Isothermality: Making speculative optimizations affordable

by

David John Pereira
B.Sc., University of Calgary, 2001
M.Sc., University of Calgary, 2003

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

Isothermality: Making speculative optimizations affordable

by

David John Pereira
B.Sc., University of Calgary, 2001
M.Sc., University of Calgary, 2003

Supervisory Committee

Dr. R. Nigel Horspool, Supervisor
(Department of Computer Science)

Dr. M. Yvonne Coady, Departmental Member
(Department of Computer Science)

Dr. William W. Wadge, Departmental Member
(Department of Computer Science)

Dr. Amirali Baniasadi, Outside Member
(Department of Electrical and Computer Engineering)

Supervisory Committee

Dr. R. Nigel Horspool, Supervisor
(Department of Computer Science)

Dr. M. Yvonne Coady, Departmental Member
(Department of Computer Science)

Dr. William W. Wadge, Departmental Member
(Department of Computer Science)

Dr. Amirali Baniasadi, Outside Member
(Department of Electrical and Computer Engineering)

ABSTRACT

Partial Redundancy Elimination (PRE) is a ubiquitous optimization used by compilers to remove repeated computations from programs. Speculative PRE (SPRE), which uses program profiles (statistics obtained from running a program), is more cognizant of trends in run time behaviour and therefore produces better optimized programs. Unfortunately, the optimal version of SPRE is a very expensive algorithm, of high-order polynomial time complexity, and unlike most compiler optimizations, which run effectively in linear time complexity over the size of the program that they are optimizing.

This dissertation uses the concept of "isothermality"—the division of a program into a hot region and a cold region—to create the Isothermal SPRE (ISPRE) optimization, an approximation to optimal SPRE. Unlike SPRE, which creates and solves a flow network for each program expression being optimized—a very expensive operation—ISPRE uses two simple bit-vector analyses, optimizing all expressions simultaneously. We show, experimentally, that the ISPRE algorithm works, on average, nine times faster than the SPRE algorithm, while producing programs that are optimized competitively.

This dissertation also harnesses the power of isothermality to empower another kind of ubiquitous compiler optimization, Partial Dead Code Elimination (PDCE), which removes computations whose values are not used. Isothermal Speculative PDCE (ISPDCE) is a new, simple, and efficient optimization which requires only three bit-vector analyses. We show, experimentally, that ISPDCE produces superior optimization than PDCE, while keeping a competitive running time.

On account of their small analysis costs, ISPRE and ISPDCE are especially appropriate for use in Just-In-Time (JIT) compilers.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

Dad, Mum, Karl, and Maud, for encouraging and consoling me in moments of despair; for believing in me; for your love and dedication.

Nigel Horspool, for his mentoring, support, encouragement, and patience.

NSERC, for funding me with a PGS-B Scholarship.

IBM Corporation, for funding me with an IBM Fellowship.
Thanks in particular to Kelly Lyons, Marin Litoiu, Kevin Stoodley, and Allan Kielstra.

GCC Developers, particularly Danny Berlin, Andrew Pinksi, Diego Novillo, and Janice Johnson.
Thanks for helping me with GCC.

JikesRVM Developers, particularly Ian Rogers, for promptly answering my questions.
Thanks for helping me with JikesRVM.

Colleagues, especially Neil Burroughs, Dale Lyons, and Mike Zastre, who would listen patiently to my academic woes.

Dear Friends, especially Cam & Hana.

And that strife was not inglorious,
though th'event was dire,
as this place testifies
John Milton, *Paradise Lost*
Book 1, 623–625

DEDICATION

PARENTIBUS MEIS
PROPTER OMNIA

ET

EAE
(QUAE MIHI ETIAM SINE NOMINE—
UBI ES, CARISSIMA MEA?)

Chapter 1

Introduction

1.1 Program Optimization via Compilation: An Introduction

Increasing the performance of computer software is a major focus of modern computing. It is a problem that is currently approached at three levels:

1. algorithm designers create more efficient algorithms;
2. hardware designers create architectures capable of higher throughput;
3. *optimizing compilers* implement meaning-preserving transformations on programs (implementing algorithms) so that they may execute more efficiently (on a given hardware architecture).

This dissertation extends the state of the art in the third category specified above—Optimizing Compilers.

Compilers are a crucial part of the software development tool-chain. They obviate the need for tedious and often error-prone hand translation of programs into assembly code, and, in doing so, insulate the programmer from the details of the underlying target architecture and provide program portability. However, compilers must provide translations that are as good as and frequently better than those a human programmer could provide. Indeed, John Backus, the creator of FORmula TRANslator (FORTRAN), one of the first compiled languages, stated[AK02, page 3]:

> It was our belief that if FORTRAN, during its first months, were to translate any reasonable "scientific" source program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger.... To this day I believe that our emphasis on object program efficiency rather than on language design was basically correct. I believe that had we failed to produce efficient programs, the widespread use of languages like Fortran would have been seriously delayed.

Yet, it is true that optimizing compilers currently produce code far superior to that produced by the majority of human translators, leaving one to ask, quite reasonably, whether the study of optimizing compilers is still a viable research topic.

1.1.1 Is Compiler Research Still Necessary?

We can answer this question with a resounding affirmative:

1. **Processors and System Architectures Expect Optimizing Compilers:** There is a fundamental synergy that exists between hardware systems and compilers; architectural features are often designed under the assumption that a compiler will be able to transform a program to take advantage of them.

   Consider, for example, the use of multiple levels of cache. In the absence of an optimizing compiler, an algorithm such as matrix multiplication will access matrix elements in an order which lacks spatial locality (the close proximity of element addresses) thereby rendering the cache less effective. However, a compiler optimization such as "strip mining" will reorder the memory accesses to increase locality, often improving performance by large factors, sometimes by as much as 500%.

2. **Compilers Provide a Cost-Effective Partnership with Hardware**: In order to obtain every last bit of performance from (expensively designed and produced) modern architectures, "help" from their (relatively inexpensive) compilers is often needed.

   For example, in theory, a superscalar architecture can look ahead in the instruction stream to find instructions which can be executed out-of-order. This may seem to obviate the need for a software instruction scheduler. However, when it is realized that the size of the processor's look-ahead window is very limited, the burden falls once again on the compiler to emit a code stream which maximizes the number of independent instructions within the look-ahead window—via a software instruction scheduler. Most importantly, this assistance is inexpensive; it is far cheaper to design and implement a software scheduler in a compiler than to design, verify, and fabricate the logic for a hardware scheduler with a larger lookahead window.

3. **Hardware Processor-Based Optimizations are Fine-Grained:** Processors do indeed optimize programs at the hardware level. For example, hardware units such as branch predictors can prefetch and pre-execute code on the more probable side of a conditional jump instruction, something which cannot be done in software.

   Yet, hardware processors have a very local understanding of program behaviour, in contrast to compilers. For example, compilers for functional languages can perform a program transformation called "deforestation" which removes the intermediate data structures used by a program—a transformation which requires a global symbolic view of a program[Wad88]. This optimization simply cannot, at present, be done by a hardware processor in a cost-effective manner.

4. **Moore's Law:** While hardware advances have caused a ten-fold increase in computing power every decade until now, advances in hardware design and fabrication processes alone may not be enough to guarantee that this trend will continue well into the future—The aggressive transformation of computer programs by compilers into equivalent, more efficient formulations will have to play a crucial role in increasing software performance.

5. **Future Languages:** Most of our current programming languages are far too close to the machine level, and research in the field of optimizing compilers is required to create optimization

methods suitable to more "abstracted" languages, a situation eloquently expounded by John Backus[AK02, page 3]:

> "In fact, I believe that we are in a similar, but unrecognized situation today: in spite of all the fuss that has been made over myriad language details, current conventional languages are still very weak programming aids, and far more powerful languages would be in use today if anyone had found a way to make them run with adequate efficiency."

In fact, it is quite immaterial whether the optimizations thus discovered are eventually implemented in hardware or software. What matters is that they are indeed discovered, so that higher-level program languages can be realized.

Hence, it can be seen that *program transformations by optimizing compilers* are indeed important to increasing the performance of computer software, and that this area of research is a currently required and immediately useful area of research.

1.2 Program Optimization via Speculative Compilation: An Introduction

The optimization of a program by a compiler, *en route* to native code, is indeed important, as evidenced by the points made in the previous section.

Recently, a new approach called **speculation** has been employed to further improve the quality of program optimization by compilers. Speculation refers to the optimization of a program taking into consideration the biases that may manifest themselves during execution of that program. For example, a program with a hundred procedures may execute only one of those procedures frequently. Consequently, that frequently executed procedure is made the focus of the compiler's optimization effort, since, speaking *speculatively*, it can be expected to execute frequently in the future too.

Speculative optimizations usually optimize a program with respect to certain run-time metrics, such as execution frequency or code-size. For example, an algorithm named Speculative Partial Redundancy Elimination (SPRE) minimizes the number of redundant computations performed in a given program.

Unfortunately, speculative optimizations, such as SPRE, are very expensive to perform. The best implementations of SPRE require computation of the maximum-flow through a flow network *for each expression* used in the program. This is an onerously expensive optimization when one considers that there are many tens of thousands of expressions in a moderately-sized computer program *and* that each flow network is linear in the size of program's flow-chart.

Clearly, speculative algorithms such as SPRE must be made more frugal (in terms of their requirements), if they are to be employed as mainstream algorithms for program optimization by compilers.

There are two main problems with optimal speculative optimizations:

1. They work at a very fine resolutions. SPRE, for example, works with exact program frequencies: it needs to know exactly how many times each program branch and program statement is executed.

2. The resolution of program metrics required are difficult of obtain cheaply. In their paper, *Dynamic recompilation and profile-guided optimizations for a* .NET JIT *compiler*[VS03], the authors note that:

> "Our results also show that the overheads of collecting accurate profile information through instrumentation to an extent outweigh the benefits of profile-guided optimizations in our implementation, suggesting the need for implementing techniques that can reduce such overhead."

In this dissertation, we use the division of a program into frequently and infrequently executed ("hot" vs. "cold") regions —a concept which we term **isothermality**—to create algorithms for speculative program optimization. Within a region, each program part is considered to have equal execution frequency ("heat") motivating the prefix "iso-".

Therefore our algorithms do not dwell on the negligible differences that become apparent in high-resolution profile data, and, in doing so, obviate the need for highly accurate profile data to be collected in the first place.

1.3 My Claims

1.3.1 A Formal Statement

I make *four* claims which my dissertation validates:

Isothermality affords the development of algorithms for speculative program optimization which:

1. are less expensive to use than their optimal counterparts;
2. give performance improvements comparable to their optimal counterparts;
3. can easily be derived from non-speculative versions of the optimization;
4. are much easier to understand, and therefore, easier to implement correctly.

Claim 1 and *claim* 2 are *quantitative*—They will be proven by experiment.
Claim 3 and *claim* 4 are *qualitative*—They will be demonstrated by argument.

1.3.2 The Importance of My Claims

Some very important positive consequences arise from the validation of the above claims. It is these consequences that comprise a significant positive contribution to research in the field of compiler construction.

Claim 1 implies that:

> This dissertation has developed algorithms for optimization that can be used in systems where compilation resources are at a premium, such as:

1. embedded systems and controllers;
2. Just-In-Time (JIT) compilers, which compile programs *on demand*, and must therefore do so *quickly*.

The reader should note that compilation resources are at a premium even in the most powerful supercomputers—any hardware that is spending time compiling a program is expending time *not running* programs.

Claim 2 implies that:

> The algorithms for optimization developed in this dissertation can be used in place of their optimal counterparts with negligible sacrifice in the level of optimization. That is, we make big gains for small pains.

Claim 3 implies that:

> It is easy to deduce speculative versions of compiler optimizations from their non-speculative counterparts.

The consequence of *claim* 4 is that:

> The engineering of compilers can be simplified, creating a new generation of more reliable compilers which require less resources than their predecessors while producing code of comparable quality.

1.4 Agenda

This section provides a map of the dissertation to show the reader where and how it validates the claims previously made.

Chapter 1 contains a statement of the claims which will be proved by this dissertation.

Chapter 2 develops the foundations that this dissertation needs to present and discuss compiler algorithms for program optimization such as the Control Flow Graph (CFG), Intermediate Representation (IR), and Profile Driven Feedback (PDF), with examples.

Chapter 3 introduces Just-In-Time (JIT) compilation, its requirements, and the fundamental concept used by this thesis—*isothermality*, with an example.

Chapter 4 develops further a very important class of optimization algorithm, Partial Redundancy Elimination (PRE), to work speculatively using the concept of isothermality—Isothermal Speculative Partial Redundancy Elimination (ISPRE).

In this chapter, I validate *claim* 3—the ease of derivation of isothermal algorithms from their non-isothermal counterparts—with respect to ISPRE. I validate this claim by argument.

Chapter 5 develops further another very important class of optimization algorithm, Partial Dead Code Elimination (PDCE), to work speculatively using the concept of isothermality—Isothermal Speculative Partial Dead Code Elimination (ISPDCE).

> In this chapter, I validate *claim* 3—the ease of derivation of isothermal algorithms from their non-isothermal counterparts—with respect to ISPDCE.
>
> I validate this claim by argument.

Chapter 6 contains an exhaustive review of the speculative and non-speculative formulations of PRE and PDCE algorithms developed to date.

> In this chapter, I validate *claim* 4—the virtue of simplicity of algorithms developed using the concept of isothermality.
> I argue for this claim by providing detailed comparisons of ISPRE and ISPDCE to their non-isothermal counterparts.

Chapter 7 contains the results of benchmarking isothermal algorithms against their non-isothermal counterparts. This chapter has two sections:

1. The first part is devoted to Partial Redundancy Elimination (PRE). We show that the Isothermal Speculative Partial Redundancy Elimination (ISPRE) algorithm developed *gives performance improvements easily on par* with its optimal competitor SPRE, *at a fraction of the cost in compile time*.

   > In this section, I validate *claim* 1 and *claim* 2—
   > isothermal optimizations are less expensive than their optimal counterparts (1),
   > yet give performance improvements comparable to their optimal counterparts (2).
   >
   > Both claims are validated experimentally.

2. The second part is devoted to Dead Code Elimination (DCE). We show that the Isothermal Speculative Partial Dead Code Elimination (ISPDCE) algorithm developed gives *performance improvements that exceed* its main competitor PDCE, *at no extra cost in compile time—in fact it is cheaper*, while performing optimizations which are simply impossible to do with PDCE.

   > In this section, I validate *claim* 1 and *claim* 2 again—
   > by showing that a naïve non-isothermal algorithm can be empowered by isothermality to work speculatively, and to optimize code more aggressively, while remaining very frugal in terms of resource requirements.
   >
   > Both claims are validated experimentally.

Chapter 8 contains a restatement of the claims and results of the dissertation. Its also enumerates avenues of future work for further development of the concept of isothermality and its applications.

Chapter 2

Background

2.1 Compilation

Compilation is the process of converting a computer program specified in a high-level language into the language of the machine on which the program will be executed. The advent of compilation is undoubtedly one of the most spectacular advances in the history of software development. Prior to the invention of compilers, programmers would rely on a cadre of (human) operators to engage in the drudgery of translating a sequence of high-level instructions into the numeric vernacular of the target machine, a process which is fraught with error and extremely time-consuming.

The language FORTRAN and its inventor John Backus changed this forever, by demonstrating that high-level languages could be translated into efficient machine code *by a computer program*, namely, the compiler. Ever since FORTRAN, software development has been increasingly empowered by ever more abstracted programming languages since they allow programmers to think in the domain of the problem that they are trying to solve instead of in terms of the peculiarities of the computer hardware that will run their finished program.

However, all this abstraction comes at a price. The compiler cannot merely produce a correct translation of a program—it is expected to produce a very efficient translation, which exceeds the quality of the very best hand-translations. Furthermore, modern high-performance processors are designed with advanced architectural features, such as deep pipelines, which required complicated translation techniques to exploit properly.

Figure 2.1 shows the structure of a modern compiler. The processes of lexical analysis, parsing, and type checking form the phases of the compiler which are analytic; they break down the source program into its constituent parts. The phases of the compiler which are synthetic, in that they build the translated program, commence with the Intermediate Representation (IR) generation phase. An IR is a machine-independent mini-language which is much simpler in structure than the source language, yet expressive enough to faithfully represent the meaning of any program written in the source language. Furthermore, the intermediate representation has a form which is amenable to easy analysis, transformation, optimization, and subsequent machine code generation for the target platform. Finally, after machine code is generated, it is linked with libraries and other run-time amenities which are required to create an executable program.

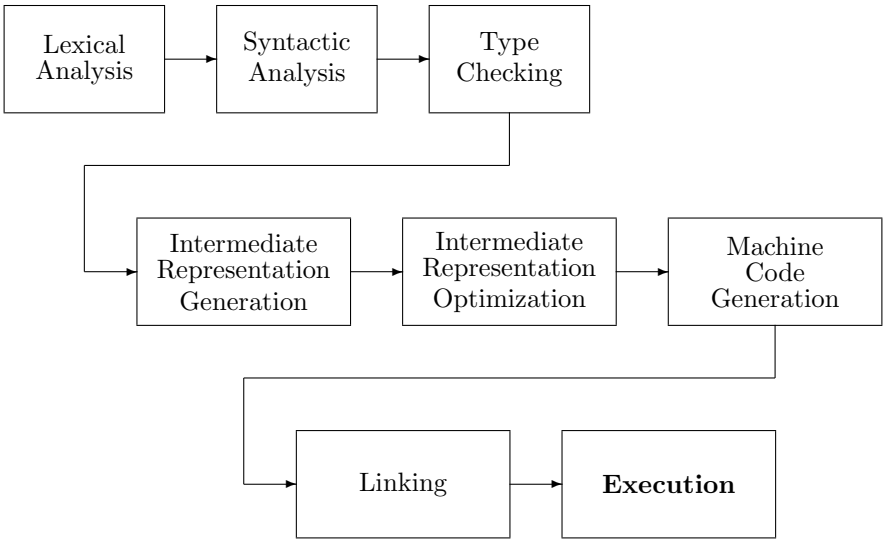


Figure 2.1: The compilation process

2.1.1 Phases in an Optimizing Compiler

It is important to note that Figure 2.1 is not drawn "to scale". Indeed, most modern compilers have IR optimizers which contain many individual parts or "phases". For example, the freely available GNU Compiler Collection (GCC)[FSF], a moderately optimizing compiler[Muc97], is comprised of a sequence of at least 100 such phases and the number is constantly growing.

Additionally, most modern compilers use multiple IRs in decreasing order of abstraction:

1. The Jikes Research Virtual Machine (JIKES RVM)[BCF+99] has 5 different IRs: a high-level IR and a low-level IR, both of which are provided in Static Single Assignment (SSA) and non-SSA form, in addition to a machine-dependent representation called MIR.

2. The Open Research Compiler (ORC)[ORC] has a single IR called Winning Hierarchical Intermediate Representation Language (WHIRL), which is available in 5 levels of abstraction, namely Very High Whirl, High Whirl, Mid Whirl, Low Whirl, and Very Low Whirl.

3. GCC has 2 IRs, GCC SIMPLE Intermediate Language (GIMPLE), an SSA based IR and Register Transfer Language (RTL) customized for the target machine.

We shall now describe, in greater detail, the sequence of optimizing phases typically found in compilers for imperative (ALGOL-like) languages, such as C, C++, Java, and FORTRAN. For brevity, we shall assume that the program module which is being compiled has already been parsed into an Abstract Syntax Tree (AST).

The notation $X \rightarrow Y$ below is interpreted as the typing of the optimization function. It should be read as "an optimization which takes a representation of the program in X to a representation of the program in Y".

1. AST→H(IR): Unlike non-optimizing compilers which would attempt to generate assembler, or even object code, from an AST representation of the original source program, an optimizing compiler will first convert the AST into an IR on which optimizing program transformations can be performed. The form of an IR is highly dependent on the transformations that will be performed on programs encoded in that representation. IRs are categorized by their height: higher-level representations are rich in constructs, while lower-level representations have fewer constructs and are, hence, more explicit. The first level of IR, being conceptually the highest, is often referred to as HIR.

2. HIR→HIR: Parallelizing and vectorizing program transformations are usually performed on the highest IR since it preserves array indexing and loop structure, the explicit knowledge of which is needed to perform dependence analysis and parallelizing program transformations.

3. HIR→MIR: After having performed a suite of parallelizing and vectorizing transformations upon programs represented in High-Level Intermediate Representation (HIR), a compiler will then translate the program into a lower representation, one in which looping structures have been converted into conditional jumps and in which array access expressions are represented by loads and stores with explicit element address calculations. We call such a program representation MIR.

4. MIR→MIR: Multiple program transformations suitable for this level of representation will then be performed. In fact, the majority of program transformations are performed at this point, including, most importantly, Partial Redundancy Elimination (PRE), Global Common Subexpression Elimination (GCSE), Loop Invariant Code Motion (LICM), and Partial Dead Code Elimination (PDCE), the algorithms which are at the heart of this dissertation.

5. MIR→LIR: The compiler will then convert programs represented in MIR to a form which is very close to the assembler for a particular machine—Low-Level Intermediate Representation (LIR). In this form, symbolic address locations are no longer used—registers and stack offsets are used to access local variables, and data segment addresses are assigned to global variables.

6. LIR→LIR: A suite of low-level optimizations is then performed which usually includes peephole optimizations and the replacement of certain instruction sequence with target platform idioms, when available.

7. LIR→machine code/assembler: Finally, the compiler will convert each LIR statement into its machine code equivalent, thereby completing the compilation process. Most compilers do not produce object code directly, but produce assembler code instead and delegate assembly to the system assembler. Optionally, a linker will combine multiple object code modules with static libraries (if required) to produce the final executable image.

Typically, most IRs are based on quadruples of the form

$$a \leftarrow b \oplus c$$

so-called since they consist of four parts: a result, a binary operator which produces the result, and two source operands. A stricter variant of this form is called Static Single Assignment (SSA) form, which requires that at most one assignment to a given name occur in a program. Each program in non-SSA form has an equivalent SSA form. SSA permits the simpler specification of many transformations due to its additional properties and is consequently popular in both compilers and compiler literature. Indeed, despite being merely a condition on an underlying Intermediate Representation, it is often thought of an IR its own right.

As we conclude this section, we ask the reader to remain mindful that despite all the computational power a compiler may bring to bear on the efficient translation of a source program (as expounded above), the compiler is itself a program much the same as any other—it must run quickly and efficiently. It can be seen, therefore, that the demands placed upon a compiler are quite severe: "Produce excellent object code *and* do so as quickly as possible."

2.2 Program Structure

Having described the structure of a compiler and the compilation process, we turn our attention to a brief exposition of the structure of the program being optimized.

In this dissertation, we restrict the scope of our algorithms to the structured and object-oriented families of imperative languages. We assume that a program is simply a collection of methods. Other higher-level structures, such as compilation units, packages, modules, and even classes, may be provided but our algorithms are indifferent to their presence.

For our purposes, there is no difference between procedures and methods. This is primarily because for languages such as C++ and Java, methods are often implemented as procedures which take an extra hidden parameter, which is typically a pointer to the object which the method is supposed to work upon. This arrangement is transparent to almost all optimizations that work upon the IR, and to our algorithms in particular. Consequently, even though our algorithms are perfectly applicable to object-oriented programming languages, we shall restrict ourselves to using the structured-programming term "procedure".

The focus of the optimization algorithms presented in this dissertation is on individual procedures. Therefore, the context in which a procedure occurs is not considered by our algorithms. That is, our algorithms do not take into consideration the static context of methods such as their containing classes or packages. Nor do our algorithms take into consideration dynamic context, such as frequent callers or frequent callees. In particular, decisions regarding the inlining of frequently called functions are left to the discretion of other parts of the compiler or host virtual machine.

Simply put, our algorithms take unoptimized procedures as inputs and produce optimized procedures as the outputs, considering only *one procedure at a time*. By definition, our algorithms are intra-procedural, as opposed to inter-procedural.

Thus, the algorithms presented in this thesis are amenable to use in compilers for languages such as C, C++, FORTRAN 77, FORTRAN 90, C#, and Java.

2.3 Procedure Representation

In this dissertation, we develop intra-procedural optimization algorithms. Consequently, the object of interest is the *procedure* and its constituent parts.

Each procedure is comprised of a finite set of basic blocks, N. A basic block is a linear sequence of IR instructions, whose forms will be introduced shortly. A basic block is executed only after some basic block transfers control ("jumps") to it. We refer to the flow of control from basic block m to basic block n as the *edge* (m, n). The edge set, E, with respect to a set of basic blocks is therefore a finite set

$$E \subseteq N \times N$$

We denote the set of basic blocks that can transfer control to block n as Pred(n) which is defined

$$\text{Pred}(n) \equiv \{u | (u, v) \in E\}$$

We denote the set of basic blocks that block n can transfer control to as Succ(n) which is defined

$$\text{Succ}(n) \equiv \{v | (u, v) \in E\}$$

Execution of a basic block always starts with the first instruction in the block, and continues through the complete linear list of instructions, in order. We specify, without loss of generality, that the last instruction in the basic block is a jump instruction (conditional or unconditional), which transfers control to some basic block.

There are two special basic blocks in each procedure. The entry basic block of a procedure, S, does not have control transferred to it from another basic block: the run-time environment transfers

control to it. The exit block of a procedure, T, does not transfer control to another basic block: after it finishes executing, the run-time environment returns control to the calling procedure or terminates the program, as required.

The CFG of a procedure is a quadruple (N, E, S, T) defined from the constituent parts just described.

2.4 Intermediate Representation

We now define the IR which our algorithms will work upon.

Our IR does *not* provide all the features found in a typical compiler. We have refrained from providing features that do not add any value to the presentation and discussion of our algorithms.

However, our IR is qualitatively complete. For example, the set of binary operators provided can be augmented to provide all the binary operators of a modern optimizing compiler's IR, such as bitwise operators. Similarly, our IR can be extended to provide operations on floating-point values. What matters is that our IR captures the essence of the IR of a modern optimizing compiler[ASU86].

In this dissertation, the view of computation will be *imperative*. Consequently, the IR instructions fall into two categories:

assignment instructions which change the value of a program datum, and

jump instructions which transfer flow of control, between basic blocks, based on the value of one or more data values.

We shall be concerned with three types of datum:

register, a particular register in the register file of a particular Instruction Set Architecture (ISA). The names that denote registers are ISA-specific. These are often referred to as "hard" registers in programming language literature.

memory location (mem), an area of Random Access Memory (RAM) whose size is equal to that of a register. Each MEM is denoted by a unique alpha-numeric name.

temporary, a name of the form T_n, where n is a natural number, which refers to an entity whose size and format is conducive to representing it with a register (via register allocation), but which may be ultimately represented as a MEM ("spilled"), if register allocation fails to assign it to a register. These are often referred to as "soft" or "symbolic registers" in programming language literature.

2.4.1 Assignment Instructions

Assignment instructions have the following form (in the following d refers to a datum, as specified above):

1. An integer assignment has the form

$$d = n$$

where n is an integer small enough to fit into the target datum d.

2. A copy instruction has the form

$$d_1 = d_2$$

3. A binary operation has the form

$$d_1 = d_2 \oplus d_3$$

where $\oplus$ is a simple operation drawn from the set $\{+, -, \times, \div\}$ with their usual interpretations. The values of d_2 and d_3 remain unchanged by the execution of this instruction.

4. A procedure invocation has the form

$$d_1 = \text{call } d_2$$

The effect of procedure invocation is to transfer control to the procedure whose address is specified specified by d_2, placing the return value of the procedure in d_1. The procedure invoked might implicitly change the value of one or more MEMs and one or more registers. However, the temporaries are guaranteed to be preserved across the execution of the instruction.

2.4.2 Jump Instructions

Jump instructions have the following form (in the following d refers to a datum, as specified above):

1. An unconditional jump has the form

$$\text{goto } B$$

where B is the name of a basic block.

2. A conditional jump has the form

$$\text{if } d_1 \oplus d_2 \text{ goto } B_1 \text{ else } B_2$$

where B_1 and B_2 are the names of basic blocks. The comparison operation $\oplus$ is drawn from the set $\{<, \leq, >, \geq, =, \neq\}$ with their usual interpretations. The values of d_1 and d_2 remain unchanged by the execution of this instruction.

Implicit Jump Instructions

The following two special abbreviations in the use of jump instructions should be noted:

1. When a basic block B_1 has exactly one immediate successor B_2, it is redundant to specify goto B_2 as the last instruction of B_1. In this case, we elide the instruction from the CFG, for brevity, *even though it exists* in the IR.

2. When a basic block B_1 has exactly two immediate successors B_2 and B_3, it is verbose to write "else B_3" at the end of

$$\text{if } d_1 \oplus d_2 \text{ goto } B_2 \text{ else } B_3$$

In this case, we elide the "else" clause from the CFG, for brevity, *even though it exists* in the IR.

2.5 An Example CFG

Figure 2.2 shows the translation of the procedure `summate`, which computes the function $\sum_{i=0}^{n} i$, into IR and a CFG.

Here, `n` (a parameter) as well as `i` and `total` (both local variables) are represented by the temporaries T_0, T_1, and T_2, respectively.

A fourth temporary T_3 has been introduced to hold the intermediate result `total+i`. This indicates that temporaries do not always correspond to program variables.

A fifth temporary T_4 has been introduced to hold the constant 1 since binary operations, in our IR, do not take constants as operands.

The do-loop has been decomposed into a body (starting at B_1) and a conditional-jump (at the end of B_1) which iterates the loop by jumping to the beginning of the body.

It should be noted that the clause "else B_2" is not written at the end of the instruction

$$\text{if } T_1 \leq T_0 \text{ goto } B_1$$

since it is implied, as previously discussed. Similarly, block B_0 does not end with the instruction goto B_1 since it is implied.

2.6 Profile Directed Optimization

Until recently, compiler optimizations were designed under the assumption that the program being optimized is the sole input to the compilation process. Under this assumption, the optimization must make only the most conservative assumptions about program properties, since it does not have statistics obtained from actual program executions to bear witness to actual run-time program properties being contrary to those conservative assumptions.

For example, in the absence of program statistics obtained from execution, an optimization is reduced to using compile-time heuristics[WL94] to distinguish coarsely between the frequency of different program paths. For example, an optimization may assume that back-edges of loops are more frequently executed, that comparisons between pointers often fail, or that tests for equality between variables and constants often fail.

However, the execution profiles of most programs would reveal to an optimization that there is typically a small subset of paths which are executed much more frequently than all other paths. Such execution profiles would allow the optimization to concentrate its efforts on precisely those paths of the program that dominate the running time of the program, since optimizing those paths will significantly improve the efficiency of the program.

Consequently, we revise the model of program compilation shown in Figure 2.1 into the model shown in Figure 2.3 consisting of the following steps:

1. The program is compiled naïvely. That is, optimizations may be performed, but only using conservative guesses as to run-time behaviour.

2. The program is run on "training" data. The input data chosen are intended to be representative of the data the program will be run on in the future. Statistics obtained from the execution of the program (the profile) are written to a database for later use by the compiler.

```
void
summate(int n)
{
    total = 0
    i = 0
    do
    {
        total = total + i
        i++
    } while (i<=n)
}
```

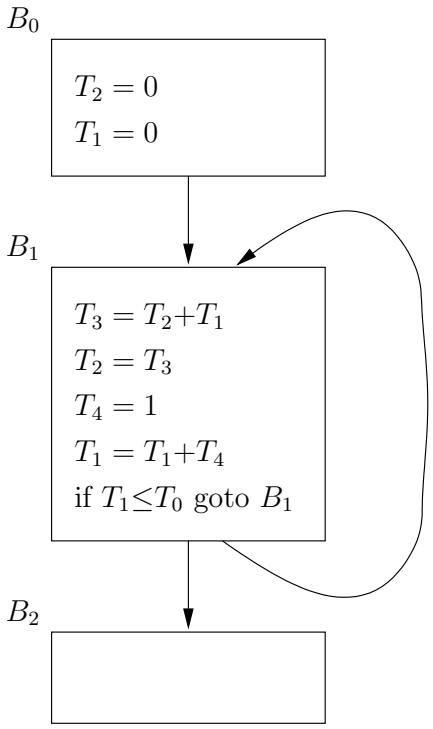
(a) The `summate` procedure written in a high-level language.

B_0:
  $T_2 = 0$
  $T_1 = 0$
B_1:
  $T_3 = T_2 + T_1$
  $T_2 = T_3$

  $T_4 = 1$
  $T_1 = T_1 + T_4$

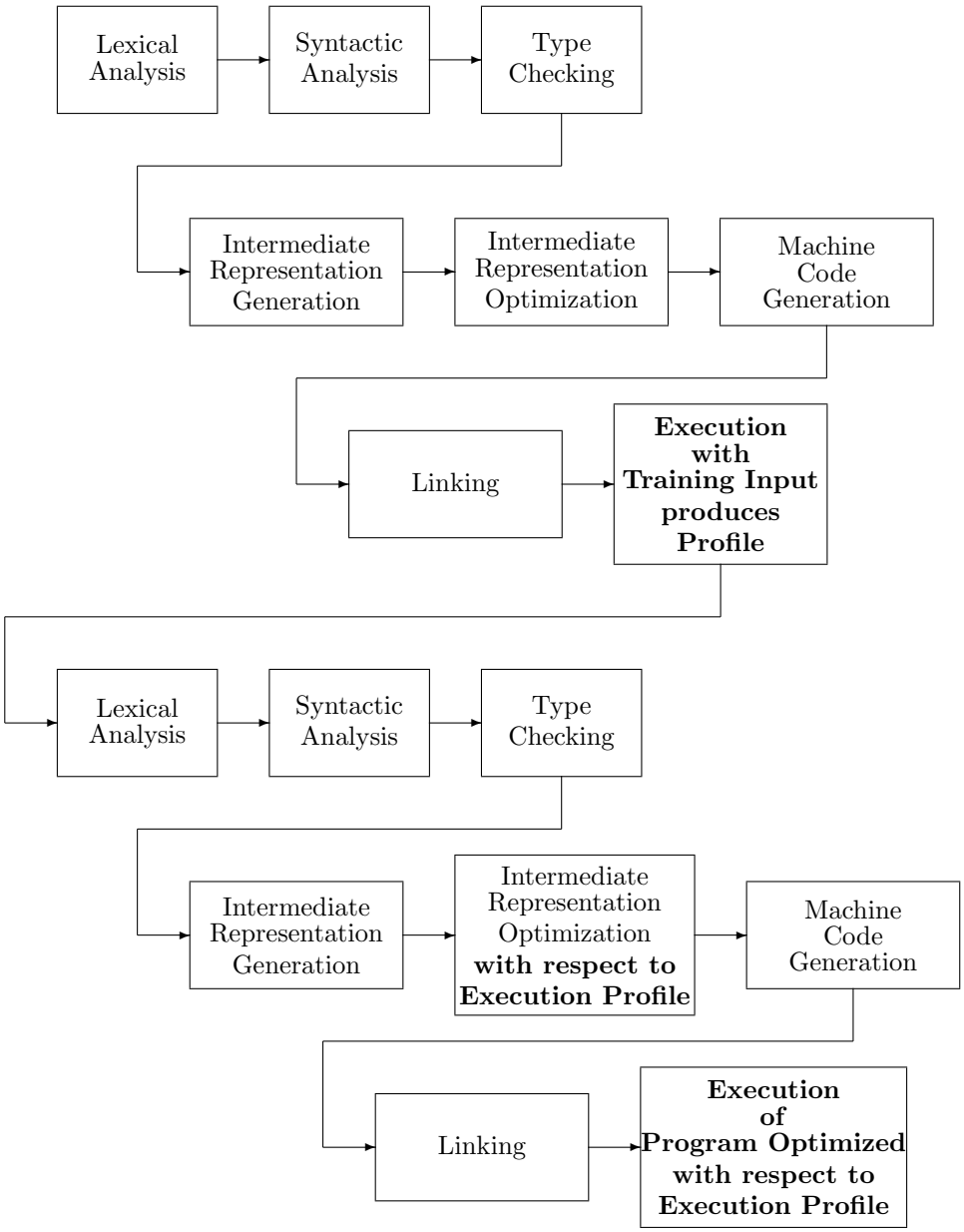
  if $T_1 \leq T_0$ goto B_1
B_2:

(b) Translation of the `summate` procedure into Intermediate Representation.



(c) Translation of the `summate` procedure into a Control Flow Graph.

Figure 2.2: Example: Translation of procedure `summate` into IR and CFG



The program is compiled, then run to produce an execution profile, and then recompiled taking the execution profile into consideration.

Figure 2.3: Profile Driven Feedback compilation

3. The program is recompiled "speculatively". That is, optimizations use the database of statistics (the profile) to make more intelligent decisions about how to optimize the program. This step is called "speculative" because the statistics are used only because the compiler *speculates* that they will be indicative of future program executions.

The concept of providing execution profiles to a subsequent compilation is called Profile Driven Feedback (PDF). This simplistic version of PDF just described is used to optimize programs written in languages that typically run without the aid of a virtual machine, such as C, C++, and FORTRAN.

2.6.1 Continuous Program Optimization

A major unsolved problem with PDF is the question of whether training data that is indicative of all future input data can be found. Indeed, if a program being compiled is trained with unrepresentative input data it may run extremely efficiently for that particular input data, but very inefficiently for the majority of its input data.

Consequently, the success of the afore-mentioned simplistic model is predicated on a very important assumption: the input data for the program execution that produces the profile (the "training data") must be representative of future input to the program. Otherwise, the profile will misguide the subsequent compilation phase.

This shortcoming is mitigated by a smarter incarnation of PDF called Continuous Program Optimization (CPO), as depicted in Figure 2.4. CPO is typically provided for languages, such as Java and C#, that are hosted by a Virtual Machine (VM). For such languages, the hosting VM monitors program execution and collects execution statistics continuously; if it observes a change in program statistics it can invoke the built-in Just-In-Time (JIT) compiler to recompile the program in the context of the new program profile. Consequently, the effects of badly chosen training inputs or volatile program properties are ameliorated.

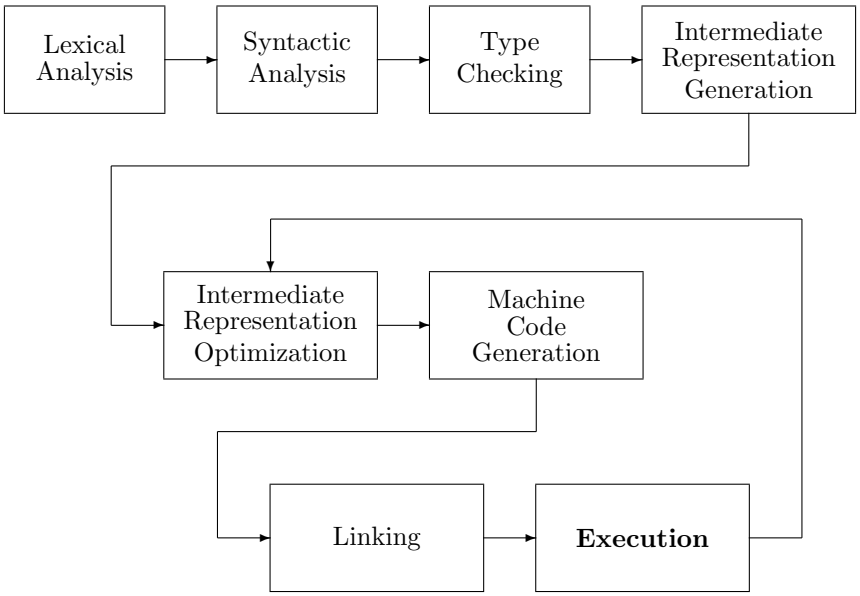
CPO even has the advantage that no "training" data is required in the first place—the real data the program is currently being executed on is its own training data. Therefore, the operator initiating the compilation does not have to consider whether or not the training data is representative.

2.6.2 Types of Program Profile

We conclude this subsection with a description of the types of program profile typically made available to a compiler from previous runs of a program. The first type of statistic is the **value profile**. This statistic associates values with variables in the source program (associating each value with a confidence level). Using this information, a compiler can specialize slices of the program for commonly-occurring run-time values.

PDF Example

Figure 2.5 shows an example of PDF. Figure 2.5 (a) shows a program that computes the quotient of two vectors of positive integers. It should be noted that most Reduced Instruction Set Computer (RISC) processors do not provide an instruction to perform division. It must be emulated by a software routine, which is slow. Even the Intel 386SX, a Complex Instruction Set Computer (CISC) processor, which provides a division instruction, takes up to 41 clock cycles to perform a 32-bit divide (IDIV), versus 3 clocks to perform a right-shift (SHR).



As the program executes, recompilation is automatically initiated by run-time environment.

Figure 2.4: Continuous Program Optimization

Suppose that running the program on training data shows that the most common value for the divisor (stored in T_1) is 2. Integer division-by-2 can be implemented much more efficiently via a "right-shift" operation (SHR). Hence, the compiler can optimize the program by inserting a check which determines if the divisor is 2, and if so, uses the optimized division.

Frequency Profiles

However, of instrumental importance to this dissertation is another type of profile called a **frequency profile**, which associates execution frequencies with sections of program code. There are three main varieties of frequency profile:

block profile This profile associates an execution frequency with each basic block in the source program.

edge profile [BL94] This profile associates an execution frequency with each edge in the source program. Edge profiles are more flexible since they can be used to compute the program's block profile, but not *vice versa*. However, they are more expensive to gather than block profiles.

path profile [BL96] This profile associates an execution frequency with each acyclic path through the source program. Path profiles are more flexible since they can be used to compute the program's edge profile, but not *vice versa*. However, they are more expensive to gather than edge profiles, requiring special algorithms both to determine where in the program to place counters and how to decode the results.

Methods of Frequency Profile Collection

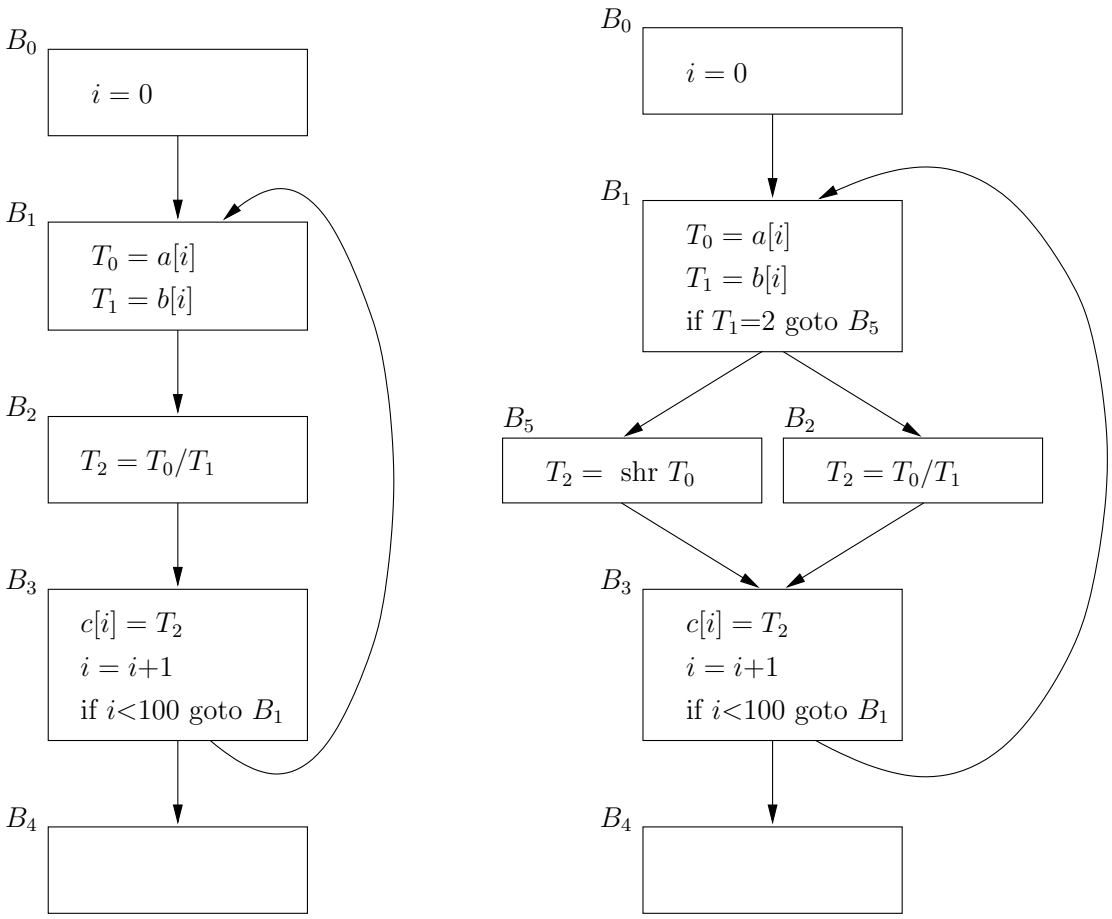
It is important, however, to distinguish between types of profile information and the methods by which those types of profile information are obtained. There are two main profiling methods:

1. **synchronous**, where operations are performed by programs or their host VMs at specific points in the program execution. The compiler must insert these operations into the compiled code as part of the compilation process, or the executable file must be rewritten[LB92] after compilation. For gathering frequency information, counters are typically used, and are of two types:

   (a) **software counters,** where the program increments the value of software counter variables as it executes. This can be performed in two ways:

      i. **occasionally,** where the program's compiler produces two versions of the compiled code—instrumented and uninstrumented[AR01]. Execution occurs primarily in the uninstrumented section, but occasionally briefly enters the instrumented section to perform some profiling by incrementing software counters, and then returns to execute in the uninstrumented section.
      This method is advantageous since operations performed in software to increment special counter variables can cause a significant degradation in performance[ABD+97].

      ii. **continuously,** where the compiler produces only instrumented code.



