Analysis and Application of Fourier-Motzkin Variable Elimination to Program Optimization

A thesis presented in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

at Massey University, Albany
New Zealand.

David Niedzielski

2019

Abstract

This thesis examines four of the most influential dependence analysis techniques in use by optimizing compilers: Fourier-Motzkin Variable Elimination, the Banerjee Bounds Test, the Omega Test, and the I-Test. Although the performance and effectiveness of these tests have previously been documented empirically, no in-depth analysis of how these techniques are related from a purely analytical perspective has been done. The analysis given here clarifies important aspects of the empirical results that were noted but never fully explained. A tighter bound on the performance of one of the Omega Test algorithms than was known previously is proved and a link is shown between the integer refinement technique used in the Omega Test and the well-known Frobenius Coin Problem. The application of a Fourier-Motzkin based algorithm to the elimination of redundant bound checks in Java bytecode is described. A system which incorporated this technique improved performance on the Java Grande Forum Benchmark Suite by up to 10 percent.

Niniejszą pracę dyplomową pragnę zadedykować mojemu patronowi św Judzie Tadeuszowi z wdzięczności za nieustanną opiekę oraz cierpliwość.

Contents

List of Figures

List of Tables

Part I.

Introduction

Chapter 1

Introduction

1.1. Purpose

The purpose of the first part of the thesis is to formally answer a question that has long implicitly existed in the literature of data dependence analysis in optimizing compilers: what is the relationship (if any) among two of the leading techniques in detecting data dependencies arising from array subscript expressions in nested loops?

The literature abounds with empirical studies to show the relative efficacy and performance of these tests, but this is the first work to specifically address the issue by directly comparing the various techniques from a purely analytical standpoint.

It is hoped that this work contributes to a deeper understanding of these integer programming techniques, and will lead to the invention of more general tests that do not sacrifice the efficiency of their predecessors.

1.2. Preliminaries

The amount of digital data in the world is growing at a staggering pace [14]. A study by Hilbert and Lopez attempted to estimate the amount of information which could possibly have been stored by the 12 most widely used families of analog storage technologies and the 13 most prominent families of digital memory, from paper-based advertisement to the memory chips installed on a credit card. The authors argue that the total amount of information grew from 2.6 optimally compressed exabytes in 1986 to 15.8 in 1993, over 54.5 in 2000, and to 295 optimally compressed exabytes in 2007 [46]. Advances in storage technology, proliferation of data collection agents, and a precipitous drop in the price of storing digital data means that more and more data is being collected and retained for processing. The era of "Big Data" requires a commensurate increase in the ability to process this flood of digital information in a timely manner, even as physical limitations in transistor density in VLSI hardware due to thermodynamics and quantum physics are rapidly being approached [73]. This thesis will argue that "throwing hardware at the problem" is no longer a viable option unless program performance can be optimized by fully exploiting the hardware resources available. In a sense, the pendulum has swung back: the focus of program performance improvement efforts, once centered on (often automatic) software optimization, took a backseat in the face of exponential growth in computing hardware capability. Now we see the greatest opportunity to improve program performance again shifting back to the software optimization realm.

As motivation, I begin with a necessarily cursory overview of strategies to improve program performance. For purposes of argument, let us say that program A performs better than program B if given the same input, program A computes the same output as B in a shorter time. There are subtleties in even this simplistic definition. For example, this does not necessarily imply that program A is more *efficient* than program B. Suppose for example that program A makes uses of parallel hardware whereas program B must run sequentially on a single processor. Even if the sum of the instructions executed by program A on all processors which it uses is greater than the instruction count executed by program B, program A is said to perform better if it finishes execution sooner than program B. Also, the definition does not preclude the possibility that program A is the same program as program B, but simply running on faster hardware.

It is worth noting that our definition of *optimal* is based solely on execution time. In a different context, the optimality may be defined using different metrics. An important example in mobile applications is the definition of optimality in terms of total energy utilized to solve a problem. If program A computes a result on a processor drawing 10 Watts of power in 10 seconds, then it will have consumed 100J of energy in doing so. Suppose program B computes the same result on a less powerful processor which draws 7 watts of power in 13 seconds. Program B will therefore require 91J of energy to complete the task. Even though program B took longer to compute the result, by this definition of optimality it is considered “better” than program A. Unless specifically stated otherwise, I will continue to use execution time as a metric in determining the optimality of a program.

We begin by discussing in general terms common ways in which a program’s performance may be improved, and then introduce the topic of this thesis.

1.2.1. Algorithmic efficiency

By definition, an efficient algorithm will always be preferable to an inefficient one. Selection of the optimum algorithm is thus arguably the most critical factor in optimizing program performance and consequently represents the most promising avenue for improving the performance of a program.

Perhaps slightly less obvious is that the definition of “optimum algorithm” is an elusive one. An algorithm that works best for a small dataset can fail miserably as the problem size is increased. Conversely, an algorithm whose performance is formally polynomial for sufficiently large input may perform poorly for small problem sizes. Furthermore, an algorithm that runs in polynomial time but is exponential or worse in terms of memory usage may be optimum on a high-end server but exhaust the resources of a commodity-class cloud instance.

Likewise, depending on the inter-task communication requirements, an algorithm that is designed to solve a problem by running in parallel may perform much worse in a networked cluster of machines than on a single multi-processor or multi-core system because of the network latency [6]. However, with the advent of cloud computing, such architectures are becoming more prevalent and cost-effective. Even the performance of a parallel algorithm on a single multi-core system may be inferior to that of a sequential algorithm because of the overhead in locking, bus contention, and other factors. Nonetheless, exploitation of massive amounts of loosely coupled commodity processors may be the only way to viably process enterprise-class data warehouses.

Note that while optimizing a program’s algorithm in response to a rapidly increasing problem size or each change in hardware technology may offer the most reward in terms of improving (or maintaining) performance, we must bear in mind that the human costs of development, testing, and deployment are likely to make this the most expensive approach as well. The key implication of this (and one to which I will return shortly) is that if optimizations could be made automatically, the benefits of algorithmic improvement could in some degree be achieved without the attendant costs of human intervention.

1.2.2. Faster Hardware

We have previously alluded to the fact that while an easy and inexpensive method in the past to incrementally improve program performance was to simply buy faster hardware, this is no longer a sustainable approach. The growth of the problem data set has exceeded improvements in processor speed by several orders of magnitude, and that trend will continue for the foreseeable future.

The oft-quoted Moore's Law [60] can be paraphrased as stating that the number of transistors in a dense VLSI chip doubles roughly every two years. The law can be applied to the resulting speedup in clock speeds as well. Moore's Law is based on the ability of manufacturers to continually miniaturize transistors, enabling more and more to be packed into the same sized silicon die. The law depends on a related "law" ("rule of thumb" would be more accurate) attributed to Robert Dennard and known as *Dennard Scaling* [29]. Dennard Scaling states in effect that because voltage and current are inversely proportional to transistor size and the distance between them, the power density of a silicon wafer is proportional to its area of the wafer independent of the number of transistors it contains. However, this miniaturization and speedup cannot continue indefinitely, and in fact we are already seeing Moore's Law and Dennard Scaling beginning to break down. This is especially apparent in the consumer market, where clock speeds have not increased to any significant degree in recent years. This is graphically illustrated in Figure 1.1, which shows how clock frequency/MIPs in Intel chips has grown over the years, and how that growth has lately stalled. The same phenomenon is depicted in Figure 1.2, which further shows how clock frequencies in architectures in the industry have leveled off after an initial period of rapid growth. Additionally, Figure 1.3 plots the performance of several processors relative to the VAX 11/780 over time, as measured by the SPEC benchmarks. We see that after a steep increase in performance from 1986 through 2003 (roughly a 52% improvement per year over that period), the rate of performance improvement has significantly leveled off in subsequent years. While new technologies such as three dimensional integrated circuits, molecular, optical, or quantum computing technology may ultimately offer some relief, these technologies will not arrive in time to deal with the already present and intractable glut of data overwhelming existing hardware.

There are several factors contributing to the breakdown in Moore's Law and Dennard Scaling [96].

Sub-threshold Conduction

As transistors become smaller and smaller, correspondingly smaller voltages can safely be applied to them, which in turn supports Dennard Scaling. However, at a certain critical voltage threshold (the exact voltage level depends on the semiconductor type), the delta between the on and off levels applied to the base of the transistor is too small to completely switch the transistor on or off. Hardware manufacturers can attempt to mitigate this effect by biasing the semiconductor via doping to reliably switch the transistor either on or off, but cannot effectively do both without a greater voltage differential.

Thermodynamics and Leakage Current

As insulators within the transistor become smaller and smaller, a quantum mechanical phenomenon known as *electron tunneling* (the same process behind the tunneling electron microscope) allows current to leak into the surrounding silicon substrate. This leakage causes increased power consumption and consequently heat buildup becomes an issue. To illustrate, whereas a commodity processor from 1993 had roughly 3 million transistors, modern processors have nearly 1 billion semiconductors in roughly the same footprint. If this miniaturization were to continue, processors would soon produce more heat per square centimeter than the surface of the sun [84].

RC delay

Transistors are not the only barrier to continued miniaturization and speedup. Another issue is the physical interconnection between the semiconductors. As components become smaller and smaller, the physical wires which conduct digital signals between them necessarily shrink as well. As they do, they not only tend to become less reliable, but they also start to exhibit a greater resistive-capacitive (RC) delay as well,

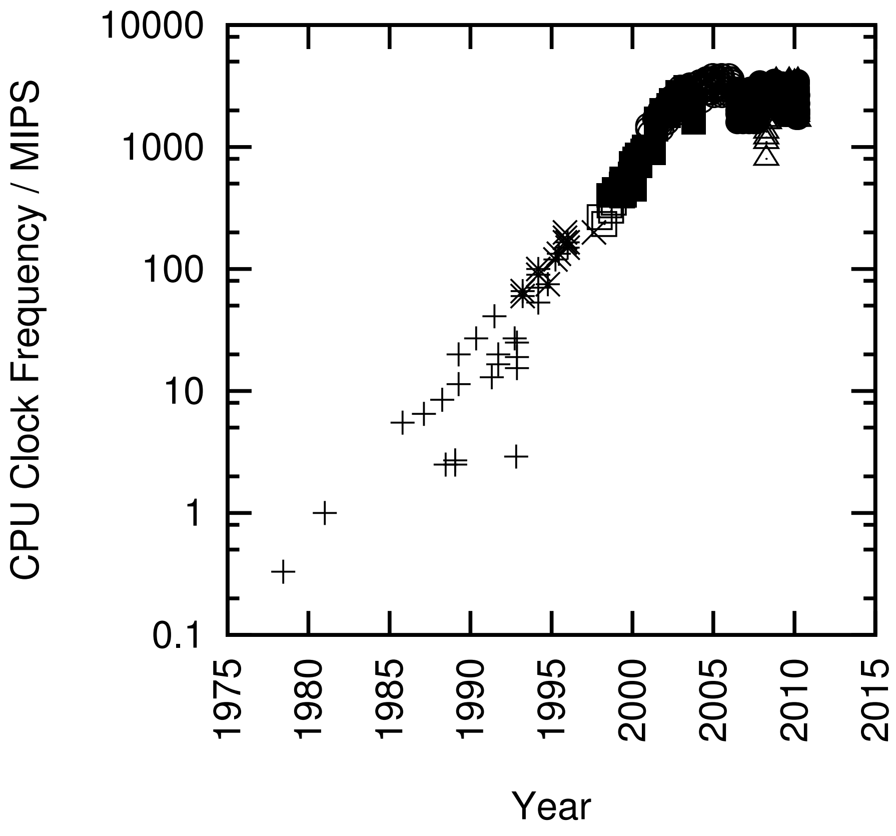

Figure 1.1.: MIPS/Clock-Frequency Trend for Intel CPUs. (Source: [58])

causing increased latency in the signal path. This is especially a problem in "long-haul" paths, where wires are used to connect components on opposite sides of the CPU chip. Engineers have attempted to mitigate the effect of the RC delay by replacing aluminum traces with copper ones, which is expensive, or by using repeaters in the circuit path, which adds to the power and heat issues already present on the CPU.

Memory Performance Limitations

Memory latency has not kept pace with improvements in CPU transistor density. One of the key issues again relates to transistor density in the DRAM chips, as the power necessary for the transistor/capacitor pairs in the billions of memory cells creates heat problems for the same reasons as outlined above.

A more fundamental issue stems from the simple fact that memory chips are generally placed on physically separate dies on the mother board due to heat and power consumption issues. The physical distance raises latency problems due to the speed of light. With a 3.33 GHz clock, a pulse of light can only travel roughly 10cm in one cycle. The problem is exacerbated when one considers that the speed of electrons traveling through a semiconductor material is currently anywhere from 3% to 30% of the speed of light, and even light through optical fiber travels at roughly 60% of the speed of light. Considering the miles of internal wiring present in modern CPUs and DRAM chips, there is simply insufficient time to communicate data from an off-chip memory store via the bus to the CPU. Engineers have typically attempted to address this issue by bringing more and more memory on-board in the form of multi-level caches, but this in a

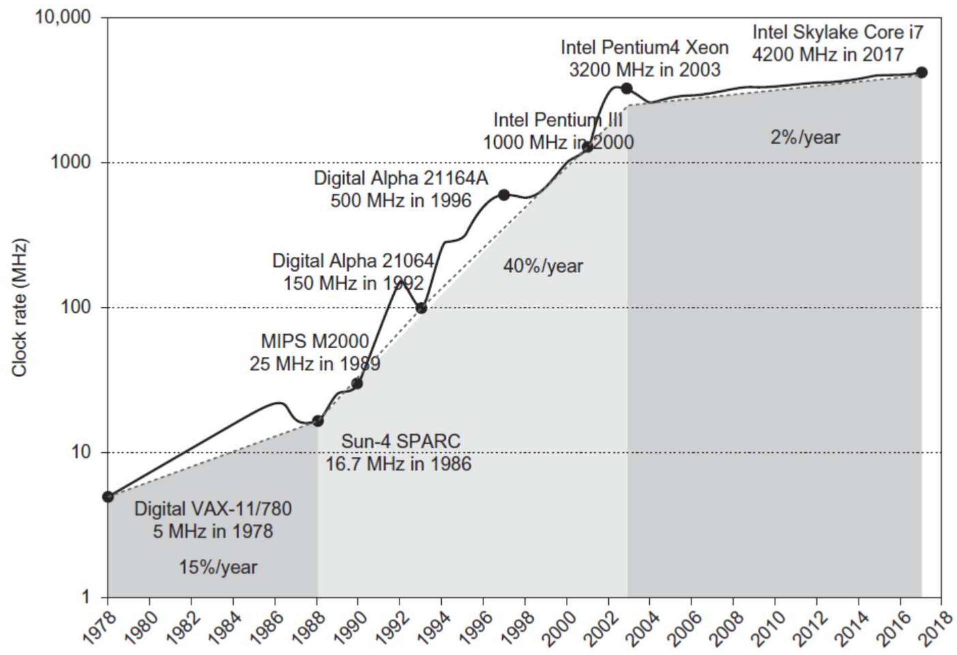

Figure 1.2.: Clock Frequency trend for multiple architectures. (Source: [45])

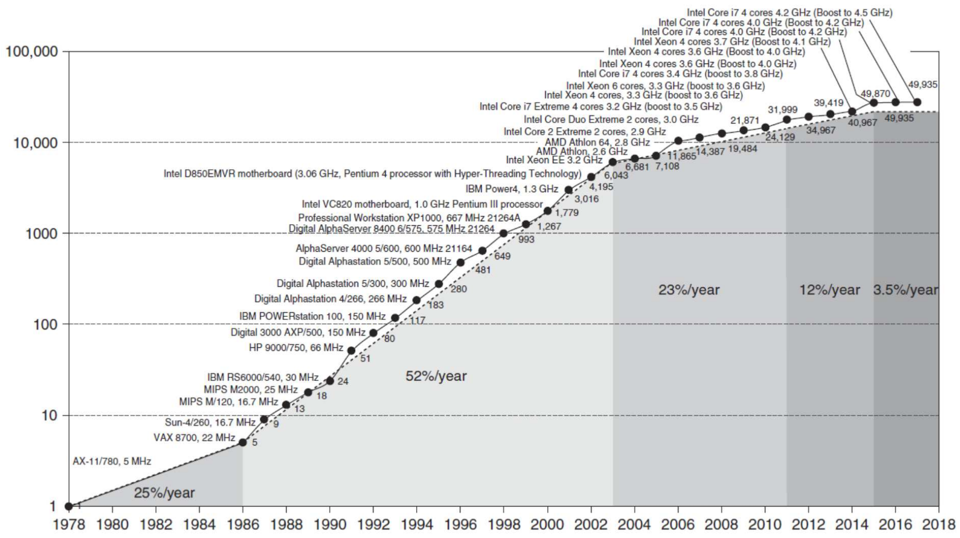

Figure 1.3.: Growth in processor performance since the late 1970s. (Source: [45])

sense only transfers the heat and power issue from one chip to another. Furthermore, additional circuitry is necessary in the main CPU to handle cache logic (write-through or write-back), and cache logic again makes the physical distance to the main memory store an issue.

von Neumann Bottleneck

The von Neumann Architecture describes a computer design wherein program instructions and data share the same memory (as opposed to the *Harvard Architecture* wherein instructions and data are stored in separate memories and accessed by separate buses). This design leads to bus contention as the same physical set of wires are used to transfer not only memory contents and I/O requests and results, but also processor instructions as well. As a result of this contention combined with I/O device and memory latency, a CPU still spends a great deal of time idle waiting for results to be returned from memory.

One solution is to increase bandwidth by increasing the width of the bus. This has successfully been employed in the past as internal CPU architectures and the associated bus have increased in width from 8 (in the initial 8088 models), to 32 and now to 64 bits. However, increasing the bus width is not without consequence. First of all, the internal data paths of the processor have to be increased as well, putting enormous pressure on transistor density with the attendant problems outlined previously. Secondly, the physical packaging of the CPU changes, leading to massive retooling and production costs. Thirdly, inductive crosstalk on the bus increases as more wires are brought into close proximity to each other.

Wirth's Law

A somewhat ironic counterproposal to Moore's Law is attributed to Niklaus Wirth and has come to be known as *Wirth's Law* [100]. It can be colloquially stated as *software is getting slower faster than hardware is getting faster*. Especially in the consumer market, the demand for more and more features results in larger and larger binaries (known as "code bloat"). Furthermore, as developers continually add more code and complexity to their code, they often incidentally introduce inefficiencies and bugs, and find it progressively difficult to adhere to established software engineering techniques. The net result is software that is (at best) no faster despite being run on hardware with successively greater processor, storage, and memory resources. This phenomenon is succinctly expressed in a corollary to Wirth's Law: *software expands to fill the available memory*. It is again worth noting that this supports the proposition that automatic optimization of code is not only not obviated by improvements in hardware technology, but perhaps counter-intuitively is even more necessary because of it.

1.2.3. Parallel Hardware

The preceding discussion highlights the inherent scalability limits of individual computing devices. To attempt to counter the slowing growth in hardware clock speeds and transistor density in recent years, hardware manufacturers have increasingly shifted toward the production of parallel hardware which permits portions of an application to execute concurrently on more than one computing device [35]. Parallel hardware comes in several flavours.

Super-Scalar Processors All modern general-purpose CPUs are *super-scalar*. A super-scalar CPU has multiple *functional units* present, each a specialized logic element designed to perform a certain task or implement a specific instruction. For example, there may be multiple adders, bit shifters, multipliers, floating point dividers, etc. These CPUs exhibit *Instruction Level Parallelism* (ILP) because multiple instructions may be in-flight concurrently, each on a different functional unit. Moreover, a technique known as *pipelining* [45] allows each functional unit to have multiple instructions of the same type processing concurrently within it, each in a separate phase of execution. A super-scalar CPU can either rely on an optimizing compiler to arrange the instruction stream such that adjacent instructions can be dispatched concurrently to separate functional units, or (more commonly), the CPU itself includes logic to schedule and dispatch independent instructions to the functional units, often in an order different than the one in which they appear in the actual compiled program binary. This is called *out of order dispatch*, and it allows ILP without the use of a compiler specifically designed for that particular CPU, although at the cost of increased logic density.

Hyper-Threaded Processors One of the challenges introduced by super-scalar processors is finding enough instructions to keep the available CPU hardware occupied. With a large number of functional units, where each unit is itself deeply pipelined, a single instruction stream may not possess the right instruction mix to fully utilize the potential of the CPU. In this case, some number of functional units remain idle while multiple instructions queue up waiting to access the logic of the other functional units. *Hyper-threading* attempts to address this issue by allowing a single physical CPU to appear to the operating system as multiple (usually two) logical processors. This allows the OS to schedule multiple processes (instruction streams) for execution at the same time, and thereby allows the instruction scheduler within the CPU the opportunity to examine more instructions in the hope of keeping all of its functional units gainfully employed.

Multi-core Processors A *multi-core* processor is one in which multiple physical CPUs are present in the same package, most often (but not necessarily) on separate dies. A *die* refers to a contiguous piece of silicon with logic etched from a single wafer, and is contained in a package. A *package* contains one or more dies along with all of the pins, wiring, plastic and metal housing necessary to interface it with the processor motherboard.

Multi-core processors have several advantages over multiple separate CPU packages. The primary advantage is that because the cores share common memory cache [1], the cache logic can be made simpler and driven at higher clock speeds. This is because the components in the same package are physically closer to one another, meaning the distances a signal has to travel is shorter. This in turn results in less signal degradation, less repeater logic, and also (importantly) less power required to drive the cache logic circuitry. The same reasoning applies to latency reduction in inter-CPU interconnects over an interconnect such as hypertransport or quickpath. Another advantage is less PCB space and circuitry is required on the system main board, which can be a significant plus in physically constrained environments, such as a cellular telephone device.

In addition to being more expensive to produce, the primary disadvantage to multi-core designs is that, relative to single CPU packages, thermal density increases as more circuitry is placed within a single package. Also, a single CPU makes more efficient use of wafer real-estate than multiple CPUs sharing the same die.

Multi-Processors A *multi-processor* is simply a system in which multiple CPU packages are present. As previously discussed, it is not possible to pack an arbitrary number of cores in the same package, primarily because of thermal and transistor density problems. On the other hand, each CPU package is independently cooled, and adding more packages means that the density within each package can be reduced. Although this approach is more scalable than the multi-core route, especially for supercomputing applications, this scalability comes at a price. The overall power consumption and signaling latency of a multi-processor system is greatly increased because of the distances signals have to travel. Obviously, more PCB space is required to house multiple packages, making this approach unfeasible for space-constrained applications.

Virtualization Similar in spirit to hyper-threading, virtualization attempts to fully utilize available hardware. Whereas hyper-threading represents a single CPU as two or more virtual processors to a single copy of an operating system, virtualization allows multiple guest *virtual machines*, each running an instance of an operating system to concurrently execute on a single physical host system. This host system runs a specialized operating system knows as a *hypervisor*, whose job it is to multiplex the hardware resources of the host machine among the various guest operating systems. Each guest operating system is (practically) unaware that it is running on a virtual machine instead of a physical machine. Examples of prominent virtualization systems are VMWare, Xen, z/VM, and Microsoft ESX.

[1] Each core in the same package has a private L1 cache, and (usually) a private L2 cache, but all cores in the same package share at least a L3 cache

Graphics Processing Units *Graphics Processing Units* (GPUS) are specialized processors designed originally to manipulate graphics-related information in a frame buffer for subsequent presentation. Because elements in such buffers can largely be processed independently of each other, GPUs are massively parallel devices wherein a single instruction stream (usually known as a *kernel*) can operate on thousands of data elements concurrently. Thus, for applications that are very highly parallelizable, GPUs can greatly improve performance over that on a general-purpose CPU.

General purpose GPUs are generally programmed in one of two ways. C language extensions such as *CUDA* (Compute Unified Device Architecture) in the case of NVIDIA devices, or OpenCL (which is meant to operate across a range of vendor devices using vendor-provided SDKs) are available to explicitly create the interface between the main application and the GPU. Alternatively, OpenACC (see below) directives can be used to identify sections of code that can be offloaded to an accelerator hardware unit such as a GPU device as well as a CPU. Unlike the parallelization approaches we have examined to this point, usage of a GPU for general-purpose computing thus requires highly specialized code and APIs, and is appropriate primarily for so-called *embarrassingly data parallel* applications.

1.2.4. Exploiting Parallel hardware

Having briefly touched upon the various flavors parallel hardware can assume, the next question is: how does an application actually take advantage of the parallel hardware present in the system? One approach is to directly make use of the Operating System multi-processing primitives such as *fork()* and *join()* to spawn heavyweight processes, or else lighter weight entities such as *threads* or *eventlets*. However, this approach is fraught with difficulty, not the least of which being the issue of properly coding such interfaces, but also the non-portability such a practice inherently engenders.

Another approach heavily used in the optimizing compiler and scientific community is to use a set of compiler directives to identify the portions of a program which the programmer or optimizer has determined can run in parallel with itself or with other portions of the code, and allow the compiler and/or support libraries to handle the actual parallel implementation. A significant advantage of this approach in the context of optimizing compilers (with which this chapter is primarily concerned) is that it provides separation of concerns: the code that detects portions of code which can run in parallel need have no knowledge of how to actually make that happen – it need only demarcate those parallelizable sections of code as such and leave the actual implementation of the parallelization to other (probably operating system and hardware specific) routines within the optimizer or runtime libraries.

OpenMP OpenMP [10] is an open, standardized specification for such a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs. These directives are processed by a supported compiler, which generates the system and operating system specific code necessary to create, schedule, and synchronize the various parallelizable entities and exploit the parallel hardware available to the machine at runtime.

As an example, Figure 1.4 shows a simple C-language program annotated with OpenMP directives. Lines 19 and 24 instruct the compiler that all iterations of the loops they precede can be run in parallel.

OpenACC The OpenACC [71] specification is closely related to OpenMP. Whereas OpenMP specifies a standard set of processor directives to identify section of code that can be run in parallel on symmetrical processors such as multi-core or multi-processor architectures, OpenACC specifies a set of directives to identify portions of code that can be offloaded to an accelerator hardware unit such as a GPU device. OpenACC was initially created by members of the OpenMP Architecture Review Board which manufactured

```c
#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <omp.h>

#define DATA_SIZE 100000000

int main() {
  struct timeval start, stop;
  int *input = (int *) malloc(DATA_SIZE * sizeof(int));
  int *output =  (int *) malloc(DATA_SIZE * sizeof(int));
  int i,j;

  gettimeofday(&start, NULL);

#pragma omp parallel for private(i)
  for (i=0; i<DATA_SIZE; i++) {
    input[i] = i;
  }

#pragma omp parallel for private(j)
  for (j=0; j<DATA_SIZE; j++) {
    output[j] = input[j] + j + 1;
  }

  gettimeofday(&stop, NULL);

  long elapsed_usec = ((stop.tv_sec * 1000000) + stop.tv_usec) - ((start.tv_sec
       * 1000000) + start.tv_usec);
  printf("Elapsed: %ld\n", elapsed_usec);

  free(input);
  free(output);
}
```

Figure 1.4.: Sample OpenMP-annotated C program

certain types of hardware accelerators to specifically address the needs of their customers. OpenACC and OpenMP are actively working to merge their specifications.

OpenCL OpenCL [42] is a specification and implementation whose goal is to exploit all of the available computing hardware on a given platform. It is comprised of a C/C++ API to allow a host to discover and configure the available computing devices available to it, as well as a dialect of C99 named OpenCL-C in which the attached conformant devices are actually programmed. Software modules called "kernels" are programmed in OpenCL-C and enqueued to a OpenCL device (most commonly a GPU) for execution. The host uses the OpenCL API to transfer data to the device to be operated on, and to logically partition that data in an application dependent fashion. The OpenCL conformant device schedules the kernels on a number of internal "Control Units", which are comprised of a device dependent number of "Processing Elements". All processing elements in the device have access to a global memory pool, and all devices with a control unit have access to a "local" memory pool, and each processing element has access to its own "private" memory pool. In order to hide memory latency, the device "overloads" the control units with index space partitions so as to keep all processing elements busy. Once all kernels have completed, the host transfers data from the device back to host memory.

Functional Programming Long a favorite in the academic community, *Functional Programming* (FP) has recently attracted a great deal of interest in industry as well [52]. As opposed to *imperative languages* such as C or Fortran which are based on the idea of sequentially executing a series of statements which read and modify memory, functional programming is a paradigm in which the execution of a program is thought of as the execution of functions in a mathematical sense, with neither data mutation nor the attendant “side effects”. Just as a mathematical function such as $\sin x$ uses its supplied parameters without modifying them or anything else to produce a result, a functional program does not mutate state during the course of its execution. In fact, a purely functional language (of which Haskell is the canonical example) does not even provide a way to change the value of a variable once it is defined. Although this is may sound like a crippling limitation, Haskell allows the programmer to achieve the effects of mutating state via the use of *monads*. Taken from category theory, monads are a subclass of *functors*, which are in turn entities which map one category to another. While some consider a monad a “container for execution”, an alternate way to understand a monad is to think of it as a *context* in which execution takes place. Haskell uses monads to allow non-functional concepts such as I/O, non-determinism, variable assignment, and exception handling to peacefully co-exist alongisde purely functional code. Such a model provides complete *referential transparency*, meaning that the same function called with the same input will always produce the same result. Referential transparency combined with language guarantees that state will not be modified allows the compiler freedom to execute code fragments (so-called *thunks* in Haskell parlance) in parallel, and to cache and reuse results from previous function invocations. This is in stark contrast to competing paradigms such as Object Oriented Programming, which merely attempts to hide modifiable state within the context of an object, while offering no guarantees (at a language level) as to referential transparency. Proponents of FP argue that it allows an algorithm to be expressed in an elegant and easy comprehendable way (with a minimum of code). Despite the growing interest and adoption of FP in industry, several factors may impede its impact on the scientific computing world. One is that despite the fact that the lack of state within a functional program allows the compiler to safely execute portions of it in parallel, functional data structures tend to have higher overhead and a larger memory footprint due to the copying and data versioning that must take place in order to ensure the non-mutative guarantees of the language. This highlights the somewhat counter-intuitive notion that parallelism in and of itself does not necessarily equate to better performance. Another more mundane reason FP has yet to make inroads in scientific computing is the simple fact that so much highly tuned and well tested imperative (especially Fortran) scientific code already exists in libraries already.

1.2.5. Distributed Processing

Whereas multi-core systems pack multiple CPUs in a single package, and multi-processor systems pack multiple packages in a single system, *distributed systems* utilize multiple network-connected systems. Nodes in a distributed systems are said to be *loosely coupled* because the latency involved in communicating between them is orders of magnitude greater than in the systems I touched upon earlier and make large-scale sharing of data impractical. Said another way, distributed nodes must be made to operate as independent as possible of the other nodes. As is the case with GPUs, distributed systems require specialized APIs to communicate among the nodes, and several frameworks have been developed to facilitate tasks such as distributing work among the members, communicating status, gracefully dealing with node and communication failures, etc. Generally speaking, distributed systems operate at a much higher level of granularity than parallel sections of code running in the same system because of the latency involved. Distributed processing is a massive topic that cannot be adequately addressed here. I will briefly describe representative samples of the current approaches to distributed parallel processing, but as it is somewhat orthogonal to the point of this section, will not touch upon them further in this chapter.

PVM: Parallel Virtual Machine Parallel Virtual Machine (PVM) [50] is a software toolkit originally developed as a joint project between the Oak Ridge National Laboratory and Emory University. It is designed to make a collection of heterogeneous computers connected over the Internet or private network function as a single virtual machine. PVM provides an API that allows a distributed application to spawn processes on participating nodes, distribute work among them, pass messages, and detect and respond to node failures.

MPI: Message Passing Interface MPI [33][67] is a specification for a message passing API and protocol for the reliable exchange of messages in a distributed or shared memory system. It provides numerous data types, support for various network topologies (i.e. point to point, grid, group, etc., as well as communication paradigms such as broadcast, multicast, one-to-one, scatter-gather, etc. Powerful, rich, and expressive, it has become the de facto standard for modeling communication among participating processes in a distributed parallel processing system. It has been implemented in a vendor specific way for a wide variety of platforms, and a open source, vendor neutral implementation, OpenMPI [76] is also available.

Cilk Plus As opposed to a library, Cilk Plus [18] is an extension to the C and C++ programming languages as well as a runtime environment that supports data and task level parallelism. Cilk Plus is based on Cilk [91], a research project at MIT. Rather than forcing the programmer to explicitly schedule workers and statically partition the work among them, CilkPlus archives parallelism via a dynamic work stealing scheduler. Proponents argue that Cilk's "serial semantics" allow the algorithm to be expressed in a more natural way, without the obfuscation introduced by artificially structuring the code to make it suitable for a particular parallel library or paradigm. There is some research [62] suggesting that Cilk not only provides performance improvements on Intel hardware, but also allows algorithms to be expressed in fewer lines of code.

Distributed Data and Processing: Hadoop Hadoop [34] is Java-based, open source framework which allows distributed processing of large amounts of data on a collection of heterogeneous computing nodes where neither the nodes themselves nor the network connections between them are assumed to be reliable. The hardware nodes are used for either processing, data storage, or both. Hadoop consists of two major components, a distributed file system which partitions and replicates a data set among the data storage nodes, and a process management layer that distributes and monitors data processing tasks among the processing nodes. It detects and responds to failed processing nodes by redistributing the failed node's workload to surviving nodes. Additionally, the HDFS (Hadoop Distributed File System) is location aware, meaning that work will be sent to the processing node closest to the data nodes holding the data for the processing in an attempt to reduce latency and network traffic. If a data node should fail, it will attempt to reconstruct the failed node's portion of the dataset on surviving nodes, and will redistribute in-flight processes to nodes closer to the new data location. The Hadoop processing model is primarily *map-reduce* [95] model where intermediate results are generated and progressively combined to produce the desired result.

1.2.6. Issues with Parallel computing

The ready availability of parallel hardware is of little value if it cannot be exploited effectively. Unfortunately, parallel programming is complex, and often beyond the grasp of inexperienced programmers. Complexity aside, it is also quite a labor intensive process in terms of development, debugging, testing, and deployment effort. Even worse, an improperly coded parallel system will either provide incorrect or intermittently incorrect results with little warning, or else perform much worse than an equivalent sequential system.

The first challenge facing a programmer is identifying parts of a program which can safely run in parallel with other parts. Very often, extensive program restructuring is necessary to expose the underlying parallelism, but doing so requires an intimate knowledge of the data flow and dependencies within the program, usage of global state, as well as side-effects produced by called sub-routines. Additionally, several low-level and processor-specific details must be considered in order to achieve optimum (or at least non-counterproductive) parallelism. Some of the issues to consider are described below.

Synchronization Parallel threads of execution communicating with or controlled by shared data structures present the possibility of race conditions. Left uncontrolled, simultaneous updates by multiple threads to the same memory location lead to non-deterministic behaviour which is notoriously difficult to reason about and debug. In order to guarantee program semantics, access to these shared structures must be carefully controlled via mechanisms such as semaphores, spinlocks, transactional memory, atomic operations, and message queues. Correct usage of these primitives is non-trivial, and their incorrect usage is itself a source of bugs and performance degradation.

Data partitioning Consideration must be be given as to how to optimally partition the data so as to minimize communication costs and expensive cache invalidations due to competing updates by different processors. The issue of cache performance requires detailed knowledge of the hardware on which the program is to be run, such as the geometry, quantity, set associativity, and sharing of the various cache levels. The distribution of and communication latency between the processing threads may also impact this decision, as greater latency may require the data to be relocated to thread-private stores rather than being shared in a common pool such as cached or main memory.

Task partitioning As opposed to data-level parallelism, where the source dataset is partitioned among multiple worker threads running the same algorithm, employing task level parallelism entails dividing the work to be done into multiple phases in a processing pipeline. Intermediate results move from stage to stage in an assembly-line fashion, and each stage runs concurrently with all others. This may benefit applications where no single phase is readily parallelizable, and also minimizes the impact of latency between the processing nodes, although communication bandwidth can become a limiting factor.

Parallel thread count The right number of parallel threads must be chosen to maximize processor utilization while at the same time eliminating expensive context switches when one process yields a processor to another process. The right number of threads is not only a question of how many processors are available, but also how much concurrency the application can exhibit. For example, if all threads are largely I/O bound, then the number of threads can greatly exceed the number of processors, since each thread will spend most of its time waiting for external events to occur. On the other hand, if the threads are CPU bound, then increasing the thread count beyond the number of processors is counter-productive, as the threads will simply compete among themselves for access to the processor, and expensive process switching is inevitable. Process switching is expensive for several reasons, including the fact that the Translation Lookaside Buffer (TLB), instruction caches and data caches may all need to be purged, and the user and processor state registers saved and reloaded each time one process must yield control of the processor to another.

