A Concurrent IFDS Dataflow Analysis Algorithm Using Actors

by

Jonathan David Rodriguez

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2010

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

There has recently been a resurgence in interest in techniques for effective programming of multi-core computers. Most programmers find general-purpose concurrent programming to be extremely difficult. This difficulty severely limits the number of applications that currently benefit from multi-core computers. There already exist many concurrent solutions for the class of regular applications, which include various algorithms for linear algebra. For the class of irregular applications, which operate on dynamic and pointer- and graph-based structures, efficient concurrent solutions have so far remained elusive. Dataflow analysis applications, which are often found in compilers and program analysis tools, have received particularly little attention with regard to execution on multi-core machines. Operating on the theory that the Actor model, which structures computations as systems of asynchronously-communicating entities, is a more appropriate method for representing irregular algorithms than the shared-memory model, this work presents a concurrent Actor-based formulation of the IFDS, or Interprocedural Finite Distributive Subset, dataflow analysis algorithm. The implementation of this algorithm is done using the Scala language and its Actors library. This algorithm achieves significant speedup on multi-core machines without using any optimistic execution. This work contributes to Actor research by showing how the Actor model can be practically applied to a dataflow analysis problem. This work contributes to static analysis research by showing how a dataflow analysis algorithm can effectively make use of multi-core machines, allowing the possibility of faster and more precise analyses.

Acknowledgements

I wish to recognize the following people:

Ondřej Lhoták, who had high expectations for this work and endured all manner of half-baked ideas before I figured out what I was doing.

Barbie Rodriguez, who helped cheer me up when the work got difficult.

Tarek Chammah, who has an inexhaustible ability to discuss anything of academic interest.

Shaun Harvey, who endeavoured to fix my problems, to keep me entertained, and has an inexhaustible ability to discuss anything, period.

Krzysztof Borowski, who endeavoured to be my friend.

Nomair Naeem, who helped write the code this work depends on.

Peter Buhr, who knows more about control flow than anyone else I have met to date.

Patrick Lam, who first introduced me to the field of static analysis.

Tim Brecht, who has a contagious excitement about multi-core computing.

Dad and Mom, who kept telling me I needed to go to graduate school.

Tobiah Rodriguez, who reminded me that life is more than graduate school.

Dedication

My intention in this work has been, as Albert Einstein once put it, to know the mind of God with regard to a particular matter. There is something divine about the process of seeking a timely and well-balanced solution to a particular problem, and of recombining old ideas to yield new possibilities.

This work is dedicated to the hunt – to the knowledge that there can be better solutions than the ones we already have.

Contents

List of Algorithms **xv**

List of Figures **xviii**

List of Tables **xix**

1 Introduction **1**

2 Related Work **5**

2.1 Concurrent Programming Difficulties 5

2.2 The Actor Model 7

2.2.1 Related Computational Models 9

2.3 The IFDS Algorithm 10

2.4 Studies of Irregular Applications 11

2.5 Previous Approaches to Concurrency 11

2.5.1 Fully Automatic Solutions 12

2.5.2 Semi-Automatic Solutions 12

2.5.3 Solutions for Preserving Determinism 14

2.5.4 Software Transactional Memory 15

2.5.5 Language Solutions 17

3 Background **19**

3.1 The Actor Model 19

3.1.1 Defining Actor Classes 20

3.1.2 Formal Properties of the Actor Model 22

3.2 The IFDS Algorithm 22

4 The E-IFDS Algorithm **31**

4.1 Demand Construction of the Exploded Supergraph 33

4.2 Demand Construction of Summary Edges 34

4.3 Caller Context for Return-Flow Functions 34

4.4 Multiple Called Procedures Per Call Site 36

4.5 Redundant Fact Removal 36

4.6 Additional Presentation Notes 37

5 The IFDS-A/AD Algorithms **39**

5.1 IFDS-A Node-Actor Classes 41

5.2 Redundant Fact Removal 45

5.3 Using Detach for Increased Concurrency 46

6 Evaluation **51**

6.1 The Variable Type Analysis 51

6.2 Implementation Details 51

6.3 Testing Methodology 53

6.3.1 Modeling Ideal Performance 53

6.3.2 Performance Assessment 54

6.3.3 Data Collection and Handling 55

6.4 Results 56

6.4.1 Input Characteristics 56

6.4.2 Ideal Performance 59

6.4.3 Performance Assessment 62

7 Conclusions **69**

A Raw Performance Data **71**

References **75**

List of Algorithms

3.1 Original IFDS Algorithm reproduced from Reps et al. [RHS95] 27
4.1 The E-IFDS Algorithm 32
4.2 E-IFDS Propagate with Redundant Fact Removal 37
5.1 The Top-Level IFDS-A Algorithm 40
5.2 IFDS-A Node-Actor Classes 42
5.3 IFDS-A Redundant Fact Removal 45
5.4 IFDS-A Redundant Fact Removal on Callee-Path-Edges 46
5.5 IFDS-AD Call-Site Node-Actor Class 48
5.6 IFDS-AD Procedure-Exit Node-Actor Class 49
5.7 IFDS-AD Intraprocedural Node-Actor Class 49

List of Figures

3.1 The Actor Abstraction 20

3.2 Actor Class Definition 21

3.3 IFDS Supergraph for One Procedure (G^*) 23

3.4 IFDS Supergraph for a Procedure Call (G^*) 24

3.5 IFDS Exploded Supergraph for One Procedure (G^* and corresponding $G^\sharp$) 25

3.6 IFDS Summary Edge Generation (G^* and corresponding $G^\sharp$) 28

4.1 Caller Context Example (Java Syntax) 35

5.1 Detach Example 47

5.2 Meaning of Detach 47

6.1 Node-Actor Reaction Loop in Scala 52

6.2 Top-Level IFDS-A Solver Code 53

6.3 Available-Parallelism Charts 58

6.4 Ideal Speedup Model (Opteron8) 60

6.5 Ideal Speedup Model (Sparc64) 60

6.6 IFDS-A Self-Speedup (Opteron8) 61

6.7 IFDS-A Self-Speedup (Sparc64) 61

6.8 IFDS-A Speedup vs. E-IFDS (Opteron8) 63

6.9 IFDS-A Speedup vs. E-IFDS (Sparc64) 63

6.10 IFDS-A Computational Efficiency (Opteron8) 64

6.11 IFDS-A Computational Efficiency (Sparc64) 64

6.12 IFDS-A / IFDS-AD Speedup (Opteron8) 66

6.13 IFDS-A / IFDS-AD Speedup (Sparc64) 66

List of Tables

6.1 Input Characteristics 56
6.2 Parallelism Characteristics 57
6.3 IFDS-A Overhead and Efficiency Summary 65

A.1 Antlr on Opteron8 72
A.2 Jython on Opteron8 72
A.3 Luindex on Opteron8 73
A.4 Antlr on Sparc64 73
A.5 Jython on Sparc64 74
A.6 Luindex on Sparc64 74

Chapter 1

Introduction

For several decades, researchers have supposed that the development of multi-core computer architectures would eventually become necessary. It has only been in the last few years, however, that certain physical realities have prevented the continued exponential acceleration of single-core processing speeds. Power consumption and heat dissipation, in particular, constitute major barriers to increasing the speed of single-core machines. Since then, there has been intense industry interest in producing multi-core architectures for both specialty applications and for the mass market.

Unfortunately, learning to effectively program multi-core architectures has been difficult. The essential issues involved in multi-core programming are two-fold. First, partitioning a problem into independently executable tasks can be difficult for certain types of computations. Second, correctly handling data dependencies among tasks is essential for correct program operation. The first issue is largely one of performance; an improper distribution of work among cores can result in poor execution performance. The second issue is of greater concern because improper handling of dependencies can result in generation of incorrect results or the production of subtle concurrency bugs.

Most common multi-core architectures offer a shared-memory abstraction. This shared-memory model offers a common address space to all tasks, and it is up to the individual tasks to not corrupt each other's data. Using locks to enforce mutual exclusion is probably the most common way of ensuring that common data structures are not corrupted. In practice, however, lock-based shared-memory programs can be extremely difficult to construct correctly, to reason about, and to debug.

In response to the need for a better model of concurrent computation, the Actor model was developed. The Actor model expresses computations as a set of independent entities which communicate by passing messages. There is no assumption of either a shared memory or a global address space in the Actor model; each task proceeds using its own local data only. Although the Actor model has been in existence for over three decades, its level of practical significance remains an open question. Because it represents a significant departure from the shared-memory model, there are currently very few real programming projects that use an Actor-based structure to perform computation.

The main question addressed by this work is whether or not an Actor-based approach can be used to efficiently solve a concurrency problem that is not easily solved through conventional techniques.

The easiest type of concurrency problem is arguably the "embarrassingly parallel" type. Embarrassingly parallel problems exhibit no communication among tasks except at task completion. Once started, a set of embarrassingly parallel tasks can run to completion without any blocking or shared-memory interactions.

Regular problems are a superset of embarrassingly parallel problems. Regular problems include many types of numeric computation and simulation problems (e.g. those involving linear algebra), which exhibit very predictable data and control-flow dependencies. As a result, it is often possible to statically discover data parallelism within regular problems. Much work has been done in this domain, and efficient solutions for many types of regular problems are known.

A more difficult class of concurrency problems is the class of *irregular problems*. Irregular problems typically involve construction or traversal of pointer-based structures such as trees and graphs. Because the dependencies among data are often not predictable, static discovery of data parallelism is impossible for many irregular problems. For this reason, very few irregular algorithms have been implemented in a concurrent fashion.

The class of Interprocedural Finite Distributive Subset problems, or IFDS problems, are a class of dataflow-analysis problems that are irregular in nature. This work presents a concurrent formulation of the IFDS solver algorithm based on the Actor model. This work is the first known concurrent formulation of the IFDS algorithm.

The algorithm is implemented in the Scala language using Scala's Actors library. Notable properties of this implementation include:

- Significant multi-core speedups despite small work unit size. The average work unit execution time for the tested inputs varied from a few microseconds to a few tens of microseconds on commodity hardware.
- Significant multi-core speedups without using any form of optimistic execution.

This work makes the following contributions:

- A case study of the application of the Actors model to a concurrency problem that is considered difficult to solve using conventional methods. This case study helps determine the practical importance of the model.

- The presentation of a concurrent IFDS algorithm. IFDS and other dataflow analysis algorithms are used in many compilers and program-analysis tools. As an example of a concurrent dataflow analysis algorithm that is shown to provide substantial speedups on multi-core architectures, this work offers insights that may help increase the speed and precision of future compilers and analysis tools.

- The presentation of a set of extensions to the sequential IFDS algorithm that increase its speed and practical utility for many types of analyses.

- The introduction of the **detach** construct, a novel construct designed to help increase concurrency in practical actor-based programs.

- An experimental methodology designed to discover the real efficiency of a concurrent algorithm implementation even in the presence of background processes and shared machine resources.

Chapter 2 contains a survey of work related to the Actor model, to the IFDS algorithm, and to various approaches to the problem of concurrency as it relates to irregular algorithms.

Chapter 3 contains background material on the Actor model and on the IFDS algorithm.

Chapter 4 presents the Extended IFDS algorithm, which contains certain modifications designed to make the IFDS algorithm more useful in practice.

Chapter 5 presents the concurrent IFDS-Actors algorithm, or IFDS-A. This chapter also introduces the **detach** construct, and presents a version of IFDS-A, called IFDS-AD, that uses **detach** to obtain increased concurrency.

Chapter 6 discusses and evaluates implementations of Extended IFDS, IFDS-A, and IFDS-AD. The relative performance, self-speedup, and computational efficiency of these implementations are evaluated on two different hardware architectures.

Chapter 2

Related Work

Many different approaches have been taken in the attempt to allieviate the difficulty of concurrent programming. A large body of existing work is dedicated to the issue of performance in *regular applications* such as scientific computing and other numeric code, but this work is not examined in this thesis. The work surveyed here is targeted at a more difficult class of applications, the class of *irregular applications*. Unlike regular applications, irregular applications typically involve operations over trees, graphs, or other pointer-based structures, which makes their dynamic behaviour very difficult to predict. As a result, efficient parallel execution of irregular applications necessitates a different approach to parallelism.

2.1 Concurrent Programming Difficulties

Current practice in concurrent programming generally uses threading libraries to set up multiple concurrent execution paths through sequential object-oriented code. The following papers discuss why explicitly-parallel code with shared memory is difficult to produce, and what may be missing in the current approach to concurrent programming.

The Intel Concurrent Collections project has identified three fundamental barriers in automatically converting sequential code into concurrent code [Kno09]. First, sequential code requires a sequential ordering to be specified between operations, even if that ordering is arbitrary. It can be very difficult for a compiler to reverse this process and uncover the real sequential dependencies in the program. Second, imperative sequential code expresses data access in terms of location-based variables, not values. Variables can be overwritten, which over-constrains possible operation orderings. Third, sequential code does not distinguish the question of if an operation is executed from the question of when it is executed.

In sequential code, the answer to the "when" question is always "now," again enforcing an arbitrary ordering of operations that can be difficult to undo.

Edward Lee has built a case that the threads model of concurrency, although apparently a straightforward extension of the sequential programming model, actually discards the most desirable properties of sequential code [Lee06]. In contrast to the sequential model, the threads model is "wildly non-deterministic" and therefore extremely difficult to reason about. It is nearly impossible to find and correct all concurrency errors in even trivial programs using standard testing techniques, and subtle concurrency errors are likely to exist in nearly all concurrent programs. Lee states that the real world has many examples of concurrent physical dynamics that are in fact very natural for humans to reason about, and therefore it is not concurrency itself that is difficult to understand. It is only the threads model itself that is difficult for humans to comprehend. Lee hypothesized that the solution to this issue is to always achieve deterministic ends using deterministic means, and only introduce non-determinism explicitly and where needed.

Lynn Andrea Stein argues that a new computational metaphor is required to make progress in the field [Ste99]. The choice of metaphor has far-reaching implications because it shapes the understanding and design of systems. Certain problems that are very difficult to comprehend under one metaphor can be easy to comprehend under another, and vice versa. The traditional computational metaphor is "computation-as-calculation," which views computation as an immutable process that starts with a set of inputs and terminates with some result. The inputs are the "problem" and the result is the "solution." The intermediate steps are only considered important insofar as they produce the desired end result. While the computation-as-calculation metaphor has been very useful in the past, it has now become a problem because it cannot effectively describe the way current computer systems are being used. Internet services, embedded systems and artificial intelligence systems are all poorly described by this metaphor. Stein suggests that a more appropriate metaphor is "computation-as-interaction" – the purpose of the computation is to produce behaviours that emerge from interactions among components. That is, "to replace the conventional metaphor *a sequence of steps* with the notion of a community of interacting entities." Under this latter metaphor, the most important thing about a computation is not the answer that it produces, but the behavioural invariants it exhibits.

Tim Sweeney of Epic Games highlighted three areas that are sorely lacking in today's mainstream programming languages [Swe06]. First, better abstraction facilities are needed to increase the flexibility and modularization of application components. Second, stronger and more flexible type systems are required to reduce run-time failures. Third, pervasive support for both implicit and explicit concurrency is required to overcome performance challenges and programmer productivity challenges. Software transactional memory and automatic parallelization of pure functions were suggested as means to increase concurrency. Sweeney remarked on the strong apparent relationship between language features that provide increased reliability and language features that enable concurrency. In particular, dependent types are able to remove sequencing constraints that would have been imposed by null pointer exceptions and array bounds violation exceptions.

2.2 The Actor Model

The Actor model was originally developed to represent computations where the code and data are distributed among some number of independent entities. Communication in the Actor model occurs exclusively through asynchronous message passing. The model does not include a built-in concept of either shared memory or a global clock – each actor is only aware of its own local state and its neighbouring actors. Section 3.1 provides more extensive discussion of the Actor model.

Events in any distributed system, including Actor-based systems, are partially ordered with respect to a global clock [Lam78]. Given two events in such a system, it is sometimes impossible to tell which event occurred "first." Lamport described an algorithm for using logical clocks to impose a total ordering on events, and showed how it could be used to solve a simple synchronization problem. This algorithm guarantees a maximum upper bound on the total skew that logical clocks may experience relative to each other. However, such a total ordering is arbitrary, and may disagree with an observer's perception of time. Lamport stresses that the ordering of events in a multi-process system is unavoidably a partial ordering.

The Actor model allows the control structure of a program to emerge as a result of message passing patterns among objects [Hew77]. The proposed programming methodology consists of the following activities: defining the types of actors present in the system, defining the types of messages each type of actor receives, and defining what each actor does in response to each type of message.

The Actor model is based on an *Actor theory* that defines certain laws followed by Communicating Parallel Processes, or CPP [HB77]. This theory is motivated by the need to model concurrent computations in a useful way without resorting to hardware or operating system primitives that lack precisely-defined semantics. In the Actor model, computations are not defined in terms of hardware primitives or functional reductions, but as partial orders of events. The model encourages computational expression in relativistic terms, replacing global notions of state, time, and name spaces with local notions. Furthermore, the model supports a dynamic communication topology allowing dynamic creation of new actors and dynamic re-arrangement of communication paths. Because actor-based computations are described directly as partial orderings of events, these orderings can be used to reduce the complexity of many correctness proofs.

Agha developed a formalized model of concurrency based on actors [Agh86]. The basic primitives in this model, namely behaviours, messages, and mailboxes, are able to express many higher-order constructs such as lazy evaluation. Abstraction in this model is accomplished by limiting an observer's view of a system to only those actors designated as interface actors. Non-interface actors cannot be seen by an external observer, but it is possible for a non-interface actor to obtain interface status if an existing interface actor passes its address to an external observer. This work furthermore demonstrates that dynamic detection and removal of semantic deadlock is possible within the Actor model. Deadlock in a strict syntactic sense cannot occur. Agha argues in this work that the Actor model is a relatively easy and natural way to express concurrent programs.

ActorScript is a direct implementation of the Actor model in a programming language [Hew09]. Its intent is to create a high level of performance, scalability, and expressiveness with a minimum of language primitives. Everything in the language is accomplished using message passing, including features such as futures, co-routines, and serializers. No references to threads, locks, or other low-level constructs are required, although the language implementation may expose them if desired. Message passing in the Actor model exhibits *unbounded non-determinism*, which permits message delivery to take an arbitrary length of time, and furthermore the message receipt order is undefined. However, message delivery is ultimately guaranteed.

Actors in C++, or ACT++, is a language that integrates the actor model proposed by Agha into the C++ language [Kaf90]. ACT++ implements synchronous messaging by means of futures and special-purpose reply mailboxes. ACT++ makes a distinction between active objects, or actors, which possess their own threads of control and mutual exclusion between their methods, and passive objects, which have neither their own threads of control nor mutual exclusion on their members. All shared data must belong to active objects; each passive object may only be accessible to a single active object.

The Erlang language implements an actor-like programming model [VWW96]. Designed for real-time industrial control applications, one of the first projects to use the language was a telephone exchange system. Erlang intentionally does not support higher-order functions, currying, lazy evaluation or other features in an effort to remain simple and robust. All communication in Erlang takes the form of message-passing. Since each Erlang object possesses its own logical thread of control, systems written in Erlang are intrinsically concurrent by default.

