Knobe, and Vivek Sarkar. Unified analysis of array and object references in strongly typed languages. July 2000.

[11] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, May 1999.

[12] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*, pages 224–234, San Francisco, CA, July 1992. *SIGPLAN Notices* 27(7), July 1992.

[13] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.

[14] Etienne Morel and Claude Renvoise. Global optimization by supression of partial redundancies. *Communications of the ACM*, 22(2):96–103, 1979.

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

[16] Loren Taylor Simpson. *Value-driven redundancy elimination*. PhD thesis, Rice University, 1996.