The preceding discussion highlights the fact that parallelization incurs overhead, whether it be the cost of cache invalidations, latency in communicating data from one processing element to another, or the cost of spawning worker threads and distributing data to them. If the benefit of parallelization is dominated by the overhead it incurs, then a sequential algorithm will obviously be a better choice.

Even if it is determined that a section of code can profitably be parallelized, the overall speedup depends on how much of the overall runtime is spent in that section of code. This is a paraphrasing of Amdahl's Law ([1]), another axiomatic rule governing the benefits of optimization in general. As applied to optimization resulting from parallelism, it can be mathematically stated as

Let:

- $n \in \mathbb{N}$ be the number of parallel threads available,
- $P \in [0,1]$ be the fraction of the code that can be parallelized
- $e \in [0,1]$ be the efficiency of the parallel code, where 1 means no time is lost in communication, cache contention, etc.
- $k \in \mathbb{R}$ be the time required to instantiate and dispose of each thread

then the time required to run the code in parallel on n threads is given by:

$$T(n) = T(1)(\frac{p}{ne} + (1-P)) + nk$$

Assuming no thread startup/teardown overhead and perfect efficiency, the maximum speedup of a parallel version of the program as compared to the serial version is given by

$$S(n) = \frac{T(1)}{T(n)} = \frac{1}{\frac{p}{n} + (1-P)}$$

1.2.7. Automatic Compiler Optimization

Common to all of the preceding techniques is the necessity for the programmer to identify those areas of the program that can be both safely and profitably parallelized and to explicitly effect parallel execution either by using language or library facilities. If a programmer is unfamiliar with either the nature of the algorithm employed or the hardware on which it will ultimately execute, then in the best case potential parallelism may be missed or unprofitably exploited, and in the worst case attempts to parallelize code that must be sequentially executed to preserve program semantics can introduce subtle or randomly occurring errors.

For these reasons, the automatic parallelization of programs by optimizing compilers that attempt to detect and exploit safe and profitable parallelization opportunities within a program have received considerable attention. The advantages of this automatic parallelization are clear: subtle opportunities that might go otherwise undetected can be found, and optimization will only be attempted if the compiler is able to prove that it will not lead to erroneous results. Furthermore, a compiler targeting a particular architecture has detailed knowledge of the hardware on which the program will ultimately execute, such as cache configuration, bus interconnect configuration and latencies, CPU counts, etc.

Loops iterating over data arrays in memory are a rich source of potential parallelism in a program. An optimizing compiler with a detailed view of the target hardware and operating system context may be able to perform valuable program transformations such as re-ordering or reversing loops to maximize cache efficiency, exploiting vector hardware, or (most relevant to this discussion) running one or more nested loops in parallel on separate computing cores.

In order to safely apply an optimizing transformation to a loop structure, the compiler performs data dependence analysis to reveal the ordering constraints among the statements in the loop that need to be preserved in order to produce valid optimized code, and tests that that the potential transformation does not violate those constraints. This essentially consists of determining the conditions (if any) under which one iteration of a loop references a memory location (most commonly an array element) that is subsequently referenced by another iteration of the loop, and whether that reference (read or update) precludes the transformation being considered. In order to permit the most fruitful transformations in terms of performance, these data dependence tests must be as accurate as possible. On the other hand, the tests need to be efficient in order to prevent the compilation time from becoming prohibitive.

1.3. Problem Statement

As mentioned, optimizing compilers rely upon data dependence analysis to reveal the ordering constraints among statements in a program that need to be preserved in order to produce valid optimized and parallel code. Testing array references for data dependence is equivalent to determining the existence of integer solutions to a system of inequalities. A number of data dependence tests have been proposed in the literature. Each test presents a different tradeoff between accuracy and efficiency.

The primary problem I address in this thesis is that although the literature includes numerous empirical comparisons of dependence tests [74, 77, 80] in which benchmarks suites are used to measure the performance and effectiveness of the various techniques, there has been no work done to explain the observations made or conclusions drawn from these studies in a purely analytical fashion. This is a significant deficiency in the literature, as no matter how objectively empirical studies are done, there is always the risk of some ambiguity as to whether the results were skewed by some environmental factor or condition of the test (algorithmic implementation, compiler optimization settings, benchmark suite chosen, etc.). That is not meant to minimize the contribution made by empirical work, as it is conversely important to know whether theoretical differences in dependence tests result in any practical impact. Nonetheless, these empirical tests alone cannot fully compare and contrast these important tests without abstracting them into a mathematical framework and performing a rigorous analysis of the algorithmic structure of the various tests being studied.

To address this shortcoming, in this thesis I study the fundamental relationships between several data dependence tests when applied to dependence problems involving single dimensional arrays or multi dimensional arrays with uncoupled subscripts. I consider the Banerjee Extreme Value test, Fourier Motzkin Variable Elimination (FMVE), the I-Test, and the Omega test, which are representative of the state of the art in data dependence analysis. The Banerjee Extreme Value test and FMVE can only determine the existence of real solutions to a system. Thus they can only disprove, but not prove, data dependence. The I-Test and the Omega test refine the Banerjee Extreme Value test and FMVE, respectively, to the integer domain and can prove data dependence. The Omega test is a more accurate data dependence test, but with worst case exponential time complexity. The I-Test is a polynomial time test, but it is not always conclusive. I first show that FMVE is equivalent to the Banerjee Extreme Value test. I then show that the Omega test's technique to refine FMVE to integer solutions (dark shadow) is equivalent to the I-Test's refinement of the Banerjee Extreme Value test to integer solutions (the accuracy condition). I then show that in the absence of direction vectors, the Omega test requires an exponential time exhaustive search to produce an exact answer (the so-called Omega Test Nightmare) if and only if the I-Test returns an inconclusive (maybe) answer. Finally, I consider dependence problems arising from direction vectors and derive the exact conditions under which the Omega Test is more accurate than the I-Test, and vice-versa.

There are several questions that this thesis seeks to answer. Among them:

- How does the Fourier-Motzkin elimination technique compare to the Banerjee Bounds Test from a analytical standpoint?

- How does the I-Test refinement to the Banerjee Bounds Test compare theoretically to the Omega Test's refinement to Fourier-Motzkin? Does one subsume the other, or are they equivalent?

- Can the I-Test be exact and efficient when the Omega Test requires exponential time to deliver an exact answer to the same problem?
- Conversely, can the Omega Test deliver an exact answer efficiently when the I-Test returns an inexact "maybe" answer?
- It has been suggested in [77] that since the Omega Test is exponential in the worst case, that the I-Test be used as a preliminary filter, and that the Omega Test be employed only if the I-Test cannot produce an exact answer. Is there merit to this idea?
- Other than performing an exhaustive search of the iteration space for an exact answer, are there other factors that might explain the conventional wisdom that the Omega Test is too expensive for practical use in optimizing compilers?
- How does the Direction Vector extension to the I-Test compare to the Omega Test when both are given equivalent problems to solve?

The answers to these research questions will be developed through the course of the thesis, and form its novel contribution to the field. I will also describe several ancillary contributions that arose during the course of the research, as well as describe a novel idea that I developed while collaborating with colleagues in a project to eliminate redundant array bound checks in Java bytecode.

1.4. Importance of Data Dependence Analysis in Modern Scientific Computing

After having started this section with an overview of modern parallel processing techniques that brimmed with the latest buzzwords (GPUs, OpenCL, OpenMPI, Hadoop, etc), the reader may be surprised that this thesis will concern itself with the mathematical constructs underlying data dependence tests used by compilers for well established imperative languages such as Fortran. While it is true that the use of languages such as Fortran has drastically diminished in industry, Fortran remains one of the most widely used languages in scientific computing (along with C++). Fortran has undergone many enhancements over the years to make it competitive with more "modern" languages such as C++. In fact, benchmarks show that in terms of performance, Fortran and C++ are almost equivalent, with Fortran outperforming C++ on the Spectral and N-Body benchmarks [94]. Fortran was designed with array- and vector-intensive computation in mind, and that is expressed in native language features such as builtin array copying, addition, multiplication, element masking, etc. For this reason, algorithms expressed in Fortran are arguably more readily understood without the extraneous clutter and obfuscation introduced by modern languages. Furthermore, Fortran is still the language of choice for a large percentage of physicists and mathematicians who rely on well tested and mature numerical and simulation libraries such as LAPACK, EISPACK, CERN, MINPACK, etc.

More importantly, the analysis in this thesis is analytical in nature and is not tied to any particular language implementation. It intends to investigate an area that has never been formally addressed in the data dependence literature, namely how the predominant tests compare from a purely theoretical standpoint. Proponents of one test or another frequently point to empirical results written for an instrumented (usually academic) compiler demonstrating performance for a specific benchmark suite on a particular architecture. This thesis takes a different approach. It attempts to explain the relationship among the tests, and compares the algorithms they employ in terms of complexity and problem domain.

Lastly, the tests I examine in this thesis are integer programming algorithms, and data dependence analysis is not the only problem to which they can be applied. For example, one of the fundamental impediments to the use of Java in the high performance computing arena is the fact that the JVM specification dictates that array accesses are always tested for out-of-bounds violations to ensure the safety of the host platform. In other work [85, 38, 69] we have demonstrated the tests I am about to examine can be used by an optimizing Java compiler to prove that an array out-of-bounds condition cannot occur, or else to specify the exact conditions under which the access might be unsafe. In the former case, the compiler can completely eliminate the bounds check. In the latter case, optimizations can be speculatively applied and the guard conditions tested at run time. I describe this work in the third part of this thesis.

1.5. Structure of Thesis

The second part of this thesis focuses on describing the dependence tests on which I focus: Fourier-Motzkin Variable Elimination (FMVE), the Banerjee Bounds Test, the Omega Test, and the I-Test, and then proving how they are related. In chapter 2, I give a brief overview of dependence analysis, introduce the tests, and review the literature relevant to the tests. Chapters 3,4, and 5 form the major contributions of the thesis. In chapter 3, I show the Banerjee Bounds Test and FMVE are isomorphic given conditions under which both are applicable. In chapter 4, I show the isomorphism of their respective integer-valued refinements (the I-Test and the Omega Test). In chapter 5, I extend the comparison to the case where the system includes *direction vectors*. The relationships I state and prove have never previously been mentioned in the literature, explain empirical results observed in the literature, and thus are novel contributions to the field. In chapter 6, I conclude the second part of the thesis with a discussion of a number of facets of FMVE and the Omega Test. I also show the relationship of the Dark Shadows Inequality to the two-dimensional Frobenius Coin problem, and further prove a tighter bound on the effectiveness of the Omega Test's equality elimination algorithm than was previously known. These also are my contributions.

The third part of the thesis discusses my contributions in using a form of FMVE in abstract interpretation to enable a runtime system to eliminate redundant array bounds checks in the Java language. I heavily reference our peer-reviewed published articles that either describe the algorithms or else utilize them as part of a larger system. Utilizing this test to identify redundant array bound checks improved performance on certain benchmarks by nearly 10%.

Part II.

Variable Elimination Techniques

Chapter 2

Dependence Analysis

In order to safely apply transformations to a nested loop, optimizing compilers perform data dependence analysis to reveal the ordering constraints among statements in a possibly nested loop that need to be preserved in order to produce valid optimized code. The problem essentially is one of determining the conditions (if any) under which one iteration of a loop references a memory location that is subsequently referenced by another iteration of the loop. If it can be proven that no such “collisions” exist, or if such collisions are harmless, (perhaps both iterations simply read the same location), then the compiler is free to schedule all iterations of that loop in parallel with one another. Otherwise a data dependence is said to be “carried” by that loop and the compiler must perform deeper analysis, perhaps to determine the so-called “stride” between collisions. That is, if loop iteration i writes a memory location that is read by loop iteration $i+n$, then the compiler could potentially schedule n iterations of the loop to execute simultaneously, but force the next block of n iterations to wait until the first has completed.

Amdahl’s Law tells us that the most beneficial optimization in terms of overall speedup is an optimization to the most-frequently executed portion of the code. In scientific computing, these “hot-spots” are most frequently loop structures in which arrays are being manipulated. The loops can be nested to an arbitrary depth, often corresponding to the logical dimensionality of the array. It is for this reason that optimizing the performance of these array-accessing loops has garnered so much attention in the literature, and in particular spawned a number of dependence analysis testing techniques.

In this chapter I discuss data dependence analysis with particular emphasis on array processing in nested loops, and present the data dependence tests that I will examine in greater detail in this thesis.

2.1. Modeling Data Dependence

One of the first tasks an optimizing compiler must perform is to analyze a program or program fragment, and distill from its textual representation the relevant information needed to identify and evaluate potential restructuring. As part of this analysis, the compiler represents this information in an abstract and program language independent manner.

2.1.1. Data Dependence Graphs

The most basic abstraction is the *data dependence graph*, whose purpose is to represent how one section of the program depends on other sections of the program. It is a directed multigraph whose vertices represent sections of the program. The word *section* is intentionally ambiguous. It can refer to either a single source code statement, a single machine language statement, or a block of such statements. For the purposes of

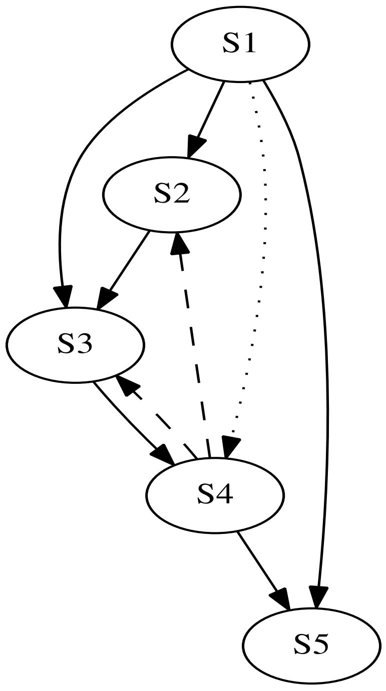

Figure 2.1.: Address-based dependency graph

this section, however, a vertex in the graph will represent a single statement in the source code. An edge in the data dependence graph represents a dependence between the edge's endpoints. An edge is annotated to describe which of the three types of dependence it represents:

Flow Dependence A *flow dependence* exists when a variable is assigned a value in an instruction, and that variable is subsequently read by another instruction. In the dependence graph, the edge points from the assigning instruction to the instruction where the variable or memory location is read.

Anti-Dependence An *anti-dependence* is essentially the opposite of flow dependence. It exists when a variable is consumed in an instruction, and that variable is subsequently assigned by another instruction. In the dependence graph, the edge points from the using instruction to the instruction where the variable or memory location is subsequently assigned.

Output Dependence An *output dependence* exists between two statements if both define the same variable, and the statement represented by the tail of the edge is reachable from the statement represented by the head of the edge.

Consider the sequence of statements shown in this program fragment:

S_1 : A = 123
S_2 : B = A + 1
S_3 : C = A + B
S_4 : A = C + 3
S_5 : D = C + A

the data dependence graph for this fragment is shown in Figure 2.1

The solid edges represent a flow dependence. For example, the variable *A* is defined by statement S1 and used in statement S2. The dotted edges represent an output dependence. In this case, the dotted edge from S1 to S4 exists because S4 redefines *A*, which was initially defined by S1. The dashed edges represent anti-dependence. Note that statement S4 defines variable *A* that was previously used by statements S2 and S3.

This style of dependence graph is properly called an *address-based* dependence graph, because it defines dependences based solely on the addresses of the variables involved. Another style is known as *value-based* graphs. To illustrate the difference, note that in Figure 2.1, a flow edge exists from statement S1 to S5, since S1 defines variable *A* and statement S5 uses that variable. However, there is an intervening assignment to variable *A* in statement S4 and statement S5 is not reachable from statement S1 without executing statement S4. In a value-based dependence graph, the assignment in S4 is said to *kill* the flow dependence from S1 to S5, and the solid edge from S1 to S5 would thus not appear in the graph.

2.1.2. Dependence Abstractions in Loops

The code fragment I considered earlier is intentionally trivial, as each statement is executed exactly once in order. Clearly, real-life programs pose a myriad of complications such as I/O, conditionals, inter-procedural calls, and especially loops.

Statements embedded in the body of loops are potentially executed many times, and a dependence may exist between a statement and another statement either within the same loop iteration, or a subsequent or later iteration. Also, a new wrinkle to the problem emerges in that a statement in one iteration of the loop can have a dependence to itself in a different loop iteration. These complications make the simple dependence graph we've already seen inadequate for expressing dependences for statements within loops, and so the compiler must therefore employ different abstractions to express such loop-based dependences.

Loop Carried and Loop Independent Dependences

If a statement in one iteration of the loop has a dependence relationship to another statement in the same iteration of the loop, we term that dependence to be *loop independent* (because the dependence exists without regard to the loop). On the other hand, suppose a dependence exists between a statement in one iteration of the loop and another statement in a *different* iteration of the loop. Then we say the dependence is *loop carried* (because the dependence exists only within the context of the loop). The program fragment in Listing 2.1 serves to illustrate the distinction.

```
for I = 1 to 100 do
    S1: A[I] = B[I] + 5
    S2: B[I+1] = A[I] + C[I]
end
```

Listing 2.1: Example of loop independent and loop carried dependences

We see that there is a flow dependence for array A from S1 to S2 because S1 defines an array element in every iteration and S2 reads that same array element. Because the definition and the read of the element with array A occurs within the same loop iteration, we classify this dependence as loop independent. There is also a flow dependence for array B from S2 to S1. In this case, however, the array element defined by S2 is not read until the *next* iteration of the loop. Therefore, this dependence is classified as a loop carried dependence.

Iteration Vectors

Optimizing compilers often find it necessary to represent a particular iteration of a nested loop. A simple way to label an iteration of a n-nested loop is to capture the values of the enclosing n loop iteration variables in an n element vector $\hat{v}$ (called an *iteration vector*), where each element v_i ($1 <= i <= n$) is assigned the value of the i-th loop variable when that particular instance of the loop is executed.

Although this is a simple technique, it is less than optimal in the event of non-unit loop strides or decreasing loop indices. For example, in the case of non-unit strides, one iteration of the loop differs from the very next iteration of the loop by a non-unit value (the value of the stride). In the case of a loop with decreasing iteration variables, an iteration vector representing a loop iteration will be lexographically greater than an iteration vector representing a subsequent iteration of the same loop. To address these shortcomings, compilers frequently represent iterations of a nested loop with a *normalized iteration vector*, which is one where each loop is assumed to start at 0 and increase by 1 with each iteration. This simplifies the compiler's analysis, and it allows the iteration space spanned by the iteration vectors to form the subset of a lattice.

Distance Vectors

Building on iteration vectors, the next abstraction used by optimizing compilers to represent dependencies is the idea of *distance vectors*. Assume a dependence exists between statement S_1 with an iteration vector $\hat{v}_1$ and a statement S_2 with an iteration vector $\hat{v}_2$. In other words, when statement S_1 executes with the enclosing loop iteration variables set to the corresponding elements of $\hat{v}_1$, there is a dependency to an instance of statement S_2 when the iteration variables for its enclosing loops hold the values described in $\hat{v}_2$. The distance vector $\hat{d}$ for this dependence is represented by the vector difference $\hat{d} = \hat{v}_2 - \hat{v}_1$. If statements S_1 and S_2 have different nesting levels, then $\hat{d}$ will have an entry for each common enclosing loop.

Direction Vectors

Distance vectors are useful in cases where there is a constant, known number of iterations of each common enclosing loop between the source and target of the dependency. In many if not most cases, however, the distance between the source and target of a dependency is non-constant. This is illustrated in Listing 2.2 when I consider the flow dependency between statement S_1 to statement S_2 for the array A. In this case, statement S_1 is the source of the dependence, and I represent the iteration vector when control reaches it as $\hat{v}_S$. Statement S_2 is the target of the dependence, and I designate the iteration vector at this point $\hat{v}_T$. We see that given the iteration vector $\hat{v}_S = (1)$, statement S_1 defines A[2]. In the very next iteration of the I loop (i.e. $\hat{v}_T = (2)$) statement S_2 reads A[2]. Thus, the dependence distance between these two iteration vectors is $\hat{d} = \hat{v}_T - \hat{v}_S = (1)$. However, given $\hat{v}_S = (2)$, statement S_1 defines A[4] which is read by S_2 when $\hat{v}_T = (4)$, and so now $\hat{d} = \hat{v}_T - \hat{v}_S = (2)$. It is obvious by inspection that each iteration i of the outer loop defines an element read by iteration $i * 2$, and so the dependence distance cannot be represented by a constant value.

```
for I = 1 to 100 do
    S1: A[2*I] = B[I] + C[I] + 5
    S2: B[I+1] = A[I] + C[I]
end
```

Listing 2.2: Example of a dependence with non-constant distance vector

Fortunately, in many loop transformations (the most important being loop reordering), it is not necessary to know the exact dependence distance. Rather, knowing whether the distance is positive, negative, or zero is sufficient to determine whether the optimization can be safely performed. This information is captured in a *direction vector*, where each component is either '<', '=', or '>', depending on whether the element of the source iteration vector ($\hat{v}_S$) is less than, equal, or greater than the corresponding entry in the target iteration vector. In the example shown in Listing 2.2, for example, the flow dependence between S_1 and S_2 is ($<$), since the iteration vector at which an element of A is defined is always less than the iteration vector at which the element is read.

2.1.3. Example of Distance Vector Usage in Loop Transformations

Arguably the most beneficial optimization an optimizing compiler can perform is *loop interchange*, which involves moving an inner nested loop outside of its enclosing loop. This can result in numerous benefits. For example, if an inner loop has a high startup cost, then that cost must be paid with each iteration of the outer loop(s). Moving the loop to the outermost position means that this overhead is incurred only once.

Another benefit can be obtained if the inner loop is the only loop which carries a dependency. If this loop can be moved to the outermost position, then the new inner loops can be executed in parallel, leaving the outer loop to execute sequentially to preserve the data dependence.

Perhaps the most important benefit from loop reordering is improving memory access patterns, especially with regard to cache. Consider the example of a 100x100 matrix initialization shown in Listing 2.3. If cache lines on this particular machine are exactly 100 bytes long, then this example demonstrates the pathologically worst case where every execution of statement S_1 causes a cache miss. On the other hand, if the J loop is moved to the outermost position, then each iteration of the I loop (except the first) will result in a cache hit.

```
for I = 1 to 100 do
    for J = 1 to 100 do
        S1: A[J,I] = 0
    end
end
```

Listing 2.3: Example of Loop Interchange to Improve Memory Access Patterns

The compiler ensures the legality of the loop interchange by examining the dependence direction vectors for the innermost loop. If the compiler is considering moving the loop at nesting level $n+1$ outside of its enclosing loop at nesting level n, the compiler switches columns n and $n+1$ in all affected direction vectors and checks whether the leftmost non-equal direction vector is "$<$". If this condition holds, then the interchange is legal, since all loop carried dependencies will still be preserved in the modified loop nest.

Take for example, the loop shown in Listing 2.4. The only dependency is the flow dependency from statement S_1 to S_2 for the array A. The distance vector for this dependency is (0, 1), and the corresponding direction vector is ($=$,$<$).

```
for I = 1 to 3 do
    for J = 1 to 3 do
        S1: A[I,J+1] = M
        S2: N = A[I,J]
    end
end
```

Listing 2.4: Example of loops that may be legally interchanged

The pattern of array accesses is shown in the table below. We see, for example, that there is a flow dependency between iterations (1,1) (which defines the element A[1,2]) and iteration (1,2) (which reads the same element). Likewise, elements A[2,2], A[3,2], A[1,3], A[2,3], and A[3,3] are each read after a prior loop execution defined it.

I	J	Read	Write
1	1	A[1,1]	A[1,2]
1	2	A[1,2]	A[1,3]
1	3	A[1,3]	A[1,4]
2	1	A[2,1]	A[2,2]
2	2	A[2,2]	A[2,3]
2	3	A[2,3]	A[2,4]
3	1	A[3,1]	A[3,2]
3	2	A[3,2]	A[3,3]
3	3	A[3,3]	A[3,4]

Now suppose the compiler wishes to interchange the I and J loops. The original direction vector (=,<) is mutated to form (<, =). The leftmost non-equal position in the vector is indeed "<", so the legality test is met. If we move the J loop outside of the I loop, the access patterns in the new loop are shown below:

I	J	Read	Write
1	1	A[1,1]	A[1,2]
2	1	A[2,1]	A[2,2]
3	1	A[3,1]	A[3,2]
1	2	A[1,2]	A[1,3]
3	2	A[2,2]	A[2,3]
3	2	A[3,2]	A[3,3]
1	3	A[1,3]	A[1,4]
2	3	A[2,3]	A[2,4]
3	3	A[3,3]	A[3,4]

Notice how all the dependencies in the original loop are preserved: the reads of elements A[1,2], A[2,2], A[3,2], A[1,3], A[2,3], and A[3,3] are each preceded by writes to those locations. Now let us modify the loop slightly as shown in Listing 2.5. Again, the only dependency is the flow dependency from statement S_1 to S_2 for the array A. The distance vector for this dependency is (1, -1), and the corresponding direction vector is (<,>).

```
for I = 1 to 3 do
    for J = 1 to 3 do
        S1: A[I+1,J-1] = M
        S2: N = A[I,J]
    end
end
```

Listing 2.5: Example of loops that cannot be legally interchanged

The pattern of array accesses is shown in the table below. Note now that there is a flow dependency between iterations (1,2) (which defines the element A[2,1]) and iteration (2,1) (which reads the same element). Likewise, elements A[2,2], A[3,1], and A[3,2] are each read after a prior loop execution defined it.

I	J	Read	Write
1	1	A[1,1]	A[2,0]
1	2	A[1,2]	A[2,1]
1	3	A[1,3]	A[2,2]
2	1	A[2,1]	A[3,0]
2	2	A[2,2]	A[3,1]
2	3	A[2,3]	A[3,2]
3	1	A[3,1]	A[4,0]
3	2	A[3,2]	A[4,1]
3	3	A[3,3]	A[4,2]

Now suppose the compiler wishes to (illegally) interchange the I and J loops. The original direction vector (<,>) is mutated to form (>, <). The leftmost non-equal position in the vector is now ">", so the legality test fails. Suppose we attempt the loop interchange anyway. The access patterns in the new loop are shown below:

I	J	Read	Write
1	1	A[1,1]	A[2,0]
2	1	A[2,1]	A[3,0]
3	1	A[3,1]	A[4,0]
1	2	A[1,2]	A[2,1]
3	2	A[2,2]	A[3,1]
3	2	A[3,2]	A[4,1]
1	3	A[1,3]	A[2,2]
2	3	A[2,3]	A[3,2]
3	3	A[3,3]	A[4,2]

Notice how the reads to elements A[2,1], A[2,2], A[3,1], and A[3,2] now occur prior to the writes to those locations, and thus the semantics of the original program have been violated.

2.2. Data Dependence Testing as an Integer Programming Problem

The problem of testing whether a pair of array accesses within (potentially nested) loops can safely be run in parallel with each other is actually a problem in integer programming. In its most general form this is an NP-complete problem, instances of which optimizing compilers may have to attempt to solve thousands or even millions of times in the compilation of large and complex kernels. In order to keep compilation times reasonable, compilers employ data dependence tests that attempt to efficiently and conservatively *approximate* exact solutions to the IP problem. An approximate test must be *conservative* in the sense that if it cannot conclusively disprove the existence of a dependence between a pair of array accesses, it must inform the compiler that a dependence *may* exist in order to ensure that valid code is produced. It should be clear that a tradeoff immediately presents itself: either expensive but more accurate data dependence tests are used that may reveal more parallelization opportunities at the cost of extending compilation times, or else less expensive tests are employed that keep compilation times tolerable but may fail to detect latent parallelism in the code being compiled.

Many data dependence tests have been proposed in the literature, each of which attempts to deliver as quickly as possible exact yes or no answers to the question of whether dependences exist between pairs of array accesses in a nested loop. Some tests are very efficient in a restricted problem domain, others have higher overhead but are applicable to a larger set of problems. Broadly speaking, two dependence tests have emerged as predominant in the literature, and these are the tests that I focus on in this thesis. Although they have a different lineage (which we shall soon examine), common to both is their approach to using "projection" to repeatedly reduce an integer programming problem in n dimensions to one in $n-1$ dimensions.

2.2.1. Formulating the Problem

I have stated that the kind of problems an optimizing compiler has to solve when attempting to determine whether a pair of array accesses buried in an set of nested loops have an ordering dependence between them is an instance of integer programming. I will expand on this idea more formally in subsequent chapters, but for now, consider the simple loop structure depicted in Listing 2.6. It consists of r nested loops, where each of the r loop iteration variables range between a lower and an upper limit. These upper and lower loop bounds are integers, as are the loop iteration variables. Furthermore, each loop iteration variable is incremented by a non-zero integer in each iteration. The loop is perfectly nested, and the statements S1 and S2 which constitute the body of the loop access an array A. Note that the array indices denoting the elements accessed are linear combinations of the enclosing loops' iteration variables (this is the predominant pattern in the majority of scientific loop kernels).

```
for $I_1 = L_1$ to $U_1$ do
    for $I_2 = L_2$ to $U_2$ do
        ⋮
        for $I_r = L_r$ to $U_r$ do
            S1: A[$f_1(I_1,I_2,\ldots,I_r), f_2(I_1,I_2,\ldots,I_r), \ldots, f_d(I_1,I_2,\ldots,I_r)$] = ...;
            S2: ... = A[$g_1(I_1,I_2,\ldots,I_r), g_2(I_1,I_2,\ldots,I_r), \ldots, g_d(I_1,I_2,\ldots,I_r)$] ;
        end
        ⋮
    end
end
```

Listing 2.6: Example of a nested loop

The values assumed by the enclosing loop iteration variables $I_1, I_2, \ldots, I_r$ during the execution of the loop can be thought of as forming a convex polyhedral subspace of $\mathbb{Z}^r$. Let the values of the loop iteration variables reaching the body of the loop during some iteration i of the loop be expressed as the r-dimensional vector $\vec{v_i}$. If there is a dependence between statements S1 and S2, then for some (possibly the same) iterations j and k of the loop, the array subscript calculated at S1 at iteration j must equal the subscript calculated at S2 at iteration k. That is, a dependence exists if for $1 <= w <= d$, $f_w(\vec{v_j}) = g_w(\vec{v_k})$, or equivalently, $f_w(\vec{v_j}) - g_w(\vec{v_k}) = 0$. If these equalities holds subject to the constraints on the loop iteration variables imposed by the loop bounds, then S1 and S2 access the same element of array A at some point during the execution of the loop. The basic job of an optimizing compiler is to prove that this inequality cannot hold subject to the bounds on the loop iteration variables (meaning that all iterations of the loop body can be executed in parallel), or else to attempt to determine exact conditions under which the equality holds (meaning that a subset of the iterations may be able to be run in parallel).

Note that in order to run the entire body in parallel, all pairs of array access statements S1 and S2 within the body must be tested for dependence. Thus, the complexity of the dependence analysis increases as the nesting level of the loop and especially as the size of the loop body increases. As integer programming is in general an NP-complete problem, dependence tests must approximate solutions in order to keep compilation times from becoming prohibitively long.

Thus we see that the crux of the matter is to determine whether the equality $f(\vec{v_j}) - g(\vec{v_k}) = 0$ holds given a set of inequality constraints on the loop iteration variables combined with the requirement that any solution be restricted to the integer domain. Rather than dealing with a system of *inequalities* (the bounds on the loop iteration variables arising from the loop limits) and a linear *equality* (arising from equating the subscript expressions in the array accesses), we shall see that the compiler converts the problem to one involving a system of linear inequalities by treating the equality constraint as a pair of inequality constraints. This is the integer programming problem optimizing compilers must solve, and the raison d'etre for the dependence tests I shall examine. How well and how quickly the tests can solve these problems has enormous impact on both compile time and run time performance of critical scientific applications.

The following sections give a brief overview of the techniques I will discuss throughout the remainder of this thesis. Each will be described in greater detail in subsequent sections, along with a discussion of data dependence analysis (especially as it applies to arrays) and a review of some mathematical principles that arise during the analysis of the algorithms.

Real Valued Techniques

One approach to efficiently approximating a solution to the problem of determining whether a system of linear inequalities has an integer solution is to relax the integer solution requirement and to instead treat the integer programming problem as a linear programming problem. That is, instead of insisting that the loop iteration variables are strictly integer valued, they are assumed to hold real values. This simplifies the problem considerably, as linear programming problems can be solved in polynomial time in general. If this approximation proves that no real-valued solution to the dependence equality exists, then obviously it has also proven that no integer value solution exists, and there is no dependence. If on the other hand these tests conclude that a real solution exists or may exist to the problem, we do not know conclusively whether the solution is an integer solution. In this case, the compiler must conservatively assume that the solution is integer.

Banerjee Extreme Value Test The Banerjee Extreme Value Test [3] finds the minimum and maximum values a linear expression can assume subject to (real-valued) variable constraints. The test defines a^+ (the positive part) and a^- (the negative part) of a real number a and uses that value to maintain an *interval equation* as it progressively examines the use of loop iteration variables in the array subscript expression. The test is guaranteed to execute in $O(r)$ time, where r is the nesting level of the loop. This test has been enormously influential, and Banerjee was the first to systematically define the dependence problem in mathematical terms. Although limited in the domain of iteration space polyhedra it is applicable to, the test is both extremely efficient and effective.

Fourier-Motzkin Variable Elimination (FMVE) Unlike the Banerjee Extreme Value Test, which was specifically crafted for use in solving the data dependence problem, Fourier Motzkin Variable Elimination [87], is a general linear programming technique that determines whether real solutions to a bounded linear programming problem can exist. It employs a technique roughly equivalent to Gauss-Jordan variable elimination to progressively reduce a system to simpler forms until the system can be trivially inspected for the possible existence of a solution. It has worst case double exponential complexity in the number of loop iteration variables.

Integer Valued Techniques

Despite the fact that integer programming is NP-complete in general, the question of whether a system of linear inequalities has an integer solution can be answered exactly and efficiently in a number of cases. Two of the predominant tests in the literature refine real valued techniques to the integer domain, and another relies upon a basic theorem from number theory.

GCD Test Consider the simple linear equality

$a_1x_1 + a_2x_2 + \ldots + a_{n-1}x_{n_1} + a_nx_n = c$

where:

$a_i, c \in \mathbb{Z}, 1 <= i <= n$

An integer solution to this equality exists if and only if the greatest common divisor of $a_1, a_2, \ldots, a_{n-1}, a_n$ also evenly divides c. Thus, the GCD offers a simple way of disproving the existence of a dependence: if the GCD of the LHS coefficients does not evenly divide the RHS, then no dependence can exist. Unfortunately, it is often the case that at least one of the LHS coefficients is ± 1, rendering the test ineffective. As we shall see, however, the GCD test is incorporated in some form in the other tests I will examine to potentially disprove a dependence in an intermediate phase of the test's execution.

I-Test The I-Test [78] is a refinement of the Banerjee Extreme Values Test to the integer domain. It grew out of the empirical observation that in the vast majority of cases, when the Banerjee test indicated that a linear solution to a set of inequalities existed, that solution was in fact an integer solution. The I-Test refines the Banerjee Test by applying the so-called *I-Test Condition* to the interval equation maintained at each step of the Banerjee test. We shall see that in many cases this has the effect of determining whether a linear solution detected by the Banerjee Test is an integer solution or not. The I-Test is always polynomial in the loop nesting depth, but is not always able to conclusively determine whether a dependence exists.

Omega Test The Omega Test [81] is a refinement to Fourier Motzkin Variable Elimination that allows it to determine whether a real solution detected by FMVE is actually an integer solution. The refinement is based on applying the *Dark Shadow Inequality* at each step of FMVE's variable elimination process. It is therefore able in frequently occurring cases to efficiently determine exactly whether or not a system of linear inequalities is solvable in integers or not. It is an exact test, although in the worst case is prohibitively expensive. As it is based on FMVE, the Omega Test has worst case exponential complexity in the nesting level of the loop body. Additionally, the Omega Test can resort to an exhaustive search of the iteration space to prove or disprove an integer solution.

2.3. Related Work

Although the major contribution of this thesis is showing the isomorphism between the I-Test and Omega Test under conditions in which both are applicable, it is nonetheless worthwhile to provide a broad outline of relevant previous work in dependency analysis.