Scala is a language designed to support higher degrees of component abstraction and composition than most existing languages [OZ05]. In addition to better abstraction and composition capabilities, one of the intentions behind the design of Scala is to provide natural support for concurrent programming and web-based applications. Unlike Erlang, Scala supports higher-order functions, currying, lazy evaluation. Scala includes three language abstractions that enable higher-degree abstraction and composition. First, abstract-type members enable classes to contain both types and values as parameters. Scala distinguishes abstract classes from concrete classes, where abstract classes may contain undefined types and values, but concrete classes cannot. Second, selftype annotations allow the programmer to explicitly specify the type of `this`. Selftype annotations are necessary when the type of `this` refers to a generic type. Third, modular mixin composition is a flexible way to perform component composition. Scala introduces *traits* for this purpose. Traits may generally be used in place of abstract classes; however, traits may not have parameters. Scala's creators argue that it should be possible to transform any assembly of static program parts into a system of reusable components that contains neither static data nor hard references. Scala compiles to Java bytecode, and is able to inter-operate with Java libraries.

The standard Scala distribution provides a library that implements the Actor model [HO09]. The message-based concurrency offered by the Actor model is seen as a viable solution to both multi-core and distributed programming challenges. The Scala Actors implementation is influenced by Erlang's massage-passing model. It requires no special compiler support, relying on Scala's support for partial functions, pattern matching, and other abstraction capabilities to provide ease-of-use. The authors of the library strongly discourage usage of shared state, but the implementation does not restrict such usage. The library provides a unification of thread-based and event-based programming models. This unification prevents an inversion of control that would otherwise occur when converting a program from a threaded or stack-based form into an event-loop-based form. The authors make a claim that the Actors implementation makes concurrent programming significantly more accessible to programmers. This claim is made based on three factors. First, accessing an actor's mailbox is a race-free operation. In many cases, this can be both safer and more convenient than shared-memory locking. Second, actors are lightweight, which enables a very large number of actors to be active simultaneously without requiring the programmer to write any thread-pooling code. Third, each Java VM thread is treated like an actor, which enables inter-operation between actors and threads.

2.2.1 Related Computational Models

The Actor model was not the first model of computation to represent concurrent programs as sets of communicating entities. Particularly notable are Petri nets and the Communicating Sequential Processes model.

Petri nets are a graphical and mathematical modeling tool for describing systems that are asynchronous, concurrent and/or non-deterministic [Mur89]. A Petri net models a system as a set of places, a set of transitions, and a set of weighted arcs that connect places to transitions and transitions to places. Each place may hold some number of *tokens*. Tokens may travel from one place to another when the conditions for firing a transition are met. In particular, the number of tokens available at each of the transition's input places must be at least as large as the weight on the input arc. When a transition fires, the required tokens are removed from its input places, and then tokens are issued to its output places in amounts equal to the weights on the outgoing arcs. A transition is not required to fire; provided its conditions are met, a transition may fire at any time or not at all. One way that Petri nets can model concurrent systems is to interpret each place as a computation state, each transition as an instruction, and each token as a thread of control.

Communicating Sequential Processes, or CSP, is founded on the principle that input and output are basic primitives of programming, and that the parallel composition of sequentially-executing processes is a fundamental method of structuring programs [Hoa78]. CSP performs communication through synchronous message passing. The send primitive "!" outputs an item to another process. The receive primitive "?" inputs an item from another process. Many kinds of useful program structures, such as monitors and procedures, may be constructed from these fundamental primitives.

2.3 The IFDS Algorithm

This thesis presents a concurrent formulation of the Extended IFDS algorithm. The IFDS and Extended IFDS Algorithms exhibit irregular parallelism, but previous work presents these alogirthms in sequential form only. Section 3.2 presents a more detailed overview of the IFDS algorithm. Chapter 4 presents a more detailed overview of the Extended IFDS algorithm.

Reps et al. proposed an algorithm to precisely solve interprocedural, finite, distributive, subset problems, or IFDS problems, that is asymptotically faster than any previously-known algorithm for this type of problem [RHS95]. "Precise" in this context means that, with respect to function calls in the input program, only call-flow and return-flow paths that may actually be taken are considered. Distributive subset problems restrict the set of facts generated at each input program-node to be a subset of the powerset of some finite set of facts. Furthermore, the flow function must distribute over either the union or intersection operator. Distributive subset problems represent a large class of useful dataflow-analysis problems.

A set of extensions to the IFDS algorithm increases its applicability to a wider range of analysis problems than the original IFDS algorithm [NLR10]. In particular, the algorithm features an extension for demand-driven construction of the exploded supergraph, an extension to provide caller-context information to return-flow functions, an extension to improve the precision of analyses using Static Single Assignment, and an extension to speed up computation on certain types of problems where some dataflow facts subsume others.

2.4 Studies of Irregular Applications

Irregular applications present certain difficulties with respect to parallelization. In particular, determining an optimal work-distribution strategy, predicting memory-access patterns, and determining the dependencies among operations are often impossible prior to application execution.

The Galois project has concluded that irregular programs, which operate on directed graphs or pointer-based structures, exhibit amorphous data-parallelism, a type of parallelism that often cannot be discovered using static techniques [KBI$^+$09]. The ParaMeter tool was developed in conjunction with the Galois project to determine how much data-parallelism actually exists in these types of programs. Each of six work-list-based algorithms is instrumented so that multiple items from the work-list are speculatively executed in parallel. If any two items interfere with each other, ParaMeter chooses one to complete in the current iteration and the other is deferred to the next iteration. Each iteration therefore contains a set of work-list items that can be executed in parallel without conflict, provided the set of available processors is sufficiently large to execute all items. Peak available parallelism was found to be in the hundreds to thousands of potentially parallel items. The level of parallelism discovered by ParaMeter is not necessarily the maximum level of parallelism possible, but is an informative approximation that may approach this maximum.

Panwar et al. have studied the implementation of irregular programs. Each problem studied is expressed as a maximally concurrent *ideal algorithm* where each fine-grain unit of computation is encapsulated in an *actor* [PKA96]. The ideal algorithm specification is portable across different hardware architectures because it naturally expresses the dependencies among computations without reference to any particular work distribution strategy. The work shows how different partitioning and distribution strategies, or PDSs, affect performance. Of the PDSs examined, dynamic load-balancing is found to be the most effective.

2.5 Previous Approaches to Concurrency

In consideration of the difficulty of concurrent programming, many approaches have been proposed to alleviate this difficulty.

2.5.1 Fully Automatic Solutions

The two methods discussed here propose techniques for parallelizing unmodified code via hardware and compiler support. In both cases, the proposed hardware is simulated.

Thread-Level Data Speculation, or TLDS, is a technique designed to automatically introduce thread-level parallelism into non-numeric sequential code [SM98]. A TLDS-enabled compiler identifies *speculative regions* in the code, normally corresponding to loops or recursive functions. Each dynamic iteration or recursive call is a set of executable instructions called an *epoch*. Multiple epochs execute in parallel, and a hardware-assisted runtime system determines if any read-after-write dependencies have been violated. If such a violation occurs, the epoch that performed the bad load and all epochs that follow it may be fully or partially rolled back and re-executed to restore data correctness. Unlike STM approaches, which require programmers to make explicit decisions about where mutually-exclusive regions are, TLDS discovers these regions automatically. In addition, the compiler is free to make potentially unsound optimizations with respect to memory accesses, since any unsound accesses are correctable at run-time. Benchmark tests were performed on simulated hardware. Benchmark tests indicated simulated speedups between 1.03 times and 3.87 times on a four-processor system. The speedups realized were heavily dependent on the percentage of execution time spent within speculative regions.

Program Demultiplexing, or PD, is a hardware-assisted technique for speculatively executing program methods in parallel with the main thread [BS06]. Based on data from execution profiles, PD selects certain program methods to begin execution before before their call sites in the main thread are reached. Their side effects are stored in an execution buffer. When the main execution thread reaches the call site for such a method, it waits for the method's execution thread to complete, then commits the contents of the method's execution buffer if no data dependency violations are detected. If a dependency violation is detected, the method's execution buffer is invalidated and the method is re-executed. Parallel method execution is triggered as soon as the method's entire read set is likely to be available. The read set availability is based on an execution profile generated during prior profiling runs of the program. This profile has a high probability of being correct, and the remaining incorrect executions are corrected at run-time. Benchmark tests were performed on simulated hardware. Benchmarked applications were all able to make use of at least three processors, and some could make use of up to six processors. Speedups ranged from 1.4 times to 2.7 times.

2.5.2 Semi-Automatic Solutions

There is a large existing corpus of sequential code that may benefit from conversion into multi-threaded code. Due to the difficulty of creating fully automatic parallelization solutions, the techniques in this section involve the programmer in making decisions about how to safely create multi-threaded code.

Rul et al. developed a framework to identify potential parallelism opportunities within sequential programs [RVDB07]. This framework attempts to discover non-speculative parallelism by observing data dependencies among functions. The technique is profile-based, so correct execution of a candidate program requires either a speculative technique such as thread-level speculation [SM98] or manual modification of the program source code. In addition to identification of data dependencies, the framework also attempts to identify the parallel constructs that best describe the program's data usage patterns. The framework attempts to cluster strongly related functions into threads, providing hints about which data structures should be thread-private and which need to be shared among threads. The bzip2 program was analyzed by the framework, then hand-modified to support multiple threads. On a four-processor system, the maximum speedup observed for the compression portion of bzip2 was 3.64, and the maximum speedup observed for the decompression portion was 1.41. Due to the data dependency structure, the decompression portion could only be divided into two threads.

ReLooper is a system designed to assist programmers with refactoring sequential iterations over standard Java arrays into parallel operations over Java's `ParallelArray` construct [DRT+09]. In many programs, there is significant latent parallelism available in iterating over arrays of objects. However, there can be dependencies among iterations that are difficult to discover, making manual transformation of sequential loops into parallel operations difficult and time-consuming. ReLooper addresses this problem by automating the process of determining whether loop iterations are safe to parallelize. If so, the loop can be automatically replaced with an equivalent parallel operation. If not, potential conflicts are shown to the programmer, who can attempt to resolve these conflicts before re-running the analysis. This approach significantly reduces the amount of effort required to re-write parallel loops, and it is fast enough to allow interactive use by programmers. ReLooper's conclusions are conservative, i.e. it will only declare a loop to be safe if it can prove its safety.

The Galois system has introduced syntactic constructs for iterating over ordered and unordered sets [KPW+07]. Iterations are optimistically executed in parallel with other iterations. A runtime system detects and rolls back potentially conflicting accesses to shared memory. The syntactic constructs introduced are designed to be integrated with a variety of object-oriented languages. The Galois approach to parallelism is informed by the beliefs: that "Optimistic parallelism is the only plausible approach to parallelizing many, if not most, irregular applications;" that new syntactic constructs embedded in an object-oriented language are necessary to enable the programmer to easily express parallelizable operations; and that "Concurrent access to mutable shared objects is fundamental, and cannot be added to the system as an afterthought as is done in current approaches to optimistic parallelization." Two irregular work-list-based algorithms were implemented twice, one version using fine-grain locking, and the other version using the Galois system to generate optimistically executing code. Although speedups over sequential code were seen with as few as two hardware threads, the Galois approach did not provide better performance or scalability than fine-grain locking.

The Concurrent Collections programming model, or CnC, enables direct expression of the real dependencies in a given computation [Kno09]. All data is logically passed by value, forming a partial ordering of data dependencies. Control dependencies are defined by control tags. A control tag is generated when it is known that a particular computation step needs to run, but it is up to the scheduler to decide when a tagged computation step is actually performed. The goal of this work is to separate the role of the domain expert from the role of the tuning expert. The domain expert, who does not have any particular expertise in parallelism or performance optimization, expresses a computation without any explicit parallel constructs. The results of this computation are deterministic regardless of scheduling order, configuration, or machine architecture. The tuning expert, which may be a person or an automated static or dynamic analysis, configures the program for optimal execution on a particular machine architecture. Implementations of CnC currently support architectures that provide task parallelism, pipeline parallelism, or data parallelism. Static and dynamic scheduling are both supported, as well as both shared and distributed memories. The focus of CnC is on computation tasks. Long-running or I/O-heavy tasks are not well-supported.

2.5.3 Solutions for Preserving Determinism

Sequential code, with the exception of I/O operations, can provide a guarantee of deterministic execution. The next two approaches, Kendo and Deterministic Shared Memory Multiprocessing, attempt to return a similar level of determinism to ordinary multi-threaded code. The insight behind these approaches is that a guarantee of deterministic execution is first of all very advantageous for locating bugs in multi-threaded code. Secondly, a determinism guarantee is likely to reduce the number of concurrency-related bugs actually experienced because the number of possible thread interleavings is drastically reduced.

Kendo is a software system that provides deterministic execution of multithreaded programs [OAA09]. Kendo uses deterministic logical clocks that count arbitrary events on each thread. A thread may only acquire a lock if it is not only free in physical time, but also free in logical time. Correct operation of this technique requires that the application be free of data races. Kendo includes a data-race detector that is guaranteed to detect the first data race encountered in any particular application execution. One issue not addressed by this work is how the fundamentally non-deterministic nature of I/O affects applications of this technique.

Deterministic Shared Memory Multiprocessing, or DMP, enables deterministic communication among threads [DLCO09]. This guarantee allows arbitrary multithreaded shared-memory programs to execute deterministically with very little performance penalty, provided some hardware support exists. Each access to data shared among processors can only occur if the accessing processor possesses the token for that piece of data. Tokens are passed from processor to processor in deterministic order. Due to guarantees of deterministic communication among threads, DMP can substantially reduce time spent debugging multithreaded programs. DMP may also increase reliability of deployed software by ensuring that deployed software executes with the same determinism as it did during testing. DMP requires hardware and OS support to function correctly.

2.5.4 Software Transactional Memory

Software transactional memory is a recently-developed method of providing mutual exclusion in shared-memory systems. However, it is not suitable for all types of concurrency problems, and its usefulness in large-scale real-world systems is still uncertain.

Software transactional memory, or STM, has been proposed as a way to simplify the construction of concurrent programs without requiring any new hardware support [ST95]. STM supports serializable transactions in shared memory. The assumption is that the set of data read or modified by any one transaction has a high probability of being independent of the data sets modified by concurrent transactions, i.e., transactions are executed optimistically. If any of a transaction's reads are invalidated before the transaction commits, the transaction is aborted and optionally restarted. The STM implementation in this paper is non-blocking and provides a guarantee that some transaction will always succeed, thereby avoiding deadlocks by design.

Dan Grossman proposed an analogy to assist in the understanding of transactional memory [Gro07]. Grossman says, "Transactional memory (TM) is to shared-memory concurrency as garbage collection (GC) is to memory management." Grossman makes a conjecture that the analogical similarity between TM and GC implies TM may follow a course of development similar to that of GC. Namely, that TM does not require hardware support to succeed, that TM may take much longer to reach mainstream than currently expected, and that unmodified TM semantics are sufficient to meet the requirements of most applications. However, TM is not likely to make concurrent programming as easy as GC makes memory management. First, TM does not provide guidance about how large atomic sections should be or where they should be placed. Second, incorrect interleavings between atomic sections are not a very easy problem to avoid.

Baldassin and Burckhardt developed alternative transaction semantics for games [BB09]. For the application studied, STM did not perform well due to a high abort frequency.

Lock-based mutual exclusion did not provide sufficient concurrency. For both STM and lock-based mutual exclusion, modifying the lengths and positions of the transactions for the purpose of performance tuning proved to be a non-trivial task. The proposed solution is to avoid transaction aborts by replicating modified objects on write and periodically merging changes so they become visible to future transactions. Merge functions and task barriers, which the programmer may use to specify dependencies among objects, ensure data consistency. Using this approach, the application was able to approach maximal concurrency on a four-core machine.

The Atomic Quake project is an evaluation of the effectiveness of STM when applied to a real-world software system [ZGU$^+$09]. A lock-based concurrent implementation of the Quake game server was restructured to use STM for all mutual-exclusion requirements. A total of 61 atomic sections were identified. They ranged in duration from 200 cycles to 1.3 million cycles, and accessed data read and write sets that ranged from a few bytes to 1.5 megabytes. Transactions were nested up to nine levels deep. A few of the atomic sections contained calls to functions with unknown side effects. A transaction that executes such a call becomes irrevocable. Only one transaction at a time may be in irrevocable mode. Examples of transactions containing error handling and recovery code were found. Examples of data accessed from both within and outside transactions were found. No examples were found where the kind of access to one piece of data depended on the value of another piece of data. The performance scalability of STM for this application was found to be poor relative to fine-grain locking – eight threads with STM were slower than two threads. Loss of scalability was attributed to transactional system overhead and to the cost of aborted transactions.

Yoo et al. identified four major types of bottlenecks that occur when using STM in complex, large-scale workloads [YNW$^+$08]. *False conflicts* occur when the conflict detection scheme is overly coarse. *Over-instrumentation* occurs when the compiler generates unnecessary read/write barriers due to lack of application-level knowledge. *Privatization-safety cost* is the overhead associated with guaranteeing that non-transactional operations do not interfere with transactional operations. Conservative assumptions made by the compiler add privatization overhead where it is not needed by the application. *Poor amortization* occurs when the costs of transaction startup and teardown overwhelm the costs of actual transaction execution, causing poor scalability. A new set of metrics for characterizing complex transactional loads was proposed, as well as a new interface that allows programmers to declare that certain transactions do not require privatization-safety. The new interface enabled an average 32% speedup over STM without the new interface.

2.5.5 Language Solutions

Language solutions are designed to support the creation of new concurrent code. Fortress and Oz support implicit concurrency. μC++, Cilk, and X10 support explicit concurrency.

Fortress is a language development effort undertaken by Sun Microsystems [Ste06]. Fortress is designed to encourage as much parallelism as possible, to the extent that sequential loops are intentionally a little harder to write than parallel loops. The language is designed to represent formulae graphically so that computations expressed in a program appear very similar to computations expressed in traditional mathematical notation. One objective of Fortress is to move as much functionality as possible out of the compiler and into the libraries. A rich parametrized polymorphic type-system guides decisions about how to organize concurrent computations and data.

The Oz language and programming model integrates the functional and object-oriented programming models with constraint logic programming [Smo95]. The result is a system that provides implicit support for concurrency. Computation is defined by a set of *reduction tasks* that operate on a *store*. A reduction task may create new reduction tasks and/or modify the store. Reduction is atomic, and once a reduction is finished its task is deleted. Concurrency is achieved by the inherent possibility that there may be many reducible tasks present in the system at any given time. Concurrency control is achieved when a reduction synchronizes on the store. Synchronization on the store means that a task is not reducible until a certain condition is met, e.g. a symbol required by the reduction becomes defined. Each reduction is deterministic, and the programming model is built in such a way that once a task becomes reducible it cannot become irreducible. The result is that reductions always proceed in a well-defined manner regardless of how many tasks may be executed in parallel.

The μC++ language adds a set of concurrency-supporting constructs and a lightweight threading system to the C++ language [BDS$^+$92]. Four concurrency abstractions are introduced as extensions to C++'s object model: coroutine, monitor, coroutine-monitor, and task. Because these abstractions are based on C++'s object model, they support all of C++'s object-oriented features, including templates, inheritance, destructors, and virtual methods. Aside from the coroutine, all of these abstractions support implicit mutual exclusion among their member methods. By default, communication between threads is synchronous and type-safe. μC++ is designed to operate with shared-memory systems.

The Cilk system adds a small number of new keywords to the C language to provide high-performance fine-grain task execution [FLR98]. The model of concurrency used by Cilk is called *structured parallelism*. The `cilk` keyword identifies a function as a Cilk task. Cilk tasks must be called using the `spawn` keyword. The `sync` keyword denotes a barrier that waits for previously spawned tasks to complete. Cilk's concurrency model is a hierarchically-structured parallelism where tasks have parent-child relationships and child tasks always complete before their respective parent tasks. The `inlet` keyword denotes a sub-function whose argument is the return value of a child task. Inlets execute atomically and may handle the return values of multiple child tasks. The `abort` keyword may be executed inside of an inlet to immediately terminate all remaining child tasks. The abort facility was designed to allow easy termination of speculative search tasks where the discovery of an answer by one task renders the other tasks unnecessary. Cilk was designed for a specific class of compute-heavy applications, and therefore does not include features to ease general-purpose concurrent programming.