(a) A program to compute the element-wise quotient of two positively-valued vectors.

(b) The same program rewritten to perform division-by-2 via the quicker right-shift instruction.

*Optimization of the division of two positively-valued vectors: if the program's value profile indicates that division-by-2 is frequent, the program can be specialized so that division-by-2 is performed via a shift-right (*SHR*) instruction.*

Figure 2.5: Example: The use of Profile Driven Feedback

(b) **hardware counters,** where the processor executing the program's instruction stream gathers hardware performance metrics such as number of instructions executed[ABL97].

Since information is gathered at predetermined points, this method has the advantage of being deterministic, producing perfectly repeatable results over multiple executions.

2. **asynchronously**, where operations are performed by the host VM at points in program execution that are not predefined.

   The most common way to do this is via a hardware interrupt service routine[Wha00] initiated at small predictable intervals. When the interrupt service routine executes, it can, for example, examine the program's instruction-pointer register to determine where the program is executing[ABD+97].

   On account of flutter in the scheduling algorithms used by the operating system and clock jitter, this method is non-deterministic—each application of it can give slightly different results.

While block profiles can be collected with any of the above methods, edge and path profiles are usually collected by using synchronous methods.

It is very important to note that our algorithms do *not* pre-suppose the use of any particular method above for gathering profiles. All that is required is that the profile gathered enables the compiler to differentiate between frequently and infrequently executed program parts. An immediate consequence is that, armed with this information, the compiler can move computations from frequently executed program parts to infrequently executed program parts.

This dissertation will develop variants of the Partial Redundancy Elimination (PRE) and Partial Dead Code Elimination (PDCE) algorithms that will perform exactly this transformation in an effort to reduce the number of computations performed at run-time. Our algorithms will use edge profiles, since they are much easier to obtain than path profiles, yet sufficiently informative for our purposes.

Chapter 3

Introduction to Topic

3.1 Late Binding in Programming Languages

The earliest programming languages supported development of self-contained programs. That is, programs were simply the sum of their constituent parts and no others. Libraries, which a program was required to link with, were always immediately available, and the end result of the compilation was invariably a complete program that was ready to run.

The advantages of such immediate assembly of the constituent parts of a program implied that the program was always available in its entirety for analysis. Indeed, very ambitious program analysers have been designed to exploit this property. The IBM Toronto Portable Optimizer (TPO) and the IBM Toronto Back-End with Yorktown (TOBEY) are but two examples of very ambitious optimizers which perform analysis of the entirety of the program they are optimizing, spanning its classes, modules, packages, and compilation units. In fact, this aggressive method of analysis and optimization is called Whole Program Optimization (WPO)[TBR+06].

However, the increasing scope of applications of computer software and the concomitant need for generality, customizability, and flexibility in computer software has compelled many software systems to be designed more as extensible frameworks than as complete programs. Currently, many programs often allow their functionality to be extended via "plug-ins". These plug-ins often take the form of dynamically-linked libraries that are not necessarily present, and, more profoundly, may not even exist at the time of development and compilation of the program. When optimizing such extensible programs, optimizers such as TOBEY and TPO are rendered powerless: the plug-ins are essentially black-boxes that are impenetrable by their analyses.

The current situation, however, is even more grave. Languages such as Java and C# are designed explicitly to support the late addition of componentry to a running program: Java's class loaders have the ability to fabricate new classes on-the-fly and deliver them to the host VM for integration into the executing program; Java's Remote Method Invocation (RMI) can even invoke methods that reside on virtual machines other than the host virtual machine; The Java Native Interface (JNI) allows a Java program to call procedures written in C and C++. These procedures typically reside in dynamically-linked libraries and preclude effective program analysis in much the same way as the plug-ins afore-mentioned. It can easily be seen that the command-line Java compiler `javac` is severely limited in its ability to optimize all but the simplest Java programs. Indeed, `javac` at

present is little more than a glorified parser and type-checker that produces byte-code files for the Java virtual machine to execute. C# is complicated further. Through its Common Intermediate Language (CIL), (a rather extensive language designed to support the semantics of most major programming languages currently used), programs in C# are not only able to call methods that they have never seen before, but are able to call methods written in entirely different languages with radically different semantics. Lest the reader think the complications just mentioned regarding Java and C# are fringe, uncommonly occurring "boundary cases", it should be pointed out that Java and C# have important frameworks, the Java 2 Enterprise Edition (J2EE) and the Microsoft .NET Framework respectively, whose functionality is based uncompromisingly on the late binding methods just discussed.

3.2 Dynamic Compilation

The model of compilation introduced in Chapter 2 compiles a program into a single self-contained unit, producing a complete executable that requires no further compilation. This executable can then be loaded and dispatched. However, this is by no means the only way to execute programs written in a high-level language. Indeed, the easiest way to execute a high-level program is to interpret it: a program called an interpreter simply carries out the (high-level) instructions in the source program, typically without producing even a single byte of machine code.

It so happens that the diverse techniques of interpretation and compilation form not just two methods for program execution, but rather the two ends of a spectrum of techniques for program execution. Indeed, it is possible to amalgamate compilation and interpretation—those hybrid techniques which lie between these two end points are collectively termed dynamic compilation.

Dynamic compilation refers to the process of compilation which occurs *after* the program has started executing. Typically, as program execution occurs, many (hitherto uncompiled) procedures are invoked which triggers their dynamic compilation into machine code. The Microsoft .NET Framework (.NET) VM, which does not include an interpreter, uses this approach. For the sake of simplicity, we shall restrict our discussion to programming languages that run under a host VM that provides run-time support such as dynamic compilation.

The explanation provided in the preceding paragraph is overly simplistic: the decision to invoke the dynamic compiler must be made judiciously. If it is invoked too eagerly, uncomfortable pauses in program execution will result, which is disconcerting for interactive applications. Additionally, over-eager invocation will result in the compilation of procedures which are invoked only once (though obviating the need for an interpreter). On the other hand, under-eager invocation will result in the primary mode of program execution being interpretive, which is much slower than natively executing compiled code. It can be seen, therefore, that a very careful decision is required.

However, it is of crucial importance to note that even though a careful decision *must* be made by the virtual machine (regarding dynamic compilation), it is a decision that *cannot* generally be made by a stand-alone compiler. A stand-alone compiler does not have the knowledge of the dynamic properties of a program, apart from execution profiles and educated but conservative guesses. Indeed, the important dynamic properties of programs are often provably undecidable problems at compile time. This reveals the strength of dynamic compilation: *since it is performed at run-time, it is performed with the knowledge of the program's dynamic properties.*

An example will suffice to make the above claim concrete. Consider a stand-alone compiler compiling a large C program. Even with Whole Program Optimization (WPO) enabled, the compiler can only make conservative guesses about the structure of the call tree of the program. In fact, in the presence of pointers-to-functions (in languages such as C) even very aggressive analyses will not increase the precision of the answer since calls via those pointers can potentially invoke many functions. Consequently, the ability to amalgamate frequently called functions into their callers (i.e., to inline) at compile-time is severely restricted. However, in a dynamically compiled system, the virtual machine can inspect the execution of the program to dynamically construct the call tree with as much precision as is desired. This can be used to guide inlining requests made to the dynamic compiler with much greater confidence in positive returns than a static analysis.

The scope of dynamic compilation is by no means restricted to making invocation and inlining decisions. Many more questions regarding dynamic program properties can be answered at run-time, and exploited by the run-time compiler to positive effect.

3.3 Motivating Dynamic Compilation

In Section 3.1 it was shown that delaying the linkage of various program components can impede the analytic capabilities of an optimizing compiler. Section 3.2 introduced dynamic compilation, which solves this problem—compilation is simply deferred to the time when all the required program components are available. A second argument for dynamic compilation was also proffered in Section 3.2, namely the undecidability of the compile-time questions regarding dynamic program properties. In this section, we elaborate further on this important point.

Dynamic program properties change over the course of the execution of the program. Consider, for example, the "localities" of a program—the sections of code which are executed frequently. While it is true that programs spend a large portion of their time in predictable localities of the program, such as loops, it is also true that localities change through the execution of a program. A program may even execute in one locality for a given input, and in another locality for another input. This variance hampers the effectiveness of even profile-driven optimizations since input-dependent localities make it difficult, if not impossible, to find a representative input on which to "train" the optimizer. Yet, it is important to be able to find localities in order to optimize them since the code comprising them dominates the execution time of a program.

Languages that are hosted on virtual machines that provide dynamic compilation are much less vulnerable to such problems since the virtual machine can monitor the dynamic properties of programs. When the dynamic properties of a program change, the virtual machine can discard one or more previously compiled methods, and request the dynamic compilation subsystem to recompile those methods in the context of the new program properties. A nascent example of such a system in found in the IBM Testarossa JIT Compiler (TR-JIT) for the IBM J9 Virtual Machine for Java (J9-JVM). The TR-JIT's dynamic compilation subsystem can be configured to compile a method at different levels of optimization. Initially, an occasionally-invoked method is compiled at the "warm" level of optimization. However, as the frequency of invocation of the method increases, the method is subjected to new increasingly rigorous optimization efforts, namely "hot", "very hot", and finally "scorching".

The fundamental observation to be made here is that dynamic program properties occur in

any language, not just those with late binding properties. Localities of execution can change in C code just as severely as they change in Java code, thereby compelling dynamic compilation even for typically closed-world languages such as C and C++. Proof of this statement is found in the program *Dynamo: A Transparent Dynamic Optimization System*[BDB00] developed at Hewlett-Packard Research. Dynamo is an interpreter that dynamically instruments and optimizes executables compiled for the Precision Architecture RISC (PA-RISC) family of processors. On many benchmarks they obtain speedups that are in excess of the those provided by the most aggressive level of static compilation.

We conclude this section by stressing that dynamic compilation is not merely a nicety which enables late-bound languages to work, but a *requirement* for high-performance execution of any program regardless of the language in which it is written.

3.4 The Requirements of Dynamic Compilation

In the previous section, we argued that dynamic compilation is a quintessential ingredient for a high-performance implementation of any programming language.

JIT compilers, however, are restricted to optimization algorithms that are relatively inexpensive to perform. This is because the time spent by a JIT compiler in optimizing a program is time during which the program itself does not run. A traditional compiler is not subject to such a tradeoff, since the time spent in compilation occurs during program development, and therefore impinges neither on job throughput nor on user-interactivity. Additionally, once a part of the program has been optimized, there is no guarantee that it will be executed frequently in the future, if at all.

Therefore, as previously stated, JIT compilers perform optimizations with the utmost discretion: Many VMs extensively instrument programs to monitor the relative execution frequencies of each part of the program. Only the most frequently executed parts are intensively optimized. However, despite making the intensity of optimization proportional to the frequency of the routine being optimized, the following remains true of JIT compilers:

1. accurate instrumentation is expensive and consumes cycles that should belong to the program;
2. even a perfect history of past execution cannot predict future execution with certainty;
3. time spent optimizing a program is time spent not running the program.

Consequently, an optimization well suited to JIT compilers is one which:

1. relies very little on program instrumentation;
2. optimizes the program using a little instrumentation data almost as well as if it had perfectly accurate instrumentation data;
3. executes quickly.

In this dissertation, we invent a new methodology for program optimization which takes into consideration the three points afore-mentioned. We demonstrate our methodology and its viability by applying it to two major algorithms used in optimizing compilers, namely:

1. Speculative Partial Redundancy Elimination (SPRE);

2. Speculative Partial Dead Code Elimination (SPDCE);

Our methodology produces algorithms that satisfy the three previously-mentioned criteria since they:

1. require the minimum amount of information per program block—the distinction between frequent and infrequent use ("hot vs cold");
2. are applicable in cases where established traditional algorithms fail;
3. are much easier to understand and implement than the established algorithms, and run more efficiently.

3.5 Isothermality: Reducing Instrumentation and Algorithm Complexity

In this thesis, we present a methodology for developing optimization algorithms that reduce the need for VMs to provide precise frequency profiles of programs, and that reduce the time complexity of compiler optimizations that use those execution profiles. While doing so, we maintain the effectiveness of the optimizations relative to the best optimization algorithms available for that type of optimization. The fundamental concept that enables this dramatic improvement is called *isothermality*.

En route to the definition, we shall consider the following. An executing program, as previously stated, has multiple localities of use, some of which are far more frequently favoured ("hot") than others ("cold"). The oft-cited 80/20 rubric, which states that 20% of a program is executed 80% of the time, alludes to this property. A VM that monitors execution of the program will attempt to provide accurate execution frequencies for each basic block in each procedure of the program. These frequency counts, which can be *perfectly* accurate, are gathered with the intent that they will be used later by the dynamic compiler. If perfectly accurate, they will distinguish, for example, between a basic block executed 10^6 times versus another executed $10^6 + 1$ times. However, it is reasonable to ask whether optimization algorithms need such precision.

When faced with large amounts of sample data representing a population, such as frequencies of basic blocks, the proper statistical operation is to summarize it. An elementary way of doing so is with a histogram: each interval in the histogram has a certain width (in terms of the numerical variable summarized by the histogram) that accommodates all values that lie between its lower and upper bounds. How many intervals should a histogram have? The smallest answer, which still maintains the summarizing property of the histogram, is 2. When this procedure is applied to program execution frequencies it is easy to see that the lower interval captures the parts of the program that execute relatively infrequently (the "cold" parts); the upper interval accommodates the parts of the program that execute relatively frequently (the "hot" parts). That is, each interval clusters together program parts of approximately equal execution frequency (or "heat"). Therefore, we call this method of classification *isothermality*[1].

In this dissertation, we show that optimization algorithms developed to use an isothermal classification of execution frequencies optimize programs almost as well as algorithms that use perfectly

[1] Gk. iso ($\iota\sigma o$): equal, thermos ($\theta\epsilon\rho\mu o\sigma$): heat.

accurate execution frequency information. Additionally, "isothermal" algorithms execute more efficiently and are much easier to understand and implement than their counterparts which require perfectly accurate frequency information.

3.6 Isothermality: An Example

Figure 3.1 shows C code that computes primes number in range $[2, limit]$ by a very simple implementation of the sieve of Eratosthenes. Figure 3.2 shows a CFG that corresponds to the code in Figure 3.1. The basic blocks have been annotated with execution counts in parentheses, which were produced by running the program with $limit = 1000$. The blocks and the edges between them form the members of the population under consideration. The frequencies of population members occupy the range $[1, 2126]$.

3.6.1 Capturing a Frequently Used Loop

We can divide the range $[1, 2126]$ into two intervals by choosing a threshold value, Θ, to be 80% of the maximum frequency. Hence, we have

$$\Theta = 1700$$

which divides the range $[1, 2126]$ into two intervals:

$$\begin{aligned} \text{Cold} &\equiv [0, 1700) \\ \text{Hot} &\equiv [1700, 2126] \end{aligned}$$

It can be seen that only blocks

1. B_8, with frequency 2126
2. B_9, with frequency 1958

and edges

1. $B_8 \rightarrow B_9$, with frequency 1958
2. $B_9 \rightarrow B_8$, with frequency 1958

comprise the Hot region. All other blocks and edges in the graph are members of the Cold region. Note that this classification captures the loop in the sieve that removes all multiples of a given prime from the final set of primes. This loop is iterated frequently since prime numbers are rare relative to composite numbers, especially as *limit* increases.

3.6.2 Capturing a Part of a Loop

We can also divide the range $[1, 2126]$ into two different intervals by choosing the threshold value, Θ, to be 35% of the maximum frequency. Hence, we have

$$\Theta = 745$$

which divides the range $[1, 2126]$ into two intervals:

$$\begin{aligned} \text{Cold} &\equiv [0, 745) \\ \text{Warm} &\equiv [745, 2126] \end{aligned}$$

The parts of the CFG that comprise the Warm region belong to both shaded regions of Figure 3.2. All other blocks and edges in the graph are members of the Cold region.

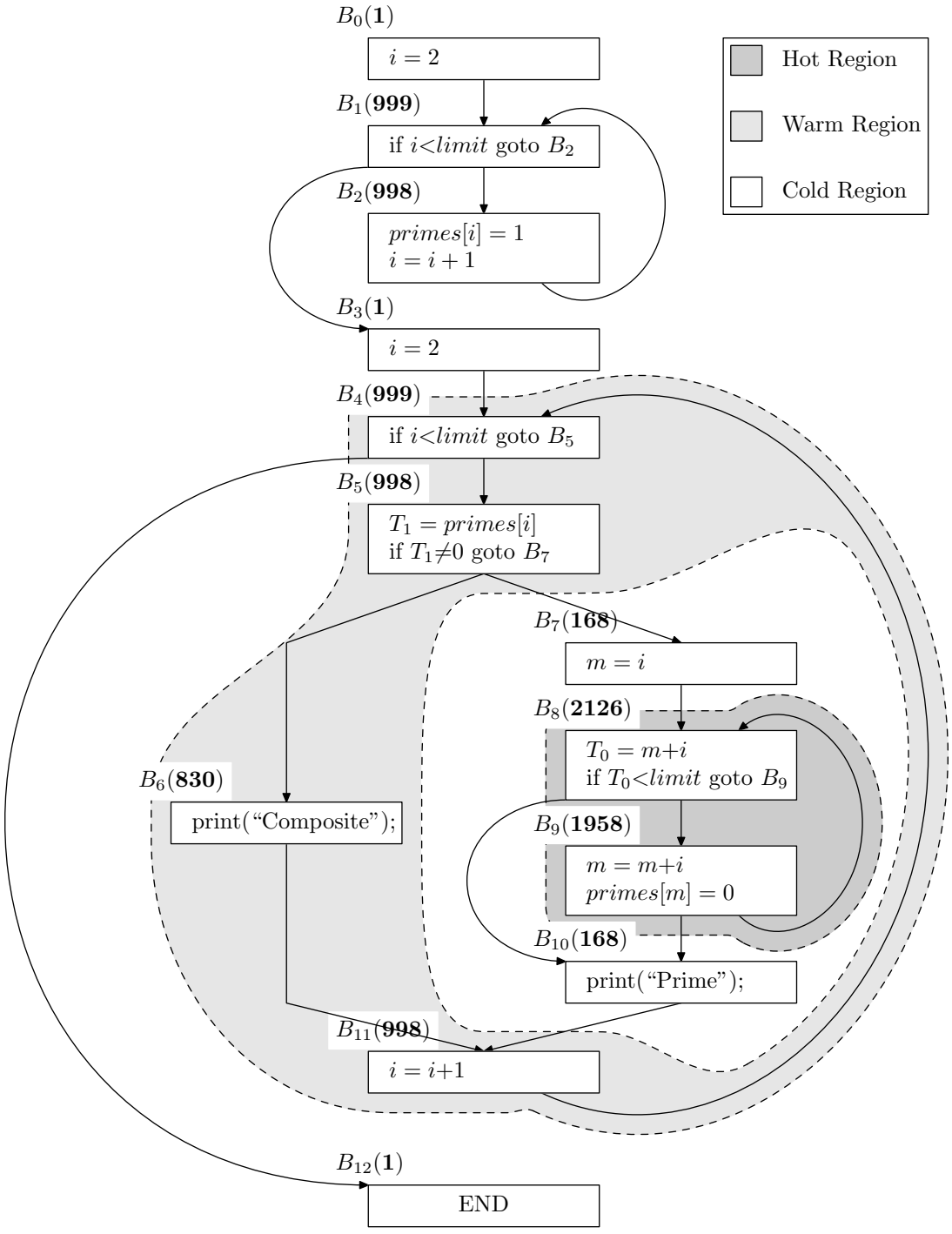
It is very important to note that the Warm region does *not* capture the outer loop completely: blocks B_7 and B_{10} are members of the Cold region.

Consequently, it can be seen that isothermal classification does *not* discover loops: It discovers sections of loops which are frequently executed, and can distinguish between loops whose bodies have a frequency bias (compare blocks B_6 and B_7 which have frequencies of 830 and 168, respectively), thereby potentially enabling code motion from a commonly used side of the loop.

```
 1  #include <stdio.h>
 2  #include <math.h>
 3
 4  #define limit 1000
 5
 6  int primes[limit];
 7
 8  int
 9  main()
10  {
11          int i;
12          for (i = 2; i < limit; ++i) {
13                  primes[i] = 1;
14          }
15
16          for (i = 2; i < limit; ++i) {
17                  if (primes[i]) {
18                          int m = i;
19                          while ((m+i) < limit) {
20                                  m = m+i;
21                                  primes[m] = 0;
22                          }
23                          printf (``Prime %d\n'', i);
24                  } else {
25                          printf (``Composite %d\n'', i);
26                  }
27          }
28
29          return 0;
30  }
```

This sieve computes primes in the range $[0, 1000]$. *The Control Flow Graph (*CFG*) corresponding to this code is shown in Figure 3.2.*

Figure 3.1: C source code for the Sieve of Eratosthenes.



This CFG *represents the Sieve of Eratosthenes, which computes primes in the range* $[0, limit]$. *Here, limit is set to* 1000.

Figure 3.2: Example: Isothermal classification of Control Flow Graph (CFG)

3.7 Applications of Isothermality

A very important task in an optimizing compiler is to remove code that has no effect on the correctness of the final output of the program. Such code often exists in programs, sometimes due to programmer carelessness, but more often as a side-effect of the process of compilation itself:

1. High-level constructs are often translated into IR by using "boilerplate" techniques. For example, to compile code that accesses an element in an array, IR must be produced that calculate the address of the indexed value by multiplying the index by the stride of the array. Two sequential accesses to the same index of different arrays therefore involve a single redundant computation.

   Consider the computation of the dot product shown in Figure 3.3. Assuming that the `int` datatype has a size of 4 bytes, *each* evaluation of the expressions `a[i]` and `b[i]` will require a computation of $4 \times i$. However, the value of `i` does not change between the evaluations of `a[i]` and `b[i]`. This redundancy will reduce program performance especially if `dot_product` is used in a scientific application, where it will be frequently called with long vectors as input.

   It is important to note here that although no redundancy exists in the high-level (C) code, the redundancy appears *after* the code has been translated.

2. Optimizing compiler phases produce redundant expressions as a side-effect of optimization. For example, an optimizing compiler might optimize the code:

```
1  value1 = a*a + 2*a*b + b*b;
2  value2 = a+b;
```

   by applying the algebraic identity $(a+b)^2 \equiv a^2+2ab+b^2$ to reduce the number of multiplications from 4 to 1. This produces the improved code:

```
1  value1 = (a+b)*(a+b);
2  value2 = a+b;
```

   which now has two redundancies. The first redundancy is produced in the computation of `value1`. The second redundancy is produced *across* the computations of `value1` and `value2`.

3. Programmers often write code to compute values that are used on only a subset of paths through the program but not on others. Typically, this is a convenience to the programmer; it allows the programmer to cluster conceptually related computations.

   Consider the computation of quadratic roots shown in Figure 3.4: If the discriminant is negative, the variables `denominator` and `numerator` have been computed needlessly. Note also, that the expression `sqrt(disriminant)` is computed redundantly when the discriminant is positive.

The type of inefficiency discussed in points 1 and 2 are called **redundancies** since they involve repeated computation of the same value. The type of inefficiency discussed in point 3 is called **dead code** since it involves the computation of a value that is not subsequently used.

```
1  int
2  dot_product(int* a, int* b, int length)
3  {
4      int i;
5      int dp = 0;
6      for (i = 0; i < length; ++i) {
7          dp = dp + a[i] * b[i];
8      }
9      return dp;
10 }
```

The expressions a[i] and b[i] both contain the computation $4 \times i$, a redundancy.

Figure 3.3: C code to compute the dot-product of two vectors

```
1  int
2  quadratic_roots(double a, double b, double c)
3  {
4      double discriminant = b*b - 4*a*c;
5      double denominator = 2*a;
6      double numerator = -b;
7
8      if (discriminant > 0) {
9          root1 = (numerator + sqrt(discriminant)) / denominator;
10         root2 = (numerator - sqrt(discriminant)) / denominator;
11         return 2;
12     } else if (discriminant == 0) {
13         root1 = (numerator + sqrt(discriminant)) / denominator;
14         root2 = root1;
15         return 1;
16     } else {
17         print(''Error: equation has no real roots.'');
18         return 0;
19     }
20 }
```

`denominator` is computed, but not always used. Hence, the computation and assignment of `2*a` in line 5 is partially dead code.

Figure 3.4: C code to compute the roots of a quadratic equation

An algorithm that removes redundancies is called a Partial Redundancy Elimination (PRE) algorithm. An algorithm that removes dead code is called a Partial Dead Code Elimination (PDCE) algorithm.

We can now briefly introduce the two completely new algorithms presented in this dissertation, that are built upon the concept of isothermality. We devote one chapter each (Chapter 4 and Chapter 5) to the exposition of each algorithm. Thereafter, a third chapter (Chapter 7) provides benchmarking data that demonstrates the effectiveness of these two algorithms.

3.7.1 Isothermal Partial Redundancy Elimination

Isothermal Speculative Partial Redundancy Elimination (ISPRE) is an algorithm to optimize programs by removing redundancies from them. The ISPRE algorithm is speculative: it uses the frequency profile of a program when transforming the program. However, it does so using the concept of isothermality to perform the speculative transformation in a manner that is cheaper both in space and time complexity than optimal SPRE but comparable in effectiveness. Chapter 4 is devoted to the exposition of this algorithm. The benchmarking of this algorithm and a discussion of the benchmarking results are contained in Chapter 7.

3.7.2 Isothermal Partial Dead Code Elimination

Isothermal Speculative Partial Dead Code Elimination (ISPDCE) is the counterpart of ISPRE to perform dead code elimination. Like ISPRE, it uses the execution frequency profile of a program when transforming the program. However, it does so using the concept of isothermality to perform the speculative transformation in a manner that is cheaper both in space and time complexity than the best PDCE algorithms but superior in optimization capability. Chapter 5 is devoted to the exposition of this algorithm. The benchmarking of this algorithm and a discussion of the benchmarking results are contained in Chapter 7.

Chapter 4

PRE - Partial Redundancy Elimination

The contents of this chapter are adapted from portions of the author's paper *Fast Profile-Based Partial Redundancy Elimination*[HPS06] presented at the 7th Joint Modular Programming Languages (JMLC) Conference 2006, Oxford, UK.

4.1 Introduction

PRE is a standard program optimization that removes redundant computations via code motion. It subsumes and generalizes the optimizations of Global Common Subexpression Elimination (GCSE) and Loop Invariant Code Motion (LICM). Recent work has generalized PRE to become Speculative Partial Redundancy Elimination (SPRE), which uses estimates of execution frequencies to find the optimal places in a program to perform computations. However, the analysis performed by the compiler is computationally intensive and hence impractical for JIT compilers.

This chapter develops a completely new and novel approach that abandons a guarantee of optimality in favour of simplicity and speed of analysis. This new approach, called Isothermal Speculative Partial Redundancy Elimination (ISPRE), achieves performance improvements that are close to optimal in practice, yet its running time is at least as good as current compiler optimizations for code motion. It is a technique suitable for use in JIT compilers. We present an *intra-procedural* algorithm. That is, each procedure of a program will be transformed by ISPRE independently of the other procedures.

We commence this chapter with a tutorial introduction to this large family of optimization algorithms, to the extent required to understand our contributions. Then, we justify the need for our algorithm, develop and state our algorithm precisely, and prove its correctness. We also provide an example of algorithm in action.

We defer our literature survey of the PRE family of algorithms to Chapter 6. Benchmarks results for the application of ISPRE to the SPEC CPU 2000 benchmark suite can be found in Chapter 7.

4.2 A PRE Tutorial

A path through a real-world program often contains one or more repeated evaluations of a computation, such that the evaluations subsequent to the first compute the same value as the first. The subsequent evaluations are traditionally said to be **redundant**.

Figure 4.1a shows a redundant expression on a program path. Naturally, in order to prevent a time-consuming recomputation of the same result in B_2, the original result should be cached in B_1 and reused later at the point of redundancy. That is, we want to effect the transformation shown in Figure 4.1b. This transformation, called Common Subexpression Elimination (CSE) has existed as a standard compiler optimization since the early FORTRAN compilers [ASU86, Bre69].

However, in order to replace the computation of an expression with a temporary (such as T_0), one must ensure that on *every* path to the replacement point from the start of the program that:

1. the expression is computed, at least once;
2. its computed value is cached in a temporary variable; and
3. the value of the expression's operands do not change after caching.

In the parlance of modern dataflow analysis, the expression is required to be **available** at a proposed substitution point, or, equivalently, redundant on all paths leading to the proposed substitution point.

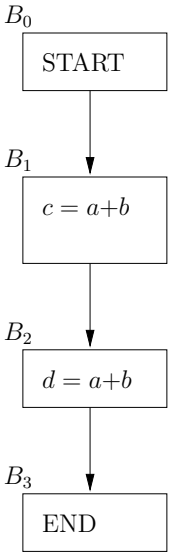
Traditionally, a well-known dataflow analysis called "available expressions" was used to deduce the availability of an expression at a proposed substitution point. If available, the instructions computing the expression under consideration are modified to cache the computed value in a temporary variable. The computation of the expression at the proposed substitution point is then replaced by the temporary variable. This sequence of operations is called Global Common Subexpression Elimination (GCSE) and was first formulated in [Coc70].

In the general case, however, an expression is very rarely available (alternatively completely redundant) at a proposed substitution point, but is often available on a subset of paths leading to a proposed substitution point. That is, an expression is often **partially redundant**, a situation depicted in Figure 4.2a, where the computation $a = b+c$ cannot be elided (or, at least, does not seem to be elidable) from B_4 since it is not available on the program path $B_0 \rightarrow B_1 \rightarrow B_3 \rightarrow B_4$.

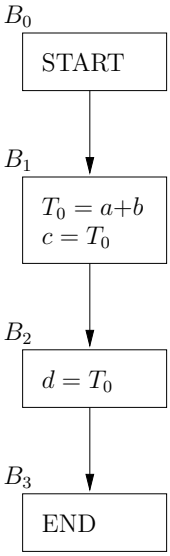
The more aggressive optimization Partial Redundancy Elimination (PRE) remedies the unavailability of a+b on the path $B_0 \rightarrow B_1 \rightarrow B_3 \rightarrow B_4$ by hoisting it against the flow of control on that path to block B_3. This renders a+b completely-redundant at the point where the two paths join, namely block B_4. The available expressions analysis can be used thereupon, to eliminate the complete redundancy. The resultant program, shown in Figure 4.2b, continues to have the same number of computations on the right path, but has one fewer computation on the left path. Hence, the number of dynamic evaluations of the expression a+b has decreased and the program has been optimized.

The generalization from GCSE to PRE was first described by Morel and Renvoise in [MR79]. They later extended their analyses to the inter-procedural case in [MJ81]. Since its invention, PRE has been used consistently in optimizing compilers because it subsumes two different program optimizations:

1. Global Common Subexpression Elimination (GCSE), as discussed above;
2. Loop Invariant Code Motion (LICM), which hoists loop-invariant computations outside loops.

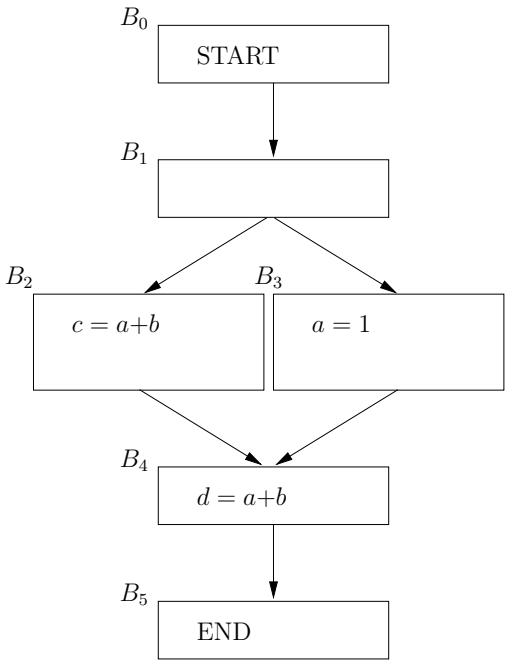


(a) b+c is computed in both B_1 and B_2 with no intervening assignment to either b or c.

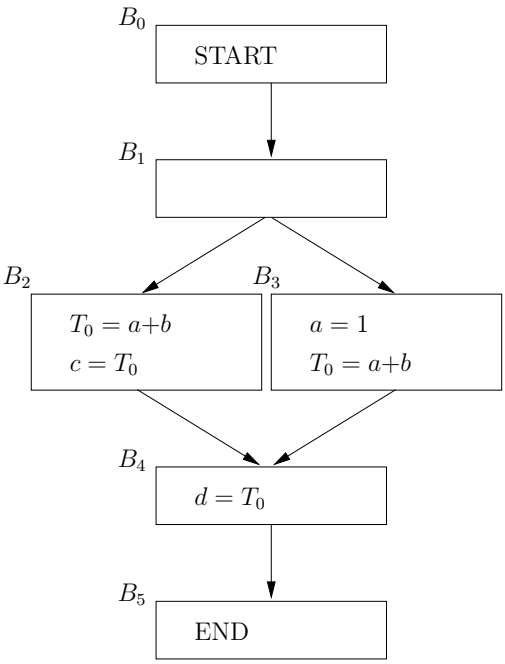


(b) Hence, the value of the first computation of b+c can be cached in a temporary T_0 and used at the point of the second computation of b+c.

Figure 4.1: Elimination of a completely redundant computation



(a) a+b cannot be deleted from B_4 since it is not computed anywhere on the path $B_0 \rightarrow B_1 \rightarrow B_3 \rightarrow B_4$.



(b) By hoisting the computation a+b from B_4 to B_3, a+b becomes pre-computed on all paths leading to B_4, and can hence be deleted from B_4.

Figure 4.2: Elimination of a partially redundant computation

Note: That PRE hoists loop-invariant computations is an easy consequence of the fact that PRE's *modus operandi* is to move computations as much against the flow of control as possible. Therefore, a loop-invariant computation, having no assignments to its operands within the loop, will be movable backwards on all paths from the loop entry node to it, and will eventually be moved out of the loop.

4.3 Motivating Speculation for PRE

Consider the CFG shown in Figure 4.3a, where each basic block has been annotated with its dynamic execution count, in parentheses. PRE, as previously described, attempts to move computations against the direction of flow of control. In this case, it would like to migrate the computation a+b:

1. along the reverse path $B_3 \rightarrow B_1 \rightarrow B_4 \rightarrow B_2$ into the end of block B_2;
2. along the reverse path $B_3 \rightarrow B_1 \rightarrow B_0$ into the end of block B_0.

Then, the computation a+b would be available at the beginning of block B_3 allowing it to be replaced by a temporary. As a consequence, 900 dynamic computations would optimized down to 100 dynamic computations.

However, PRE, in its original formulation, obeys an important optimality criterion:

> Never insert a computation on a path where it was not *already* present.

The purpose of the optimality criterion is to ensure that no path is de-optimized in an attempt to optimize other paths. It effectively prevents PRE from transforming the CFG shown in Figure 4.3a. since the path $B_0 \rightarrow B_1 \rightarrow B_2 \rightarrow B_4 \rightarrow B_5$ does not contain an occurrence of a+b [1]. Indeed, transforming the CFG shown in Figure 4.3a into the CFG shown in Figure 4.3b would have been an unmitigated disaster if the frequencies of basic blocks B_2 and B_3 were interchanged. Therefore, it is not enough to simply seek a formulation of PRE which abandons the optimality criterion.

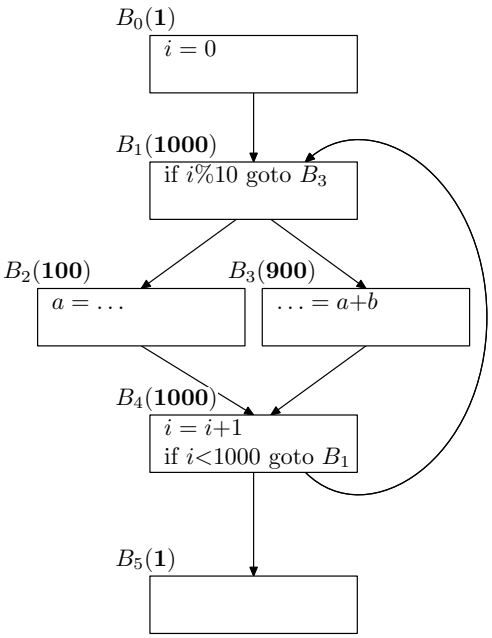
One should ask, instead, whether PRE can be reformulated to use the dynamic execution frequencies of computation as its optimality criterion.

4.4 Speculative PRE

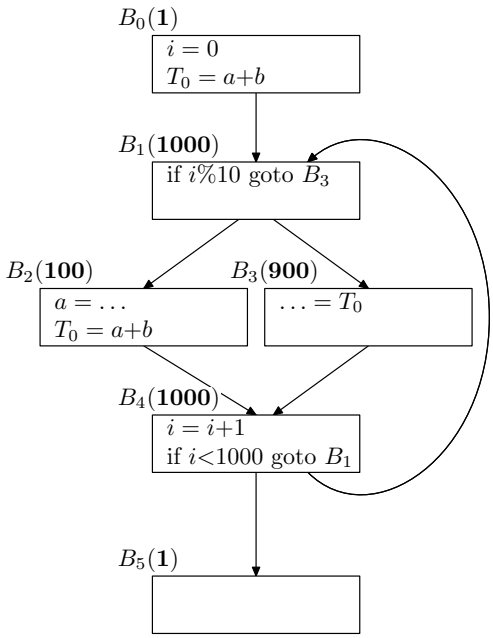
The idea of using profiling information to improve the expected performance of PRE is due to Horspool and Ho [HH97]. The name Speculative Partial Redundancy Elimination (SPRE) has been applied to the problem, and algorithms for finding optimal solutions have been presented [CX03, SHK04].

Cai and Xue [CX03] minimize the number of dynamic computations via a formulation called Minimum-Cut Partial Redundancy Elimination (MC-PRE). Their algorithm superimposes a flow network on the CFG being optimized. The flow network's edge capacities are derived from the CFG's edge frequency profile. From a minimal-cut of the flow network, the appropriate places in the CFG where computations should be inserted and deleted that *minimize* the dynamic number of

[1] Note that it is indeed possible to inspect the CFG shown in Figure 4.3a, and deduce that the path in question cannot be taken. However, this deduction is not in the domain of the PRE family of algorithms. Further, whether or not a path can be taken is effectively a question of reachability, which is, in general, undecidable.



(a) A CFG whose optimization is precluded by the PRE optimality criterion. 900 dynamic computations of a+b occur.



(b) Speculative PRE optimizes the CFG so that only 100 dynamic computations of a+b occur.

Figure 4.3: Motivating Speculative PRE

computations is derived. Subsequent work[SHK04] has shown that the problem can be mapped to a form of network flow problem known as Stone's Problem[Sto77].

These optimal solutions minimize the expected number of computations of the candidate expressions, based on the execution frequencies of the different paths through the program obtained from the program's profile.

4.4.1 Time Complexity

However, there are two problems with this approach:

1. Computing minimal-cuts (or, alternatively, maximal-flow) is an expensive operation. Overall, maximal-flow algorithms run in $O(n^3)$ time where n is the number of nodes in the flow network. Various attempts have been made to improve their efficiency: one of the best solutions so far, Goldberg's push-relabel HIPR algorithm[CG97], runs in $O(n^2\sqrt{m})$, where m is the number of edges in the flow network. However, the following facts remain:

   (a) Maximal-flow algorithms run in super-quadratic polynomial time.

   (b) The more efficient algorithms are also much more complicated and hence difficult to implement.

   (c) The process of reduction of SPRE to MC-PRE increases the number of nodes and edges under consideration. [SHK04] doubles the number of nodes, n, under consideration in the flow network, effectively scaling the time complexity by a factor of four (if $O(n^2\sqrt{m})$) or eight (if $O(n^3)$). [SHK04, CX03] both add edges that were not present in the original CFG.

2. A flow network must be built and minimally-cut for *each* expression under consideration.

SPRE is therefore very computationally expensive, a fact which we will verify experimentally in Chapter 7.

4.4.2 Conceptual Complexity

A collaboration with the IBM in Toronto, Canada, has shown us that intricate, mathematically involved algorithms are rarely adopted in industrial compilers.

The algorithms which reduce SPRE to the problem of optimizing flow networks are hardly intuitive, except, perhaps, to their inventors. Indeed, SPRE has been a rather technically involved algorithm:

1. [CX03] requires the CFG to be reduced with a pair of dataflow analyses, before transforming it into a flow network;

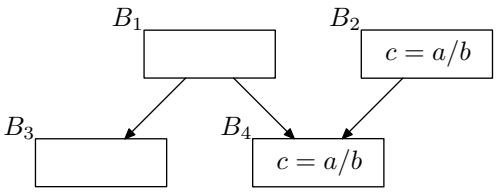
2. [SHK04] requires the CFG to be transformed into an instance of Stone's problem, before transforming it into a flow network;

3. Even suboptimal solutions such as [GBF98] and [HH97], which we discuss later in Chapter 6, require many complicated time-consuming dataflow analyses.

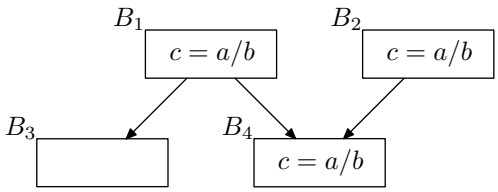
This complexity prevents the adoption of SPRE algorithms in industrial strength compilers, and motivates the development of computationally cheaper, intuitive approaches, which are competitive in optimizing capability.

4.4.3 SPRE: A Caveat

The optimality criterion of classical PRE is also its *safety* criterion: Preventing an expression from being inserted on a path which does not contain an instance of it *also* prevents a potentially faulting expression (such as a division) from being inserted on a path which does not contain an instance of it. Therefore, the transformed program will fault *only if* the original program faults, albeit earlier.



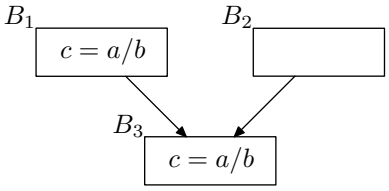
(a) $c = a/b$ is partially redundant in B_4.



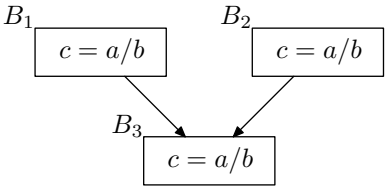
(b) $c = a/b$ cannot be inserted in B_1 since the path $B_1 \rightarrow B_3$ did not previously contain that computation, which may fault if b=0.

Preventing motion of potentially faulting expressions.

Figure 4.4: Example: The Safety Criterion for PRE



(a) $c = a/b$ is partially redundant in B_3.



(b) $c = a/b$ can be inserted in B_2 since the path $B_2 \rightarrow B_3$ will eventually perform that computation (in B_3). If a fault does occur (with b=0) it will occur in B_2 instead of B_3.

Allowing motion of potentially faulting expressions.

Figure 4.5: Example: The Safety Criterion for PRE

By disregarding the optimality criterion of classical PRE, SPRE simultaneously disregards the safety criteria, and should hence restrict itself only to the optimization of non-faulting expressions.

The Java Language and Precise Exception Semantics

The motion of expressions (to turn partial redundancies into full redundancies) is constrained considerably by the Java language's precise exception semantics [JSGB00, Section 11.3.1]:

> "Exceptions are precise: when the transfer of control takes place, all statements executed and expressions evaluated before the point from which the exception is thrown must appear to have taken place. No expressions, statements or parts thereof that occur after the point from which the exception is thrown may appear to have been evaluated."

That is, an exception handler in an optimized program must see exactly the same value for global variables (object fields and static variables) and local variables as it would in the unoptimized program. This is required since Java allows arbitrarily complex exception handlers which can access any program variable in their scopes.

This requirement, unfortunately, prevents even a *non-faulting* computation which assigns a value to a variable from being moved from after a Potentially Excepting Instruction (PEI) to before a PEI, since the exception handler would see different values for the variable in the optimized program versus the unoptimized program.

Similarly, Java's precise exception semantics would also prohibit the code motion shown in Figure 4.5b (which moves $c = a/b$ to the top of B_2), if the last instruction in B_2 contained an assignment to a variable.

Our collaboration with IBM on their TR-JIT for the Java language has shown that the verification that an optimization satisfies Java precise exception semantics can be conveniently implemented in a module which vetoes illegal code movements. In fact, the TR-JIT PRE phase does precisely this: it performs PRE via the Lazy Code Motion (LCM) algorithm—which is completely ignorant of Java's precise exception semantics and its constraints on expression movement. Then, a subsequent pass vetoes movements which are illegal.

The TR-JITs decoupling of the LCM algorithm from the module which implements the requisite Java semantics indicates that it is plausible for a compiler to use a PRE algorithm which did not originally make recourse to Java language semantics. We shall avail of this decoupling to present our algorithm in a language independent manner.