The Lambda test [53] is a very intuitive test that extends the ideas behind the Banerjee Extreme Value test to handle multidimensional array references with coupled subscripts, but it is only able to prove real solutions. The Power Test [101] has much in common with the Omega test. They both use the Fourier Motzkin Variable Elimination method to prove consistency or inconsistency among the dependence problem constraints. A significant difference is that the Power Test uses simplified constraints after performing the

Extended GCD test [3], and that it does not force itself to produce an exact answer when this is going to be expensive. The Range Test [8] is a different way of applying the Banerjee Extreme Value test, which makes it suitable for problems involving symbolic variables and non-linear constraints. This test also introduces the notion of constraint propagation that can simplify problems with coupled subscripts.

Maydan et al. [57] proposed a small set of efficient algorithms, each one exact for special case inputs, combined with FMVE as an expensive back up test. They show empirically that they derive an exact answer in all cases on the Perfect Benchmarks [54]. Goff et al. [40] proposed a dependence testing scheme based on classifying pairs of subscripts. This approach is based on the fact that most array references in scientific programs are fairly simple. Efficient and exact tests are presented for certain classes of commonly occurring array references involving zero index variables (ZIV) and single index variables (SIV). In case of multiple index variable (MIV) subscripts their techniques rely on the Banerjee Extreme Value test. The I-Test is exact and more general than all special exact SIV cases. The experiments in [80] also show that the I-Test is exact in many cases that the Banerjee Extreme Value test is inconclusive and it is also as efficient as the Extreme Value test.

Petersen and Padua [74] performed an experimental evaluation of a proposed sequence of dependence tests on the Perfect Benchmarks and the linear algebra libraries, Eispack and Lapack. This sequence consists of the Banerjee Extreme Value test, an integer programming test based on the Simplex method, and the Omega test. In case of an Extreme Value test inconclusive answer, exponential tests such as integer programming and the Omega test were applied to elicit a yes or no answer. Their results indicated that the Extreme Value test is for all practical purposes as accurate as the more complex Omega test. Their results also show that most of the applicability of the integer programming and Omega tests is in proving the existence of dependences, in contrast with the Banerjee Extreme Value test, which can only disprove dependences. Experiments in [80] show that this can be done, i.e. the existence of dependences can be proven (efficiently in many cases) by applying the I-Test rather than by applying expensive integer programming tests or the Omega test.

Rauchwerger's LRPD test [83] takes a rather different approach to dependence analysis. Recognizing that compile time unknowns can limit the optimization a compiler can perform ahead of time, he proposes that testing be deferred until run-time when all critical information is resolved. He proposes a lightweight "inspector loop" that mimics the memory accesses of the original program, and that runs alongside a speculatively optimized version of that code. In the event that the inspector loop determines that the optimized code is unsafe, changes are reverted and the unoptimized code is run instead. Nonetheless, the effectiveness of this approach can still benefit from compile time analysis, especially if safety conditions are pre-determined at compile time via "protected variables" (which I will discuss in chapter 6).

Several studies have acknowledged the limitations of certain dependence analysis techniques presented in this thesis and they have shown that non-linear symbolic analysis is necessary in order to uncover parallelism in scientic applications [43],[32]. Data dependence analysis has evolved over the years to consider more complex instances of the dependence problem. However, future developments are based on the fundamentals of the data dependence tests and their relationships considered in this thesis.

2.4. Comparing the Techniques

In the following chapters, I will present the aforementioned tests more formally, and then will proceed to demonstrate the close equivalence of the Fourier-Motzkin method to the Banerjee Bounds test, and of the Omega Test to the I-Test. Since the Fourier-Motzkin method and Omega Test are more general than the Banerjee Bounds and I-Test techniques, in order to make the comparisons more meaningful I will assume the tests are applied to instances of the dependence problem to which all tests can be effectively applied.

This will require the following conditions on problem instances (the reasons for which to be made clear as I examine the tests in the following chapters):

2.4.1. Loop Bounds

Because the Banerjee Bounds and I-Tests eliminate terms by calculating the minimum and maximum values they can achieve subject to the bounds on the loop iteration variables, they effectively require that bounds on all loop iteration variables be known and constant. I make that assumption in the analysis in chapters 3 and 4. Although Banerjee extended his test to calculate inexact (conservative) bounds in *trapezoidal* regions where bounds on a loop at nesting level j are linear expressions among iteration variables $x_1, x_2, \ldots, x_j$, the I-Test assumes simple rectangular regions with known and constant bounds. As mentioned in [80] and [74], researchers have tried to address these limitations in their empirical comparisons by either attempting to extend Banerjee's test to accommodate infinite bounds or by substituting large constants for unknowns to allow the I-Test to attempt to proceed (with potential loss of accuracy in both cases). In contrast, the Fourier-Motzkin and Omega techniques are able to handle unbound variables and trapezoidal regions without modification.

2.4.2. Unit Coefficient

As I will detail in chapter 4, both the I-Test and Omega Tests require that at least one term in the equality arising from the array subscript be ± 1. Whereas the Omega Test includes a pre-processing phase that forces that condition to be met (detailed in section 6.1.1), the I-Test does not. Consequently, I will assume that there is a unity coefficient present initially in the subscript equality. As demonstrated in [80], this condition is met almost always in practice.

2.4.3. Multi-Dimensional Arrays

Refer again to section 2.2.1, and in particular to Listing 2.6. In order for two instances j and k of the same or distinct references nested in a loop of depth r ($r \geq 1$) to an d-dimensional array ($d > 1$) to refer to the same element, iteration vectors $\vec{v}_j$ and $\vec{v}_k$ must cause corresponding array subscript expressions in the references to *simultaneously* be equal. Using the terminology in section 2.2.1, a dependence exists if $f_w(\vec{v}_j) - g_w(\vec{v}_k) = 0$ for all $1 <= w <= d$. As an example, consider the simple loop in Listing 2.7 and the problem of determining whether there is a loop carried output dependence from statement $S1$ to $S1$. Although the first subscript expression produces the same index of 8 when $I = 10$ and $J = 0$ as it does in a later iteration when $I = 11$ and $J = 1$, and although the second subscript expression produces the same index of 14 when $I = 10$ and $J = 1$ as it does in a later iteration when $I = 11$ and $J = 0$, no two assignments to I and J allowed within the iteration space will cause the two array subscript expressions to produce the same *pair* of values.

```
for I = 10 to 100 do
    for J = 0 to 7 do
        ⋮
        S1: A[I-J-2, I+J+3] = ...;
        ⋮
    end
end
```
Listing 2.7: Example of multi-dimensional array access with coupled subscripts

Listing 2.7 represents a problem where the subscripts are said to be *coupled*, because at least one iteration variable participates in more than one subscript expression (in this case, both do). On the other hand, if no loop iteration variable appears in more than one subscript expression, the subscripts are said to be *separable*.

As the Banerjee Bounds and I-Tests deal with only a single constrained inequality, in the presence of multi-dimensional arrays, the tests essentially resort to testing each subscript independently. If one of the d subscript tests indicate that no dependence can exist, then the tests report that no dependence exists between the two statements overall. On the other hand, if the subscripts are separable and the I-Test detects an integer solution to all individual subscripts tests, it reports that a dependence definitely exists. As the Banerjee test cannot distinguish between real and integer solutions, if all separable subscript tests indicate a real solution, it will return a *maybe* answer, as the solutions may or may not be integer. Finally, in the case of coupled subscripts, if the tests are unable to disprove a dependence in any of the d subscript problems, then the tests will report that a dependence *may* exist, as these simpler tests cannot determine if any pair of loop iteration variable assignments *simultaneously* cause all corresponding subscript expressions to achieve equal values.

The Fourier-Motzkin and Omega Tests consider all dimensions of the array access at once, as they determine simultaneous solutions to systems of constrained linear equalities (returning *no* or *maybe* in the case of Fourier-Motzkin, and *yes* or *no* in the case of the Omega Test).

Thus, in order to ensure that all the tests I consider are equally applicable in case of multi-dimensional array access, I will assume that all array accesses are either single-dimensional or else consist of separable subscript expressions.

Chapter 3

Real Valued Dependency Tests: Fourier Motzkin and Banerjee Bounds

3.1. Preface

As we've discussed, data dependence analysis is the key to the detection of implicit parallelism in sequential programs. It provides the compiler with the necessary information to perform valid transformations such as those for improving memory locality, load balancing, and efficient scheduling. Furthermore, data dependence information is essential in detecting loop iterations that can be executed in parallel on multicore and multiprocessor architectures. Data dependence relations capture the essential ordering constraints among statements in a program that have to be preserved in any compiler transformation to ensure the validity of the generated code. The dependence problem is equivalent to integer programming, which is an NP-complete problem and so cannot be efficiently solved in general. Nonetheless, certain instances of the problem are solvable in polynomial time. A number of data dependence tests have been proposed in the literature. In each test there are different tradeoffs between accuracy and efficiency. Data dependence tests must always be conservative (namely dependence is assumed if independence cannot be proved) to prevent unsafe optimizations. Since scientific applications have been a major focus of compiler optimization and automatic parallelization, most of the work in data dependence analysis has considered programs involving array references inside loop nests. An r-nested loop including references to a d-dimensional array is depicted in Listing 3.1.

```
for $I_1$ = $L_1$ to $U_1$ do
    for $I_2$ = $L_2$ to $U_2$ do
        ⋮
        for $I_r$ = $L_r$ to $U_r$ do
            ⋮
            S1: A[$f_1(I_1,I_2,\ldots,I_r)$, $f_2(I_1,I_2,\ldots,I_r)$, ..., $f_d(I_1,I_2,\ldots,I_r)$] = ...;
            S2: ... = A[$g_1(I_1,I_2,\ldots,I_r)$, $g_2(I_1,I_2,\ldots,I_r)$, ..., $g_d(I_1,I_2,\ldots,I_r)$] ;
            ⋮
        end
        ⋮
    end
end
```

Listing 3.1: Example of a nested loop

For the above loop nest the data dependence problem between an instance (x_1, x_2, ...,x_r) of S1 and an instance (x_{r+1}, x_{r+2}, ..., x_{2r}) of statement S2 can be formulated as follows:

$$\begin{aligned} f_1(x_1,x_2,\ldots,x_r) &= g_1(x_{r+1},x_{r+2},\ldots,x_{2r}) \\ f_2(x_1,x_2,\ldots,x_r) &= g_2(x_{r+1},x_{r+2},\ldots,x_{2r}) \\ &\vdots \\ f_d(x_1,x_2,\ldots,x_r) &= g_d(x_{r+1},x_{r+2},\ldots,x_{2r}) \end{aligned} \tag{3.1}$$

where

$$\begin{aligned} L_1 \leq \ & x_1,x_{r+1} \leq U_1 \\ L_2 \leq \ & x_2,x_{r+2} \leq U_2 \\ & \vdots \\ L_r \leq \ & x_r,x_{2r} \leq U_r \end{aligned} \tag{3.2}$$

Note that (3.1) is a set of equalities arising from array subscript expression and (3.2) is a set of inequalities formed by the loop bounds. The data dependence tests I consider require that the subscripts $f_1(),f_2(),\ldots,f_d(),g_1(),g_2(),\ldots,g_d()$ be affine expressions of the enclosing loop iteration variables. These tests must determine if integer solutions exist to the system of linear equations in (3.1) subject to the constraints in (3.2).

In this chapter I consider data dependence problems involving single dimensional arrays or multi-dimensional arrays with uncoupled subscripts. In this case, each variable x_k appears in at most one of the d equations in (3.1) and in one of the inequalities in (3.2). We thus can express the dependence problem as a set of d independent linear equalities of the form:

$$\sum_{1\leq i\leq n} a_i x_i = c \quad \text{(where } n = 2r\text{)} \tag{3.3}$$

subject to the constraints in (3.2).

For consistency, the tests I examine deal with a system of linear inequalities only. Thus, I express a linear equation of the form of (3.3) as a pair of inequalities:

$$c \leq \sum_{1\leq i\leq n} a_i x_i \leq c \tag{3.4}$$

In the following sections, I will examine how several data dependence tests attempt to determine the existence of integer solutions to a set of inequalities of the form (3.4) subject to the constraints in (3.2). All tests operate by repeatedly removing a term from the center summation of (3.4), and adjusting the upper and lower bounds on the RHS and LHS of (3.4), respectively, by taking into consideration the variable's loop bounds from (3.2) and variable's coefficient in (3.4). This has the effect of "spreading apart" the constant bounds on the LHS and RHS. We can assume without loss of generality that $|a_i| \leq |a_i + 1|$, for $1 \leq i \leq n-1$. We represent (3.4) after $k-1$ terms have been eliminated as:

$$c_L \leq \sum_{k \leq i \leq n} a_i x_i \leq c_U \tag{3.5}$$

where initially $c_L = c_U = c$ and $k = 1$. The shorthand notation in (3.5) is equivalent to the pair of linear inequalities:

$$\sum_{k \leq 1 \leq n} a_i x_i - c_U \leq 0 \tag{3.6}$$

$$-\sum_{k \leq 1 \leq n} a_i x_i + c_L \leq 0 \tag{3.7}$$

In this chapter I will describe the fundamental relationship between the Banerjee Extreme Value test and Fourier Motzkin Variable Elimination (FMVE).

3.2. Overview of FMVE

Fourier-Motzkin Variable Elimination (FMVE) [26] determines whether a real solution exists to a system of linear equations in the form of (3.1) subject to inequality constraints in the form of (3.2). The lack of a real solution means that no integer solution exists and therefore, no dependence exists either. However, the fact that a real solution exists does not imply the existence of an *integer* solution. This means that FMVE cannot prove data dependence, but only disprove it.

FMVE determines the existence of a real solution to a system of linear inequalities by eliminating variables. Each variable elimination produces an equivalent system with one less variable. In general, the number of inequalities in the system may increase exponentially as variables are eliminated, although this rarely occurs when FMVE is applied to data dependence problems. Conceptually, eliminating a variable finds the $n-1$ dimensional shadow cast by an n-dimensional object. FMVE eliminates variables in order of increasing coefficient absolute value. A variable x_k is eliminated by scaling and combining pairs of upper and lower bounds on x_k so as to produce a zero coefficient for x_k in all resulting combinations. This is the same principle used by such methods as Gauss-Jordan elimination to progressively reduce systems of linear inequalities to simpler but equivalent forms. In general, given the following upper bound on x_k:

$$a_k x_k \leq \alpha \qquad \text{(where } a_k > 0\text{)} \tag{3.8}$$

and the following lower bound on x_k:

$$a'_k x_k \leq \beta \qquad \text{(where } a'_k < 0\text{)} \tag{3.9}$$

we eliminate x_k by multiplying (3.8) by $|a'_k|$, multiplying (3.9) by a_k, and combining the results, which produces:

$$a_k \beta \leq 0 \leq |a'_k| \alpha \tag{3.10}$$

Note that this inequality does not guarantee that an integer multiple of $|a'_k|a_k$ exists between the upper and lower bounds. Dropping the center term produces the following inequality involving one less variable:

$$a_k\beta \leq |a'_k|\alpha \tag{3.11}$$

If (3.11) is inconsistent (meaning of the form $c \leq 0$, where c is a constant and $c > 0$), no real solution exists to the system. We refer to this condition as a *contradiction*. In addition, (3.11) is ignored if it contains only constants and is not contradictory. Once all lower bounds on x_k have been combined with all upper bounds on x_k, the resulting inequalities in the form of (3.11) are added to the system and all inequalities containing x_k are removed from the system. FMVE continues eliminating variables in this fashion until either a contradiction is reached, in which case it stops and reports that no real solution exists to the system, or until all variables have been eliminated without contradiction, in which case it reports that a real solution exists to the system.

The preceding discussion describes how FMVE works in the general case. However, in the case of the data dependence problems I consider, a variable x_k appears in only the following expressions:

$$-\sum_{k\leq i\leq n} a_i x_i + c_L \leq 0 \tag{3.12}$$

$$\sum_{k\leq i\leq n} a_i x_i - c_U \leq 0 \tag{3.13}$$

$$-x_k + L_k \leq 0 \tag{3.14}$$

$$x_k - U_k \leq 0 \tag{3.15}$$

where c_L, c_U, L_k, and U_k are integer constants, and $L_k \leq U_k$. Inequalities (3.12) and (3.13) arise from an array subscript expression, whereas (3.14) and (3.15) are derived from the loop bounds on x_k.

Suppose we wish to eliminate x_k from inequalities (3.12)-(3.15). We first isolate x_k from (3.12) and (3.13) respectively, producing:

$$-a_k x_k - \sum_{k+1\leq i\leq n} a_i x_i + c_L \leq 0 \tag{3.16}$$

$$a_k x_k + \sum_{k+1\leq i\leq n} a_i x_i - c_U \leq 0 \tag{3.17}$$

If $a_k > 0$ in (3.3), then (3.16) is a lower bound on x_k, and (3.17) is an upper bound on x_k. Combining the lower bound (3.16) with the upper bound (3.17) produces the inequality $c_L \leq c_U$ (which I will show is always true in Section 3.2.1. Likewise, combining the lower bound (3.14) with the upper bound (3.15) produces $L_k \leq U_k$, which is trivially true.

Combining the lower bound (3.16) with the upper bound (3.15) produces

$$c_L - a_k U_k \leq \sum_{k+1\leq i\leq n} a_i x_i \tag{3.18}$$

Next, we combine the upper bound (3.17) with the lower bound (3.14), yielding

$$\sum_{k+1\leq i\leq n} a_i x_i \leq c_U - a_k L_k \tag{3.19}$$

Combining (3.18) and (3.19), we see that when $a_k > 0$, the elimination of variable x_k produces:

$$c_L - a_k U_k \leq \sum_{k+1 \leq i \leq n} a_i x_i \leq c_U - a_k L_k \tag{3.20}$$

Having now eliminated x_k, we set $c_L = c_L - a_k U_k$, and $c_U = c_U - a_k L_k$. This leaves the system in the same form as shown in (3.12) through (3.15), except with one fewer variable. We can now continue with the elimination of the next variable.

If $a_k < 0$, then (3.16) is an upper bound on x_k, and (3.17) is a lower bound on x_k. Combining these bounds again produces the inequality $c_L \leq c_U$, and combining the lower bound (3.14) with the upper bound (3.15) produces $L_k \leq U_k$.

Combining the lower bound (3.17) with the upper bound (3.15) produces:

$$\sum_{k+1 \leq i \leq n} a_i x_i \leq c_U + |a_k| U_k \tag{3.21}$$

Combining of the lower bound in (3.14) with the upper bound (3.16) produces:

$$|a_k| L_k + c_L \leq \sum_{k+1 \leq i \leq n} a_i x_i \tag{3.22}$$

Combining (3.21) with (3.22) we see that when $a_k < 0$, the elimination of x_k yields:

$$c_L + |a_k| L_k \leq \sum_{k+1 \leq i \leq n} a_i x_i \leq c_U + |a_k| U_k \tag{3.23}$$

Having now eliminated x_k, we set $c_L = c_L + |a_k| L_k$, and $c_U = c_U + |a_k| U_k$. This leaves the system in the same form as shown in (3.12) (3.15), except with one fewer variable. We can now continue with the elimination of the next variable.

3.2.1. Relationship of the Bounds

Referring to (3.12) and (3.13), I claimed earlier that $c_L \leq c_U$ is always true. As shown in equations (3.20) and (3.23), c_L and c_U are adjusted to account for the coefficient and bounds of the eliminated variable. We now prove that after each variable is eliminated via FMVE, $c_L \leq c_U$.

Lemma 1. *At each step of FMVE, $c_L \leq c_U$.*

Proof. We use induction on k, which is the index of the next variable to be eliminated. Initially, $k = 1$, (because no variables have been eliminated). At this point we begin with the pair of inequalities shown in (3.6) and (3.7), and $c_L = c_U = c$, establishing the basis.

For the inductive step, assume that $c_L \leq c_U$ after $k-1$ eliminations. We next eliminate x_k, and the equalities in the systems containing x_k are those in (3.12)-(3.15).

Assuming that $a_k > 0$, the elimination of x_k produces the inequalities shown in (3.20). We need to show that $c_L - a_k U_k \leq c_U - a_k L_k$.

Since $L_k \leq U_k$ and $a_k > 0$, we have $a_k L_k \leq a_k U_k \Rightarrow -a_k U_k \leq -a_k L_k$. Combining $-a_k U_k \leq -a_k L_k$ with $c_L \leq c_U$, we obtain $c_L - a_k U_k \leq c_U - a_k L_k$.

Now assume that $a_k < 0$. The elimination of x_k produces the inequalities shown in (3.23). In this case, we need to show that $c_L + |a_k| L_k \leq c_U + |a_k| U_k$.

Since $L_k \leq U_k$ and $|a_k| > 0$, we have $|a_k| L_k \leq |a_k| U_k$. Combining $|a_k| L_k \leq |a_k| U_k$ with $c_L \leq c_U$, we obtain $c_L + |a_k| L_k \leq c_U + |a_k| U_k$. □

3.2.2. Example of FMVE

We illustrate how FMVE works with an example. Given the following linear equation and set of loop bounds:

$$x+4y+7z-34=0 \quad \text{where:} \begin{cases} 0 \le & x & \le 15 \\ -5 \le & y & \le 10 \\ 3 \le & z & \le 12 \end{cases}$$

we first create the equivalent system of linear inequalities:

$$\begin{array}{rcl} 34 \le & x+4y+7z & \le 34 \\ 0 \le & x & \le 15 \\ -5 \le & y & \le 10 \\ 3 \le & z & \le 12 \end{array} \tag{3.24}$$

We first eliminate x from the system since it has the smallest coefficient. Rearranging the first inequality in terms of x:

$$\begin{array}{rcl} 34-4y-7z \le & x & \le 34-4y-7z \\ 0 \le & x & \le 15 \\ -5 \le & y & \le 10 \\ 3 \le & z & \le 12 \end{array} \tag{3.25}$$

Comparing upper and lower bounds on x yields:

Lower Bound	Upper Bound	Combination	Result
$34-4y-7z \le x$	$x \le 34-4y-7z$	$34-4y-7z \le x \le 34-4y-7z$	$0 \le 0$
$34-4y-7z \le x$	$x \le 15$	$34-4y-7z \le x \le 15$	$19 \le 4y+7z$
$0 \le x$	$x \le 34-4y-7z$	$0 \le x \le 34-4y-7z$	$4y+7z \le 34$
$0 \le x$	$x \le 15$	$0 \le x \le 15$	$-15 \le 0$

The first and last results are consistent and involve only constant terms, and so are ignored. Combining the middle two results yields the inequalities $19 \le 4y+7z \le 34$. We add them to the system, and remove from it the inequalities involving x. We are then left with the simpler system:

$$\begin{array}{rcl} 19 \le & 4y+7z & \le 34 \\ -5 \le & y & \le 10 \\ 3 \le & z & \le 12 \end{array}$$

We next eliminate y. Rewriting in terms of y gives:

$$\begin{array}{rcl} -7z+19 \le & 4y & \le -7z+34 \\ -5 \le & y & \le 10 \\ 3 \le & z & \le 12 \end{array} \tag{3.26}$$

Comparing upper and lower bounds on y:

Lower Bound	Upper Bound	Combination	Result
$-7z+19 \le 4y$	$4y \le -7z+34$	$-28z+76 \le 16y \le -28z+136$	$-60 \le 0$
$-7z+19 \le 4y$	$y \le 10$	$-7z+19 \le 4y \le 40$	$-21 \le 7z$
$-5 \le y$	$4y \le -7z+34$	$-20 \le 4y \le -7z+34$	$7z \le 54$
$-5 \le y$	$y \le 10$	$-5 \le y \le 10$	$-15 \le 0$

produces:

$$\begin{aligned} -21 \leq & \quad 7z \quad \leq 54 \\ 3 \leq & \quad z \quad \leq 12 \end{aligned} \tag{3.27}$$

which we add to the system. We next eliminate z:

Lower Bound	Upper Bound	Combination	Result
$-21 \leq 7z$	$7z \leq 54$	$-147 \leq 49z \leq 378$	$-525 \leq 0$
$-21 \leq 7z$	$z \leq 12$	$-21 \leq 7z \leq 84$	$-105 \leq 0$
$3 \leq z$	$7z \leq 54$	$21 \leq 7z \leq 54$	$0 \leq 33$
$3 \leq z$	$z \leq 12$	$3 \leq z \leq 12$	$-9 \leq 0$

Since we have eliminated all variables and have not reached any contradictions, we conclude that the original system of linear inequalities has a real solution.

3.3. Overview of the Banerjee Extreme Value Test

The Banerjee Extreme Value [4] determines whether a real solution exists to a single linear equation in the form of (3.1) subject to inequality constraints in the form of (3.2). As with FMVE, the inability to distinguish between real and integer solutions makes the Banerjee Extreme Value test an inexact test. However, an additional potential source of inaccuracy exists in the Banerjee Extreme Value test: since it tests a single linear equation at a time (subscript by subscript testing) it can not prove the existence of a *simultaneous* real solution to a system of constrained linear equations arising from coupled array subscripts.

The Banerjee Extreme Value test finds the minimum and maximum values the linear expression in (3.4) can assume subject to the constraints in (3.2). The Banerjee Extreme Value test defines a^+ (the positive part) and a^- (the negative part) of a real number a as follows:

$$a^+ = \begin{cases} a & \text{if } a \geq 0, \\ 0 & \text{if } a < 0 \end{cases} \qquad a^- = \begin{cases} 0 & \text{if } a \geq 0, \\ -a & \text{if } a < 0 \end{cases}$$

The functions $Min(a_k x_k)$ and $Max(a_k x_k)$ are defined as:

$$\begin{aligned} Min(a_k x_k) &= {a_k}^+ L_k - {a_k}^- U_k \\ Max(a_k x_k) &= {a_k}^+ U_k - {a_k}^- L_k \end{aligned}$$

For each term $a_k x_k$ in the center summation of inequality (3.4), we calculate $Min(a_k x_k)$ and $Max(a_k x_k)$, and subtract these values from the RHS and LHS, respectively, of the inequality. As with FMVE, this produces an expression in the form of (3.5), where c_L and c_U are the adjusted bounds after eliminating the term $a_k x_k$. We continue removing terms in this manner, and each such elimination spreads apart the bounds in (3.5) according to the eliminated variables coefficient and loop iteration bounds, as is the case with FMVE. When all the terms have been eliminated from the center summation of (3.5), we are left with a pair of inequalities of the form:

$$c - \sum_{1 \leq i \leq n} Max(a_i x_i) \leq 0 \leq c - \sum_{1 \leq i \leq n} Min(a_i x_i)$$

If these inequalities are satisfied, we report that a real solution exists. Otherwise, we report no real solution exists.

3.3.1. Example of Banerjee Extreme Value Test

Using the same example shown in (3.24), the Banerjee Extreme Value test first removes x from the expression $34 \leq x+4y+7z \leq 34$ via the calculations:

$$\begin{array}{rrcccl}
 & 34 - Max(x) & \leq & 4y+7z & \leq & 34 - Min(x) \\
\Rightarrow & 34-((1)^{+}(15)-(1)^{-}(0)) & \leq & 4y+7z & \leq & 34-((1)^{+}(0)-(1)^{-}(15)) \\
\Rightarrow & 19 & \leq & 4y+7z & \leq & 34
\end{array}$$

Next, y is removed:

$$\begin{array}{rrcccl}
 & 19 - Max(4y) & \leq & 7z & \leq & 34 - Min(4y) \\
\Rightarrow & 19-((4)^{+}(10)-(4)^{-}(-5)) & \leq & 7z & \leq & 34-((4)^{+}(-5)-(4)^{-}(10)) \\
\Rightarrow & -21 & \leq & 7z & \leq & 54
\end{array}$$

Finally, z is eliminated:

$$\begin{array}{rrcccl}
 & -21 - Max(7z) & \leq & 0 & \leq & 54 - Min(7z) \\
\Rightarrow & -21-((7)^{+}(12)-(7)^{-}(3)) & \leq & 0 & \leq & 54-((7)^{+}(3)-(7)^{-}(12)) \\
\Rightarrow & -105 & \leq & 0 & \leq & 33
\end{array}$$

Note that the final inequality, $-105 \leq 0 \leq 33$, is the same as that obtained via FMVE. We proceed to show that this equivalence between FMVE and the Banerjee Extreme Value test is true in general.

3.4. Equivalence of FMVE and Banerjee Extreme Value Test

Lemma 1. *The variable elimination steps performed by the Banerjee Extreme Value test and FMVE are equivalent, and thus FMVE will detect the existence of real solutions if and only if the Banerjee Extreme Value test does so.*

Proof. We use induction on the number of variables eliminated so far. We show the inequalities produced after every variable elimination are identical.

For the basis FMVE and the Banerjee Extreme Value test begin with the same inequality, shown in (3.4). Now assume that after $k-1$ variables have been eliminated, the LHS and RHS of the inequality are the same using both FMVE and the Banerjee Extreme Value test. For consistency, I refer to the LHS of the inequality as c_L, and the RHS as c_U. We next eliminate variable x_k. There are two cases to consider. If $a_k > 0$, the Banerjee Extreme Value test proceeds as follows:

$$\begin{array}{rrcccl}
 & c_L - Max(a_k x_k) & \leq & \sum_{k+1 \leq i \leq n} a_i x_i & \leq & c_U - Min(a_k x_k) \\
\Rightarrow & c_L - ({a_k}^{+} U_k - {a_k}^{-} L^k) & \leq & \sum_{k+1 \leq i \leq n} a_i x_i & \leq & c_U - ({a_k}^{+} L_k - {a_k}^{-} U_k) \\
\Rightarrow & c_L - (a_k U_k - 0) & \leq & \sum_{k+1 \leq i \leq n} a_i x_i & \leq & c_U - (a_k L_k - 0) \\
\Rightarrow & c_L - a_k U_k & \leq & \sum_{k+1 \leq i \leq n} a_i x_i & \leq & c_U - a_k L_k
\end{array} \tag{3.28}$$

This result is identical to that produced by FMVE, shown in (3.20). If $a_k < 0$, the Banerjee Extreme Value test performs the following:

$$\begin{array}{rrcccl}
 & c_L - Max(a_k x_k) & \leq & \sum_{k+1 \leq i \leq n} a_i x_i & \leq & c_U - Min(a_k x_k) \\
\Rightarrow & c_L - ({a_k}^{+} U_k - {a_k}^{-} L^k) & \leq & \sum_{k+1 \leq i \leq n} a_i x_i & \leq & c_U - ({a_k}^{+} L_k - {a_k}^{-} U_k) \\
\Rightarrow & c_L - (0 + a_k L_k) & \leq & \sum_{k+1 \leq i \leq n} a_i x_i & \leq & c_U - (0 + a_k U_k) \\
\Rightarrow & c_L + |a_k| L_k & \leq & \sum_{k+1 \leq i \leq n} a_i x_i & \leq & c_U + |a_k| U_k
\end{array} \tag{3.29}$$

This result is identical to that produced by FMVE, shown in (3.23). We conclude that FMVE and the Banerjee Extreme Value test are equivalent. □

3.5. Summary

In this chapter I have presented the Fourier-Motzkin variable elimination technique and the Banerjee Bounds Test, which are two tests that determine whether a real solution exists to a system of linear inequalities. I have described the steps that each algorithm employs when solving the type of dependence problems I consider in this thesis, and gave a worked example of how each step solves the same representative dependence problem. Although the Fourier-Motzkin technique is applicable to more general dependence problems, I showed the two tests are isomorphic under the conditions presented in section 2.4. That is, each test takes the same steps and adjusts bounds identically as it eliminates terms, and ultimately will efficiently reach the same conclusion as to the existence of real solutions given the same dependence problem of the type I consider in this thesis. This result immediately suggests that replacing the Banerjee Bounds Test with Fourier-Motzkin elimination incurs no significant disadvantages, and to the contrary offer numerous advantages due to its ability to determine real solutions to problems involving iteration spaces too complex for the Banerjee Bounds (or even its extension to trapezoidal regions) to handle. In fact, since a *maybe* answer is effectively and conservatively treated the same a *yes* answer by the compiler, I suggest that if one is willing to trade slightly fewer broken dependencies for a significant reduction in compilation time [80], Fourier-Motzkin is a viable replacement for the Omega Test as well. This presents a slight complication in the case of direction vector testing, however. As we shall later see, the Omega Test protects variables in this case in an attempt to produce exact dependence distances rather than test the direction vector hierarchy.

Chapter 4

Integer Valued Dependency Tests: The Omega Test and the I-Test

In this chapter I examine the refinements to the tests considered in the previous chapter to the integer domain.

4.1. Overview of the Omega Test

The Omega test [81] is based on and refines FMVE to the integer domain. It determines whether an integer solution exists to a system of linear equations in the form of (3.1) subject to inequality constraints in the form of (3.2). It produces exact yes/no answers, but has worst case exponential time complexity.

Given an upper bound α on x_k:

$$a_k x_k \leq \alpha \qquad \text{(where } a_k > 0\text{)} \tag{4.1}$$

and a lower bound β on x_k:

$$a'_k x_k \leq \beta \qquad \text{(where } a'_k < 0\text{)} \tag{4.2}$$

we eliminate x_k by multiplying (4.1) by $|a'_k|$, multiplying (4.2) by a_k, and combining the results, which produces:

$$a_k\beta \leq |a'_k|a_k x_k \leq |a'_k|\alpha \tag{4.3}$$

This does not imply that an integer multiple of $|a'_k|a_k$ exists between the upper and lower bounds. This is the reason why FMVE, like the Banerjee Extreme Value test, cannot prove the existence of integer solutions, only real solutions.

Suppose that no integer multiple exists. Then there is an integer i such that:

$$|a'_k|a_k i < a_k\beta \leq |a'_k|a_k x_k \leq |a'_k|\alpha < |a'_k|a_k(i+1) \tag{4.4}$$

Since both $|a'_k|a_k i$ and $a_k\beta$ are multiples of a_k, we have:

$$a_k\beta - |a'_k|a_k i \geq a_k \quad \Rightarrow \quad -a_k\beta \leq -|a'_k|a_k i - a_k \tag{4.5}$$

and since both $|a'_k|\alpha$ and $|a'_k|a_k(i+1)$ are multiples of $|a'_k|$, we have:

$$|a'_k|a_k(i+1) - |a'_k|\alpha \geq |a'_k| \quad \Rightarrow \quad |a'_k|\alpha \leq |a'_k|a_k(i+1) - |a'_k| \tag{4.6}$$

By adding the last two inequalities we get:

$$|a'_k|\alpha - a_k\beta \leq |a'_k|a_k - a_k - |a'_k| \tag{4.7}$$

Inequality (4.7) gives the maximum possible distance between the upper bound $|a'_k\alpha|$ and the lower bound $a_k\beta$ on $|a'_k|a_kx_k$ such that an integer multiple of $|a'_k|a_k$ is not guaranteed to lie between them. Conversely, if the distance between the upper and lower bounds on $|a'_k|a_kx_k$ exceeds $|a'_k|a_k - a_k|a'_k|$, we are guaranteed an integer multiple of $|a'_k|a_k$ falls somewhere between them. Specifically, an integer solution exists if:

$$\begin{array}{rrcl} & |a'_k|\alpha - a_k\beta & > & |a'_k|a_k - a_k - |a'_k| \\ \Rightarrow & |a'_k|\alpha - a_k\beta & \geq & |a'_k|a_k - a_k - |a'_k| + 1 \end{array}$$

Rewriting the final inequality above, we produce the "dark shadow" inequality:

$$|a'_k|\alpha - a_k\beta \geq (a_k - 1)(|a'_k| - 1) \tag{4.8}$$

This inequality is the Omega test refinement to FMVE, as it specifies the condition under which an integer solution exists to a constrained system of linear inequalities. Note that if either $|a_k| = 1$ or $|a'_k| = 1$, the dark shadow inequality will be identical to the real shadow inequality in (3.11). This is termed an *exact* projection. Since in the case of the data dependence problems we consider the coefficients in the loop bounds expressions (3.14) and (3.15) are 1 and -1, the only combination where the dark shadow inequality differs from the real shadow inequality is in the combination of the lower and upper bounds arising from the array subscript expressions (3.16) and (3.17). This gives rise to the real shadow inequality:

$$a_kc_L \leq |a'_k|a_kx_k \leq |a'_k|c_U \quad \Rightarrow \quad a_kc_L \leq |a'_k|c_U$$

The corresponding dark shadow inequality is produced by simply adding the extra constant term to the LHS of the real shadow inequality:

$$a_kc_L + (a_k - 1)(|a'_k| - 1) \leq |a'_k|c_U$$

Note also that the dark shadow inequality requires that at least one coefficient be ± 1 in the original equation, as the distance between the bounds c_L and c_U is initially 0. If none of the terms in the original equation is ± 1, then the first application of the dark shadows inequality will fail.