The Cilk implementation includes what is claimed to be the first "provably good" work-stealing scheduler [BL99]. "Provably good" means that the expected execution time, the maximum bound on space required, and the maximum bound on total inter-task communication are all known and reasonable for any well-structured computation. The communication bound is evidence that work-stealing schedulers, which balance loads by letting idle threads steal work items from other threads, are more communication-efficient than work-sharing schedulers, which attempt to evenly distribute work items to threads as the items are generated.

The X10 language is built on a work-stealing scheduler similar to Cilk's scheduler [ABB+07]. X10 enhances the Cilk model in a number of ways, notably the notion of executing computations at explicitly-specified logical *places*. A place binds together a set of shared variables and a set of activities that operate on those shared variables. Programs that use X10's `async, finish, atomic` and `place` constructs are guaranteed to be deadlock-free. X10 also provides a generalized form of barriers called *clocks*. Programs that use clocks are guaranteed to be deadlock-free provided computation, memory, and communication resources are unbounded. X10 is designed to accommodate a much larger range of applications than Cilk.

Chapter 3

Background

This chapter describes the Actor model, followed by a presentation of the original IFDS algorithm.

3.1 The Actor Model

The Actor model is a computational model that seeks to define the operation of concurrent programs as simply as possible. The basic notion behind the Actor model is that any computation can be defined as a set of entities, or *actors*, which communicate by passing messages. Each actor processes the messages it receives in some sequential order. The actor buffers received messages until it can process them, as shown in Figure 3.1.

The Actor model also suggests a particular design process for writing actor-based programs. This process is similar to the process of object-oriented programming. The three stages of this design process are:

1. Identify the types of actors in the system, which is similar to the object-oriented process of determining how to model various physical or logical entities as classes.
2. Decide the types of messages each type of actor should respond to, which is similar to defining class interfaces.
3. Define what each type of actor does in response to each type of message, which is similar to creating implementations of the defined interfaces.

The main difference between actor-based programming and object-oriented programming is that actor-based message-passing is *asynchronous* and *unordered*. *Asynchronous*

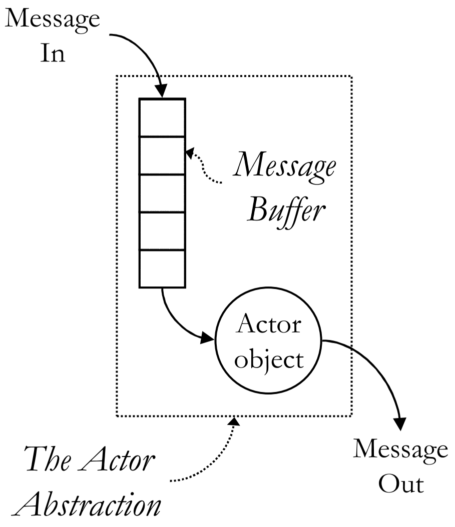

Figure 3.1: The Actor Abstraction

means that the sender of a message continues execution immediately after sending a message, without waiting for the receiver to process it. *Unordered* means that messages can arrive in any order, not just the order in which they were sent. Messages in object-oriented programming languages, which are often optimized to be simple function calls, are normally *synchronous* and therefore also *ordered*. Whenever a synchronous function call takes place, the caller waits until the function returns before continuing execution. It is possible for an actor-based program to use synchronous message-passing by pausing to wait for a reply after sending a message, but this process is typically less efficient than making an equivalent function call.

3.1.1 Defining Actor Classes

The notation used to define actors is shown in Figure 3.2. An actor definition is called an *actor class* to distinguish it from an ordinary class. This notation contains the following components:

1. A name.

2. An optional list of arguments. These must be supplied when the actor is created.

3. An initial set of executable statements $stmts_{init}$. These statements are the actor's constructor and execute immediately after creation.

```
def ActorName($arguments$)
  $stmts_{init}$
begin ($message$) switch
  case $message$ matches $pattern_1$ : $stmts_1$
  . . .
  case $message$ matches $pattern_n$ : $stmts_n$
  finally : $stmts_{final}$
end
```

Figure 3.2: Actor Class Definition

4. A set of cases that match incoming messages with typed patterns. A match only succeeds if the message type and the number of message parameters is the same as the pattern. For example, the message AddEdge⟨"A","B"⟩ matches the pattern AddEdge$\langle d_1, d_2 \rangle$. When the match succeeds, the values of d_1 and d_2 are "A" and "B", respectively.

5. A set of executable statements that follow each case statement.

6. A set of statements following a **finally** keyword that are executed every time a message is received. Any code that follows a successful case match is executed first, followed by the code after **finally**.

All argument variables and all local variables created by $stmts_{init}$ persist for the lifetime of the actor and are visible to all statements $stmts_1$ through $stmts_n$ and $stmts_{final}$. This set of variables is analogous to member variables in object-oriented languages. These variables persist until the actor is garbage-collected. Any variables created by $stmts_1$ through $stmts_n$ or $stmts_{final}$ are only live and visible inside their respective statement blocks. These variables are only created in response to a received message and so do not need to persist after the message is processed.

This notation's similarity to ordinary function definitions is intentional. Like an ordinary function, an actor takes a set of arguments. Also like an ordinary function, an actor allocates an execution frame and executes a set of statements when it is invoked. The contents of all arguments and local variables are stored in this execution frame. Unlike an ordinary function, it does not return any values. When $stmts_{init}$ finishes execution, the address of the actor is returned instead. Normally, this address points to a heap-allocated object that contains the execution frame. Formally speaking, it returns a function closure. Because this closure is persistent, the notation "**new** ActorName($arguments$)" is preferred to denote the creation of a new actor instead of the more concise function-call-style "ActorName($arguments$)".

When the actor receives a message, it selects at most one case statement for processing. The local variables created by the statements following **case** are stored in a temporary frame which is discarded as soon as those statements finish execution. If the actor's closure is like a heap object in object-oriented programming, then these temporary frames are like the stack frames created during calls to member functions. Unlike member function calls, which normally create these temporary frames on the caller's stack, temporary frames created in response to messages are normally created by the receiver of the message.

Logically, each actor consists of exactly one thread that is either idle while waiting to receive a message, or busy executing a received message. If a message arrives while the actor is busy, it is normally placed in some kind of buffer. The implementation is free to choose any kind of buffering mechanism it wants, under the condition that all sent messages are guaranteed to be eventually received.

Implementations of the Actor model do not normally assign each actor a unique OS thread, but use a scheduler to dynamically assign actor-generated tasks to a pool of OS threads. It is accurate to say that each actor is assigned a unique user-thread, and the job of the scheduler is to map user-threads to OS threads.

3.1.2 Formal Properties of the Actor Model

A pure actor-based program is an expression of partial orderings or dependencies among operations. The act of sending a message to another actor represents a sequential dependency between the sender of the message and the receiver of the message.

In addition to being unordered, message-passing in the Actor model has an unbounded delay, which means that there may be an arbitrarily large amount of time between the sending of a message and its receipt.

The Actor model provides a guarantee that all messages sent will eventually be delivered. The Actor model considers the guarantee of delivery to be orthogonal to guarantees of timeliness or ordering.

3.2 The IFDS Algorithm

The IFDS algorithm introduced by Reps et al. is a *precise* dataflow-analysis algorithm that solves interprocedural, finite, distributive, subset problems in polynomial time [RHS95].

- *Precise* refers to a merge-over-all-valid-paths solution, in contrast to a merge-over-all-paths solution. The all-valid-paths solution only includes control-flow paths that can actually be taken given standard call/return method invocation semantics. In contrast, an all-paths solution treats the problem as a giant intra-procedural problem, and therefore may include control-flow paths that do not obey call semantics, e.g. paths at one return-site that are only reachable through call-sites in a different procedure. The inclusion of these additional dataflow edges would never make the solution unsound; it merely means that the static approximation is less precise.

- *Interprocedural* problems take called methods into account when performing the analysis. The information computed for any given method may vary depending on the information computed for other methods called by that method.

- *Finite* problems are those problems where the domain of possible dataflow facts that may be computed for any given instruction is finite.

- *Distributive* problems are those where the flow-functions distribute over the merge operator, typically the union operator. Applying a flow-function to the union of multiple facts is equivalent to applying the flow-function to each fact individually and performing a union on the results. For fact sets a and b and flow-function f, $f(a) \cup f(b) = f(a \cup b)$. If the merge operator is the intersection operator, it is possible to transform the problem instance into one that uses the union operator. If the problem instance is "must-be-X" with the intersection operator, then it is equivalent to a "may-not-be-X" instance with the union operator.

- *Subset* problems are those where the set of facts present at any given instruction is a subset of the total set of facts D. The domain of the flow-functions is therefore the powerset of D.

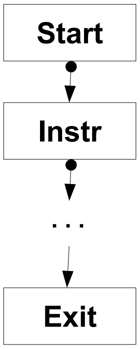

Figure 3.3: IFDS Supergraph for One Procedure (G^*)

Each instruction in the input program corresponds to a single node $n \in N^*$, where N^* represents the complete set of program instructions. There is a corresponding set of directed edges E^* that represents the control-flow dependencies between instructions. Figure 3.3 shows the structure of a single procedure. The boxes are nodes in N^*, and the lines are edges in E^*. Each procedure must have a single start node and a single exit node.

If the procedure contains multiple return statements, additional edges must be created so that all possible control flows pass through the exit node before leaving the procedure.

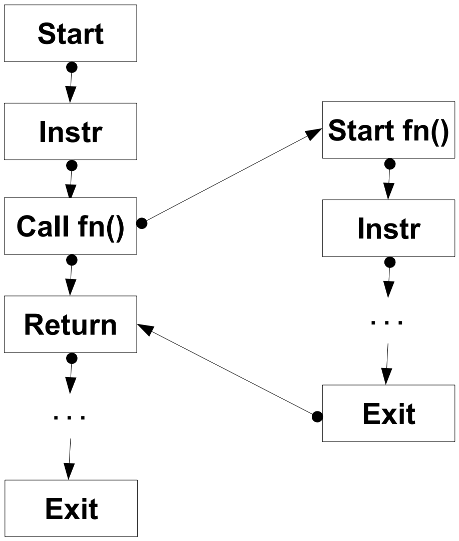

Figure 3.4: IFDS Supergraph for a Procedure Call (G^*)

The graph $G^* = (N^*, E^*)$, or the *supergraph*, represents all instructions in the program plus the intra-procedural and inter-procedural dataflow dependencies between them. For each procedure p, there is:

- a single start node s_p;
- a single exit node e_p;
- a set of call-site nodes $Call_p$;
- a set of return-site nodes Ret_p;
- an edge from each $c_p \in Call_p$ to the start of its corresponding called procedure s'_p;
- an edge from each e'_p to each corresponding $r_p \in Ret_p$;
- and an edge from each c_p to its corresponding r_p.

See Figure 3.4 for an illustration of a procedure call that includes all of the required edges and nodes. N_p is the subset of nodes in N^* that comprise procedure p.

A particular IFDS problem instance is denoted by the graph $G^\sharp_{IP} = (N^\sharp, E^\sharp)$, or the *exploded supergraph*. Each node n in N^* is "exploded" so that the corresponding node

$n^\sharp$ in $N^\sharp$ consists of all pairs $\langle n, d\rangle$ where d is either a fact in D or the special zero fact. Formally, $N^\sharp = N^* \times (D \cup \{\mathbf{0}\})$. The zero fact (roughly) represents the empty set. Passing the zero fact to a flow-function means that the flow-function should return those facts that are always true at the given node. In general, passing the zero fact to a flow-function should always include the zero fact in the result. Formally, $\mathbf{0} \in f(\mathbf{0})$.

Figure 3.5 shows how the supergraph G^* relates to the exploded supergraph $G^\sharp$. At each node in G^*, there are a number of nodes in $G^\sharp$, each corresponding to a single fact in D. In Figure 3.5, D contains the elements **A** and **B**, so $G^\sharp$ contains nodes corresponding to **0**, **A**, and **B** for each node in G^*. Likewise, for each edge in G^*, there are a number of edges in $G^\sharp$. If the source node of an edge in $G^\sharp$ is an argument to a flow-function, then the destination node is contained in the result set of that flow-function.

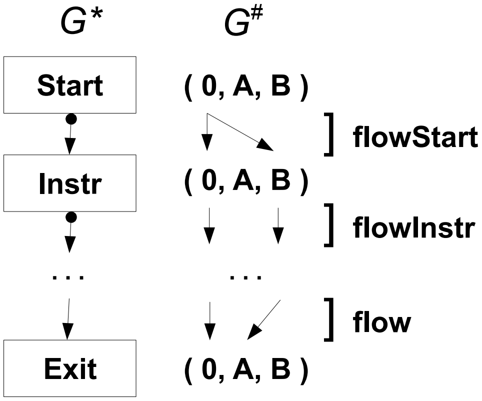

Figure 3.5: IFDS Exploded Supergraph for One Procedure (G^* and corresponding $G^\sharp$)

It is important to note that IFDS does not *call* any flow-functions; the edges in $E^\sharp$ *are* the flow-function – they represent the dataflow relationships between the nodes in $N^\sharp$. Essentially, the IFDS algorithm represents the dataflow-analysis problem as a graph-reachability problem.

Formally, if $FLOW(m, n)$ is the flow-function from node m to node n in N^*, and d_1, d_2 are elements of $D \cup \{\mathbf{0}\}$, then:

$$E^\sharp = \{\langle m, d_1\rangle \to \langle n, d_2\rangle \mid m \to n \in E^* \text{ and } d_2 \in FLOW(m, n)(\{d_1\})\}$$

An edge in $E^\sharp$ that starts with the zero fact represents a constant function.

The key property that enables a graph-based representation of the flow function is the distributivity. This means that for a fact set D in the domain of a distributive flow-function f, calling f with the argument D gives the same result as calling f separately on each element of D and on the empty set, and performing a union of the results. Formally, the following identity holds:

$$f(D) = f(\emptyset) \cup \bigcup_{d \in D} f(\{d\})$$

One implication of the distributivity property is that flow-function composition is "compressible." A flow-function may be represented by a graph with at most $(|D|+1)^2$ edges. This maximum bound consists of every fact in set D plus the zero fact mapped to all facts in D plus the zero fact. For any two flow-functions f and g, the composition $g \circ f$ also has at most $(|D|+1)^2$ edges. The accumulated effects of any number of flow-functions are therefore representable in a compact fashion.

The IFDS algorithm is shown in Algorithm 3.1. The algorithm uses the following functions:

- $returnSite(c)$ maps a call node c to its corresponding return-site node;
- $procOf(n)$ maps a node n to the name of its enclosing procedure;
- $calledProc(c)$ maps a call node c to the name of the called procedure;
- $callers(p)$ maps a procedure name p to the set of call nodes that represent calls to that procedure.

The IFDS algorithm operates by finding all nodes in $N^\sharp$ that are reachable from the node $\langle s_{main}, \mathbf{0} \rangle$ through some valid path in $E^\sharp$. Performing an intra-procedural analysis of any procedure p yields a set of edges of the form $\langle s_p, d_1 \rangle \rightarrow \langle n_p, d_2 \rangle$ where s_p is the procedure start node, n_p is some node in the procedure, d_1 is a fact present at the start node, and d_2 is a fact at n_p that is reachable from $\langle s_p, d_1 \rangle$. Edges of this form are called path-edges.

Whenever the algorithm discovers a new path-edge, it adds that edge to a work list. This addition occurs in the Propagate function on line 9. A common implementation of the IFDS algorithm uses a queue structure for the work list, thereby processing new path-edges in the order of discovery. However, the algorithm places no constraints on the order in which the work-list contents are processed. Implementations are free to choose alternate orderings.

The main ForwardTabulateSLRPs function terminates when the work list is empty. An empty work list implies that no additional nodes in $N^\sharp$ are reachable from $\langle s_{main}, \mathbf{0} \rangle$. Lines 6 through 8 in the algorithm then collect in X_n the set of reachable facts at node n.

The algorithm retains precision by adding edges according to the all-valid-paths criteria instead of the all-paths criteria. The all-valid-paths criteria means that when the algorithm

```
declare PathEdge, WorkList, SummaryEdge: global edge set

algorithm Tabulate($G^{\sharp}_{IP}$)
begin
  Let ($N^{\sharp}$, $E^{\sharp}$) = $G^{\sharp}_{IP}$
  PathEdge := { $\langle s_{main}, \mathbf{0} \rangle \rightarrow \langle s_{main}, \mathbf{0} \rangle$ }
  WorkList := { $\langle s_{main}, \mathbf{0} \rangle \rightarrow \langle s_{main}, \mathbf{0} \rangle$ }
  SummaryEdge := $\emptyset$
  ForwardTabulateSLRPs()
  for each $n \in N^*$ do
    $X_n$ := { $d_2 \in D \mid \exists\, d_1 \in (D \cup \{\mathbf{0}\})$ such that $\langle s_{procOf(n)}, d_1 \rangle \rightarrow \langle n, d_2 \rangle \in$ PathEdge }
  od
end

procedure Propagate($e$)
begin
  if $e \notin$ PathEdge then Insert $e$ into PathEdge; Insert $e$ into WorkList fi
end

procedure ForwardTabulateSLRPs()
begin
  while WorkList $\neq \emptyset$ do
    Select and remove an edge $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ from WorkList
    switch $n$

      case $n \in Call_p$ :
        for each $d_3$ such that $\langle n, d_2 \rangle \rightarrow \langle s_{calledProc(n)}, d_3 \rangle \in E^{\sharp}$ do
          Propagate($\langle s_{calledProc(n)}, d_3 \rangle \rightarrow \langle s_{calledProc(n)}, d_3 \rangle$)
        od
        for each $d_3$ such that $\langle n, d_2 \rangle \rightarrow \langle returnSite(n), d_3 \rangle \in (E^{\sharp} \cup$ SummaryEdge$)$ do
          Propagate($\langle s_p, d_1 \rangle \rightarrow \langle returnSite(n), d_3 \rangle$)
        od
      end case

      case $n = e_p$ :
        for each $c \in callers(p)$ do
          for each $d_4, d_5$ such that $\langle c, d_4 \rangle \rightarrow \langle s_p, d_1 \rangle \in E^{\sharp}$ and $\langle e_p, d_2 \rangle \rightarrow \langle returnSite(c), d_5 \rangle \in E^{\sharp}$ do
            if $\langle c, d_4 \rangle \rightarrow \langle returnSite(c), d_5 \rangle \notin$ SummaryEdge then
              Insert $\langle c, d_4 \rangle \rightarrow \langle returnSite(c), d_5 \rangle$ into SummaryEdge
              for each $d_3$ such that $\langle s_{procOf(c)}, d_3 \rangle \rightarrow \langle c, d_4 \rangle \in$ PathEdge do
                Propagate($\langle s_{procOf(c)}, d_3 \rangle \rightarrow \langle returnSite(c), d_5 \rangle$)
              od
            fi
          od
        od
      end case

      case $n \in (N_p - Call_p - \{e_p\})$ :
        for each $\langle m, d_3 \rangle$ such that $\langle n, d_2 \rangle \rightarrow \langle m, d3 \rangle \in E^{\sharp}$ do
          Propagate($\langle s_p, d_1 \rangle \rightarrow \langle m, d_3 \rangle$)
        od
      end case

    end switch
  od
end
```

Algorithm 3.1: Original IFDS Algorithm reproduced from Reps et al. [RHS95]

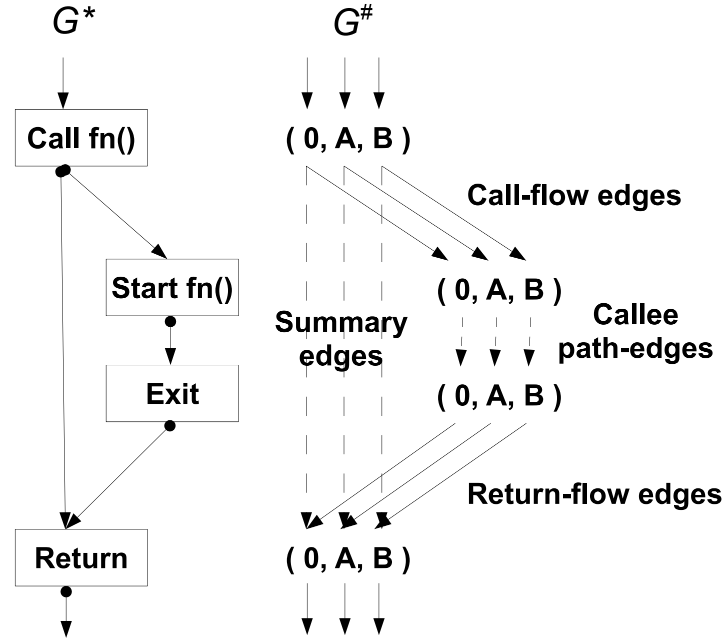

Figure 3.6: IFDS Summary Edge Generation (G^* and corresponding $G^\sharp$)

encounters a procedure call, it only considers the call edges and return edges that are applicable to that particular call-site. An edge that results from the composition of a call-flow-function, a set of intra-procedural summary edges, and the corresponding return-flow-function is called a Summary Edge. Intra-procedural summary edges are called *callee path-edges* in the remainder of this document. Callee path-edges are exactly those path-edges that end at the exit node of some procedure. A single set of callee path-edges is generated for each procedure, but there is a different set of summary edges that is generated for each site that calls the procedure.