4.5 Motivating Isothermal Speculative Partial Redundancy Elimination

In this section, we sow the seed of an idea for an effective approximation to SPRE.

Optimal SPRE often goes to unnecessary trouble in its quest to transform a program to have a minimal number of computations: it considers minor, irrelevant differences in the execution frequencies between basic blocks. Consider two basic blocks, whose execution frequencies differ by a single count. SPRE will faithfully take that unit difference into consideration, by constructing and solving a flow network whose capacities are derived from those block frequencies, to little avail—if the remaining blocks in the program execute relatively infrequently, all that really matters is that the two blocks are "hot", not that one of them executes one more time than the other. In fact, that one block executes just one more time than the other could be true only for the particular run of the program which generated the profile. Consequently, what really matters is the differentiation of the program into oft-taken and almost-never-taken sections. Focusing on the oft-taken sections to receive special attention can be viewed as a special case of Amdahl's law[Amd67].

To model this dichotomy, we use the concept of isothermality previously developed. Revisiting

the CFG in Figure 4.3b, we find that the maximum execution frequency of any edge or block therein, is 1000. By choosing a threshold $\Theta = 80\%$, we derive a hot-cold division shown in Figure 4.6a. It is now easy to see that, in order to profitably transform the flow graph, the computation of $a + b$ occurring *inside* the hot region must be replaced by a temporary whose value was initialized to $a+b$ *outside* the hot region.

We optimistically place the computation of

$$T_0 = a+b$$

on each of the edges

$$B_0 \rightarrow B_1 \quad \text{and} \quad B_2 \rightarrow B_4$$

since they are the edges in the cold region of the CFG which are closest to the hot region. Since these edges transfer control from the cold region into the hot region, they are called **ingress** edges. Ingress edges are the best place to insert these optimistic computations, because:

1. They are part of the cold region of the graph, and so the cost of inserting computations on them is not onerous.

2. If an expensive computation in the hot region is rendered redundant, and replaced by a temporary, that temporary will have been held over the shortest possible path, as compared to all other possible insertion points in the cold region.

   For example, if $T_0 = a+b$ had been inserted on the edge $B_1 \rightarrow B_2$ (a cold edge), T_0 would need to hold its value over block B_2, thereby increasing the register pressure over that block.

Having inserted optimistic computations on the ingress edges, it can been seen that the expression a+b is available on all paths leading to block B_3. Hence, the computation of a+b in block B_3 can be replaced by temporary T_0.

After block straightening (the appending of a block with exactly one predecessor into that predecessor) has been performed to amalgamate the computations inserted on the ingress edges into the predecessor blocks B_0 and B_2, the resultant CFG is exactly as shown in Figure 4.3b, which is produced by optimal SPRE. That is, both ISPRE and optimal SPRE have moved the expression a+b *sideways* in the loop body from a frequently executed block to an infrequently executed block, thereby saving 800 computations.

Most importantly, since the availability analysis used to determine replacability is an efficient bit-vector analysis, all expressions in the program can be simultaneously optimized by ISPRE. This is unlike optimal SPRE where reduction to the problem of finding the maximum-flow of a flow network *for each expression* precludes the simultaneous processing of all expressions.

In fact, although the worst case time complexity of a bit-vector based dataflow algorithm is quadratic in the number of nodes in the CFG, it has been shown that the maximum number of times a node needs to be processed by the analysis is bounded by the loop nesting depth of the flow graph—a value which is rarely more than two[KU77]. Consequently, in practice, the performance of bit-vector algorithms is linear in the number of CFG nodes. Hence, this algorithm is much more efficient than SPRE. Even its closest non-speculative competitor, Lazy Code Motion (LCM), requires *four* unidirectional bit-vector analyses.

In the following sections, we will evolve the above sketch of ISPRE into a robust and efficient algorithm for effective, albeit suboptimal, Speculative Partial Redundancy Elimination, specifically one which is suitable for use in JIT compilers.

4.6 Isothermal Speculative Partial Redundancy Elimination

Isothermal SPRE (ISPRE) is a complete reformulation of SPRE. It is, by design, an approximate technique for performing code motion using information obtained from program profiles.

We distinguish two versions of ISPRE with the names *Single Pass ISPRE* and *Multipass ISPRE*, according to whether just one pass or several transformation passes over the program are performed. In the following, we describe single pass ISPRE.

4.6.1 Hot-Cold Division

ISPRE initially uses frequency profile information to divide a CFG G into two isothermal subgraphs:

1. a **hot** region G_{hot} consisting of the nodes and the edges executed more frequently than the given threshold frequency Θ; and

2. a **cold** region G_{cold} consisting of the remaining nodes and edges.

A pictorial representation of a division of a CFG into its hot and cold regions was shown in Figure 3.2 and is shown in Figure 4.6. In these figures, the shaded region represents G_{hot}. Please note that an edge can be contained in G_{hot} if and only if it emanates from a node in G_{hot} *and* enters a node in G_{hot}.

Topology

The example in Figure 3.2 illustrates that isothermal regions may be disconnected. Unlike other types of region, used in compiler literature, such as intervals, regions, tree-regions, and extended basic blocks, an isothermal region is simply a collection of nodes and edges whose execution frequencies are sufficiently near to each other, as determined by the frequency threshold, Θ. There are no additional notions of connectedness.

Structures that are identified by G_{hot} include (in order of complexity):

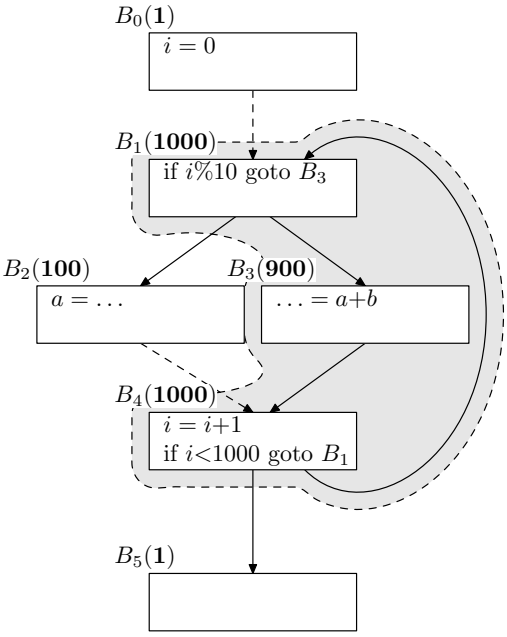
1. the body of a frequently iterated loop;

2. the particular side of a loop which executes frequently, in the case of a loop with a biased branch in its body;

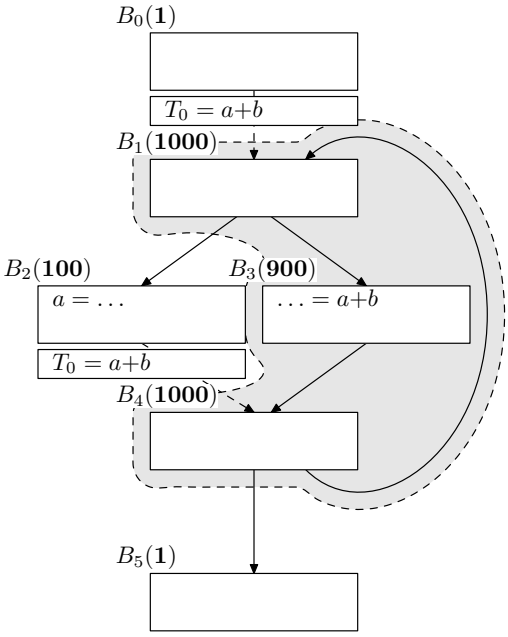
3. a frequently executed trace in a CFG; and

4. a single basic block, which has several cold predecessors and several cold successors.

Similarly, it is also possible for G_{cold} to contain a single disjoint edge, which has source and target basic blocks in G_{hot}.



(a) Division of the CFG shown in Figure 4.3a into hot and cold regions, using the threshold $\Theta = 80\%$. The ingress edges (which transfer control from the cold region to hot region) are $B_0 \rightarrow B_1$ and $B_2 \rightarrow B_4$.



(b) Insertion of computations on the ingress edges, create availability of expression a+b in B_3. a+b can therefore be deleted from B_3, yielding the same result as optimal SPRE (Figure 4.3b) saving 800 computations.

Figure 4.6: Example: Introducing Isothermal Speculative PRE

Definition

More formally, consider a CFG (N, E, s, t) with

1. a frequency profile represented by the function

$$\text{freq} : N \cup E \to \mathbb{Z}$$

2. a threshold, Θ.

Then, we prescribe

$$G_{hot} \equiv \langle N_H, E_H \rangle \tag{4.1a}$$
$$G_{cold} \equiv \langle N_C, E_C \rangle \tag{4.1b}$$

where

$$N_H \equiv \{u \mid u \in N \wedge \text{freq}(u) > \Theta\} \tag{4.2a}$$
$$E_H \equiv \{e \mid e \in E \wedge \text{freq}(e) > \Theta\} \tag{4.2b}$$
$$N_C \equiv N - N_H \tag{4.2c}$$
$$E_C \equiv E - E_H \tag{4.2d}$$

Given the division into hot and cold regions, we now define the *Ingress* edges as

$$\mathsf{Ingress} \equiv \{(u, v) \mid u \in N_C \wedge v \in N_H\} \tag{4.3}$$

The *Ingress* set consists of precisely those edges which transfer control from a cold node to a hot node. They act as "gates" into the hot region. Furthermore, since every edge adjacent to a cold node must be cold, the following condition holds:

$$\mathsf{Ingress} \subseteq E_C$$

Consequently, ingress edges, if split, can profitably act as landing pads for computations moved out of the hot region.

4.6.2 Analyses

ISPRE operates by inserting computations on edges in the *Ingress* set, in an attempt to make some computations in hot nodes become fully redundant. Those computations made fully redundant *in the hot region* are then replaced by references to temporaries which hold the saved values of equivalent computations *performed in the cold region*. This operation effectively moves computations from G_{hot} to G_{cold}.

The code motion is driven by the results of two analyses:

removability, which deduces instances of computations in the hot region that can be removed; and

necessity, which deduces edges in the *Ingress* set where computations must be inserted, to ensure the correctness of the deletions determined by the removability analysis.

Both *removability* and *necessity* are formulated as analyses that fall within the monotone dataflow framework of Kam and Ullman [KU77]. This implies that they can be implemented as unidirectional analyses using bit-vector representations of sets of expressions. That is, we can efficiently compute removability and necessity for all candidate expressions simultaneously.

4.6.3 Removability Analysis

An expression e is a possible candidate for removal if:

1. e is a **safe** expression. An expression is safe at a particular program point if computing it at that point cannot generate an exception. Exactly which expressions can be considered safe is processor dependent, processor-configuration dependent, language dependent, and (unfortunately) even dependent on instruction form.

   For an example of instruction safety being processor dependent, consider that the DIV instruction on the x86 processor executes the Interrupt 0 handler if the divisor is zero[Int90]. On the POWERPC processor, however, the fixed-point DIVD instruction does not raise an exception when the divisor is zero, but only sets a bit in the Fixed-Point Exception Register (XER)[PPC93].

   To see instruction safety being processor-configuration dependent, consider that the behaviour of floating-point division on the POWERPC is configurable via the Floating-Point Status and Control Register (FPSCR), which contains an "enable" bit to allow/disallow a system error handler to run when a divide exception occurs. A similar configurability applies on the POWERPC for:

   (a) the floating-point comparison instruction (FCMP) (which may be given an operand which is Not-a-Number (NAN);

   (b) the floating-point subtraction instruction (FSUB) (which may be given infinity as its operands);

   (c) the floating-point multiplication instruction (FMUL) (which may be given infinity and zero as its operands);

   (d) almost any floating-point arithmetic operation (whose operands may cause it to compute a result which overflows, underflows, or loses precision).

   The mandating of instruction safety by a programming language is exemplified by the Java programming language whose IDIV instruction must check for the divisor being equal to zero and, if so, raise an `ArithmeticException`[JSGB00]. Hence, the Java IDIV instruction is unsafe on all processors, regardless of instruction semantics and processor configuration. Note, however, that if the divisor is known to be non-zero, the IDIV instruction can be considered safe.

   Finally, the role of particular forms of an instruction in discussing safety cannot be underestimated. The x86 processor, though passively reporting overflows and underflows for its arithmetic instructions (via status bits in the FLAGS register), allows operands to be accessed directly from memory—it has a CISC instruction set. Any memory address accessed outside addressable boundaries will cause a General Protection Fault and execution of the fault-handler

thereupon. This renders even the mundane ADD instruction unsafe when it uses memory operands.

2. there is an **upwards exposed** instance of e in at least one node $n \in N_H$. An expression instance $a \oplus b$, for some operator $\oplus$ is upwards exposed (or **anticipatable**) in a basic block if it is not preceded in that basic block by any assignments to a or b. Equivalently, in the terminology of dataflow analysis, it is not preceded by any statements which "kill" e.

   Intuitively, if an instance of e is upwards exposed it can be moved to the beginning of the block without changing the value that it computes, or violating precise exception semantics.

The removability analysis is based on the assumption that *every* candidate expression is **available** on *every* edge in the *Ingress* set. An expression e is available at a point P if it has been computed on every path leading to P without being subsequently killed (i.e., no operand of e has been modified). The necessity analysis will ensure that our assumption is satisfied.

Given the assumption, removability analysis just becomes the available expressions analysis [ASU86]. A candidate expression e is removable from node n if and only if:

1. n contains an upwards exposed use of e; and
2. e is available on entry to n.

We depict these conditions of removability in Figure 4.7. The failure of first condition is depicted in Figures 4.7b and 4.7c. The failure of second condition is depicted in Figures 4.7a and 4.7c. Figure 4.7d show both conditions being fulfilled. In that case, replacement *may* occur as long as both conditions are also true for all other paths leading to n.

The dataflow equations, with modifications to incorporate our assumption, can now be stated. Our dataflow equations, are described by two components:

basic block summaries, which state properties of each basic block. A block summary is derived for a given block *without* looking at any other basic blocks.

dataflow equations, which derive further properties of each basic block, by propagating properties of other basic blocks through the CFG.

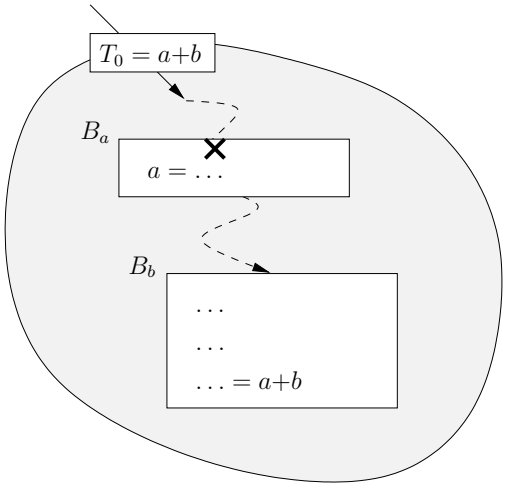
Basic Block Summaries

First, the following sets are computed for each basic block, b, by examining the IR in the block.

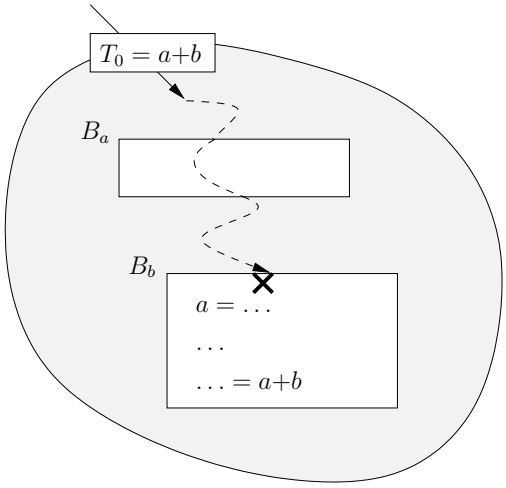
$$\mathsf{X\text{-}USES}(b) \equiv \{e \mid \text{expression } e \text{ occurs in } b \text{ and is not preceded by any reassignment of operands in } e\ \} \tag{4.4a}$$

$$\mathsf{COMP}(b) \equiv \{e \mid \text{expression } e \text{ occurs in } b \text{ and is not followed by any reassignment of operands in } e\ \} \tag{4.4b}$$

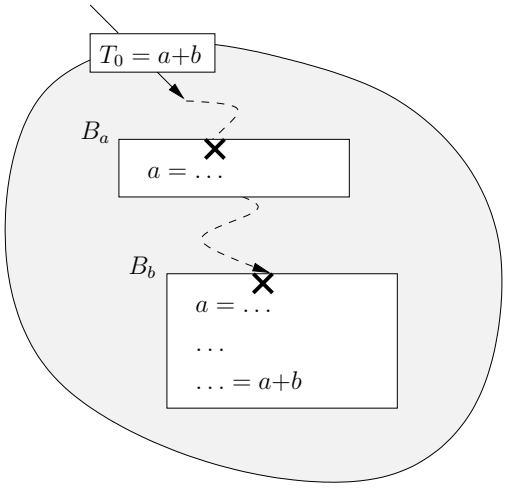
$$\mathsf{KILL}(b) \equiv \{e \mid \text{block } b \text{ contains a statement which may reassign an operand of } e\ \} \tag{4.4c}$$



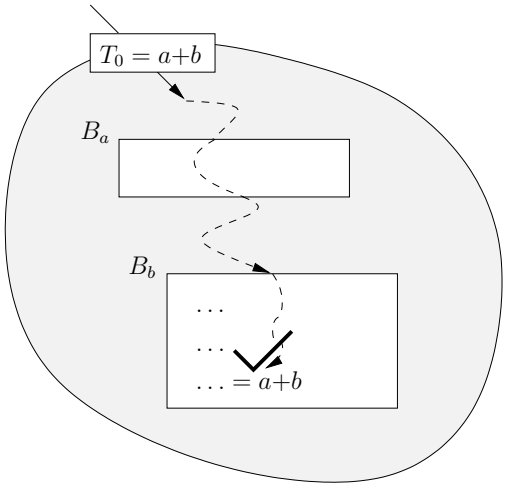
(a) The assumed computation of $a + b$ on the ingress edge is killed before it can reach hot block B_b. Hence, $a + b$ is not removable from B_b.



(b) The assumed computation of $a + b$ on the ingress edge reaches hot block B_b. But, $a + b$ is not upwards exposed in B_b and so cannot be removed.



(c) The assumed computation of $a + b$ is killed, and the occurrence of $a + b$ is not upwards exposed. Removal definitely cannot occur.



(d) The assumed computation of $a + b$ on the ingress edge is not killed *and* $a + b$ is upwards exposed in B_b. Removal may occur.

The hot region is shown shaded. The split ingress edge, shown entering the hot region, has the computation $T_0 = a{+}b$ placed on it.

Figure 4.7: Removability of a computation from the hot region

Definition 4.4a yields the set of upwards exposed expressions in a basic block. Definition 4.4b yields the set of expressions computed in a basic block. Definition 4.4c yields the set of expressions killed by a basic block.

Dataflow Equations

In the following, *Candidates* represents the set of all candidate expressions. The following equations are solved simultaneously for the largest solution:

$$\forall b \in N\text{:}$$

$$\mathsf{AV\text{-}OUT}(b) = (\mathsf{AV\text{-}IN}(b) - \mathsf{KILL}(b)) \cup \mathsf{COMP}(b) \tag{4.5a}$$

$$\mathsf{AV\text{-}IN}(b) = \bigcap_{p \in preds(b)} \begin{cases} \text{Candidates}, & \text{if } (p, b) \in \mathsf{Ingress}; \\ \mathsf{AV\text{-}OUT}(p), & \text{otherwise.} \end{cases} \tag{4.5b}$$

Equation 4.5a states that an expression is available at the end of a basic block, if either:

1. it was available at the beginning *and* not killed; or
2. it is computed in the block.

Equation 4.5b states that an expression is available at the entry to a block, b, if for all predecessor blocks, p:

1. the expression is available at the end of p; or failing that
2. the edge from p to b is an ingress edge which, by assumption, contains a computation of the expression.

Note that the equations for AV-IN and AV-OUT are solved for all blocks in the CFG, not just in the hot region. This is because expression availability within the cold region is useful in completing the necessity analysis.

Initialization

The most common method for solving sets of equations involves initializing the value of $\mathsf{AV\text{-}IN}(b)$ and $\mathsf{AV\text{-}OUT}(b)$ for all basic blocks, b, before iterating Equations 4.5a and 4.5b in the order shown for all basic blocks, until the sets converge. For such an "iterate-to-convergence" algorithm, the appropriate initialization is:

$$\mathsf{AV\text{-}IN}(s) = \emptyset \tag{4.6a}$$

$$\forall b \in N - \{s\}\text{:}$$

$$\mathsf{AV\text{-}IN}(b) = \text{Candidates} \tag{4.6b}$$

The initialization in Equation 4.6a is performed since no expressions are available at the beginning of the entry basic block, s. This is because our analysis is intra-procedural. Equation 4.6b optimistically assumes that all candidate expressions are initially available everywhere (except at

entry). This is the standard initialization; it is provable that at convergence only those expressions which are actually available will remain in the set [KU77].

The Removability Solution

After the above dataflow analysis has converged, we may compute the solutions to the equations for *Removable*, which indicate which upwards exposed uses of expressions can be removed from each node in G_{hot}.

$\forall b \in N_H$:

$$\mathsf{REMOVABLE}(b) \equiv \mathsf{AV\text{-}IN}(b) \cap \mathsf{X\text{-}USES}(b) \tag{4.7}$$

Definition 4.7 simply states what is depicted graphically in Figure 4.7d, namely that if an expression is available on entry to a basic block and is upwards-exposed within that basic block, then that occurrence can be removed.

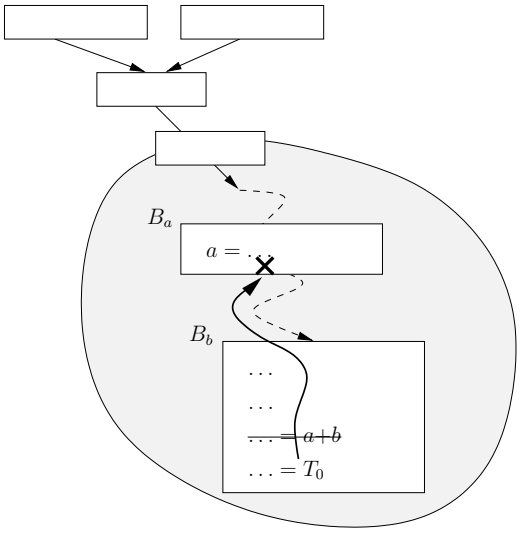
4.6.4 Necessity Analysis

The solutions for the REMOVABLE sets assume that computations of all candidate expressions are available on the Ingress edges. That assumption could be satisfied by inserting the computations on all those edges. However, that would be a suboptimal solution because not all the insertions would be needed. There are two reasons why inserting an expression e on an edge (u, v) in the Ingress set may be unnecessary for a given path to a replacement point:

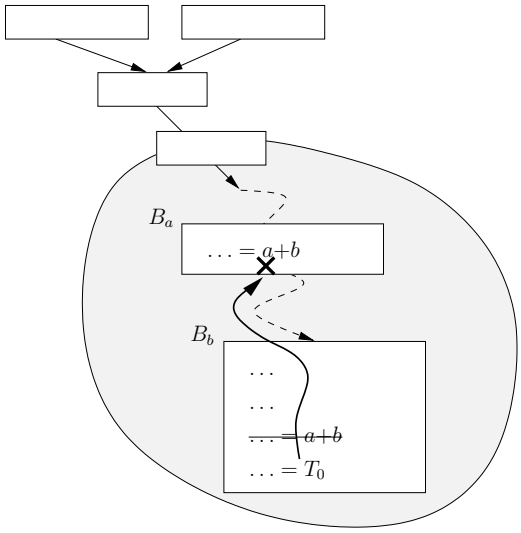
1. *It is useless:*
   (a) the expression may not reach any exposed use of e in the hot region which has been deemed to be removable;
   (b) the expression e may be killed *en route* to the replacement point.
   This situation is depicted in Figure 4.8a.
2. *It is redundant:*
   (a) the expression e may be computed *en route* to the replacement point.
   This situation is depicted in Figure 4.8b.
   (b) the expression e may already be available at the end of block u.
   This situation is depicted in Figure 4.8c.

Dataflow Equations

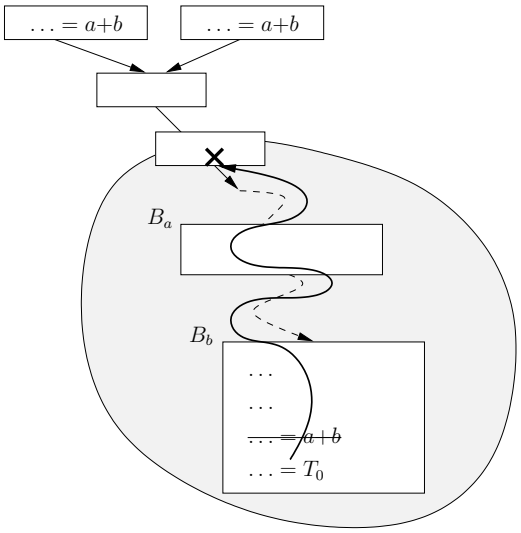
The following equations for NEED-IN and NEED-OUT determine whether insertions of the candidate expressions on ingress edges would be useless or not, *regardless of external availability.* The following equations are solved simultaneously for the largest solution.



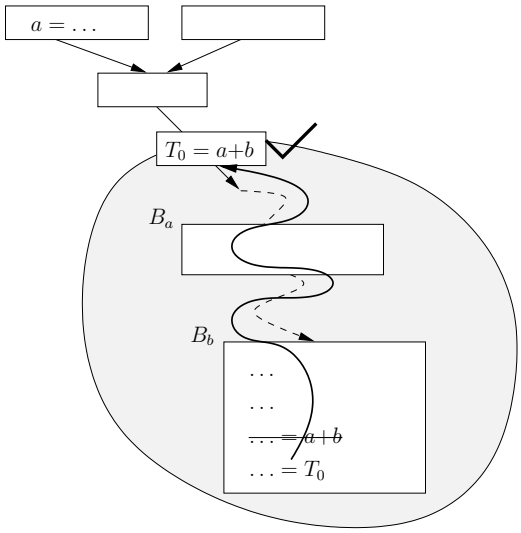
(a) Insertion on the ingress edge is useless since the expression will be killed *en route* to the replacement point.



(b) Insertion on the ingress edge is redundant since the expression will be computed en route to the replacement point.



(c) Insertion on the ingress edge is redundant since it is available there.



(d) The expression is not available on the ingress edge *and* its insertion will be neither redundant nor killed en route to the replacement point. Insertion occurs.

The computation a+b in the hot region (shown shaded) has been deemed removable by the removability analysis. Hence, $\ldots = a+b$ has been deleted and replaced with $\ldots = T_0$. The necessity analysis must now decide whether to insert $T_0 = a+b$ on the ingress edge (shown entering the hot region).

Figure 4.8: Necessity Analysis

$\forall b \in N_H$:

$$\mathsf{NEED\text{-}IN}(b) = (\mathsf{NEED\text{-}OUT}(b) - \mathsf{COMP}(b)) \cup \mathsf{REMOVABLE}(b) \tag{4.8a}$$

$$\mathsf{NEED\text{-}OUT}(b) = \bigcup_{s \in succs(b)} \mathsf{NEED\text{-}IN}(s) \tag{4.8b}$$

Equation 4.8a states that an expression is necessary at the beginning of a basic block if:

1. it has been removed from that block; or
2. it is necessary at the end of the block (due to having been removed from a successor block) and is not computed in the block.

Equation 4.8b states that an expression is necessary at the end of a block only if it is necessary at the beginning of any immediate successor block.

This analysis is performed only for basic blocks in the hot region: since we are removing computations only in the hot region, we need only consider paths from the ingress edges to the replacement points which remain in the hot region. If there exists a path from an ingress edge to a replacement point which ever exits the hot region, a suffix of that path will still travel from an ingress edge (via which it re-enters the hot region) to the replacement point. That suffix path will be considered by the necessity analysis.

Initialization

As with the availability analysis previously described, an "iterate-to-convergence" algorithm will commonly be used to find the largest solution. For this algorithm, the appropriate initialization is:

$\forall b \in N_H$:

$$\mathsf{NEED\text{-}OUT}(b) = \emptyset \tag{4.9a}$$

The initialization of Equation 4.9a is performed since it is optimistically assumed that no expression insertions are initially necessary. As with the available expressions analysis, this initialization is standard; it is provable that at convergence all those expressions that are necessary will be in the set [KU77].

The Insertion Solution

The solution to the previous set of equations are used to construct the INSERT sets, which take external availability into consideration as depicted in Figure 4.8c. They derive precisely those ingress edges on which computations must be inserted.

$\forall (u, v) \in \mathsf{Ingress}$:

$$\mathsf{INSERT}(u, v) \equiv \mathsf{NEED\text{-}IN}(v) \; - \; \mathsf{AV\text{-}OUT}(u) \tag{4.10}$$

Equation 4.10 says that a computation should be inserted on an ingress edge only if it is needed in the hot region and not already available on that ingress edge.

4.7 ISPRE Algorithm Specification

Algorithm: Isothermal Speculative PRE
Input: A CFG (N, E, s, t), frequency profile $\text{freq} : (N \cup E) \to \mathbb{Z}$, *Candidates*, threshold $\Theta \in (0\%, 100\%)$
Output: A deletion set, $\mathsf{REMOVABLE}(b)$, $b \in N_H$
An insertion set, $\mathsf{INSERT}(e)$, $e \in \mathsf{Ingress}$

1. Hot Cold Categorization: Use Θ to divide the CFG into two isothermal regions, G_{cold} and G_{hot}:

$$N_H = \{u \mid u \in N \wedge \text{freq}(u) > \Theta\}$$
$$E_H = \{e \mid e \in E \wedge \text{freq}(e) > \Theta\}$$
$$N_C = N - N_H$$
$$E_C = E - E_H$$
$$G_{hot} = \langle N_H, E_H \rangle$$
$$G_{cold} = \langle N_C, E_C \rangle$$

2. Derive the ingress edges which lead from G_{cold} to G_{hot}.

$$\mathsf{Ingress} = \{(u, v) \mid u \in N_C \wedge v \in N_H\}$$

3. Analyze $G_{hot} \cup G_{cold}$: Assume computations are available on ingress edges and compute availability:

$$\mathsf{AV\text{-}OUT}(b) = (\mathsf{AV\text{-}IN}(b) - \mathsf{KILL}(b)) \cup \mathsf{COMP}(b)$$
$$\mathsf{AV\text{-}IN}(b) = \bigcap_{p \in preds(b)} \begin{cases} \text{Candidates}, & \text{if } (p, b) \in \mathsf{Ingress}; \\ \mathsf{AV\text{-}OUT}(p), & \text{otherwise.} \end{cases}$$

4. Analyze G_{hot} only:

   (a) Derive redundant computations in G_{hot}:

$$\mathsf{REMOVABLE}(b) = \mathsf{AV\text{-}IN}(b) \cap \mathsf{X\text{-}USES}(b)$$

   (b) Derive computations which must be inserted on ingress edges to maintain availability:

$$\mathsf{NEED\text{-}IN}(b) = (\mathsf{NEED\text{-}OUT}(b) - \mathsf{COMP}(b)) \cup \mathsf{REMOVABLE}(b)$$
$$\mathsf{NEED\text{-}OUT}(b) = \bigcup_{s \in succs(b)} \mathsf{NEED\text{-}IN}(s)$$
$$\mathsf{INSERT}(u, v) = \mathsf{NEED\text{-}IN}(v) - \mathsf{AV\text{-}OUT}(u)$$

5. From each basic block $b \in N_H$, remove upwards-exposed expressions which are members of $\mathsf{REMOVABLE}(b)$.

6. On each edge $e \in \mathsf{Ingress}$, insert expressions which are members of $\mathsf{INSERT}(e)$.

4.8 An ISPRE Example

Consider Figure 4.9 which depicts a loop with a biased body.

```
1  i = 0;
2  do {
3          if ((i % 10) != 0) {
4                  /* Frequent */
5                  ... = a + b;
6          } else {
7                  /* Infrequent */
8                  a = ...;
9          }
10         i = i + 1;
11 } while (i < 1000);
```

Figure 4.9: Example: Program to be optimized by ISPRE

A CFG corresponding to this code is shown in Figure 4.10a, where the annotations in parentheses next to basic block numbers show their expected execution frequencies. We shall assume that the execution frequencies have been collected from a profile of a previous execution of the program.

Our sole candidate expression shall be a+b: it is computed in the hot region and is upwards exposed in the hot basic block where it occurs, B_3. Initially 900 computations of a+b can be expected in block B_3.

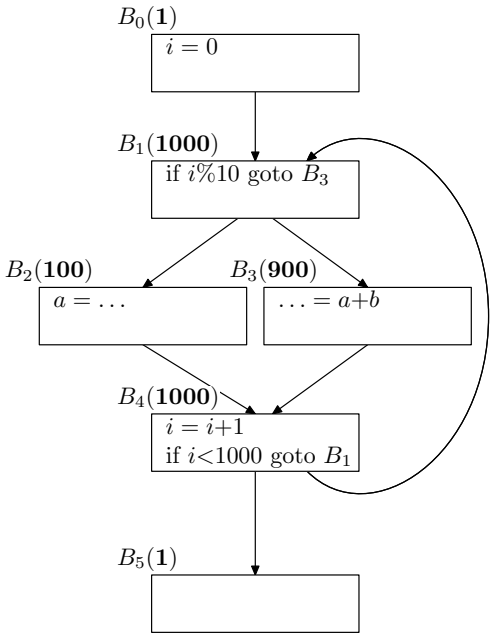
Using $\Theta = 90\%$, we derive the isothermal regions G_{hot} and G_{cold} shown in Figure 4.10b, where G_{hot} is depicted as a shaded region. The hot region G_{hot} consists of blocks B_1, B_3, and B_4, and edges $B_1 \rightarrow B_3$, $B_3 \rightarrow B_4$, and $B_4 \rightarrow B_1$. The set of Ingress edges consists of edges $B_0 \rightarrow B_1$ and $B_2 \rightarrow B_4$, which are drawn dashed. It is important to note that the entire loop has not been identified as G_{hot}, but only the frequently executed side of the loop. Figure 4.10c simplifies the presentation of this example by ignoring computations that are not of interest (i.e., those that do not change the value of a or b).

Figure 4.10d depicts the assumption made by the removability analysis, namely that candidate expressions are available on the ingress edges. Their depiction here is purely for the sake of exposition.

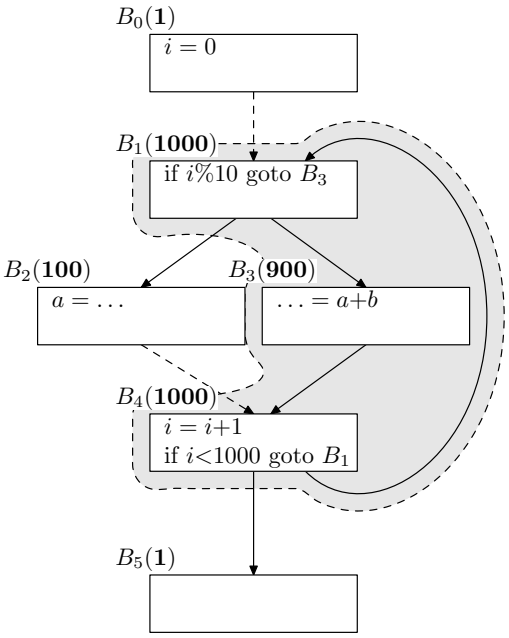
Figure 4.10e shows that the removability of a+b from B_3 arises from availability of that computation on *all* paths from the ingress edges.

Figure 4.10f shows the result of removing a+b from B_3. Then, the neededness analysis is performed, in this case, by simply tracing the availability paths in reverse (from the replacement point to the ingress edges). On no such reverse path is an assignment to a or b, or a computation of a+b encountered. Hence, the insertion of a+b is necessary on the ingress edges (as far as G_{hot} is concerned). Furthermore, there are no computations of a+b from G_{cold} that reach either ingress edge, so insertions on both ingress edges are required.

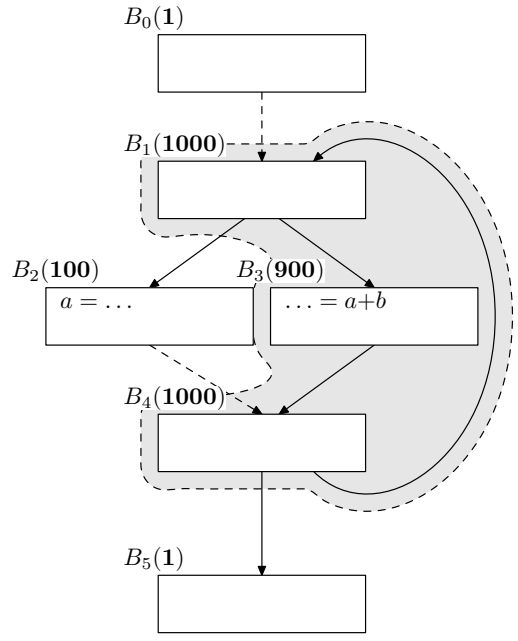
Figure 4.10g depicts the result of inserting computations on ingress edges. Block straightening has been performed: Since blocks B_0 and B_2 have no immediate successors other than blocks B_1 and B_4 respectively, any code inserted on edges leading to those immediate successors can be integrated



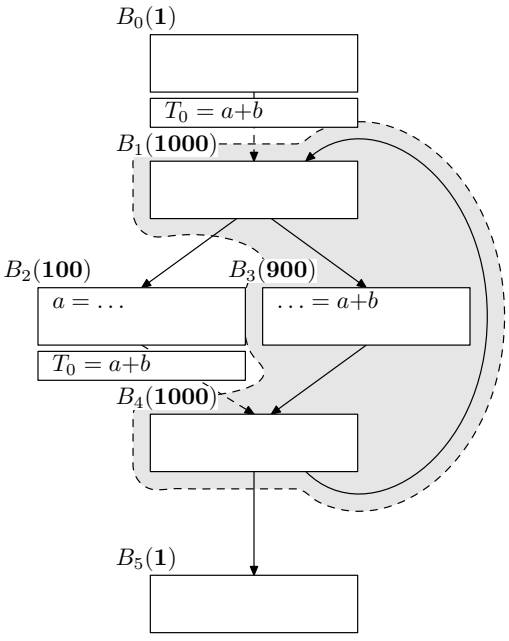
(a) A CFG with 900 computations of a+b.



(b) Using $\Theta = 90\%$, derive G_{cold} and G_{hot} (shown shaded). Only edges contained completely within the shaded region are part of G_{hot}.

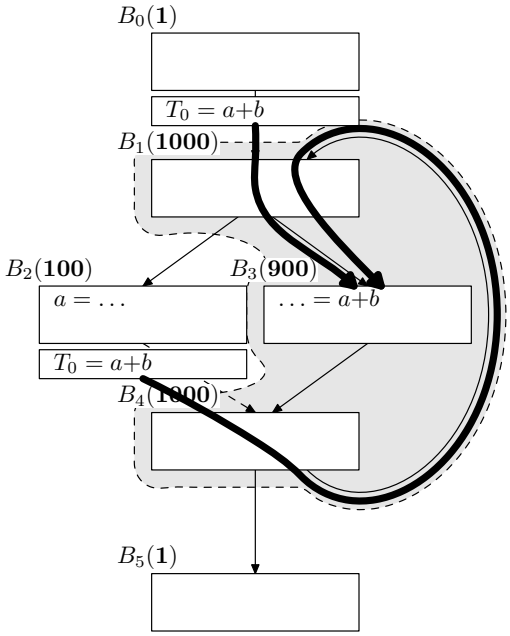


(c) Disregard all expressions not involving a+b, for clarity.

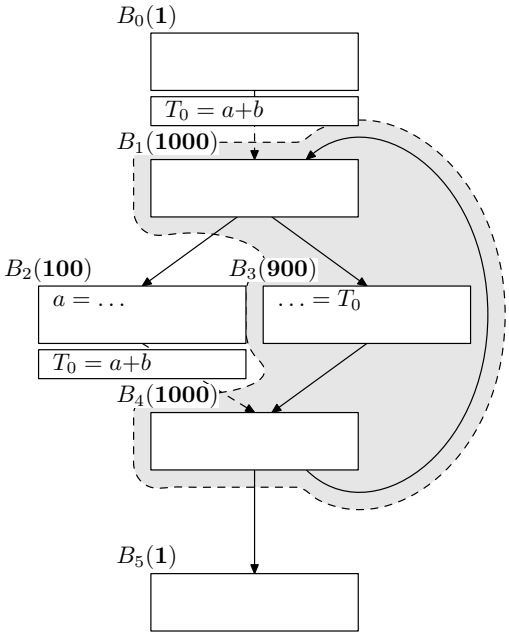


(d) *Tentatively* insert computations on ingress edges to satisfy the "available on ingress edges" assumption which will be made by the removability analysis.

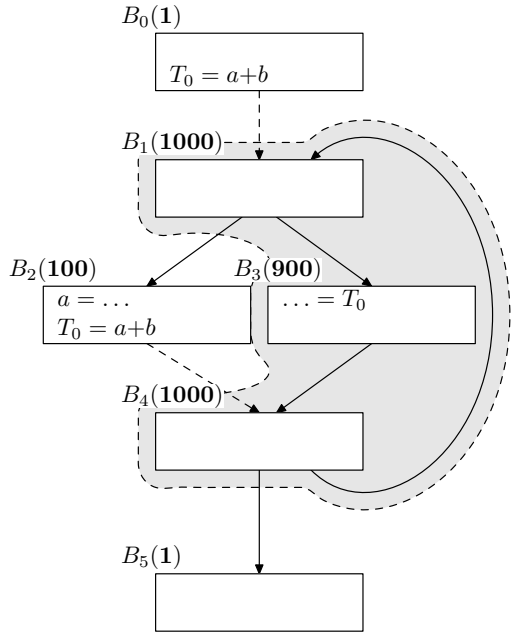
Figure 4.10: Example: ISPRE in action



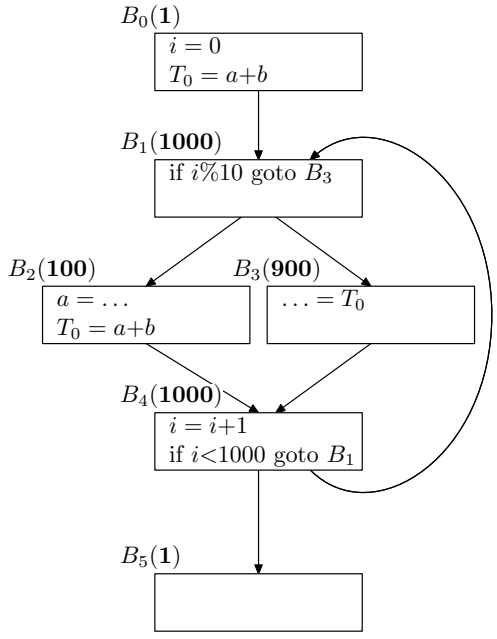
(e) Removability analysis deduces that a+b is available on all paths taken from the ingress edges to the upwards exposed occurrence of a+b in B_3.



(f) The occurrence of a+b in B_3 is replaced by the temporary T_0. Neededness analysis shows that both "tentative" ingress edge insertions are indeed required.



(g) Block straightening is performed to amalgamate the insertions on the egress edges into their immediate predecessor blocks.



(h) All previously hidden expressions (not involving a+b) are brought back into view. Only 100 computations of a+b now occur.

Figure 4.10: Example: ISPRE in action (continued)

into blocks B_0 and B_1. This prevents the creation of new basic blocks and new unconditional jumps.

Finally, all irrelevant expressions that were elided for clarity of exposition are restored showing the final result in Figure 4.10h, where only 100 computations of a+b can be expected to occur.

4.9 Proof of Correctness

The ISPRE algorithm consists of two analyses: removability analysis and necessity analysis, in that order.

4.9.1 Correctness of the Removability Analysis

Removability analysis inserts computations (tentatively) on ingress edges. The result of each computation is assigned to a temporary (e.g., T_0), hitherto unused in the CFG. Hence, no computation in the program produces a different result than it previously did. Further, since ISPRE restricts itself to inserting safe computations on ingress edges, a fault will not occur during computations on ingress edges. Thus far, program behaviour has been preserved.

The removability analysis then:

1. deletes computations in the hot region which are available; and
2. replaces deleted computations by temporary variables (assigned on ingress edges).

The operations of availability analysis, available expression deletion and replacement by temporaries have been proven correct in [Hec77]. Hence, thus far, program behaviour has still been preserved.

4.9.2 Correctness of the Necessity Analysis

The necessity analysis, *by definition*, determines, for each computation inserted on an ingress edge, whether a path exists to a deletion of that computation in hot region, such that the value computed on the ingress edge is used at the point of deletion (via a temporary introduced by the removability analysis). If no such path exists, the computation on the ingress edge is not necessary, and is deleted since its value is never used.

Consequently, program behaviour is still preserved. (Note: Even if this analysis were skipped, program behaviour would still be correct, albeit inefficient, since computations whose values are never used—dead computations—would occur).