As discussed in Section 3.2, FMVE calculates the real $n-1$ dimensional shadow cast by an n dimensional polyhedron. If this real shadow contains no integers, then no integer solution exists to the system. However, if the real shadow does contain integers, we cannot be certain that they correspond to integers in the original object. The dark shadow inequality calculates a conservative sub-shadow wherein every integer point is guaranteed to correspond to an integer point in the original object. If you imagine the original polyhedron as being translucent, the dark shadow is the sub-shadow under the object where the polyhedron is of at least unit thickness. The Omega test essentially calculates two bounds for each variable combination: one corresponding to FMVE's real shadow, the other to the dark shadow. The cost of the additional shadow calculation is minimal, since it involves the addition of a constant term to each inequality. After a variable x_k has been eliminated, a check is made to see if the dark shadow calculation has produced a contradiction. If it has not, the Omega test continues to eliminate variables. If it manages to eliminate all variables without producing a contradiction in the dark shadow calculation, an integer solution must exist to the original system of linear equations, and the Omega test reports "yes".

On the other hand, if the dark shadow inequality from the elimination of the variable x_k produces a contradiction, the dark shadow is empty, and integer solutions cannot be guaranteed. The Omega test then checks if the real shadow is also empty by checking the real shadow inequality for contradictions (essentially resorting to standard FMVE). If the real shadow is empty (i.e. produces a contradiction), then no real (and thus no integer) solution exists, and the Omega test reports “no”. However, if the real shadow is not empty, then integer solutions may still exist, and the Omega Test begins an essentially exhaustive search of the solution space, recursively generating and solving integer programming problems until integer solutions are either found or disproved.

4.1.1. Example of the Omega Test

Using the running example shown in 3.25, and eliminating the variable x, we see that combining upper and lower bounds on x produces the same results as in FMVE, because the coefficients of x are 1 in all inequalities, and the constant added to the real shadow inequality (show in parenthesis) to form the dark shadow inequality is therefore $(1-1)(1-1) = 0$:

Lower Bound	Upper Bound	Combination	Result
$34-4y-7z \le x$	$x \le 34-4y-7z$	$34-4y-7z(+0) \le x \le 34-4y-7z$	$0 \le 0$
$34-4y-7z \le x$	$x \le 15$	$34-4y-7z(+0) \le x \le 15$	$19 \le 4y+7z$
$0 \le x$	$x \le 34-4y-7z$	$0(+0) \le x \le 34-4y-7z$	$4y+7z \le 34$
$0 \le x$	$x \le 15$	$0(+0) \le x \le 15$	$0 \le 15$

Rewriting the remaining inequalities in terms of y leaves the same system as shown in 3.26. We now proceed to eliminate y. The constant added to the real shadow inequality to form the dark shadow inequality in the first row is $(4-1)(4-1) = 9$.

Lower Bound	Upper Bound	Combination	Result
$-7z+19 \le 4y$	$4y \le -7z+34$	$-28z+76(+9) \le 16y \le -28z+136$	$-51 \le 0$
$-7z+19 \le 4y$	$y \le 10$	$-7z+19(+0) \le 4y \le 40$	$-21 \le 7z$
$-5 \le y$	$4y \le -7z+34$	$-20(+0) \le 4y \le -7z+34$	$7z \le 54$
$-5 \le y$	$y \le 10$	$-5(+0) \le y \le 10$	$0 \le 15$

The elimination of z begins with rewriting the remaining inequalities in terms of z, leaving the system shown in 3.27. The constant added to the real shadow inequality to form the dark shadow inequality in the first row is $(71)(71) = 36$.

Lower Bound	Upper Bound	Combination	Result
$-21 \le 7z$	$7z \le 54$	$-147(+36) \le z \le 378$	$-489 \le 0$
$-21 \le 7z$	$z \le 12$	$-21(+0) \le 7z \le 84$	$-105 \le 0$
$3 \le z$	$7z \le 54$	$21(+0) \le 7z \le 54$	$0 \le 33$
$3 \le z$	$z \le 12$	$3(+0) \le z \le 12$	$-9 \le 0$

Since we have eliminated all variables and have not reached any contradictions, we conclude that the dark shadow is not empty, and report that integer solutions exist.

4.2. Overview of the I-Test

The I-Test [78] is based on and refines the Banerjee Extreme Value test to the integer domain. Whereas the Banerjee Extreme Value test is unable to distinguish real from integer solutions, the I-Test can conclusively prove or disprove the existence of integer solutions (and hence dependences) in many cases. The I-Test arose from an observation that most real solutions predicted by the Banerjee Extreme Value test are in fact integer solutions. This insight led to the development of a condition, which if met, guarantees that a given linear expression achieves every integer value between the minimum and maximum values calculated by the Banerjee Extreme Value test. This accuracy condition states the necessary and sufficient relationship between the coefficients of loop index variables to the range of values they can assume in order to guarantee that every integer value between the extreme values is achievable. In practice these conditions are met so frequently that the I-Test is in most cases a linear time exact test. The I-Test in some cases can disprove a dependence even if the Banerjee Extreme Value test fails to do so.

The I-Test is applied on a subscript by subscript basis in the case of multidimensional array references. If dependence is disproved when considering any of the subscripts then there can be no dependence. However, if all the individual tests produce "yes" answers, a dependence is known to exist only if the subscripts are uncoupled, as the I-Test (like the Banerjee Extreme Value test) does not test for simultaneous solutions to systems arising from array references with coupled subscripts. The I-Test refines the Banerjee Extreme Value test in the same fashion as the Omega test extends FMVE: it adds a condition (called the accuracy condition), which if met, allows a variable to be safely eliminated from the system. "Safely" means the reduced system created by the elimination has an integer solution if and only if the original system has an integer solution. Conceptually, it performs the same calculations done in the Banerjee Extreme Value test: eliminating a term at a time from the center of a linear equality in the form of (3.5), and subtracting the term's maximum and minimum value from the LHS and RHS, respectively, of the inequality. In I-Test terminology, the upper and lower bounds (c_U and c_L, respectively, from (3.5)) form the RHS of an *interval equation*, which is a notation for a set of linear equations maintained as variables are eliminated, while the set of terms yet to be eliminated from the center of (3.5) form the LHS of this interval equation. The I-Test determines if the original linear equality in (3.3) has an integer solution by checking if the magnitudes of the coefficients of variables being eliminated are "small" relative to the distance between the upper and lower bounds in (3.2). This condition almost invariably arises in practice. We now describe the necessary relationship between variable coefficient and the bounds:

Let c_U and c_L be the current RHS and LHS, respectively, of the inequality in (3.5). We can move a term $a_k x_k$ from the center and adjust the lower and upper bounds accordingly if:

$$|a_k| \leq c_U - c_L + 1 \tag{4.9}$$

This implies, as in the case of the Omega test, that at least one term of the original linear equation has a coefficient of ± 1. If we remove all the terms and the interval equation contains 0, we report that an integer solution to the original linear equality in (3.3) exists. Otherwise, no integer solution exists. If we reach a point where a term cannot be moved, we continue moving terms anyway (essentially reverting to the Banerjee Extreme Value test), in the hope of disproving real solutions. If the result produces a contradiction, we know no real solutions exist, and "no" is returned. Otherwise, we know a real solution exists, but are unsure if an integer solution exists. In this case, the I-Test returns a "maybe" answer.

Note that the I-Test performs the same calculations as a term-by-term application of the Banerjee Extreme Value test, except that before moving a term, we check the magnitude of the coefficient against the length of the distance between the bounds. If all the terms qualify, the end interval equation will be identical to that produced by the standard Banerjee Extreme Value test.

4.2.1. Example of the I-Test

We use the interval equation format

$$\sum_{k \le i \le n} a_i x_i = [c_L, c_U]$$

This is to be understood as the set of linear equations

$$\begin{array}{rcl}
\sum_{k\le i\le n} a_i x_i & = & c_L \\
\sum_{k\le i\le n} a_i x_i & = & c_L + 1 \\
 & \vdots & \\
\sum_{k\le i\le n} a_i x_i & = & c_U - 1 \\
\sum_{k\le i\le n} a_i x_i & = & c_U
\end{array}$$

where as long as the accuracy condition in (4.9) is met, all equations are guaranteed to have an integer solution. If this condition is not met, then the bounds have the same meaning as in the Banerjee Extreme Value test: namely, the region in which a real solution exists. We continue with the running example system shown in 3.24

$$\begin{array}{rl}
x + 4y + 7z = [34, 34] & \text{(Initial system)} \\
4y + 7z = [19, 34] & \text{x moved because } (|1| \le 34 - 34 + 1) \\
7z = [-21, 54] & \text{y moved because } (|4| \le 34 - 19 + 1) \\
0 = [-105, 33] & \text{z moved because } (|7| \le 54 - (-21) + 1)
\end{array}$$

Since $-105 \le 0 \le 33$, the I-Test returns a “yes” answer, meaning that integer solutions definitely exist to this system.

4.3. Equivalence of the Omega Test and the I-Test

The Omega and I-Test are equivalent when the I-Test is applicable. To show this, we need to compare the dark shadows inequality in (4.8) and the I-Test accuracy condition in (4.9). Recall that when we eliminate a variable x_k, we combine pairs of upper and lower bounds on x_k. The bounds are those described in (3.12) - (3.15). Note that x_k has a coefficient of ± 1 in three of the four combinations to be made in eliminating x_k. Recall also that in this case the dark inequality is inconsequential, since when either $a_k = 1$ or $a'_k = 1$ in (4.8), the dark shadow inequality adds a constant of 0.

The one combination where the dark shadow inequality plays a role is the combination of (3.16) and (3.17), and is also where the I-Test accuracy condition compares the coefficient a_k to the range of the interval $c_U - c_L + 1$ to decide whether it was legal to move the term from the center to either side of the inequality. Since the coefficient of x_k is the same in both bounds (a_k), the dark shadow inequality becomes $a_k(c_U - c_L) \ge (a_k - 1)^2$. We now show that the Omega Test's dark shadow inequality (4.8) and the I-Test's accuracy condition (4.9) are equivalent in this case. Note that the coefficient a_k is positive. If the coefficient is negative, we simply reverse signs and swap the LHS and RHS of the inequality. To begin, we state and prove a simple lemma:

Lemma 1. *If $a_k, x \in \mathbb{Z}^+$ and $a_k \ge 1$, then $a_k x \ge (a_k - 1)^2$ if and only if $x \ge a_k - 1$.*

Proof. (if-part)

$$\begin{array}{rl}
x \ge a_k - 1 & \text{Given} \\
(a_k - 1)x \ge (a_k - 1)^2 & \text{(since } a_k \ge 1 \text{ implies } (a_k - 1) \ge 0) \\
a_k x \ge (a_k - 1)^2 & \text{(Since } a_k \ge (a_k - 1) \text{ and } x \ge 0)
\end{array}$$

□

Proof. only-if

$$\begin{aligned} a_k x &\geq (a_k - 1)^2 \\ a_k x &\geq {a_k}^2 - 2a_k + 1 \\ x &\geq a_k - 2 + \frac{1}{a_k} \\ x &\geq a_k - 1 \quad \text{(Since } a_k, x \in \mathbb{Z}^+\text{, and } a_k \geq 1\text{)} \end{aligned}$$

□

We now show that the dark shadows inequality is equivalent to the I-Test's inequality as seen in (4.9) when the coefficients of x_k in the bounds are equal:

$$\begin{aligned} a_k(c_U - c_L) &\geq (a_k - 1)^2 && \text{Dark Shadows inequality when } a_k = a'_k \\ c_U - c_L &\geq a_k - 1 && \text{Using Lemma 3 with } x = (c_U - c_L) \\ c_U - c_L + 1 &\geq a_k && \text{I-Test condition} \end{aligned}$$

and

$$\begin{aligned} c_U - c_L + 1 &\geq a_k && \text{I-Test Condition} \\ c_U - c_L &\geq a_k - 1 && \text{Rearranging} \\ a_k(c_U - c_L) &\geq (a_k - 1)^2 && \text{(Since } a_k \geq 1\text{. (Dark Shadows inequality))} \end{aligned}$$

This leads us to the following theorem:

Theorem 1. 1. *If the I-Test returns a yes answer, the Omega test will return a yes answer.*

2. *If the I-Test returns a no answer, the Omega test will return a no answer*

3. *If the I-Test returns a maybe answer, the Omega test will return a yes or no, but only after exhaustively searching between the upper and lower bounds on some variable x_k.*

Proof. (Claim 1) If the I-Test returns a yes answer, it is because all terms passed the I-Test's checks, and the resulting interval equation contained 0. If this is true, then by Lemma 3 all terms would pass the dark shadows inequality as well. Furthermore, because all the remaining comparisons involve a coefficient of 1, the calculation of the dark shadow will be identical to that of FMVE, which in turn is identical to the Banerjee Extreme Value test. Since the I-Test's final interval is equal to that obtained by the Banerjee Extreme Value test, the Omega test would also find no contradiction, and return yes.

(Claim 2) Similar to the first: if the I-Test returned no, it is because either all terms were moved and 0 did not lie between the interval bounds, or because a term could not be moved, but the Banerjee Extreme Value test found that no real solutions existed. In the former case, Lemma 3 ensures that the Omega test would also move all terms and produce the same results as FMVE, which are in turn the same results produced with the Banerjee Extreme Value test. Since the I-Test's final interval is equal to that obtained by the Banerjee Extreme Value test, the Omega test would make the same observation as the I-Test, and return no. In the latter case, the Omega test would also be unable to move the same term, and would revert to FMVE to try to rule out real solutions. If the Banerjee Extreme Value test ruled out real solutions, FMVE would as well, and the Omega test would return no.

(Claim 3) If the I-Test produced a maybe answer, it is because some term $a_k x_k$ did not pass the I-Test's check, and furthermore, the Banerjee Extreme Value test was unable to disprove real solutions. By Lemma 3, we know that the Omega test would also detect a contradiction in the real shadow's calculation after moving $a_k x_k$, and would attempt to disprove the existence of real solution by checking the real shadow produced by FMVE. Since the Banerjee Extreme Value test was unable to rule out real solutions, FMVE would also be unable to do so. Thus, the Omega test would begin exhaustively searching bounds on x_k for an integer solution. □

4.4. Multidimensional Arrays

The previous results apply to multidimensional arrays provided the subscript expressions are uncoupled. If so, then regardless of the number of array dimensions, when any of the tests we've considered eliminates a variable x_k, there are only four inequalities in the system that contain x_k. The first two arise from the inequality describing the loop bounds on x_k:

$$L_k \leq x_k \leq U_k$$

and the last two arise from the subscript expression in which x_k appears:

$$c_L \leq \sum_{k \leq i \leq n} a_i x_i \leq c_U$$

The tests we've considered will simply make a single pass through the variables in order of non-descending coefficients. As each variable is considered, the tests will attempt to eliminate it from one of the d subscript expressions in which it appears.

4.5. Summary

This chapter closely follows the outline established in chapter 3. I have presented the Omega Test and the I-Test, which are two tests that determine whether an integer solution exists to a system of linear inequalities. I have described how each test refines its real valued predecessor presented in chapter 3, Fourier-Motzkin Variable Elimination and the Banerjee Bounds Test, respectively. I have described the steps that each algorithm employs when solving the type of dependence problems I consider in this thesis, and gave a worked example of how each step solves the same representative dependence problem. I then showed the equivalence of the tests' refinement to the integer domain, that is, neither refinement subsumes the other. Although the Omega Test is applicable to more general dependence problems, I showed that the two tests are isomorphic under the conditions presented in section 2.4. I concluded with a discussion of dependence analysis in the presence of multi-dimensional arrays, which augments the discussion in section 2.4.3. The key observation is that when presented with the types of dependence problems considered in this thesis (as described in section 2.4), the Omega test does not require exponential time to produce an exact answer (i.e. fall into the "Omega Nightmare") without the I-Test necessarily being forced to return an inexact "maybe" answer. Said another way, the Omega Test only incurs the expense of the exhaustive search if it is attempting to provide more information than the I-Test would have been able to provide. Conversely, the I-Test only produces a *maybe* answer when the Omega Test would require and expensive search in order to return a more accurate answer. However, if one were to relax the conditions in section 2.4 and present the tests with non-unity coefficients, unbound variables, more complex loop regions, or dependence problems arising from coupled subscripts, then it is indeed possible for the Omega Test to efficiently return exact answers when the I-Test is forced to return inexact answers. This suggests that if the Omega Test were augmented so as to accept a flag instructing it to forego the exhaustive search if it cannot efficiently return an exact answer, then it would be applicable to a more general class of dependence problems than the I-Test while at the same time delivering close to the same performance. The performance gap would be closed even more if upon receipt of this flag, the Omega Test were to forego the overhead associated with forcing a unity coefficient (to be detailed in chapter 6) if one did not already exist. Finally, I note that some researchers have suggested that the I-Test be used as a initial filter, and only invoke the Omega Test if it is not able to conclusively prove or disprove a dependency [77]. However, the results in this section show that the Omega test would reach the conclusion that an efficient answer was not possible as fast (or nearly so) as it would take the I-Test to return a “maybe” answer. In other words, this section suggests that there is very little to be gained by such a strategy, and that letting the Omega test handle the problem alone is a more reasonable approach.

Chapter 5

Extending the Comparison to Direction Vectors

5.1. Distance and Direction Vectors

If a dependence exists between statements S1 and S2 in Listing 3.1 on page 48, then at least two sets of values for the loop iteration variables resolve to the same array location. That is, (3.1) can be rewritten in vector notation as

$$\begin{aligned} f_1(\vec{x}) &= g_1(\vec{x'}) \\ f_2(\vec{x}) &= g_2(\vec{x'}) \\ &\vdots \\ f_d(\vec{x}) &= g_d(\vec{x'}) \end{aligned} \tag{5.1}$$

where the r-dimensional vectors $\vec{x}$ and $\vec{x'}$ (where r is the loop nesting level) represent two sets of iteration variables at statements S1 and S2, respectively, that refer to the same array location. Note that there may be many such pairs of vectors that give rise to data dependence. Compilers seek to characterize the relationship between such pairs of vectors in an attempt to optimize the loop nest. In particular, recall from chapter 2 that the *distance vector* ([57],[102]) is the vector difference $\vec{d} = \vec{x} - \vec{x'}$. Informally, it gives the number of times each loop iterates between accesses to the same location. Frequently, this distance is not constant. If only the sign of the elements of $\vec{d}$ are relevant to a particular optimization, and not necessarily the magnitude, then a less precise characterization called a *direction vector* $\vec{\psi}$ will suffice to describe the relationship between the vectors. Each element ψ of the direction vector is one of {'=', '<', '>'} if the corresponding element of $\vec{x}$ is respectively always equal to, less than, or greater than the same element of $\vec{x'}$. If the relationship between corresponding elements of $\vec{x}$ and $\vec{x'}$ is not constant (or is immaterial), then the corresponding element of $\vec{\psi}$ is set to '*'.

The feasibility of certain compiler optimizations (loop interchange, for example) depends on the compiler being able to determine if dependences exist with a particular type of direction vector. In order to make this determination, the compiler adds additional constraints to (3.2) and tests whether the system has integer solutions. For example, suppose we are interested in knowing whether a dependence exists between statements S1 and S2 in Listing 3.1 on page 48 such that the value of x_2 when statement S1 executes is less than the value of x_2 when statement S2 executes (that is, the compiler seeks to determine if the second element of the direction vector $\vec{\psi}$ is '<'). To make this determination, the compiler adds the additional constraint $x_2 < x_{r+2}$ to (3.2) and tests for integer solutions to the system.

In this section, we compare how the I-Test and Omega Test compare when dealing with system of linear inequalities that incorporate one or more additional inequalities arising from direction vectors. It is important to note that the Omega Test does test the direction vector hierarchy in the same way the I-Test does (that is, by systematically and iteratively introducing a direction vector constraints and then testing the system). Rather, the Omega uses the variable protection technique I will discuss in section 6.2 to attempt to determine precise *dependence distances*. However, for purposes of the comparison in this chapter, I assume that a similar constraint has been added to the system to be solved by the Omega Test. Such a system may reasonably arise, for example, if a conditional constraint arising from an if statement is added to the problem presented to the Omega Test. This will allow me to draw an "apples to apples" comparison of the Direction Vector I-Test with the Omega Test.

5.1.1. I-Test Direction Vector Extension

As the Omega Test deals with arbitrary systems of linear inequalities, it requires no modifications to handle constraints comparable to those introduced by direction vectors. On the other hand, the (original) I-Test as described in the preceding sections cannot accommodate the additional constraint. However, extensions to the I-Test as described in [79] extend the I-Test to accommodate a system of linear inequalities augmented with direction vector constraints. The Direction Vector I-Test groups the variables to be eliminated into two sets: those that are coupled to other variables via a direction vector, and those that are not. As with the original I-Test the DV I-Test eliminates variables after first testing a condition that guarantees that the simplified problem resulting from the elimination is equivalent to the original problem. Variables that are not related by direction vectors are tested and eliminated in exactly the same way as in the original I-Test. However, when a pair of variables are related by a direction vector, the condition tested is more complex, and the DV I-Test eliminates *both* paired variables simultaneously, and adjusts the bounds to account for not only the paired variables' coefficients and loop bounds, but also the relationship between the variables expressed by the direction vector which relates the pair.

The τ Value

Whereas the original I-Test condition (shown in Eq. 4.9) is a simple comparison of the magnitude of a term's coefficient to the distance between the expression's lower and upper bounds, the DV I-Test compares a τ value to the distance between the bounds. If the term to be eliminated is a simple (uncoupled) term, then the τ value is simply the magnitude of the term's coefficient (as in the original I-Test). However, if a pair of coupled terms is to be eliminated, the corresponding τ value is a slightly more complex function of the pairs' coefficients. Specifically, let α_i be either an uncoupled term a_ix_i or the pair of coupled terms $a_ix_i + a_jx_j$, where x_i and x_j are related by the direction vector $x_i < x_j$. The DV I-Test defines the τ value as:

$$\tau_i = \begin{cases} |a_i| & \text{if } x_i \text{ is uncoupled,} \\ max(|a_i|, |a_j|) & \text{if } a_ia_j > 0 \\ max(min(|a_i|, |a_j|), |a_i + a_j|) & \text{if } a_ia_j < 0 \end{cases} \tag{5.2}$$

The DV I-Test's condition for moving α_i is simply:

$$\tau_i \leq c_U - c_L + 1 \tag{5.3}$$

where c_U and c_L are defined as in Eq. 4.9. It is obvious that the original I-Test can be seen as a degenerate case of the DV I-Test where all terms are uncoupled.

Adjusting the Bounds

If the above τ condition is satisfied, the term α_i is eliminated, and the bounds adjusted. How the bounds are adjusted depends on whether α_i represents a single uncoupled term $a_i x_i$ or a pair of coupled terms $a_i x_i + a_j x_j$.

In the case of a single uncoupled term $a_i x_i$, c_L and c_U are adjusted as in the original I-Test:

$$\begin{aligned} c_L &= c_L - {a_i}^+ U_i + {a_i}^- L_i \\ c_U &= c_U - {a_i}^+ L_i + {a_i}^- U_i \end{aligned} \tag{5.4}$$

In the case of a pair terms $a_i x_i + a_j x_j$ coupled with a direction vector $x_i < x_j$, the bounds are adjusted as follows (note that x_i and x_j share common loop bounds. Let the lower and upper bounds on the variables be represented as M and N, respectively, so that:

$$M \leq x_i < x_j \leq N$$

The DV I-Test eliminates the pair of terms and adjusts c_L and c_U as follows:

$$\begin{aligned} c_L &= c_L - ({a_i}^+ + a_j)^+(N - M - 1) - (a_i + a_j)M - a_j \\ c_U &= c_U + ({a_i}^- + a_j)^+(N - M - 1) - (a_i + a_j)M - a_j \end{aligned} \tag{5.5}$$

The essence of the above bounds adjustments is that in either case, the new upper bound c_U is the old upper bound minus the minimum value the uncoupled term (or pair of coupled terms) can achieve subject to constraints, and the new lower bound c_L is the old lower bound minus the maximum value the uncoupled term (or pair of coupled terms) can achieve subject to constraints. The DV I-Test operates by repeatedly moving either single uncoupled terms or pairs of coupled terms whose τ-values satisfy (5.3) until all terms are exhausted, or until no more terms can be found to satisfy (5.3). In the former case, if 0 is found to lie between the simplified RHS bounds expression resulting from the removal of all the terms, the DV I-Test reports that a dependency exists. Otherwise, or if no more terms can be found satisfying (5.3), the DV I-Test reports that a dependency *may* exist.

5.1.2. The Omega Test and Direction Vectors

Unlike the I-Test, the Omega Test requires no additional modifications to handle additional constraints such as those arising from direction vectors. It eliminates variables in the same way as discussed in the first part of the thesis, namely by combining all upper bounds on a variable with all lower bounds on the variable, and tightening the resulting inequality constraint via the Dark Shadow term. Let $a_i x_i$ and $a_j x_j$ be the two terms related by the direction vector $x_i < x_j$, and let

$$a_\lambda = \begin{cases} |a_i| & \text{if } a_i < a_j, \\ |a_j| & \text{otherwise} \end{cases} \qquad a_\mu = \begin{cases} |a_i| & \text{if } a_i > a_j, \\ |a_j| & \text{otherwise} \end{cases} \tag{5.6}$$

$$x_\lambda = \begin{cases} x_i & \text{if } a_i < a_j, \\ x_j & \text{otherwise} \end{cases} \qquad x_\mu = \begin{cases} x_i & \text{if } a_i > a_j, \\ x_j & \text{otherwise} \end{cases} \tag{5.7}$$

In contrast to the DV I-Test, the Omega Test does not eliminate the pair of coupled terms $a_i x_i + a_j x_j$ together, but independently. Neither does it necessarily eliminate $a_\mu x_\mu$ immediately after eliminating $a_\lambda x_\lambda$, but may instead eliminate other variables in between.

Inequalities in the Direction Vector Dependence Problem

Table 5.1 lists the inequalities comprising a dependence problem with a direction vector $x_i \leq x_j - 1$. The symbol ϕ represents the sum of all non-eliminated terms in the center of (3.5) with the exception of those joined by the direction vector ($a_i x_i$ and $a_j x_j$). That is:

$$\phi = \sum_{k \notin \{i,j\}} a_k x_k$$

Dark Shadows and Direction Vectors

When eliminating a term related by a direction vector to another term, the Dark Shadow term plays a role (that is, the DS term can potentially be non-zero) in three bound combinations. As we shall see, these combinations correspond to the clauses in the derivation of the τ condition shown in (5.2). As we will refer to them in subsequent discussions, we label and describe them as follows:

- DS_0 Occurs when bounds (6) and (7) are combined to eliminate the term $a_\lambda x_\lambda$. The Dark Shadows term added to the LHS is $(a_\lambda - 1)^2$, and the resulting inequality: $(a_\lambda - 1)^2 \leq c_U - c_L$ holds when $a_\lambda \leq c_U - c_L + 1$ by Lemma 4.3. The DS_0 inequality is a comparison of constants - that is, DS_0 is not a bound on any variable.

- DS_1 Similar to DS_0, but arises during the subsequent elimination of $a_\mu x_\mu$. The inequality tests whether the distance between the bounds after the elimination of $a_\lambda x_\lambda$ has spread them apart is at least as great as a_μ. As we shall see in the next section, the specific bound combinations giving rise to this inequality vary depending on the direction vector and the magnitude of the coefficients involved, but all have the form $(a_\mu - 1)^2 \leq a_\mu((c_U - c_L) + a_\lambda(N - M))$. Note that this assumes the elimination of $a_\mu x_\mu$ follows immediately after the elimination of $a_\lambda x_\lambda$. If other terms have been eliminated after the elimination of $a_\lambda x_\lambda$, then the bounds will have been spread even farther apart, making it less likely this combination will produce an inconsistency. Like DS_0, this is an inequality involving constant terms.

- DS_2 This combination occurs as part of the elimination of x_μ, and pairs an inequality whose x_μ coefficient is a_μ with an inequality whose x_μ coefficient is either $a_\mu - a_\lambda$ if $a_\mu a_\lambda < 0$ or $a_\mu + a_\lambda$ otherwise. Like DS_1, the specific combinations leading to this combination varies according to the magnitude of the coefficients of the variables involved in the direction vector. DS_2 is the only one of the three Dark Shadow combinations that produce an inequality in which ϕ is not eliminated. Unlike DS_0 and DS_1, the inequality resulting from the DS_2 combination will be either a lower or upper bound on ϕ, depending on the relative magnitudes and signs of a_i and a_j. As we shall see, if c_L and c_U are sufficiently far apart, the DS_2 inequality will be a weaker constraint on ϕ than that produced by other combinations, and in particular than that produced by the DV I-Test. However, if $c_U - c_L$ is small, then the DS_2 inequality can be a *stronger* constraint than that produced by the DV I-Test.

Cases of the Direction Vector Dependence Problem

The workings of the Omega Test is most easily understood by examining the eight possible variations of the direction vector problem with pairs of coupled terms of the form $a_i x_i + a_j x_j$ subject to $x_i < x_j$. Table 5.2 describes each of these cases, and shows the bounds produced by the Omega Test's elimination of the coupled terms from the system shown in 5.1, and illustrates how these resulting bounds compare with those calculated by the DV I-Test. Consistent with Table (5.1), the bounds resulting from the elimination are on

Label	Bound
1	$M \leq x_i$
2	$x_i \leq N$
3	$M \leq x_j$
4	$x_j \leq N$
5	$x_i \leq x_j - 1$
6	$c_L \leq a_i x_i + a_j x_j + \phi$
7	$a_i x_i + a_j x_j + \phi \leq c_U$

Table 5.1.: Bounds arising from direction vectors

ϕ, which represent the remaining terms after $a_i x_i$ and $a_j x_j$ have been eliminated. For each case, Table 5.2 lists several pieces of information:

- **Variation:** Describes how the values of a_i and a_j relate to zero and to one another for the particular case.
- **LB:** Describes the exact lower bound on ϕ calculated by the Ω Test after the expression $a_i x_i + a_j x_j : M \leq x, y \leq N, x_i < x_j$ is eliminated, as well as listing the sequence of bounds combinations from Table 5.1 which produces it. Note the expression is invariant: the new lower bound on ϕ is the old lower bound c_L minus the maximum value the expression $a_i x_i + a_j x_j$ can achieve subject to all constraints.
- **UB:** Similar to "LB". In all cases, the new upper bound on ϕ is the old upper bound c_U minus the minimum value the expression $a_i x_i + a_j x_j$ can achieve subject to all constraints.
- DS_2 **Bound:** The bound on ϕ produced by the DS_2 combination. Depending on the case, this is either a lower or upper bound on ϕ, and grows weaker relative to the corresponding exact upper or lower bound as $c_U - c_L$ increases. If $c_U - c_L$ is sufficiently small, this inequality may be stronger than the corresponding exact upper or lower bound, meaning that the Omega Test can potentially produce a contradiction and resort to an exhaustive search when the DV I-Test efficiently produces an exact answer. The Dark Shadow term added to the LHS of the inequality is the same in all cases: $(|a_i + a_j| - 1)(a_\mu - 1)$. This value is represented by DS_2 for brevity.
- **Accuracy Condition:** Specifies the conditions under which the DS_2 bound is weaker or equal to the corresponding upper or lower exact bound. It is derived by comparing the DS_2 bound with either the exact UB or LB. Since the DS_2 bound becomes stronger as $c_U - c_L$ diminishes, this condition is a relationship between the DS_2 term and $c_U - c_L$. If this condition is not met, the Omega Test produces an overly restrictive constraint that may eventually lead to a contradiction and exhaustive search for an integer solution.

Importantly, note that in each case of the direction vector problem, the Omega Test calculates the same upper and lower bound (that is, new values for c_U and c_L respectively) as is calculated by the DV I-Test via Equation (5.5). Thus, it is clear that the new bounds calculated by the Omega Test after eliminating a pair of coupled terms will be at least as strong as those calculated by the DV I-Test.