Whenever a new callee path-edge is generated, the algorithm queries $E^\sharp$ on line 23 to obtain the call-flow edges and return-flow edges that connect with it. The composition of these edges is stored in the SummaryEdge set on line 25. These summary edges represent the effect of the called procedure in the *caller's* context; summary edges are dynamically generated dataflow edges from the call-site to the return-site. Figure 3.6 illustrates this process of summary edge generation. The solid lines on the right-hand-side of the figure represent edges in $E^\sharp$, and the dashed lines represent dynamically computed edges.

Once a new summary edge is discovered, the tabulation must be made to proceed as if that summary edge already existed as part of $E^\sharp$. Lines 26 through 28 of Algorithm 3.1 propagate any new path-edges that may be generated as a result of adding a new summary edge. New path-edges may also need to be propagated when a new path-edge ending at a call-site is discovered. Line 17 takes the known summary edges into account when performing the call-site to return-site query.

Chapter 4

The E-IFDS Algorithm

The original IFDS algorithm as presented by Reps et al. [RHS95] has certain limitations when applied in practice. In particular, many analyses involving objects or pointers to objects are unsuitable for use with IFDS without certain modifications. The modifications listed here comprise E-IFDS, shown in Algorithm 4.1:

- Demand construction of the exploded supergraph. See Section 4.1.
- Demand construction of summary edges. See Section 4.2.
- Caller context for return-flow-functions. See Section 4.3.
- Multiple called procedures per call-site. See Section 4.4.
- Redundant fact removal. See Section 4.5.

The following sections detail these modifications. The E-IFDS algorithm shares some characteristics with the Extended IFDS algorithm presented by Naeem et. al. [NLR10], but the two algorithms are not identical. The differences include the following:

- The Extended IFDS algorithm maintains a SummaryEdge set, whereas E-IFDS does not.
- The Extended IFDS algorithm supports the Static Single Assignment form, or SSA, without loss of precision. As presented in this thesis, E-IFDS does not make any special provisions for SSA.
- E-IFDS explicitly allows multiple called procedures at a single call-site, whereas the Extended IFDS algorithm does not.

```
declare PathEdge, WorkList: global set of triples from $D^0 \times N^* \times D^0$
declare CallEdgeInverse: global set of 4-tuples from ProcNames $\times D^0 \times N^* \times D^0$

algorithm Tabulate($G^*$, flow$_i$, flow$_{call}$, flow$_{ret}$, flow$_{thru}$)
begin
  Let ($N^*$, $E^*$, $s_{main}$) = $G^*$
  PathEdge := { $\mathbf{0} \rightarrow \langle s_{main}, \mathbf{0} \rangle$ }
  WorkList := { $\mathbf{0} \rightarrow \langle s_{main}, \mathbf{0} \rangle$ }
  CallEdgeInverse := $\emptyset$
  ForwardTabulateSLRPs()
  for each $n \in N^*$ do
    $X_n$ := { $d_2 \in D \mid \exists\, d_1 \in D^0$ such that $d_1 \rightarrow \langle n, d_2 \rangle \in$ PathEdge }
  od
end

procedure Propagate($e$)
begin
  if $e \notin$ PathEdge then Insert $e$ into PathEdge; Insert $e$ into WorkList fi
end

procedure ForwardTabulateSLRPs()
begin
  while WorkList $\neq \emptyset$ do
    Select and remove an edge $d_1 \rightarrow \langle n, d_2 \rangle$ from WorkList; Let p = $procOf(n)$
    switch $n$

      case $n \in Call_p$ :
        for each $p', d_3$ such that $p' \in calledProcs(n)$ and $d_3 \in$ flow$_{call}(\langle n, d_2 \rangle, p')$ do
          Propagate($d_3 \rightarrow \langle s_{p'}, d_3 \rangle$)
          Insert $\langle p', d_3 \rightarrow \langle n, d_2 \rangle \rangle$ into CallEdgeInverse
          for each $d_4$ such that $d_3 \rightarrow \langle e_{p'}, d_4 \rangle \in$ PathEdge do
            for each $d_5 \in$ flow$_{ret}(\langle e_{p'}, d_4 \rangle, \langle n, d_2 \rangle)$ do
              Propagate($d_1 \rightarrow \langle returnSite(n), d_5 \rangle$)
            od
          od
        od
        for each $d_3 \in$ flow$_{thru}(\langle n, d_2 \rangle)$ do
          Propagate($d_1 \rightarrow \langle returnSite(n), d_3 \rangle$)
        od
      end case

      case $n = e_p$ :
        for each $\langle c, d_4 \rangle$ such that $c \in callers(p)$ and $\langle p, d_1 \rightarrow \langle c, d_4 \rangle \rangle \in$ CallEdgeInverse do
          for each $d_5 \in$ flow$_{ret}(\langle e_p, d_2 \rangle, \langle c, d_4 \rangle)$ do
            for each $d_3$ such that $d_3 \rightarrow \langle c, d_4 \rangle \in$ PathEdge do
              Propagate($d_3 \rightarrow \langle returnSite(c), d_5 \rangle$)
            od
          od
        od
      end case

      case $n \in (N_p - Call_p - \{e_p\})$ :
        for each $m, d_3$ such that $n \rightarrow m \in E^*$ and $d_3 \in$ flow$_i(\langle n, d_2 \rangle)$ do
          Propagate($d_1 \rightarrow \langle m, d_3 \rangle$)
        od
      end case

    end switch
  od
end
```

Algorithm 4.1: The E-IFDS Algorithm

4.1 Demand Construction of the Exploded Supergraph

The original IFDS algorithm requires construction of the entire exploded supergraph $G^{\sharp}_{IP}$ prior to tabulation. The number of nodes in this graph is proportional to the size of the fact set D times the number of instructions in the program, i.e. $|N^{\sharp}| = |N^{*}| \times (|D| + 1)$. For many types of analyses, the fact-set domain D can be extremely large, resulting in a correspondingly large number of nodes in $N^{\sharp}$. For example, an analysis designed to determine the possible types of variables in a program might define the elements of D to have the form $\langle v, T\rangle$ where v is a variable name and T is a possible type of v. Since the size of N^{*} is proportional to the number of instructions in the program, the size of $N^{\sharp}$ is proportional to the number of instructions in the program, times the number of variables in the program, times the number of possible types in the program. When $|D|$ is large, $G^{\sharp}_{IP}$ can take a very long time to construct. In addition, the memory required to store $G^{\sharp}_{IP}$ can be prohibitive for all but the most trivial analyses.

For many analyses, the IFDS algorithm only traverses a small percentage of the edges in $G^{\sharp}_{IP}$. E-IFDS replaces queries of $E^{\sharp}$ in the original IFDS algorithm with calls to flow-functions that compute the set of exploded-supergraph edges leading out of a given node. The following flow-functions are defined:

- $\text{flow}_i(n^{\sharp})$ computes intra-procedural dataflow edges.
- $\text{flow}_{call}(n^{\sharp}, p)$ computes call-to-start dataflow edges when $n^{\sharp}$ is at a call-site. This function also includes the called procedure name p because more than one procedure may be callable from each call-site. See section 4.4.
- $\text{flow}_{ret}(n^{\sharp}, c^{\sharp})$ computes exit-to-return-site dataflow edges when $n^{\sharp}$ is at an exit node. The $c^{\sharp}$ argument is the caller context for $n^{\sharp}$. See section 4.3.
- $\text{flow}_{thru}(n^{\sharp})$ computes call-to-return-site dataflow edges when $n^{\sharp}$ is at a call-site. These edges represent intra-procedural information that is not affected by called procedures.

The $n^{\sharp}$ and $c^{\sharp}$ parameters are elements of $N^{\sharp}$, which are of the form $\langle n, d\rangle$ where $n \in N^{*}$ and $d \in (D \cup \{\mathbf{0}\})$. Note that in addition to the elements of D, d may be the zero fact.

One difficulty with demand-driven construction is that the original algorithm requires the inverse flow-function in the first query on line 23 of Algorithm 3.1. The query is attempting to find all call nodes $\langle c, d_4\rangle$ where there is an edge to the procedure start-node $\langle s_p, d_1\rangle$. Enumerating all possible edges for these flow-functions is not practical, and writing an inverse flow-function may not be straightforward. E-IFDS works around this issue by memoizing the call-flow edges as they are encountered on line 16 of Algorithm 4.1.

Finding the inverse of the call-flow-function is simply looking up the memoized edges on line 28.

An additional issue with demand querying of $E^\sharp$ is that the complete set of call-flow edges is not necessarily available at the query on line 28. The original IFDS algorithm is able to make this assumption on line 23 because it assumes the entire set of dataflow edges $E^\sharp$ is constructed in advance. Because E-IFDS cannot make this assumption, the effects of the exit node computation on lines 28 through 34 must be replicated at the call-site node computation on lines 17 through 21 as new call-flow edges are discovered.

4.2 Demand Construction of Summary Edges

Every time a new call-flow edge or callee start-to-exit edge is discovered, the set of summary edges must be updated. Unlike the original IFDS algorithm, which stores these computed edges in the SummaryEdge set, E-IFDS propagates new summary edges immediately. Although the SummaryEdge set is never stored, it is a straightforward matter to reconstruct it either during or immediately after tabulation if required.

The original IFDS algorithm obtains some measure of efficiency by not performing any summary edge computation at the call-site nodes. This efficiency is predicated on static construction of $E^\sharp$. Since E-IFDS uses demand construction of $E^\sharp$, this particular efficiency is impossible. Since there is no reason to store the SummaryEdge set in the E-IFDS algorithm, any new summary edge discovered is propagated immediately on lines 19 and 31 in Algorithm 4.1.

Whenever a new call-flow edge is discovered, it is composed with the known set of path-edges that represent the called procedure, then with the return-flow-function for that procedure. These functions correspond with lines 17 and 18 of Algorithm 4.1. Likewise, whenever a new path-edge is discovered at the exit node of a procedure, it is composed with the known set of call-flow edges to that procedure, and then with the return-flow-function for that procedure. These functions correspond with lines 29 and 30 of Algorithm 4.1.

4.3 Caller Context for Return-Flow Functions

The original IFDS algorithm cannot provide return-flow-functions with information about which call-flow edges are taken. As illustrated below, this information may be important to certain analyses, but is unavailable in the original algorithm due to the assumption that $E^\sharp$ is statically constructed.

For a flow fact $n^\sharp$ at the exit node of a procedure, and each call-site node c that calls that procedure, there exists some set of edges in $E^\sharp$ that map from $n^\sharp$ to some set of facts at the return-site following c. In the original IFDS algorithm, the second query on line 23 obtains this mapping. This mapping, however, does not take into account any facts that may exist in the caller's context at c. The equivalent query in the E-IFDS algorithm calls flow$_{ret}$ on lines 18 and 29. In addition to the fact $n^\sharp$, flow$_{ret}$ also takes a fact in the caller's context $c^\sharp$. $n^\sharp$ is always reachable from $c^\sharp$ through some fact at the procedure's start node.

Unlike the original algorithm, where each return-flow edge either exists in $E^\sharp$ or it does not, the existence of an edge from $n^\sharp$ to some particular fact at the return-site may depend on which predecessor fact $c^\sharp$ is being considered. Where the original algorithm requires a conservative approximation here, the extended algorithm is able to retain precision.

```
void caller() {
  Shape x = ...;
  ensureCircle(x);
}
void ensureCircle(Shape y) {
  Shape z = y;
  (Circle) z;
}
```

Figure 4.1: Caller Context Example (Java Syntax)

For example, consider a variable types analysis on the code in Figure 4.1. For the `ensureCircle` procedure, the original IFDS algorithm is able to deduce that `y` and `z` point to the same object, and that this object must be a subtype of `Circle` if `ensureCircle` exits normally. This information appears in the PathEdge set as $\langle s_{\texttt{ensureCircle}}, \langle \texttt{y, Shape}\rangle\rangle \rightarrow \langle e_{\texttt{ensureCircle}}, \langle \texttt{z, Circle}\rangle\rangle$.

Because the original IFDS algorithm represents all flow-functions as edges in the exploded supergraph, the only pieces of information available to the return-flow function are the called procedure's exit node, i.e. $e_{\texttt{ensureCircle}}$, the facts at that exit node, i.e. $\langle \texttt{z, Circle}\rangle$, and the caller's return-site node, i.e. some $r \in Ret_{\texttt{caller}}$. Although $E^\sharp$ contains the edge $\langle c, \langle \texttt{x, Shape}\rangle\rangle \rightarrow \langle s_{\texttt{ensureCircle}}, \langle \texttt{y, Shape}\rangle\rangle$, where c is the call-site node in $Call_{\texttt{caller}}$, there is no way for the tabulation to use this piece of information to infer anything about `x` after the call to `ensureCircle` returns. As a result, it must conservatively say that `x` may be any `Shape`.

To correct this problem, E-IFDS includes $\langle c, \langle \texttt{x, Shape}\rangle\rangle$ as the second parameter to the return-flow-function flow$_{ret}$, which indicates that `x` points to the same object as the first parameter to flow$_{ret}$, e.g. `z`. The PathEdge set contains $\langle s_{\texttt{ensureCircle}}, \langle \texttt{y, Shape}\rangle\rangle \rightarrow \langle e_{\texttt{ensureCircle}}, \langle \texttt{z, Circle}\rangle\rangle$, and the CallEdgeInverse set contains $\langle s_{\texttt{ensureCircle}}, \langle \texttt{y, Shape}\rangle\rangle \rightarrow \langle c, \langle \texttt{x, Shape}\rangle\rangle$. Therefore, the algorithm is able to determine that a valid path-edge to $\langle e_{\texttt{ensureCircle}}, \langle$`z, Circle`$\rangle\rangle$ goes through $\langle c, \langle$`x, Shape`$\rangle\rangle$. That is, `x` in `caller` points to the same object as `z` in `ensureCircle`, and this object is a subtype of `Circle` after `ensureCircle` completes. This determination is made on line 28 of the extended algorithm.

4.4 Multiple Called Procedures Per Call Site

The original IFDS algorithm assumes a single called procedure per call-site. In practice, this can be problematic for the analysis of object-oriented programs that include virtual functions or reflection. Proper analysis of such programs requires either that the dataflow dependency graph G^* be transformed so that each possible procedure call is associated with a unique call-site node, or that the algorithm be augmented to support multiple called procedures per call-site. E-IFDS incorporates the latter option.

In the E-IFDS algorithm, the *calledProc* function, which maps a call-site node to a single target, is replaced by the *calledProcs* function, which maps a call-site node to a set of potential targets.

4.5 Redundant Fact Removal

The original IFDS algorithm treats each element in the fact set D equally, without assuming any kind of structural relationship among the elements of D. For many analyses, however, structural relationships among the elements of D do exist. For example, an analysis designed to determine possible types of variables might define the elements of D to have the form $\langle v, T\rangle$ where v is a variable name and T is a possible type of v. For example, if a variable x at a particular node is determined to be of type `Shape` or of type `Circle`, and `Circle` is a subtype of `Shape`, then the fact associating x with `Circle` is redundant.

For an arbitrary analysis to take advantage of set structure, it must define a partial order $\leq$ on the elements of D, and a corresponding partial order $\sqsubseteq$ on the subsets of D. Normally, the relationship between $\leq$ and $\sqsubseteq$ is such that:

$$D_1 \sqsubseteq D_2 \iff \forall d_1 \in D_1 \exists d_2 \in D_2 \text{ such that } d_1 \leq d_2$$

To ensure that the analysis always terminates, the flow-functions must be monotone on this partial order, i.e. $a \sqsubseteq b \Rightarrow \text{flow}(a) \sqsubseteq \text{flow}(b)$.

For the types-analysis example, $\langle x, \texttt{Circle}\rangle \leq \langle x, \texttt{Shape}\rangle$ means that the fact "x is a subtype of `Circle`" is less than or equal to "x is a subtype of `Shape`." Because `Circle` is a subtype of `Shape`, the former statement implies the latter. Therefore, if both of these facts

```
        procedure Propagate($e$)
        begin
[9]       Let $d_1 \to \langle n, d_2 \rangle = e$
[9.1]     if $\nexists\ d_1 \to \langle n, d_3 \rangle \in$ PathEdge such that $d_2 \leq d_3$ then
[9.2]        Remove all edges $d_1 \to \langle n, d_4 \rangle$ such that $d_4 \leq d_2$ from PathEdge
[9.3]        Insert $e$ into PathEdge
[9.4]        Insert $e$ into WorkList
[9.5]     fi
        end
```

Algorithm 4.2: E-IFDS Propagate with Redundant Fact Removal

are present at a single program node, it is possible to remove the "smaller" fact $\langle x, \texttt{Circle} \rangle$ without affecting the soundness or precision of the analysis.

The Extended Propagate procedure in Algorithm 4.2, which is a drop-in replacement for the Propagate procedure in 4.1, only propagates a particular edge if no "equivalent" or "larger" facts (with respect to the ordering defined by $\leq$) already exist in the PathEdge set. This condition is shown on line 9.1. Before an edge is propagated, all "smaller" facts are removed from PathEdge on line 9.2. In contrast, the original propagate procedure only checks for the existence of an "equivalent" fact; it does not take into account any "larger" or "smaller" relationships among facts.

4.6 Additional Presentation Notes

Edges in the original algorithm are presented in the form $\langle m, d_1 \rangle \to \langle n, d_2 \rangle$. However, storing m explicitly is redundant. For all path-edges stored by E-IFDS, m is always equal to $s_{procOf(n)}$. Therefore, edges stored by E-IFDS are presented in the equivalent form $d_1 \to \langle n, d_2 \rangle$.