4.10 The Derivation of ISPRE from PRE

Having formulated ISPRE, we must tout one of its prime virtues—it leverages the design of non-speculative formulations of PRE.

PRE algorithms basically involve a two-step process:

1. The placement of computations, which turn partial redundancies into full redundancies; and
2. The detection and deletion of (the newly created) fully redundant computations.

Most PRE algorithms use multiple bit-vector analyses to derive placement points for computations. For example, PRE's mostly widely adopted formulation, LCM [KRS92], uses four bitvector analyses to derive placement points for computations. The original formulation of PRE [MR79], by Morel and Renvoise, uses two bit-vector analyses.

ISPRE, likewise, must obtain placement points for computations and does so as a simple consequence of the division of the program into hot and cold regions—the ingress edges, which lead from cold basic blocks to hot basic blocks, are the chosen locations. No bit-vector analyses are required.

Thereupon, the detection and deletion of fully redundant computations, is performed via an available expressions analysis, as do most PRE formulations.

Thus, the original structure of PRE algorithms is preserved in ISPRE. Yet, we remind the reader that ISPRE performs *speculative* PRE. That is, we have leveraged knowledge of the formulations of non-speculative versions of an optimization in the construction of a speculative version of that optimization.

It is impossible to belabour the importance of this virtue: there are many compiler optimizations awaiting conversion to use speculation. The construction of ISPRE demonstrates that knowledge of the formulation of a non-speculative optimization need not go to waste, but can be used as a sound foundation on which to construct speculative versions of those optimizations.

Thus, we have demonstrated *claim* 3 with respect to ISPRE.

In conclusion, this chapter has introduced a new way to implement PRE in a compiler. Unlike other PRE implementations, ISPRE does not provably provide optimality for any cost metric—neither for lifetimes of cached expression values nor for the expected number of expression computations. However, the method is simple to implement, is fast, and produces results that are close to those produced by optimal SPRE algorithms. **These claims are verified experimentally in Chapter 7.**

Since this technique is fast enough to be used by JIT compilers, ISPRE has the potential to become the PRE algorithm of choice in future compilers, especially JIT compilers.

Chapter 5

PDE - Partial Dead Code Elimination

In the previous chapter, we demonstrated how an important class of speculative optimization (SPRE) used in optimizing compilers can benefit from the isothermal method.

In this chapter, we harness the isothermal technique to empower another kind of speculative optimization—Speculative Partial Dead Code Elimination (SPDCE)—by keeping it as cost effective as its less powerful non-speculative cousin, PDCE.

5.1 Introduction

Dead Code Elimination (DCE) is the removal of computations whose results are never used. An elementary opportunity for application of this optimization is shown in Figure 5.1:

```
1  x = a + b
2  x = c + d
3  print x
```

Figure 5.1: Example: A fully dead assignment

Here, we assume that each name refers to a unique storage location; that is, we assume that aliasing of variable names does not occur. We also assume that expressions on the right-hand side of the assignment operator (such as a+b) are **pure**: their only effect is to compute the specified mathematical relation without affecting their operands or any other datum or generating an exception. With these assumptions, it is clear that after the first computation, the value in x is not used but is overwritten by the result of the subsequent computation. An equivalent optimized program is shown in Figure 5.2:

```
1  x = c + d
2  print x
```

Figure 5.2: Example: A fully dead assignment eliminated

The optimization producing the program in Figure 5.2 is easily driven by an elementary program analysis called liveness analysis. Liveness analysis computes, for each assignment to a variable, whether there is a path from the assignment to some use of that variable such that the path does not contain an assignment to that variable. If one or more such paths exist, the variable is considered to be **live** at the point of assignment. Otherwise, the variable is considered **dead** at the point of assignment and the assignment removed.

However, dead variables are rare. It is much more common to find variables that are dead only on a subset of program paths from the point of assignment. These variables are called **partially dead variables** and are amenable to optimization by a well-known algorithm called Partial Dead Code Elimination (PDCE).

An elementary example of a program amenable to PDCE is shown in Figure 5.3:

```
1  x = e1 /* expression 1 */
2  if (a < b) {
3    x = e2 /* expression 2 */
4  } else {
5    y = ...
6  }
7  print (x+y)
```

Figure 5.3: Example: A partially dead assignment

The value stored in x by the first assignment (on line 1) is overwritten by the second assignment (on line 3) *only* if the if-statement condition evaluates to true. Hence, x is live at the first assignment since its value will be used in the statement "print (x+y)" if the if-condition evaluates to false. Consequently, plain DCE cannot delete the first assignment to x.

PDCE, however, will perform a two-step optimization of code insertion followed by code deletion. It will first *sink* the first assignment to x into both branches of the if-statement to obtain the program shown in Figure 5.4a:

```
1  x = e1 /* expression 1 */
2  if (a < b) {
3    x = e1 /* expression 1 */
4    x = e2 /* expression 2 */
5  } else {
6    x = e1 /* expression 1 */
7    y = ...
8  }
9  print (x+y)
```

(a) Sinking of the partially dead assignment x=e1.

```
1
2  if (a < b) {
3
4    x = e2 /* expression 2 */
5  } else {
6    x = e1 /* expression 1 */
7    y = ...
8  }
9  print (x+y)
```

(b) Removal of the now fully dead assignment x=e1.

Figure 5.4: Example: Removing a partially dead assignment

PDCE will then perform plain DCE to obtain the program shown in Figure 5.4b, in which the partially dead assignment to x (on line 1) has been eliminated.

5.2 Adding Speculation

PDCE, however, does not consider the relative execution frequencies of different paths through the program. Therefore, it must work conservatively, being careful not to worsen performance for any set of execution frequencies of the paths through the program. In this chapter, we develop a successor to PDCE called Isothermal Speculative Partial Dead Code Elimination (ISPDCE), so named because:

1. It is *speculative:* Execution frequencies of blocks and edges in the CFG are taken into account.
2. It is *isothermal:* The division of the CFG into two regions of differing "heat" guide the optimization, in the manner of the ISPRE algorithm discussed in Chapter 4.

5.2.1 ISPRE: A Recapitulation

To create the ISPDCE algorithm, we reprise the technique used to effect ISPRE.

The essence of ISPRE is the tentative insertion of computations on edges that depart from the cold region and enter the hot region, the so-called *ingress edges*. Thereupon, plain Global Common Subexpression Elimination (GCSE) is performed and, on account of the tentative insertions previously made, detects that hitherto partially redundant computations in the hot region are now completely redundant and eliminates them. Finally, those tentative insertions that actually contributed to the creation of fully redundant computations are made permanent, thereby resulting in code motion from the hot region to the cold region.

The crucial observation to be made here is that ISPRE, being a PRE algorithm, is enabled by *hoisting* of expressions, the movement of expressions *against the direction* of the flow of control. By placing computations on the ingress edges, hoisting implicitly occurs.

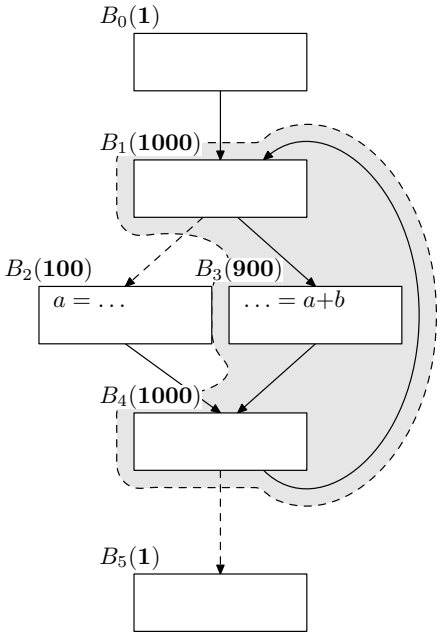
The motivating ISPRE example, which we reprise from the previous chapter, is shown in Figure 5.5a. Here the computation $\ldots = a+b$ is partially redundant; successive iterations of the right-hand side of the loop recompute it, despite no intervening change to the value of either a or b.

5.2.2 ISPDCE: An Analogue to ISPRE

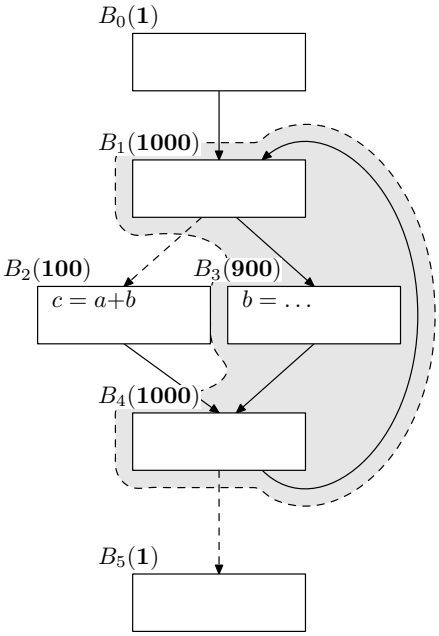
Similarly, consider Figure 5.5b, where the assignment $b = \ldots$ is performed redundantly in the right-hand side of the loop; successive iterations of the right-hand side of the loop serve only to overwrite the value that was previously computed with exactly the same value.

In much the same way that PRE algorithms are founded on the concept of hoisting, PDCE algorithms are founded on the concept of sinking, the movement of expressions *in the direction* of the flow of control. Therefore, we might hypothesize an analogue to ISPRE algorithm, illustrated in Figure 5.6 as follows:

1. Divide the CFG into hot and cold regions, as shown in Figure 5.6a.
2. Place assignments which are partially dead onto edges that leave the hot region and enter the cold region (which we call **egress edges**). This will turn partially dead assignments into fully dead assignments, as shown in Figure 5.6b.
3. Delete assignments to dead variables via plain DCE, as shown in Figure 5.6c, and



(a) 900 partially redundant computations of a+b in G_{hot}.



(b) 900 partially dead assignments (involving computations) of $b = \ldots$ in G_{hot}.

Figure 5.5: Using ISPRE to motivate ISPDCE.

4. Perform block straightening on the CFG to yield the optimized program, as shown in Figure 5.6d.

The incorrectness of naïve ISPDCE

A hint of the incorrectness of our naïve ISPDCE algorithm can already be seen in Figure 5.6d. Here, after the loop has finished executing, the assignment $b = \ldots$ is *always* executed in the optimized program, even though it may not have executed even once in the original program.

A more profound failure of the algorithm is shown in Figure 5.7. The unoptimized version, shown in Figure 5.7a, has a statement in B_0 which initializes the variable b. This definition can be used, in the unoptimized program, in B_2 via the path $B_0 \rightarrow B_1 \rightarrow B_2$. This access is depicted in Figure 5.7b.

However, in the optimized version, shown in Figure 5.7c, the value assigned to b in B_0 can *never* be used in B_2 since it is overwritten by the assignment that was removed from B_3. This blockade is depicted in Figure 5.7d.

5.3 R3PDE: 3-Region Partial Dead Code Elimination

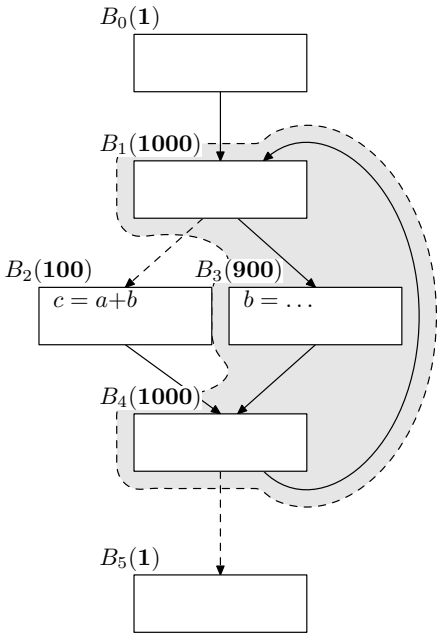
3-Region Partial Dead Code Elimination (R3PDE) is a reformulation of ISPDCE. It is designed to remedy the incorrectness of its naïve prototype, previously discussed.

It is by design, in the spirit of ISPRE, an approximate technique for performing code motion using information obtained from program profiles.

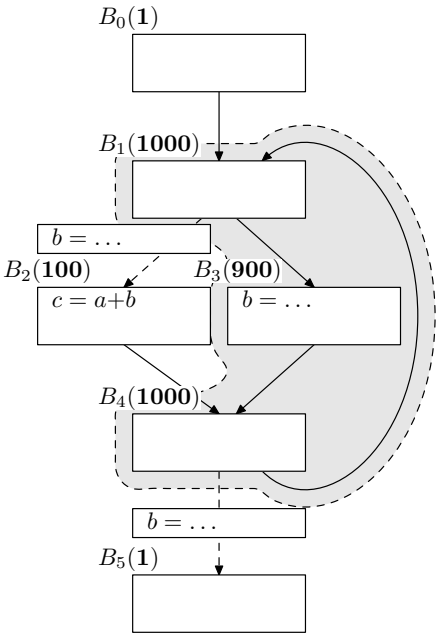
The major part of the performance gains obtained by R3PDE are made from one pass over the target program. Further improvements can be made with additional passes, although returns diminish. Consequently, we differentiate between *Single-Pass* R3PDE and *Multi-Pass* R3PDE, according to whether one or several transformation passes over the program are performed.

In the following, we describe single-pass R3PDE. Overall, our program analysis and transformation consist of the following steps, in order:

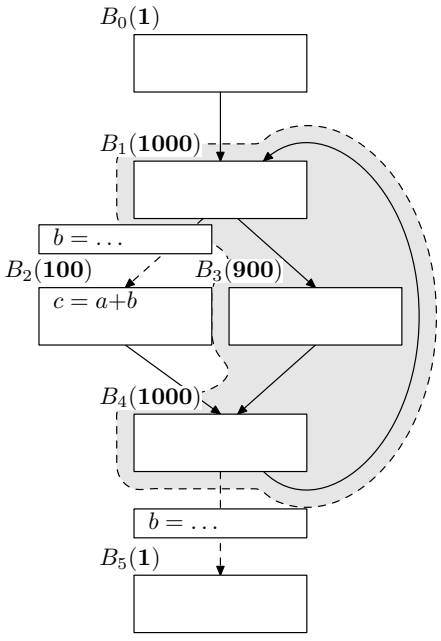
1. **hot-cold division** of the target CFG, performed in a manner identical to the ISPRE algorithm (Section 5.3.1);
2. **assignment sinking** which attempts to move assignments in the hot region onto edges which leave the hot region (Sections 5.3.2, 5.3.3, and 5.3.4);
3. **hot region versioning** which creates two copies of the hot region: the first copy is the "guard" region, the second copy is the "guarded" region—where optimization actually takes place (Section 5.3.5);
4. **linking** which creates control-flow edges which link the three regions together (Section 5.3.6);
5. **dead code removal** which removes assignments in the guarded copy of the hot region which have become dead as a result of *tentative* insertions of sunken assignments onto edges which leave the guarded copy of the hot region (Section 5.3.7); and
6. **egress insertions** which permanently insert, onto edges which leave the guarded copy of the hot region, *only* those tentatively inserted assignments that contributed to the elimination of one or more partially dead assignments from the guarded copy of the hot region (Section 5.3.8).



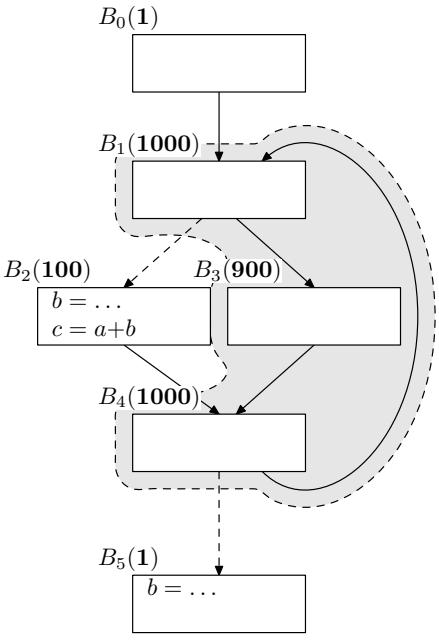
(a) Division of the CFG into hot and cold regions.



(b) Insertion of the partially dead computation $b = \ldots$ on edges which leave the hot region.

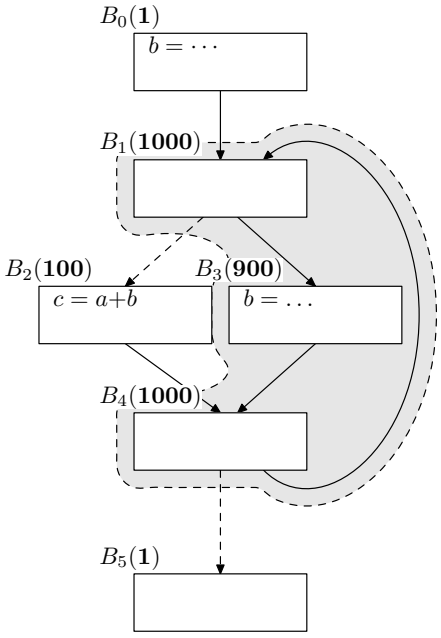


(c) Deletion of the *now* fully dead computation $b = \ldots$ from B_3 of the hot region.

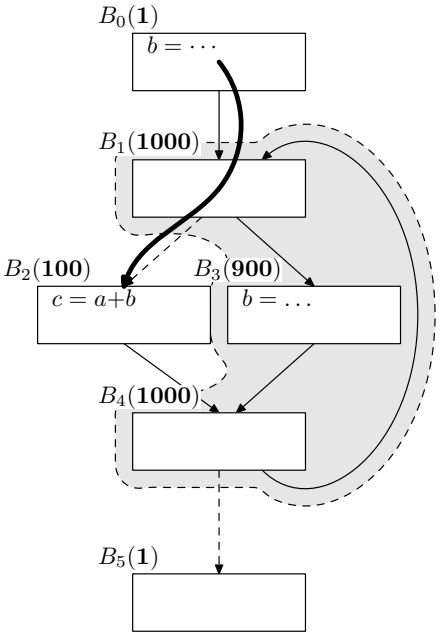


(d) Block-straightening applied to the CFG. The result: 800 partially dead computations removed.

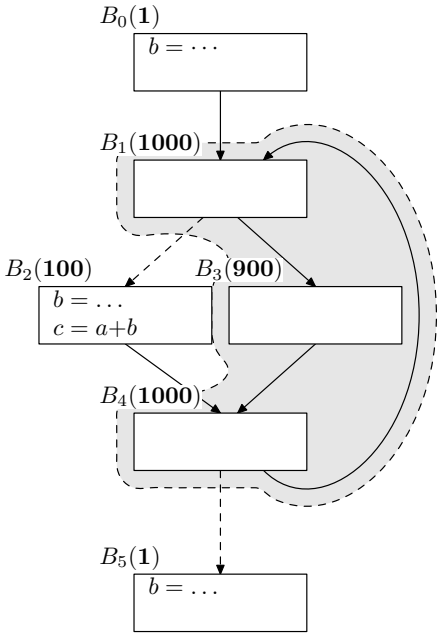
Figure 5.6: Application of ISPDCE to the motivating example.



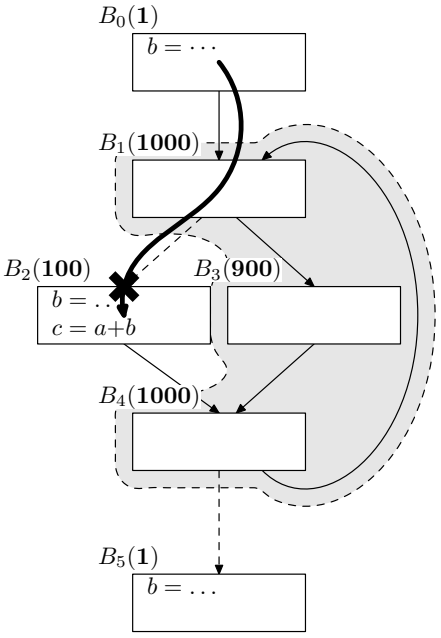
(a) The original unoptimized program.



(b) A use of b reached by the assignment in B_0, in the unoptimized program.



(c) The incorrectly optimized program.



(d) The blockade of the definition of b in B_0.

Figure 5.7: The incorrectness of ISPDCE revealed by its application to the motivating example.

5.3.1 Hot-Cold Division

R3PDE uses profile information to divide the CFG into two isothermal regions, in *exactly* the same manner as the algorithm ISPRE. Consequently, we refer the reader back to Section 4.6.1 for a detailed explanation, but re-state fundamental definitions here, for convenience:

Consider a CFG $G = (N, E, s, t)$ with

1. a frequency profile represented by the function

$$\text{freq} : N \cup E \to \mathbb{Z}$$

2. a threshold, Θ.

Then, we define

$$G_{hot} \equiv \langle N_H, E_H \rangle \tag{5.1a}$$
$$G_{cold} \equiv \langle N_C, E_C \rangle \tag{5.1b}$$

where

$$N_H \equiv \{u \mid u \in N \wedge \text{freq}(u) > \Theta\} \tag{5.2a}$$
$$E_H \equiv \{e \mid e \in E \wedge \text{freq}(e) > \Theta\} \tag{5.2b}$$
$$N_C \equiv N - N_H \tag{5.2c}$$
$$E_C \equiv E - E_H \tag{5.2d}$$

Given the division into hot and cold subgraphs, we define the *Egress* edges as

$$\mathsf{Egress} = \{(u, v) \mid u \in N_H \wedge v \in N_C\}$$

That is, the *Egress* set consists of precisely those edges which transfer control from a hot node to a cold node. They act as "exits" from the hot region. Furthermore, since every edge adjacent to a cold node must be cold, the following condition holds:

$$\mathsf{Egress} \subseteq E_C$$

Consequently, split egress edges can profitably act as landing pads for assignments which are moved out of the hot region. Indeed, R3PDE operates by inserting copies of assignments *containing computations* which occur in the hot region onto egress edges leaving the hot region, in order to make some partially dead assignments in the hot region become fully dead. Those dead assignments *and their computations* can be deleted, thereby achieving code motion from G_{hot} to G_{cold}.

As previously observed, since real program variables are modified by R3PDE (as opposed to temporary variables, in the case of ISPRE), assignments inserted by R3PDE can fail to preserve the behaviour of the original program unless extra conditions are enforced.

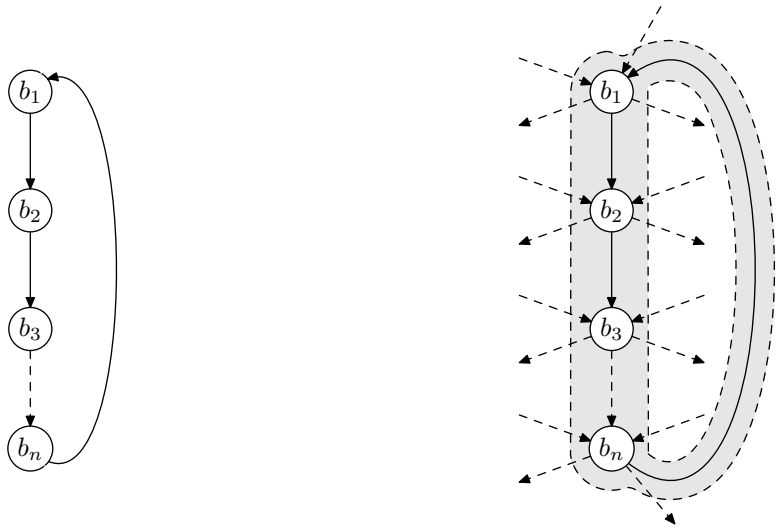
In our reformulation of R3PDE, we restrict the topology of the hot region so that it is possible to decide, for each egress edge exiting the hot region, whether a given candidate assignment (for dead code elimination) *always* reaches that egress edge. Recall that in Figure 5.7a, assignment $b = \cdots$

in B_3 *may* reach B_2 (only if the right-hand side of the loop executes), rendering the transformed program (Figure 5.7d) incorrect for executions where the right-hand side of the loop never executes.

The property of decidability just stated manifests itself in conjunction with the manner in which R3PDE restructures the CFG, which will be introduced shortly (in Sections 5.3.5 and 5.3.6).

The Topology of the Hot-Region

Our algorithm handles the case where the hot region has the following shape:



(a) The basic form: A loop of *hot* basic blocks and edges.

(b) The actual form: Edges to and from *cold* predecessors nodes are also allowed.

Figure 5.8: Topology of the hot region.

The hot region depicted above has the following important properties:

1. Each node b_i has exactly one *hot* successor, but may have arbitrarily many cold successors.

2. Each node b_i has exactly one *hot* predecessor, but may have arbitrarily many cold predecessors.

These restrictions, which might seem to exclude a large number of hot regions, actually exclude very few. Experiments on the SPEC JVM 98 benchmark suite show that 92% of the hot regions in the benchmark programs are recognized and optimized by R3PDE using this restricted region topology.

We define the following terms:

1. b_1 is called the entry node.

2. b_n is called the reiteration node.

Any cyclic ordering of $b_1 \ldots b_n$ suffices.

5.3.2 Assignment Sinking

R3PDE operates by attempting to move assignments which are computed in the hot region onto the egress edges. Since the egress edges are cold edges, doing so achieves code motion from a frequently executed region to an infrequently executed region, thereby improving program performance.

For the sinking of an assignment to be legal:

1. the right-hand side expression of the assignment must compute the same value on the egress edge as in its original location; and

2. the assignment must not overwrite the target variable with an out-of-date value.

On account of the above requirements, the candidate assignments amenable to movement by R3PDE are found by the following analysis:

immutability, which deduces assignments which can be sunk to the egress edges, while preserving their values.

This analysis falls within the monotone dataflow framework of Kam and Ullman [KU77]. This implies that it can be implemented as a unidirectional analysis using bit-vector representations of sets of assignments. That is, we can efficiently compute immutability for all candidate assignments simultaneously.

In the following exposition, the dataflow equations stated are described by two following major components:

basic block summaries, which state properties of each basic block. A basic block summary is derived for a given block *without* looking at any other basic blocks.

dataflow equations, which derive further properties of each basic block, by propagating properties of other basic blocks through the CFG.

We also describe the initialization method and solution method for each set of equations, for completeness.

5.3.3 Reached-Uses Analysis

The reached-uses analysis is a "safety-net" which prevents the algorithm from focusing its effort on assignments whose target variables are used in the hot region.

It derives, for each assignment of a variable, the set of uses which that assignment can reach. If one or more uses occur in the hot region, the assignment to that variable is not considered a candidate for removal.

Basic Block Summaries

First, the following sets of "used" variables are computed for each basic block, b, by examining the IR in the block.

$\forall b \in N$:

$$\mathsf{U\text{-}GEN}(b) \equiv \{(s,x) \mid \text{statement } s \text{ in } b \text{ uses variable } x \text{ and is not preceded by any redefinition of } x\ \} \tag{5.3a}$$

$$\mathsf{U\text{-}KILL}(b) \equiv \{(s,x) \mid \text{statement } s \text{ does not occur in } b \text{ and } s \text{ uses variable } x \text{ and } x \text{ is defined in } b\ \} \tag{5.3b}$$

Definition 5.3a defines the set of variables used (and their locations of use) in a basic block, such that assignments made to those variables at the entry to the block would reach those uses. Definition 5.3b defines the set of variables (and their locations of use) in other blocks that would be unaffected by assignments made to them at the entry of the block b, since block b itself contains assignments to those variables.

Dataflow Equations

The following equations for "reached" uses are solved simultaneously for the largest solution:

$\forall b \in N$:

$$\mathsf{RU\text{-}IN}(b) = (\mathsf{RU\text{-}OUT}(b) - \mathsf{U\text{-}KILL}(b)) \cup \mathsf{U\text{-}GEN}(b) \tag{5.4a}$$

$$\mathsf{RU\text{-}OUT}(b) = \bigcup_{s \in succs(b)} \mathsf{RU\text{-}IN}(s) \tag{5.4b}$$

Equation 5.4a states that an assignment made to a variable at the beginning of a block can reach a use of that variable, if either:

1. an assignment made to the variable at the end of the block could reach that use of the variable *and* there is no preceding definition of the variable in the block; or
2. it is used in the block without prior definition.

Equation 5.4b states that an assignment made at the end of a block b to a variable can reach a use of that variable, if, for at least one successor block s, an assignment made to that variable at the beginning of block s could reach that use of the variable.

The result of the analysis is, for each block b, the set of uses which definitions in block b reach.

Initialization and Solution

The standard method for solving simultaneous sets of equations involves initializing the values of $\mathsf{RU\text{-}IN}(b)$ and $\mathsf{RU\text{-}OUT}(b)$ for all basic blocks b, and then iterating the above equations in the order shown for all basic blocks, *until the sets converge*. For such an "iterate-to-convergence" algorithm, the appropriate initialization is:

$$\mathsf{RU\text{-}OUT}(t) = \emptyset \tag{5.5a}$$

$\forall b \in N$:

$$\mathsf{RU\text{-}IN}(b) = \mathsf{U\text{-}GEN}(b) \tag{5.5b}$$

The initialization in Equation 5.5a is performed since there are no uses of any variable to be reached from the end of the terminal block t of the CFG. This is true because our analysis is intra-procedural,

not inter-procedural. Equation 5.5b provides the set of uses reached locally within a block as the initial approximation to its RU-IN set. This is the standard initialization; it is provable that at convergence all those non-local uses which are reachable from the beginning of the block will eventually be added to its RU-IN set[KU77].

5.3.4 Immutability Analysis

The purpose of the immutability analysis is to determine, for each egress edge:

> the (unique) assignment in the hot region which *may* reach that egress edge.

Note that:

1. Two separate occurrences of the assignment $a = b \oplus c$ are considered *different* assignments.
2. Assignments of the form $a = a \oplus b$ and $a = b \oplus a$ (so-called *faint* assignments) are not considered for dead-code elimination. Their elimination requires the solution of non-monotonic dataflow equations, as demonstrated in [KRS94b].

Basic Block Summaries

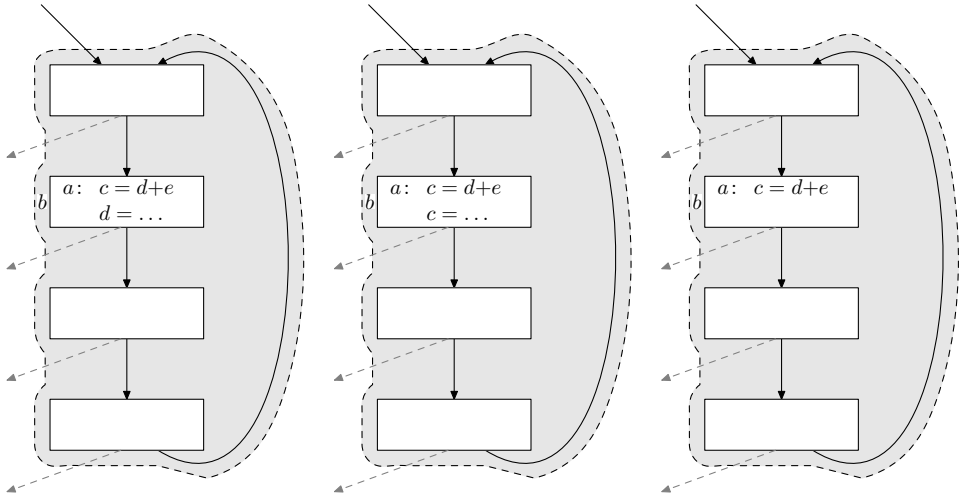
$$\begin{aligned}
&\forall b \in N_H\text{:}\\
&\quad \mathsf{COMP}(b) \equiv \{a \mid \text{block } b \text{ contains assignment } a\\
&\qquad\qquad \text{with no subsequent redefinition of its operands} \qquad (5.6a)\\
&\qquad\qquad \text{and no subsequent assignment to left-hand side variable of } a\ \}\\
&\quad \mathsf{KILL}(b) \equiv \{a \mid \text{block } b \text{ contains an assignment } a_1 \text{ to } c\\
&\qquad\qquad \text{and } c \text{ is a right-hand side operand of assignment } a\\
&\qquad\qquad\qquad \text{or the left-hand side variable of } a \qquad (5.6b)\\
&\qquad\qquad \text{and } a \neq a_1\}
\end{aligned}$$

Equations 5.6a and 5.6b are explained graphically in Figure 5.9 The condition $a \neq a_1$ ensures that two separate occurrences of an assignment (which by the note above are considered different assignments), will kill each other but *not also* themselves.

Dataflow Equations

The first of the following dataflow equations lacks a confluence operator ($\bigcup$ or $\bigcap$). However, this is reasonable since each node in the hot region has exactly one hot predecessor.

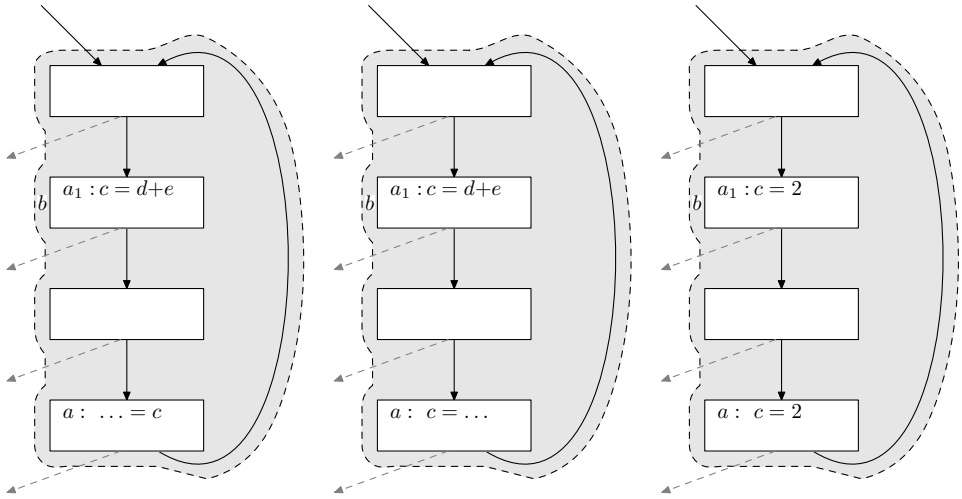
$$\begin{aligned}
&\forall b \in N_H\text{:}\\
&\qquad \mathsf{IM\text{-}IN}(b) = \mathsf{IM\text{-}OUT}(p),\ p \text{ is the unique predecessor of } b \text{ in } N_H \qquad (5.7a)\\
&\quad \mathsf{IM\text{-}OUT}(b) = (\mathsf{IM\text{-}IN}(b) - \mathsf{KILL}(b)) \cup \mathsf{COMP}(b) \qquad (5.7b)
\end{aligned}$$



(a) $a \notin \mathsf{COMP}(b)$ because of subsequent redefinition of operand d.

(b) $a \notin \mathsf{COMP}(b)$ because of subsequent redefinition of target variable c.

(c) $a \in \mathsf{COMP}(b)$.



(d) $a \in \mathsf{KILL}(b)$ because a_1 redefines the operand that assignment a uses.

(e) $a \in \mathsf{KILL}(b)$ because a_1 redefines the target variable of assignment a.

(f) $a \in \mathsf{KILL}(b)$ because b redefines the target variable of assignment a, *and* $a \neq a_1$ although they are syntactically identical.

The local properties COMP and KILL are used by the immutability analysis which is performed in the hot region (shown shaded).

Figure 5.9: The local properties COMP and KILL .

Initialization

$\forall b \in N_H$:

$$\mathsf{IM\text{-}OUT}(b) = \mathsf{COMP}(b) \tag{5.8a}$$

Note: The cold predecessors of nodes in the hot region are *not considered* in either the dataflow equations or their initialization.

The reason why it is not necessary to consider them is because the linking phase of the algorithm, to be described shortly, creates a copy of the hot region (the *guarded* region) that is entered only after an entire iteration of another copy of the hot region (the *guard* region) is executed. Under these circumstances the node b_i in the *guarded* copy of hot region does not have an external predecessor—*only* the hot predecessor $b_{i-1 \pmod n}$. Optimization takes place in the guarded copy of the hot region only.

Solution

We iterate Equations 5.7a and 5.7b in the order shown.

Note: The iterate-to-convergence process for this analysis is very efficient as convergence occurs after a single iteration. This is a consequence of the fact that the number of iterations required is bounded by the maximum number of backward edges along any acyclic path. Since there is only one backward edge, only one iteration is required.

5.3.5 Hot Region Versioning

Consider the hot region $G_H = \langle N_H, E_H \rangle$ as previously defined.

Our algorithm duplicates the hot region twice—to create a *guard* hot region and a *guarded* hot region. Consequently, we define two duplication operators: $'$ and $''$.

The Duplication Operator $'$

The duplication operation ($'$), which creates the guard region, is defined as follows:

1. For a node $n \in N_H$, the duplicate node n' is produced.
2. For an edge $e \in E_H$ defined
   $$m \rightarrow n$$
   the duplicate edge
   $$m' \rightarrow n'$$
   is produced.
3. For an edge $e \in \mathsf{Egress}$ defined
   $$m \rightarrow n$$
   the duplicate edge
   $$m' \rightarrow n$$
   is produced.

4. No edge $e \in (N_C \times N_H)$ is duplicated.

Consider, therefore, the guard copy of the hot region, G'_H defined:

$$G'_H = \langle N'_H, E'_H \cup \mathsf{Egress}' \rangle \tag{5.9}$$

to be an exact duplicate (i.e., version) of the hot region.

It is important to note that at the end of the duplication ($'$) operation:

1. G is reachable from the G', via the duplicated egress edges.
2. However, G' is not *yet* reachable from G.

The Duplication Operator $''$

The duplication operation ($''$) is defined analogously (replace $'$ with $''$ in the above) and yields a second copy of the hot region (i.e., the guarded hot region region) defined:

$$G''_H = \langle N''_H, E''_H \cup \mathsf{Egress}'' \rangle \tag{5.10}$$

It is important to note that at the end of the second duplication ($''$) operation:

1. G is reachable from the G'', via the duplicated egress edges.
2. However, G'' is reachable from *neither* G *nor* G'.

5.3.6 Linking

It remains, therefore, to:

1. Link G to G', so that control can enter the guard region; and
2. Link G' to G'', so that control can enter the guarded region from the guard region.

Note: We do not link G directly to G''—this would bypass the guard region, thereby invalidating the results of the immutability analysis. Nor do we link G'' to G'—it would allow the guarded region to be re-entered without being preceded by an entire iteration of the guard region.

To link:

1. G to G': Consider the node reiteration node b_n. Modify the edge

   $$b_n \rightarrow b_1$$

   to become the edge

   $$b_n \rightarrow b'_1$$

2. G' to G'': Consider the duplicated reiteration node b'_n. Modify the edge

   $$b'_n \rightarrow b'_1$$

to become the edge

$$b'_n \rightarrow b''_1$$

A graphical depiction of the steps of versioning and linking is shown in Figure 5.10.

No more transformations to the structure of CFG are required.

5.3.7 Deletions

The following analysis is used to determine the variables in the hot region which are not live, assuming insertions of sinkable assignments on egress edges.

Thereupon, any assignment a in the hot region, whose target variable is not live (assuming the tentative assignments on the egress edges) is deleted and we construct the set:

$$\mathsf{REMOVABLE} = \{a \mid \text{target variable of assignment } a \text{ is not live } \} \tag{5.11a}$$

Block Properties

$\forall b \in N_H$:

$$\begin{aligned} \mathsf{X\text{-}VARS}(b) \equiv \{v \mid v \text{ is a variable used in block } b \\ \text{without prior definition } \} \end{aligned} \tag{5.12a}$$

$$\mathsf{D\text{-}VARS}(b) \equiv \{v \mid v \text{ is a variable defined in block } b \} \tag{5.12b}$$

DataFlow Equations

$\forall b \in N_H$:

$$\mathsf{LIVE\text{-}IN}(b) = (\mathsf{LIVE\text{-}OUT}(b) - \mathsf{D\text{-}VARS}(b)) \cup \mathsf{X\text{-}VARS}(b) \tag{5.13a}$$

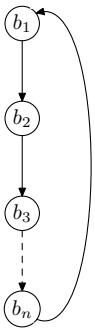
$$\mathsf{LIVE\text{-}OUT}(b) = \bigcup_{s \in succs(b)} \begin{cases} \mathsf{LIVE\text{-}IN}(s), & \text{if } s \in N_H; \\ \{v\}, & v = a \oplus b \notin \mathsf{IM\text{-}OUT}(b). \end{cases} \tag{5.13b}$$

Equation 5.13a states that a variable is live at the beginning of a block, if either:

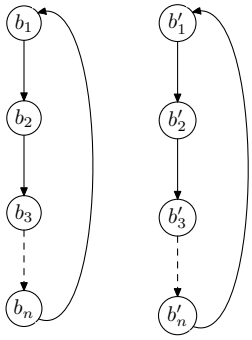
1. it was live at the end of the block *and* is not defined in the block; or
2. it is used in the block without prior definition.

Equation 5.13b states that a variable is live at the end of a block, if:

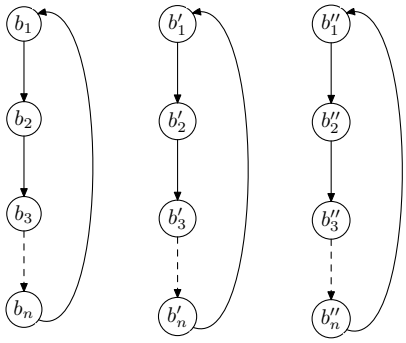
1. it is live at the entry to its (unique) hot successor. (It should be recalled that each block in the hot region has only one hot successor); or
2. it is impossible to move an assignment to that variable onto the egress edge which emanates from the block. We assume conservatively therefore, that the variable must be live along the path starting with the egress edge which leaves the block.



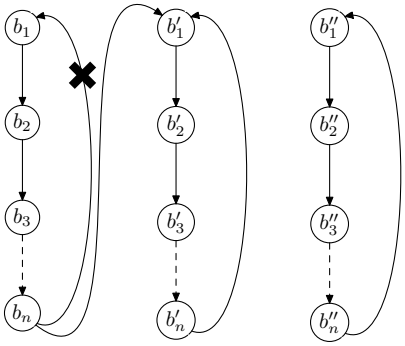
(a) The original hot region G_H.



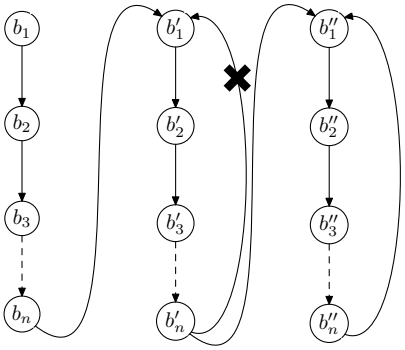
(b) Application of the duplication operator $'$ to create the guard region.



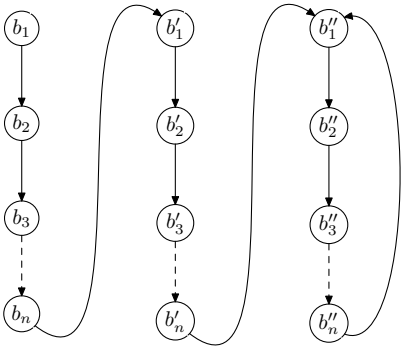
(c) Application of the duplication operator $''$ to create the guarded region.



(d) Rewriting edge $b_n \rightarrow b_1$ into $b_n \rightarrow b'_1$ thereby connecting the original hot region to the guard region.



(e) Rewriting edge $b'_n \rightarrow b'_1$ into $b'_n \rightarrow b''_1$, thereby connecting the guard region to the guarded region.



(f) The final result.

Figure 5.10: Illustrating the steps of versioning and linking.

Initialization

$\forall b \in N_H$:

$$\mathsf{LIVE\text{-}IN}(b) = \mathsf{X\text{-}VARS}(b) \tag{5.14a}$$

This is a conservative initialization for each LIVE-IN set.

Solution

As is the case with immutability analysis, only a single iteration of the "iterate-to-convergence" algorithm is required, since the hot region has only a single back edge.

5.3.8 Egress Insertions

Each assignment previously deleted in the hot region was removed under the assumption that one or more copies of that assignment could be moved to the egress edges.

Having taken advantage of that tentative assumption, we must now enforce that assumption, for program correctness:

```
for each assignment a ∈ REMOVABLE:
    for each block b in N''_H:
        if a ∈ IM-OUT(b) then
            for each egress edge e emanating from b:
                INSERT(e) = INSERT(e) ∪ {a}
```

This final step completes the algorithm.

5.4 R3PDE Algorithm Specification

We now present a summary of the R3PDE algorithm, which has been described in detail in the previous pages.

Please note that, in the remainder of this dissertation, we shall refer to this algorithm as R3PDE only when we want to emphasize the failure of the original prototype, which was based on ISPRE. Otherwise, we shall simply refer to this algorithm as ISPDCE, in deference to its original purpose.