Example

To illustrate, we will step through the relevant bound combinations performed by the Omega Test to derive the DS_2 inequality for Case 5 in Table (5.2). In this case, we have $a_i > 0, a_j < 0, a_i < |a_j|$, $a_\lambda = |a_i|$, and $a_\mu = |a_j|$. Beginning with the elimination of x_i, the combination of the bounds from Table (5.1) include the following:

Case 1	Variation	$a_i > 0, a_j > 0, \lvert a_i\rvert < \lvert a_j\rvert$	
	LB	$c_L - (a_i(N-1) + a_jN) \leq \phi$	(5+6)+(4)
	UB	$\phi \leq c_U - a_iM + a_j(M+1)$	(1+5)+(1+7)
	DS_2 Bound	$\phi \leq c_U - (a_iM + a_j(M+1)) + \lvert\frac{a_j}{a_i}\rvert(c_U - c_L) - \frac{DS_2}{\lvert a_i\rvert}$	(5+6)+(1+7)
	Acc. Cond.	$\lvert a_j\rvert(c_U - c_L) \geq DS_2$	
Case 2	Variation	$a_i > 0, a_j > 0, \lvert a_i\rvert < \lvert a_j\rvert$	
	LB	$c_L - (a_i(N-1) + a_jN) \leq \phi$	(6+4)+(5+4)
	UB	$\phi \leq c_U - (a_iM + a_j(M+1))$	(5+4)+(1)
	DS_2 Bound	$c_L - (a_i(N-1) + a_jN) - \lvert\frac{a_i}{a_j}\rvert(c_U - c_L) + \frac{DS_2}{\lvert a_j\rvert}$	(6+4)+(5+7)
	Acc. Cond.	$\lvert a_i\rvert(c_U - c_L) \geq DS_2$	
Case 3	Variation	$a_i < 0, a_j < 0, \lvert a_i\rvert < \lvert a_j\rvert$	
	LB	$c_L - (a_iM + a_j(M+1) \leq \phi$	(1+5)+(1+6)
	UB	$\phi \leq c_U - (a_i(N-1) + a_j(N))$	(7+5)+(4)
	DS_2 Bound	$c_L - (a_iM + a_j(M+1) - \lvert\frac{a_j}{a_i}\rvert(c_U - c_L) + \frac{DS_2}{a_i} \leq \phi$	(7+5)+(1+6)
	Acc. Cond.	$\lvert a_j\rvert(c_U - c_L) \geq DS_2$	
Case 4	Variation	$a_i < 0, a_j < 0, \lvert a_i\rvert > \lvert a_j\rvert$	
	LB	$c_L - (a_iM + a_j(M+1) \leq \phi$	(1)+(5+6)
	UB	$\phi \leq c_U - (a_i(N-1) + a_j(N))$	(7+4)+(5+4)
	DS_2 Bound	$\phi \leq c_U - (a_i(N-1) + a_j(N)) + \lvert\frac{a_i}{a_j}\rvert(c_U - c_L) + \frac{DS_2}{\lvert a_j\rvert}$	(7+4)+(5+6)
	Acc. Cond.	$\lvert a_j\rvert(c_U - c_L) \geq DS_2$	
Case 5	Variation	$a_i > 0, a_j < 0, \lvert a_i\rvert < \lvert a_j\rvert$	
	LB	$c_L - (a_iM + a_j(M+1)) \leq \phi$	(1+5)+(6+5)
	UB	$\phi \leq c_U - (a_iM + a_jN)$	(1+7)+(4)
	DS_2 Bound	$c_U - (a_iM + a_j(M+1)) - \lvert\frac{a_j}{a_i}\rvert(c_U - cL) + \frac{DS_2}{a_i} \leq \phi$	(1+7)+(6+5)
	Acc. Cond.	$(\lvert a_j\rvert - a_i)(c_U - c_L) \geq DS_2$	
Case 6	Variation	$a_i > 0, a_j < 0, \lvert a_i\rvert > \lvert a_j\rvert$	
	LB	$c_L - (a_i(N-1) + a_jN) \leq \phi$	(6+5)+(4)
	UB	$\phi \leq c_U - (a_iM + a_jN)$	(1+7)+(4)
	DS_2 Bound	$c_U - (a_i(N-1) + a_jN) - \frac{a_i}{\lvert a_j\rvert}(c_U - c_L) + \frac{DS_2}{\lvert a_j\rvert} \leq \phi$	(6+5)+(7+4)
	Acc. Cond.	$(a_i - \lvert a_j\rvert)(c_U - c_L) \geq DS_2$	
Case 7	Variation	$a_i < 0, a_j > 0, \lvert a_i\rvert < \lvert a_j\rvert$	
	LB	$c_L - (a_iM + a_jN) \leq \phi$	(1+6)+(4)
	UB	$\phi \leq c_U - (a_iM + a_j(M+1))$	(1+5)+(7+5)
	DS_2 Bound	$\phi \leq c_L - (a_iM + a_j(M+1)) + \frac{a_j}{\lvert a_i\rvert}(c_U - c_L) - \frac{DS_2}{\lvert a_i\rvert}$	(1+6)+(7+5)
	Acc. Cond.	$(a_j - \lvert a_i\rvert)(c_U - c_L) \geq DS_2$	
Case 8	Variation	$a_i < 0, a_j > 0, \lvert a_i\rvert > \lvert a_j\rvert$	
	LB	$c_L - (a_iM + a_jN) \leq \phi$	(1+6)+(4)
	UB	$\phi \leq c_U - (a_i(N-1) + a_jN)$	(7+5)+(5+4)
	DS_2 Bound	$\phi \leq c_L - (a_i(N-1) + a_jN) + \frac{\lvert a_i\rvert}{a_j}(c_U - c_L) - \frac{DS_2}{a_j}$	(7+5)+(6+4)
	Acc. Cond.	$(\lvert a_i\rvert - a_j)(c_U - c_L) \geq DS_2$	

Table 5.2.: Bounds resulting from the elimination of coupled variables

$$\begin{array}{llll}
(1) & M \leq & x_i & \\
(5) & & x_i & \leq x_j - 1 \\
\hline
(1+5) & M+1 \leq & x_j &
\end{array}$$

$$\begin{array}{llll}
(1) & M \leq & x_j & \\
(7) & & a_i x_i & \leq c_U + |a_j| x_j - \phi \\
\hline
(1+7) & a_i M - c_U + \phi \leq & |a_j| x_j &
\end{array}$$

$$\begin{array}{llll}
(6) & c_L + |a_j| - \phi \leq & a_i x_i & \\
(5) & & x_i & \leq x_j - 1 \\
\hline
(6+5) & & (|a_j| - a_i) x_j & \leq -c_L - a_i + \phi
\end{array}$$

$$\begin{array}{llll}
(1+7) & a_i M - c_U + \phi \leq & |a_j| x_j & \\
(4) & & x_j & \leq N \\
\hline
(1+7)+(4) & & \phi & \leq c_U - [a_j N + a_i M]
\end{array}$$

$$\begin{array}{llll}
(1+5) & M+1 \leq & x_j & \\
(6+5) & & (|a_j| - a_i) x_j & \leq -c_L - a_i + \phi \\
\hline
(1+5)+(6+5) & c_L - [a_i M + a_j (M+1)] \leq & \phi &
\end{array}$$

$$\begin{array}{lllll}
(1+7) & a_i M - c_U + \phi \leq & |a_j| x_j & & \\
(6+5) & & (|a_j| - a_i) x_j & \leq -c_L - a_i + \phi & \\
\hline
(1+7)+(6+5) & (|a_j| - a_i)(a_i M - c_U + \phi) + DS_2 & \leq & & |a_j|(-c_L + \phi - a_i)
\end{array}$$

Note that the combination (1+7)+(4) is the upper bound calculated by the DV I-Test. To verify this, observe that for Case 5 the upper bound calculation in Equation 5.5 becomes:

$$\begin{aligned}
c_U &= c_U + (a_i^- + a_j)^+ (N - M - 1) - (a_i + a_j) M - a_j \\
&= c_U + (0 + |a_j|)^+ (N - M - 1) - (a_i + a_j) M + |a_j| \\
&= c_U + |a_j| N - a_i M \\
&= c_U - [a_j N + a_i M]
\end{aligned}$$

Likewise, the combination (1+5)+(6+5) is the lower bound calculated by the DV I-Test, and is shown as the exact LB in Table 5.2. To verify this, observe that for Case 5 the lower bound calculation in Equation 5.5 becomes:

$$\begin{aligned}
c_L &= c_L - (a_i^+ + a_j)^+ (N - M - 1) - (a_i + a_j) M - a_j \\
&= c_L - (a_i - |a_j|)^+ (N - M - 1) - (a_i + a_j) M + |a_j| \\
&= c_L - (0)(N - M - 1) - (a_i - |a_j|) M + |a_j| \\
&= c_L - [a_i M + a_j (M+1)]
\end{aligned}$$

The DS_2 inequality for this case arises from the combination (1+7)+(6+5), and the corresponding DS_2 term is $(|a_j| - a_i - 1)(|a_j| - 1)$. We can derive an explicit bound on ϕ from the combination as follows:

$$\begin{array}{lrcl}
\Rightarrow & |a_j|a_iM - |a_j|c_U + |a_j|\phi - a_i{}^2M + a_ic_U - a_i\phi + DS_2 & \leq & -|a_j|c_L + |a_j|\phi - |a_j|a_i \\
\Rightarrow & a_ic_U - a_i{}^2M + |a_j|a_iM + |a_j|a_i - |a_j|c_U + |a_j|c_L + DS_2 & \leq & a_i\phi \\
\Rightarrow & a_i[c_U - a_iM + |a_j|M + |a_j|] - |a_j|(c_U - c_L) + DS_2 & \leq & a_i\phi \\
\Rightarrow & c_U - a_iM + |a_j|M + |a_j| - \frac{|a_j|}{a_i}(c_U - c_L) + \frac{DS_2}{a_i} & \leq & \phi \\
\Rightarrow & c_U - a_iM + |a_j|(M+1) - \frac{|a_j|}{a_i}(c_U - c_L) + \frac{DS_2}{a_i} & \leq & \phi \\
\Rightarrow & c_U - [a_iM - |a_j|(M+1)] - \frac{|a_j|}{a_i}(c_U - c_L) + \frac{DS_2}{a_i} & \leq & \phi \\
\Rightarrow & c_U - [a_iM + a_j(M+1)] - \frac{|a_j|}{a_i}(c_U - c_L) + \frac{DS_2}{a_i} & \leq & \phi
\end{array}$$

The final inequality above is recorded as the DS_2 bound for Case 5 in Table 5.2. Like the combination (1+7)+(6+5), it is a lower bound on ϕ. Subtracting the LHS of the above version of (1+7)+(6+5) from the LHS of (1+5)+(6+5) tells us how much stronger a lower bound (1+5)+(6+5) is compared to (1+7)+(6+5):

$$\begin{array}{rl}
& c_L - [a_iM + a_j(M+1)] - [c_U - [a_iM + a_j(M+1)] - \frac{|a_j|}{a_i}(c_U - c_L) + \frac{DS_2}{a_i}] \\
= & \frac{|a_j|}{a_i}(c_U - c_L) + c_L - c_U - \frac{DS_2}{a_i} \\
= & \frac{|a_j|}{a_i}(c_U - c_L) - (c_U - c_L) - \frac{DS_2}{a_i} \\
= & \frac{|a_j| - a_i}{a_i}(c_U - c_L) - \frac{DS_2}{a_i}
\end{array}$$

This latter inequality shows that as $c_U - c_L$ increases, the DS_2 bound (1+7)+(6+5) tends to be weaker than the exact bound ((1+5)+(6+5), and so is inconsequential. That is, when

$$\begin{array}{lrcl}
& \frac{|a_j| - a_i}{a_i}(c_U - c_L) - \frac{DS_2}{a_i} & \geq & 0 \\
\Rightarrow & \frac{|a_j| - a_i}{a_i}(c_U - c_L) & \geq & \frac{DS_2}{a_i} \\
\Rightarrow & (|a_j| - a_i)(c_U - c_L) & \geq & DS_2 \\
\Rightarrow & (|a_j| - a_i)(c_U - c_L) & \geq & (|a_j| - a_i - 1)(|a_j| - 1)
\end{array}$$

the Omega Test produces the same constraints as the DV I-Test. This last inequality is reflected in Table 5.2 as the accuracy condition for Case 5. When it does not hold, the Omega Test places an overly restrictive bound into the system, which potentially could later cause the Omega Test to perform an exhaustive search for integer solutions.

Comparison of the Omega Test and DV I-Test

Before attempting to eliminate a pair of coupled terms, the DV I-Test checks whether the τ condition specified in Equation (5.3), which guarantees the existence of integer solutions, is satisfied. If it is, the terms are eliminated and the bounds of the expression (c_U and c_L) are adjusted according to Equation (5.5) to reflect the exact minimum and maximum values the terms can attain subject to constraints. If the condition is not satisfied, the DV I-Test conservatively reports that an integer solution *may* exist. In contrast, the Omega Test always attempts to eliminate the pair of variables, and adjusts the bounds sufficiently such that, if satisfied, an integer solution is guaranteed to exist. In a sense, the Omega Test's integrality test is encoded into the resulting bounds instead of being explicitly applied prior to the elimination of the terms.

In the first part of this thesis, I showed that in the absence of direction vectors, the two tests are essentially equivalent. However, when direction vectors are introduced into the problem, either test can out-perform the other under a specific set of circumstances. For instance, the DV I-Test τ condition may fail to hold, causing the DV I-Test to conservatively return a "maybe" response, whereas the Omega Test can successfully eliminate both terms and efficiently return an exact answer. As an example, consider the following simple direction vector problem:

$$\begin{aligned} 15 \le 5x + 7y + 8z \le 20 \\ x < y \\ 1 \le x, y \le 10 \\ 1 \le z \le 10 \end{aligned}$$

the τ inequality is not satisfied because $max(5,7) > 20 - 15 + 1$, and thus the DV I-Test conservatively reports that a dependency may exist. On the other hand, the Omega Test checks the *smallest* coefficient against the distance between the bounds, and since $5 \le 20 - 15 + 1$, it is able to eliminate the x term (i.e. DS_0 is satisfied). This spreads the bounds far enough apart to subsequently move the y (i.e. DS_1 is satisfied) and z terms in succession and ultimately return an exact answer in polynomial time. The key distinction is that the Omega Test eliminates each term independently while still retaining the relationship imposed by the direction vector.

On the other hand, the DV I-Test is a customized test that (provided the τ inequality holds) calculates *exact* bounds for the specific case of the dependence problem involving constant loop bounds and one or more direction vectors. The Omega Test is a more general test that can in some cases (via the DS_2 inequality) produce more conservative bounds than necessary for the same instance of the problem. These overly-conservative bounds may result in the Omega Test encountering an inconsistency and beginning an exhaustive search for integer solutions.

We first observe that if *any* of the three clauses comprising the τ inequality are satisfied, then neither DS_0 nor DS_1 will produce a contradiction. The DS_0 inequality ($(a_\lambda - 1)^2 \le c_U - c_L$) holds when $a_\lambda \le c_U - c_L + 1$ by Lemma 4.3. If $a_i a_j > 0$ and the τ condition is satisfied, then $a_\lambda + a_\mu \le c_U - c_L + 1$, and this implies $a_\lambda \le c_U - c_L + 1$. On the other hand, if $a_i a_j < 0$ and the τ condition is satisfied, then $max(a_\lambda, a_\mu - a_\lambda) \le c_U - c_L + 1$, and DS_0 will again be satisfied.

The DS_2 inequality ($(a_\mu - 1)^2 \le a_\mu((c_U - c_L) + a_\lambda(N - M))$) is likewise satisfied when any of the τ clauses are satisfied. If $a_i a_j > 0$ and the τ condition is satisfied, then $a_\mu + a_\lambda \le c_U - c_L + 1$, and certainly $a_\mu - 1 \le c_U - c_L$. Since $N - M \ge 1$, we can then rewrite DS_2 as follows:

$$\begin{aligned} (a_\mu - 1)^2 &\le a_\mu((c_U - c_L) + a_\lambda(N - M)) \\ (a_\mu - 1)^2 &\le a_\mu((a_\mu - 1) + a_\lambda(N - M)) \\ (a_\mu - 1)^2 &\le a_\mu((a_\mu - 1) + a_\lambda) \\ a_\mu^2 - 2a_\mu + 1 &\le a_\mu^2 - a_\mu + a_\mu a_\lambda \\ 1 &\le a_\mu + a_\mu a_\lambda \\ 1 &\le a_\mu(1 + a_\lambda) \end{aligned}$$

which always holds. On the other hand, if $a_i a_j < 0$ and the τ condition holds, then $a_\mu - a_\lambda - 1 \le c_U - c_L$ since $MAX(a_\lambda, a_\mu - a_\lambda) \le c_U - c_L + 1$. Since $N - M \ge 1$, we can then rewrite DS_2 as follows:

$$\begin{aligned}
(a_\mu - 1)^2 &\leq a_\mu((a_\mu - a_\lambda - 1) + a_\lambda(N - M)) \\
(a_\mu - 1)^2 &\leq a_\mu((a_\mu - a_\lambda - 1) + a_\lambda) \\
(a_\mu - 1)^2 &\leq a_\mu(a_\mu - 1) \\
a_\mu^2 - 2a_\mu + 1 &\leq a_\mu^2 - a_\mu \\
1 &\leq a_\mu
\end{aligned}$$

which again always holds. Thus, if the τ condition holds, the Omega Test will not immediately detect a contradiction via the DS_0 and DS_1 inequalities. However, under certain circumstances the DS_2 may produce a bound on the remaining variables that are tighter than the exact bounds produced by the DV I-Test.

Recall that unlike DS_0 and DS_1, which are inequalities among constants that can immediately be validated, the DS_2 inequality produce a bound on the remaining variables. This bound is weaker than the corresponding (exact) bound produced by the DV I-Test When the accuracy condition listed in Table (5.2) is satisfied. However, when the accuracy condition is not met, the DS_2 inequality is tighter than necessary and may eventually produce a contradiction.

To summarize the accuracy conditions in Table (5.2), if $a_i a_j > 0$, the Omega Test produces overly restrictive bounds when:

$$\frac{a_\mu}{|a_i + a_j| - 1} < \frac{a_\mu - 1}{c_U - c_L} \tag{5.8}$$

If $a_i a_j < 0$, the Omega Test will produce tighter bounds than is necessary when:

$$\frac{|a_i + a_j|}{|a_i + a_j| - 1} < \frac{a_\mu - 1}{c_U - c_L} \tag{5.9}$$

As an example, consider the simple (and contrived) system

$$\begin{aligned}
21 \leq 5x + 7y + 8z \leq 27 \\
x < y \\
1 \leq x, y \leq 4 \\
1 \leq z \leq 10
\end{aligned}$$

This system fails the accuracy condition for Case 1 from Table (5.2) since $(7)(6) \ngeq (7 + 5 - 1)(7 - 1)$. The relevant bounds combinations are as follows:

$$\begin{array}{llll}
(6) & 21 - 7y - 8z \leq & 5x & \\
(5) & & x & \leq y - 1 \\
\hline
(6{+}5) & 13 - 4z \leq & 6y & \text{(After normalization)}
\end{array}$$

$$\begin{array}{llll}
(1) & 1 \leq & x & \\
(5) & & x & \leq y - 1 \\
\hline
(1{+}5) & 2 \leq & y &
\end{array}$$

$$\begin{array}{lllll}
(1) & 1 \leq & x & & \\
(7) & & 5x & \leq 27 - 7y - 8z & \\
\hline
(1+7) & 7y \leq & 22 - 8z & &
\end{array}$$

$$\begin{array}{llll}
(5+6) & 13 - 4z \leq & 6y & \\
(4) & & y & \leq 4 \\
\hline
(1+5) & -11 \leq & 4z &
\end{array}$$

$$\begin{array}{lllll}
(1+5) & 2 \leq & y & & \\
(1+7) & & 7y & \leq 22 - 8z & \\
\hline
(1+5) & z \leq & 1 & & \text{(After normalization)}
\end{array}$$

$$\begin{array}{lrlll}
(6+5) & 13 - 4z \leq & 6y & & \\
(1+7) & & 7y & \leq 22 - 8z & \\
\hline
 & (91 - 28z) + (6-1)(7-1) \leq & 132 - 48z & & \\
 & 20z \leq & 11 & & \\
(1+5) & z \leq & 0 & & \text{(After normalization)}
\end{array}$$

Note that the DS_2 combination (6+5)+(1+7) is a stronger constraint than (1+7)+(1+5). When the DS_2 ($z \leq 0$) inequality is subsequently combined with $1 \leq z$, the Omega Test detects a contradiction. On the other hand, the DV I-Test is able to move the pair of coupled terms $5x + 7y$ since $max(5,7) \leq (27 - 21 + 1)$, which then spread the bounds far enough apart to allow the elimination of the final term.

As a final note, the conditions under which one test is more accurate than the other do not arise in practice. The bounds on the loop variables would have to be relatively small compared to the loop variable coefficients. That being the case, even if they did occur and an exhaustive search needed to be performed, the small loop iteration space would make the search relatively inexpensive.

5.2. Summary

In this chapter I consider how the extension to the I-Test to handle direction vectors compares to the Omega Test when each are given an equivalent problem to solve. I showed that the Direction-Vector I-Test is a specialized test to handle a particular instance of the dependence problem which arises when a simple inequality of the form $x_i < x_j$ is added to the dependence problem of the form described in chapter 4. I also derived and described the conditions under which each test could at least theoretically perform better than the other, and noted that these conditions are highly unlikely to arise in practice, meaning that the equivalence of the tests for all practical purposes extends to the case when direction vectors are added to the problem.

Chapter 6

Further Discussion

This chapter contains discussions of a number of aspects of the Omega Test that have not been discussed earlier because they do not directly concern the comparison of the Omega Test to the I-Test. However, several facets of the test are interesting in their own right and ought to be examined.

6.1. Dealing with non-Unit Coefficients in Equality Constraints

As noted at several points in the preceding presentation, both the I-Test and and Omega Test require that the inequalities arising from the equality constraint have at least one term with a coefficient with an absolute value of 1. If this condition is not met, then the I-Test condition and the Dark Shadow inequality will fail when comparing the two inequalities arising from this constraint, owing to the fact the distance between the limits are initially zero.

However, if the equality constraint has no such suitable term, the Omega Test performs a series of substitutions that eventually create a system with a coefficient with an absolute value of 1 that is equivalent to the original system (meaning it has an integer solution if and only if the original system has an integer solution). In this section, I describe the mechanics of this substitution and determine the rate at which coefficients are reduced.

6.1.1. The Coefficient Reduction Strategy

Suppose the equality constraint in the system is:

$$\sum_{1 \le i \le n} a_i x_i = c \tag{6.1}$$

Without loss of generality, the Omega Test creates the equivalent equation

$$\sum_{i \in V} a_i x_i = 0 \tag{6.2}$$

where V is the set of indices from equation 6.1 plus 0 (i.e. $V = 0..n$, $x_0 = 1$) and $a_0 = c$. Assume no a_i for $1 <= i <= n$ is ± 1. The Omega Test will perform a series of substitutions designed to systematically reduce the absolute value of the coefficients until a system with at least one term with a coefficient of ± 1.

Assume that a_k is the coefficient with the smallest absolute value in equation 6.1. The Omega Test defines the following:

$$\begin{aligned}
\widehat{modf}(a,m) &= \left\lfloor \frac{a}{m} + \frac{1}{2} \right\rfloor \\
\widehat{modh}(a,m) &= a - m\,\widehat{modf}(a,m) \\
m &= |a_k| + 1 \\
sign(x) &= \begin{cases} -1 & : \quad x < 0 \\ 0 & : \quad x = 0 \\ +1 & : \quad x > 0 \end{cases}
\end{aligned}$$

Intuitively, $\widehat{modh}(a,m)$ returns the difference between a and the closest multiple of m to a, and $\widehat{modf}(a,m)$ returns the coefficient of m required to produce that closest multiple. Therefore, the following relationship holds:

$$x = m\,\widehat{modf}(x,m) - \widehat{modh}(x,m) \tag{6.3}$$

The Omega Test then creates a new variable σ, as well as the following equality:

$$m\sigma = \sum_{i \in V} (\widehat{modh}(a_i,m))x_i \tag{6.4}$$

Note that by definition of the $\widehat{modh}$ function and m, $\widehat{modh}(a_k,m) = -sign(a_k)$:

$$m\sigma = -sign(a_k)x_k + \sum_{i \in V-\{k\}} (\widehat{modh}(a_i,m))x_i \tag{6.5}$$

Solving equation 6.5 for x_k produces:

$$x_k = -sign(a_k)m\sigma + \sum_{i \in V-\{k\}} (sign(a_k)\ \widehat{modh}(a_i,m))x_i \tag{6.6}$$

We now can substitute the RHS of equation 6.6 for x_k in equation 6.1 as well as all occurrences of x in the inequalities arising from loop constraints and direction vectors. In the case of the substitution into equation 6.1, the result is:

$$-|a_k|m\sigma + \sum_{i \in V-\{k\}} (a_i + (|a_k|\ \widehat{modh}(a_i,m)))x_i = 0 \tag{6.7}$$

Since $|a_k| = m - 1$ by definition, we have:

$$-|a_k|m\sigma + \sum_{i \in V-\{k\}} ((a_i - \widehat{modh}(a_i,m)) + \widehat{modh}(a_i,m))x_i = 0 \tag{6.8}$$

As discussed earlier, $\widehat{modh}(a,m)$ produces the smallest difference between a and a multiple of m. Using this fact and equation 6.3, the coefficients of all terms are now divisible by m. We simplify equation 6.8 and produce:

$$-|a_k|\sigma + \sum_{i \in V-\{k\}} (\widehat{modf}(a_i,m) + \widehat{modh}(a_i,m))x_i = 0 \tag{6.9}$$

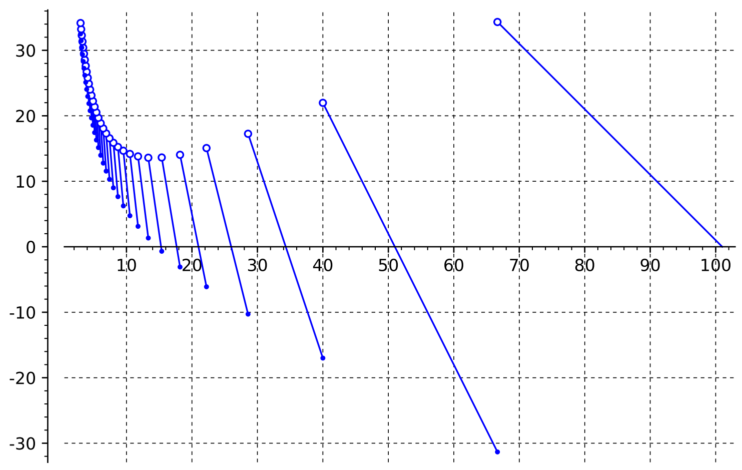

Figure 6.1.: Graph of $\phi(a,m)$ for $3 \leq m \leq a+1$, $(a = 100)$

All of the coefficients in equation 6.9 will be less than the corresponding coefficients in equation 6.1 (I will prove this in the next section when I describe how quickly the coefficients reduce). Therefore, repeated applications of this technique will eventually produce a coefficient of ± 1, allowing the Omega Test to begin to eliminate variables and apply the Dark Shadows inequality as described in the earlier sections of this thesis.

6.1.2. The Coefficient Rate of Reduction

I now investigate the rate at which coefficients are reduced by the methodology described above. We define a new function $\phi(a,m)$ based on 6.9 that will describe the value of the new coefficient for a variable with an original coefficient a and an m value described in the preceding section.

$$\phi(a,m) = \widehat{modf}(a,m) + \widehat{modh}(a,m) \tag{6.10}$$

Thus, $\phi(a_i,m)$ will equal the new coefficient for x_i after the previously described substitution algorithm produces a new equality constraint. Using this new function definition, we can express equation 6.9 as follows:

$$-|a_k|\sigma + \sum_{i \in V - \{k\}} \phi(a_i,m)x_i = 0 \tag{6.11}$$

Note that $3 \leq m \leq |a|+1$ since $m = |a_k|+1$ and $2 \leq |a_k|$ by definition, and $|a_k| \leq |a_i|$ for all $i \in V - \{k\}$ by virtue of the fact that x_k was chosen to be the variable whose coefficient has the smallest absolute value in the original equality constraint. A graph of $\phi(a,m)$ for $a = 100$ where $3 \leq m \leq a+1$ is shown in Figure 6.1. The discontinuities and open circles in the graph result from the floor function in the definition of $\widehat{modf}(a,m)$ causing the function to *instantaneously* transition when $\frac{a}{m}$ reaches an *inflection point* (defined below).

If we disregard the floor functions contained in the $\widehat{modf}$ and $\widehat{modh}$ components of the ϕ function defined in equation 6.10, we can conservatively approximate the upper bound on *phi* with g and the lower bound on ϕ with f defined as follows:

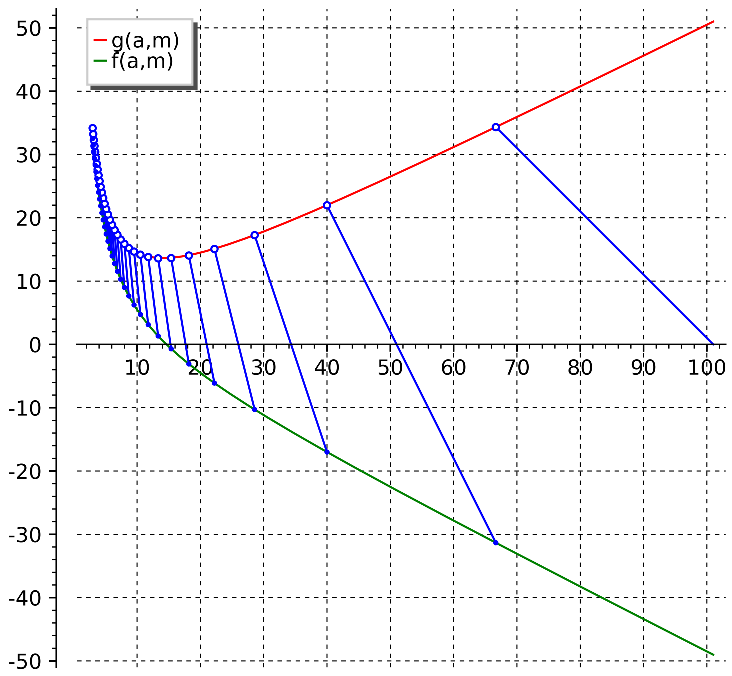

Figure 6.2.: Graph of $\phi(a,m)$ for $3 \leq m \leq a+1$, $(a = 100)$ with bounding functions

$$g(a,m) = a - \frac{m^2 - 2a - m}{2m}$$

$$f(a,m) = a - \frac{m^2 + 2a - m}{2m}$$

To illustrate, Figure 6.2 shows the same *phi*-function along with the bounding functions $f()$ and $g()$.

To describe how effective the Omega Test's substitution algorithm is at reducing coefficients, we must derive the worst case relationship between a_i and $|\phi(a_i,m)|$ (worst case means determining the maximum value $|\phi(a_i,m)|$ can obtain, which is the worst case coefficient reduction).

Let us examine the case where $a > 0$ (the argument for negative a is symmetrical). Since $\phi(a,m)$ has $g(a,m)$ as an upper limit, we begin by fixing a and letting m span its range of possible values. Taking the partial derivatives of the bounding functions $g(a,m)$ with respect to m yields:

$$y = g(a,m)$$

$$z = f(a,m)$$

$$\frac{\partial y}{\partial m} = -\frac{a}{m^2} + \frac{1}{2}$$

$$\frac{\partial z}{\partial m} = -\frac{a}{m^2} - \frac{1}{2}$$

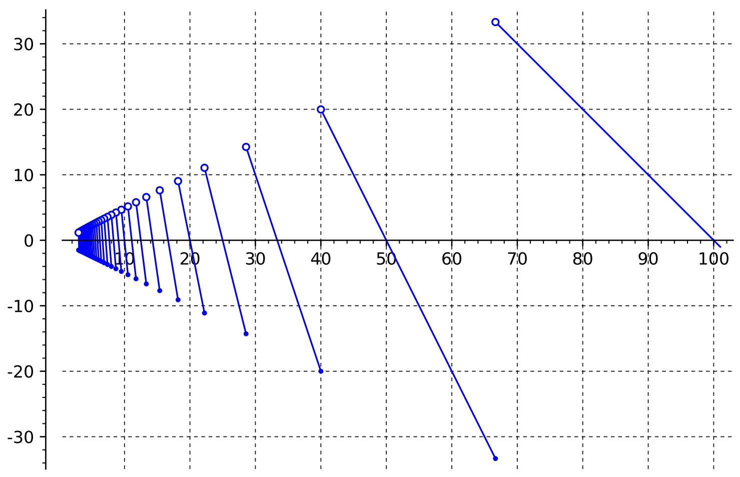

Figure 6.3.: Graph of $modh(a,m)$ for $3 \leq m \leq a+1$, $(a = 100)$

Setting the partial derivatives to zero and solving for m yields:

$$\begin{aligned}\frac{\partial y}{\partial m} &= 0 \Rightarrow \left[m = -\sqrt{2}\sqrt{a}, m = \sqrt{2}\sqrt{a}\right] \\ \frac{\partial z}{\partial m} &= 0 \Rightarrow \left[m = -\sqrt{2}\sqrt{-a}, m = \sqrt{2}\sqrt{-a}\right]\end{aligned}$$

From this we determine that f has no real critical points, whereas g has a critical point at $m = \sqrt{2a}$. Referring once again to Figure 6.2, we see that this is a lower limit, and that we must examine the function at the endpoints of the range to determine upper limits.

To do so, let us examine the $\widehat{modh}$ function's contribution to ϕ. The graph of $\widehat{modh}$ alone is depicted in Figure 6.3. Recall that $\widehat{modh}(a,m)$ returns the smallest distance from a to an integer multiple of m.

Let c be integer and let $c = \frac{a}{m}$. Obviously, when a is an integer multiple of m already, $\widehat{modh}$ returns 0. For example, note how the graph of $\widehat{modh}$ in Figure 6.3 crosses the y axis at $m = 20, 25, 33\frac{1}{3}, 50, 100$, corresponding to $c = 5,4,3,2,1$. I will refer to m-values at such points as *zero-points*. Also note that $\widehat{modh}$ reaches local maximum and minimum values when m is exactly halfway between these integer values of c, i.e. $c = \frac{11}{2}, \frac{9}{2}, \frac{7}{2}, \ldots$. Such points at which $\widehat{modh}$ reaches a local extreme will be referred to as *inflection points*. Let m' be some zero-point corresponding to an integer value of c, say c'. Then the closest inflection point for $m_p < m'$ corresponds to a c value of $\frac{2c'+1}{2}$.

The corresponding value of m_p then becomes

$$\begin{aligned}m_p &= \frac{a}{\frac{2c'+1}{2}} \\ &= \frac{2a}{2c'+1}\end{aligned}$$

and $\widehat{modf}(a,m_p)$ is calculated as follows:

$$\begin{aligned}
\widehat{modf}(a,m_p) &= \left\lfloor \frac{a}{\frac{2a}{2c'+1}} + \frac{1}{2} \right\rfloor \\
&= \left\lfloor \frac{a(2c'+1)}{2a} + \frac{1}{2} \right\rfloor \\
&= \left\lfloor c' + \frac{1}{2} + \frac{1}{2} \right\rfloor \\
&= c'+1
\end{aligned}$$

substituting this result along with the value of m_p into the formula for $\widehat{modh}$, we obtain:

$$\begin{aligned}
\widehat{modh}(a,m_p) &= a - \frac{2a}{2c'+1}(c'+1) \\
&= -\frac{a}{2c'+1}
\end{aligned}$$

This negative value requires special attention. Substituting this value for $\widehat{modh}(a,m_p)$ and $c'+1$ for $\widehat{modf}(a,m_p)$ would lead to $\phi(a,m_p) = c'+1-\frac{a}{2c'+1}$. However, this leads to a coefficient with an absolute value that is not conservatively large. To see why, note that as m approaches $m_p = \frac{2a}{2c'+1}$ from above, $\widehat{modh}(a,m)$ asymptotically approaches $+\frac{a}{2c'+1}$. Furthermore, as long as $m > \frac{2a}{2c'+1}$, $\widehat{modf}(a,m) = c'$ and $\phi(a,m)$ approaches $\frac{a}{2c'+1}+c'$. Stated formally,

$$\begin{aligned}
\lim_{m\to\frac{2a}{2c'+1}^+} \widehat{modh}(a,m) &= \frac{a}{2c'+1} \\
\lim_{m\to\frac{2a}{2c'+1}^+} \widehat{modf}(a,m) &= c' \\
\lim_{m\to\frac{2a}{2c'+1}^+} \phi(a,m) &= \frac{a}{2c'+1}+c'
\end{aligned}$$

This latter value for $\phi(a,m)$ is conservatively large for $a>0$, and so we define

$$\phi(a,m_p) = \frac{a}{2c'+1}+c' \tag{6.12}$$

We calculate m_n along with the associated values for $\widehat{modh}(a,m_n)$, $\widehat{modf}(a,m_n)$ and $\phi(a,m_n)$ similarly. The closest inflection point for $m_n > m'$ corresponds to a c value of $\frac{2c'-1}{2}$.

The corresponding value of m_n then becomes

$$\begin{aligned} m_n &= \frac{a}{\frac{2c'-1}{2}} \\ &= \frac{2a}{2c'-1} \end{aligned}$$

and $\widehat{modf}(a, m_n)$ is calculated as follows:

$$\begin{aligned} \widehat{modf}(a, m_n) &= \left\lfloor \frac{a}{\frac{2a}{2c'-1}} + \frac{1}{2} \right\rfloor \\ &= \left\lfloor \frac{a(2c'-1)}{2a} + \frac{1}{2} \right\rfloor \\ &= \left\lfloor c' - \frac{1}{2} + \frac{1}{2} \right\rfloor \\ &= c' \end{aligned}$$

substituting this result along with the value of m_n into the formula for $\widehat{modh}$, we obtain:

$$\begin{aligned} \widehat{modh}(a, m_n) &= a - \frac{2a}{2c'-1}(c') \\ &= -\frac{a}{2c'-1} \end{aligned}$$

As before, we conservatively use $\frac{a}{2c'-1}$ as the value for $\widehat{modh}(a, m_n)$. Substituting this value of $\widehat{modh}(a, m_n)$ and $\widehat{modf}(a, m_n)$ into the formula for $\phi(a, m)$, we have

$$\phi(a, m_n) = \frac{a}{2c'-1} + c' \tag{6.13}$$

We are now ready to analyze the behaviour of the ϕ function at its endpoints. To describe the behaviour of ϕ when m assumes its smallest value of 3, we consider three cases, corresponding to the three possible values of a mod m.

- When $(a \bmod 3) = 0$, m evenly divides m, so $\widehat{modh}(a, 3) = 0$. On the other hand, $\widehat{modf}(a, 3)$ returns $\frac{a}{3}$, and so we add these components to see that $\phi(a, 3) = \frac{a}{3}$.
- When $(a \bmod 3) = 1$, $\widehat{modh}(a, 3) = 1$, since a value of 1 must be subtracted from a to produce an integer multiple of m. Also, $\widehat{modf}(a, 3)$ returns $\lfloor \frac{a}{3} \rfloor$, and so $\phi(a, 3) = \lfloor \frac{a}{3} \rfloor + 1$.
- When $(a \bmod 3) = 2$, $\widehat{modh}(a, 3) = -1$, since 1 must be *added* to a to reach an integer multiple of m. Meanwhile, $\widehat{modf}(a, 3) = \lceil \frac{a}{m} \rceil$, and so $\phi(a, 3) = \lceil \frac{a}{3} \rceil - 1$.

Note that in all of the above cases, $\phi(a, 3) \leq \frac{a}{3} + 1$.

At the other extreme, as m approaches its maximum value with respect to a we note that at $m = a$, $\widehat{modh}(a,a) = 0$ and $\widehat{modf}(a,a) = 1$, so $\phi(a,a) = 1$. At m's extreme value of $a+1$, $\widehat{modh}(a,a+1) = -1$ and $\widehat{modf}(a,a+1) = 1$, meaning $\phi(a,a+1) = 0$.