The notation $d_1 \to \langle n, d_2 \rangle$ is equivalent to the tuple $\langle d_1, n, d_2 \rangle$. Similarly, the notation $\langle p, d_1 \to \langle n, d_2 \rangle \rangle$ is equivalent to the tuple $\langle p, d_1, n, d_2 \rangle$. Because these tuples logically refer to edges in $E^\sharp$, the arrow notation is preferred for clarity.

The set D^0 includes all elements of the fact set D plus the zero fact. Formally, $D^0 = D \cup \{\mathbf{0}\}$.

Line 28 of the E-IFDS algorithm contains a spurious condition "$c \in callers(p)$." Although not strictly necessary, this condition is retained for continuity of presentation with the original IFDS algorithm and the IFDS-A algorithm.

Chapter 5

The IFDS-A/AD Algorithms

The E-IFDS algorithm improves performance over the original IFDS algorithm through demand construction of the exploded supergraph and through removal of redundant facts. However, neither algorithm is able to harness any performance gains that may be possible due to multi-core architectures. The IFDS Actors algorithm, or IFDS-A, is designed to correct this problem. IFDS-A is the first known IFDS-derived algorithm expressed in concurrent form.

IFDS-A takes the same parameters and produces the same results as E-IFDS, but takes advantage of parallelism opportunities. Algorithm 5.1 shows the main IFDS-A algorithm. Algorithm 5.2 defines the IFDS-A actors that respond to path-edge propagation.

The differences between E-IFDS and IFDS-A include:

- E-IFDS operates on shared data sets, but IFDS-A partitions these data sets into actor-local sets.
- The IFDS-A design encodes decisions about *place*, or the logical location where computations are performed.
- IFDS-A introduces a Tracker object to solve the problem of knowing when the computation is complete.

The IFDS-A algorithm is built on a conceptually straightforward abstraction. Each node in the supergraph G^* is represented by a single actor, called a *node-actor*, and each path-edge propagation is represented by a message passed from one actor to another.

IFDS-A's Tabulate function signature in Algorithm 5.1 is identical to E-IFDS's Tabulate function signature in Algorithm 4.1. Both Tabulate functions take the supergraph G^*, which consists of the set of instruction nodes N^*, the set of control-flow edges E^*, and a

```
       algorithm Tabulate($G^*$, flow$_i$, flow$_{call}$, flow$_{ret}$, flow$_{thru}$)
       begin
[1]      Let ($N^*$, $E^*$, $s_{main}$) = $G^*$
[2]      for each $n \in N^*$, let $p = procOf(n)$ do
[3]        switch $n$
[4]          case $n \in Call_p$ : $N^A[n]$ := new CallSiteActor($n$) end case
[5]          case $n = e_p$ : $N^A[n]$ := new ProcExitActor($p$) end case
[6]          case $n \in (N_p - Call_p - \{e_p\})$ : $N^A[n]$ := new IntraActor($n$) end case
[7]        end switch
[8]      od
[9]      Tracker := new TrackerActor($currentThread$)
[10]     Propagate($s_{main}$, AddPathEdge$\langle \mathbf{0}, \mathbf{0} \rangle$)
[11]     Wait for Done$\langle\rangle$
[12]     for each $n \in N^*$ do
[13]       $X_n$ := { $d_2 \in D \mid \exists\, d_1 \in D^0$ such that $d_1 \rightarrow d_2 \in N^A[n]$.PathEdge }
[14]     od
       end

       pure function Propagate($n$, $message$)
       begin
[15]     Send synchronous Inc$\langle\rangle$ to Tracker
[16]     Send $message$ to $N^A[n]$
       end

       def TrackerActor($receiver$)
[17]     local count: Integer := 0
       begin ($message$) switch
[18]     case $message$ matches Inc$\langle\rangle$ : count := count + 1
[19]     case $message$ matches Dec$\langle\rangle$ :
[20]       count := count - 1
[21]       if count = 0 then Send Done$\langle\rangle$ to $receiver$ fi
       end
```

Algorithm 5.1: The Top-Level IFDS-A Algorithm

reference to the start node of the main function s_{main}. Both Tabulate functions also take the set of flow-functions flow$_i$, flow$_{call}$, flow$_{ret}$, and flow$_{thru}$, which must be referentially transparent.

Lines 2 through 8 in Algorithm 5.1 create the set of node-actors. There is exactly one node-actor for each element of N^*. There are three different types of node-actors: the CallSiteActor, the ProcExitActor, and the IntraActor. These three types of actors correspond to the three different types of behaviours in E-IFDS, i.e. the behaviours that correspond to the cases $n \in Call_p$, $n = e_p$, and $n \in (N_p - Call_p - \{e_p\})$, respectively. Compare the switch statement starting on line 3 of Algorithm 5.1 with the switch statement starting on line 12 of Algorithm 4.1.

The algorithm begins by propagating the dummy edge $\mathbf{0} \rightarrow \mathbf{0}$ to the node s_{main} on line 10 of Algorithm 5.1. This is similar to lines 2 through 5 in Algorithm 4.1, which add the dummy edge explicitly to PathEdge and WorkList, then call the tabulation procedure directly. The asynchronous nature of IFDS-A requires that it explicitly wait for tabulation to complete on line 11 of Algorithm 5.1 before proceeding.

The Actor model provides no built-in way of determining the status of other actors; each actor is aware of its own local state only. As a result, once tabulation begins, there is no easy way to determine when it is complete. The Tracker actor is introduced to solve this problem. The Tracker actor, which is created on line 9 and defined on lines 17 through 21 of Algorithm 5.1, keeps track of the total number of unprocessed node-actor messages, and sends a message to wake up the main thread when no unprocessed messages remain. The activity of the Tracker is in some ways similar to a barrier, except that the total number of unprocessed tasks may increase as a result of processing previous tasks. Before a node-actor message may be dispatched, the Tracker must be notified by an Inc$\langle\rangle$ message, which increases the internal count by one. Correspondingly, the Tracker must be notified of the completion of message processing by a Dec$\langle\rangle$ message, which decreases the internal count by one. When the count reaches zero as a result of a Dec$\langle\rangle$ message, indicating that all outstanding messages have been processed, the Tracker sends a Done$\langle\rangle$ message back to the main thread informing it that tabulation has been completed. To prevent the count from reaching zero before tabulation has been completed, there are two conditions that must be met. First, the Tracker must receive the Inc$\langle\rangle$ message before the corresponding Dec$\langle\rangle$ message. Second, if the processing of any node-actor message results in the sending of any new node-actor messages, then the Tracker must receive the Inc$\langle\rangle$ corresponding to each of the new messages before it receives the Dec$\langle\rangle$ corresponding to the original node-actor message. To meet these conditions, it is sufficient to send the Inc$\langle\rangle$ message synchronously, as is done in the Propagate function on line 15 of Algorithm 5.1. Although this logically incurs an overhead cost of two messages sent to the Tracker and one message received from the Tracker for every node-actor message passed, in practice this overhead cost is minimal because the implementation (see Section 6.2) reduces the Inc$\langle\rangle$ and Dec$\langle\rangle$ operations to hardware-supported atomic integer operations.

Both the IFDS-A and the E-IFDS Tabulate functions compute identical results. Lines 12 through 14 in Algorithm 5.1 build the same result set X_n as lines 6 through 8 in the E-IFDS algorithm, Algorithm 4.1. The only difference between these two computations of X_n is that IFDS-A queries the contents of each node-actor to obtain the result, whereas E-IFDS queries a single PathEdge set.

5.1 IFDS-A Node-Actor Classes

Algorithm 5.2 defines the IFDS-A node-actor classes. The three node-actor classes in IFDS-A are CallSiteActor, which handles all path-edge propagation at call-site nodes; ProcExitActor, which handles all path-edge propagation at procedure exit nodes; and IntraActor, which handles path-edge propagation at other types of nodes.

Each node-actor contains a local PathEdge set, which stores all of the intra-procedural

```
def CallSiteActor($n$)
  local PathEdge: set of pairs from $D^0 \times D^0 := \emptyset$
  local CallEdge: set of triples from ProcName $\times D^0 \times D^0 := \emptyset$
  local CalleePathEdge: set of triples from ProcName $\times D^0 \times D^0 := \emptyset$
begin (message) switch
  case message matches AddPathEdge$\langle d_1, d_2 \rangle$ :
    if $d_1 \rightarrow d_2 \notin$ PathEdge then
      Insert $d_1 \rightarrow d_2$ into PathEdge
      for each $p, d_3$ such that $p \in calledProcs(n)$ and $d_3 \in \text{flow}_{call}(\langle n, d_2 \rangle, p)$ do
        Propagate($s_p$, AddPathEdge$\langle d_3, d_3 \rangle$)
        Insert $\langle p, d_2 \rightarrow d_3 \rangle$ into CallEdge
        for each $d_4$ such that $\langle p, d_3 \rightarrow d_4 \rangle \in$ CalleePathEdge do
          for each $d_5 \in \text{flow}_{ret}(\langle e_p, d_4 \rangle, \langle n, d_2 \rangle)$ do
            Propagate($returnSite(n)$, AddPathEdge$\langle d_1, d_5 \rangle$)
          od
        od
      od
      for each $d_3 \in \text{flow}_{thru}(\langle n, d_2 \rangle)$ do
        Propagate($returnSite(n)$, AddPathEdge$\langle d_1, d_3 \rangle$)
      od
    fi
  case message matches AddCalleePathEdge$\langle p, d_1, d_2 \rangle$ :
    Insert $\langle p, d_1 \rightarrow d_2 \rangle$ into CalleePathEdge
    for each $d_4$ such that $\langle p, d_4 \rightarrow d_1 \rangle \in$ CallEdge do
      for each $d_5 \in \text{flow}_{ret}(\langle e_p, d_2 \rangle, \langle n, d_4 \rangle)$ do
        for each $d_3$ such that $d_3 \rightarrow d_4 \in$ PathEdge do
          Propagate($returnSite(n)$, AddPathEdge$\langle d_3, d_5 \rangle$)
        od
      od
    od
  finally : Send Dec$\langle\rangle$ to Tracker
end

def ProcExitActor($p$)
  local PathEdge: set of pairs from $D^0 \times D^0 := \emptyset$
begin (message) switch
  case message matches AddPathEdge$\langle d_1, d_2 \rangle$ :
    if $d_1 \rightarrow d_2 \notin$ PathEdge then
      Insert $d_1 \rightarrow d_2$ into PathEdge
      for each $c \in callers(p)$ do Propagate($c$, AddCalleePathEdge$\langle p, d_1, d_2 \rangle$) od
    fi
  finally : Send Dec$\langle\rangle$ to Tracker
end

def IntraActor($n$)
  local PathEdge: set of pairs from $D^0 \times D^0 := \emptyset$
begin (message) switch
  case message matches AddPathEdge$\langle d_1, d_2 \rangle$ :
    if $d_1 \rightarrow d_2 \notin$ PathEdge then
      Insert $d_1 \rightarrow d_2$ into PathEdge
      for each $m, d_3$ such that $n \rightarrow m \in E^*$ and $d_3 \in \text{flow}_i(\langle n, d_2 \rangle)$ do
        Propagate($m$, AddPathEdge$\langle d_1 \rightarrow d_3 \rangle$)
      od
    fi
  finally : Send Dec$\langle\rangle$ to Tracker
end
```

Algorithm 5.2: IFDS-A Node-Actor Classes

path-edges that terminate at the node-actor's node. Specifically, if some fact d_2 at some node n is reachable from some fact d_1 at the start of the procedure $s_{procOf(n)}$, and d_1 is reachable from the initial fact $\langle s_{main}, \mathbf{0} \rangle$, then the PathEdge set of the node-actor corresponding to n contains the edge $d_1 \rightarrow d_2$ by the time tabulation is completed.

The CallSiteActor, defined on lines 1 through 29 of Algorithm 5.2, defines three sets and responds to two types of messages. The AddPathEdge$\langle\rangle$ message informs the CallSiteActor of a path-edge that reaches the CallSiteActor node, and AddCalleePathEdge$\langle\rangle$ informs the CallSiteActor of a path-edge that reaches the exit node of a procedure called from the CallSiteActor node. The three sets contained by the CallSiteActor are:

- The PathEdge set contains path-edges that reach the call-site node associated with the CallSiteActor.
- The CallEdge set contains the reachable flow-function edges from the call-site node to the start nodes of called procedures.
- The CalleePathEdge set contains the path-edges that reach the exit nodes of called procedures.

The contents of the CallEdge set, the contents of the CalleePathEdge set, and the return-flow-function flow$_{ret}$ suffice to generate the set of summary edges that begin at the call-site node.

The CallEdge set contains the set of call-flow edges that are reachable from the call-site node. Each call-flow edge is a 3-tuple $\langle p, d_2, d_3 \rangle$, or $\langle p, d_2 \rightarrow d_3 \rangle$, where p is the procedure being called, d_2 is a reachable fact at the call-site node, and d_3 is the destination fact at the start node of procedure p. The contents of the CallEdge set, which are inserted on line 9, are the memoized edges discovered from calling flow$_{call}$ on line 7. Just as in E-IFDS, the purpose of this memoization is to provide the inverse of flow$_{call}$ that is needed to compute summary edges. The CalleePathEdge set contains the reachable path-edges that have been discovered at a called procedure's exit node. CalleePathEdge elements are 3-tuples of the form $\langle p, d_3, d_4 \rangle$, or $\langle p, d_3 \rightarrow d_4 \rangle$, where p is a called procedure, d_3 is a fact at the procedure's start node, and d_4 is a fact at the procedure's exit node. The CalleePathEdge set provides the necessary path-edges for summary edge generation since the PathEdge set of the called procedure's exit node is not available from within the CallSiteActor.

When the CallSiteActor receives an AddPathEdge$\langle\rangle$ message, it propagates three different types of path-edges. The first type of propagated edge is the call-flow edge, which is discovered on line 7, propagated on line 8, and memoized on line 9. The second type of propagated edge is the path-edge generated in response to the generation of a new summary edge. Given a new call-flow edge, lines 10 and 11 generate the set of facts that should appear at the return-site. The set of generated summary edges is the set of edges starting at the fact d_2 at the call-site node and ending at each of the d_5 facts at the return-site node. Instead of generating these summary edges explicitly, the composition of the path-edge at the call-site and each summary edge is propagated directly. Line 12 propagates the path-edge from the source fact d_1 to each summary edge destination fact d_5. The third type of propagated edge is the flow-through edge that represents information that is unmodified by called procedures. The flow-through edges are discovered on line 16 and propagated on line 17.

The second message received by the CallSiteActor is the AddCalleePathEdge⟨⟩ message. The ProcExitActor sends this message whenever a new callee path-edge is discovered. This edge is memoized on line 21 for possible future use by an AddPathEdge⟨⟩ message. Lines 22 and 23 compose the set of CallEdges, the new callee path-edge, and the return-flow edges to obtain the set of possibly-new summary edges. Line 24 composes the set of reachable path-edges at the call-site node with these new summary edges, and the resulting path-edges are propagated to the return-site node on line 25. For E-IFDS, this propagation occurs in the end-node logic at lines 28 through 34 of Algorithm 4.1. However, because this propagation depends on the path-edges that belong to the call-site, IFDS-A forwards new callee path-edges to the call-sites for processing. Because summary edges are always composed with path-edges received at the call-site node, and the resulting path-edges are always propagated to the return-site node that follows the call-site node, it is logical to handle all summary edge logic from within the CallSiteActor.

The ProcExitActor, defined on lines 30 through 36 of Algorithm 5.2, is only responsible for forwarding all path-edges received at a procedure exit node to all of the corresponding call-site nodes. Although this causes replication of all such path-edges to all applicable call-sites, it avoids versioning issues that may otherwise occur if these path-edges were only stored in a single location. Specifically, execution of the exit node logic must be serializable with respect to the execution of the call-site logic.

The IntraActor, defined on lines 37 through 45 of Algorithm 5.2, performs path-edge propagation for all non-call-site and non-exit nodes. Line 41 queries the intra-procedural flow-function, and line 42 propagates the composition of the incoming path-edge with the result of this flow-function.

One consequence of defining the PathEdge sets in this manner described in this section is that a redundant path-edge can only be rejected at the destination node-actor. The sender of the AddPathEdge⟨⟩ message does not have access to the destination node-actor's PathEdge set, and therefore cannot know whether or not the new AddPathEdge⟨⟩ message is redundant.

Instead of defining PathEdge to contain edges that end at the current node, it is possible to define PathEdge to contain edges that end at successor nodes. This change enables many redundant path-edges to be rejected before being sent to successor nodes. However, this approach causes redundant messages to be sent to node-actors that have multiple predecessors, and each predecessor must store its own copy of every path-edge, increasing storage requirements and causing an increase in the number of redundant-edge checks that must be executed. How beneficial this alternative PathEdge definition could be depends on the efficiency of the message-passing implementation, the nature of the input program, and the nature of the analysis used.

In summary, the IFDS-A algorithm requires several changes and additions with respect to the E-IFDS algorithm. The Tracker object solves the problem of knowing when tabulation is complete. Shared data sets in E-IFDS are partitioned into actor-local data sets. In the case of IFDS-A, a straightforward partitioning of this data among actors is possible, where the notion of *place*, or the logical location where a computation is performed, becomes important. In particular, the design encodes decisions about where to check for already-seen path-edges and where to handle path-edges appearing at called-procedure exit nodes. The properties of the Actor model automatically exclude the possibility of deadlock and shared-memory corruption. The handling of each message is atomic with respect to all other messages, removing any need to reason about instruction-level execution interleavings. However, in cases where two or more actors contribute updates to the same set of data, reasoning about message-level interleavings becomes necessary. For example, ProcExitActors forward callee path-edges to CallSiteActors so that the summary edge generation and propagation code does not miss any updates to the callee path-edge sets.

5.2 Redundant Fact Removal

IFDS-A can support redundant fact removal in a fashion similar to the Extended Propagate Procedure in Algorithm 4.2. If a partial order $\leq$ is applied to the elements of the fact-set D in accordance with the conditions specified in Section 4.5, then it is possible to support redundant fact removal by extending the conditionals that check for prior existence of path-edges. Algorithm 5.3 shows the replacement of the conditional on line 5 of Algorithm 5.2. Similar replacements apply to lines 32 and 39.

```
[5]      if $\nexists\, d_1 \to d_3 \in$ PathEdge such that $d_2 \leq d_3$ then
[5.1]        Remove all edges $d_1 \to d_4$ such that $d_4 \leq d_2$ from PathEdge
```

Algorithm 5.3: IFDS-A Redundant Fact Removal

Because path-edges at exit nodes are sent to the CallSiteActor, redundant edge removal is also moved to the CallSiteActor. Algorithm 5.4 shows the addition of the removal operation into the call-site logic.

```
[20.1]    Remove all edges $\langle p, d_1 \rightarrow d_3 \rangle$ such that $d_3 \leq d_2$ from CalleePathEdge
```

Algorithm 5.4: IFDS-A Redundant Fact Removal on Callee-Path-Edges

5.3 Using Detach for Increased Concurrency

IFDS-A provides substantial concurrency because each node-actor is able to execute independently of other node-actors in the system. However, messages received by any particular node-actor are processed serially. This serialization can cause an execution bottleneck if the number of outstanding messages destined for distinct actors is smaller than the number of OS (Operating System) threads for any length of time. (For the purposes of this discussion, there is a 1:1 correspondence between the number of OS threads assigned to execute actor-based computations and the number of independent program-counters, or "processors," supplied by the underlying hardware.) This bottleneck can be especially problematic for relatively long computations, such as those performed by the CallSiteActor.