Algorithm:	3-Region Partial Dead Code Elimination (R3PDE)
Input:	A CFG $G = (N, E, s, t)$, *Candidates*,
	frequency profile $\text{freq} : (N \cup E) \to \mathbb{Z}$, threshold $\Theta \in (0\%, 100\%)$
Output:	Deletion set $\mathsf{REMOVABLE} \subseteq$ *Candidates*
	Insertion sets $\mathsf{INSERT}(e)$, $e \in \mathsf{Egress}$

1. Hot Cold Categorization: Use Θ to divide the CFG into G_{cold} and G_{hot}:

$$N_H = \{u \mid u \in N \wedge \text{freq}(u) > \Theta\}$$
$$E_H = \{e \mid e \in E \wedge \text{freq}(e) > \Theta\}$$
$$N_C = N - N_H$$
$$E_C = E - E_H$$
$$G_{hot} = \langle N_H, E_H \rangle$$
$$G_{cold} = \langle N_C, E_C \rangle$$
$$\mathsf{Egress} = \{(u, v) \mid u \in N_H \wedge v \in N_C\}$$

2. Assignment Sinking I: Rejection of Candidates (analysis performed in G):

$$\mathsf{RU\text{-}IN}(b) = (\mathsf{RU\text{-}OUT}(b) - \mathsf{U\text{-}KILL}(b)) \cup \mathsf{U\text{-}GEN}(b)$$
$$\mathsf{RU\text{-}OUT}(b) = \bigcup_{s \in succs(b)} \mathsf{RU\text{-}IN}(s)$$

Remove from *Candidates* any assignment whose target variable is used in G_H.

3. Immutability Analysis (analysis performed in G_H):

$$\mathsf{IM\text{-}IN}(b) = \mathsf{IM\text{-}OUT}(p),\ p \in N_H$$
$$\mathsf{IM\text{-}OUT}(b) = (\mathsf{IM\text{-}IN}(b) - \mathsf{KILL}(b)) \cup \mathsf{COMP}(b)$$

4. Hot region versioning: Creation of guard region and guarded hot region.

$$\forall n \in N_H : n \mapsto n' \quad \forall e \in E_H : m \to n \mapsto m' \to n' \quad \forall e \in \mathsf{Egress} : m \to n \mapsto m' \to n$$
$$\forall n \in N_H : n \mapsto n'' \quad \forall e \in E_H : m \to n \mapsto m'' \to n'' \quad \forall e \in \mathsf{Egress} : m \to n \mapsto m'' \to n$$
$$G'_H = \langle N'_H, E'_H \cup \mathsf{Egress}' \rangle \qquad \mathsf{Egress}' = \{(x', y) | (x, y) \in \mathsf{Egress} \wedge x' \in N'_H\}$$
$$G''_H = \langle N''_H, E''_H \cup \mathsf{Egress}'' \rangle \qquad \mathsf{Egress}'' = \{(x'', y) | (x, y) \in \mathsf{Egress} \wedge x'' \in N''_H\}$$

5. Linking of G to G' and G' to G'':

$$(b_n \to b_1) \mapsto (b_n \to b'_1)$$
$$(b'_n \to b'_1) \mapsto (b'_n \to b''_1)$$

6. Deriving deletions (analysis performed in G_H):

$$\mathsf{LIVE\text{-}IN}(b) = (\mathsf{LIVE\text{-}OUT}(b) - \mathsf{D\text{-}VARS}(b)) \cup \mathsf{X\text{-}VARS}(b)$$
$$\mathsf{LIVE\text{-}OUT}(b) = \bigcup_{s \in succs(b)} \begin{cases} \mathsf{LIVE\text{-}IN}(b), & \text{if } s \in N_H; \\ \{v\}, & v = a \otimes b \notin \mathsf{IM\text{-}OUT}(b). \end{cases}$$
$$\mathsf{REMOVABLE} = \{a \mid \text{target variable of assignment } a \text{ is not live }\}$$

7. Deriving insertions (on Egress''): For each deletable assignment a, add a into $\mathsf{INSERT}(e)$ for each egress edge e that it can move to immutably.

5.5 An R3PDE Example

Figure 5.11 shows an example of R3PDE in action:

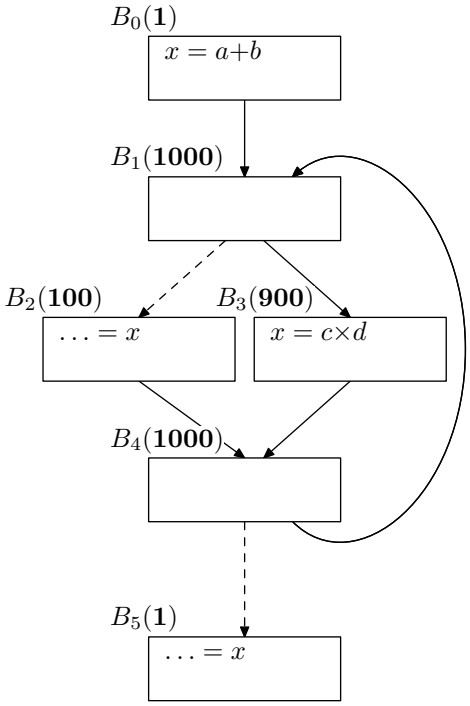
1. Figure 5.11a shows the CFG which naïve ISPDCE prototype failed on:
   (a) The definition of x in B_0 could never reach its use in B_2 in the optimized program.
   (b) After the optimized loop finished executing, the definition of x from the right-hand side always executed, even though the loop may have iterated only on its left side, which has no definition of x.

   This CFG is also shown as source code in Figures 5.11h and 5.11i (with explicit jumps).
2. Figure 5.11b shows the division of the CFG into hot and cold regions.
3. Figure 5.11c shows the duplication of the hot region into guard and guarded regions.
4. Figure 5.11d shows the egress edges connecting the two copies of the hot region back to the cold region. The empty basic blocks on the egress edges from the guarded region remind the reader that those egress edges can act as landing pads for assignments moved out of the guarded hot region.
5. Figure 5.11e shows that
   (a) Edge $B_4 \rightarrow B_1$ is changed to $B_4 \rightarrow B_{1'}$ so that control is transferred into the guard region when an attempt is made to iterate the loop.
   (b) Edge $B_{4'} \rightarrow B_{1'}$ is changed to $B_{4'} \rightarrow B_{1''}$ so that by the time control is transferred into the guarded region an entire iteration of the guard region has already been performed.
6. Figure 5.11f depicts the crucial phase. The assignment $x = c \times d$ in $B_{3''}$ is immutable, and therefore can be repeated on any Egress'' edge without:
   (a) computing a different value than it previously did (since there are no modifications of c or d after the original assignment); or
   (b) assigning a stale value to x (since there are no intervening assignments to x).

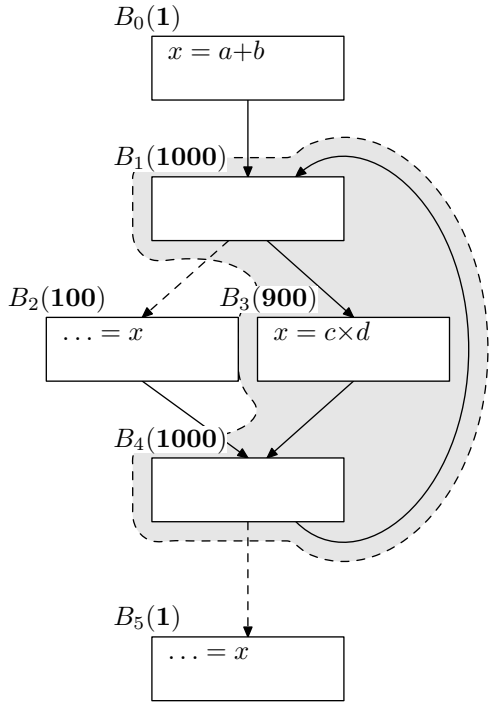
   Under this assumption, the value x in $x = c \times d$ is dead, and the entire assignment can be removed.
7. Figure 5.11g shows the insertion of the assumed assignments on the Egress'' edges, to ensure that the assumption made in the previous step is valid.

Notice in the finished CFG how:

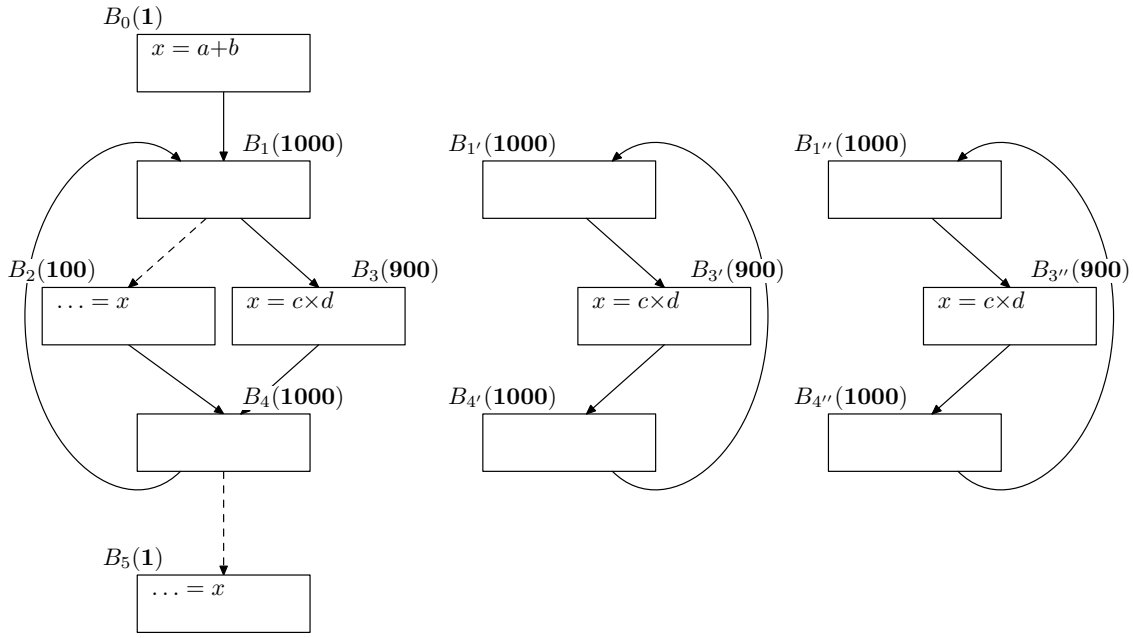
1. The definition of x in B_0 *does* reach its use in B_2 in the optimized program.
2. After the optimized loop finished executing, the definition of x from the right-hand side does not execute if the last iteration of the loop was through the left side.
3. **Most importantly, 900 dynamic computations of $x = c \times d$ have been deleted.**



(a) The motivating example. Note the initialization of x in B_0.

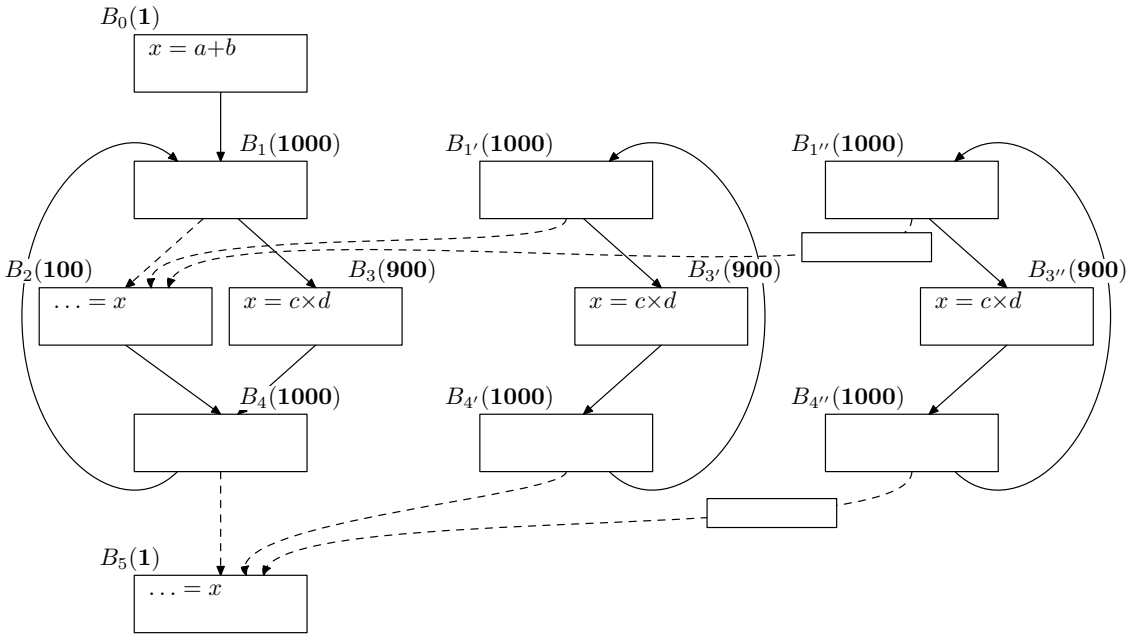


(b) The CFG divided into hold and cold regions, with $\Theta = 90\%$.

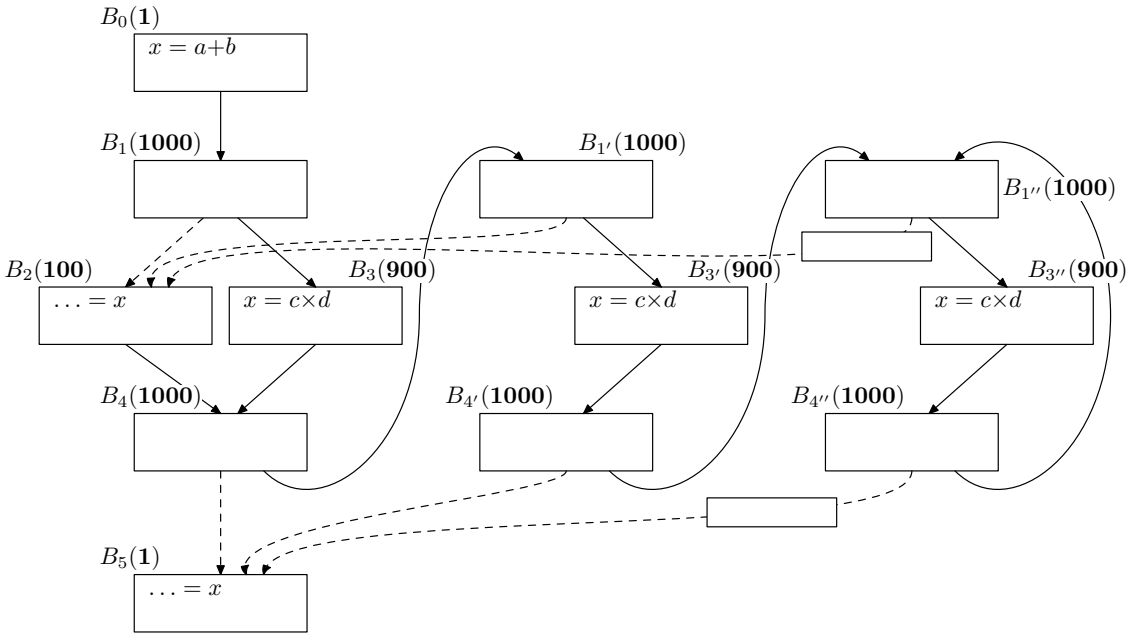


(c) Creation of the *guard* region and the *guarded* region.

Figure 5.11: Application of R3PDE to the motivating example.

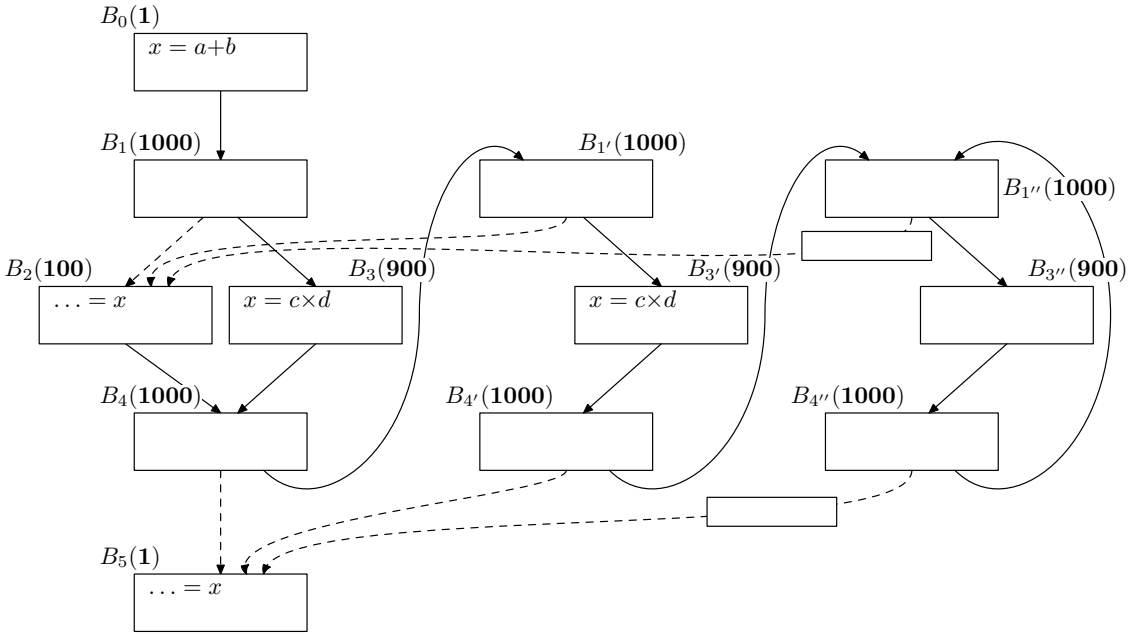


(d) The guard and guarded region showing egress edges linking them back to the cold region. Note the empty blocks on the egress edges from the guarded region—these will act as landing pads for sunken assignments.

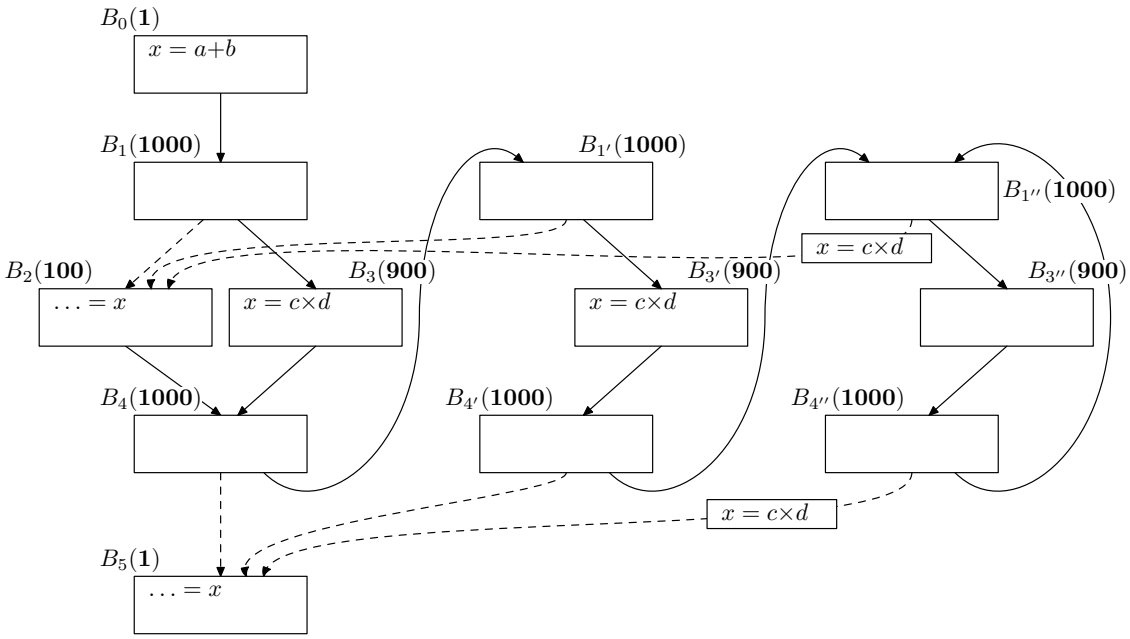


(e) Linking of the original program to the guard region, *and* the linking of the guard region to the guarded region.

Figure 5.11: Application of R3PDE to the motivating example (continued).



(f) The immutability analysis determines that $x = c{\times}d$ can be moved to all Egress'' edges without changing program meaning. Under this assumption, the sole occurrence of $x = c{\times}d$ in $B_{3'}$ is no longer live.



(g) The deletion of $x = c{\times}d$ from $B_{3'}$ requires the insertions of $x = c{\times}d$ on the Egress'' edges.

Figure 5.11: Application of R3PDE to the motivating example (continued).

```
1  x = a+b
2  do {
3    if (infrequently_true_test) {
4      v1 = x
5    } else {
6      x = c*d       /* frequently executed */
7    }
8  } while (cond)
9  v2 = x
```

(h) Original program, as source code, corresponding to Figure 5.11a

```
1  x = a+b                                  /* B0 */
2  label original_loop_start:
3    if (infrequently_true_test)            /* B1 */
4      goto left_side_of_loop:
5    label right_side_of_loop:
6      x = c*d                              /* B3 */
7      goto reiteration_check;
8    label left_side_of_loop:
9      v1 = x                               /* B2 */
10   label reiteration_check:
11     if (cond)
12       goto loop_original_start;          /* B4 */
13
14 label first_block_after_loop:
15 v2 = x                                   /* B5 */
16 exit();
```

(i) The original program (as above), but with loops converted into explicit jumps.

```
1  x = a+b                                  /* B0 */
2  label original_loop_start:
3    if (infrequently_true_test)            /* B1 */
4      goto left_side_of_loop:
5    label right_side_of_loop:
6      x = c*d                              /* B3 */
7      goto reiteration_check;
8    label left_side_of_loop:
9      v1 = x                               /* B2 */
10   label reiteration_check:
11     if (cond)
12       goto guard_region;                 /* B4 */
13
14 label first_block_after_loop:
15 v2 = x                                   /* B5 */
16 exit();
17
18 label guard_region:
19   if (infrequently_true_test)            /* B1' */
20     goto left_side_of_loop:
21   x = c*d                                /* B3' */
22   if (cond)
23     goto guarded_region;                 /* B4' */
24   goto first_block_after_loop;
25
26 label guarded_region:
27   if (infrequently_true_test) {          /* B1'' */
28     x = c*d                              /* B1''→B2 */
29     goto left_side_of_loop:
30   }
31   ~~x = c*d~~                            /* B3'' */
32   if (cond)
33     goto guarded_region;                 /* B4'' */
34
35   x = c*d                                /* B4''→B5 */
36   goto first_block_after_loop;
```

(j) The optimized program, as source code, corresponding to Figure 5.11g

Original and optimized programs shown as source code.

Figure 5.11: Application of R3PDE to the motivating example (continued).

5.6 Proof of Correctness

In the following proof, we assume for simplicity that blocks are trivial—there is one statement per block. Then, each statement has a one-to-one correspondence with its block number. More precisely, each assignment gets a unique number i, the number of its block b_i.

Note: This does not invalidate the formulation of the R3PDE algorithm since any basic block can be decomposed into a sequence of trivial basic blocks each containing exactly 1 IR statement from that block (in the order the IR statements originally occurred), without changing program meaning.

Further, [KKS98] provides experimental data to show that CFGs composed of trivial blocks can be analyzed and transformed as efficiently as (and sometimes even faster than) CFGs composed of basic blocks. Hence, in addition to the use of trivial blocks in the proof, it is feasible for an implementation of R3PDE to use trivial blocks, if desired.

Our proof of correctness involves two steps:

1. Proof that insertion of assignments on egress edges does not change the meaning of the program.

2. Proof that deletion of assignments from the hot region *in the presence of inserted assignments on egress edges* does not change the meaning of the program.

5.6.1 Correctness of Assignment Insertions

Definition 1 (Path)**.**

1. *A* path *is a string of alternating nodes (n) and edges (e).*

2. *Paths may start (end) with either type of element.*

3. *A path has at least one element.*

Definition 2 (Complete Path through a CFG)**.** *A path representing a complete execution of a* CFG *has the form:*

$$n_0{=}s, e_1, n_2, e_3, \ldots, n_{l-2}, e_{l-1}, n_l{=}t$$

where s is the start node of the CFG*, and t is the terminal node.*

Definition 3 (Sub-path)**.**

1. ***unoptimizable path**, which consists of nodes (both cold and hot) and edges (both cold and hot, except the reiteration edge). This path starts with a cold node.*

2. ***almost-optimizable path**, which consists of*

   - *an unoptimizable path (as described above) ending at the reiteration node b_n; followed by*
   - *the reiteration edge ($b_n{\rightarrow}b_1$); followed by*
   - *a path of the form $b_1, \ldots, b_k, e$ whose nodes and edges are only those of the hot region with the sole exception of e, an egress edge (cold). No b_i is ever repeated.*

Intuitively, this path (after the unoptimizable path executes) starts at the top of the hot region at b_1 but stops in the hot region and exits via egress edge e, without ever taking the edge $b_n \to b_1$.

In notation this path is:

$$\langle \textit{unoptimizable path} \rangle, (b_n \to b_1), b_1, \ldots, b_k, e$$

3. ***optimizable path**:*

   - *an unoptimizable path (as described above) ending at the reiteration node b_n; followed by*
   - *the reiteration edge ($b_n \to b_1$); followed by*
   - *a path of the form $b_1, \ldots, b_n$ whose nodes and edges are only those of the hot region. No b_i is ever repeated; followed by*
   - *the reiteration edge ($b_n \to b_1$); followed by*
   - *a path of the form $(b_1, \ldots, b_n, b_n \to b_1)^*$ whose nodes and edges are only those of the hot region (where $*$ mean zero or more repetitions); followed by*
   - *a path of the form $b_1, \ldots, b_k$ whose nodes and edges are only those of the hot region. No b_i is ever repeated; followed by*
   - *an egress edge e.*

   In notation this path is:

$$\langle \textit{unoptimizable path} \rangle, (b_n \to b_1), b_1, \ldots, b_n, (b_n \to b_1), (b_1, \ldots, b_n, b_n \to b_1)^*, b_1 \ldots b_k, e$$

   Intuitively, this path (after the unoptimizable path executes) represents at least one iteration of the hot region.

Determining subpaths

To extract the subpaths from a complete path through a CFG, start with first element and construct the longest type of subpath constructible. Then, repeat the process starting with the first element that has not yet been assigned to a subpath, until no such elements remain.

Mapping subpaths from the original program to subpaths in the optimized program

We now describe how each type of subpath is mapped into a subpath in the optimized program by the R3PDE algorithm:

1. **unoptimizable path**: remain unchanged.
2. **almost-optimizable path**: These paths have the form (where c is a cold node):

$$\langle \text{unoptimizable path} \rangle, (b_n \to b_1), b_1, \ldots, b_k, e = (b_k \to c)$$

   and these paths are mapped to the new form

$$\langle \text{unoptimizable path} \rangle, (b_n \to b'_1), b'_1, \ldots, b'_k, e = (b'_k \to c)$$

Intuitively, this corresponds to the linking operation which makes the reiteration edge enter the guard region.

3. **optimizable path**: These paths have the form (where c is a cold node):

   $\langle$unoptimizable path$\rangle, (b_n{\rightarrow}b_1), b_1, \ldots, b_n, (b_n{\rightarrow}b_1), (b_1, \ldots, b_n, b_n \rightarrow b_1)^*, b_1, \ldots, b_k, e = (b_k{\rightarrow}c)$

   and these paths are mapped to the new form

   $\langle$unoptimizable path$\rangle, (b_n{\rightarrow}b'_1), b'_1, \ldots, b'_n, (b'_n{\rightarrow}b''_1), (b''_1, \ldots, b''_n, b''_n \rightarrow b''_1)^*, b''_1, \ldots, b''_k, e = (b''_k{\rightarrow}c)$

   Intuitively, this corresponds to the linking operation which makes the original reiteration edge enter the guard region, and (if a full iteration of the guard region is performed thereupon) to enter the guarded region. Then control flow can iterate zero or more times in the guarded hot region, finally exiting via an egress edge.

Proving program correctness

Definition 4 (Correctness)**.** *We define a (transformed) path P' as correct relative to the (original) path P, iff at the end of each element on the path P', each variable has the same value as at the end of the corresponding element on path P.*

We may now prove program equivalence by structural induction over subpaths:

Theorem 1 (Correctness of Insertions on Egress Edges)**.** *Consider the original* CFG *and a path through it:*

$$P = S_1 S_2 \ldots S_n. \tag{5.15}$$

where S_i is a subpath. The corresponding transformed path in the optimized CFG *is:*

$$P' = S'_1 S'_2 \ldots S'_n. \tag{5.16}$$

If correctness is maintained upto the start of path S'_n that it is maintained upto the start of path S'_{n+1}.

Proof. **(Base Case)** S_0 is a cold path. Hence $S'_0 = S_0$ and correctness is maintained until the start of S'_1.
(Induction Step) Consider path S_i where $i > 0$:

1. **unoptimized path**: unoptimized paths remain unchanged. Hence $S'_i = S_i$ and correctness is maintained until S'_{i+1}.

2. **almost optimizable path**: almost-unoptimized paths remain unchanged. Hence $S'_i = S_i$ and correctness is maintained until S'_{i+1}.

3. **optimizable path**: These paths have the form (where c is a cold node):

   $\langle$unoptimizable path$\rangle, (b_n{\rightarrow}b_1), b_1, \ldots, b_n, (b_n{\rightarrow}b_1), (b_1, \ldots, b_n, b_n{\rightarrow}b_1)^*, b_1, \ldots, b_k, e = (b_k{\rightarrow}c)$

which is mapped to

$$\langle \text{unoptimizable path} \rangle, (b_n \rightarrow b'_1), b'_1, \ldots, b'_n, (b'_n \rightarrow b''_1), (b''_1, \ldots, b''_n, b''_n \rightarrow b''_1)^*, b''_1, \ldots, b''_k, e = (b''_k \rightarrow c)$$

We have program correctness up to the end of:

$$\langle \text{unoptimizable path} \rangle, (b_n \rightarrow b'_1), b'_1, \ldots, b'_n, (b'_n \rightarrow b''_1), (b''_1, \ldots, b''_n, b''_n \rightarrow b''_1)^*, b''_1, \ldots, b''_k$$

However, we still need to prove correctness past the final element on the subpath, namely

$$b''_k \rightarrow c$$

If this egress edges contains no insertions, we are done. Otherwise, it contains an inserted assignment $j \in \mathsf{IM\text{-}OUT}(b_k)$.

All that $j \in \mathsf{IM\text{-}OUT}(b_k)$ guarantees us is: *if* assignment j occurred previously, it will reach the end of b_k without modification to its operands or left-hand side variable.

We must make sure that assignment j did indeed occur.

We have three cases:

(a) $j = k$: In this degenerate case, an assignment has simply been pushed onto the egress edge of block which contains it. Assignment j definitely has occurred.

(b) $j < k$: Since b''_j occurs before b_k in $(b''_1, \ldots, b''_k)$ we know that assignment j definitely has occurred.

(c) $j > k$: b_j (alternatively assignment j) does *not* occur before b_k in $(b''_1, \ldots, b''_k)$ Nor does it necessarily occur in $(b''_1, \ldots, b''_n, b''_n \rightarrow b''_1)^*$ which may not execute even once. However, it is guaranteed to occur in $(b'_1, \ldots, b'_n, b'_n \rightarrow b''_1)$. Hence, assignment j does occur.

Hence, correctness is maintained until the end of $(b''_k \rightarrow c)$ and consequently upto the end of path S_{i+1}. □

5.6.2 Correctness of Assignment Deletions

Assignment deletion is driven by the results of a live-variables analysis which has been proven correct in [Hec77].

5.7 The Derivation of ISPDCE from PDCE

In Section 4.10, we demonstrated that the design of ISPRE leverages the design of non-speculative formulations of PRE. In this section, we continue the same line of argument, but with respect to ISPDCE.

PDCE algorithms basically involve a two-step process:

1. The movement of computations, which turn partially dead assignments into fully dead assignments; and

2. The detection and deletion of (the newly created) fully dead assignments.

Most PDCE algorithms, such as [KRS94b], use multiple bit-vector analyses to derive placement points for computations. The essence of these analyses is the computation of delayability for partially dead assignments—the extent to which an assignment can be pushed in the direction of flow of control while preserving program semantics. ISPDCE, likewise, must move assignments out of the hot region and onto the egress edges, and uses the immutability analysis, a bit-vector analysis, to ensure legality of the movements.

Thereupon, the detection and deletion of fully dead assignments is performed via a live variables analysis, in a manner characteristic of most PDCE formulations.

Thus, the original structure of PDCE algorithms is preserved in ISPDCE. Yet, we remind the reader that ISPDCE performs *speculative* PDCE. We repeat, once again, that we have thereby leveraged knowledge of the formulation of a non-speculative version of an optimization in the construction of a speculative version of that optimization.

The construction of ISPDCE (in addition to the construction of ISPRE) demonstrates that knowledge of the formulation of a non-speculative optimization need not be ignored, but can be used as a model from which a speculative version of the optimization can be constructed.

In doing so, we have demonstrated *claim* 3 with respect to ISPDCE.

Chapter 6

Previous Work on Speculative and Non-Speculative PRE and PDCE

In this chapter, we shall categorize and review the research that has been done to date in the areas of PRE and PDCE. Even though an exhaustive, comprehensive survey of the field is provided, it is important to note that, for the purposes of this dissertation, we are interested in one particular dichotomy—the one which distinguishes between speculative and non-speculative versions of PRE and PDCE.

6.1 Related PRE Research

6.1.1 Classical PRE

The most primitive form of PRE is Local Common Subexpression Elimination (LCSE)[ASU86]. It eliminates an expression from a basic block if that expression was computed earlier in the block and the values of its operands have not changed since then. An algorithm which does not require an exact match for a previous occurrence of an eliminatable expression is described in [Bre69]. It finds common subexpressions from a sequence of larger expressions. Consider the following:

$$\begin{aligned} E_1 &= a+b+c+e \\ E_2 &= a+b+e+f \\ E_3 &= a+b+d+e+f \end{aligned}$$

For this particular example, their algorithm stores $a+b$ in a temporary variable, so that it can be re-used in the computations of E_2 and E_3. Nevertheless, both [ASU86] and [Bre69] only work locally on basic blocks.

The first attempt at Global Common Subexpression Elimination (GCSE), via an available expressions analysis, is described in [Coc70]. The first efficient formulation of the available expressions analysis, however, was later presented in [Ull72] and [Ull73], which described an $O(n^2)$ bit-vector algorithm for unstructured flow graphs and an $O(m \log n)$ algorithm for structured flow graphs, where n is the number of basic blocks and m is the number of expressions under consideration. Finally, the

algorithm which could perform redundant expression elimination even in the absence of complete redundancy (i.e. PRE) was first formulated in [MR79] by Morel and Renvoise, who later extended it to the inter-procedural case in [MJ81].

It should be noted that the original formulations of PRE [MR79, MJ81] required bidirectional dataflow analyses, which are more complicated, less efficient, and which were less well-understood at the time, than standard unidirectional analyses. In [Dha88] and [DP93], Dhamdhere showed that using edge-placement (the insertion of computations on CFG edges) with bidirectional analyses results in an algorithm whose performance is competitive with one that uses unidirectional analyses. Overall, Dhamdhere's approach, however, became less important, as a consequence of unidirectional reformulations, which we discuss shortly.

Dhamdhere's approach, however, had one important lasting effect. It established the splitting of "critical" edges as a crucial preparation for CFGs. A critical edge is an edge whose source basic block has multiple successors and whose target basic block has multiple predecessors. Critical edges present a quandary to those PRE algorithms which restrict themselves to inserting code *only* in basic blocks: Any insertion of code in the source or target block, will also affect code paths that do *not* include the critical edge. The solution is to split the critical edge by creating a new basic block through which it passes. The new basic block can be used as a landing pad for inserted instructions, thereby affecting only those paths which include that edge. Consequently, the splitting of critical edges makes CFGs more amenable to block-based PRE algorithms.

An excellent retrospective of classical PRE appears in [Mor83]. Two informative accounts of its implementation appear in [Cho83] and [Lam00].

6.1.2 Important Reformulations of Classical PRE

In [Wol99], Wolfe showed that PRE could be reformulated using only unidirectional analyses. The most important unidirectional reformulation, "Lazy Code Motion (LCM)", is described in [KRS92], where the original PRE algorithm is extended to be optimal with respect to register pressure. Register pressure refers to the scarcity of registers faced by a compiler during the process of register allocation. PRE which holds computed values in temporary variables (which are often implemented as registers) exacerbates the problem of register pressure by leaving fewer registers to be assigned to other program entities. In fact, the original formulation of PRE holds values in registers over very long paths in the CFG. Such a path, over which the value of a computation is held in a temporary variable, is called a live range. The original PRE algorithm, now often called "Busy Code Motion", often maximizes live ranges for no increase in the number of redundant computations eliminated. The laziness in Lazy Code Motion, however, increases the live ranges (i.e. moves computation "higher" in the flow graph) *only* when it will actually facilitate one or more redundancy eliminations. Knoop, Rüthing, and Steffen describe the theory behind LCM in [KRS94a].

[DS93] reformulated LCM to be used for general basic blocks since the original LCM formulation assumed that each basic block contained exactly one instruction. The inventors of LCM subsequently added another optimality criterion to their PRE algorithm—code size—and described their approach in [RKS00a] and [RKS00b]. Their approach is not LCM specific; it can be retrofitted to the original PRE algorithm[MR79]. Gupta and Bodik generalized LCM in [GB99], by distinguishing between the frequency with which various block were expected to execute; less frequently executed blocks were allowed to have high register pressures since worse register allocation in these blocks would not

have a deleterious effect owing to their infrequent execution. They further reduce register-pressure by using the heuristic of accessing original program variables (containing the required expression values) where possible, instead of the temporary variables introduced by LCM.

The next reformulation, [SMH03a] and [KCJ00], involves the use of predication to achieve redundancy elimination. The essence of this approach is the instruction format:

$$p \ ? \ (a \leftarrow b \oplus c)$$

where the operation $\oplus$ and its assignment to a are performed only if the predicate variable p is true. However, predicated PRE can incur performance overheads and thus needs a cost-benefit model to decide, for each partially redundant expression instance, whether the overheads of predication are acceptable[SMH03b].

The algorithms described thus far preserve the structure of the CFG; that is, they merely perform code motion within a CFG. A fourth major reformulation—one which mutates the CFG—is due to Bodik[BGS98]. The necessity of CFG restructuring to effect complete PRE is indicated quite dramatically, wherein it is demonstrated that three-quarters of loop-invariant statements cannot be hoisted from loops by code motion alone, but are amenable to hoisting through CFG restructuring. [BGS98] also provides an algorithm to confine, within acceptable limits, the resulting CFG's increased size, an unfortunate consequence of restructuring. A commensurately aggressive algorithm is presented in [KRS99] which is liable to the same dramatic increase in program size so that the authors advocate using it only on "considerably small computational hot-spots." All PRE algorithms thus far described move computations, not assignments. The moved computation is then made part of an assignment to a temporary, and that temporary is substituted as the right-hand side of the original assignment.

A fifth, very different approach—assignment motion—was advocated in [KRS95]. Assignment motion moves the entire assignment, which is advantageous because it allows upward movement of expressions that are dependent on the outputs of redundancy-eliminated assignments. As an example, consider the statement sequence:

$$\begin{aligned} a &= b + c \\ d &= a + e \end{aligned}$$

If *expression*-motioning PRE moves $b + c$ earlier, caching its value in t_0 to obtain:

$$\begin{aligned} a &= t_0 \\ d &= a + e \end{aligned}$$

it would not subsequently be able to move $a + e$ earlier because of the assignment to a. However, if the entire expression $a = b + c$ had been moved, $a + e$ would also be able to move upwards. Dhamdhere provides a similar, but suboptimal, algorithm in [Dha91].

6.1.3 Speculative PRE

We now describe the formulations of PRE that are at the very heart of this research proposal: Speculative PRE.

Sub-Optimal Solutions

An early paper in the area of SPRE is [GBF98]. It commences with a very interesting program metric: 65% of the program methods in the SPEC CPU 95 benchmark suite have at most five paths with non-zero execution frequency. This observation is used to justify the individual examination of each executed path through a method. For each path under consideration, the cost and benefit of eliminating redundancies along that path alone is computed. These metrics are drawn from execution profiles obtained from Profile Driven Feedback (PDF). The cost of a computation in a block is the number of times the block performs that computation; its removal results in a benefit of the same magnitude. The cost of a path is the sum of its individual costs; the benefit of a redundancy-eliminated-path is the sum of its individual benefits.

In order to remove a redundancy along a path, an instance of the expression under consideration must be inserted elsewhere (i.e. another path) to turn the partial redundancy into a full redundancy. Hence, the benefit from a redundancy elimination on a path induces a cost to another path. Two cost-benefit dataflow analyses are used to decide the appropriate deletion and insertion points for the computations. This method of SPRE unfortunately suffers from the flaw that there exist methods over which a hundred or more paths are taken with non-zero probability, rendering the analysis prohibitively expensive in that case. Moreover, though conceptually simple, the algorithm is, in its implementation, quite complicated.

A similar but simpler alternative is provided in [HH97]. It also optimizes the program to reduce the dynamic number of instructions executed via a pair of analysis. The first analysis determines the costs that would be incurred to delete various instructions. The second analysis computes the benefits that would be achieved by deleting various instructions. The product of this pair of analyses are two sets which tell where to delete instructions and where to insert them in order to minimize the number of dynamic computations. This algorithm does not approach the problem of speculative optimization on a path-by-path basis but handles the entire CFG *en masse*. A most unfortunate artefact of this method is a lack of proof of termination; indeed, a problem instance was found on which the algorithm fails to terminate.

6.1.4 Optimal Speculative PRE

Thus far, all proposed solutions to the problem of SPRE have been suboptimal.

The first optimal solution with respect to execution counts was presented in [CX03]. It reduces the problem of finding an optimal set of insertions and deletions to the problem of cutting a flow network through which maximum flow has been pushed. Their approach, while optimal, suffers from four important setbacks:

1. the reduction from CFG to flow network is not an intuitive one;

2. two bit-vector analyses are required to reduce the CFG before transforming it into a flow network;

3. the best maximum-flow solvers work in high-order polynomial times;

4. the algorithm can only process one expression at a time, resulting in a prohibitively expensive solution for all expressions in a control flow graph.

A similar paper, [SHK04], ameliorates the conceptual difficulty of reduction from flow graph to flow network by reducing the problem of SPRE to an optimization problem named Stone's problem[Sto77] in a very intuitive manner. Thereupon, Stone's problem is reduced to a flow network and solved. This two-step presentation of SPRE improves on the previous one by allowing dynamic instruction counts, code size, or even a combination of both metrics to be the target criterion for optimization. In fact, it can even be extended to use register pressure as an additional optimization criterion. These four papers are the very core of the research done in SPRE to date.

Unfortunately, they all present algorithms which suffer from the problem of being computationally exorbitant, and, in doing so, motivate the creation of an efficient, albeit suboptimal, approximation.

6.1.5 The Elegant Simplicity of ISPRE

Table 6.1 compares the leading algorithms which perform Speculative Partial Redundancy Elimination. It presents a summary of features which a compiler-writer would consult.

	[GBF98]	[HH97]	[SHK04]	[CX03]	ISPRE
Number of bit vector analyses	3	1	0	2	2
Number of network flow analyses	0	0	1	1	0
Correct	√	×	√	√	√
Optimal	×	×	√	√	×
Level of parallelism	Path	Full	Expression	Expression	Full

Table 6.1: Feature Matrix for Speculative PRE algorithms

It can be seen from this table that ISPRE is a very attractive choice. It uses a modest number of bit-vector analyses, undermined only by [HH97] which requires just one—but, which is, unfortunately, an incorrect algorithm since it is not guaranteed to terminate, as noted earlier. [GBF98] requires three bit-vector analyses and unfortunately optimizes the program one path at a time.

[SHK04] quite nicely requires no network flow analyses, but instead requires a flow network to be built and solved for *each* expression in the target program being optimized. Simple network flow algorithms, such as Ford-Fulkerson do not suffice; more efficient, and complicated, algorithms must be used, as discussed in Chapter 7. [CX03] is as onerous to implement as [SHK04], and additionally requires two bit-vector analyses to simplify flow networks prior to solving them. Nevertheless, a flow network must be built and solved for each expression.

ISPRE represents the most attractive algorithm to a compiler writer. It requires just two bit-vector analyses, no network flow analyses, and optimizes all expressions simultaneously. Although,

it is suboptimal (by design), Chapter 7 will argue that it gives results as good as optimal SPRE (as implemented by [SHK04] and [CX03]), on average.

Its simplicity lends itself to clean, bug-free implementation, making it a serious contender amongst other speculative PRE algorithms, if not the champion, for implementation in industrial strength compilers.

Thus, we have substantiated *claim* 4 with respect to ISPRE.

6.1.6 Further Reformulations of PRE

The remaining reformulations of PRE constitute two categories: a) reformulation in Static Single Assignment (SSA) form and b) simplifications of previous formulations. Both of these categories are important. The first allows PRE to be recast to use the increasingly popular SSA-based IRs. The second affords the widespread acceptance of PRE.

An important reformulation of PRE to use SSA is presented in [CCK+97] and [KCL+99]. Their algorithm, SSAPRE, is notoriously difficult to implement. Consequently, another SSA based algorithm, anticipation-based PRE [vDH04a] provides a much simpler solution and even eliminates more redundancies than SSAPRE, in fewer passes.