Recall that $\widehat{modh}(a,m)$ reaches a local maximum value at the inflection point m_p and then decreases to 0 as m approaches a zero-point from the left, and then increases as m approaches the next inflection point at m_n. The greatest zero-point in m's domain occurs at $m = a$ (corresponding to $c = 1$). Accordingly, we set $c' = 1$ and $m' = a$, and find the bracketing inflection points. We observe that m_n is $\frac{2a}{2(1)-1} = 2a$, which is not in the range of m for $a > 2$ (when $a = 2$, $m = 3$, and $\phi(2,3) = 0$). However, m_p occurs at $\frac{2a}{2c'+1} = \frac{2a}{3}$, and according to Equation 6.12, we have

$$\begin{aligned}\phi(a,m_p) &= \frac{a}{2c'+1} + c' \\ &= \frac{a}{2(1)+1} + 1 \\ &= \frac{a}{3} + 1\end{aligned}$$

We thus conclude that the reduced coefficient for x_i after the Omega Test substitution algorithm is used is at most $\frac{a_i}{3} + 1$, where a_i is the original value of the coefficient. This is actually a stronger result than has been previously published. For example, the original Omega Test paper [81] states

"In the original constraint, the absolute value of the coefficient of σ is the same as the absolute value of the original coefficient of x_k. For all other variables, the absolute value of coefficients are reduced to at most 2/3rds of their previous value."

6.1.3. Impact of Substitutions on Other Inequalities

In this section I wish to examine an aspect of the substitution algorithm that to the best of my knowledge has heretofore not been explored in the literature.

The Omega Test employs the substitution algorithm described in the preceding subsection in an attempt to force at least one variable in the equality constraint to have a coefficient of ± 1 where no such coefficient originally existed. To reiterate, if no inequality in the system met this criteria, then the Omega Test's Dark Shadow would immediately fail when it attempted to compare the upper and lower bound inequalities arising from the equality constraint.

In general, successful application of the Dark Shadow inequality relies on variable coefficients being relatively small in relation to the loop bounds. While on one hand the substitution algorithm reduces the coefficients in the equality constraint, it can have the undesirable effect of increasing coefficients in the inequalities in the system (i.e. those arising from bounds on the loop iteration variables as well as direction vectors).

To see this, recall that the formula for x_k as shown in equation 6.6 is substituted for x_k not only in the original equality constraint, but also in all other constraints where x_k appears. One of these will be the inequality arising from the loop iteration bounds on x_k.

Assume that after the equality constraint is rewritten in the form of equation 6.6, the system arising from the new equality along with the existing constraints arising from bounds on the loop iteration variables are as follows:

$$\begin{aligned} \sum_{i \in V} a_i x_i &= 0 \\ L_j \leq x_j &\leq U_j \qquad (1 <= j <= n) \end{aligned}$$

where U_j and L_j are the upper and lower bounds, respectively, on the loop iteration variables. Assuming that no coefficient in the equality constraint is ± 1, we add a new variable σ_k (I have added subscripts on the synthetic variable σ_k to denote this variable arises from the substitution for variable x_k; likewise, m_k refers to $|a_k| + 1$) and employ the $\widehat{modh}$-based algorithm. This first part derives an equation for x_k in terms of the remaining variables and the σ_k variable:

$$x_k = -sign(a_k) m_k \sigma_k + \sum_{i \in V - \{k\}} (sign(a_k)\ \widehat{modh}(a_i, m_k)) x_i \tag{6.14}$$

After substitution (and using the ϕ function defined in the previous section), the system now appears as follows:

$$\begin{aligned} -|a_k|\sigma + \sum_{i \in V - \{k\}} \phi(a_i, m_k) x_i &= 0 && (6.15) \\ L_k \leq -sign(a_k) m_k \sigma_k + \sum_{i \in V - \{k\}} (sign(a_k)\ \widehat{modh}(a_i, m_k)) x_i &\leq U_k && (6.16) \\ L_j \leq x_j &\leq U_j \qquad (j \in V - \{k\}) && (6.17) \end{aligned}$$

Assume again that no coefficient in the equality constraint is ± 1. Further assume that the variable in the equality constraint whose coefficient has the smallest absolute value is x_q (it could also have been the σ synthetic variable)

When the process is repeated, the equation for the new substitution becomes

$$x_q = -sign(a_q) m_q \sigma_q + \widehat{modh}(-|a_k|, m_q) \sigma_k + \sum_{i \in V - \{k,q\}} \widehat{modh}(\phi(a_i, m_k), m_q) x_i \tag{6.18}$$

This expression has to be substituted for all occurrences of x_q. Note that x_q appears in three places in the system represented by 6.15-6.17:

- In the equality constraint (6.15), with the coefficient $\phi(a_q, m_k)$
- In the interior of the first inequality (6.16), which were the bounds on x_k before being substituted with the expression for x_k in the first step. In this inequality, x_q has the coefficient $(sign(a_k)\ \widehat{modh}(a_q, m_k))$
- In the inequality representing the bounds on x_q (6.17)

In the expression for x_k (6.18), the absolute value of the coefficient for σ_k could be as great as $|a_k| + 1$, and the ϕ expression in the summation in (6.18) could be as great as $\frac{a_i}{m_k} + 1$. This entire expression will be multiplied by the absolute value of the coefficient of x_q in (6.16), which could be as great as $|a_q| + 1$.

As can be seen, each time a synthetic variable is introduced in an attempt to drive a coefficient to ± 1, substituting the expression for occurrences of the variable to be eliminated may cause the absolute values of the remaining variables in the system to increase significantly, perhaps preventing the Dark Shadows Inequality from being successfully applied once a coefficient of ± 1 appears in the equality constraint and the Omega Test begins to eliminate variables as described in the preceding chapters.

6.2. Symbolic Projection and Protected Variables

In this thesis I have described the Omega Test as a yes/no decision procedure for dependence analysis. In order to do this, the test eliminates all variables from the system and then bases its decision on whether a contradiction was encountered at any point in the process, However, it is actually more general, which enables it to be applied to a broader range of problems.

For example, one can optionally pass a system of equalities and inequalities to the system (as we have done heretofore), but additionally pass a list of *protected variables* as well. These protected variables form a set of variables P that will not be eliminated. Rather, all non-protected variables will be projected onto the set of protected variables. The output of the Omega Test in this instance will be a set of conditions (inequalities) over the variables in P which if met imply that the original system has an integer solution.

The Omega Test enables this functionality by not only excluding the protected variables from elimination, but also by recording the steps it took in eliminating the non-protected variables. For example, during the elimination of equality constraints described earlier involving a non-protected variable, a protected variable may undergo substitution via the *sigma* mechanism. As the system will no longer refer to the original variable, the Omega Test must report the *sigma* variable in the output, along with a formula for converting the *sigma* to the original protected variable. Additionally, the elimination of a non-protected variable may trigger the Omega nightmare, so the Omega Test must report the dark shadow calculation as well as the constraints near the lower bound (used in the Nightmare algorithm).

It should be noted that protecting variables may increase the risk of the Omega Test falling into the Nightmare scenario, as a protected variable may be required to drive the bounds far enough apart such that other variables can efficiently be eliminated.

This capability of the Omega Test makes it useful in a variety of contexts. For example, I shall show in a later part of this thesis that protecting variables enables a compiler to create a system of checks that a runtime system can evaluate to determine if certain optimizations are possible. This often arises in cases where loop bounds are unknown at compile time.

6.3. Presburger Formulas

In this thesis, I have discussed FMVE and the Omega Test in the context of dependency analysis, and have focused on how it solves problems in the domain of problems that can also be solved with the Banerjee Bounds test and I-Test. However, FMVE and the Omega Test can be used to solve a much broader range of problems. In general, they can be used as a decision procedure for determining whether a *Presburger formula* is consistent, that is true with respect to normal arithmetic. FMVE accomplishes this when variables in the formula can be taken from the real domain, whereas the Omega Test can determine whether the formula is consistent when the variables are restricted to the integer domain.

A Presburger formula [89, 49, 70] is a theorem proposed in a relatively weak language in first order logic known as *Presburger arithmetic*. It is weak in the sense that the only arithmetic operators it allows are addition, equality, and negation. Multiplication of a numeral and variable is allowed, although it is actually a shortcut for repeated addition. Nevertheless, multiplication of two variables is not allowed. It also cannot express concepts such as divisibility and prime numbers. This makes it less powerful than *Peano arithmetic* [48], which is also qualified and is capable of expressing theorems using multiplication and addition. However, this limitation of Presburger arithmetic has a significant advantage: Presburger arithmetic is decidable because it is not strong (expressive) enough for Gödel's Incompleteness Theorem to apply to it. Specifically, theorems in Presburger arithmetic are:

- complete: Every statement in Presburger arithmetic can be derived from axioms, or else its negation can be proven from its axioms
- consistent: A statement in Presburger arithmetic is either true or false, never both
- decidable: There exists an effective procedure for determining whether a statement in Presburger arithmetic is true or false

FMVE and the Omega Test can serve as decision procedures for Presburger formulas. The key to deciding the validity of a Presburger formula is eliminating qualifiers from the formula, as an unqualified Presburger formula can trivially be validated via the rules of ordinary arithmetic. FMVE and the Omega Test are used to systematically reduce a system with m quantifiers to a system with n quantifiers, with $(n < m)$. Both tests require a system with only existential quantifiers. Furthermore, these existential quantifiers must be in the subclauses in the innermost layer.

The first step is to normalize the formula into the form described above via a series of boolean logic identities. For example, universal quantifiers are converted into existential quantifiers via

$$(\forall x.P(x)) \equiv \neg(\exists x.\neg P(x))$$

and the following are used to convert the system into a disjunction of conjunctions:

$$\begin{aligned}
p \wedge (q \vee r) &\equiv (p \wedge q) \vee (p \wedge r) \\
(p \vee q) \wedge r &\equiv (p \wedge r) \vee (q \wedge r) \\
\neg(p \wedge q) &\equiv \neg p \vee \neg q \\
\neg(p \vee q) &\equiv \neg p \wedge \neg q \\
\neg\neg p &\equiv p
\end{aligned}$$

Additionally, the existential quantifiers can be moved inward via

$$\begin{aligned}
(\exists x.p \vee q) &\equiv (\exists x.p) \vee (\exists x.q) \\
(\exists x.P(x) \wedge q) &\equiv (\exists x.P(x)) \wedge q
\end{aligned}$$

At this point, FMVE or the Omega Test is called upon to systematically remove the existentially quantified variables via a number of axiomatic transformation. In the real domain, FMVE eliminates existentially quantified variables that should look quite familiar at this point:

$$\begin{aligned}
(\exists x.c \leq ax \wedge bx \leq d) &\equiv bc \leq ad \\
(\exists x.c < ax \wedge bx \leq d) &\equiv bc < ad \\
(\exists x.c \leq ax \wedge bx < d) &\equiv bc < ad \\
(\exists x.c < ax \wedge bx < d) &\equiv bc < ad
\end{aligned}$$

The general method to eliminate an existentially qualified variable given a mixture of upper and lower bounds containing both less than and less than or equal inequalities is ([70]):

$$\exists x.(\bigwedge_h c_h \leq a_h x) \wedge (\bigwedge_i c_i \leq a_i x) \wedge (\bigwedge_j b_j x \leq d_j) \wedge (\bigwedge_k b_k x \leq d_k)$$

$$\equiv$$

$$(\bigwedge_{h,j} b_j c_h \leq a_h d_j) \wedge (\bigwedge_{h,k} b_k c_h < a_h d_k) \wedge (\bigwedge_{i,j} b_j c_i \leq a_i d_j) \wedge (\bigwedge_{i,k} b_k c_i < a_i d_k)$$

Despite the cosmetic differences, this is essentially the same procedure described earlier, where each upper bound is compared to each lower bound on x to eliminate x.

In the integer domain, the Omega Test proceeds by first converting equalities to inequalities using the $\widehat{modh}$ and $\widehat{modf}$ functions I've already covered. It then applies the Dark Shadow inequality as it eliminates existentially quantified variables using the procedure described above for the real domain. As expected, if a contradiction is reached after applying the Dark Shadow inequality, but no contradiction is reached after reverting to FMVE, the Omega Test performs an exhaustive search of the iteration space to determine the validity of the formula.

The Omega Library [93] (written at the University of Maryland by Bill Pugh and his team) is a C++ implementation of the Omega Test. It exposes a number of classes which allows a user to express a Presburger formula and to subsequently request the Omega Test to prove or disprove its validity. The Omega Library has been incorporated into several automated proof assistant systems. One example is Coq [92], wherein the Omega Library is used to power the "omega" tactic [21]. However, this implementation does not take advantage of all of the expressivity of the Omega Library, as (for example) it requires that the user remove the quantifiers apriori (via the "intros" tactic, among others). It has also been observed [21] that theorem proving via this tactic is quite slow. Although this may be due to a less than optimal implementation, another possibility is that use cases in general Presburger theorem proving (unlike array dependency analysis) cause the Omega Test to employ the constraint reduction strategy, or to fall into the Omega Nightmare as it exhaustively searches for a definitive answer to the formula validity problem.

6.4. The Frobenius Coin Problem

Investigating the Omega Test led me to a famous problem in linear Diophantine equations that has fascinated researchers for well over a century. The *Frobenius Coin Problem* can be stated as follows: Given a monetary system of n kinds of coins with denominations given by $a_1, a_2, a_3, ..., a_n$ what is the largest amount that cannot be represented? For example, given coins of denomination 7 and 9, one can represent amounts of $7x + 9y(x, y \geq 0)$ (where x and y are the numbers of coins of value 7 and value 9, respectively. In this case, one can verify that 47 cannot be represented by these coins, but any larger amount can.

This problem is attributed to Georg Ferdinand Frobenius (1849-1917) [97], who according to legend, often raised this problem in his lectures. Although no closed form solution to this problem exists for values greater than $n > 3$, the case of $n = 2$ has been well investigated. In particular, James Joseph Sylvester proved in 1884 that for $n = 2$, the largest non-representable number is $a_1 b_1 - a_1 - b_1$ [55, 90]. Said another way, any number $n(n \geq (a_1 - 1)(b_1 - 1))$ is representable as $n = a_1 x + b_1 y(x, y > 0)$. The equivalence to the Dark Shadow term is apparent, and in fact serves the same purpose: defining the point in a set of integers above which all values are representable by an linear equation in integers.

And so a solution from 1884 to a seemingly unrelated problem (that of assuring that a particular monetary amount is achievable using a given set of coins) is the same key to insuring that a linear integer solution in a convex region is captured in the shadow cast when the region is projected onto fewer dimensions.

6.4.1. Additional results

It is interesting to note that since the Frobenius Coin problem is related to the Omega Test, other results long ago obtained related to the latter may in fact be usefully applied to the former.

For example, Sylvester also showed [55, 90] that exactly half of the numbers c between the lower bound and $ab-a-b$ inclusive (that is, $(a-1)(b-1)-1$ are representable as a integer linear combination $(ax+by=c)$. That tells us that 50% of the time the Omega Test falls into the Nightmare exhaustive search, it will on average prove independence 50% of the time.

Another interesting result related to the Frobenius Coin problem is due to to Schur [36] which states that an upper limit on the Frobenius problem with denominations given by $(a_1,a_2,a_3,...,a_k)$ is $(a_1-1)(a_2+a_3+...+a_{k-1})$. This could be usefully employed in a pre-processing step to detect dependencies early in the test.

6.5. FMVE and the Simplex Method

In this section I will briefly discuss Dantzig's Simplex method [25], one of the most important algorithms in the 20th century, as well as its relationship to FMVE.

6.5.1. Overview of the Simplex Algorithm

The Simplex method is a solver for linear programming problems (LPPs), in which a cost function is to be optimized subject to a system of constraints. It accepts a problem in the following *general linear programming problem (GLPP)* form [98]:

optimize: $\mathbf{c}^{\mathbf{T}}\cdot\mathbf{x}$ (the *objective function*)

subject to: $\mathbf{Ax}\leq\mathbf{b}$ and $\forall i, x_i\geq 0$ (the constraints)

where:

$\mathbf{x}=(x_1,\ldots,x_n)$ are the variables of the problem,

$\mathbf{c}=(c_1,\ldots,c_n)$ are the coefficients of the objective function,

$\mathbf{A}$ is a $p\times n$ matrix,

$\mathbf{b}=(b_1,\ldots,b_p)$ are nonnegative constants, and $(\forall j, b_j\geq 0)$

The graphical intuition behind Simplex is visualizing the *feasible solution space* as an n-dimensional convex polytope formed by the intersection of the p hyperplanes. The problem is determining which point within that feasible region causes the objective function to attain its maximum value. The Simplex algorithm relies on the following properties of convex spaces [66]:

- If some point in the feasible region causes the objective function to reach its extreme value, then that point must be at one of the extreme points (informally, "sharp exterior points" where the bounding planes intersect, rather than some point within the region) of the feasible region.

- If one of these extreme points p of the feasible region is not maximal for the objective function, then that point has an incident edge that connects it to another extreme point $p'$where the value of the objective is greater than or equal to that at p. Conversely, if all reachable vertices from p cause the objective function to decrease, then p is the desired global maximum point.

The Simplex method exploits these properties by first finding a *basic feasible solution* (BFS) at an extreme point of the feasible region, and then greedily visiting adjacent extreme points on the surface of the polytope until it reaches the global extreme point that maximizes the value of the objective function. The Simplex method has worst case exponential cost [30], but only for a carefully crafted problem which causes Simplex to visit all extreme points of the feasible region. In practice, Simplex has polynomial complexity, which is why it still is the leading LPP solver in industry.

The key phases of the Simplex method are as follows:

Convert problem from GLPP form to SLPP In this pre-processing phase, the GLPP is converted into *Standard Linear Programming Problem (SLPP)* form. SLPP has the following characteristics:

- The objective function is optionally re-stated so that the "optimization" is defined as maximization
- All inequality constraints are rewritten as equalities via the addition of so-called *slack variables*. That is, if the constraint $3x - 6y + 7z \leq 5$ is one of the inequality constraints, it is expressed as $3x - 6y + 7z + s = 5$ (s is a slack variable that absorbs the difference between the LHS and RHS).

Find an initial BFS As mentioned, this step finds an initial starting extreme point in the feasible region. One of the advantages of expressing the LPP in SLPP form is that the origin is always a feasible extreme point, so a trivial (and common) approach is to select the BFS by setting the variable vector to the zero vector.

Iteratively choose an adjacent BFS from among the adjacent extreme points (pivot) This step begins by selecting the variable x_p in the objective function with the greatest non-negative coefficient (since increasing that variable will cause the objective function to increase the fastest). Next, the amount by which x_p can be increased is determined by examining the constraints and determining the maximum increment to x_p that does not violate the non-negativity constraints. The standard method for doing so is the *minimum ratio test*, which scans the inequalities and selects the tightest bound by examining the ratio of the constraints RHS to the coefficient of x_p in that constraint. Once that constraint is found, it is expressed in terms of x_p and substituted into the remaining constraints, including the objective function (essentially performing a form of Gaussian elimination). This step has the effect of finding and transitioning to a more optimal corner vertex on the polytope, and continues until further iterations serve only to reduce the value of the objective function.

6.5.2. The Simplex Method's Links to FMVE

According to Dantzig [24] his development of the Simplex method was influenced by an 1826 paper by Fourier in which he suggested visiting the exterior extreme points of a polytope successively until an extreme point was reached (although he thought it would be too expensive to be practical). He also quotes Motzkin's PhD thesis which suggested the use of the mathematical simplex as an approach to the LPP (although the simplex does not directly play a role in Dantzig's method).

Although FMVE can be used as a LPP solver using back substitution [87], it is far too impractical to be considered for use in problems involving hundreds or even thousands of constraints, which Simplex routinely solves in real-world scenarios.

Conversely, the branch and bound [51] method can be used in conjunction with Simplex to search for integer solutions to a system of linear inequalities, although it has gained little traction in dependence analysis because Simplex has a relatively high startup cost which makes it unsuitable to the small problems encountered in dependence analysis. FMVE's simple projection strategy of pairwise pairing of bounds makes it much more practical in this realm.

6.6. Work Related to FMVE Refinement

Although it is arguably the most widely known and implemented method of refining FMVE to the integer domain, the Omega Test is not the first approach to adapting linear programming solutions to the integer domain.

One of the first and most widely known attempts is one that still is of at least theoretical interest today. R. Gomory's approach [41] was to iteratively use the Simplex algorithm to create a linear solution to a problem, and then adapt it to integers by using "cutting planes" to shave off fractional solutions. The basis for this approach was actually discovered a few years earlier by Dantzig (the inventor of the simplex method for LP), et al. [23] in an attempt to solve the Traveling Salesman Problem. The approach was of theoretical interest, but never seemed to garner much attention outside of academia due to its runtime costs.

Similar methods have arisen based on the idea of first creating sample solutions and then (expensively) attempting to iteratively refine them using cutting planes or similar search space techniques such as branch and bound [51] (not unlike the Omega Nightmare algorithm). In particular, [56] specifically addresses FMVE. In the paper introducing the Power Test [101], Wolfe, et. al. describe determining when FMVE produces conservative results, but do not implement an exact decision procedure. Shostak, et al. [88] describe a inexact (real-valued) method for deciding Presburger formulas by constructing a convex hull.

Cooper's well-known algorithm [16],[44] for deciding Presburger formulas is in my opinion quite interesting, and the variable elimination algorithm bears a resemblance to that used the Omega Test (although the Dark Shadows inequality is replaced by a set of divisibility tests represented in *Conjunctive Normal Form*(CNF). Like the Omega Test, it removes quantifiers from the inside out, but it doesn't require that Presburger formulas be converted to DNF first. However, it does require several normalization steps, such as conversion of all occurrences of $\leq, \geq$ to $<, >$, respectively, and the use of DeMorgan's laws so that negations are not applied to variables.

6.7. Summary

In this chapter I have discussed several topics that arose during my research into the Omega Test and related techniques. I presented the algorithm used by the Omega Test to force a unit coefficient to appear in the system of constraints if one was not present intially. I also proved that the technique's rate of reduction is greater than than mentioned in the Omega Test paper. I discuss the projection technique that the Omega Test uses to "protect" variables so as to produce a set of conditions that produce an integer solution to a set of linear inequalities. This is relevant to the discussion on direction vectors, as the Omega Test does not test a direction vector hieracrchy as the the I-Test does, but rather attempts to produce a more precise dependence distance by protecting variables. I briefly introduce Presburger formulas, as one must express formulas to the Omega library to solve in terms of these formulas. I then mention the Frobenius problem and describe how the solution to the two-coin instance of the problem (known since the late 19th century) is identical to the Omega Test's Dark Shadow inequality. To my knowledge, this thesis makes the first mention of this correlation. I conclude with an overview of the famous Simplex algorithm and outline its relationship with FMVE.

It is worth noting that the ancillary operations performed by the Omega Test that I've described in chapter, such as forcing unit coefficients, projecting variables, and especially the Omega Library's use of Presburger formulas to express problems contribute to the test's run-time overhead. The library's use of Presburger formulas is particularly worrisome as the client must first express a dependence problem in this form to present to the library, and then the library must convert the problem into canonical format and into internal data structures as described in section 6.3. I suggest this may explain why the Omega Test has been observed [80] to be *slightly* less efficient than the I-Test, even when non-exponential. However, in order to conclusively prove this hypothesis, one would need to instrument and profile the Omega Library while an appropriate compiler compiled a benchmark suite.

Part III.

FMVE-Based Program Constraint Analysis

Chapter 7

FMVE-based Approaches to Redundant Bounds Check Elimination in Java Programs

7.1. Chapter Introduction

To this point, I have discussed FMVE in the context of dependence analysis, and especially (in some detail) how the Omega Test employs it and several related algorithms to enable it to solve general Presburger formulas. In this chapter, I briefly describe peer reviewed and published work that I've done that uses a modified form of FMVE in the context of a different program optimization - the elimination of redundant array bound checks. As this thesis concerns itself chiefly with FMVE, I will only briefly describe the details of the algorithm dealing with orthogonal issues such as flow control and provability. For more details and psuedo-code, I would refer the interested reader to the published literature [85][69][38][37].

As I will develop more fully in the introduction to the Constraint Analysis System (CAS), the problem we were addressing is that in order to reduce the costs of runtime program analysis, deeper analysis can be done at compile time, but much of the information needed to make the correct optimization decisions may not be known. If necessary and sufficient conditions for the safe application of an optimization can be determined at compile time, they can be encoded as a series of checks in the code (which detract from performance), or else somehow communicated to the runtime system where they can be checked once all key compile-time unknowns have been resolved. The optimization can then be fully implemented prior to actual program execution. In the case of Java, a way to encode such meta information already exists: the classfile architecture [72] allows the inclusion of *user-defined attributes* that can be interpreted by the runtime system or else simply ignored.

But this raises another issue. How can the runtime be sure that a malicious pre-processor did not encode "optimizations" that were actually unsafe? If the runtime were to blindly trust these optimizations, then the safety guarantees afforded by the Java architecture would be rendered useless. Consequently, not only does the preprocessor have to identify optimization opportunities (such as array accesses that are always safe or safe given a set of conditions), it must also be able to record its reasoning in the classfile in a way that can be verified by a "sceptical" runtime system.

My contribution was developing a technique to identify potential or actual opportunities to eliminate array bound checks and to pass to other components in the system the bookkeeping information behind the reasoning so that it could be recorded in classfile attributes and later checked and acted upon at runtime.

My idea was to construct a system of linear inequalities by traversing an intermediate representation of the code in *extended Static Single Assignment (e-SSA)* form (to be described). Having done so, I then create a new system by adding a *proposed constraint* to the system that stated that an array access was unsafe (i.e. beyond the bounds of the array). Since the original system was derived directly from the compiler, it is assumed to be consistent. If the new system (including the proposed constraint) was demonstrably inconsistent (determined by using a variation of FMVE), then it must be because the proposed constraint was inconsistent with the semantics of the program and must be rejected. The implication is that the array access is in fact safe. In order to support the reasoning, references to the instructions giving rise to the constraints in the system as well as to the relevant basic blocks was reported as well. This is the common theme in both the Constraint Analysis System as well as the BQVE system.

It is worth noting that I originated the idea of CAS as a purely runtime optimization scheme. In fact, the original prototype was meant to be used at class load time, where components were dynamically incorporated into the JVM only when needed as part of Java's class loading mechanism. For this reason, performance was of paramount importance, even at the cost of forgoing analysis of potential optimization hiding in complicated flow-control graphs. Nonetheless, we used CAS as an offline analyzer because it was still able to identify profitable optimization opportunities, could handle general linear constraints, and its basis on FMVE made it possible to record its reasoning in a way that could be verified at runtime.

In contrast, BQVE is an extension to my work meant for offline analysis from the ground up. In BQVE, colleagues augmented the constraints in the system with references to the blocks in which they were valid and synthesized additional blocks in order to deduce additional constraints on variables.

7.2. Overview of the Constraint Analysis System

The Java Virtual Machine specification requires that all array accesses are checked at run time, and that an out-of-bounds reference cause an *ArrayIndexOutOfBoundsException* to be thrown. However, these run time checks (especially those occurring in nested loops) degrade performance, and so eliminating redundant checks can significantly increase the the performance of Java programs.

Redundant bounds check elimination for Java programs thus relies on the optimizer's ability to determine that the index used to index an array is always both strictly less than the array's upper bound and also non-negative. If the optimizer is able to make that determination, then the bounds check can be safely eliminated. If this analysis is performed at run-time, then it must be done as efficiently as possible so as not to degrade performance. However, techniques that sacrifice precision for efficiency are not able to detect as many redundant checks, with resulting sub-optimal run time performance. Alternatively, if optimizations were able to be performed at compile time, then it is possible to devote more time to the analysis in an attempt to locate more redundant checks. However, any system that relies solely on compile time analysis is susceptible to a malicious optimizer that claims that an unsafe access is actually safe, thereby compromising system integrity. Therefore, the results of the compile time analysis must be encoded, communicated to the run time system, decoded, and verified at run time.

In contrast, traditional approaches generally sacrifice performance for precision, or vice versa. For example, in an attempt to reduce the cost of the analysis, some systems [11, 82] restrict the relationship among the variables to being only simple difference constraints of the form $x - y + c \leq 0$, where x and y are program variables, and c is a constant. Other more precise techniques [20, 7] are not efficient enough to be used at run time, nor are capable of producing verifiable proofs.

In previous work, we described how proofs of the redundancy of a check can be efficiently represented and verified [85]. This chapter describes the Constraint Analysis System (CAS), a symbolic program constraint analyzer that detects redundant checks and provides a proof of its redundancy claims that can be passed to and verified by a run time system. Furthermore, in order to exploit as much known information as possible, CAS uses general linear relationships among variables. That is, our system processes linear constraints over program variables of the form

$$\sum_{1 \le i \le n} a_i x_i + c \le 0 \tag{7.1}$$

CAS constructs a $ConstraintSystem(CS)$ consisting of the variables in a program and the linear relationships among them implied by program logic. It then determines if a *proposed inequality constraint* (a logical relationship among program variables that is not directly implied by program statements) is consistent with those constraints that are *known* to hold because they are directly derived from statements in the program (*program constraints*). CAS does so by finding sequences of program constraint combinations which produce an inconsistent (illogical) result ($0 < c \le 0$) when combined with the proposed constraint. Constraints are combined via elementary row operations to produce an equivalent constraint with fewer variables. A sequence of program constraint combinations corresponds to a particular control flow (CF) path in the program. If each CF path (that is, the corresponding sequence of program constraint combinations) can be combined with the proposed constraint to produce an inconsistency, then the proposed constraint can be rejected. Specifically, if the proposed constraint disproven by CAS states that an index is out of bounds, then the array access is safe. Alternatively, a consistent result indicates the proposed (unsafe) constraint may hold over at least that CF path, and safety checks need to be added. CAS records the sequence of constraint combinations from which an inconsistency or consistency was derived, enabling its reasoning to be later checked by verification systems.

This set produces an inconsistent inequality $0 < c \le 0$ when a sequence of *elementary row operations* are applied to the constraints to eliminate the variables. Exactly one of the n constraints in this sequence is a proposed constraint, whereas the remaining $n-1$ constraints are program constraints. Since program constraints were derived from the program itself, they are by definition consistent with program semantics, and the proposed constraint necessarily causes the inconsistency. CAS employs negated logic, proposing a constraint that implies an index is out of bounds, and so when it discovers inconsistencies it is actually discovering paths for which the index will always be within bounds. Because of the nodes in the CS correspond to variables in the e-SSA representation of the program, which are defined in specific blocks of the control flow graph (CFG), an inconsistent path in the CS means that the proposed constraint never holds in the program executions that traverse those blocks. CAS produces only constraints and variable sequences that correspond to valid paths in the CFG, and further ensures that all CFG paths have been examined and are safe after its graph exploration is complete. These inconsistent paths can then be used to supply a verifiable proof that the bounds check is unnecessary. [1]. When exploring the **CS**, CAS explores all possible paths. Using a set of validity rules and essentially mathematical induction, CAS collapses and summarizes the effect of assignment loops such that its conclusions hold regardless of the number of loop iterations actually taken.

CAS is novel in several respects. First of all, it employs a new elimination algorithm over an e-SSA program representation to ensure that constraint combinations correspond to valid CF paths. Secondly, it is more general than other systems, and the sequence of constraint combinations leading to a contradiction that CAS produces can be quickly validated as authentic by a properly-equipped run time system. This analysis is intra-procedural. However, in Gampe et al. [38], we describe an orthogonal approach similar to that of Würthinger et al. [103] that speculatively removes checks whose redundancy can be determined by inserting an additional but less expensive check at runtime; this technique is effective in reducing the overhead of many of the bounds checks that could only otherwise be removed with an interprocedural static

[1]Alternatively, the proposed constraints could imply that accesses are safe, and if CAS was *unable* to find an inconsistency, then the bounds check would be fully redundant. Although appropriate for a fully run-time implementation, this approach sacrifices verifiability, as CAS can only prove the validity of inconsistencies, not the validity of their *absence*

analysis.

7.2.1. Overview of the ABCD Algorithm

CAS derives from and is meant to extend the ABCD algorithm [11], a lightweight but effective runtime technique for eliminating Java bound checks on demand. ABCD is novel in the sense that it converts the problem of identifying redundant or partially redundant bounds check into the problem of traversing a sparse, weighted *Inequality Graph* (IG) whose nodes and edge weights are derived from simple *difference constraints* [17] among variables in an e-SSA (*extended-Static Single Assignment*) program representation using a customized path search algorithm that uses a non-standard concept of a shortest path. I will briefly outline the ABCD algorithm, and describe these constituent components in greater detail. Section 7.2.1 is particularly relevant, since CAS also makes use of the e-SSA program representation.

e-SSA Representation

ABCD considers program control and data flow to make meaningful conclusions about how program variables relate to one another because the IG used by the system is derived from the program's e-SSA [11] representation. e-SSA extends traditional *Static Single Assignment (SSA)* [22] representation with "π-assignments", which reflect constraints resulting from control flow paths taken subsequent to conditional statements. These π-assignments create new variables (aliases) along control flow paths that are dominated by the outcome of a conditional expression. The π-variables referenced in constraints implicitly identify a particular control flow path within the program, and are later combined via an SSA ϕ-assignment at a control flow merge point. For example, the following code fragment in SSA form:

```
read(a);
read(b)
if (a < b) then
   {Block A}
else
   {Block B}
end
{Rest of program}
```

would be transformed into the equivalent e-SSA fragment:

```
read(a)
read(b)
if (a < b) then
   a_1 = pi(a)    /* a_1 < b_1 in Block A       */
   b_1 = pi(b)
   {Block A}      /* a,b replaced with a_1, b_1 */
else
   a_2 = pi(a)    /* a_2 >= b_2 in Block B      */
   b_2 = pi(b)
   {Block B}      /* a,b replaced with a_2, b_2 */
end
a_3 = phi(a_1, a_2)
b_3 = phi(b_1, b_2)
{Rest of program} /* a,b replaced with a_3, b_3 */
```

There are several interesting properties of e-SSA representation. First of all, its π assignments allow the encoding of constraints arising from conditional expressions or successful bounds checks. Secondly, variable live range splitting via π and ϕ assignments mean that constraints derived from a program in e-SSA form are valid wherever the variables they reference are live. This latter property allows the IG derived from a program in e-SSA form to be flow-insensitive.

ABCD Inequality Graph (IG)

Once ABCD has constructed an e-SSA representation of the program, it identifies simple difference constraints [17] of the form $x - y \leq c$ where x and y are program variables or a symbolic literal (an important example is the length of an array) and c is a constant. For example, the Java assignment $a = b + 1$ would result in the inferred constraint $a - b \leq 1$.

Once all the constraints are gathered, the IG is constructed. For the constraint $x - y \leq c$, two nodes representing x and y are added to the IG, and an edge from y to x with weight c.

ABCD's Shortest Path

The fact that ABCD represents only simple difference constraints in the graph instead of more complicated linear relationships has the happy consequence of keeping the graph sparse. More importantly, it means that relationships among program variables can be derived via a non-standard shortest path search algorithm.

The reason that the shortest path is non-standard is because ϕ nodes force the algorithm to conservatively select the *greatest* incoming weight (i.e. the *weakest* constraint on the ϕ variable), whereas other nodes select the *least* incoming weight (i.e. the *strongest* constraint on the variable).

The algorithm employed by ABCD takes this distinction into consideration by distinguishing ϕ nodes from non-ϕ nodes and propagating weights accordingly. It detects negative and positive weight cycles by tracking the nodes already visited along a path and the path weight when it was visited, and also makes heavy use of memoization.

For an array access of the form A[x], ABCD searches for shortest path from the node $A.length$ (i.e. the node representing the symbolic length of the array A) to the node x. If the length of that path has negative length, then ABCD can infer that x will always be less than the length of the array and the upper bound check can be eliminated. The lower bound check elimination process is symmetrical.

ABCD's Efficiency

There are two reasons why ABCD is efficient. As has been mentioned, representing only difference constraints means that the IG is relatively sparse and the shortest path algorithm converges quickly. Secondly, ABCD can be incorporated into a JIT compiler and only invoked if a particular array check is a program "hot spot". For this reason, ABCD has become a widely-used algorithm, and has been incorporated into several production compiler systems.

e-SSA and Array Aliasing in Java

Aliasing [47] refers to a situation where the same memory object can be referenced by different names, and is a complication with which optimizers have to contend in order to ensure the generation of safe code. This is a notoriously difficult issue that most often manifests itself in languages such as C and C++ which allow pointers to variables to be accessed and passed and manipulated as variables themselves. Consider the following simple C++ code fragment:

```
#include <iostream>
using namespace std;

int main()
{
    int x = 72889;
    int *p = &x;

    cout << "x is: " << x << endl;

    *p = 0;
    cout << "x is: " << x << endl;

}
```

The program fragment produces

```
x is 72889
x is 0
```

A compiler would have to be aware of the fact that both *p* and *x* refer to the same memory location, and track these aliases as part of its analysis.