If there were a way to de-serialize some of the CallSiteActor computations, these computations could be performed on otherwise-idle OS threads while the CallSiteActor itself is freed to begin processing additional messages. Some parts of the CallSiteActor cannot be de-serialized because they contain updates to the actor's mutable state. Other parts of the CallSiteActor, however, only perform computations and send messages without needing to touch mutable state.

The **detach** construct is a novel construct designed to perform exactly this function. The code between the **begin detach** and **end detach** keywords may be spawned as a separate execution task that is concurrent with the surrounding code. There is an arbitrary length of time between spawning the detached section and its execution.

A practical implementation of the **detach** function takes as an argument a first-class function closure, or a structure containing a function pointer and the data the function requires. The closure contains the code inside the **detach** section. The **detach** function passes this closure to the scheduler's `execute` method, which queues the closure for execution by an OS thread. The details of the scheduler's implementation determine which OS thread executes the closure, and when the execution actually takes place. Note that it is semantically valid (but not necessarily beneficial) to avoid calling the scheduler altogether by simply executing the closure in-line.

```
def storeAndForward(nextActor)
  local edgeSet: set of edges := $\emptyset$
begin (message) switch
  case message matches AddEdge$\langle edge \rangle$ :
    Insert edge into edgeSet
    begin detach
      Send AddEdge$\langle edge \rangle$ to nextActor
    end detach
end
```

Figure 5.1: Detach Example

```
def storeAndForward(nextActor)
  local edgeSet: set of edges := $\emptyset$
begin (message) switch
  case message matches AddEdge$\langle edge \rangle$ :
    Insert edge into edgeSet
    Let detached = new _anon_(edge, nextActor)
    Send $\langle \rangle$ to detached
end
def _anon_(edge, nextActor)
begin (message) switch
  case $\langle \rangle$ :
    Send AddEdge$\langle edge \rangle$ to nextActor
end
```

Figure 5.2: Meaning of Detach

For example, Figure 5.1 shows a simple actor that receives a message with a single value, stores that value in a local mutable set, and forwards that value to another actor. The insertion statement is not part of the detached section because it updates mutable state, but the message send statement is part of the detached section because it does not touch mutable state. This code is small for the sake of example. Real programs would normally only use detached sections for operations that are more computationally expensive than a single message send.

Figure 5.2 shows how Figure 5.1 could be de-sugared into an actor-based program without **detach**. Logically, execution of a detached section is equivalent to spawning a new actor and sending that actor a single message telling it to execute the detached code. The only variables accessible by the detached section are those that can be passed by value to the new actor. The detached section does not see any modifications of variables that may occur after the detached section is spawned.

Algorithms 5.5, 5.6, and 5.7 show modified versions of CallSiteActor, ProcExitActor, and IntraActor which take advantage of the **detach** construct. These actor definitions combined with the top-level IFDS-A code in Algorithm 5.1 collectively constitute the IFDS Actors-with-Detach algorithm, or IFDS-AD.

Note that the Tracker must be informed when a detached execution takes place. Because the detached execution constitutes a logically independent unit of work, sending a synchronous Inc⟨⟩ to the Tracker before entering each detached section and an asynchronous Dec⟨⟩ when the detached section completes ensures that the Tracker count only reaches zero when tabulation completes.

Within each detached section, no interaction with mutable state can occur. To accomodate this requirement, the algorithm requires certain minor adjustments. For example, the operation of the detached section beginning on line 9.3 of Algorithm 5.5 depends only on the contents of d_1, d_2, D_4, n, and p. The query of CalleePathEdge, previously on line 10, becomes a query of the fact set D_4. The query of CalleePathEdge moves to line 9.1 and

```
def CallSiteActor($n$)
  local PathEdge: set of pairs from $D^0 \times D^0 := \emptyset$
  local CallEdge: set of triples from ProcName $\times D^0 \times D^0 := \emptyset$
  local CalleePathEdge: set of triples from ProcName $\times D^0 \times D^0 := \emptyset$
begin (message) switch
  case message matches AddPathEdge$\langle d_1, d_2 \rangle$ :
    if $d_1 \rightarrow d_2 \notin$ PathEdge then
      Insert $d_1 \rightarrow d_2$ into PathEdge
      for each $p, d_3$ such that $p \in calledProcs(n)$ and $d_3 \in \mathrm{flow}_{call}(\langle n, d_2 \rangle, p)$ do
        Propagate($s_p$, AddPathEdge$\langle d_3, d_3 \rangle$)
        Insert $\langle p, d_2 \rightarrow d_3 \rangle$ into CallEdge
        Let $D_4 = \{ d_4 \mid \langle p, d_3 \rightarrow d_4 \rangle \in$ CalleePathEdge $\}$
        Send synchronous Inc$\langle\rangle$ to Tracker
        begin detach
          for each $d_4 \in D_4$ do
            for each $d_5 \in \mathrm{flow}_{ret}(\langle e_p, d_4 \rangle, \langle n, d_2 \rangle)$ do
              Propagate($returnSite(n)$, AddPathEdge$\langle d_1, d_5 \rangle$)
            od
          od
          Send Dec$\langle\rangle$ to Tracker
        end detach
      od
      Send synchronous Inc$\langle\rangle$ to Tracker
      begin detach
        for each $d_3 \in \mathrm{flow}_{thru}(\langle n, d_2 \rangle)$ do
          Propagate($returnSite(n)$, AddPathEdge$\langle d_1, d_3 \rangle$)
        od
        Send Dec$\langle\rangle$ to Tracker
      end detach
    fi
  case message matches AddCalleePathEdge$\langle p, d_1, d_2 \rangle$ :
    Insert $\langle p, d_1 \rightarrow d_2 \rangle$ into CalleePathEdge
    for each $d_4$ such that $\langle p, d_4 \rightarrow d_1 \rangle \in$ CallEdge do
      Let $D_3 = \{ d_3 \mid d_3 \rightarrow d_4 \in$ PathEdge $\}$
      Send synchronous Inc$\langle\rangle$ to Tracker
      begin detach
        for each $d_5 \in \mathrm{flow}_{ret}(\langle e_p, d_2 \rangle, \langle n, d_4 \rangle)$ do
          for each $d_3 \in D_3$ do
            Propagate($returnSite(n)$, AddPathEdge$\langle d_3, d_5 \rangle$)
          od
        od
        Send Dec$\langle\rangle$ to Tracker
      end detach
    od
  finally : Send Dec$\langle\rangle$ to Tracker
end
```

Algorithm 5.5: IFDS-AD Call-Site Node-Actor Class

```
       def ProcExitActor($p$)
[30]     local PathEdge: set of pairs from $D^0 \times D^0 := \emptyset$
       begin ($message$) switch
[31]     case $message$ matches AddPathEdge$\langle d_1, d_2 \rangle$ :
[32]       if $d_1 \rightarrow d_2 \notin$ PathEdge then
[33]         Insert $d_1 \rightarrow d_2$ into PathEdge
[33.1]       Send synchronous Inc$\langle\rangle$ to Tracker
[33.2]       begin detach
[34]           for each $c \in callers(p)$ do Propagate($c$, AddCalleePathEdge$\langle p, d_1, d_2 \rangle$) od
[34.1]         Send Dec$\langle\rangle$ to Tracker
[34.2]       end detach
[35]       fi
[36]     finally : Send Dec$\langle\rangle$ to Tracker
       end
```

Algorithm 5.6: IFDS-AD Procedure-Exit Node-Actor Class

```
       def IntraActor($n$)
[37]     local PathEdge: set of pairs from $D^0 \times D^0 := \emptyset$
       begin ($message$) switch
[38]     case $message$ matches AddPathEdge$\langle d_1, d_2 \rangle$ :
[39]       if $d_1 \rightarrow d_2 \notin$ PathEdge then
[40]         Insert $d_1 \rightarrow d_2$ into PathEdge
[40.1]       Send synchronous Inc$\langle\rangle$ to Tracker
[40.2]       begin detach
[41]           for each $m, d_3$ such that $n \rightarrow m \in E^*$ and $d_3 \in$ flow$_i(\langle n, d_2 \rangle)$ do
[42]             Propagate($m$, AddPathEdge$\langle d_1 \rightarrow d_3 \rangle$)
[43]           od
[43.1]         Send Dec$\langle\rangle$ to Tracker
[43.2]       end detach
[44]       fi
[45]     finally : Send Dec$\langle\rangle$ to Tracker
       end
```

Algorithm 5.7: IFDS-AD Intraprocedural Node-Actor Class

the results are cached in the immutable set D_4 that is passed into the detached section. This prevents future modifications of CalleePathEdge from interfering with the operation of the detached section.

Chapter 6

Evaluation

This chapter discusses the implementation and empirical evaluation of the IFDS-A and IFDS-AD algorithms.

6.1 The Variable Type Analysis

The analysis used for evaluation is the Variable Type Analysis, or VTA, described by Naeem et al. [NLR10]. This analysis defines the fact-set D to be the set of all pairs $\langle v, T\rangle$ where v is a variable and T is a class type in the source program. The presence of a fact $\langle v, T\rangle$ in the result set means that the variable v may point to an object of type T. Stated differently, the presence of $\langle v, T\rangle$ means that the analysis is unable to prove that v will *not* point to an object of type T.

The set of types is structured because some types are subtypes of other types. If facts $\langle v, T\rangle$ and $\langle v, \mathit{superclass}(T)\rangle$ both exist in the same subset, then $\langle v, T\rangle$ is redundant and may be removed. Furthermore, this fact does not need to be considered in any future processing.

6.2 Implementation Details

The E-IFDS, IFDS-A, and IFDS-AD algorithms and VTA flow-functions are implemented in the Scala language. The original IFDS algorithm is not implemented because it is impractical for use with VTA.

The Scala language has high-level facilities for managing sets and maps, for partial functions, and for pattern matching. As a result, many of the lines in the algorithms have a one-to-one correspondence with lines of Scala code.

The node-actor classes inherit from the `Actor` class in the Scala Actor library. This class defines an abstract method called `act` that defines the actor's behaviour. Figure 6.1 shows the implementation of `act` for all the node-actors.

```
def act = {
  loop {
    react {
      case msg:  Any => {
        message(msg)
        Tracker.dec()
      }
    }
  }
}

def message(msg:  Any) = {
  msg match {
    case AddPathEdge(d1, d2) => ...
  }
}
```

Figure 6.1: Node-Actor Reaction Loop in Scala

Each node-actor also implements a `message` function that takes the incoming message as an argument. Because the incoming message is of type `Any`, a supertype of all other types, it is dynamically matched against type-patterns to determine what type of message was actually sent.

The `loop` and `react` "keywords" are implemented as functions in the Actor library; Scala's flexible syntax and support for first-class functions and partial functions enable library functions to look like built-in constructs. The `react` function waits for a message to be received and then matches it against a set of **`case`** statements. The `react` function does not block; if no message is available, it returns control to the scheduler. In essence, `react` blocks a user-thread, not an OS thread. One advantage of this approach is that any number of user-threads may be mapped to a small number of pooled OS threads, thereby conserving memory and reducing context-switching time. In this implementation, there is a 1:1 correspondence between user-threads and actors, a 1:n correspondence between user-threads and pooled OS threads, and a 1:1 correspondence between pooled OS threads and independent hardware program-counters, or "processors."

One disadvantage of using `react` is that it loses its calling context when it returns control to the scheduler. As a result, `react` never returns after it executes the contents of its body. The `loop` function mitigates this issue by implicitly creating a new actor to restart execution after `react` completes.

```
. . . Initialize node-actors . . .
val s_main = . . . get the Start node of the main function . . .
propagate(s_main, (Zero, Zero))
self.receive {
  case Done =>
}
. . . Collect results and return . . .
```

Figure 6.2: Top-Level IFDS-A Solver Code

The Actor library also provides a `receive` function that blocks an OS thread while waiting for a message. The top-level IFDS solver code uses `receive` to wait for the algorithm to finish executing. Figure 6.2 shows how the IFDS-A algorithm starts execution and then waits for the actors to finish. `Zero` is a constant representing the zero fact in D^0. The `self` object represents the currently-executing actor. If the current execution context is just an OS thread, then the Actor library generates an implicit proxy object that allows the thread to behave like an actor. This code in this figure corresponds to lines 10 and 11 in Algorithm 5.1.

The Tracker object is implemented as a Scala object that contains an `AtomicInteger` count variable. Calls to the `Tracker.inc()` function increment this count, and calls to `Tracker.dec()` decrement this count and send a `Done` message to the main thread when the count reaches zero.

The algorithm implementations are compiled with Scala 2.8.0 Beta 1 and run with Oracle JRockit JVM version 3.1.2. All of the implemented algorithms extend Soot [VRCG$^+$99], a Java bytecode optimization framework.

6.3 Testing Methodology

6.3.1 Modeling Ideal Performance

Arriving at an accurate assessment of algorithm performance on a multi-core machine requires an accurate model of *ideal machine performance*. Without an accurate model, it is difficult to assess the algorithm's true scalability. Processor architectures are complex and varied, and multiple processors may share common resources such as memory busses, caches, and computation units. Furthermore, the load placed on these shared resources may vary considerably from application to application. Consequently, it is inaccurate to assume a perfectly linear speedup model, i.e. that two non-communicating parallel threads can perform twice as much work as one, and four can perform twice as much as two, etc.

Building a model of ideal machine performance starts with the Worklist-A algorithm. The Worklist-A algorithm computes results identical to IFDS-A, except that the concurrent task-scheduling and message-passing systems are replaced by a single-threaded scheduler which uses a work-list to track unprocessed messages. For a single thread, the difference in execution time between IFDS-A and Worklist-A is the cost of task-scheduling and message-passing overhead.

Running t instances of Worklist-A on t parallel OS threads provides an estimation of the maximum work that a particular machine can perform per unit time. If the machine exhibits truly linear speedup, then the execution of t parallel instances occurs in the same amount of wall-clock time as the execution of a single instance. If the execution of parallel instances produces contention for shared machine resources, then some degree of slowdown is expected as t increases. Running a large number of Worklist-A instances in parallel causes the allocation of a very large amount of memory, making a complete solve infeasible for larger benchmarks. The Ideal solver alleviates this problem by executing precisely the same set of work units in the same manner as Worklist-A, but only produces a partial solution for the given input. Whereas Worklist-A executes until its work-list is empty, the Ideal solver terminates after 50,000 executing work units, regardless of the total number of work units required to produce a complete solution (e.g. Worklist-A with the Variable Type Analysis on the antlr input executes approximately 9.1 million work units). Under the assumption that the first 50,000 work units are a representative sample of the complete set of work units in terms of memory consumption, types of computation performed, etc., then:

$$I_t/I_1 = W_t/W_1$$

where I_t and W_t are respectively the execution times of t parallel instances of Ideal and Worklist-A.

Therefore, the expected limit to the speedup S of IFDS-A given t OS threads is:

$$S_t = tI_1/I_t$$

6.3.2 Performance Assessment

The *scalability*, or *self-speedup*, of IFDS-A at t OS threads is the increase in work-per-unit-time relative to IFDS-A with one OS thread, or:

$$\text{SelfSpeedup} = A_1/A_t$$

where A_1 and A_t are the execution time of IFDS-A with one thread and t threads, respectively.

The *reference speedup* of IFDS-A at t OS threads is its performance relative to E-IFDS. Given A_t, or IFDS-A execution time with t threads, and E, or E-IFDS execution time, then:

$$\text{ReferenceSpeedup} = E/A_t$$

Reference speedup is also computed for IFDS-AD.

The *computational efficiency* of IFDS-A at t OS threads is

$$\text{Efficiency} = W_1/(S_t A_t)$$

where S_t is the speedup limit derived from the Ideal solvers at t threads. Intuitively, this efficiency is the proportion of available processor time IFDS-A spends performing computations that contribute to the final result. Any inefficiency, i.e. the "wasted" computational time $1-$ Efficiency, is attributable to scheduling overhead, message-passing overhead, and idle processor time. For a single thread, all inefficiency results from scheduling and message-passing overhead. For larger numbers of threads, lack of sufficient parallelism opportunities may cause inefficiency due to idle processor time.

6.3.3 Data Collection and Handling

Due to the difficulty of obtaining exclusive access to the machines used to perform the experiments, the raw execution timings exhibit some degree of noise and bias. Identifying and removing outliers from the data set mitigates the effect of noise, although noise may still cause the reported error ranges to be larger than they otherwise would have been. Comparing ratios instead of raw data eliminates the effect of bias in the results.

Timing data are collected in sets of replications, where each replication contains one run of each of the following solvers:

$$E, W_1, \{I_t | t \in \{1...T\}\}, \{A_t | t \in \{1...T\}\}, \{D_t | t \in \{1...T\}\}$$

where E is E-IFDS, W is Worklist-A, I is Ideal, A is IFDS-A, D is IFDS-AD, and $\{1...T\}$ is the set of thread counts tested. Within each replication, solvers are executed in randomized order.

Means and confidence intervals are taken with respect to performance, then inverted to obtain speedups. For example, the self-speedup mean and error range is computed using the following:

$$\begin{aligned}
\text{SelfSpeedup}_t &= 1/\text{mean}(\text{all } A_t/A_1)\\
\text{SelfSpeedup}_{tmin} &= 1/(\text{mean}(\text{all } A_t/A_1)+\text{confidence}(\text{all } A_t/A_1, \alpha = 0.05))\\
\text{SelfSpeedup}_{tmax} &= 1/(\text{mean}(\text{all } A_t/A_1)-\text{confidence}(\text{all } A_t/A_1, \alpha = 0.05))
\end{aligned}$$

6.4 Results

All tests were performed on two machines. The first machine is a 4-socket SMP machine with a dual-core AMD Opteron in each socket. It has 16 gigabytes of memory and a total of eight processors. The remainder of this document uses the name Opteron8 to refer to this machine.

The second machine is a single-socket Sun UltraSparc T2. It has 32 gigabytes of memory and 64 processors. The remainder of this document uses the name Sparc64 to refer to this machine.

6.4.1 Input Characteristics

Input	Methods	Variables	Instructions	Possible Types
antlr	949	10,839	16,621	257
jython	5489	56,090	74,031	1079
luindex	1306	12,519	18,131	617

Table 6.1: Input Characteristics

The test programs are part of the DaCapo Benchmark Suite version 2006-10-MR2 [BGH$^+$06]. The antlr, jython, and luindex benchmarks are selected as inputs. Table 6.1 contains the number of methods, variables, instructions, and types within each of the inputs. The number of instructions is the same as the number of nodes in the G^* graph. The size of the fact-set D is the number of variables times the possible types.