Simplifications to the original PRE algorithm have also been discovered. E-path-PRE, based on eliminatability paths, is described in [Dha02]. It is pedagogically important, since it provides an "intuitive feel" of the action of PRE on programs. Another simple PRE algorithm, which is computation- and lifetime-optimal, is developed and proved correct in [PSS03]. It is also of great pedagogical importance since it provides lifetime optimality, but, unlike the theory of LCM described in [KRS94a], does not require 40 pages of intricate proofs. Finally, [Bro04] describes and proves correct a computationally optimal formulation of PRE in exactly one page; the rest of the paper is dedicated to describing an enhancement performing load-hoisting and store-sinking.

6.1.7 Theoretical Results

Having discussed the various formulations of PRE, we shall now turn to the theoretical work that has been done in the field of PRE—work which did not result in new PRE algorithms *per se*, but which provide important and subtle theoretical results crucial to the construction of effective, real-world PRE implementations.

The original "classic" requirements which every code motion algorithm, regardless of formulation, had to obey were outlined by Kennedy in [Ken72]. Knoop, provides a modern update to this paper in [KR99]. In [AJU77], it is shown that the generation of optimal code for expressions containing common subexpressions is, in general, an NP-complete problem.

In [Sor96], it is shown that a careless formulation of equations for bidirectional dataflow analyses can slow down the rate of convergence of the analyses; corrections to the same are also presented. Ryder provided a rigorous mathematical framework for bidirectional dataflow analysis in [MMR95]. Dhamdhere shows in [KD99] that bidirectional dataflow analysis cannot generally be decomposed into sets of unidirectional dataflow analyses; PRE luckily has a favourable structure which often allows this decomposition to occur. Dhamdhere also proves that weak bidirectional dataflow analyses can be efficiently implemented and gives an efficient algorithm in [DP90].

6.1.8 Applications of PRE

We conclude our survey of PRE with a quick survey of its applications in compiler construction. An important optimization for object-oriented languages, such as Java, is the removal of redundant access path expressions. These expressions are the form:

$$p.f$$

and

$$p[i]$$

where

1. f is the name of a field within the referenced object;
2. i is an index within the referenced array;
3. p is
   (a) a pointer to an object or array; or
   (b) *itself a path access expression.*

An algorithm for their elimination is described in [HNW+01]. A similar objective is realized, although in a speculative setting, in [KKN04].

A related second application of PRE is in the movement of large expressions—expressions which contain more than one operator. [DS88] describes an enhancement to the original PRE algorithm to handle large expressions at the cost of doubling the number of basic blocks, a deficiency which is indicated in [Sor89]. A live-range optimal algorithm, based on LCM, which handles large subexpressions is presented in [Rüt98].

A third important application arises in the implementation of languages, such as Pascal and Java, which check array accesses to ensure that indices are within appropriate bounds. Quite often, a large subset of these time-consuming checks is redundant. An algorithm for their elimination is presented in [KW95].

A fourth application, the extension of PRE to eliminate redundant memory accesses is shown in [FE04], but postpones the optimization until link-time. [LCK+98] eliminates redundant loads and stores to enable register promotion.

A fifth important application of PRE is its extension to eliminate redundant Potentially Excepting Instruction (PEI)s in Java programs. Java's precise-exception semantics render this transformation very difficult. [OH05] "solves" this problem by using ordinary PRE to move PEIs, producing an optimized program which does not preserve the exception dependences of the original program. Code patching techniques are used to de-optimize the program into the original program in the event of a re-ordered PEI throwing an exception; the crucial assumption is that PEIs are rarely triggered (i.e. rarely throw exceptions). This solution is quite unsatisfactory since the code patching techniques used require run-time support from a virtual machine—it is not a *bona fide* program-to-program transformation.

A sixth important application of PRE, found in [CSV01], is its use in making a program more amenable to operator strength reduction. A final, non compiler-based application of PRE is an

inter-procedural extension of PRE in [ASD95] which is used to optimally position communication primitives used in distributed memory parallel machines.

6.1.9 Related Algorithms

Global Value Numbering

This survey would not be complete without a discussion of Global Value Numbering (GVN), an optimization which is intimately related to PRE. It often performs many of the same redundancy eliminations as PRE; in fact, the Jikes Research Virtual Machine (JIKES RVM) performs GVN instead of PRE. GVN, as described in [Cli95], can be viewed as the semantic counterpart to PRE. Essentially, GVN associates integers called "value numbers" with each expression instance such that any two expression instances which have the same value number are guaranteed to have the same value. Consequently, if expressions $a + b$ and $c + d$ are associated with the same value number, either can be used in lieu of the other. In contrast, PRE would optimize each expression independently, viewing each expression as being no more than the sum of its parts and hence unrelated to any other expressions with different parts.

[BCS97] and [CS95] present early attempts to combine GVN with PRE, but fall short since they can only remove available (fully redundant) computations. The algorithms are described in greater detail in the PHD dissertation of one of the authors[Sim98]. [BC94] shows how GVN can be used as an "enabling" transformation for PRE; it does not, however, present a hybrid algorithm.

The semantic-syntactic gap between GVN and PRE is bridged in [SKR90] where a Value-Flow Graph is constructed and used to represent knowledge of semantic equivalence of program terms in a syntactic manner which is amenable to use by the (syntactic) PRE algorithm. However, it formulates PRE in terms of "Herbrand Equivalences" (see also [KRS98] and [SKR91b]), and is consequently theoretically demanding relative to other PRE formulations. [vDH04b] describes GVN-PRE, a simple, elegant, practical, and easily-implemented hybrid. It is described in great detail in [vD04], the author's PhD dissertation, where Daniel Berlin, an expert compiler implementor from the IBM T.J. Watson Laboratories is quoted as saying:

> Having gone through hell implementing and maintaining SSAPRE[KCL+99, CCK+97], I looked upon the relative simplicity of GVN-PRE with glee. In fact, I'm considering implementing it.

After implementing it, he said:

> I'd like to point out that my GVN-PRE implementation can already properly build GCC with itself turned on. It took me almost two years to do this with SSAPRE.

This is a resounding endorsement for GVN-PRE.

Strength Reduction

The second algorithm intimately related to PRE is Strength Reduction (SR), the replacement of computations with equivalent, less expensive computations. The relation arises from the fact that instead of performing an expensive computation, the value required might be available in a variable

assigned by a cheaper computation, a concept quite similar to PRE (in which the variable is assigned by a previous computation).

A combination of LCM and SR is provided in [KRS93]. Subsequently, a more aggressive version was provided in [Hai98]. [CCK+97] also amalgamates PRE and SR, but acts upon an SSA-based IR. The use of Herbrand Equivalences is reprised by Knoop et al. in their amalgamation [SKR91a]. Finally, Dhamdhere presents his amalgamation in [Dha82a], [Dha82b] and [Dha89]. A further formulation, [DD95], is capable of performing PRE and SR on large expressions.

6.2 Related PDCE Research

In this section, we explore the research done to date on algorithms for DCE. DCE, the removal of assignments whose values will not subsequently be used, is (almost) the dual problem of PRE. Just as PRE naturally evolves from GCSE in an attempt to eliminate computations that are not fully redundant, so PDCE naturally emerges as an extension of DCE which can remove assignments which are not fully dead.

6.2.1 Classic PDCE

The genesis of DCE is found in a classic text on compiler construction[ASU86]. Here, a bit-vector analysis called liveness analysis is described, which, for each assignment instance, ascertains whether the target variable is subsequently used (live). If the target variable is not live, the assignment instance is deleted. However, this requirement for assignment deletion is too strong. Far more common, in real-world programs, are assignments whose target variables are used on a proper subset of program paths from the point of assignment. These assignments are termed **partially dead**.

The removal of partially dead assignments is addressed in the seminal paper [KRS94b], in which the authors provide the first algorithm for the removal of all partially dead assignments from a program. The algorithm is comprised of two phases: assignment sinking and assignment elimination. Assignment sinking moves candidate assignments as forward in the CFG as possible, deferring their computation as far as possible towards their point of use. Assignment elimination simply consists of the application of the afore-mentioned DCE algorithm[ASU86]. Consequently, it can be seen that the assignment sinking phase transforms the problem of partial DCE into the problem of (full) DCE.

The algorithm produces an optimal result for the given CFG. The time complexity of the algorithm is dominated by the three bit-vector analyses which comprise it. At first glance, this seems paradoxical when one takes into consideration that the optimal solution to the *simpler* problem of PRE requires *four* bit-vector analysis. However, to obtain the optimal solution to PDCE, the algorithm must be repeated until the CFG converges to a fixed point, effectively multiplying the number of analyses performed.

This is not a deficiency of the algorithm but rather an inherent complication in the problem: the operations of assignment sinking and assignment elimination may enable other assignments sinkings and assignment eliminations that were not previously possible. This is because assignments in the original program change actual program variables, and consequently their range of motion is restricted to precisely the area in which data dependencies are preserved. For assignments of the form $E : a = b + c$, there are three types of dependencies:

1. True dependence: E cannot be placed before a preceding instruction which defines b or c.
2. Anti dependence: E cannot be sunk below a succeeding instruction which uses a.
3. Out dependence: E cannot be sunk below a succeeding instruction which defines a.

However, the sinking or deletion of another assignment which shares a data-dependency with assignment E increases the possible range of motion for E, which was not originally available. This compels repeated iteration of the algorithm. The authors call this phenomenon *second-order effects*.

The authors further weaken the concept of partial deadness into partial faintness, in which a variable is termed faint if it is either dead or used by another faint variable. This is simply a scheme to capture a chain of assignments, each of whose target variables is used by the next assignment, and whose last target variable is not used at all or is used by the first assignment. While this increases the number of optimization opportunities, it cannot be modelled efficiently as a bit-vector problem, thereby precluding its adoption in industrial strength compiler implementations.

6.2.2 Important Reformulations of Classical PDCE

[BG97] takes a completely novel approach to the elimination of dead code from programs by using predication. Their approach is quite different from the established method of [KRS94b], due to Knoop, which attempts to sink assignments in the direction of control flow. This approach, in contrast, attempts to predicate assignments such that they will execute if and only if the result that they compute is guaranteed to be used. Hence, no migration of assignments is required, and so this method can achieve dead code eliminations which are not realizable by methods which rely on the sinking approach.

This algorithm requires two steps: predication and branch elimination. In the predication phase, for each assignment in the program, a Multiple Entry Multiple Exit (MEME) use region is derived. Quite contrary to its name, an important property of the use region is that there are *no* uses of the computed value in the use region. The use region extends along all paths from the assignment under consideration, until a use is encountered. At each exit from the region, the assignment's computed value is either live or dead. Then the backward slice of the region is computed which determines, for each live exit edge, the set of predicates which control whether that edge will be taken. The algorithm attempts to hoist these predicates against the control flow so that they execute before the partially dead assignment. The partially dead assignment is then moved into the true branch emanating from the predicate, effectively guarding it with that predicate. However, a side effect of this transformation is that a hoisted predicate renders redundant an occurrence of the same predicate later in the code. Therefore, the second phase of the algorithm, branch elimination, removes duplicate branches by threading control flow over predicates which are known to evaluate to true. Thereafter, no predicates can be evaluated redundantly.

[CGX03] also provides an algorithm for PDCE on predicated code. Unlike [BG97], their algorithm works with Single Entry Multiple Exit (SEME) regions.

6.2.3 Speculative PDCE

In [GBF97], Gupta et al. present a PDCE algorithm which uses predication to sink assignments in a manner reminiscent of [BG97], an earlier paper due to the same author, but unlike the algorithm

in that paper, takes profile information into account to determine whether the sinking is beneficial. This algorithm is comprised of four parts: analysis of costs and benefits for sinking assignments, predication enabling for sunken assignments, assignment sinking, and assignment elimination. It can be seen that this algorithm extends the classical PDCE framework of Knoop presented in [KRS94b], which consists of the last two phases: assignment sinking and assignment elimination.

The first phase, the cost-benefit analysis, initially determines for each join point in the program, the set of all paths from the start of the procedure to it (the "prefix" paths) and the set of all paths from it to the end of the procedure (the "suffix" paths). This operation initially seems computationally infeasible due to the large (or possibly infinite) number of such paths. However, the cost-benefit analysis is done only for acyclic paths in the program. Further, not all acyclic paths are considered: an experiment done by the authors determined that in the SPEC CPU 95 benchmark suite, 65% of the functions had no more than five paths with non-zero execution frequency. Consequently, the cost-benefit analysis expediently restricts its consideration to only those paths. Each "prefix" path for which an assignment can be moved to the bottom of the path is concatenated with each "suffix" path for which the same assignment occurs and can be moved to the top of the path. Essentially, this operation forms precisely the set of compound paths which contain an eliminatable dead assignment; they represent those program paths along which the elimination of partially dead code is a clear win. Other combinations of prefix and suffix paths represent costs. If the total benefit exceeds the total cost for a partially dead assignment statement, it can be sunk. In all, four analyses are used to perform the cost-benefit analysis.

Thereupon, the second step, predication enabling, is performed to determine predicate sinkability. This is performed with a single analysis. Assignment sinking, in a manner resembling [KRS94b] is performed in two analyses. In all, seven analyses are used to perform path-profile based predicated PDCE.

We urge the reader to note, carefully, that ISPDCE does not require four analyses to perform a cost-benefit evaluation, since its division of the target program into hot and cold parts does precisely this. Nor does it require an instruction set which supports predication, a requirement which precludes universal adoption. Overall, ISPDCE requires a mere three bit-vector analyses, compared to the required seven of [GBF97].

It is easy to see that ISPDCE is an elegant and simple algorithm, which can be easily understood and implemented. Thus, we have demonstrated *claim* 4 with respect to ISPDCE.

6.2.4 Optimal Speculative PDCE

Despite no evidence at present to suggest the infeasibility of the problem, it appears that nobody has formulated a solution.

6.2.5 Further Formulations: Imperative

Before continuing to describe other formulations of the solution to PDCE, we describe algorithms which bridge the problem of PRE and PDCE.

PRE aims to remove redundantly computed *expressions* of the form $a \oplus b$ from a CFG. However, almost all solutions to the PRE problem completely ignore the reality that a redundant computation

of $a \oplus b$ might occur as part of a redundant *assignment*, $c = a \oplus b$. Consider:

$$c = a \oplus b$$
$$\ldots$$
$$c = a \oplus b$$

The optimization of the second occurrence of $c = a \oplus b$ via PRE will at best result in the substitution of right-hand side of the redundant assignment by a compiler generated temporary result, yielding:

$$c = a \oplus b$$
$$\ldots$$
$$c = t_0$$

Consequently, the redundant assignment has *not* been removed, but merely transformed into a copy, which may be later removable. [Dha91] and [KRS95] advocated a more complete approach in their PRE algorithms in which the entire redundant assignment $c = a \oplus b$ would be removed, thereby emulating PDCE.

In [Dha91], Dhamdhere advocates an approach which removes partially redundant *assignments*. He modifies the original algorithm of Morel and Renvoise[MR79], by adjusting the property-summary functions of basic blocks to represent the data dependencies (enumerated in Section 6.2.1), prior to running their algorithm. This algorithm, since it deletes assignments, requires multiple applications until convergence, as previously explained. It is restricted to what Knoop calls "immediately profitable" assignment hoistings. That is, an assignment will be hoisted if it is itself redundant and hoisting it will result in deletion of the redundancy. Unfortunately, if it is not hoistable, it will remain in its original location, and may block the hoisting of a truly redundant assignment.

This deficiency is rectified in [KRS95], where Knoop develops a completely new algorithm to perform elimination of partially redundant assignments. It is much more aggressive than Dhamdhere's algorithm; while Dhamdhere simply deletes redundant assignments *in situ*, Knoop aggressively *moves* assignments, thereby precluding the blocking phenomenon of Dhamdhere's algorithm. This algorithm produces a program which is optimal both with respect to the number of expressions evaluated and the number of assignments performed. Unfortunately, this algorithm requires six dataflow analyses. The author cautions that while its running time is expected to be quadratic on well-structured CFGs, that it can degenerate to quartic time for arbitrary CFGs and thus should typically be used "for the optimization of time-critical sections of code of moderate size." In deference to repeated application of the algorithm being needed, (as is characteristic of assignment motion algorithms), the author adds the disclaimer: "Alternatively, one may limit the number of allowed hoisting and elimination steps heuristically."

[FKCX94] also presents an interesting initial approach to the problem of PDCE. The authors make the observation that partially dead code can be eliminated, if the assignment which defines a partially dead variable is moved to a location where it is "less dead." They define the degree of deadness of an assignment's target variable in terms of the set of execution pre-conditions required to be true for the assignment itself to execute. Optimization of the program involves increasing this set of pre-requisite conditions, effectively migrating the assignment in the direction of control flow into

the specific context in which the value held in the assignment's target variable is required. Since, their algorithm renders assignments "less dead," they call their algorithm "The Revival Transformation".

6.2.6 Further Formulations: Functional

The function programming language community has been grappling with the problem of DCE as well, albeit under the monikers Useless Code Elimination (UCE) and Useless Variable Elimination (UVE). Their approach is based on type systems, whose fundamentals are described in [BCDG00]. Two new types are introduced to augment the type system of the functional programming language under consideration:

1. δ, the type of a term in the program which may contribute to the final result;
2. ω, the type of a term which does not contribute to the final result.

Initially, certain very small parts of the program are annotated with these types. For example, the constant function $f(x) = 1$ (written in λ-calculus notation as):

$$\lambda x.1$$

is annotated to read

$$\lambda x^{\omega}.1^{\delta}$$

since the input to the function is not used *but* its result may be used. A type inference system is then applied to the entire program which annotates each program term with one of δ or ω. Terms which are annotated with an ω can finally be deleted.

[WS99] and [Kob00] use this technique to develop powerful type systems to perform UVE for languages such as Standard ML. [AEL07] and [AEL02] use this technique to remove unused arguments from function invocations in functional programs. [AEL07] also contains experimental results demonstrating speedup gains from 4% to 100%, on programs in the PACKS benchmark suite for the Curry functional programming language. [LS99] describes a type system to eliminate partially dead *recursive data*—data structures whose unused sections form recursive substructures—from functional programs. [Xi98] describe a type system based on dependent types for the ML family of languages, which allows unused unreachable clauses in pattern matching to be eliminated.

6.2.7 Further Formulations: Parallel

In [Kno98] and [Kno96], Knoop approaches the problem of PDCE for parallel programs by enhancing the approach taken in [KRS94b]. However, unlike, ordinary PDCE, the assignment sinking phase of parallel PDCE has to be careful not to sink assignments from regions of the program that can be executed in parallel with other regions into purely sequential parts of the program. This would have the effect of serializing a parallel program, thereby severely reducing its performance. Knoop introduces an appropriate side condition to prevent these undesirable code motions.

The analyses which this optimization uses are complicated by the fact that, on account of parallel execution, a given node may have predecessors which are not necessarily its immediate processors in the CFG. Basically, any node which can execute in parallel with another node, has the potential to finish its execution earlier, and to thus *effectively* be a predecessor of that node. This effect is termed

"interleaving" and the dataflow analyses must be formulated to take all possible interleavings into account.

[KM97] extends the PDCE of [KRS94b] to optimize the placement of data-distribution assignments for languages such as High Performance Fortran (HPF). Data-distribution assignments move sections of arrays to different processors. However, these redistributions are time consuming and motivate the removal of redundant re-distributions and unused re-distributions. Their algorithm consist of a phase of PDCE followed by a phase of Partially Redundant Assignment Elimination (PRAE). As with all algorithms hitherto seen which move assignments, their algorithm must be applied repeatedly until convergence. Certain minor modifications to this algorithm allow the compiler to optimize the program with a preference towards either algorithm efficiency or optimization capability:

1. reducing the number of iterations of the algorithm naturally results in a faster running optimization phase;

2. replacing the PDCE phase with a Partial Faint Code Elimination (PFCE) phase allows many more dead data-distributions to be removed.

6.2.8 Further Formulations: Hardware

The cost of executing a dead instruction is prohibitive, both in terms of the hardware resources involved in performing the computation as well as in the amount of power used by the processor. The problem of DCE in processor hardware is addressed in [BS02] where a dead instruction predictor is used to identify instructions which produce unused values. The predictor is implemented as a cache which remembers instructions that previously produced unused values. The cache achieves a prediction accuracy of up 90%; this is facilitated by the empirically observed property that most dynamically partially dead instructions arise from a very small set of statically partially dead instructions. An overall program performance improvement of between 5% and 10% was measured using a hardware simulation.

This technique is worth noting since the dynamic elimination of dead instructions reduces the onus on compiler writers to develop optimizations that are cognizant of the dynamic costs of instructions. It also assists software optimizations, especially those performed by traditional offline compilers which cannot always be relied upon since their optimizers may occasionally be misguided by unrepresentative program profiles. Further, compiler optimizations subsequent to PDCE often produce their own partially dead instructions. All these shortcomings are rectified by DCE implemented in the processor.

A more ambitious technique for processor-based DCE is attempted in [Rot99]. Instead of using a cache of previously dead instructions to predict future dead instructions, the processor constructs the dataflow graph of executed instructions dynamically. Instructions which store unused values in registers that are overwritten by subsequent instructions or which produce the same values that were produced by previously executed instructions are deemed "ineffectual." This property is propagated backward along the dependence edges of the dataflow graph to discover other ineffectual instructions. This technique allows the ineffectual instructions in a program trace to be identified and eliminated. Subsequently, re-execution of the trace can be performed with the shorter version which contains only "effectual" instructions. However, unlike the cache of [BS02] this dataflow graph constructed by the processor can grow arbitrarily large. This author remedies this by arbitrarily restricting the

size of the dataflow graph to 64 KB, which, unfortunately, is still considerably larger than the 5 KB cache of [BS02].

Unfortunately, the back-propagation of the "ineffectuality" property along the dependence edges of the dataflow graph is difficult to implement in hardware. [KR04] solves the problem of back-propagation by substituting it with a simpler technique called implicit back-propagation. This technique is based on removing only immediately ineffectual instructions. Then, a pruned dataflow graph is constructed to model the newly optimized program trace which is further monitored to discover new immediately ineffectual instructions (that would have been found by a single iteration of back-propagation). This process is repeated, and approximates true back-propagation (which provides a performance improvement of 12.3%) well enough to provide a performance improvement of 11.8%.

An excellent example of a hardware based optimization which stresses the value of cooperation between the compiler and processor is developed in [MRF97]. The authors make the simple but important observation that most information discovered by a compiler via its various analyses is often discarded at the end of the compilation. Often, the results of these discarded analyses must be recomputed by the processor. The authors remedy this prodigality by inserting annotations indicating register liveness into an instruction stream. At run-time the processor need only read these annotations (which the authors call Dead-Value Information (DVI)) to discover which registers are unused, and thereupon, which instructions are dead. The authors implement three hardware based program optimizations which use DVI: physical register file reduction, elimination of dead save and restore instructions in procedure calls, and elimination of dead save and restore instructions across context switches. They reduce approximately 50% of the unnecessary dynamic saves and restores across procedure calls and context switches, leading to an overall program performance improvement of 5%. However, this technique does require the processor's ISA to be modified to include DVI annotation instructions.

6.3 Previous Work on Hot-Cold Division

The division of a program into a hot region and a cold region—a process which we term **isothermality**—is an approach that has been used many times previously. Indeed, the reader should note that since the concept of isothermality is quite elementary, its value resides in its *application* to various optimizations, rather than as a self-contained concept. The articles cited below have been selected primarily, therefore, to showcase noteworthy applications of the isothermal method to various speculative optimizations, and, secondarily, to record its provenance in compiler literature.

[PH90] uses program profiles to order methods within a program's binary image, and to order basic blocks within a method's instruction stream: procedures which frequently call each other (i.e., have a hot call-graph edge between them) are juxtaposed in the program image; basic blocks which frequently transfer control to each other (i.e., have a hot CFG edge between them) are juxtaposed in the method's instruction stream. This juxtaposition of sections of code which frequently transfer control to each other increases the chance that calls and jumps will transfer control to sections of code already present in the instruction cache, reducing the delay to fetch them from memory, and thereby increasing program performance.

[CL96] provides a study of isothermality-based optimization to applications running on the Win-

dows operating system. In their approach, frequently executed traces through a method are computed from profile information. Those traces are then optimized to reduce the amount of time spent in leaf routines, to decrease code size (achieved by reducing of the length of hot traces), to remove partially dead code, to eliminate unnecessary copy propagation, and to eliminate unnecessary saves and restores of registers.

[HHR95] provides a framework which enables a compiler to better concentrate its optimization effort on the hot regions of programs. In this approach, tightly-coupled methods are inlined into their callers. This exposes opportunities for optimization across procedural boundaries that would not be apparent to a compiler that performs intraprocedural analysis. Procedure splitting can then be performed to shed the cold parts of the amalgamated procedure. The result of this sequence of integration and splitting is the creation of hot *methods*; these can be optimized aggressively by an intraprocedural optimizer. The advantages of this method are that it reduces the compiler's reliance on interprocedural analysis and optimization by reconstituting the program to create methods worthy of the compiler's attention *in their entirety*.

[WBP00] extends [HHR95] by formulating an algorithm to discover regions in a demand-driven manner. This laziness offsets high memory usage during compilation and even decreases compilation time. [SYN03] extends [HHR95] by performing region based compilation in a JIT compiler. Their fundamental contribution is the initiation of recompilation when control-flow exits a region boundary; the method currently being executed is subjected to on-stack replacement, and the newly optimized routine is executed instead. Similarly, [Wha01] performs dataflow analysis on only the subset of hot paths with a method; when control proceeds onto a cold path (thereby invalidating the results of the dataflow analysis), recompilation is initiated.

The Ablego framework [ZA07] uses an isothermal classification of the program to perform "outlining"—the removal of infrequently executed code from procedures. It is worthy to note that, in contrast to the algorithms described above, Ablego focuses its effort on the *cold* region to perform perform outlining. A function upon which outlining has been performed can then be inlined into any function which frequently calls it without dramatically increasing the size of that function.

The works cited above show conclusively that the hot-cold division of a program—isothermality—is a concept which has been used previously to empower various optimizations. We conclude with a list of salient features of our simple formulation of isothermality, for comparison:

1. A single threshold value, Θ, is used to distinguish between frequently and infrequently executed program parts.

2. The program parts under consideration are CFG nodes and edges, not traces.

3. Our formulation is intraprocedural.

This simple formulation empowers the optimizations of Speculative Partial Redundancy Elimination (SPRE) and Speculative Partial Dead Code Elimination (SPDCE) two important compiler optimizations whose provenance has been discussed earlier.

Chapter 7

Results and Analysis

7.1 ISPRE

7.1.1 Experimental Procedure

Compiler Suite

The GNU Compiler Collection (GCC) version 4.1.0 is the compiler suite used to gather experimental validation for the Isothermal Speculative Partial Redundancy Elimination (ISPRE) algorithm.

It was chosen for our experiments involving ISPRE because:

1. It provides Profile Driven Feedback (PDF) which is required by ISPRE.

   It is, in fact, the first version of GCC to provide PDF to the phase of the compiler where PRE algorithms are implemented, namely the module `gcse.c`. Previous versions of GCC did offer PDF, but this information was only made available long after the `gcse.c` module executed. The following quote from a personal communication[Bos06] with the GCC developers verifies this:

   > `You can't have profile information before PRE in any GCC 3.x based compiler. There is a pass commonly known as the "old RTL loop optimizer" (see loop.c) which destroys the CFG. That means that if you'd have profile information before loop, it would be destroyed along with the CFG. So GCC 3.x builds profile information _after_ running loop.`
   >
   > `Now if you look at the pass schedule, you'll see that loop runs after PRE.`

   Note: This detail is provided because there exist publications on the topic of SPRE (not due to the author of this dissertation), which use the GCC 3.x family to derive experimental results. The author of this thesis, however, has used version 4.1.0 to ensure correct, credible results.

2. It contains front-ends for many different programming languages. In particular, C, C++, FORTRAN 77, and FORTRAN 90 are officially supported. This facilitates better coverage of benchmark suite used, SPEC CPU 2000.

3. It is a highly-optimizing compiler and therefore allows ISPRE to be evaluated not in isolation but in the company of many other unrelated optimizations which are effectively "competing" simultaneously with ISPRE to optimize the program.

Compiler Modification and Configuration

The GCC compiler's implementation of PRE is partitioned into two components:

1. an "oracle", which inspects the CFG (and program profile) to derive changes which should be made to the CFG;

2. a "PRE machinery phase", which applies the changes computed by the oracle to the CFG.

GCC 4.1.0, by default, has one PRE algorithm already implemented—LCM—in the module `lcm.c`. The optimizations ISPRE and SPRE (in the modules `ispre.c` and `spre.c`) were custom written to replace the module `lcm.c` as depicted in Figure 7.1.

Since both components work on RTL (an IR which is source-language independent), no source-language specific modifications are required.

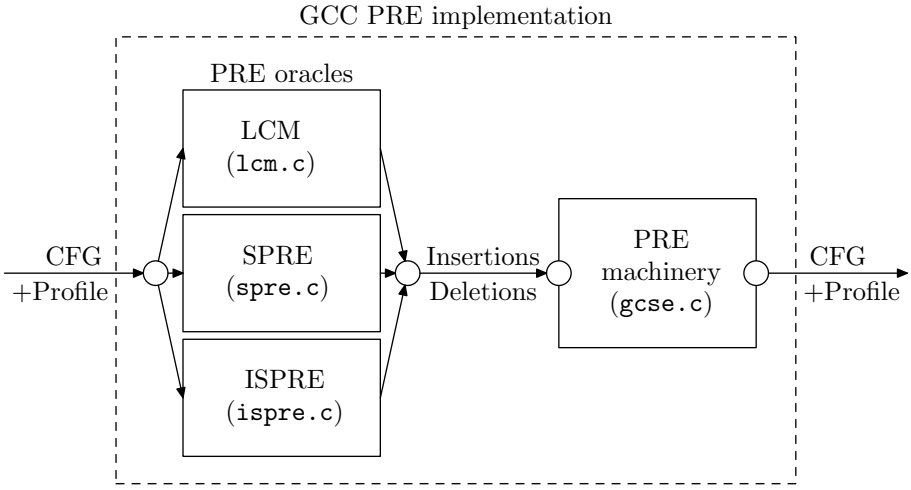


Figure 7.1: Implementation of PRE algorithms in GCC.

Benchmark Suite

The benchmark suite used for our experiments is SPEC CPU 2000 version 1.3. It was chosen for our experiments involving ISPRE because:

1. It is an industry-standard benchmark suite designed to provide a common point of comparison for optimizing compilers. It is also used extensively in research, both academic and industrial.

2. Its benchmarks programs are written in C, C++, FORTRAN 77, and FORTRAN 90 thereby allowing the ISPRE algorithm to be tested over a variety of languages.

3. Its benchmark programs span a wide variety of computation-intensive applications involving data compression, natural and artificial language processing, computer graphics, artificial intelligence, fluid dynamics, and particle physics.

Benchmark Suite Configuration

The SPEC CPU 2000 benchmark configuration file was written to invoke the GCC compiler with the switch `-O3` to induce the highest level of optimization to be performed.

As previously stated, such a high level of optimization must be used so that ISPRE can be evaluated alongside other compiler optimizations that are competing with it to optimize the program—as would be the case in any industrial strength compiler. It is an interesting question as to how much PRE benefits from the presence of other optimizations. Table A.1 in the appendix shows that PRE produces better overall benefits when other optimizations are enabled. It is left for future research to discover the mechanisms by which different optimizations interact.

Host Machine Configuration

The machine used to perform the benchmarking is an IBM PC with a Pentium-4 3.20 GHZ processor and 512 MB of RAM. The operating system used was Fedora Core 4.2 running Linux kernel version 2.6.11.

Experimental Procedure

SPEC CPU 2000 provides three modes for running benchmarks:

`train`, which represents the training run performed for speculative optimizations.

`test`, which represents a non-training run of the benchmarks on small inputs. This mode is typically used during the phase of compiler development and debugging.

`ref`, which represents a non-training run of the benchmarks on large inputs. This mode is used to conclusively test the effect of compiler optimizations.

All experimental results presented in this chapter were performed using the sequence `train`+`ref`, for each benchmark. Furthermore, validation testing was performed to compare the output of the benchmarks optimized by ISPRE and SPRE against reference outputs provided by the SPEC corporation. This ensures correctness of the optimized programs. All program runs from which timing data is collected are initiated by the benchmark driver `runspec` which is provided with the benchmark suite. This driver, in addition to performing validity checking, will also:

1. run each benchmark until convergence in its execution time occurs; and
2. average the execution times collected over a series of runs to cancel out flutter in the runtimes.

Miscellaneous Notes

The following 2 benchmarks were excluded from the experimental process:

1. `252.eon`, which does not compile with GCC 4.1.0.
2. `200.sixtrack`, which causes the SPRE algorithm to create data structures (used by the SPRE optimization) larger than the memory available on the test platform, eventually crashing it. For this benchmark, we show compilation times and code size measurements for modules that compiled successfully.

In the following experiments, ISPRE is performed with $\Theta = 90\%$. In preliminary investigations, the performance results vary in an unpredictable way as Θ changes, but within small limits.

7.1.2 Executable Size

Benchmark	LCM	SPRE		ISPRE	
	Size	Size	Ratio	Size	Ratio
		CINT			
164.gzip	38,572	39,848	(1.033)	38,336	(0.994)
175.vpr	118,572	122,852	(1.036)	116,912	(0.986)
176.gcc	1,384,560	1,583,682	(1.144)	1,176,684	(0.850)
181.mcf	10,104	14,236	(1.409)	10,200	(1.010)
186.crafty	187,944	196,648	(1.046)	187,832	(0.999)
197.parser	139,644	152,716	(1.094)	147,344	(1.055)
253.perlbmk	510,424	568,540	(1.114)	517,648	(1.014)
254.gap	454,184	469,700	(1.034)	425,896	(0.938)
255.vortex	486,184	487,100	(1.002)	484,216	(0.996)
256.bzip2	35,364	37,508	(1.061)	31,588	(0.893)
300.twolf	194,372	194,608	(1.001)	185,176	(0.953)
		CFP			
168.wupwise	24,904	25,904	(1.040)	24,040	(0.965)
171.swim	5,592	5,656	(1.011)	5,624	(1.006)
172.mgrid	14,472	15,446	(1.067)	13,576	(0.938)
173.applu	57,480	59,794	(1.040)	56,936	(0.991)
177.mesa	525,976	590,106	(1.122)	460,444	(0.875)
178.galgel	239,608	241,902	(1.010)	241,464	(1.008)
179.art	13,716	17,584	(1.282)	17,012	(1.240)
183.equake	18,596	24,272	(1.305)	19,176	(1.031)
187.facerec	62,136	64,068	(1.031)	60,520	(0.974)
188.ammp	109,688	121,508	(1.108)	107,380	(0.979)
189.lucas	44,648	45,070	(1.009)	44,296	(0.992)
191.fma3d	1,108,968	1,115,736	(1.006)	1,136,376	(1.025)
200.sixtrack	810,504	815,848	(1.007)	826,184	(1.019)
301.apsi	110,904	113,374	(1.022)	108,920	(0.982)
TOTAL	6,707,116	7,123,706	(1.062)	6,443,780	(0.961)

Table 7.1: Executable Size (in bytes): LCM vs. SPRE vs. ISPRE

Measurements

Table 7.1 shows the difference in the quantity of executable code produced (measured in bytes), under the effect of 3 PRE algorithms: LCM, SPRE, and ISPRE.

Column 2 ("LCM size") shows the effect of compilation with LCM, the default algorithm used by GCC. The values in column 2 establish a baseline, which demonstrates the effectiveness of both the SPRE and ISPRE algorithms. (In the absence of a baseline, it would not be clear how SPRE and ISPRE influence the metric in *absolute* terms).

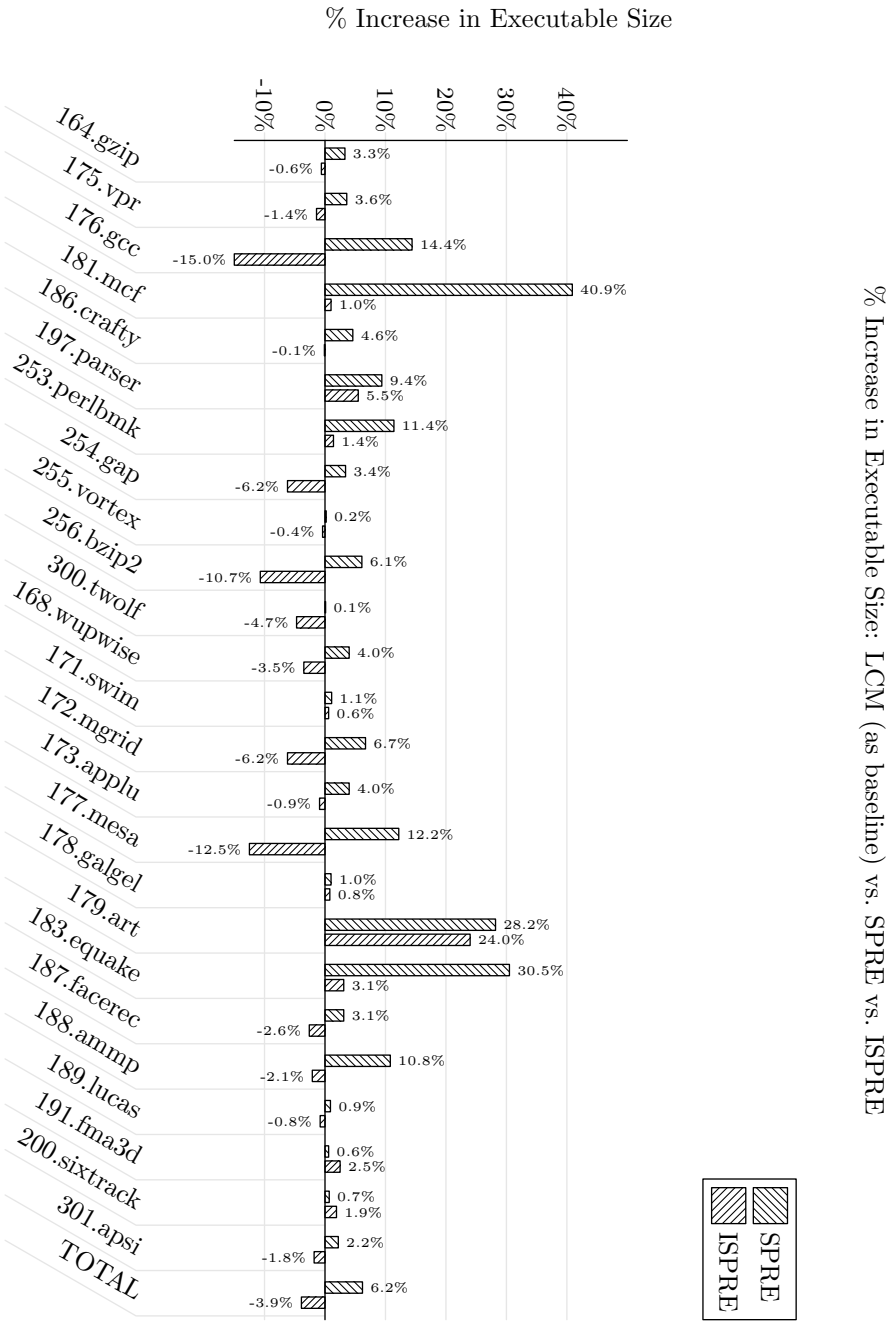


Figure 7.2: % Increase in Executable Size: LCM (as baseline) vs. SPRE vs. ISPRE

Further, each measurement in column 2 is the result of a 2-step `train+ref` (i.e., a PDF compilation). A two-step PDF compilation is required, even though LCM is not a speculative algorithm, because it is important to change only one variable at a time when measuring cause and effect—If PDF compilation were to be disabled, it would implicitly disable other PDF phases in the compiler, which are active during the compilations which produce SPRE and ISPRE measurements.

Column 3 ("SPRE Size") and Column 5 ("ISPRE Size") show the effect of PDF compilation with the SPRE and ISPRE algorithms respectively. Column 4 ("SPRE Ratio") and Column 6 ("ISPRE Ratio") each show the value of the previous column divided the baseline.

Finally, all size measurements provided were obtained by retrieving the size of the `text` (i.e., code) segment of the executables produced by the compiler with the program `size`. The overall executable size is not used in making comparisons because other segments in the executable such as the `bss` and `data` segments would obscure changes in the size of the `text` segment. This also enables our measurements to be used in talking credibly about change in the size of code caches in JIT compilers. No size changes occurred in other segments, as can be expected.

The measurements (relative to the baseline) in Table 7.1 are depicted graphically in Figure 7.2.

Analysis

Table 7.1 shows a very interesting overall measurement:

1. ISPRE decreases the size of code produced by 4%, relative to the baseline, on average;
2. SPRE increases the size of code produced by 6%, relative to the baseline, on average.

By itself, this is an impressive result. It shows that ISPRE, on account of its code size reduction, is suitable for use in compilers for embedded systems, where code storage-space is at a premium. It is *preferable to* LCM, on average, and *preferable to* SPRE both on average *and* on a benchmark-by-benchmark basis (`191.fma3d` and `200.sixtrack` being the only cases where SPRE produced smaller code segments than ISPRE). SPRE, in particular, has increased the size of all 25 benchmarks. ISPRE has actually decreased the size of 10 benchmarks relative to the baseline.

The afore-stated overall measurement is much more important, however, when it is also considered that ISPRE optimizes programs as effectively as SPRE (a fact which is demonstrated by Table 7.4). It becomes apparent, thereupon, that SPRE, in its quest for minimizing the number of redundant computations, is being overzealous: it is inserting large numbers of computations into almost completely unused parts of programs, in order to remove redundancies in other parts—which are almost as completely unused. ISPRE, in contrast and *by design*, concentrates only on removing redundancies from the hot region (which is typically quite small) by inserting computations on the ingress edges (which are few in number).

The only benchmark on which ISPRE increased the code size dramatically (24%) is `179.art`. SPRE, however, caused dramatic code size increases for `179.art` (28%) as well as `176.gcc` (14%), `181.mcf` (40%), and `183.equake` (30%). Nevertheless, `179.art` shows that ISPRE is susceptible to dramatic code size increases, just like SPRE. This can be accounted for by considering the modus operandi of ISPRE: A partially redundant computation must be placed on all the ingress edges to turn a partially redundant computation in the hot region into a fully redundant computation (which can then be deleted). Therefore, if there are many ingress edges which enter the hot region, a large increase in code size can result.

7.1.3 Compilation Time: PRE Phase Only

Benchmark	LCM	SPRE		ISPRE	
	Time	Time	Ratio	Time	Ratio
		CINT			
164.gzip	0.030	0.110	(3.67)	ϵ	(0.00)
175.vpr	0.020	0.570	(28.50)	0.080	(4.00)
176.gcc	0.750	10.600	(14.13)	0.650	(0.87)
181.mcf	0.020	0.020	(1.00)	0.000	(0.01)
186.crafty	0.130	1.420	(10.92)	0.070	(0.54)
197.parser	0.080	0.760	(9.50)	0.060	(0.75)
253.perlbmk	0.340	4.540	(13.35)	0.120	(0.35)
254.gap	0.210	2.480	(11.81)	0.180	(0.86)
255.vortex	0.220	1.120	(5.09)	0.190	(0.86)
256.bzip2	ϵ	0.110	———	0.010	———
300.twolf	0.100	1.650	(16.50)	0.120	(1.20)
		CFP			
168.wupwise	0.010	0.010	(1.00)	ϵ	(0.00)
171.swim	ϵ	ϵ	———	ϵ	———
172.mgrid	ϵ	ϵ	———	ϵ	———
173.applu	0.040	0.040	(1.00)	0.030	(0.75)
177.mesa	0.310	5.360	(17.29)	0.350	(1.13)
178.galgel	0.330	ϵ	(0.00)	0.250	(0.76)
179.art	0.030	ϵ	(0.00)	ϵ	(0.00)
183.equake	0.010	0.170	(17.00)	ϵ	(0.00)
187.facerec	0.020	0.020	(1.00)	0.040	(2.00)
188.ammp	0.070	0.590	(8.43)	0.070	(1.00)
189.lucas	0.020	0.020	(1.00)	0.030	(1.50)
191.fma3d	0.480	ϵ	(0.00)	0.470	(0.98)
200.sixtrack	0.460	0.540	(1.17)	0.530	(1.15)
301.apsi	0.050	ϵ	(0.00)	0.030	(0.60)
TOTAL	3.730	30.130	(8.08)	3.280	(0.88)

Table 7.2: Compilation Time (in seconds) for PRE phase only: LCM vs. SPRE vs. ISPRE

Measurements

Table 7.2 shows the compilation time required *by the* PRE *phase* of the GCC compiler when each of the following 3 PRE algorithms are used: LCM, SPRE, and ISPRE.

Note: A subsequent experiment will show time measurements for the *entire* compilation, for each PRE algorithm.