Despite the fact that one cannot take the address of a variable in Java, the potential for aliasing still exists via references. Consider the following fragment:

```
class Point
{
  double x;
  double y;
}

class PointDemo
{
    public static void main(String args[])
    {
    Point p1 = new Point();

    Point p2 = p1;

    p1.x = 10.0;
    p2.x = 20.0;

    System.out.println("p1.x : " + p1.x);
    System.out.println("p2.x : " + p2.x);

    }
}
```

The program output is:

```
p1.x: 20.0
p2.x: 20.0
```

However, ABCD's array bound analysis is safe due to its use of e-SSA representation, and the fact that it is not possible to change the size of a Java array referenced by a Java local variable. Two kinds of array aliasing can occur. The first occurs when a variable holding the address of an array is reassigned to another variable, and that second variable is redefined. Consider this fragment:

```
1:   a = new long[25];
2:   b = a;
3:   b = new long[10];
4:   ... = a[15];
```

The reference to array *a* on line 4 will in this case be proven redundant. The e-SSA representation of this example will record the definition of *a* on line 1, followed by uses of *a* on lines 2 and 4. No modification of *a* is implied, as lines 2 and 3 do not affect *a*.

The second form of aliasing occurs through references stored in object fields. Consider the following:

```
1:   a.r = new long[25];
2:   b = a;
3:   b.r = new long[10];
4:   ... = a.r[15];
```

After line 2, *a.r* and *b.r* are aliases for the same array. However, the assignment to *b.r* on line 3 redefines that array. Fortunately, e-SSA treats the reference to *a.r* on line 4 as the load of a reference to an unknown array, and therefore does not eliminate the check. Thus, line 4 will result in an *ArrayIndexOutOfBounds* exception being thrown. It is clear that e-SSA (and by extension, ABCD) is safe (although conservative) with regard to potential aliasing.

7.2.2. Elementary Row Operations in CAS

CAS reduces the problem of determining whether a proposed constraint holds in the context of a program to deciding whether a system of linear inequalities is consistent. Several of the techniques used to solve this and related problems, such as Gaussian, Gauss-Jordan, and Fourier-Motzkin elimination, operate on the principle of iteratively reducing the original system to simpler but equivalent forms until it is able to determine consistency (or solutions). One of the fundamental concepts underlying such techniques is that of *elementary row operations*, which are transformations to the set of linear inequalities which do not change the solution set of the system. Techniques such as Gaussian and Gauss-Jordan elimination reduce a matrix representing a system of linear equations into an equivalent but simpler form by performing only the following row operations:

Row Switching A row within the matrix can be switched with another row

Row Multiplication Each element in a row can be multiplied by the same non-zero constant

Row Addition A row can be replaced by the sum of that row and a multiple of another row

A related technique to determine the consistency of a system of linear inequalities via elementary row operations is Fourier-Motzkin Variable Elimination (FMVE) [87]. FMVE continually applies elementary row operations to eliminate variables from the system until the its consistency is readily decidable. It eliminates a variable by combining all upper bounds on a variable x with all lower bounds on x. Whenever a lower bound on x is paired with an upper bound on x, a new inequality constraint is produced in which x does not appear. After all variables have been eliminated, the system contains constraints of the form $c \leq 0$. If all such constraints are valid (that is, c is negative or zero), then the original system is consistent (has a real solution). Otherwise, the original system is inconsistent.

7.2.3. CAS Algorithm in Brief

Unlike other approaches based on DFS graph searches, CAS's approach is based on FMVE: it eliminates a variable from all constraints at the same time (roughly akin to a breadth first graph traversal). CAS constructs an *Constraint System* (CS) and combines constraints within it to reason about inequality relationships imposed by the program's logic and dataflow. Our representation can be loosely thought of as a graph, with nodes representing variables and constraints representing difference constraints between adjacent vertices. Similarly, the algorithm CAS uses to disprove proposed constraints can be likened to a traversal of the graph. Although this analogy makes the algorithm easier to understand, bear in mind that it breaks down if the program contains general linear versus simple difference constraints. We now describe these two facets of CAS: representation via the CS and reasoning via constraint combination.

The CAS Constraint System

CAS builds and manipulates a *ConstraintSystem* (**CS** = (**V**,**C**)) to represent relationships among program variables. **V** contains representations of the variables in the program, while **C** contains linear inequalities of the form in Figure 7.1 representing constraints over those variables. These elements of **CS** are derived from the program's assignment and conditional statements. CAS continually reduces **CS** by eliminating variables in **V** until it is able to determine consistency. To eliminate a variable x_n in **V**, CAS combines each lower bound (LB) on x_n in **C** with every upper bound (UB) on x_n in **C** via elementary row operations that produce a zero coefficient for x_n in the result. At the conclusion of this step, x_n is removed from **V**, all new constraints formed by these combinations are added to **C**, and all former bounds on x_n are removed from **C**. Thus, as processing continues, the set **V** becomes progressively smaller, while **C** (potentially) becomes larger.

Vertices in CS and their Properties

CS contains a vertex for each variable in the e-SSA representation of the program (including π and ϕ variables). Each vertex $v \in \mathbf{V}$ has several properties, including:

v.LB the set of constraints $e \in \mathbf{C}$ representing lower bounds on x_n.

v.UB the set of constraints $e \in \mathbf{C}$ representing upper bounds on x_n.

v.PHI a boolean indicating whether v is a phi variable

Constraints in CS and their Properties

The constraints in **C** represent linear constraints over the program's variables. CAS deals generally with two kinds of constraints: the constraints in **C** that are derived from program statements are called *Program Constraints*, and are known to be mutually consistent, whereas a *Proposed Constraint* represents a linear constraint over variables that CAS attempts to disprove. In our work, a proposed constraint represents an array-out-of-bounds condition (either upper or lower bound). Since the program constraints are self-consistent (arising directly from program logic), and since a proposed constraint state an access is unsafe (exceeds array bounds), an inconsistent system means that the array access is actually safe. Since all constraints represent less-than-or-equal relationships, assignments and equalities are represented by a pair of inverted edges. That is, if the assignment statement $x = y$ occurs in the program, then two constraints are added to **C**: $x - y \leq 0$ and $y - x \leq 0$. In order to distinguish between them and to ensure constraints are combined in a consistent manner, constraints in **C** have a "direction" flag associated with them to distinguish whether data flows from the LHS of the inequality to the RHS, or vice versa. In the earlier example, since $x = y$ implies a data flow from y to x, the constraint $y - x <= 0$ has a "forward" direction, whereas $x - y \leq 0$ that a "reverse" direction. Since inequalities and equalities so not involve the flow of data, they (initially) have a direction of "independent". As we shall see later, if an "independent" constraint is combined with a constraint from an assignment, the new constraint will carry the direction of the assignment. Additionally, proposed constraints are initially assigned an "independent" direction as well. Another flag associated with a constraint is the "proposed" flag which is a binary indication of whether a constraint is a program or proposed constraint. If a proposed constraint is combined with a non-proposed constraint, the proposed constraint is set on the result. Finally, the "deleted" property determines whether the constraint has been logically deleted from the system.

Any constraint in **C** in which x_n's coefficient is negative is a lower bound on x_n, and is added to the collection $x_n.LB$, while those with positive coefficients for x_n are upper bounds and appear in the collection $x_n.UB$. Since a single constraint can contain terms for multiple variables, the same constraint can simultaneously be in the multiple UB or LB collections.

```
   int A[] = new int[y];
   /* y == A.length */
   x0 = 0
L: x1 = phi(x0, x3)
   if (x1 >= y) goto E:
    x2 = pi(x1)
    y1 = pi(y)
    /* x2 < y2 */
    A[x2] = ...
    x3 = x2 + 1
    goto L:
E: x4 = pi(x1)
   y2 = pi(y)
   /* y2 <= x4 */
...
```

Figure 7.1.: e-SSA Version

Constraint	Direction
$x0 - ZERO + 0 \leq 0$	Rev
$-x0 + ZERO + 0 \leq 0$	Fwd
$A.length - y + 0 \leq 0$	Ind
$-A.length + y + 0 \leq 0$	Ind
$x0 - x1 + 0 \leq 0$	Fwd
$-x0 + x1 + 0 \leq 0$	Rev
$x2 - x1 + 0 \leq 0$	Rev
$-x2 + x1 + 0 \leq 0$	Fwd
$y1 - y + 0 \leq 0$	Rev
$-y1 + y + 0 \leq 0$	Fwd
$x2 - y1 + 1 \leq 0$	Ind
$x2 - x3 + 1 \leq 0$	Fwd
$-x2 + x3 - 1 \leq 0$	Rev
$x3 - x1 + 0 \leq 0$	Fwd
$-x3 + x1 + 0 \leq 0$	Rev
$x4 - x1 + 0 \leq 0$	Rev
$-x4 + x1 + 0 \leq 0$	Fwd
$y2 - y + 0 \leq 0$	Rev
$-y2 + y + 0 \leq 0$	Fwd
$y2 - x4 + 0 \leq 0$	Ind

Figure 7.2.: Constraint System (CS)

An Example of a CS

As an example, consider a simple program that allocates an integer array of length y elements, and then assigns to each element in ascending order. The e-SSA listing is shown in Figure 7.1, and the resulting **CS** is shown in Figure 7.2.

Constraint Combination in CAS

Once all program constraints and the proposed constraint are added to **CS**, CAS determines the consistency of the proposed constraint by eliminating variables. Unlike conventional FMVE where each upper bound on a variable is combined with each lower bound on a variable to eliminate it from the system, CAS restricts which constraints can be combined in order to produce a meaningful result.

Direction Compatibility

As a simple example of why this is necessary, consider the consequences of the indiscriminate constraint pairing given the following simple code fragment in which the phi variable x is assigned twice: $x = y; ...; x = z$; The first assignment produces the constraints $x - y + 0 \leq 0$ and $-x + y \leq 0$, and the second $-x + z \leq 0$ and $x - z \leq 0$. Combining upper and lower bounds to eliminate x gives us $z - y \leq 0$ and $-z + y \leq 0$, which imply $y == z$. However, this relationship does not follow based on the above code fragment. Therefore, CAS prohibits the combination of bounds with incompatible directions. That is, the bounds must have the same direction, or at least one must have an "independent" direction. If that condition holds, then the combination's direction is either "independent" if both parents had independent directions, or the nonindependent direction of the parent(s). In the case above, the combinations to eliminate x would be allowed, but the combination of the results would be prohibited.

Sub-Cycle Elimination

CAS also disallows a combination if the same variable was previously eliminated in the production of both the LB and the UB (preventing "sub-cycles" in graph terminology). CAS detects (in)consistencies through the formation of inequalities in which all variables have been eliminated, leaving only the constant term (we refer to these as "reduced" constraints). This can be viewed as a "cycle" in which each term with a positive coefficient is paired with a term with an negative coefficient. For example, combining $x - y \leq 0$ with $y - z + 1 \leq 0$ produces $x - z + 1 \leq 0$, which when combined with $z - x \leq 0$ produces the inconsistency $1 \leq 0$. Cycles such as this can occur as a result of loops, which CAS detects and summarizes in the course of variable elimination. CAS further assumes that loop back edges are traversed either an infinite number of times, or else not taken at all, depending on which gives the most conservative (safest) answer. (As an aside, this is the reason why proposed constraints are assumed to state an unsafe condition – this assumption enables the algorithm to produce conservative but safe results.) A "sub-cycle" corresponds to a loop being taken exactly once, which cannot be assumed to occur during execution.

Phi and Pi nodes

As we shall see, CAS caches the "optimum" intermediate results as constraints are combined. The definition of "optimum" affects performance as well as safety decisions and depends upon whether the variable being eliminated is a phi node. As described in [11], phi nodes are "minimum" nodes, whereas other nodes are "maximum" nodes. We ensure that only the weakest constraints are propagated when a phi node is eliminated, and the strongest constraint otherwise. To achieve this, CAS maintains a cache hash keyed by the set of variable terms (excluding the constant) on the LHS of the inequality, plus the direction of the constraint. The value of the entry is the "best" constraint with those terms. This hash initially is populated by program and proposed constraints as they are added to the system. Thereafter, when a new constraint is produced via LB-UB combinations, the cache is consulted to see if a "better" value (depending on the type of eliminated variable) has already been produced. This is done by comparing the constant in the new constraint with the constant in the cached constraint. If the new constraint and cached constraint are of equal strength, a constraint carrying the "proposed" indicator takes preference. If not, the cache is updated, and the old constraint is deleted from the system.

Caching a reduced result requires a bit of explanation. Consider the combination of $z - y \leq 0$ (Forward) and $y - z \leq 0$ (Forward) to eliminate y. The reduced result ($0 \leq 0$) (Forward) has no terms, and the cache key cannot directly be derived. In this case, CAS constructs a *key constraint* to be used as a hash key. The key constraint is of the the form (α, *direction*), where α is the set of all terms in the LB, excluding the variable being eliminated, and *direction* is the direction of the reduced constraint. For example, suppose we wish to combine $3x + 4y - 2z <= 0$ (Forward) with $-3x - 4y + 2z \leq 0$ (Forward) to eliminate z. Since the result is reduced, our key constraint is ($3x + 4y$, Forward). Furthermore, if the constant in the reduced constraint is non-zero and the reduced constraint is not proposed, then the constant in the key constraint is set to $+\infty$ or $-\infty$, depending on whether the reduced constant is positive or negative, respectively. This allows CAS to safely summarize the effect of loops.

Unbound Variables

CAS presently detects unbound variables but does not generate run-time checks when they are encountered. When a proposed LB is selected for pairing, a check is done to see if the variable being eliminated is unbound. If so, CAS conservatively reports that the proposed constraint holds. Equivalently, the unbound variable will remain after all other variables are eliminated, but this approach allows us to terminate the algorithm quickly.

Initialization

CAS processing begins by initializing an empty **CS**, and program constraints are added as the program is being parsed. CAS creates the internal representations of the constraint and referenced variables, adds new variables to **V**, sets the direction vector in the new constraint, and adds the new constraint to the appropriate variable's LB and UB queues. Additionally, if the constraint arises from an assignment or equality, the constraint is copied, the coefficients and directions inverted in the copy, and the copy is queued to UB and LB queues. Once all the program constraints are added, a proposed constraint is formulated and passed to the `propose()` method. This routine creates and initializes a new constraint, sets its proposed flag, and adds it to appropriate UB and LB collections. Next, assignments in the program are examined. For each assignment constraint where none of the variables in either the LHS or RHS are phi variables, the direction in the original and copied constraint is changed to "Independent". This enables CAS to infer relationships between in simple cases such as $x = 5; y = 6$.

Processing

Next, `propose()` begins eliminating variables by calling the `eliminate()` method. This method combines each LB on a variable with all compatible UBs on the variable, checks for unbound variables, and invokes the method `update-cache()` to manage the cache of "best" constraints. As part of the combination, a new direction is computed, and the parent UB, parent LB, and the eliminated variable are recorded in the result for possible verification at a later time. If a cycle is detected, it creates key constraints and passes them to `update-cache()` to record cycles.

CAS first eliminates all non-phi variables, leaving only the strongest constraints linking the phi variables. Thereafter, the phi variables are eliminated taking only the weakest results at each intermediate stage. After CAS eliminates a variable v, it checks the cache with keys (v, Independent), (v, Forward), and (v, Reverse) for a non-negative proposed cycle. If one is found, it immediately returns "true" meaning that the unsafe proposed constraint may hold. Otherwise, if a negative proposed constraint is found, a flag is set to indicate that at least one inconsistency was encountered.

Termination

Once all variables have been eliminated without discovering a proposed consistency, `propose()` returns "false" if at least one proposed inconsistency was discovered. Otherwise, `propose()` conservatively returns "true".

7.2.4. Experimental Results

CAS was implemented in the SafeTSA compiler [2] and was used to identify redundant bound checks and produce verifiable proofs of its redundancy claims. These proofs were encoded in the form of annotations and were passed to a run-time system for verification. Additionally, we added an annotation verifier and annotation directed optimization into the SafeTSA classloader and JIT compiler of the SafeTSA virtual machine (which is based on Jikes RVM 2.2.0).

For comparison, we also took the implementation of ABCD [11] found in the Jikes RVM Optimizing bytecode compiler [13] and ported it to work with the SafeTSA data structures. Several features of SafeTSA required extensions to the original algorithm, most notably, SafeTSA's type infrastructure. SafeTSA includes several explicit type coercion instructions to simplify type-checking; these were accounted for in

```
proposed_inconsistencies = 0 ;
foreach v in pc do
    if v's coefficient is negative then
        Add pc to v.LB;
    else
        Add pc to v.UB;
    end
end
foreach non-phi v in V do
    if v is assigned in constraint c and c has no phis then
        c.direction = Independent;
    end
end
foreach non-phi v in CS do
    eliminate(v);
end
foreach phi v in CS do
    eliminate(v);
end
if proposed_inconsistencies > 0 then
    return FALSE;
else
    return TRUE;
end
```

Procedure propose(pc)

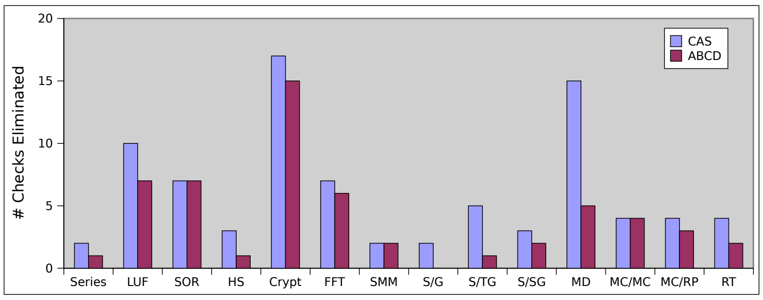

Figure 7.3.: Precision of CAS and ABCD

ABCD by extending ABCD's existing Global Value Numbering system. In addition, since the existing ABCD implementation analyzed only upper-bounds checks, we extended ABCD to support lower-bounds checks.

We evaluated our prototype system using the Java Grande Forum benchmarks [12]. The benchmarks were modified to make some of the array bounds limits be expressed as symbolic constants rather than passed in as parameters so that more array bounds could be eliminated with conservative, intraprocedural analysis. These benchmarks were compiled into SafeTSA and optimized using common subexpression elimination, which eliminates duplicate bounds checks using SafeTSA's safe-element-reference type. This version of each class was used as the baseline to which CAS and ABCD were applied. All of the experiments were conducted on a 1.5GHz G4 PowerMac with 1GB of RAM running a Linux 2.6.15 kernel. All timing measurements were made by repeatedly running the benchmark program in a fresh virtual machine at least 200 times. The first fifty runs were discarded and the mean of the subsequent runs is reported.

Figure 7.3 shows the number of bounds checks (beyond those eliminated by common subexpression elimination) that were eliminated by CAS and ABCD, respectively, in each of the benchmark classes. Since CAS can reason about general linear constraints while ABCD is limited to a subclass of difference

```
foreach LB in v.LB do
    if LB.proposed AND v is unbounded then
        EXIT(TRUE);
    end
    foreach UB in v.UB do
        if LB and UB have compatible directions AND do not form a subcycle then
            new_con = combine(LB,v,UB);
            if new_con is reduced then
                calculate key_constraint from LB and v;
                if new_con.constant $\neq$ 0 and !new_con.proposed then
                    if new_con.constraint $>$ 0 then
                        hash_key.constraint = $+\infty$;
                    else
                        hash_key.constraint = $-\infty$;
                    end
                else
                    hash_key.constant = new_con.constant;
                end
                update-cache(key_constraint);
            else
                update-cache(new_con);
            end
        end
    end
end
delete all bounds in v.UB and v.LB;
if cache[v, *].constant <= 0 then
    EXIT(TRUE)
end
if cache[v, *].constant ¡= 0 then
    proposed_constraints += 1
end
```

Procedure eliminate(v)

```
if cache[(con.terms,con.direction)] exists then
    old_con = cache[(con.terms,con.direction)];
    if con.terms contains a non-phi then
        if con.constant > old_con.constant OR (con.constant == old_con.constant AND
        con.proposed) then
            cache[(con.terms,con.direction)] = con delete old_con;
        end
    else
        if con.constant < old_con.constant OR (con.constant == old_con.constant AND
        con.proposed) then
            cache[(con.terms,con.direction)] = con delete old_con;
        end
    end
else
    cache[(con.terms,con.direction)] = con
end
```

Procedure update-cache(con)

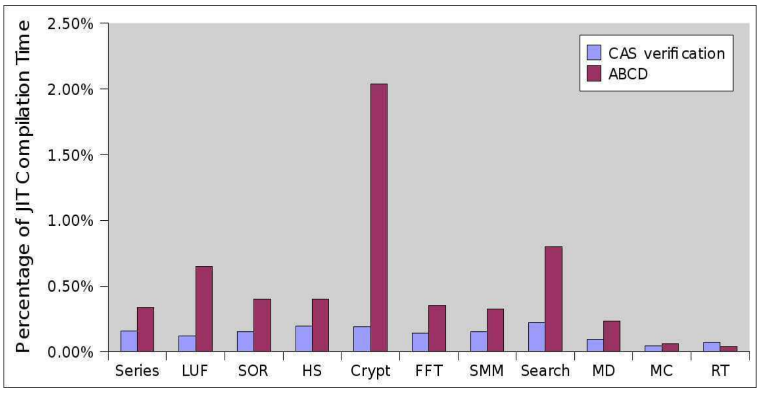

Figure 7.4.: Runtime Cost of CAS and ABCD

constraints (which is a subclass of linear constraints), one would expect CAS to eliminate more bounds checks than ABCD. For our benchmarks, all of the bounds checks removed by ABCD were also eliminated by CAS, and in nearly three-quarters of the classes, CAS's extra precision results in more eliminated bounds checks. Notably, for the the Moldyn benchmark, CAS was able to eliminate 15 bounds checks where ABCD could only remove 5. Manual inspection of several bounds checks revealed that these were bounds checks that the ABCD algorithm is unable to prove because ABCD can only reason about a restricted form of difference constraints. In particular, it only supports constraining the variable being defined by an instruction relative to the variables being used by that same instruction. This restriction speeds up ABCD significantly (since it allows ABCD to piggy back on the existing compiler data structures) but does so at the cost of precision. The following code fragment is an example where this class of difference constraints is insufficient to eliminate a bounds check:

```
x = y;
if (y < A.length) {
   A[x];
}
```

ABCD cannot infer that the access is safe in this case because its traversal algorithm follows the def-use chain, whereas CAS is able to prove the access as safe.

Figure 7.4 shows the cost of the runtime verification of CAS's proofs vs. the runtime use of ABCD as a percentage of baseline JIT compilation time. For CAS this is primarily the time required to verify annotations at run-time, whereas for ABCD this is the time required to carry out the ABCD algorithm. With one exception (RayTracer), verification has a lower runtime cost than ABCD. For some of the benchmarks (notably Crypt, Search, and LUFact), verification is several times faster. The primary reason that verification outperforms the ABCD algorithm is that the verification component only needs to verify those bounds checks that are actually unnecessary, whereas ABCD checks all bounds checks. Compared to the total JIT compilation time, however, both methods impose a very small runtime cost (up to about 0.2% for verification and up to about 2.0% for ABCD).

Figure 7.5 shows the speedup in total benchmark execution time resulting from the use of verification compared to the baseline SafeTSA version of the benchmark. The bounds check elimination resulted in improvements of nearly 10% on the SOR benchmark and over 3% on the SparseMatMult and MolDyn benchmarks. On the other benchmarks the speedup was small or negligible; this is because most of the bounds checks that could be eliminated in those benchmarks were not in the benchmark's inner loop. Since time spent in verification was relatively small (a couple milliseconds) compared to the total execution time (15s–200s), the effect of verification cost is not noticeable.

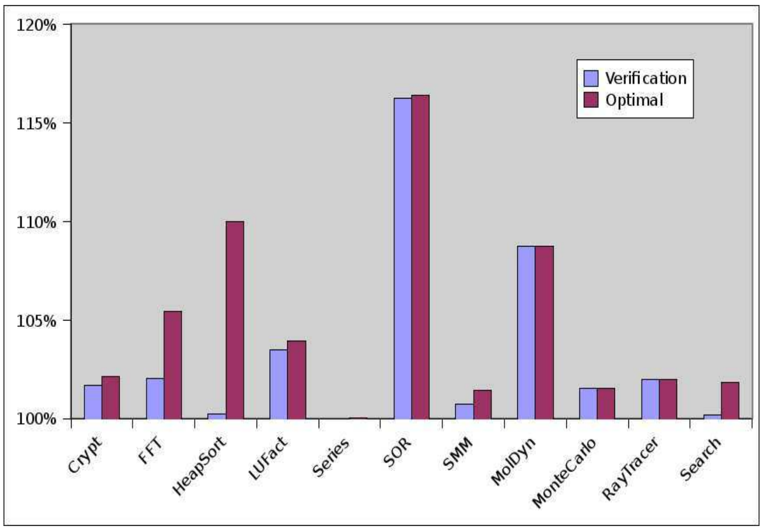

Figure 7.5.: Overall Execution Speedup with CAS

7.2.5. Related Work

There have been several works addressing the array bounds check problems in Java. Moreira et al. [61] used heavy-weight loop-based transformations and optimizations to optimize bounds checks in scientific applications; their goal was to provide a traditional static compiler for Java programs that provides performance approaching that of traditional optimizing compilers for Fortran, so their approach does not support just-in-time compilation and is not a general solution to the Java bounds check problem.

The ABCD algorithm [11] provides a runtime global bounds check elimination based on extended-SSA (e-SSA) form and difference constraints, it is quite efficient but has some limitations since it can only obtain difference constraints that can be overlayed onto the e-SSA graph. Menon et al. [59] extended the ABCD algorithm to produce optimized programs augmented with verifiable proof variables. Like ABCD, CAS represents programs using e-SSA, but CAS's constraint system is more general and allows it to reason about general linear inequalities. Linear programming techniques such as Fourier-Motzkin Variable Elimination [87] also handle such inequalities, but CAS is more efficient because of the path pruning that takes place as the graph is explored. Furthermore, our system takes into account control flow when choosing paths through the graph as well as when presenting proofs.

Qian et al.[82] use an iterative dataflow analysis based on difference constraints to annotate bytecode with an indication of which bounds checks are unnecessary, but it does not provide verifiable proofs of its claims. Chen and Kandemir [15] describe a method for annotating the fixed point of a iterative dataflow analysis of integer variable ranges which can then be verified using a single iteration of the same algorithm, but their constraints are limited to a subset of difference constraints.

Zhao et al.'s [105] optimization is restricted to limited loop forms (and is, therefore, less comprehensive than our approach) but is quite efficient during JIT compilation. Würthinger et al. [103] have developed a bounds check elimination for use in the HotSpot JIT compiler, which similarly identifies simple patterns in the source code but adds speculation to reduce the overhead of some of the bounds checks that cannot be completely eliminated statically.

Besson et al.'s [5] proof-carrying code architecture is based on Cousot's abstract interpretation [19] and produces a certificate at analysis time that is passed to a consumer for certified verification. Their system checks if all accesses within a program are safe, rather than testing the safety of individual array accesses. Their experimental results are based on a small set of codes rather than general benchmarks, and their technique appears to be less efficient than our approach.

7.2.6. CAS: Discussion

I present in this part of the thesis a lightweight theorem prover suitable for determining whether a proposed constraint over variables is consistent with the semantics of a particular code fragment. CAS uses e-SSA representation to form a constraint system expressing relationships among variables with sufficient control flow information to ensure semantically meaningful conclusions. It then simplifies the system until it can determine whether the proposed constraint is consistent with the system. This information can be used to determine that array bounds checking is not necessary along particular control flow paths. We present experimental results that demonstrate that CAS finds more redundant checks than previous work, and improves the runtime of JAVA benchmarks up to 10%. Our results show how CAS is useful when used at compile time to identify fully or partially redundant bounds checks, and having its conclusions encoded in the form of annotations which are efficiently verified at run-time.

7.3. BQVE: Block Qualified Variable Elimination

The *Block Qualified Variable Elimination (BQVE)* algorithm is based on CAS and extends it by qualifying each constraint with a *Region of Validity (ROV)]* where the constraint is known to hold. Primarily due to von Ronne in [38][37], BQVE allows multiple proposed constraints to be tested at once with synthesized blocks in the e-SSA control flow graph.

Being derived from CAS, BQVE initializes in a similar fashion. The e-SSA representation of the graph is read and linear constraints deduced from assignment instructions and targets of conditional expressions. However, and as mentioned, the key feature of BQVE is that a linear constraint is qualified by the location in the e-SSA graph where the expression is known to hold. For constraints derived from conditional expressions, the inequality is known to hold in the region of the code dominated by the target of the conditional branch (or fall through). For constraints arising from assignments, the constraint holds as long as the variables involved in the variable definition are in scope and unmodified (and since the graph is in e-SSA form, that means at all locations dominated by the block in which the assignment occurs). Furthermore, when constraints are combined to eliminate a variable, the resulting constraint is likewise qualified by the area of the e-SSA graph where both of the the combined constraints were valid (the region of validity of the resulting constraint is essentially the intersection of the ROVs of the constraints combined to create it):

$$ROV(m,n) = \begin{cases} ROV(m) & : \; if\ n\ dom\ m \\ ROV(n) & : \; if\ m\ dom\ n \\ nil & : \; otherwise \end{cases}$$

When BQVE adds a proposed constraint P to the system that is assumed to hold in some basic block B, it *synthesizes* a new basic block P_B, such that P_B is dominated by all blocks that dominate B (including B itself), and P_B dominates only itself. If an inconsistency is eventually reached that involved P, the inconsistent constraint will carry a qualifier of P_B. Alternatively, an inconsistent constraint without a synthetic block as its ROV was caused by an unreachable code block. Multiple assumed constraints can be tested simultaneously using this method.

BQVE also creates synthetic ϕ parameters for each predecessor block of a *phi* join node. To ensure safety, only the weakest constraint on one of these synthetic *phi* variables are transferred (the literature refers to this as *lifting*).

BQVE attempts to preserve precision as constraints are combined by eliminating ϕ variables only after non-*phi* variables have been eliminated, and by cloning certain variables so constraints derived from their elimination can be made stronger as edges through the CFG are explored. Additionally, cycles detected

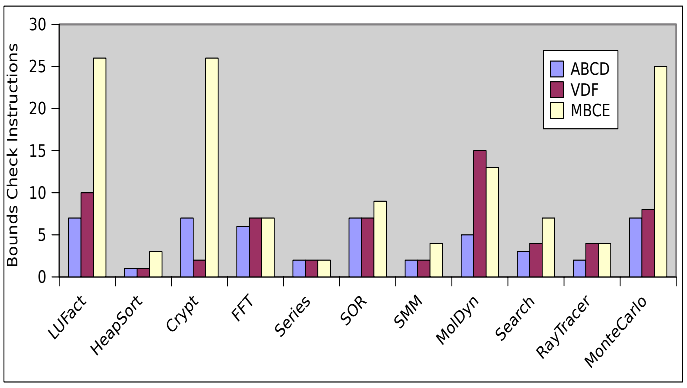

Figure 7.6.: Array Bound Checks Removed by BQVE-equipped System

during graph exploration may indicate monotonically increasing or decreasing loop iteration variables, and BQVE incorporates logic to assign precise lower (upper) bounds in the case of monotonically increasing (decreasing) cycles.

Finally, the Java Virtual Machine specification dictates that integer overflow and underflow results in “silently” wrapping using 2’s complement arithmetic. This can cause issues as, for example, a sufficiently large index being incremented in a loop could wrap and become negative, which should cause an IndexOutOfBoundsException. BQVE provides safeguards by adding constraints to the system which specify that such overflow is possible, and then using the CAS inverse logic machinery it inherited from CAS to disprove the conjectures.

Figure 7.6 demonstrates that the BQVE system was quite successful in identifying and removing bound checks from the Java Grande Forum Benchmarks [75]. The graph indicates how Gampe’s BQVE-equipped system (MBCE) [37] fared against Bodik’s ABCD [11] and Chen and Kandemir’s Verifiable Attribute [15] algorithms on a set of benchmarks.

7.3.1. Related Work

Xi and Pfenning’s [104] used a restrictive form of dependent type systems in Standard ML to address array bounds checking. They rely on program annotations to develop a system of linear inequalities, whereas BQVE is designed for imperative languages and does not depend on annotations. Dor [31] employs a system based on Couset’s work [20] to address the related problem of detecting buffer overruns, but this technique requires the creation and solution of a series of linear programming problems. Ganapathy [39] also addresses the buffer overrun problem, and attempts to reduce the problem to a single system of linear inequalities, but appears to lose precision by not fully taking all control flow paths into account. Rugina [86] also reduces the problem to a single system of inequalities, but restricts coefficients to positive integers.

7.3.2. Satisfiability Modulo Theories (SMT)

Satisfiability Modulo Theories (SMT) [65][64] is potentially quite useful in program optimization in general, and in detecting redundant array bound checks specifically. SMT can be thought of as a generalized *propositional constraint* Satisfiability (SAT) solver, which accepts a formula over a *theory* (a set of sentences or axioms that describe or govern how symbols in a formula interact or are to be interpreted) and decides whether the formula is satisfiable (that is, whether there is an assignment to the variables in the formula that makes the formula true). We’ve seen that the Omega Test can be used to decide whether a formula over Presburger arithmetic is satisfiable. SMTs generalize this by using a set of modules per theory (i.e. Peano arithmetic, boolean logic, arrays, difference arithmetic, bit-vectors, etc) with domain-specific heuristics and algorithms along with a generalized solution search (theorem prover) engine (most commonly based on the well-known DPLL algorithm [27]).

Z3 [28] is an automated SMT solver that employs the DPLL as a core theory solver, along with embedded support for a number of theories, such as linear arithmetic, bit-vectors, arrays, tuples, etc. It allows the expression of formulas either via a textual format, or else via a programmtic API (similar in spirit to the Omega Test's API for expressing formulas over Presburger arithmetic).

Despite its promise, SMT solvers are still relatively slow, and most offer no way of automatically producing a verifiable proof. Even when a verifiable proof is produced, it has been noted [63] that verifying the proof can be an order of magnitude more expensive than producing the proof itself.

7.4. Concluding Remarks

This chapter was included in the thesis because the two algorithms I briefly discussed are based on FMVE, and thus are related to the main theme of the thesis. I thought it demonstrates that although FMVE seems a more natural fit for inclusion in dependency analysis algorithms such as those I discussed in the preceding part of the thesis, it is just as useful in other areas of program optimization, such as the elimination of redundant array bound checks in Java byte code.

The second reason I included it was to demonstrate that my work (and the work derived from it) related to using FMVE as a lightweight theorem prover, and the notion of injecting proposed constraints into a system to test arbitrary conjectured relationships among program variables, was an additional contribution.

Part IV.

Conclusion

Chapter 8

Conclusion

As each of the preceding sections are relatively self-contained, and as each concluded with summary remarks, the conclusion to this thesis will be brief.

8.1. Summary of Contributions

The contribution of this thesis consists of the answers to the research questions raised in section 1.3. I summarize these findings below.

- Although the Fourier-Motzkin technique is in general applicable to more general dependence problems, under the conditions I specified in section 2.4, Fourier-Motzkin and the Banerjee Bounds Test are equivalent: each test takes the same steps and adjusts bounds in an identical fashion as it eliminates terms, and ultimately reaches the same conclusion as to the existence of real solutions to the dependence problem.
- Likewise, I have proven that the Omega Test's refinement to Fourier-Motzkin (the Dark Shadows Inequality) is exactly equivalent to the the I-Test's refinement to the Banerjee Bounds Test (the "accuracy condition"): the real solution is restricted to the same integer sub-solution, and neither refinement subsumes the other.
- When the Omega Test requires exponential time to determine an exact solution to the dependence problem, the I-Test is guaranteed to return an inexact "maybe" answer. It is not possible for the I-Test to be exact and efficient if the Omega Test requires exponential time to determine an exact answer.
- Under the conditions I specified in section 2.4, when the I-Test returns an inexact "maybe" answer, the Omega Test is guaranteed to require exponential time to return an exact answer. It is not possible for the Omega Test to be efficient and exact if the I-Test returns an inexact "maybe" answer. However, if the conditions in section 2.4 are relaxed, then it is possible for the Omega Test to be efficient and exact while the I-Test returns an inexact "maybe" answer.
- As a result of the above, using the I-Test as a filter and resorting to the Omega Test only when the I-Test is inexact offers very little advantage: the Omega Test will detect that it needs to resort to an exhaustive search in virtually the same time as it would take the I-Test to realize that it cannot produce an exact answer and must resort to the Omega Test.
- The most likely reason the Omega Test (implemented via use of the Omega library) has been observed to be slightly less efficient than the I-Test even when it is non-exponential is very likely to be related to

the overhead of the compiler's expressing a problem in a Presburger formula and the Omega library's parsing and converting that expression into data structures before the test actually begins. Other ancillary steps taken by the Omega library, such as forcing a unit coefficient in the system, are also expensive, but are not relevant given the conditions in section 2.4. Profiling of the library would be necessary to definitively answer this question.