Table 6.2 shows the parallelism characteristics of the inputs. The Total Work Units column is the total number of messages processed during execution of the single-thread Worklist-A solver. The IFDS-AD statistics are gathered with a version of Worklist-A that also counts each **detach** section as an additional work unit. Note that the exact number of work units processed by IFDS-A and IFDS-AD may vary from run to run due to non-deterministic ordering of work units. The Mean Parallel Width column is the mean number of operations that may be executed in parallel at any given time. To derive the mean parallel width, the work-list execution loop in Worklist-A is transformed into a doubly-nested loop. The outer loop finds a maximal set of work units from the work-list

Input	Algorithm	Total Work Units	Mean Parallel Width	Parallel Height	Mean Grain Size Opteron8	Mean Grain Size Sparc64
antlr	IFDS-A	9.1 mil.	86.5	104,873	32 μs	193 μs
	IFDS-AD	15.0 mil.	142.7		19 μs	117 μs
jython	IFDS-A	80.9 mil.	134.7	600,395	7 μs	27 μs
	IFDS-AD	107.2 mil.	178.6		5 μs	20 μs
luindex	IFDS-A	14.4 mil.	4.8	301,653	42 μs	243 μs
	IFDS-AD	19.6 mil.	6.5		31 μs	179 μs

Table 6.2: Parallelism Characteristics

that could be executed in parallel, and the inner loop executes this set of work units. Two work units are deemed to be executable in parallel if and only if they operate on different actors. Of the three inputs, luindex is likely to be at a performance disadvantage because its mean parallel width is relatively small when compared with antlr and jython. The Parallel Height column is the number of iterations that the outer loop requires to complete execution. The parallel height is the length of the *critical path*, or the longest chain of sequentially-dependent work units. The exact length of the critical path may vary depending on the order in which the work units are processed. The parallel height remains the same whether or not **detach** sections are enabled, since **detach** sections are presumed to execute in the same outer-loop iteration as their parent work units. The Mean Grain Size is the average length of time required to execute a work unit, computed by taking the quotient of the total work units and the mean Worklist-A execution time. The grain size affects execution time because larger grain sizes are more easily able to amortize the overhead costs of message passing and task scheduling. Of the three inputs, jython is likely to be at a performance disadvantage because its mean grain size is relatively small when compared with antlr and luindex.

Figure 6.3 shows graphically the parallel width, or *available parallelism*, for each input as a function of the percentage of work units completed. These charts provide some indication of how many processors the algorithm can keep busy for the given input, and for how long. For example, antlr provides sufficent parallelism to keep 100 processors busy for the first 85% of work units processed. The last 15% of work units will take longer than 15% of the total execution time because of insufficient parallelism to keep 100 processors busy. Jython has a similarly large available parallelism. In contrast, luindex has considerably smaller available parallelism, where a substantial fraction of the total work done exhibits a parallel width of only 2 units. This lack of available parallelism may severely limit the scalability of luindex.

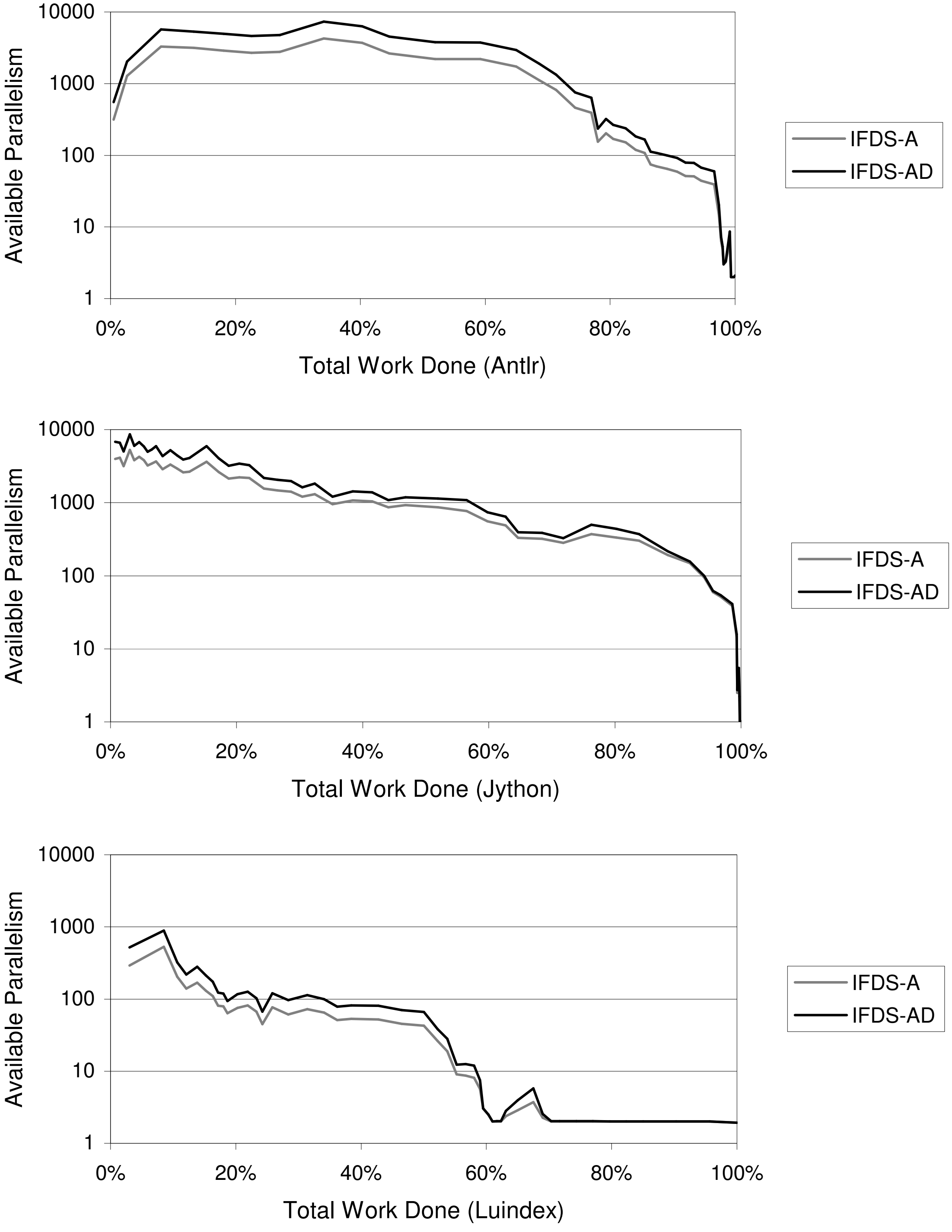

Figure 6.3: Available-Parallelism Charts

6.4.2 Ideal Performance

An *ideal speedup function* $S(t)$ models the speedup possible on real machines assuming zero message-passing and scheduling overhead and unlimited available parallelism. Deriving $S(t)$ requires first that the mean of all ideal speedups S_t for all inputs be taken, and secondly that a model be fit to these means. No significant difference in Ideal solver performance between antlr, jython, and luindex was found, so the mean speedup is taken across all I_t/I_1 samples. The ideal speedup function for the Opteron8 machine, shown in Figure 6.4, is chosen to be a best-linear-fit function under the constraint $S(1) = 1$. (The $S(1) = 1$ constraint reflects the fact that one thread always yields a speedup of 1 by definition.) The ideal fit function for Opteron8 is $S(t) = 0.8154t + 0.1846$, which says that the expected ideal performance improvement for each additional OS thread added is approximately 81.5%. The measured ideal speedup S_2 is somewhat lower than the ideal model speedup $S(2)$, but because the measured IFDS-A speedups for two threads exceed S_2, it is likely that there is some detrimental interaction between the Ideal solver and the hardware at two OS threads which does not occur when running IFDS-A. Therefore, $S(2)$ is assumed to be closer to the actual ideal performance.

The Sparc64 machine is fit with a quadratic function, as shown in Figure 6.5. Unlike Opteron8, where each processor had dedicated computational resources, each "processor" on Sparc64 shares computational resources with seven other "processors." It is, therefore, reasonable to expect a linear (or nearly-linear) speedup up to eight OS threads, but also a reduction in the speedup factor thereafter. The architecture of the Sparc64 machine has eight compute cores, each of which maintains eight hardware threads. For numbers of OS threads larger than eight, the ideal speedup possible on this machine depends on the application's characteristic behaviour, particularly the ratio of compute time to memory latency. The characteristic behaviour of IFDS-A with the Variable Type Analysis at eight threads is nearly linear ($S(8) = 7.8$), but is notably sub-linear at 64 threads ($S(64) = 38.5$). One anomaly encountered on Sparc64 is that the performance of the Ideal solver bifurcates at 64 OS threads. The faster 50% of the Ideal solver timings have a mean speedup of 38.5x with a narrow error interval, but the slower 50% have a mean speedup of 6.4x with a similarly narrow error interval. Because the majority of IFDS-A timings exceed 6.4x speedup at 64 threads, and 38.5x provides a much cleaner fit to an ideal model, the slower 50% of Ideal timings at 64 threads are likely to be the result of an interaction between the Ideal solver and the hardware that does not occur when testing IFDS-A. Therefore, the ideal fit function $S(t)$ does not take the slower 50% into account. Figure 6.5 shows the slower 50% as Ideal Data B.

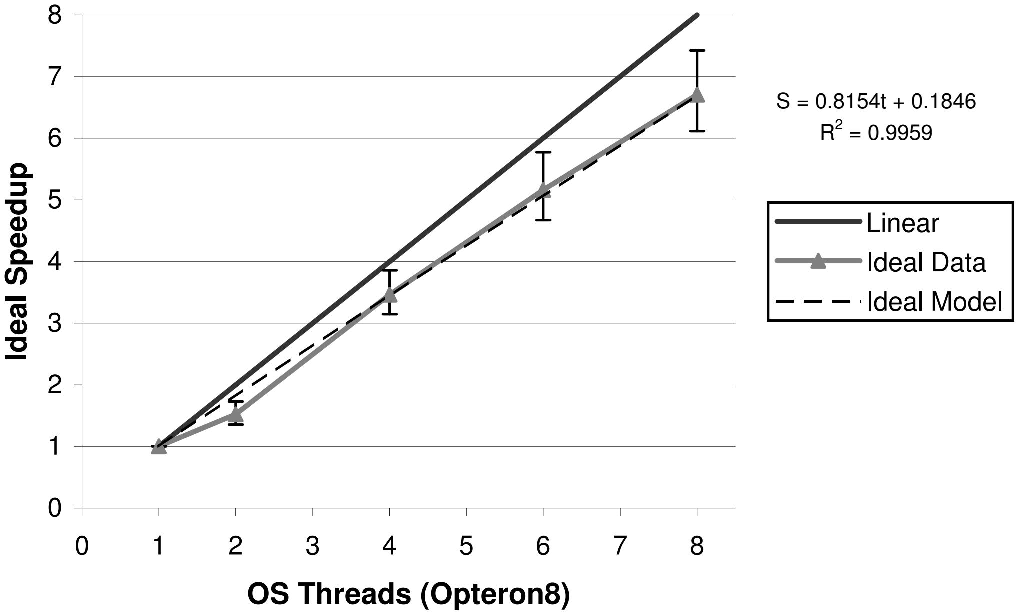

Figure 6.4: Ideal Speedup Model (Opteron8)

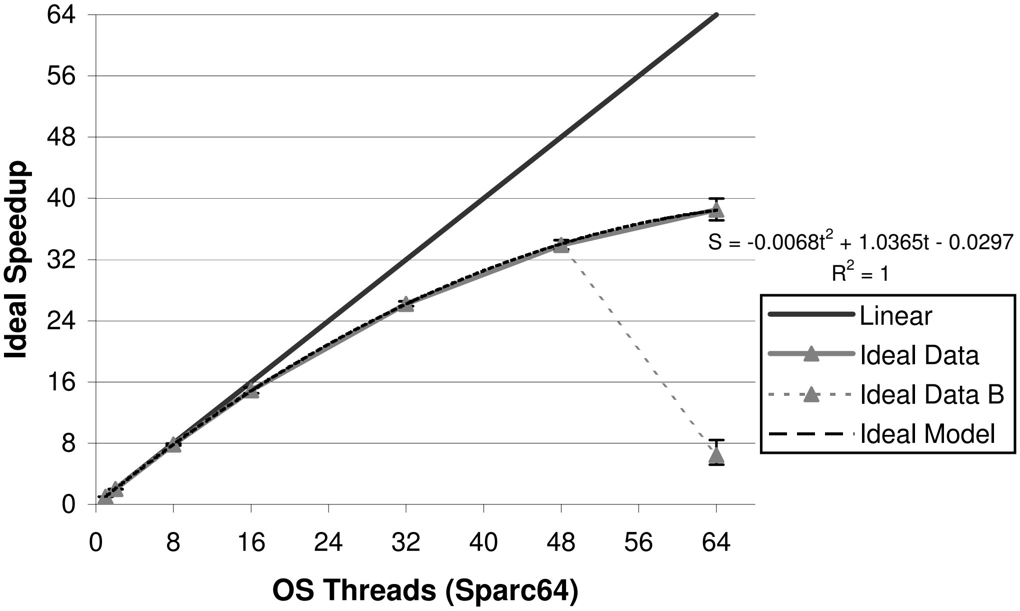

Figure 6.5: Ideal Speedup Model (Sparc64)

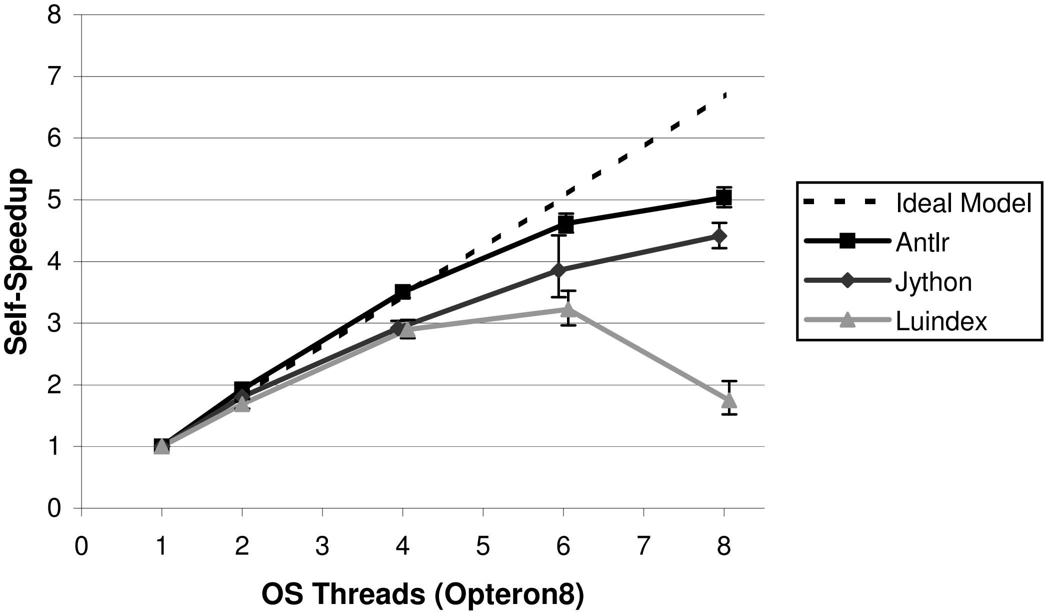

Figure 6.6: IFDS-A Self-Speedup (Opteron8)

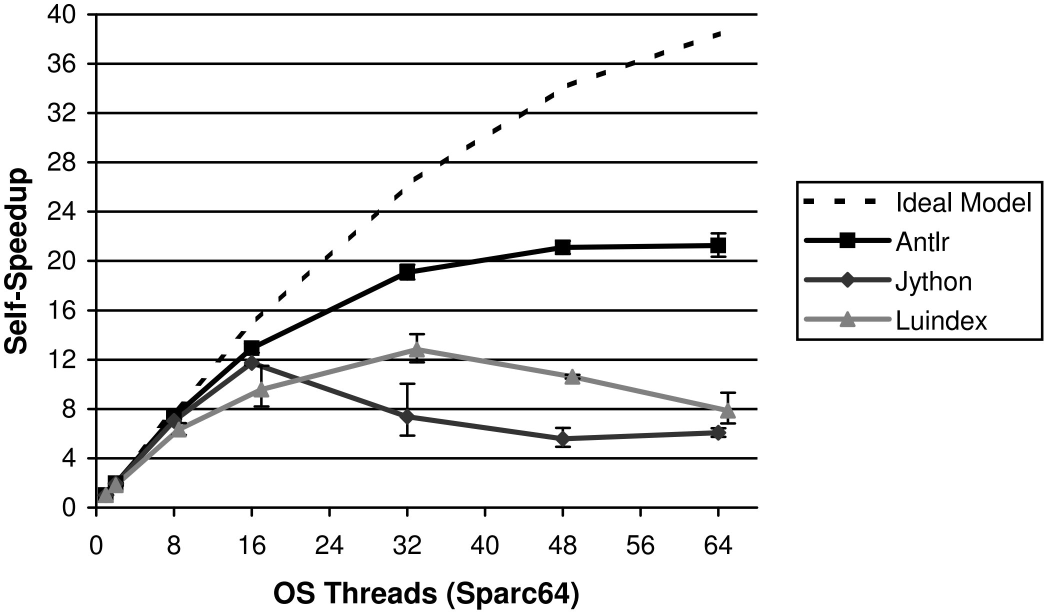

Figure 6.7: IFDS-A Self-Speedup (Sparc64)

6.4.3 Performance Assessment

Figure 6.6 shows the self-speedup of IFDS-A for the three inputs on Opteron8, along with the ideal fit function $S(t)$. Data points in this and following graphs may be adjusted slightly on the horizontal axis to show the error interval bars clearly. Antlr exhibits the best self-speedup of about 5x out of an ideal maximum of 6.7x using 8 OS threads. Luindex exhibits the worst self-speedup, achieving a maximum of 3.2x at 6 OS threads, which decreases to 1.7x at 8 OS threads. Since luindex exhibits a mean parallel width of only 4.8, it is reasonable to expect that it reaches maximum performance between 4 and 6 threads. Adding additional threads beyond 6 is likely to incur additional overhead without significantly increasing parallelism.

Figure 6.7 shows self-speedup on Sparc64. As expected, antlr shows the best self-speedup, peaking at 21.3x out of an ideal maximum of 38.5x using 64 OS threads. Luindex peaks at 32 OS threads despite a mean parallel width of 4.8. A possible explanation for this behaviour is that it takes 32 OS threads to fully utilize four cores. Jython self-speedup peaks at 16 OS threads before dropping below luindex performance. A possible reason for this behaviour is that Jython's small grain size causes significant synchronization overhead with larger thread counts.

Figures 6.8 and 6.9 show reference speedups on Opteron8 and Sparc64, respectively. Peak speedups over E-IFDS on Opteron8 are 2.8x, 1.6x, and 1.5x for antlr, jython, and luindex, respectively. Peak speedups on Sparc64 are 11.2x, 3.2x, and 5.5x, respectively.

Figure 6.10 shows the computational efficiency of IFDS-A on Opteron8. As expected, the computational efficiency tends to decrease as the number of OS threads increases. Antlr exhibits the highest efficiency. The efficiency of luindex decreases more rapidly than antlr because its mean available parallelism is smaller, dropping to 24% efficiency at eight threads while antlr maintains 62% efficiency. The efficiency of jython is lower than antlr because it incurs more overhead due to a smaller grain size, but jython's efficiency decreases at approximately the same rate at antlr's due to similar levels of available parallelism. At one thread, jython's efficiency is 17% lower than antlr's, and at eight threads jython's efficiency is 16% lower than antlr's.