Column 2 ("LCM time") shows the time required for compilation with LCM, the default PRE algorithm used by GCC. The values in column 2 establish a baseline. Also, each measurement in

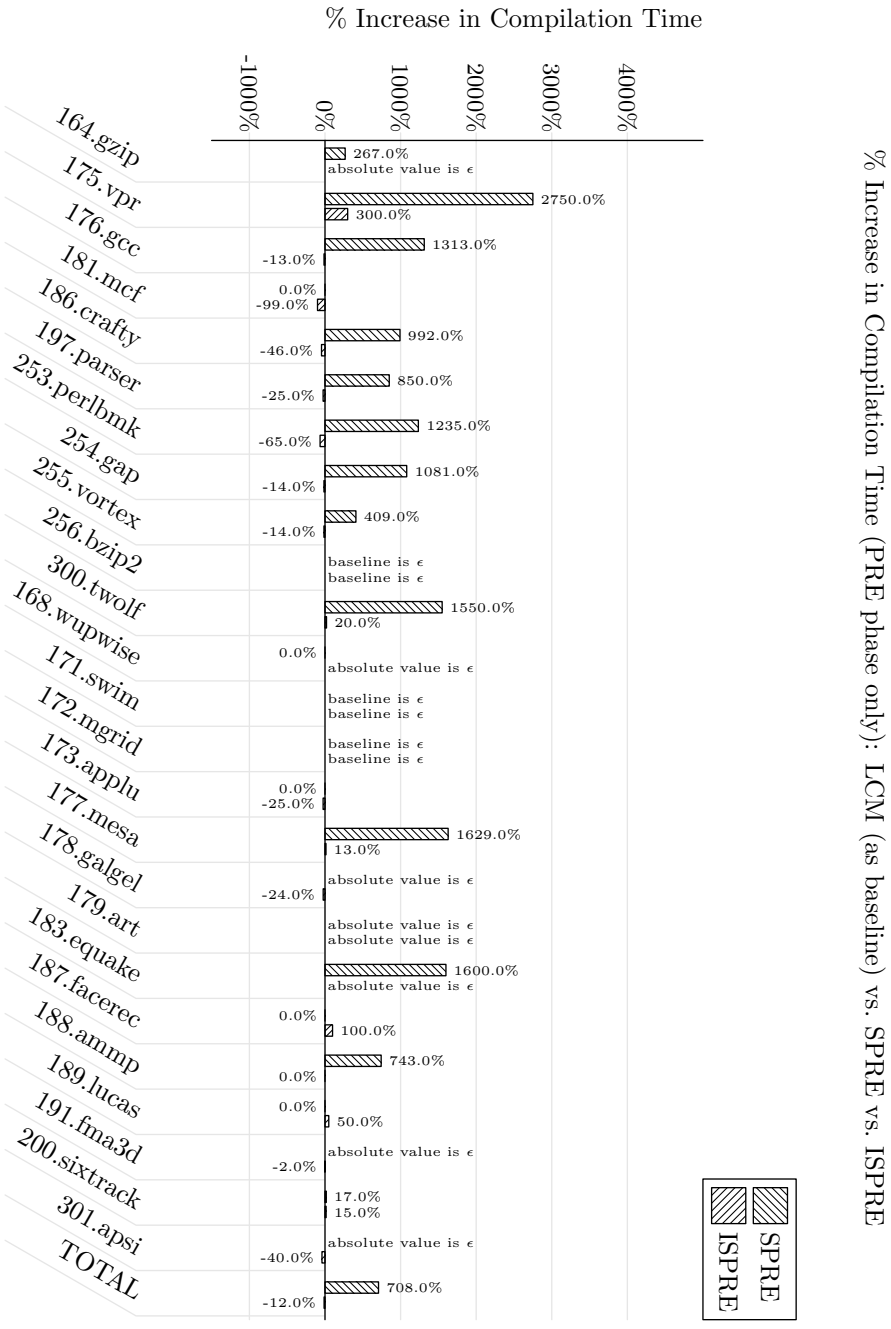


Figure 7.3: % Increase in Compilation Time (PRE phase only): LCM (as baseline) vs. SPRE vs. ISPRE

column 2 is the result of a 2-step `train+ref` (i.e., a PDF) compilation. The need for a baseline and a two-step PDF compilation for LCM measurements was previously discussed in Section 7.1.2.

Column 3 ("SPRE Time") and Column 5 ("ISPRE Time") show the effect of PDF compilation with the SPRE and ISPRE algorithms, respectively. Column 4 ("SPRE Ratio") and Column 6 ("ISPRE Ratio") each show the value of the previous column divided by the baseline.

If any time measurement was so small that it was reported as zero by the GCC time measurement routines (provided in the module `timevar.c`) which were used to gather timing measurements, an "ϵ" is used to denote that. When an ϵ occurs in an entry in the baseline column ("LCM Time"), ratios (found by dividing by the baseline value) cannot be computed and their absence is denoted by a triple-dash (———).

The measurements (relative to the baseline) in Table 7.2 are depicted graphically in Figure 7.3.

Analysis

Table 7.2 shows a very promising overall measurement:

1. ISPRE *decreases* the time required to perform PRE by about 12%, relative to the LCM baseline, on average;

2. SPRE *increases* the time required to perform PRE by about 800%, relative to the LCM baseline, on average.

Note: This measurement in concert with the results of a subsequent experiment—which shows that ISPRE optimizes programs as well as SPRE—validates *claim* 1 and *claim* 2 of this dissertation.

ISPRE vs. SPRE

The aggregate measurements show that ISPRE is approximately 9.18 times faster ($\frac{8.08}{0.88}$) than SPRE. It is imperative to justify these numbers by recourse to the working of each algorithm since they are almost an order of magnitude apart.

It is easy to see why ISPRE is much faster than SPRE. ISPRE, as explained earlier, uses only 2 bit-vector analyses. However, modern formulations of SPRE, ours included, involve the creation of a flow network derived from the CFG of the function being optimized, where the execution frequencies of blocks and edges (obtained from the program profile) are used as capacities for the edges of the network. Then, a maximum-flow solver is used to push a maximum-flow through the network. (The fastest maximum-flow solver available—Andrew Goldberg's HIPR algorithm[CG97]—was used for this experiment.) Finally, the flows for each block and edge are inspected to deduce the deletions and insertions of instructions that should be made to the blocks and edges of the CFG in order to minimize the number of redundant computations.

An Aside: It is interesting to note that Goldberg's algorithm is so difficult to understand that we were compelled to use *his* own source code, made available at his website[Gol03]. Alternative algorithms, such as the classic Ford-Fulkerson algorithm, give SPRE even worse compilation times. When the inefficiency of simpler network-flow algorithms is considered, coupled with the conceptual difficulty and *copyright status* of high-performance flow-solvers such as HIPR, it remains to be seen how well SPRE would be implemented in an industrial setting, where legal issues and algorithmic complexity would be prevailing considerations.

ISPRE, however, is *not* always a clear winner over SPRE. Consider the benchmarks `178.galgel`, `187.facerec`, `189.lucas` and `191.fma3d`. In particular, `187.facerec` shows that ISPRE *can* take twice as long as SPRE. These 4 benchmarks are the FORTRAN 90 benchmarks of the SPEC CPU 2000 benchmark set. At present, we cannot account for this anomalous behaviour other than to make reference to the fact that a new FORTRAN front-end was used in the GCC 4.0 series of compilers, citing [GCC07]:

> "A new FORTRAN front end has replaced the aging GNU FORTRAN 77 front end. The new front end supports FORTRAN 90 and FORTRAN 95. It may not yet be as stable as the old FORTRAN front end."

Nevertheless, it gives us the opportunity to discuss the pathological cases in which SPRE can run faster than ISPRE.

ISPRE vs. SPRE: Can SPRE run faster than ISPRE ?

Since ISPRE and SPRE have such entirely different modes of operation (flow networks vs. bit-vector analyses), this is difficult to answer conclusively.

However, it should be noted that certain types of programs may have CFGs whose flow networks are relatively simple and quickly solvable. A large number of numerically intensive programs fall into this category—they have large basic blocks and minimize the amount of branching performed. Consequently, the flow networks for numerically intensive programs are often simpler and more quickly solvable, in contrast to the flow networks for a program like `176.gcc`, a compiler, which is effectively a collection of finite state machines (for parsers, lexical analysers, instruction selectors, instruction schedulers, etc.) where the basic block (the "action") for each state is small, and excessive jumping between states occurs, sometimes to the extent that the CFG is dense and the flow network derived from it is likewise considerably dense.

ISPRE vs. LCM

The LCM algorithm performs 4 global bit-vector analyses. ISPRE, in contrast, performs only 2 bit-vector analyses, one of which is local. As expected, it runs faster than LCM on 16 benchmarks, and breaks even on 2 others (`171.swim` and `172.mgrid`). Yet, the 7 benchmarks `175.vpr`, `256.bzip2`, `300.twolf`, `177.mesa`, `187.facerec`, `189.lucas` and `200.sixtrack` show compilation times with ISPRE exceeding compilation time with LCM (sometimes by as much as a factor of 4 in the case of `175.vpr`).

As was the case with SPRE, these results give us the opportunity to discuss the pathological cases in which LCM can run faster than ISPRE.

ISPRE vs. LCM: Can LCM run faster than ISPRE ?

Yes. Despite using 1 local analysis, ISPRE can run slower than LCM which uses 4 global analyses. It seems paradoxical to think this, since local analyses are performed on the relatively small hot region, while global analyses must examine the entire CFG.

Consider, however, a point of confluence in the CFG where two edges meet. Consider, further, for example, a forward analysis whose confluence operator is intersection: in order for the program

property (under consideration) to be true at a confluence point, it must be true for all edges leading into that confluence point. In a global analysis performed by LCM, *which must consider all nodes and edges*, a lack of the property (under consideration) at one incoming edge will prevent the property from being true at the confluence point. By contrast, in a local analysis performed by ISPRE, *certain incoming edges at a confluence point are not even considered*, since those edges are cold. Therefore, even if the property is false on those cold incoming edges, it will *not* prevent the property from being true at the confluence point. Consequently, the property will propagate to the confluence point *and* the node at the confluence point will be put onto a worklist, so that the property can, in turn, propagate to other nodes in the CFG.

Hence, it can be seen that ISPRE is not always the clear winner just because it has 1 local analysis instead of 4 global analyses: global analyses can terminate earlier.

7.1.4 Compilation Time: All Phases

Benchmark	LCM	SPRE		ISPRE	
	Time	Time	Ratio	Time	Ratio
		CINT			
164.gzip	2.360	4.000	(1.69)	2.210	(0.94)
175.vpr	7.170	10.000	(1.39)	6.450	(0.90)
176.gcc	75.050	78.000	(1.04)	63.280	(0.84)
181.mcf	1.020	1.170	(1.15)	0.000	(0.00)
186.crafty	9.970	11.350	(1.14)	9.860	(0.99)
197.parser	8.690	8.780	(1.01)	7.900	(0.91)
253.perlbmk	30.510	32.010	(1.05)	18.930	(0.62)
254.gap	26.840	27.800	(1.04)	25.150	(0.94)
255.vortex	21.770	23.120	(1.06)	22.220	(1.02)
256.bzip2	1.810	2.000	(1.10)	1.540	(0.85)
300.twolf	12.870	14.050	(1.09)	12.250	(0.95)
		CFP			
168.wupwise	0.980	0.990	(1.01)	0.990	(1.01)
171.swim	0.270	ϵ	(0.00)	0.260	(0.96)
172.mgrid	0.600	0.600	(1.00)	0.600	(1.00)
173.applu	2.780	2.780	(1.00)	2.770	(1.00)
177.mesa	25.510	28.760	(1.13)	23.490	(0.92)
178.galgel	9.960	12.000	(1.20)	9.970	(1.00)
179.art	0.790	0.800	(1.01)	0.760	(0.96)
183.equake	1.000	1.240	(1.24)	1.000	(1.00)
187.facerec	2.650	2.680	(1.01)	2.670	(1.01)
188.ammp	6.560	6.990	(1.07)	5.880	(0.90)
189.lucas	1.800	1.850	(1.03)	1.810	(1.01)
191.fma3d	90.810	94.000	(1.04)	90.450	(1.00)

continues on next page...

. . . continued from previous page

Benchmark	LCM	SPRE		ISPRE	
	Time	Time	Ratio	Time	Ratio
`200.sixtrack`	42.130	44.000	(1.04)	42.110	(1.00)
`301.apsi`	5.880	5.890	(1.00)	5.900	(1.00)
TOTAL	389.780	414.860	(1.06)	358.450	(0.92)

Table 7.3: Compilation Time (in seconds) for all phases: LCM vs. SPRE vs. ISPRE

Measurements

Table 7.3 shows the compilation time required *by all compilation phases* of the GCC compiler when each of the following 3 PRE algorithms are used: LCM, SPRE, and ISPRE.

All measurements shown in this table are defined analogously to those in Section 7.1.3, with the exception that the duration of the *entire* compilation is measured.

The measurements (relative to the baseline) in Table 7.3 are depicted graphically in Figure 7.4.

Analysis

Table 7.3 shows a very important and promising overall measurement:

1. ISPRE *decreases* the entire compilation time by about 8%, relative to the LCM baseline, on average;

2. SPRE *increases* the entire compilation time by about 6%, relative to the LCM baseline on average.

This is a very important result: it shows that the 9-fold (PRE phase only) improvement in the running time of ISPRE over SPRE was *not* "lost in the noise" of the hundred-plus optimization phases that are part of the GCC 4.1.0 compiler. Indeed, compilation using ISPRE is 14% more efficient than compilation using SPRE, on average. In fact, ISPRE would be preferable to SPRE on a benchmark-by-benchmark basis, if not for `171.swim`, the only benchmark on which overall compilation time is lower with SPRE.

Furthermore, it can be seen that ISPRE is even preferable to LCM. Although it does take longer than LCM on 4 benchmarks (`255.vortex`, `168.wupwise`, `187.facerec`, and `189.lucas`), it never does so by more than 2%. It breaks even on 7 benchmarks (`172.mgrid`, `173.applu`, `178.galgel`, `183.equake`, `191.fma3d`, `200.sixtrack`, and `301.apsi`) and takes less time than LCM on the remaining 14 others.

The reader should note the amount of variance in compilation with SPRE over the baseline. It is as high as 69% (in the case of `164.gzip`). ISPRE, in contrast, with a variance of only 2% at most above the baseline, is much more predictable. This measurement is important to note because JIT compilers and their Adaptive Optimization System (AOS) often like to know the "expense" of each optimization so that they can compute cost-effective tradeoffs appropriately, when deciding whether or not to apply an optimization to a procedure in a program. SPRE, unfortunately, due to the afore-stated variance, cannot make a credible guarantee to the AOS about how long it will take. ISPRE, by contrast, could and is therefore more suitable for JIT compilation, in this respect.

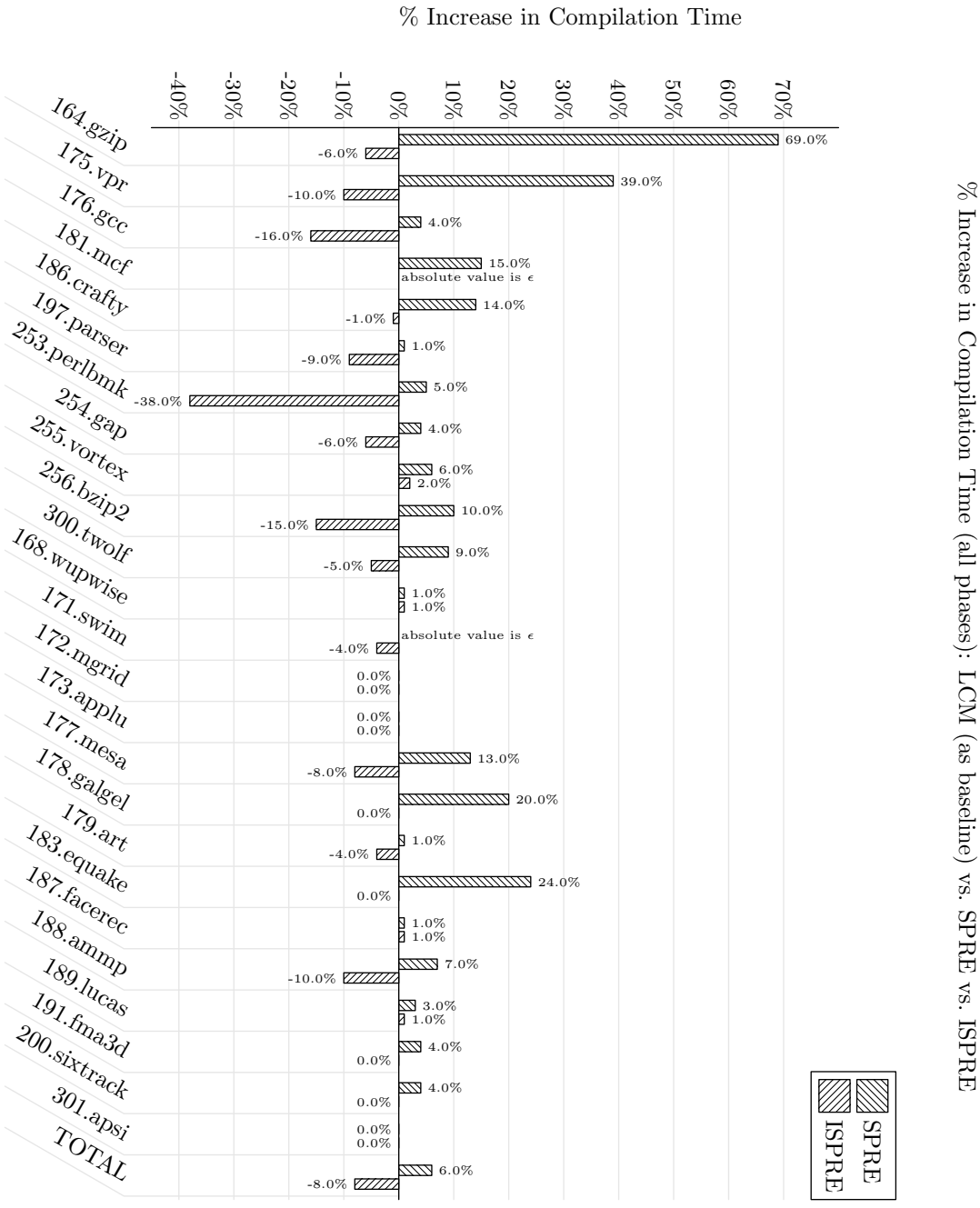


Figure 7.4: % Increase in Compilation Time (all phases): LCM (as baseline) vs. SPRE vs. ISPRE

7.1.5 Execution Time

Benchmark	LCM	SPRE		ISPRE	
	Time	Time	Ratio	Time	Ratio
		CINT			
164.gzip	176.938	171.989	(0.97)	172.376	(0.97)
175.vpr	156.547	154.142	(0.98)	154.676	(0.99)
176.gcc	70.527	67.581	(0.96)	67.534	(0.96)
181.mcf	145.904	144.687	(0.99)	143.431	(0.98)
186.crafty	105.810	104.191	(0.98)	99.484	(0.94)
197.parser	183.814	154.861	(0.84)	156.137	(0.85)
253.perlbmk	141.086	132.226	(0.94)	141.377	(1.00)
254.gap	88.160	79.139	(0.90)	80.675	(0.92)
255.vortex	143.389	136.041	(0.95)	135.658	(0.95)
256.bzip2	150.811	147.824	(0.98)	147.937	(0.98)
300.twolf	211.039	206.507	(0.98)	206.342	(0.98)
		CFP			
168.wupwise	163.596	130.924	(0.80)	128.337	(0.78)
171.swim	166.107	165.464	(1.00)	165.613	(1.00)
172.mgrid	302.776	309.076	(1.02)	309.257	(1.02)
173.applu	198.493	198.503	(1.00)	195.764	(0.99)
177.mesa	184.451	187.687	(1.02)	186.231	(1.01)
178.galgel	481.414	477.412	(0.99)	480.583	(1.00)
179.art	324.219	309.770	(0.96)	304.942	(0.94)
183.equake	74.599	73.805	(0.99)	73.710	(0.99)
187.facerec	223.618	218.550	(0.98)	222.152	(0.99)
188.ammp	289.520	268.804	(0.93)	265.343	(0.92)
189.lucas	260.200	255.989	(0.98)	256.143	(0.98)
191.fma3d	260.376	257.915	(0.99)	257.772	(0.99)
301.apsi	262.299	281.686	(1.07)	283.253	(1.08)
TOTAL	4765.692	4634.774	(0.97)	4634.729	(0.97)

Table 7.4: Execution Time (in seconds): LCM vs. SPRE vs. ISPRE

Measurements

Table 7.4 shows the execution time (in seconds) of 24 benchmarks from the SPEC CPU 2000 benchmark suite, when each of the following 3 PRE algorithms are used: LCM, SPRE, and ISPRE.

Note:

1. All execution times shown are for the `ref` (very large) problem input set, as recommended by the SPEC Corporation—the `test` (small) problem set is recommended for use only during compiler development.

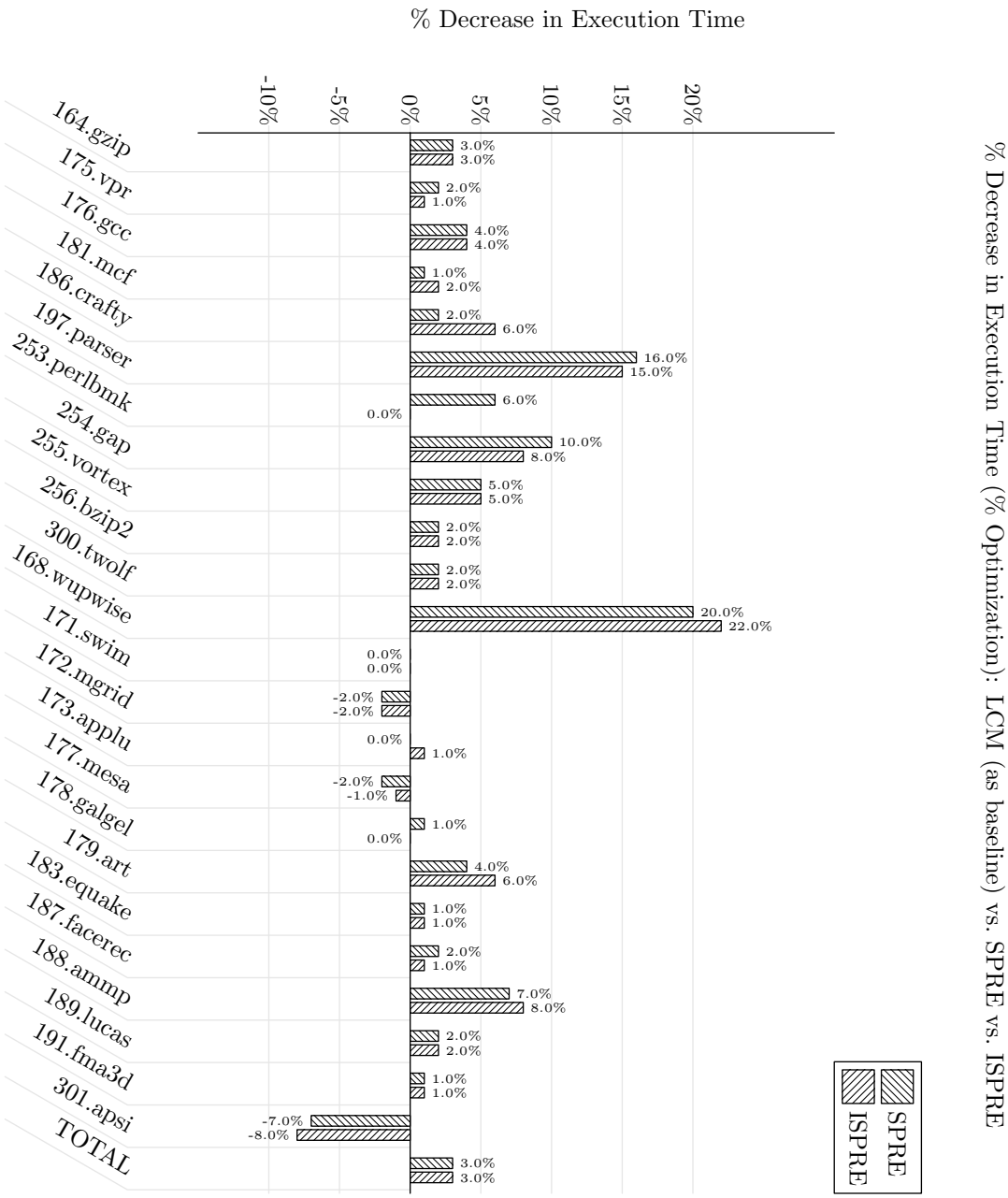


Figure 7.5: % Decrease in Execution Time: LCM (as baseline) vs. SPRE vs. ISPRE

2. For the reasons previously discussed (in Section 7.1.1), measurements for `200.sixtrack` and `252.eon` could not be provided.

Column 2 ("LCM time") shows the execution times of each benchmark when PRE is performed with LCM, the default PRE algorithm used by GCC. The values in column 2 establish a baseline, which shows that both SPRE and ISPRE improve program performance in absolute terms. Also, each measurement in column 2 is the result of a 2-step `train+ref` (i.e., a PDF) compilation. The need for a baseline and a two-step PDF compilation for LCM measurements was previously discussed in Section 7.1.2.

Column 3 ("SPRE Time") and Column 5 ("ISPRE Time") show the execution times of each benchmark when PRE is performed via the SPRE and ISPRE algorithms, respectively. Column 4 ("SPRE Ratio") and Column 6 ("ISPRE Ratio") each show the value of the previous column divided by the baseline.

The measurements (relative to the baseline) in Table 7.4 are depicted graphically in Figure 7.5.

Analysis

Table 7.4 contains the *most important measurements* presented in this dissertation, since, in concert with the measurements presented in Table 7.2 and Table 7.3 it validates the first two major claims of this thesis, namely:

1. Optimizations based on isothermality are less expensive to use than their optimal counterparts.

   (a) By itself, SPRE takes 9 times longer to execute than ISPRE.
   Table 7.2 demonstrated this.

   (b) Over an entire compilation, SPRE takes 14% longer to execute than ISPRE.
   Table 7.3 demonstrated this.

   and

2. Optimizations based on isothermality give performance improvements comparable to their optimal counterparts.
   We will now validate this statement with an examination of the measurements in Table 7.4.

Table 7.4 shows the following overall measurement:

> ISPRE and SPRE both decrease the overall execution time of the 24 benchmarks tested by 3%.

This validates claim 2, stated above, because it shows that ISPRE optimizes programs comparably well (in this case, equally well) as SPRE, its optimal competitor. Note, additionally, that on individual benchmarks, ISPRE sometimes provides superior optimization:

> ISPRE better optimized `181.mcf`, `186.crafty`, `168.wupwise`, `179.art`, and `188.ammp` than SPRE.

It is crucial to note that this measurement is *relative to the* LCM *baseline*, which shows that SPRE and ISPRE actually optimize the program. In the absence of a baseline, it would not be known

whether SPRE actually increases program performance *in absolute terms*. But the baseline shows that SPRE, and consequently ISPRE, are indeed *optimizations*.

However, it is important to keep this measurement in perspective:

1. Both SPRE and ISPRE decrease the performance of the benchmarks `172.mgrid`, `177.mesa`, and `301.apsi`.
2. Both SPRE and ISPRE have no effect on `171.swim`.
3. SPRE has no effect on `173.applu`.
4. ISPRE has no effect on `253.perlbmk` and `178.galgel`.

SPRE vs. ISPRE: Register Pressure

The decrease of the performance of the benchmarks `172.mgrid`, `177.mesa` and `301.apsi` by both SPRE and ISPRE is important. It shows that despite being "optimal", SPRE can still decrease the performance of a program.

We know that the main reason for this is that SPRE is register-pressure naïve. It will hoist expressions against control-flow higher than necessary, thereby increasing register pressure on paths from the point of placement to the original location of the expression.

ISPRE also suffers from this problem. However, the situation is not as dire for ISPRE since it does not hoist computations as far as possible, but rather hoists them to the ingress edges of the hot region. Since the hot region is typically small, the length of paths between the original location of the expression and the ingress edges on which it is placed can likewise be expected to be small.

Indeed, benchmarks such as `181.mcf`, `186.crafty`, `168.wupwise`, `179.art`, and `188.ammp` (which were optimized better by ISPRE) may benefit from ISPRE's better treatment of register pressure.

Multi-Pass ISPRE

Experiments performed to ascertain the difference in optimized program execution time when:

1. ISPRE is performed once at a high value of Θ (e.g., 90%);
2. ISPRE is performed twice: first using a high value of Θ (e.g., 90%) and then a low value of Θ (e.g., 50%)

give timings that are statistically indistinguishable from each other.

It should be noted that a low value of Θ makes the hot region of the program very large and the ISPRE algorithm effectively degenerates into ordinary Global Common Subexpression Elimination (GCSE).

7.2 ISPDCE

7.2.1 Experimental Procedure

Compiler/Virtual Machine Suite

The JIKES RVM 2.4.5 is the compiler/virtual machine infrastructure used to provide experimental validation for the Isothermal Speculative Partial Dead Code Elimination (ISPDCE) algorithm.

It was chosen for our experiments involving ISPDCE because:

1. It provides Profile Driven Feedback (PDF) which is required by ISPDCE.

   Note, however, that unlike GCC, described in Section 7.1.1, the JIKES RVM uses Continuous Program Optimization (CPO) to monitor the execution frequency of the methods of a program and recompiles those methods that are frequently used. The instrumentation and recompilation is performed without user intervention.

2. It is a VM for the Java programming language, thereby allowing us to benchmark the ISPDCE optimization on a very popular, frequently used programming language.

   In combination with the experimental results of ISPRE, we will have experimental validation of our algorithms over a large subset of the most popular programming languages in use.

3. It contains a highly-optimizing Just-In-Time (JIT) compiler and therefore allows ISPDCE to be evaluated not in isolation, but in the company of many other unrelated optimizations which are effectively "competing" simultaneously with ISPDCE to optimize the program.

Compiler Modification

Our experiments use the `optimizing` compiler, the compiler module provided by the JIKES RVM 2.4.5 for program optimization with Profile Driven Feedback.

The `optimizing` compiler has 3 levels of IR: High-Level Intermediate Representation (HIR), Low-Level Intermediate Representation (LIR), and Machine-Level Intermediate Representation (MIR). Our optimizations are implemented to work on HIR, and we modified the compiler driver module (`OPT_OptimizationPlanner`) to run our optimizations with the set of HIR optimizations provided by the `optimizing` compiler.

Both ISPDCE and its competitor optimization PDCE were custom-implemented for this series of experiments.

Note:

1. Previous versions of the JIKES RVM have used the name `adaptive` to refer to an optimizing compiler module which provides PDF, while using the name `optimizing` to refer to an optimizing compiler module which does *not* provide PDF. This is not the nomenclature used by JIKES RVM 2.4.5.

2. There exists another compiler module provided by the JIKES RVM called `baseline` which produces "obviously correct" translations of Java methods to native code and which does not have any optimization infrastructure. This compiler is of no interest to us, for the experiments concerning ISPDCE. Similarly, a third compiler module called `quick` (deleted from JIKES RVM version 2.9.1) is of no interest to us.

Benchmark Suite

The benchmark suite used for our experiments is SPEC JVM 98 version 1.04. It was chosen for our experiments involving ISPDCE because:

1. It is an industry-standard benchmark suite designed to provide a common point of comparison for Java virtual machines. It is also used extensively in research, both academic and industrial.

2. Its benchmark programs span a wide variety of computation-intensive applications involving data compression, parsing and compilation, databases, and computer graphics.

Benchmark Suite Configuration

The JIKES RVM provides many configurations for its compilation, garbage collection, and adaptive optimization subsystems.

For the purposes of benchmarking the ISPDCE algorithm, we used the `production` configuration. The JIKES RVM User's Guide (`http://www.jikesrvm.org/Configuring+the+RVM`) states that:

> This[the `production` configuration] is the highest performance configuration of Jikes RVM and is the one to use for benchmarking and performance analysis.

The `production` configuration ensures that the virtual machine is itself compiled with full optimization, and that the `optimizing` compiler and AOS is included. By contrast, the `prototype-opt` configuration, which is intended for use only during compiler development, is not used.

In contrast to the ISPRE experiments described previously, where GCC was instructed to compile benchmark programs with the `-O3` switch, the JIKES RVM is not passed any such switch. The level of compilation that it wants to use is at the discretion of the AOS.

Host Machine Configuration

The machine used to perform the testing is an IBM PC with a Pentium-4 1.8 GHZ processor and 1.0 GB of RAM.

The operating system used was Ubuntu 6.06 running Linux kernel version 2.6.15.

Experimental Procedure

SPEC JVM 98 provides three problem sizes for running benchmarks:

1, which represents a problem set of the smallest size;

10, which represents a problem set of medium size; and

100, which represents the largest problem set provided by the benchmark suite. This problem size is recommended by the SPEC Corporation for conclusively testing the effect of compiler optimizations.

Problem size 100 has been used for all ISPDCE experiments presented in this chapter.

Furthermore, validation checking is enabled so that the outputs of the benchmarks optimized by ISPDCE can be compared against reference outputs provided by the SPEC Corporation.

All program runs from which timing data is collected are initiated by the benchmark driver `SpecApplication` which is provided with the benchmark suite. This driver, in addition to performing validity checking, will also:

1. run the benchmarks until convergence in their run-times occurs; and

2. average the run-times collected over a series of runs to cancel out flutter in the run-times.

In the following experiments, R3PDE is performed with $\Theta = 90\%$.

7.2.2 Executable Size

Benchmark	Default	PDCE		ISPDCE	
	Size	Size	Ratio	Size	Ratio
_201_compress	47,496	47,777	(1.006)	48,248	(1.016)
_202_jess	53,632	53,848	(1.004)	53,848	(1.004)
_205_raytrace	56,680	56,757	(1.001)	56,805	(1.002)
_209_db	52,472	52,703	(1.004)	53,032	(1.011)
_213_javac	78,448	78,712	(1.003)	79,662	(1.015)
_222_mpegaudio	70,720	71,174	(1.006)	71,332	(1.009)
_227_mtrt	42,008	42,067	(1.001)	42,085	(1.002)
_228_jack	73,132	73,327	(1.003)	73,497	(1.005)
TOTAL	474,588	476,365	(1.004)	478,509	(1.008)

Table 7.5: Number of Instructions Before/After Optimization: Default JIKES RVM vs. PDCE vs. ISPDCE

% Increase in Instruction Count: Default Jikes RVM (as baseline) vs. PDCE vs. ISPDCE

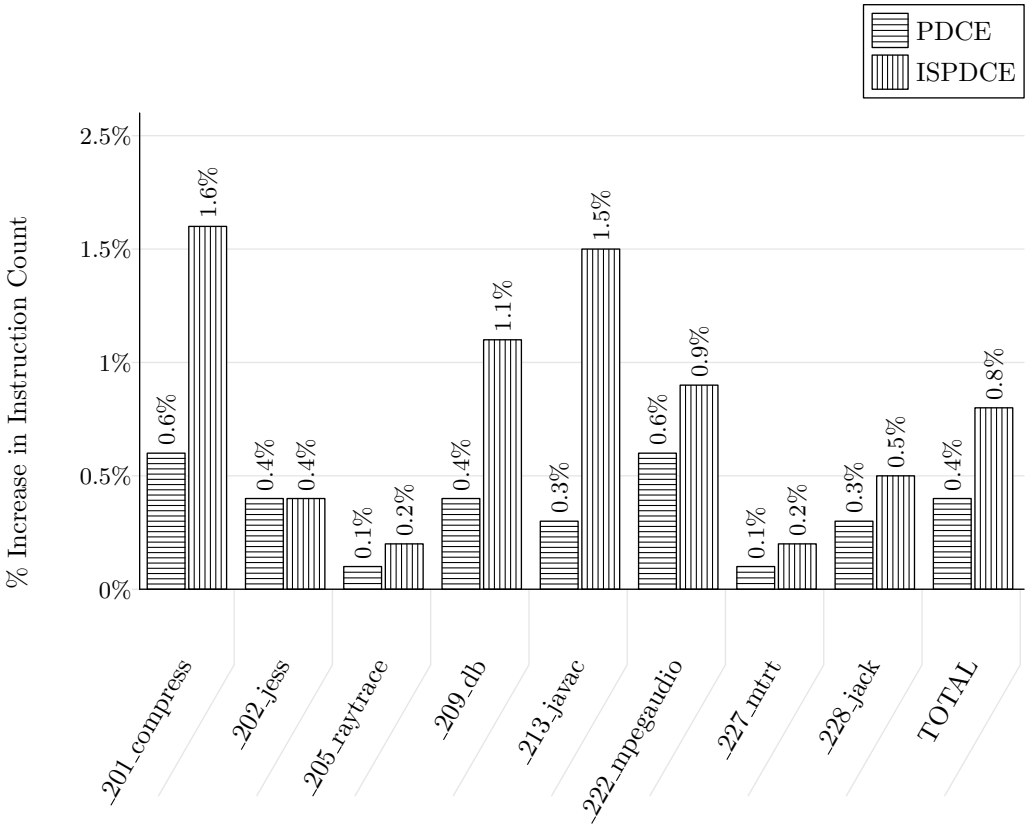


Figure 7.6: % Increase in Instruction Count: Default JIKES RVM (as baseline) vs. PDCE vs. ISPDCE

Measurements

Table 7.5 shows the quantity of code produced (measured in JIKES RVM IR instructions) under the effect of 2 DCE algorithms: PDCE, and ISPDCE.

Note:

1. Unlike the experiments of Section 7.1.2, where the amount of the native code produced (in bytes) was obtained simply by examining binary executables with the utility `size`, the JIKES RVM does not have a comparable method.

   Hence, we provide here a record of the total number of JIKES RVM IR instructions before and after our optimizations are performed. For each method being optimized, we record the number of IR instructions prior to optimization, as well as the number of IR instructions after optimization. These two numbers, totalled over all methods optimized for a given benchmark, give the before and after code sizes for that benchmark.

2. Unlike the experiments of Section 7.1.2, where the baseline for comparison was the LCM algorithm which was already implemented by GCC, the JIKES RVM does not provide a DCE algorithm *per se*. Instead, DCE in JIKES RVM 2.4.5 is provided as a "cleaning-up" mini-phase for other phases (in the modules `OPT_Simple` and `OPT_Simplifier`) in a very naïve and *ad hoc* manner which makes it unsuitable for direct comparison with ISPDCE. Hence, we have custom-implemented PDCE to facilitate meaningful comparison.

Column 2 ("Default Size") shows the amount of code produced by the default JIKES RVM (with its *ad hoc* DCE). The values in column 2 establish a baseline, which demonstrates the effect of both the PDCE and ISPDCE algorithms. (In the absence of a baseline, it would not be clear how PDCE and ISPDCE influence the metric in *absolute* terms).

Column 3 ("PDCE Size") and Column 5 ("ISPDCE Size") show the effect of optimization with the PDCE and ISPDCE algorithms respectively. Column 4 ("PDCE Ratio") and Column 6 ("ISPDCE Ratio") each show the value of the previous column divided the baseline.

The measurements (relative to the baseline) in Table 7.5 are depicted graphically in Figure 7.6.

Analysis

The purpose of this experiment is only to gain assurance that ISPDCE is not increasing the code size of programs explosively.

By design, ISPDCE will duplicate the hot region twice: once to create a "guard" region, and once more to create the region that is actually optimized. When one considers the colloquially proferred 80-20 rule, which states that 20% of a program is executed 80% of the time, it may seem, thereupon, that ISPDCE would increase code size by 40%.

The measurements in this table show that, on average, ISPDCE increases code size by less than 1%. The largest increase in code size occurred with the `_201_compress` benchmark at 1.6%.

It can also be seen that ISPDCE always produces more code than PDCE, (most severely in the case of `_213_javac`: 1.5% vs. 0.3%, a factor of 5) but this is to be expected since ISPDCE duplicates the hot region twice, while PDCE does not change the CFG structure at all.

7.2.3 Compilation Time: DCE Phase Only

Benchmark	PDCE	ISPDCE	
	Time	Time	Ratio
_201_compress	0.228	0.229	(1.007)
_202_jess	0.313	0.281	(0.896)
_205_raytrace	0.257	0.242	(0.941)
_209_db	0.282	0.292	(1.034)
_213_javac	0.381	0.375	(0.984)
_222_mpegaudio	0.369	0.335	(0.907)
_227_mtrt	0.298	0.265	(0.888)
_228_jack	0.297	0.295	(0.992)
TOTAL	2.425	2.313	(0.953)

Table 7.6: Compilation Time (in seconds) for DCE phase only: PDCE vs. ISPDCE

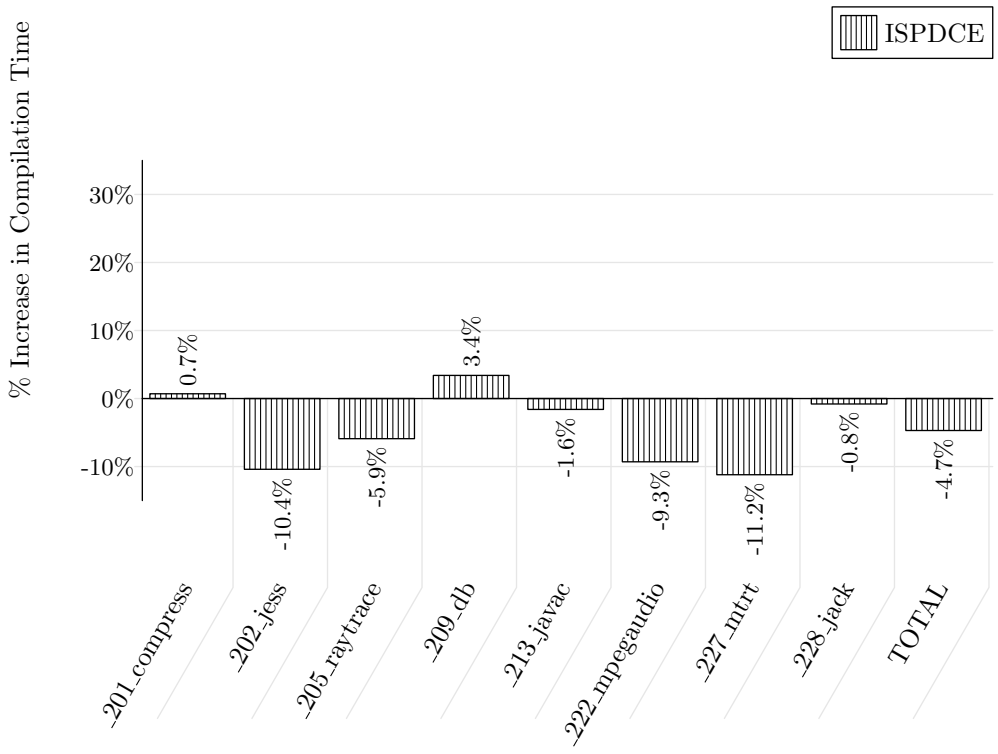


Figure 7.7: % Increase in Compilation Time (DCE phase only): PDCE (as baseline) vs. ISPDCE

Measurements

Table 7.6 shows the compilation time for the DCE *phase only* in the JIKES RVM when DCE is performed via PDCE and ISPDCE.

Column 2 ("PDCE Time") shows the compilation time required by the PDCE algorithm.

Column 3 ("ISPDCE Time") shows the compilation time required by the ISPDCE algorithm.

Column 4 ("ISPDCE Ratio") shows the value of column 3 divided by column 2, effectively using the ("PDCE Time") column as a baseline.

The measurements in Table 7.6 are depicted graphically in Figure 7.7.

Note: Unlike, the corresponding experiment for ISPRE (in Section 7.1.3) which used the default LCM implementation provided by GCC as the baseline, the *ad hoc* DCE implementation in JIKES RVM 2.4.5 is not suitable for timing comparison since it is part of a phase which performs other tasks.

Analysis

The purpose of this experiment is to evaluate the compilation time required for the DCE *phase only* under the effect of DCE via PDCE and ISPDCE.

In 6 out of 8 cases (with the exceptions of `_201_compress` and `_209_db`), DCE via ISPDCE was faster than via PDCE. Overall, ISPDCE is 4.7 % faster than PDCE. This is of important note since ISPDCE is a *speculative* optimization which is taking the execution frequency profile of each method into account when optimizing it. PDCE, however, ignores all execution frequency profile information.

By this we demonstrate that the concept of isothermality can make speculative optimizations very affordable. Just as ISPRE performed approximately 9-times faster than SPRE but gave comparable optimization, we will show that better optimization is provided by ISPDCE than PDCE, while having the nearly the same compilation time requirements.

This frugality arises from the fact that ISPDCE does not perform global bit-vector analyses. Rather, it performs local analyses in the hot region. Furthermore, since the hot region is restricted in topology (as defined in Chapter 5) such that there is at most one back-edge (namely the re-iteration edge), the local bit-vector analyses are guaranteed to coverage after a maximum of one iteration. PDCE, by comparison, has *global* bit-vector analyses that have no guarantee of convergence after one iteration.

By the same argument, it can even be seen that ISPDCE is more frugal than even ISPRE. Even though ISPRE has local bit-vector analyses, the topology of the hot regions that it optimizes are general, possibly possessing multiple nested back-edges. Therefore, its local bit-vector analyses are not guaranteed to terminate as quickly.