- The Direction-Vector I-Test is a customized test to handle a special instance of the dependence problem which arises when a simple inequality of the form $x_i < x_j$ is added to the dependence problem of the form described in chapter 4. I derived and described the conditions under which each test could at least theoretically perform better than the other, and noted that these conditions are highly unlikely to arise in practice, meaning that the equivalence of the tests for all practical purposes extends to the case when direction vectors are added to the problem.
- The Banerjee Bounds Test could reasonably be replaced with Fourier-Motzkin with little practical negative impact; Fourier-Motzkin will report the same answer as Banerjee's Test at the same cost if the iteration space is simple enough for the latter to accommodate. On the other hand, FMVE is applicable to more general iteration spaces than Banerjee's Test. The expense of Fourier-Motzkin's bound pairings will certainly increase rapidly with the number of constraints in the system, but the number of constraints arising from dependence problems is unlikely to be so great that compilation time becomes untenable.
- Replacing the Omega Test with Fourier-Motzkin in the hopes of reducing compilation time is slightly more complicated in light of the Omega Test's use of variable protection to produce exact dependence distances. If the cost of compilation time is more of an issue than the potential loss of a relatively small number of broken dependences, then FMVE could also replace the Omega Test, at least at lower optimization levels. When direction vectors need to be considered (for example, of loop interchange is being contemplated by the compiler), then testing the hierarchy as is done by Banerjee and the I-Test would have to be done. Alternatively, FMVE would have to be augmented with a suitable variable protection mechanism by the implementation. Nonetheless, trading the expense of the Omega Test nightmare for that of direction vector hierarchy testing is still very likely to result in reduced compilation times, at the cost of slightly fewer broken dependences.
- If a non-unit coefficient does not appear in the original problem, the Omega Test employs a substitution technique to reduce the coefficients in the system until a unit coefficient appears. The rate at which this technique reduces coefficients as described in the Omega Test paper is conservative. The reduced coefficient for x_i after the Omega Test substitution algorithm is used is actually at most $\frac{a_i}{3}+1$, where a_i is the original value of the coefficient.
- As a minor historical curiosity, the Dark Shadow inequality used to refine Fourier Motzkin to the integer domain is actually the same formula in Sylvester's 1884 solution to the two-coin Frobenius Coin Problem, and I show the equivalence of the two problems.

8.2. Discussion

This thesis arose from the observation that although numerous works in the literature exist which empirically compare the various dependence tests I examine in this work, there has been no work done to examine the tests from a purely analytical point of view. Without reducing the tests to their mathematical underpinnings and examining them in the abstract, it is difficult to definitively explain the observations and importantly, to defend some of the conclusions drawn in these empirical studies.

This is by no means an attempt to minimize the significant contributions made by this empirical work. In fact, this thesis highlights the usefulness of even further empirical studies. As an example, I mention the usefulness of instrumenting the Omega Test to determine whether Presburger formula conversions indeed accounts for the observed Omega Test overhead (as I suggest). As another example, I cite how analytical work could determine whether replacing the Omega Test with Fourier Motzkin in an attempt to reduce compilation cost is practical given that doing so would require testing direction vector hierarchies to compensate for the loss of the Omega Test's variable protection technique to determine precise dependence distances.

Nonetheless, empirical work alone (no matter how carefully done) is likely to raise as many questions as it answers. Such studies generally involve researchers implementing their technique in a compiler (most likely a research compiler such Polaris [9] or SUIF [99]). For purposes of comparison, the researchers would then have to implement the algorithms of other researchers, run a series of tests using established benchmarks, describe the experimental conditions (benchmark names, compiler optimization flags, machine architecture, etc), present the results, and in some cases attempt to rationalize why some results favored one test while others favored another. Reviewers of such empirical results submitted to conferences or journals (not unreasonably) raised questions as to whether the researchers completely understood competing techniques and implemented them as efficiently as the original authors might have. If the authors attempted to use the published results of other researchers, and attempted to create as faithfully as possible identical conditions, questions nonetheless might be raised about slight differences in test conditions, such as cache configuration, or perhaps in the way results were tabulated, or about whether newer developments in the competing algorithms were incorporated that might favor a particular implementation environment. And such a publication might trigger another empirical comparison in response, and the cycle continued.

Even if a research group provided a library that implemented their technique and was optimized by the original researchers as much as possible, empirical results using that library were still subject to some degree of debate. As a case in point, this occurred with the publication of the Omega Library [93], a tool for deciding the satisfiability of general Presburger formulas, and thus expected problems for it to solve to be phrased as such. Even simple systems had to be built up using the Presburger-oriented APIs provided by the library. Then the general-purpose library would begin the steps we discussed earlier, including quantifier elimination, conversion to DNF, simplification, identifying protected variables, etc, in order to reproduce the original system. Results of the processing then similarly had to be converted to a canonical format before being returned to the caller. This overhead alone could easily dominate the time taken to return a result to the caller of the Omega Library, and it was thus difficult to fairly characterize the performance of the techniques.

In general, despite the fact that the overwhelming majority of such literature was produced with the utmost integrity, and every attempt was made to be as fair as possible, empirical comparisons seemed to give rise to as many questions as they answered. For example, if the Omega Test were to bypass the exhaustive search and simply return an inexact "maybe", would its performance be comparable to the I-Test? Were the accuracy conditions of the two tests directly comparable, and if so, was one a subset of the other, and what were the precise conditions under which one test would be more overly conservative?

These were the questions which had never been addressed in the literature, and are the questions I wanted to answer. This thesis shows that the original I-Test and the Omega Test are more than equivalent when both are applicable – they are actually identical, meaning that they proceed in lock-step fashion under the conditions mentioned in section 2.4. I show the same isomorphism between FMVE and the Banerjee Bounds Test (from which the Omega Test and the I-Test respectively derive and refine), again under conditions under which both are applicable. This is the major contribution of this thesis, and most importantly, serves to explain many observed empirical results in comparisons among the tests. These results are considered relevant and of some value in the community, as evidenced by the publication of a portion of this work in the May, 2018 edition of the International Journal of Parallel Programming [68]. When direction vectors are added to the mix, then I've shown that either the I-Test or the Omega Test may be more accurate than the other in some extremely rare circumstances. Specifically, the DV I-Test precisely calculates the accuracy condition in this case (i.e. is a hand-crafted test for this instance of the problem), but may be thwarted by its insistence that both variables related by the direction vector must be moved simultaneously (which may mean the accuracy condition may fail). The Omega Test, on the other hand, does not treat the inclusion of constraints similar to those introduced by direction vectors as a special case, and uses the same FMVE-based variable elimination scheme to eliminate the related variables independently. This may mean that the elimination of the first variable increases the distance between the upper and lower limits such that all variables can be eliminated and the result still be valid in the integer domain. However, one can at least theoretically construct conditions (as I've done in chapter 5) where the FMVE-based solution calculates overly-restrictive bounds.

I discuss the Omega Test as a decision procedure for the satisfiability of formulas over Presburger arithmetic, comment on its correlation to the Frobenius Coin Problem, and show that a result from the late nineteenth century (Sylvester's Theorem) is the same idea that the Dark Shadows in predicated on. At least in theory, that may allow research done into that well-known problem to be useful in optimizing the Omega Test further. Also, I prove that the effectiveness of the Omega Test's equality elimination algorithm based on substitution is actually better than was previously known.

Finally, I present an overview of my work in bringing FMVE to bear on the problem of redundant array bound check elimination in Java via the Constraint Analysis system, and also describe how that work served as a basis for BQVE. I also showed several graphs demonstrating the effectiveness of these techniques in the elimination of these redundant checks.

8.3. Future Work

During preparation of this thesis and review of the literature, I have identified several opportunities for further research which I am anxious to pursue.

- Determining how much of the observed overhead of the Omega Test (even when non-exponential) is due to conversion to and from Presburger formulas
- Researching the question of whether reducing compilation times is best achieved via a specialized Omega Test that delivers a maybe answer upon determining a real solution may exist when an integer solution cannot be proven to exist, or alternatively using Fourier-Motzkin (possibly after augmenting it with a variable projection technique to efficiently handle direction vector testing).
- Extending FMVE into more complicated non-convex iteration spaces by generalizing the concept of eliminating a variable via a linear combination of constraints to elimination based on functional composition of simple non-linear functions.
- Exploring whether a derivative of the simplex algorithm might be devised that is efficient and effective when applied to problems derived from dependence analysis.
- Improving the Omega Test Nightmare performance by using the ideas behind the branch and bound technique in integer programming
- Simplify and expand on the ideas behind the CAS algorithm to develop a successor algorithm that employs FMVE as the machinery to traverse and reason about a program representation in e-SSA form.

Appendices

DRC 16

MASSEY UNIVERSITY
GRADUATE RESEARCH SCHOOL

STATEMENT OF CONTRIBUTION TO DOCTORAL THESIS CONTAINING PUBLICATIONS

(To appear at the end of each thesis chapter/section/appendix submitted as an article/paper or collected as an appendix at the end of the thesis)

We, the candidate and the candidate's Principal Supervisor, certify that all co-authors have consented to their work being included in the thesis and they have accepted the candidate's contribution as indicated below in the *Statement of Originality*.

Name of Candidate: David Niedzielski

Name/Title of Principal Supervisor: Dr Martin Johnson

Name of Published Research Output and full reference:

David Niedzielski, Kleanthis Psarris and Theoharis Theoharis. "An Analytical Evaluation of Data Dependence Analysis Techniques". In: International Journal of Parallel Programming (2018). ISSN: 1573-7640. DOI: 10.1007/s10766- 018- 0577- 7. URL: https://doi.org/10.1007/s10766-018-0577-7.

In which Chapter is the Published Work: 4,5

Please indicate either:

- The percentage of the Published Work that was contributed by the candidate: 100%

  and / or

- Describe the contribution that the candidate has made to the Published Work:

  The work described in the paper is 100% the work of the candidate, the other authors contributed some of the literature review and minor style edits.

David Niedzielski
Candidate's Signature

17/8/2018
Date

Martin Johnson Digitally signed by Martin Johnson DN: cn=Martin Johnson, o=massey, ou=inms, email=M.J.Johnson@massey.ac.nz, c=NZ Date: 2018.08.16 09:50:06 +12'00'
Principal Supervisor's signature

16/8/18
Date

DRC 16

MASSEY UNIVERSITY
GRADUATE RESEARCH SCHOOL

STATEMENT OF CONTRIBUTION TO DOCTORAL THESIS CONTAINING PUBLICATIONS

(To appear at the end of each thesis chapter/section/appendix submitted as an article/paper or collected as an appendix at the end of the thesis)

We, the candidate and the candidate's Principal Supervisor, certify that all co-authors have consented to their work being included in the thesis and they have accepted the candidate's contribution as indicated below in the *Statement of Originality*.

Name of Candidate: David Niedzielski

Name/Title of Principal Supervisor: Dr Martin Johnson

Name of Published Research Output and full reference:

David Niedzielski et al. "A Verifiable, Control Flow Aware Constraint Analyzer for Bounds Check Elimination". In: Static Analysis. Ed. by Jens Palsberg and Zhendong Su. Vol. 5673. Lecture Notes in Computer Science. 10.1007/978-3-642-03237-0-11. Springer Berlin / Heidelberg, 2009, pp. 137–153. URL: http://dx.doi.org/10.1007/978-3-642-03237-0-11.

In which Chapter is the Published Work: 7

Please indicate either:

- The percentage of the Published Work that was contributed by the candidate: 40%

  and / or

- Describe the contribution that the candidate has made to the Published Work:

  As described in Chapter 7, the candidate came up with the basic redundant check identification idea/algorithm, and the co-authors took his output and used it at runtime to actually remove the checks.

[illegible]
Candidate's Signature

17/8/18
Date

Martin Johnson
Digitally signed by Martin Johnson
DN: cn=Martin Johnson, o=massey, ou=inms, email=M.J.Johnson@massey.ac.nz, c=NZ
Date: 2018.08.16 09:59:08 +12'00'
Principal Supervisor's signature

16/8/18
Date

Bibliography

[1] Gene M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560. URL: http://doi.acm.org/10.1145/1465482.1465560.

[2] Wolfram Amme, Jeffery von Ronne and Michael Franz. “SSA-based mobile code: Implementation and empirical evaluation”. In: *ACM Trans. Archit. Code Optim.* 4.2 (2007), Article 13. ISSN: 1544-3566. DOI: http://doi.acm.org/10.1145/1250727.1250733.

[3] U. Banerjee. *Dependence Analysis*. Loop Transformation for Restructuring Compilers. Springer US, 2007. ISBN: 9780585281223. URL: https://books.google.co.nz/books?id=ZjZjFVY5iBsC.

[4] Utpal K. Banerjee. *Dependence Analysis for Supercomputing*. Norwell, MA, USA: Kluwer Academic Publishers, 1988. ISBN: 0898382890.

[5] Frédéric Besson, Thomas Jensen and David Pichardie. “Proof-carrying code from certified abstract interpretation and fixpoint compression”. In: *Theoretical Computer Science* 364.3 (2006), pp. 273 –291. ISSN: 0304-3975. DOI: http://dx.doi.org/10.1016/j.tcs.2006.08.012.

[6] Nadya Bliss. “Addressing the Multicore Trend with Automatic Parallelization”. In: *Lincoln Laboratory Journal* 17.1 (Nov. 2007), pp. 187–198. ISSN: 0018-9162. URL: https://www.ll.mit.edu/publications/journal/pdf/vol17_no1/17_1_10Bliss.pdf.

[7] William Blume and Rudolf Eigenmann. “Demand-Driven, Symbolic Range Propagation”. In: *LCPC ’95: Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*. London, UK: Springer-Verlag, 1996, pp. 141–160. ISBN: 3-540-60765-X.

[8] William Blume and Rudolf Eigenmann. “Nonlinear and Symbolic Data Dependence Testing”. In: *IEEE Trans. Parallel Distrib. Syst.* 9.12 (Dec. 1998), pp. 1180–1194. ISSN: 1045-9219. DOI: 10.1109/71.737695. URL: http://dx.doi.org/10.1109/71.737695.

[9] William Blume et al. “Polaris: Improving the effectiveness of parallelizing compilers”. In: *Languages and Compilers for Parallel Computing*. Ed. by Keshav Pingali et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 141–154. ISBN: 978-3-540-49134-7.

[10] OpenMP Architecture Review Board. *The OpenMP API specification for parallel programming*. Oct. 2014. URL: http://www.openmp.org/.

[11] Rastislav Bodík, Rajiv Gupta and Vivek Sarkar. “ABCD: eliminating array bounds checks on demand”. In: *PLDI ’00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. Vancouver, British Columbia, Canada: ACM Press, 2000, pp. 321–333. ISBN: 1-58113-199-2. DOI: http://doi.acm.org/10.1145/349299.349342.

[12] J. M. Bull et al. “A benchmark suite for high performance Java”. In: *Concurrency: Practice and Experience* 12.6 (May 2000), pp. 375–388.

[13] Michael G. Burke et al. "The Jalapeño dynamic optimizing compiler for Java". In: *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*. San Francisco, California, United States: ACM, 1999, pp. 129–141. ISBN: 1-58113-161-5. DOI: `http://doi.acm.org/10.1145/304065.304113`.

[14] Microsoft News Center. *The Big Bang: How the Big Data Explosion Is Changing the World*. Feb. 2013. URL: `http://news.microsoft.com/2013/02/11/the-big-bang-how-the-big-data-explosion-is-changing-the-world/`.

[15] Guangyu Chen and Mahmut Kandemir. "Verifiable Annotations for Embedded Java Environments". In: *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES '05. San Francisco, California, USA: ACM, 2005, pp. 105–114. ISBN: 1-59593-149-X. DOI: `10.1145/1086297.1086312`. URL: `http://doi.acm.org/10.1145/1086297.1086312`.

[16] D. COOPER. "Theorem Proving in Arithmetic without Multiplication". In: *Machine Intelligence* (1972). URL: `https://ci.nii.ac.jp/naid/10022285772/en/`.

[17] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[18] Intel Corporation. *Cilk Plus*. [Online; accessed 30-December-2015]. 2015. URL: `https://www.cilkplus.org/`.

[19] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. Los Angeles, California: ACM, 1977, pp. 238–252. DOI: `http://doi.acm.org/10.1145/512950.512973`.

[20] Patrick Cousot and Nicolas Halbwachs. "Automatic discovery of linear restraints among variables of a program". In: *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. Tucson, Arizona: ACM Press, 1978, pp. 84–96. DOI: `http://doi.acm.org/10.1145/512760.512770`.

[21] Pierre Crgut. *Omega: a solver for quantifier-free problems in Presburger Arithmetic*. URL: `https://coq.inria.fr/refman/addendum/omega.html`.

[22] Ron Cytron et al. "Efficiently computing static single assignment form and the control dependence graph". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.4 (1991), pp. 451–490. ISSN: 0164-0925. DOI: `http://doi.acm.org/10.1145/115372.115320`.

[23] G Dantzig, R Fulkerson and S Johnson. "Solution of a large-scale traveling-salesman problem". In: *Operations Research* 2 (1954), pp. 393–410.

[24] George B. Dantzig. "A History of Scientific Computing". In: ed. by Stephen G. Nash. New York, NY, USA: ACM, 1990. Chap. Origins of the Simplex Method, pp. 141–151. ISBN: 0-201-50814-1. DOI: `10.1145/87252.88081`. URL: `http://doi.acm.org/10.1145/87252.88081`.

[25] George B. Dantzig. *Linear programming and extensions*. Rand Corporation Research Study. Princeton, NJ: Princeton Univ. Press, 1963. XVI, 625. URL: `http://gso.gbv.de/DB=2.1/CMD?ACT=SRCHA&SRT=YOP&IKT=1016&TRM=ppn+180926950&sourceid=fbw_bibsonomy`.

[26] George B. Dantzig and B. Curtis Eaves. "Fourier-Motzkin Elimination and Its Dual". In: *J. Comb. Theory, Ser. A* 14.3 (1973), pp. 288–297.

[27] Martin Davis, George Logemann and Donald Loveland. "A Machine Program for Theorem-proving". In: *Commun. ACM* 5.7 (July 1962), pp. 394–397. ISSN: 0001-0782. DOI: `10.1145/368273.368557`. URL: `http://doi.acm.org/10.1145/368273.368557`.

[28] Leonardo De Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS’08/ETAPS’08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. ISBN: 3-540-78799-2, 978-3-540-78799-0. URL: `http://dl.acm.org/citation.cfm?id=1792734.1792766`.

[29] Robert H. Dennard et al. “Design of ion-implanted MOSFETs with very small physical dimensions”. In: *IEEE J. Solid-State Circuits* (1974), p. 256.

[30] Yann Disser and Martin Skutella. “The Simplex Algorithm Is NP-Mighty”. In: *ACM Trans. Algorithms* 15.1 (Nov. 2018), 5:1–5:19. ISSN: 1549-6325. DOI: `10.1145/3280847`. URL: `http://doi.acm.org/10.1145/3280847`.

[31] Nurit Dor, Michael Rodeh and Mooly Sagiv. “CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C”. In: *SIGPLAN Not.* 38.5 (May 2003), pp. 155–167. ISSN: 0362-1340. DOI: `10.1145/780822.781149`. URL: `http://doi.acm.org/10.1145/780822.781149`.

[32] Robert A. van Engelen et al. “A Unified Framework for Nonlinear Dependence Testing and Symbolic Analysis”. In: *Proceedings of the 18th Annual International Conference on Supercomputing*. ICS ’04. Malo, France: ACM, 2004, pp. 106–115. ISBN: 1-58113-839-3. DOI: `10.1145/1006209.1006226`. URL: `http://doi.acm.org/10.1145/1006209.1006226`.

[33] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. June 2015. URL: `http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`.

[34] Apache Software Foundation. *hadoop*. Sept. 2014. URL: `http://hadoop.apache.org/`.

[35] Geoffrey Fox et al. *Solving Problems On Concurrent Processors Vol. 1: General Techniques and Regular Problems*. Vol. 3. Jan. 1988. ISBN: 0138230226.

[36] Jean Gallier. *The Frobenius Coin Problem Upper Bounds on The Frobenius Number*. URL: `https://pdfs.semanticscholar.org/9ea2/8885ab0f1b3d8f0a0475bd21e78fbafc4ec6.pdf`.

[37] Andreas Gampe et al. “Safe, multiphase bounds check elimination in Java”. In: *Softw., Pract. Exper.* 41 (2011), pp. 753–788.

[38] Andreas Gampe et al. “Speculative Improvements to Verifiable Bounds Check Elimination”. In: *Proceedings of the International Conference on Principles and Practice of Programming In Java (PPPJ 2008)*. Modena, Italy: ACM Press, 2008.

[39] Vinod Ganapathy et al. “Buffer Overrun Detection Using Linear Programming and Static Analysis”. In: *Proceedings of the 10th ACM Conference on Computer and Communications Security*. CCS ’03. Washington D.C., USA: ACM, 2003, pp. 345–354. ISBN: 1-58113-738-9. DOI: `10.1145/948109.948155`. URL: `http://doi.acm.org/10.1145/948109.948155`.

[40] Gina Goff et al. “Practical Dependence Testing”. In: 1991, pp. 15–29.

[41] R. Gomory. “Outline of an Algorithm for Integer Solutions to Linear Programs”. In: *Bulletin of the American Mathematical Society* 64 (1958), pp. 275–278.

[42] Khronos Group. *The OpenCL Specification*. [Online; accessed 13-February-2018]. 2015. URL: `https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.html`.

[43] Mohammad Reza Haghighat. “Symbolic analysis for parallelizing compilers”. PhD thesis. Champaign, IL, USA: University of Illinois at Urbana-Champaign, 1995.

[44] Michael Hansen. *Presburger Arithmetic: Coopers algorithm*. URL: `http://www2.imm.dtu.dk/courses/02917/Presburger1.pdf`.

[45] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 6th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 978-0123838728.

[46] Martin Hilbert and Priscila López. “The World’s Technological Capacity to Store, Communicate, and Compute Information”. In: *Science* 332.6025 (Apr. 2011), pp. 60–65. ISSN: 1095-9203. DOI: `10.1126/science.1200970`.

[47] Michael Hind. “Pointer analysis: Havent we solved this problem yet?” In: *PASTE’01*. ACM Press, 2001, pp. 54–61.

[48] Richard Kaye. *Models of Peano Arithmetic*. Clarendon Press, 1991.

[49] Georg. Kreisel and J. L. Krivine. *Elements of mathematical logic. (Model theory) [By] G. Kreisel and J. L. Krivine*. English. North Holland Pub. Co Amsterdam, 1967, xi, 222 p.

[50] Oak Ridge National Laboratory. *PVM: Parallel Virtual Machine*. Dec. 2011. URL: `http://www.csm.ornl.gov/pvm/`.

[51] E. L. Lawler and D. E. Wood. “Branch-and-Bound Methods: A Survey”. In: *Oper. Res.* 14.4 (Aug. 1966), pp. 699–719. ISSN: 0030-364X. DOI: `10.1287/opre.14.4.699`. URL: `http://dx.doi.org/10.1287/opre.14.4.699`.

[52] John Leonard. *The stealthy rise of functional programming*. [Online; accessed 01-March-2018]. 2017. URL: `https://www.computing.co.uk/ctg/analysis/3003123/the-slow-but-steady-rise-of-functional-programming`.

[53] Z. Li, P. . Yew and C. . Zhu. “An efficient data dependence analysis for parallelizing compilers”. In: *IEEE Transactions on Parallel and Distributed Systems* 1.1 (1990), pp. 26–34. ISSN: 1045-9219. DOI: `10.1109/71.80122`.

[54] Joanne L. Martin. “Supercomputer Performance Evaluation: The PERFECT Benchmarks”. In: *Supercomputing*. Ed. by Janusz S. Kowalik. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 239–248. ISBN: 978-3-642-75771-6.

[55] *Mathematical Questions with Their Solutions: From the ”Educational Times”*. v. 40-42. 1884. URL: `https://books.google.co.nz/books?id=XvgJAAAAIAAJ`.

[56] D. E. Maydan, J. L. Hennessy and M. S. Lam. “Effectiveness of data dependence analysis”. In: *In Proceedings of the NSF-NCRD Workshop on Advanced Compilation Techniques for Novel Architectures* (1991).

[57] Dror E. Maydan, John L. Hennessy and Monica S. Lam. “Efficient and exact data dependence analysis”. In: *SIGPLAN Not.* 26 (6 1991), pp. 1–14. ISSN: 0362-1340. DOI: `http://doi.acm.org/10.1145/113446.113447`. URL: `http://doi.acm.org/10.1145/113446.113447`.

[58] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* Corvallis, OR, USA: kernel.org, 2010. URL: `http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html`.

[59] Vijay S. Menon et al. “A verifiable SSA program representation for aggressive compiler optimization”. In: *POPL ’06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Charleston, South Carolina, USA: ACM Press, 2006, pp. 397–408. ISBN: 1-59593-027-2. DOI: `http://doi.acm.org/10.1145/1111037.1111072`.

[60] Gordon E. Moore. “Cramming more components onto integrated circuits”. In: *Electronics* 38.8 (1965).

[61] José; E. Moreira, Samuel P. Midkiff and Manish Gupta. “From flop to megaflops: Java for technical computing”. In: *ACM Trans. Program. Lang. Syst.* 22.2 (2000), pp. 265–295. ISSN: 0164-0925. DOI: `http://doi.acm.org/10.1145/349214.349222`.

[62] John Morris, Kyuho Lee and Junseong Kim. “Cilk versus MPI: Comparing Two Parallel Programming Styles on Heterogeneous Systems”. In: *High-Performance Computing*. John Wiley & Sons, Inc., 2006, pp. 81–97. ISBN: 9780471732716. DOI: `10.1002/0471732710.ch5`. URL: `http://dx.doi.org/10.1002/0471732710.ch5`.

[63] Michał Moskal. “Rocket-Fast Proof Checking for SMT Solvers”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 486–500. ISBN: 978-3-540-78800-3.

[64] Leonardo de Moura and Nikolaj Bjørner. “Satisfiability Modulo Theories: An Appetizer”. In: *Formal Methods: Foundations and Applications*. Ed. by Marcel Vinícius Medeiros Oliveira and Jim Woodcock. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 23–36. ISBN: 978-3-642-10452-7.

[65] Leonardo de Moura, Bruno Dutertre and Natarajan Shankar. “A Tutorial on Satisfiability Modulo Theories”. In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. Vol. 4590. Lecture Notes in Computer Science. Springer, 2007, pp. 20–36.

[66] Katta G. Murty. *Linear Programming*. John Wiley & Sons, 1983. ISBN: 047109725X. URL: `https://www.amazon.com/Linear-Programming-Katta-G-Murty/dp/047109725X?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=047109725X`.

[67] Dan Nagle. “MPI – The Complete Reference, Vol. 1, The MPI Core, 2Nd Ed., Scientific and Engineering Computation Series, by Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker and Jack Dongarra”. In: *Sci. Program.* 13.1 (Jan. 2005), pp. 57–63. ISSN: 1058-9244. DOI: `10.1155/2005/653765`. URL: `http://dx.doi.org/10.1155/2005/653765`.

[68] David Niedzielski, Kleanthis Psarris and Theoharis Theoharis. “An Analytical Evaluation of Data Dependence Analysis Techniques”. In: *International Journal of Parallel Programming* (2018). ISSN: 1573-7640. DOI: `10.1007/s10766-018-0577-7`. URL: `https://doi.org/10.1007/s10766-018-0577-7`.

[69] David Niedzielski et al. “A Verifiable, Control Flow Aware Constraint Analyzer for Bounds Check Elimination”. In: *Static Analysis*. Ed. by Jens Palsberg and Zhendong Su. Vol. 5673. Lecture Notes in Computer Science. 10.1007/978-3-642-03237-0-11. Springer Berlin / Heidelberg, 2009, pp. 137–153. URL: `http://dx.doi.org/10.1007/978-3-642-03237-0-11`.

[70] Michael Norrish. *Deciding Presburger Arithmetic*. URL: `http://ssll.cecs.anu.edu.au/files/slides/norrish.pdf`.

[71] *OpenACC: Directives for Accelerators*. URL: `http://www.openacc-standard.org`.

[72] Oracle. *Java Virtual Machine Specification. Chapter 4: the class file format*. URL: `https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html`.

[73] Ian Paul. *The end of Moore’s Law is on the horizon, AMD says*. Apr. 2013. URL: `http://www.pcworld.com/article/2032913/the-end-of-moores-law-is-on-the-horizon-says-amd.html`.

[74] P. M. Petersen and D. A. Padua. “Static and dynamic evaluation of data dependence analysis techniques”. In: *IEEE Transactions on Parallel and Distributed Systems* 7.11 (1996), pp. 1121–1132. ISSN: 1045-9219. DOI: 10.1109/71.544354.

[75] Michael Philippsen et al. “JavaGrande— High Performance Computing with Java”. In: *Applied Parallel Computing. New Paradigms for HPC in Industry and Academia*. Ed. by Tor Sørevik et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 20–36.

[76] The Open MPI Project. *Open MPI: Open Source High Performance Computing*. Oct. 2014. URL: http://www.open-mpi.org/.

[77] K. Psarris and K. Kyriakopoulos. “Data dependence testing in practice”. In: *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425)*. 1999, pp. 264–273. DOI: 10.1109/PACT.1999.807571.

[78] Kleanthis Psarris. “On exact data dependence analysis”. In: *ICS ’92: Proceedings of the 6th international conference on Supercomputing*. Washington, D. C., United States: ACM Press, 1992, pp. 303–312. ISBN: 0-89791-485-6. DOI: http://doi.acm.org/10.1145/143369.143424.

[79] Kleanthis Psarris, Xiangyun Kong and David Klappholz. “Extending the I test to direction vectors”. In: *ICS ’91: Proceedings of the 5th international conference on Supercomputing*. Cologne, West Germany: ACM Press, 1991, pp. 330–340. ISBN: 0-89791-434-1. DOI: http://doi.acm.org/10.1145/109025.109106.

[80] Kleanthis Psarris and Konstantinos Kyriakopoulos. “An Experimental Evaluation of Data Dependence Analysis Techniques”. In: *IEEE Transactions on Parallel and Distributed Systems* 15.3 (2004), pp. 196–213. ISSN: 1045-9219. DOI: http://doi.ieeecomputersociety.org/10.1109/TPDS.2004.76.

[81] William Pugh. “A practical algorithm for exact array dependence analysis”. In: *Commun. ACM* 35.8 (1992), pp. 102–114. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/135226.135233.

[82] Feng Qian, Laurie J. Hendren and Clark Verbrugge. “A Comprehensive Approach to Array Bounds Check Elimination for Java”. In: *CC ’02: Proceedings of the 11th International Conference on Compiler Construction*. London, UK: Springer-Verlag, 2002, pp. 325–342. ISBN: 3-540-43369-4.

[83] Lawrence Rauchwerger and David Padua. “The LRPD Test: Speculative Run-time Parallelization of Loops with Privatization and Reduction Parallelization”. In: *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*. PLDI ’95. La Jolla, California, USA: ACM, 1995, pp. 218–232. ISBN: 0-89791-697-2. DOI: 10.1145/207110.207148. URL: http://doi.acm.org/10.1145/207110.207148.

[84] Intel Research and Development. “Architecting the Era of Tera”. In: (Feb. 2004). URL: https://software.intel.com/sites/default/files/27/61/Tera_Era.pdf.

[85] Jeffery von Ronne et al. “Safe Bounds Check Annotations”. In: *Concurrency and Computations: Practice and Experience* 21.1 (2009). DOI: 10.1002/cpe.1341, pp. 41–57. ISSN: 1057-4514.

[86] Radu Rugina and Martin Rinard. “Automatic Parallelization of Divide and Conquer Algorithms”. In: *SIGPLAN Not.* 34.8 (May 1999), pp. 72–83. ISSN: 0362-1340. DOI: 10.1145/329366.301111. URL: http://doi.acm.org/10.1145/329366.301111.

[87] Alexander Schrijver. *Theory of Linear and Integer Programming*. New York, NY, USA: John Wiley & Sons, Inc., 1986. ISBN: 0-471-90854-1.

[88] Robert E. Shostak. “On the SUP-INF Method for Proving Presburger Formulas”. In: *J. ACM* 24.4 (Oct. 1977), pp. 529–543. ISSN: 0004-5411. DOI: 10.1145/322033.322034. URL: http://doi.acm.org/10.1145/322033.322034.

[89] Ryan Stansifer. “Presburgers Article on Integer Airthmetic: Remarks and Translation”. In: TR84-639 (1984). URL: http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cs/TR84-639.

[90] James Joseph Sylvester. “On the Partition of Numbers”. In: *Quarterly Journal of Mathematics* 1.1 (1884), pp. 141–152.

[91] Massachusetts Institute of Technology. *The Cilk Project*. [Online; accessed 30-December-2015]. 2015. URL: http://supertech.csail.mit.edu/cilk/.

[92] *The Coq Proof Assistent*. URL: https://coq.inria.fr.

[93] *The Omega Library Version 1.00 Interface Guide*. URL: http://www.cs.umd.edu/projects/omega/interface_doc/interface.html.

[94] Unknown. *Why physicists Still Use Fortran*. [Online; accessed 13-February-2018]. 2015. URL: http://moreisdifferent.com/2015/07/16/why-physicsts-still-use-fortran/.

[95] Wikipedia. *MapReduce — Wikipedia, The Free Encyclopedia*. [Online; accessed 29-October-2014]. 2014. URL: http://en.wikipedia.org/w/index.php?title=MapReduce&oldid=630978092.

[96] Wikipedia. *MOSFET — Wikipedia, The Free Encyclopedia*. [Online; accessed 25-October-2014]. 2014. URL: http://en.wikipedia.org/w/index.php?title=MOSFET&oldid=629735282.

[97] Wikipedia contributors. *Ferdinand Georg Frobenius — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Ferdinand_Georg_Frobenius&oldid=823514346. [Online; accessed 2-August-2018]. 2018.

[98] Wikipedia contributors. *Simplex algorithm — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-February-2019]. 2019. URL: https://en.wikipedia.org/w/index.php?title=Simplex_algorithm&oldid=882327520.

[99] Robert P. Wilson et al. “SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers”. In: *SIGPLAN Not.* 29.12 (Dec. 1994), pp. 31–37. ISSN: 0362-1340. DOI: 10.1145/193209.193217. URL: http://doi.acm.org/10.1145/193209.193217.

[100] Niklaus Wirth. “A Plea for Lean Software”. In: *Computer* 28.2 (Feb. 1995), pp. 64–68. ISSN: 0018-9162. DOI: 10.1109/2.348001. URL: http://dx.doi.org/10.1109/2.348001.

[101] M. J. Wolfe and C. Tseng. “The Power test for data dependence”. In: *IEEE Transactions on Parallel and Distributed Systems* 5.3 (1992), pp. 591–601.

[102] Michael Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, 1996. ISBN: 0-8053-2730-4.

[103] Thomas Würthinger, Christian Wimmer and Hanspeter Mössenböck. “Array bounds check elimination for the Java HotSpot client compiler”. In: *PPPJ ’07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*. Lisboa, Portugal: ACM, 2007, pp. 125–133. ISBN: 978-1-59593-672-1. DOI: http://doi.acm.org/10.1145/1294325.1294343.

[104] Hongwei Xi and Frank Pfenning. “Eliminating Array Bound Checking Through Dependent Types”. In: *SIGPLAN Not.* 33.5 (May 1998), pp. 249–257. ISSN: 0362-1340. DOI: 10.1145/277652.277732. URL: http://doi.acm.org/10.1145/277652.277732.

[105] Jisheng Zhao et al. “Loop Parallelisation for the Jikes RVM”. In: *Proceedings of the Sixth International Conference on Parallel and Distributed Computing (PDCAT’05)*. IEEE Computer Society, 2005, pp. 35–39. DOI: 10.1109/PDCAT.2005.164.