7.2.4 Compilation Time: All Phases

Table 7.7 shows the total compilation time required by the JIKES RVM 2.4.5 when each of the following 2 PRE algorithms are used: PDCE, ISPDCE. For comparison, the total compilation time required by JIKES RVM 2.4.5 when neither algorithm is used is also shown.

Benchmark	Default	PDCE		ISPDCE	
	Time	Time	Ratio	Time	Ratio
_201_compress	5.784	6.065	(1.049)	6.072	(1.050)
_202_jess	8.429	8.683	(1.030)	8.691	(1.031)
_205_raytrace	8.152	8.396	(1.030)	8.319	(1.021)
_209_db	7.596	7.833	(1.031)	7.869	(1.036)
_213_javac	13.029	13.159	(1.010)	13.241	(1.016)
_222_mpegaudio	9.976	10.343	(1.037)	10.304	(1.033)
_227_mtrt	6.762	7.283	(1.077)	7.087	(1.048)
_228_jack	8.794	9.113	(1.036)	9.028	(1.027)
TOTAL	68.521	70.875	(1.034)	70.611	(1.030)

Table 7.7: Compilation Time (in seconds) for all phases: Default JIKES RVM (as baseline) vs. PDCE vs. ISPDCE

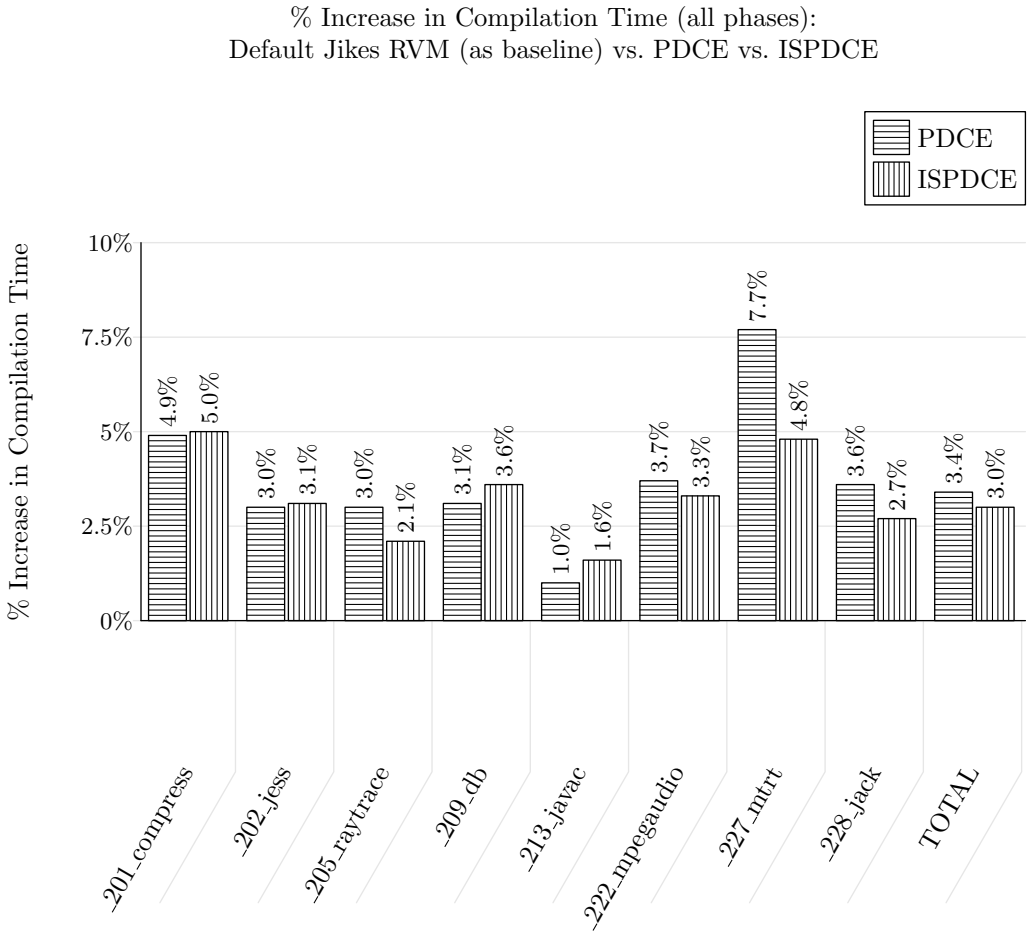


Figure 7.8: % Increase in Compilation Time (all phases): Default JIKES RVM (as baseline) vs. PDCE vs. ISPDCE

Measurements

Column 2 ("Default Time") shows the time required for compilation with an unmodified JIKES RVM. The values in column 2 establish a baseline.

Column 3 ("PDCE Time") and Column 5 ("ISPDCE Time") show the overall compilation time required when the PDCE and ISPDCE algorithms are used, respectively. Column 4 ("PDCE Ratio") and Column 6 ("ISPDCE Ratio") each show the value of the previous column divided by the baseline.

The measurements in Table 7.7 are depicted graphically in Figure 7.8.

Analysis

The purpose of this experiment is to evaluate the total compilation time required under the effect of DCE via PDCE and ISPDCE, as well as by the default JIKES RVM 2.4.5.

Table 7.7 (and Figure 7.7) show that on 4 (of 8) benchmarks ISPDCE was faster than PDCE, namely `_205_raytrace`, `_222_mpegaudio`, `_227_mtrt`, and `_228_jack`.

The percentage increase in overall compilation time is 3.4 % for PDCE and 3.0 % for ISPDCE showing that *they have comparable overall compilation cost on average.*

7.2.5 Execution Time

Benchmark	Default	PDCE		ISPDCE	
	Time	Time	Ratio	Time	Ratio
`_201_compress`	26.080	26.420	1.013	25.776	0.988
`_202_jess`	28.168	26.601	0.944	27.625	0.981
`_205_raytrace`	23.663	23.558	0.996	22.910	0.968
`_209_db`	34.452	34.107	0.990	32.899	0.955
`_213_javac`	36.023	34.577	0.960	35.800	0.994
`_222_mpegaudio`	30.320	30.752	1.014	29.211	0.963
`_227_mtrt`	26.300	26.092	0.992	25.236	0.960
`_228_jack`	28.707	30.646	1.068	29.403	1.024
TOTAL	233.713	232.753	0.996	228.860	0.979

Table 7.8: Execution Time (in seconds): Default JIKES RVM (as baseline) vs. PDCE vs. ISPDCE

Measurements

Table 7.8 shows the execution time (in seconds) of the 8 benchmarks from the SPEC CPU 98 benchmark suite when each of the following 2 DCE algorithms are used: PDCE and ISPDCE.

Note: All execution times shown are for the 100 (very large) problem input set, as recommended by the SPEC Corporation—the 1/10 (small/medium) problem set is recommended for use only during compiler development.

Column 2 ("Default time") shows the execution times of each benchmark when a pristine copy of JIKES RVM 2.4.5 is used. The values in column 2 establish a baseline, which shows that both PDCE and ISPDCE improve program performance in absolute terms. The need for a baseline was previously discussed in Section 7.2.2.

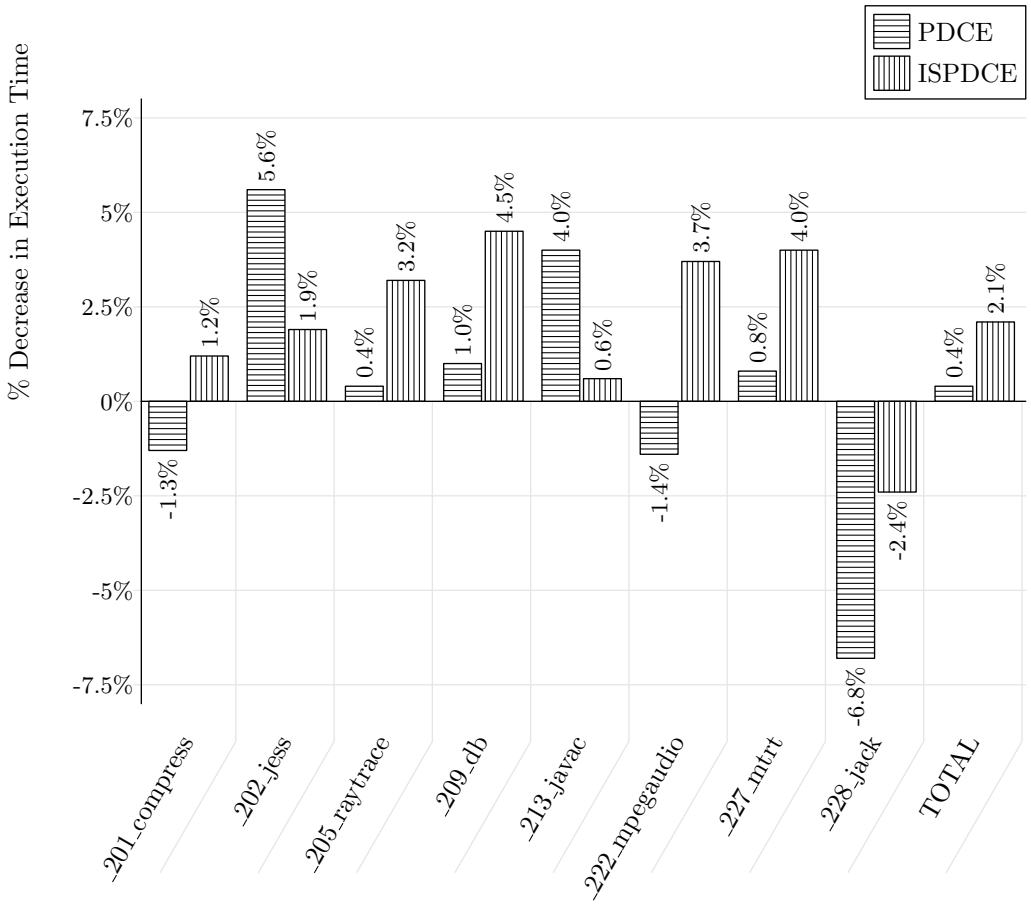


Figure 7.9: % Decrease in Execution Time: Default JIKES RVM (as baseline) vs. PDCE vs. ISPDCE

Column 3 ("PDCE Time") and Column 5 ("ISPDCE Time") show the execution times of each benchmark when DCE is performed via the PDCE and ISPDCE algorithms, respectively. Column 4 ("PDCE Ratio") and Column 6 ("ISPDCE Ratio") each show the value of the previous column divided by the baseline.

The measurements in Table 7.8 are depicted graphically in Figure 7.9.

Analysis

The aim of this experiment is to examine how well ISPDCE optimizes relative to its non-speculative competitor PDCE.

Table 7.8 contains the *most important measurements* for this series of experiments, since, in concert with the measurements presented in Table 7.6 and Table 7.7, it shows that concept of isothermality can be used to create very affordable speculative optimizations, which produce superior optimization while running as cheaply as their optimal non-speculative counterparts.

It can be seen from Table 7.8 that ISPDCE produces an overall percentage increase in program performance by 2.1% relative to PDCE at 0.4%.

In particular, ISPDCE produces better throughput on 5 out of 8 benchmarks namely _202_jess, _205_raytrace, _209_db, _222_mpegaudio, and _227_mtrt. Although ISPDCE also produces better optimization than PDCE on the benchmark _228_jack, we do not count it as an improvement over PDCE, since both PDCE and ISPDCE, in the case of _228_jack, actually decrease throughput, relative to the baseline.

It is interesting to note why _202_jess and _213_javac may not optimized by ISPDCE in a manner superior to PDCE. _202_jess is an interpreter for a set of rules describing an "expert system". _213_javac is a compiler for the Java programming language. Both compilers and interpreters are characterized by the heavy use of state machines which involves large numbers of transitions between states with little work ("the action") done at each state. We believe that, consequently, the concept of a localized hot region where the main action of the program occurs, does not exist in as defined a manner for _202_jess and _213_javac as for numerical programs, which often spin in a tight loop thereby cleanly demarcating the hot region. Consequently, PDCE which does not focus only on the hot region of the program, but takes a global view of program optimization, produces better code in these two cases.

Chapter 8

Conclusions & Future Work

8.1 Conclusions

This dissertation has demonstrated that:

> Speculative optimizations can be performed efficiently.

We achieved efficiency by using *isothermality*—the division of the CFG of a program into hot parts (frequently executed) and cold parts (infrequently executed)—as a way to focus the attention of speculative optimizations on the frequently executed parts of a program.

We proved experimentally that isothermality can approximate a very computationally intensive compiler optimization called SPRE, which removes redundant computations from a program. The algorithm that we invented, called ISPRE, since it uses the concept of isothermality, **executes nine times faster than** SPRE but **optimizes the target program competitively**, sometimes even better.

We also demonstrated the power of the concept of isothermality by constructing a speculative version of a classic compiler optimization called PDCE, which removes computations whose results are not used. In contrast to other versions of SPDCE, our version uses just two bit-vector analyses. Experimental validation showed that ISPDCE **retains a running time competitive with non-speculative** PDCE, but **offers superior optimization**.

We have shown, through the process of construction of ISPRE and ISPDCE, that the concept of **isothermality enables the derivation of speculative optimizations from their non-speculative archetypes, in an easy, straightforward manner.** Indeed, our account of the previous work done by the programming language community on speculative optimizations has shown that most competitive approaches are very complicated—requiring multiple bit-vector analyses, predicated instructions, SEME and MEME region derivations, and even the construction and optimization of flow networks.

The simplicity of our isothermal optimizations affords correct, bug-free implementations making them contenders for inclusion in the industrial-strength compilers, especially JIT compilers, where efficient algorithms are of paramount importance.

8.2 Future Work

8.2.1 Isothermality: A Framework for Designing Speculative Optimizations

The optimization algorithms developed in this thesis, namely ISPRE and ISPDCE, were developed using the concept of isothermality as a starting point. Indeed, both algorithms commence with the division of the CFG into hot and cold regions.

Thereafter, however, the developmental procedure was *ad hoc*, where conscious design decisions had to be taken which best preserved the essence and effect of the original non-isothermal algorithm. The reader will recall how the prototype of ISPDCE (the direct analogue of ISPRE) failed to preserve program behaviour—versioning of the hot region was necessary to perform assignment motion correctly—giving rise to R3PDE.

This motivates the development of a framework for the automatic conversion of non-isothermal speculative optimizations to use isothermality, such that the design process (after the initial CFG division) is well defined. We are not so naïve as to expect automation in a pure sense: via a program or algorithm. We expect, rather, to develop a set of guidelines which simplifies the task of a person converting an optimization to use isothermality.

This is a worthy pursuit because it renders isothermality as not only a simplifying concept but also as a *methodology* for thinking about speculative program optimization.

8.2.2 Inter-Procedural/Inter-Modular Optimization

At present, ISPRE and ISPDCE work on the CFG of a *single* procedure. That is, they rely on the compiler/VM to create aggregate procedures, via inlining, which can then be optimized isothermally.

The ability of the compiler/VM to inline frequently called procedures into their callers may seem to preclude the need to develop an inter-procedural version of ISPRE and ISPDCE. However, consider advanced VMs which have the ability to identify frequently executed traces—which extend *across* procedures.

The identification of a hot program trace by the VM effectively identifies an inter-procedural (or even inter-modular) hot region, just awaiting optimization via an isothermal variant of the above algorithms.

Consequently, the formulation of an inter-procedural version of ISPRE and ISPDCE would be an immense contribution to the area of Whole Program Optimization (WPO).

8.2.3 Register Pressure

The ISPDCE and ISPRE algorithms attempt to move expressions and assignments to the ingress and egress edges of the hot region.

However, those edges may be separated by an arbitrary distance from the original block in which the expression/assignment resided. Although the hot region can be expected to be small, it is not a requirement of the isothermal approach.

In fact, the non-isothermal version of ISPRE, called LCM, contains two extra analyses to prevent expressions from being hoisted higher in the flow graph than required. ISPRE does not have any such check, and it is conceivable that a particular hoisting may cause the live range of a temporary

variable to increase so dramatically that it interferes with the live ranges of many other temporary variables.

This would result in a suboptimal register allocation where most temporary variables would then be allocated to memory locations ("spilled") instead of allocated to registers. Since memory access (even in the presence of an L1 cache) is an order of magnitude slower than accessing the register file, such spillage could completely nullify any effects of redundancy elimination.

It would be beneficial therefore to create a register-pressure cognizant version of ISPRE and ISPDCE.

8.2.4 Optimizing For Code Size

The ISPRE and ISPDCE algorithms optimize the target program to reduce the number of redundant and dead computations, respectively. They do *not* attempt to minimize program size.

However, embedded systems and controllers often have storage space considerations. Additionally, most embedded systems also try to minimize their power consumption.

Consequently, the concept of isothermality should be extended, from its current focus on execution frequency, to handle arbitrary metrics such as code size and power consumption. In fact, it is probably most beneficial to improve the isothermal concept so that it can handle weighted combinations of arbitrary metrics, so that an engineer tuning an embedded system can manipulate the weights to optimize the code for the right balance of speed, size, and power consumption required.

This generalized approach should, perhaps, be called "isometric" optimization.

8.2.5 Choosing Θ

Our benchmarks have documented the application of ISPRE and ISPDCE at a particular threshold value, Θ. However, this value of Θ was chosen by hand, for the purposes of experimentation.

It would be preferable to automate the choice of Θ. In particular, the following questions arise:

1. Is it possible to choose a $\Theta \in (0, 100)$ which gives the *best* isothermal optimization possible for a given procedure and its profile data ?
2. How should the CFG of the candidate procedure be inspected in order to choose Θ optimally ?
3. In such a method of choice exists, can it be performed with negligible overhead ?

8.2.6 Dynamic Instruction Counts

While the benchmark results show conclusively that ISPRE and ISPDCE optimize programs competitively, it would be very instructive to run the benchmarks in a cycle-accurate simulator to discover the precise number of instructions and cycles that are saved.

In Summary

The items of work previously enumerated can be expected to further improve both ISPDCE and ISPRE, this dissertation's two elegantly simple algorithms which optimize programs speculatively while maintaining low overheads in terms of both time and space complexity.

Appendix A

Additional Benchmark Results

A.1 Execution Time

Table A.1 shows the execution times measured when *only* the PRE optimization runs.

Benchmark	LCM	SPRE		ISPRE	
	Time	Time	Ratio	Time	Ratio
		CINT			
164.gzip	251.983	248.194	(0.985)	248.825	(0.987)
175.vpr	222.004	223.324	(1.006)	221.124	(0.996)
176.gcc	109.634	109.998	(1.003)	110.089	(1.004)
181.mcf	176.166	159.388	(0.905)	159.388	(0.905)
186.crafty	156.561	157.201	(1.004)	156.305	(0.998)
197.parser	288.209	300.585	(1.043)	291.647	(1.012)
253.perlbmk	255.779	257.744	(1.008)	261.442	(1.022)
254.gap	161.525	161.981	(1.003)	160.156	(0.992)
255.vortex	180.478	183.489	(1.017)	183.828	(1.019)
256.bzip2	257.110	257.189	(1.000)	257.984	(1.003)
300.twolf	326.859	342.910	(1.049)	341.451	(1.045)
		CFP			
168.wupwise	239.964	240.735	(1.003)	241.217	(1.005)
171.swim	279.316	283.271	(1.014)	281.293	(1.007)
172.mgrid	913.679	913.179	(0.999)	916.127	(1.003)
173.applu	627.239	624.604	(0.996)	618.015	(0.985)
177.mesa	249.073	251.822	(1.011)	259.446	(1.042)
178.galgel	779.793	781.382	(1.002)	783.288	(1.004)
179.art	257.312	255.532	(0.993)	258.904	(1.006)
183.equake	175.372	176.010	(1.004)	176.649	(1.007)
187.facerec	331.581	332.286	(1.002)	332.286	(1.002)
188.ammp	464.104	461.140	(0.994)	470.091	(1.013)

continues on next page...

continued from previous page...

Benchmark	LCM	SPRE		ISPRE	
	Time	Time	Ratio	Time	Ratio
`189.lucas`	281.013	283.817	(1.010)	281.332	(1.001)
`191.fma3d`	378.969	426.340	(1.125)	426.340	(1.125)
`301.apsi`	599.696	599.124	(0.999)	600.698	(1.002)
TOTAL	7963.419	8031.247	(1.009)	8037.927	(1.009)

Table A.1: Execution Time (in seconds): LCM vs. SPRE vs. ISPRE

The data in Table A.1 suggest that phases in the compiler that run prior to PRE—such as web creation and Global Value Numbering (GVN)—enable PRE to be more efficacious.

This is because PRE is "syntactic" in its operation and therefore does not differentiate between unrelated uses of same variable. "Semantic" algorithms, such as Global Value Numbering (GVN), can differentiate between unrelated uses of variables and rename them appropriately, thereby giving PRE a more refined program to optimized, one where spurious interactions between expressions have been removed.

It should also be noted that rematerialization of computations later in the compilation also help PRE in its optimization effort by preventing redundancy eliminations which excessively increase register pressure.

Bibliography

[ABD+97] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone? Technical Report 016, Digital Equipment Corporation Systems Research Center, Palo Alto, California, 1997. 19, 21

[ABL97] G. Ammons, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 85–96, 1997. 19

[AEL02] M. Alpuente, S. Escobar, and S. Lucas. Removing redundant arguments of functions. In *AMAST '02: Proceedings of the 2002 Conference on Algebraic Methodology and Software Technology*, pages 117–131, 2002. 99

[AEL07] M. Alpuente, S. Escobar, and S. Lucas. Removing redundant arguments automatically. *Theory and Practice of Logic Programming*, 7(1-2):3–35, 2007. 99

[AJU77] A. Aho, S. Johnson, and J. Ullman. Code generation for expressions with common subexpressions. *Journal of the Association for Computing Machinery*, 24(1):146–160, 1977. 92

[AK02] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002. 1, 3

[Amd67] G. Amdahl. Validity of the single processor approach to achieving large scale computing capability. In *AFIPS '67: Proceedings of the 1967 Conference of the American Federation of Information Processing Societies*, pages 483–485, 1967. 40

[AR01] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 168–179, 2001. 19

[ASD95] G. Agrawal, J. Saltz, and R. Das. Interprocedural partial redundancy elimination and its application to distributed memory compilation. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 258–269, 1995. 93

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986. 12, 34, 46, 87, 95

[BC94] P. Briggs and K. Cooper. Effective partial redundancy elimination. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 159–170, 1994. 94

[BCDG00] S. Berardi, M. Coppo, F. Damiani, and P. Giannini. Type-based useless-code elimination for functional programs. In *SAIG '00: Proceedings of the 2000 International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 172–189, 2000. 99

[BCF+99] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapenõ dynamic optimizing compiler for Java. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 129–141, 1999. 9

[BCS97] P. Briggs, K. Cooper, and L. Simpson. Value numbering. *Software—Practice and Experience*, 27(6):701–724, 1997. 94

[BDB00] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000. 25

[BG97] R. Bodík and R. Gupta. Partial dead code elimination using slicing transformations. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 159–170, 1997. 96

[BGS98] R. Bodík, R. Gupta, and M. Soffa. Complete removal of redundant computations. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming Language Design and Implementation*, pages 1–14, 1998. 89

[BL94] T. Ball and J. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994. 19

[BL96] T. Ball and J. Larus. Efficient path profiling. In *Micro-29: Proceedings of the IEEE/ACM 1996 International Symposium on Microarchitecture*, pages 46–57, 1996. 19

[Bos06] S. Bosscher. Re: Corrupted profile information. Email from `stevenb.gcc@gmail.com`, January 2006. Personal Communication. 103

[Bre69] M. Breuer. Generation of optimal code for expressions via factorization. *Communications of the Association for Computing Machinery*, 12(6):333–340, 1969. 34, 87

[Bro04] D. Bronnikov. A practical adoption of partial redundancy elimination. *SIGPLAN Notices*, 39(8):49–53, 2004. 92

[BS02] J. Butts and G. Sohi. Dynamic dead-instruction detection and elimination. In *ASPLOS '02: Proceedings of the 2002 International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–210, 2002. 100

[CCK+97] F. Chow, S. Chan, R. Kennedy, S. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on SSA form. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming Language Design and Implementation*, pages 273–286, 1997. 92, 94

[CG97] B. Cherkassky and A. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997. 38, 111

[CGX03] Q. Cai, L. Gao, and J. Xue. Region-based partial dead code elimination on predicated code. In *CC '03: Proceedings of the 2003 International Conference on Compiler Construction*, pages 150–166, 2003. 96

[Cho83] F. Chow. *A portable machine-independent global optimizer—design and measurements*. PhD thesis, Computer Systems Laboratory, Stanford University, 1983. 88

[CL96] R. Cohn and G. Lowney. Hot cold optimization of large Windows/NT applications. In *MICRO 29: Proceedings of the 1999 Annual ACM/IEEE International Symposium on Microarchitecture*, pages 80–89, 1996. 101

[Cli95] C. Click. Global code motion/Global value numbering. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 246–257, 1995. 94

[Coc70] J. Cocke. Global common subexpression elimination. In *Proceedings of a(n unlisted) symposium on compiler optimization*, pages 20–24, 1970. 34, 87

[CS95] K. Cooper and L. Simpson. Value-driven code motion. Technical Report CRPC-TR95637-S, Center For Research on Parallel Computation, Rice University, October 1995. 94

[CSV01] K. Cooper, L. Simpson, and C. Vick. Operator strength reduction. *ACM Transactions on Programming Languages and Systems*, 23(5):603–625, 2001. 93

[CX03] Q. Cai and J. Xue. Optimal and efficient speculation-based partial redundancy elimination. In *CGO '03: Proceedings of the ACM/IEEE 2003 Symposium on Code Generation and Optimization*, pages 91–104, 2003. 36, 38, 90, 91

[DD95] V. Dhaneshwar and D. Dhamdhere. Strength reduction of large expressions. *Journal of Programming Languages*, 3(2):95–120, 1995. 95

[Dha82a] D. Dhamdhere. A composite hoisting strength reduction transformation for global program optimization: Part I. *International Journal of Computer Mathematics*, 11(1):21–41, 1982. 95

[Dha82b] D. Dhamdhere. A composite hoisting strength reduction transformation for global program optimization: Part II. *International Journal of Computer Mathematics*, 11(2):111–126, 1982. 95

[Dha88] D. Dhamdhere. A fast algorithm for code movement optimisation. *SIGPLAN Notices*, 23(10):172–180, 1988. 88

[Dha89] D. Dhamdhere. A new algorithm for composite hoisting and strength reduction. *International Journal of Computer Mathematics*, 27:1–14, 1989. 95

[Dha91] D. Dhamdhere. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, 1991. 89, 98

[Dha02] D. Dhamdhere. E-path-PRE: Partial redundancy elimination made easy. *SIGPLAN Notices*, 37(8):53–65, 2002. 92

[DP90] D. Dhamdhere and H. Patil. An efficient algorithm for bi-directional data flow analysis. Technical Report TR-016-90, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, India, 1990. 92

[DP93] D. Dhamdhere and H. Patil. An elimination algorithm for bidirectional data flow problems using edge placement. *ACM Transactions on Programming Languages and Systems*, 15(2):312–336, 1993. 88

[DS88] K. Drechsler and M. 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, 1988. 93

[DS93] K. Drechsler and M. Stadel. A variation of Knoop, Ruthing, and Steffen's Lazy Code Motion. *SIGPLAN Notices*, 28(5):29–38, 1993. 88

[FE04] M. Fernandez and R. Espasa. Link-time path-sensitive memory redundancy elimination. In *HPCA '04: Proceedings of the 2004 International Symposium on High Performance Computer Architecture*, pages 300–310, 2004. 93

[FKCX94] L. Feigen, D. Klappholz, R. Casazza, and X. Xue. The revival transformation. In *POPL '94: Proceedings of the 1994 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 421–434, 1994. 98

[FSF] GNU Compiler Collection. `http://gcc.gnu.org`. Last accessed August 2007. 9

[GB99] R. Gupta and R. Bodík. Register pressure sensitive redundancy elimination. In *CC '99: Proceedings of the 1999 International Conference on Compiler Construction*, pages 107–121, 1999. 88

[GBF97] R. Gupta, D. Berson, and J. Fang. Path profile guided partial dead code elimination using predication. In *PACT '97: Proceedings of the 6th International Conference on Parallel Architectures and Compilation Techniques*, pages 102–115, 1997. 96, 97

[GBF98] R. Gupta, D. Berson, and J. Fang. Path profile guided partial redundancy elimination using speculation. In *ICCL '98: Proceedings of the 1998 IEEE International Conference on Computer Languages*, pages 230–239, 1998. 38, 90, 91

[GCC07] GCC 4.0 release series: Changes, new features and fixes. Webpage, `http://gcc.gnu.org/gcc-4.0/changes.html`, July 2007. Last accessed August 2007. 112

[Gol03] A. Goldberg. HIPR algorithm. `http://www.avglab.com/andrew/soft.html`, 2003. Last accessed August 2007. 111

[Hai98] M. Hailperin. Cost-optimal code motion. *ACM Transactions on Programming Languages and Systems*, 20(6):1297–1322, 1998. 94

[Hec77] M. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Publishers, B.V., Amsterdam, 1977. 56, 85

[HH97] R. N. Horspool and H. Ho. Partial redundancy elimination driven by a cost-benefit analysis. In *ICCSSE '97: Proceedings of the 1997 Israeli Conference on Computer-Based Systems and Software Engineering*, pages 111–118, 1997. 36, 38, 90, 91

[HHR95] R. Hank, W. Hwu, and B. Rau. Region-based compilation: an introduction and motivation. In *MICRO 28: Proceedings of the 1995 Annual ACM/IEEE International Symposium on Microarchitecture*, pages 158–168, 1995. 101, 102

[HNW+01] A. Hosking, N. Nystrom, D. Whitlock, Q. Cutts, and A. Diwan. Partial redundancy elimination for access path expressions. *Software—Practice and Experience*, 31(6):577–600, 2001. 93

[HPS06] R. N. Horspool, D. Pereira, and B. Scholz. Fast profile-based partial redundancy elimination. In *JMLC '06: Proceedings of the 2006 Joint Modular Languages Conference*, pages 362–376, 2006. 33

[Int90] *386(TM) DX Microprocessor Programmer's Reference Manual*. Intel Press, 1990. 45

[JSGB00] B. Joy, G. Steele, J. Gosling, and G. Bracha. *The Java(TM) Language Specification (2nd Edition)*. The Java Series. Prentice Hall, 2000. 39, 45

[KCJ00] J. Knoop, J. Collard, and R. Ju. Partial redundancy elimination on predicated code. In *SAS '00: Proceedings of the 2000 International Static Analysis Symposium*, pages 260–279, 2000. 89

[KCL+99] R. Kennedy, S. Chan, S. Liu, R. Lo, P. Tu, and F. Chow. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, 1999. 92, 94

[KD99] U. Khedker and D. Dhamdhere. Bidirectional data flow analysis: Myth and reality. *SIGPLAN Notices*, 34(6):47–57, 1999. 92

[Ken72] K. Kennedy. Safety of code motion. *International Journal of Computer Mathematics, Section A*, 3(2-3):117–130, 1972. 92

[KKN04] M. Kawahito, H. Komatsu, and T. Nakatani. Partial redundancy elimination for access expressions by speculative code motion. *Software—Practice and Experience*, 34(11):1065–1090, 2004. 93

[KKS98] J. Knoop, D. Koschützki, and B. Steffen. Basic-block graphs: Living dinosaurs? In *CC '98: Proceedings of the 1998 International Conference on Compiler Construction*, pages 65–79, 1998. 82

[KM97] J. Knoop and E. Mehofer. A powerful and flexible approach for distribution assignment placement. In *Proceedings of the 14th Annual Workshop of the GI-FG 2.1.4 "Alternative Konzepte für Sprachen und Rechner"*, Bericht Nr. 9712, pages 77–84, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, Germany, 1997. 99

[Kno96] J. Knoop. Partial dead code elimination for parallel programs. In *Euro-Par '96: Proceedings of the 1996 European Conference on Parallel Processing*, pages 441–450, 1996. 99

[Kno98] J. Knoop. Eliminating partially dead code in explicitly parallel programs. *Theoretical Computer Science*, 196(1–2):365–393, 1998. 99

[Kob00] N. Kobayashi. Type-based useless variable elimination. In *PEPM '00: Proceedings of 2000 Conference on Partial Evaluation and Semantics-Based Program Manipulation*, pages 84–93, 2000. 99

[KR99] J. Knoop and O. Rüthing. The impact of paradigm shifts on safety and optimality of code motion. In *Joint Proceedings of the 16th Annual Workshop of the GI-FG 2.1.4 "Alternative Konzepte für Sprachen und Rechner" and the 11th Workshop of the GI-FG 3.6.2 "German ENCRESS (European Network of Clubs for Reliability and Safety of Software-Intensive Systems)" on "Sicherheit und Zuverlässigkeit software-basierter Systeme"*, Bericht ISTec-A-367, pages 6–20, Institut fur Sicherheitstechnologie (ISTec) GmbH, Abteilung Leittechnik, Physikzentrum Bad Honnef, Germany, 1999. 92

[KR04] J. Koppanalil and E. Rotenberg. A simple mechanism for detecting ineffectual instructions in slipstream processors. *IEEE Transactions on Computers*, 53(4):399–413, 2004. 100

[KRS92] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming Language Design and Implementation*, pages 224–234, 1992. 57, 88

[KRS93] J. Knoop, O. Rüthing, and B. Steffen. Lazy strength reduction. *Journal of Programming Languages*, 1(1):71–91, 1993. 94

[KRS94a] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, 1994. 88, 92

[KRS94b] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 147–158, 1994. 69, 86, 95, 96, 97, 99

[KRS95] J. Knoop, O. Rüthing, and B. Steffen. The power of assignment motion. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 233–245, 1995. 89, 98

[KRS98] J. Knoop, O. Rüthing, and B. Steffen. Code motion and code placement: Just synonyms? In *ESOP '98: Proceedings of the 1998 European Symposium On Programming*, pages 154–169, 1998. 94

[KRS99] J. Knoop, O. Rüthing, and B. Steffen. Expansion-based removal of semantic partial redundancies. In *CC '99: Proceedings of the 1999 International Conference on Compiler Construction*, pages 91–106, 1999. 89

[KU77] J. Kam and J. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:309–317, 1977. 41, 45, 48, 51, 67, 69

[KW95] P. Kolte and M. Wolfe. Elimination of redundant array subscript range checks. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 270–278, 1995. 93

[Lam00] P. Lam. Common subexpression removal in Soot. Technical Report 2000-1, Sable Research Group, School Of Computer Science, McGill University, 2000. 88

[LB92] J. Larus and T. Ball. Rewriting executable files to measure program behavior. Technical Report CS-TR-92-1083, Department of Computer Science, University of Madison, Wisconsin, 1992. 19

[LCK+98] R. Lo, F. Chow, R. Kennedy, S. Liu, and P. Tu. Register promotion by sparse partial redundancy elimination of loads and stores. *ACM SIGPLAN Notices*, 33(5):26–37, 1998. 93

[LS99] Y. Liu and S. Stoller. Eliminating dead code on recursive data. In *SAS '99: Proceedings of the 1999 Static Analysis Symposium*, pages 211–231, 1999. 99

[MJ81] S. Muchnik and N. Jones, editors. *Program Flow Analysis: Theory and Applications*, chapter "Interprocedural Elimination of Partial Redundancies". Prentice Hall, 1981. 34, 88

[MMR95] S. Masticola, T. Marlowe, and B. Ryder. Lattice frameworks for multisource and bidirectional data flow problems. *ACM Transactions on Programming Languages and Systems*, 17(5):777–803, 1995. 92

[Mor83] E. Morel. Data flow analysis and global optimization. In *Methods and tools for compiler construction: An advanced course*. Cambridge University Press, 1983. 88

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the Association for Computing Machinery*, 22(2):96–103, 1979. 34, 57, 88, 98

[MRF97] M. Martin, A. Roth, and C. Fischer. Exploiting dead value information. In *Micro-30: Proceedings of the 1997 International Symposium on Microarchitecture*, pages 125–135, 1997. 101

[Muc97] S. Muchnik. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. 9

[OH05] R. Odaira and K. Hiraki. Sentinel PRE: Hoisting beyond exception dependency with dynamic deoptimization. In *CGO '05: Proceedings of the 2005 International Symposium on Code Generation and Optimization*, pages 328–338, 2005. 93

[ORC] ORC: Open Research Compiler for the Itanium^{tm} Processor Family. `http://ipf-orc.sourceforge.net/ORC-MICRO34-tutorial.pdf`. Last accessed August 2007. 9

[PH90] K. Pettis and R. Hansen. Profile guided code positioning. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 16–27, 1990. 101

[PPC93] *The POWERPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann, 1993. 45

[PSS03] V. Paleri, Y. Srikant, and P. Shankar. Partial redundancy elimination: a simple, pragmatic, and provably correct algorithm. *Science of Computer Programming*, 48(1):1–20, 2003. 92

[RKS00a] O. Rüthing, J. Knoop, and B. Steffen. Code-size sensitive partial redundancy elimination. In *Proceedings of the 17th Annual Workshop of the GI-FG 2.1.4 "Programmiersprachen und Rechenkonzepte"*, Bericht Nr. 2007, pages 25–35. Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, Germany, 2000. 88

[RKS00b] O. Rüthing, J. Knoop, and B. Steffen. Sparse code motion. In *POPL '00: Proceedings of the ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, pages 170–183, 2000. 88

[Rot99] E. Rotenberg. Exploiting large ineffectual instruction sequences. Technical Report CESR-TR-02-3, Center for Embedded Systems Research, North Carolina State University, 1999. 100

[Rüt98] O. Rüthing. Optimal code motion in the presence of large expressions. In *ICCL '98: Proceedings of the 1998 International Conference on Computer Languages*, pages 216–226, 1998. 93

[SHK04] B. Scholz, N. Horspool, and J. Knoop. Optimizing for space and time usage with speculative partial redundancy elimination. In *LCTES '04: Proceedings of ACM SIGPLAN/SIGBED 2004 Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 221–230, 2004. 36, 38, 91

[Sim98] L. Simpson. Value-driven redundancy elimination. Technical Report TR98-308, Center For Research on Parallel Computation, Rice University, 1998. 94

[SKR90] B. Steffen, J. Knoop, and O. Rüthing. The value flow graph: A program representation for optimal program transformations. In *ESOP '03: Proceedings of the 2003 European Symposium On Programming*, pages 389–405, 1990. 94

[SKR91a] B. Steffen, J. Knoop, and O. Rüthing. Efficient code motion and an adaption to strength reduction. In *TAPSOFT '91: Proceedings of the 1991 International Joint Conference on Theory And Practice of Software Development*, pages 394–415, 1991. 94

[SKR91b] B. Steffen, J. Knoop, and O. Rüthing. Optimal code motion within flow graphs: A practical approach. Technical Report Bericht Nr. 9102, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität, Germany, 1991. 94

[SMH03a] B. Scholz, E. Mehofer, and R. N. Horspool. Partial redundancy elimination with predication techniques. In *Euro-Par '03: Proceedings of 2003 International European Conference on Parallel Processing*, pages 242–250, 2003. 89

[SMH03b] B. Scholz, E. Mehofer, and R. N. Horspool. Predicated partial redundancy elimination using a cost analysis. *Parallel Processing Letters*, 13(4):525–536, 2003. 89

[Sor89] A. Sorkin. Some comments on "A solution to a problem with Morel and Renvoise's 'Global optimization by suppression of partial redundancies' ". *ACM Transactions on Programming Languages and Systems*, 11(4):666–668, 1989. 93

[Sor96] A. Sorkin. Some comments on Morel and Renvoise's "Global optimization by suppression of partial redundancies". *SIGPLAN Notices*, 31(12):69–72, 1996. 92

[Sto77] H. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, 3(1):85–93, 1977. 36, 91

[SYN03] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for a Java Just-In-Time compiler. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 312–323, 2003. 102

[TBR+06] S. Triantafyllis, M. Bridges, E. Raman, G. Ottoni, and D. August. A framework for unrestricted whole-program optimization. *SIGPLAN Notices*, 41(6):61–71, 2006. 22

[Ull72] J. Ullman. A fast algorithm for the elimination of common subexpressions. *Foundations of Computer Science*, pages 161–176, 1972. 87

[Ull73] J. Ullman. Fast algorithms for the elimination of common subexpressions. *Acta Informatica*, 2:191–213, 1973. 87

[vD04] T. van Drunen. *Partial Redundancy Elimination for Global Value Numbering*. PhD thesis, Department of Computer Science, Purdue University, 2004. 94

[vDH04a] T. van Drunen and A. Hosking. Anticipation-based partial redundancy elimination for static single assignment form. *Software—Practice and Experience*, 34(15):1413–1439, 2004. 92

[vDH04b] T. van Drunen and A. Hosking. Value-based partial redundancy elimination. In *CC '04: Proceedings of the 2004 International Conference on Compiler Construction*, pages 167–184, 2004. 94

[VS03] K. Vaswani and Y. Srikant. Dynamic recompilation and profile-guided optimizations for a .NET JIT compiler. *IEEE Proceedings—Software*, 150(5):296–302, 2003. 4

[Wad88] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP '88: Proceedings of the 1988 European Symposium on Programming*, pages 344–358, 1988. 2

[WBP00] T. Way, B. Breech, and L. Pollock. Region formation analysis with demand-driven inlining for region-based optimization. In *PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, pages 24–36, 2000. 102

[Wha00] J. Whaley. A portable sampling-based profiler for Java virtual machines. In *Proceedings of the ACM 2000 Java Grande Conference*, pages 78–87, 2000. 21

[Wha01] J. Whaley. Partial method compilation using dynamic profile information. In *OOPSLA '01: Proceedings of the 2001 ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 166–179, 2001. 102

[WL94] Y. Wu and J. Larus. Static branch frequency and program profile analysis. In *MICRO 27: Proceedings of the 1994 International Symposium on Microarchitecture*, pages 1–11, 1994. 14

[Wol99] M. Wolfe. Partial redundancy elimination is not bidirectional. *SIGPLAN Notices*, 34(6):43–46, 1999. 88

[WS99] M. Wand and I. Siveroni. Constraint systems for useless variable elimination. In *POPL '99: Proceedings of the ACM 1999 Symposium on Principles of Programming Languages*, pages 291–302, 1999. 99

[Xi98] H. Xi. Dead code elimination through dependent types. In *PADL '99: Proceedings of the 1999 International Workshop on Practical Aspects of Declarative Languages*, pages 228–242, 1998. 99

[ZA07] P. Zhao and J. Amaral. Ablego: a function outlining and partial inlining framework. *Software—Practice and Experience*, 37(5):465–491, 2007. 102

List of Acronyms

AOS	Adaptive Optimization System
AST	Abstract Syntax Tree
CFG	Control Flow Graph
CIL	Common Intermediate Language Previously called MSIL—Microsoft Intermediate Language
CISC	Complex Instruction Set Computer
CPO	Continuous Program Optimization
CSE	Common Subexpression Elimination
DCE	Dead Code Elimination
.NET	Microsoft .NET Framework
DVI	Dead-Value Information
FORTRAN	FORmula TRANslator
GCC	GNU Compiler Collection
GCSE	Global Common Subexpression Elimination
GIMPLE	GCC SIMPLE Intermediate Language
GVN	Global Value Numbering
HIR	High-Level Intermediate Representation
HPF	High Performance Fortran
IR	Intermediate Representation
ISA	Instruction Set Architecture
ISPDCE	Isothermal Speculative Partial Dead Code Elimination
ISPRE	Isothermal Speculative Partial Redundancy Elimination
J2EE	Java 2 Enterprise Edition

J9-JVM ..... IBM J9 Virtual Machine for Java

JIT ..... Just-In-Time

JNI ..... Java Native Interface

LCM ..... Lazy Code Motion

LCSE ..... Local Common Subexpression Elimination

LICM ..... Loop Invariant Code Motion

LIR ..... Low-Level Intermediate Representation

MC-PRE ..... Minimum-Cut Partial Redundancy Elimination

MEM ..... memory location

MEME ..... Multiple Entry Multiple Exit

MIR ..... Machine-Level Intermediate Representation

ORC ..... Open Research Compiler

PA-RISC ..... Precision Architecture RISC

PDCE ..... Partial Dead Code Elimination

PDF ..... Profile Driven Feedback

PEI ..... Potentially Excepting Instruction

PFCE ..... Partial Faint Code Elimination

PRAE ..... Partially Redundant Assignment Elimination

PRE ..... Partial Redundancy Elimination

R3PDE ..... 3-Region Partial Dead Code Elimination

RAM ..... Random Access Memory

RISC ..... Reduced Instruction Set Computer

RMI ..... Remote Method Invocation

RTL ..... Register Transfer Language

Jikes RVM ... Jikes Research Virtual Machine

SEME ..... Single Entry Multiple Exit

SPDCE ..... Speculative Partial Dead Code Elimination

SPRE ..... Speculative Partial Redundancy Elimination

SR ..... Strength Reduction

SSA ..... Static Single Assignment

TOBEY ..... IBM Toronto Back-End with Yorktown

TPO ..... IBM Toronto Portable Optimizer

TR-JIT ..... IBM Testarossa JIT Compiler

UCE ..... Useless Code Elimination

UVE ..... Useless Variable Elimination

VM ..... Virtual Machine

WHIRL ..... Winning Hierarchical Intermediate Representation Language

WPO ..... Whole Program Optimization