Figure 6.11 shows the computational efficiency of IFDS-A on Sparc64. Antlr and luindex both have an efficiency of 81% at one OS thread, but the efficiency of luindex drops much more rapidly than antlr due to lack of available parallelism. At 64 threads, antlr maintains a 45% efficiency, but luindex drops to 17% efficiency. The cost of a smaller grain size on Sparc64 is higher than on Opteron8. At one thread, jython only reaches a 47% efficiency, and at 64 threads, it drops to 7%. Jython's peak self-speedup occurs at 16 threads (see Figure 6.7), where it maintains a 37% efficiency. At 32 threads, jython's efficiency drops to 13%, outweighing any benefit gained by additional parallelism. A possible cause

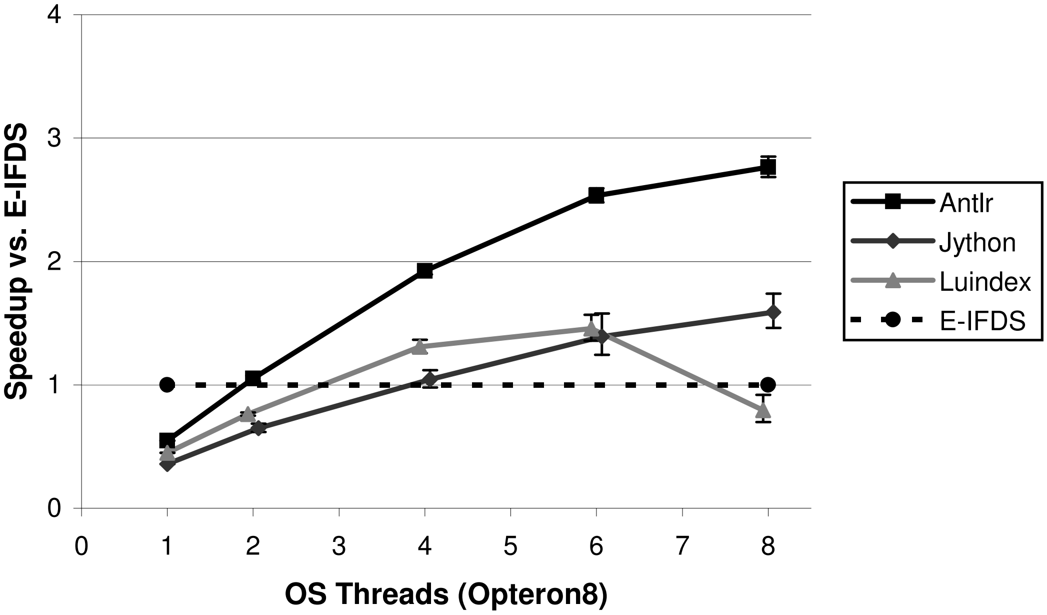

Figure 6.8: IFDS-A Speedup vs. E-IFDS (Opteron8)

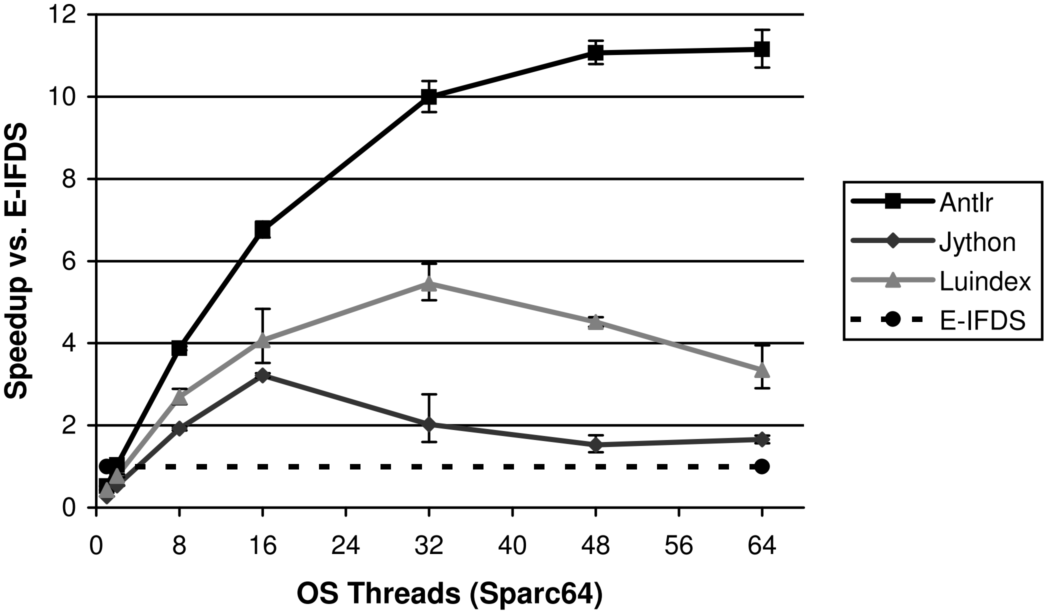

Figure 6.9: IFDS-A Speedup vs. E-IFDS (Sparc64)

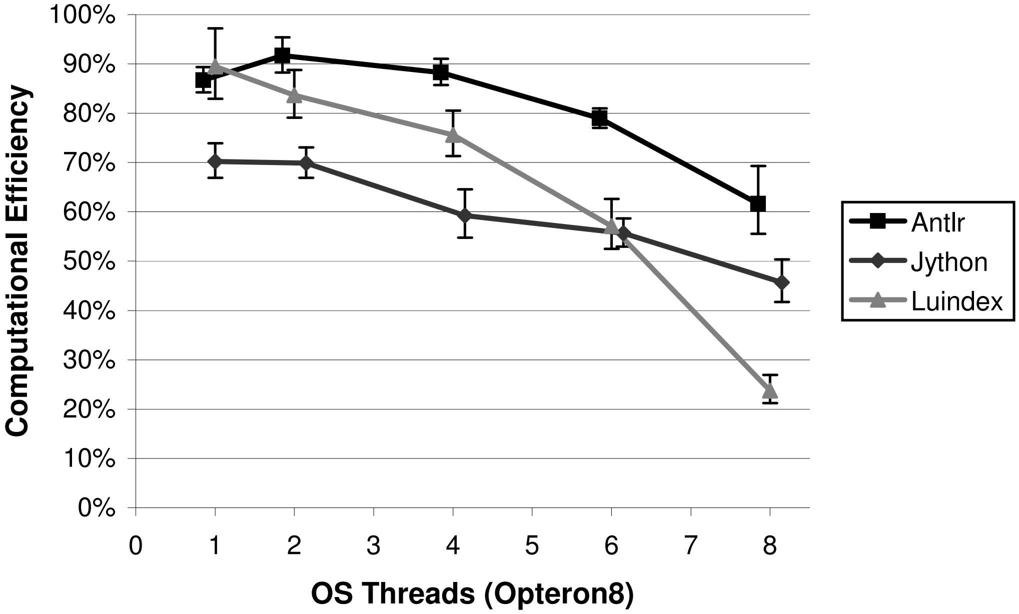

Figure 6.10: IFDS-A Computational Efficiency (Opteron8)

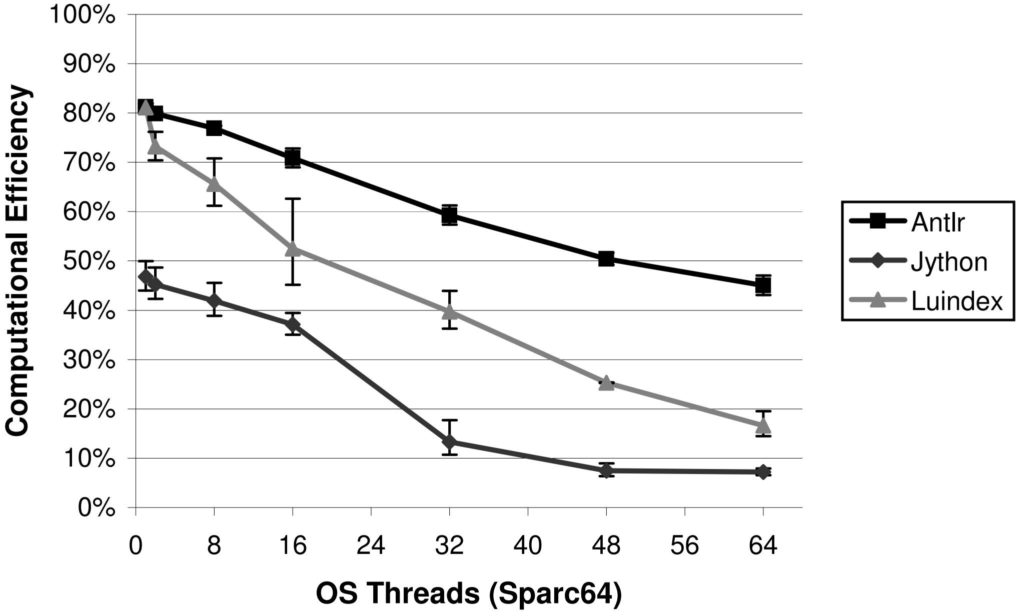

Figure 6.11: IFDS-A Computational Efficiency (Sparc64)

of this efficiency drop is that contention for entry into `synchronized` methods reaches a critical performance threshold between 16 and 32 threads due to small grain size.

	1-Thread Overhead		1-Thread Efficiency		"Best" Reference Speedup		"Best" Efficiency	
Input	Opteron8	Sparc64	Opteron8	Sparc64	Opteron8	Sparc64	Opteron8	Sparc64
antlr	13%	19%	87%	81%	2.8x	11.2x	62%	45%
jython	30%	53%	70%	47%	1.6x	3.2x	46%	37%
luindex	11%	19%	89%	81%	1.5x	5.5x	57%	40%

Table 6.3: IFDS-A Overhead and Efficiency Summary

Table 6.3 summarizes the efficiency and overhead for each of the inputs. Shown in this table are the overhead costs and the computational efficiencies of IFDS-A, the maximum speedups IFDS-A achieves relative to E-IFDS, and the computational efficiencies at these speedups.

Figures 6.12 and 6.13 show the mean reference speedups across all inputs for IFDS-A and IFDS-AD. These results do not indicate any significant difference between the performance of IFDS-A and IFDS-AD. Although IFDS-AD has a larger mean parallel width than IFDS-A, it also incurs a proportionally larger scheduling overhead. Furthermore, IFDS-AD does not significantly increase parallel width in the low-parallelism "tail" of the computation, as shown in Figure 6.3. Lack of parallelism in this "tail" results from a lack of new work – most of the work done at the end of the computation is checking for previously-seen or redundant path-edges.

In summary, the major findings of these experiments include:

- The maximum attainable performance of IFDS-A with the Variable Type Analysis depends on the characteristics of the input. Larger work unit sizes and greater available parallelism tend to increase maximum attainable performance.
- Maximum self-speedups range from 3.2x to 5.0x on the Opteron8 machine, and from 11.7x to 21.3x on the Sparc64 machine.
- Relative to the E-IFDS reference implementation, maximum speedups range from 1.5x to 2.8x on the Opteron8 machine, and from 3.2x to 11.2x on the Sparc64 machine.
- The IFDS-A implementation spends a significant amount of time performing message-passing and scheduling activities, between 11% and 53% of total processing time for a single thread.

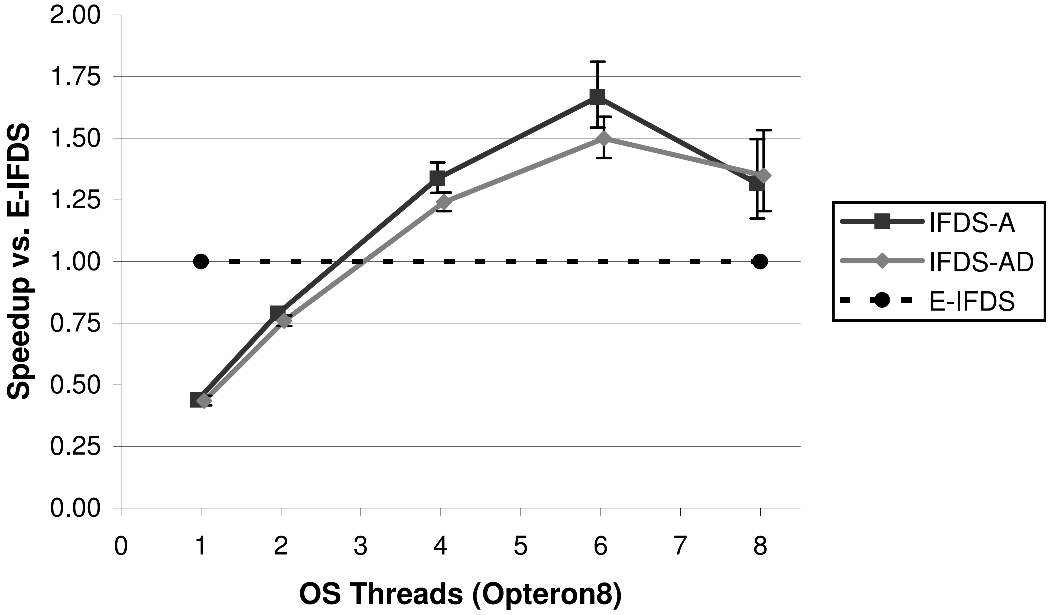

Figure 6.12: IFDS-A / IFDS-AD Speedup (Opteron8)

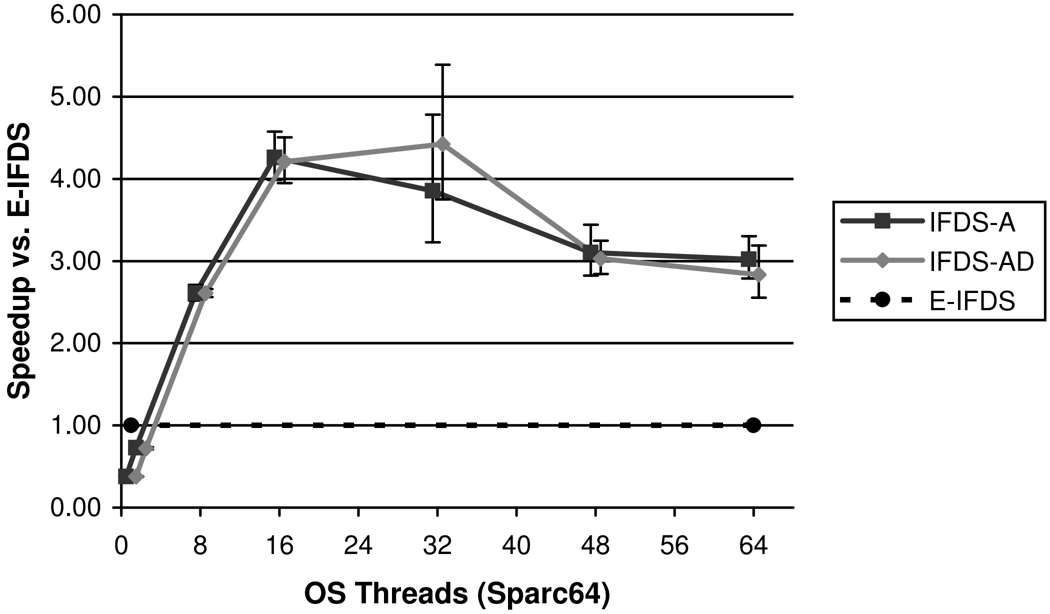

Figure 6.13: IFDS-A / IFDS-AD Speedup (Sparc64)

- The performance of IFDS-AD does not differ significantly from IFDS-A. Although IFDS-AD increases available parallelism, that increase does not offset the costs of increased scheduling overhead.

Chapter 7

Conclusions

Most current compilers and program-analysis tools fail to take full advantage of multi-core computer architectures. Dataflow-analysis algorithms, which are prime consumers of computation time in these applications, are generally not parallelized due to the difficulty of constructing concurrent formulations of such algorithms using the threads model.

This work, culminating in the creation of the IFDS Actors algorithm, or IFDS-A, is a case study that examines the application of the Actor model to a dataflow-analysis algorithm, following the intuition that the Actor model might lend itself readily to the concurrent formulation of such an algorithm.

The Actor model was developed over 30 years ago in anticipation of distributed and multi-core computing. Despite its theoretical maturity as a computational model, there are currently very few practical applications of this model. The practical importance and usefulness of the model remains unproven.

The first contribution of this work is a presentation of the E-IFDS algorithm, which contains extensions to the IFDS algorithm that are able to increase its performance and utility for many types of analyses. The E-IFDS algorithm allows for demand construction of the exploded supergraph, demand construction of summary edges, providing caller context to return-flow functions, multiple called procedures per call site, and removal of redundant facts.

The second contribution of this work is the IFDS-A algorithm, which computes the same results as E-IFDS, but is able to make use of multi-core computers. IFDS-AD, a variant of IFDS-A, offers increased concurrency through use of the **detach** construct.

IFDS-A and IFDS-AD were implemented in the Scala language and Scala's Actor library. Subjectively, the implementation process was very straightforward; no deadlocks were possible, and very little reasoning about data conflicts or consistency was required.

The key findings of this work are:

- The maximum self-scalability of IFDS-A ranges from 3.2x to 5.0x on an eight-processor machine, and from 11.8x to 21.3x on a 64-processor machine. The maximum ideal scalability for this algorithm is 6.7x on the eight-processor machine and 38.5x on the 64-processor machine.
- The maximum speedup IFDS-A achieves over the sequential E-IFDS algorithm ranges from 1.5x to 2.8x on an eight-processor machine, and from 3.2x to 11.2x on a 64-processor machine.
- No significant difference in performance between IFDS-A and IFDS-AD was found. The **detach** construct, although able to provide increased concurrency, was not able to translate that increase into improved performance.

Open questions include:

- How well does IFDS-A perform with analyses other than the Variable Type Analysis?
- What other types of analysis algorithms are easily expressible in Actor-based terms, and do the implementations of these algorithms exhibit significant performance improvements on multi-core machines?
- Which other Actor-based algorithms can make use of the Tracker-object concept?
- Which other Actor-based algorithms can make use of the **detach** construct?
- Based on computational efficiency measurements and work-unit grain sizes, Scala's Actor library incurs and overhead of between 2 and 5 μs on the eight-processor machine for every message passed. By how much can this overhead be reduced?

This work endeavours to open up new ways of thinking about the problem of concurrency. The expression of the IFDS-A algorithm in concurrent form, although presenting certain challenges not encountered in the construction of the sequential E-IFDS algorithm, was not fundamentally difficult. Future work may yield more answers about concurrent program construction in general and the applicability of the Actor model in particular.

Appendix A

Raw Performance Data

The following tables contain the raw data obtained from performance measurements. All measurements are in wall-clock seconds. The gray cells are data points that appear to be outliers, and have been excluded from the results.

The first column in each table is the name of the solver implementation. For names that end in a number, the number is the number of OS threads used for the timings in that row.

Solver	Replications							
Implementation	1	2	3	4	5	6	7	8
E-IFDS	189.228	184.773	185.834	181.066	185.139	180.136	177.168	191.738
Worklist-A	286.596	283.375	298.303	299.274	279.837	288.911	299.501	297.857
IFDS-A-1	332.311	348.396	356.823	326.618	321.785	341.979	328.749	333.260
IFDS-A-2	183.131	179.684	169.998	169.615	175.423	167.225	176.637	244.096
IFDS-A-4	96.668	97.472	97.292	96.132	98.314	92.443	92.328	95.683
IFDS-A-6	71.351	71.944	72.320	75.866	72.907	73.502	69.155	74.798
IFDS-A-8	64.626	68.933	95.764	66.883	66.300	63.309	66.688	83.261
IFDS-AD-1	325.583	320.771	348.954	319.422	318.092	346.162	331.615	357.602
IFDS-AD-2	182.520	179.754	179.777	177.081	188.825	185.136	183.428	177.790
IFDS-AD-4	111.760	107.828	103.673	101.268	109.287	110.003	100.173	109.167
IFDS-AD-6	86.359	86.880	78.721	77.115	81.209	77.848	80.968	91.457
IFDS-AD-8	96.706	70.941	107.394	72.832	71.114	73.846	80.418	98.479
Ideal-1	0.905	0.928	0.693	0.651	0.737	0.632	1.121	1.562
Ideal-2	1.045	1.144	0.876	1.219	1.013	1.203	0.907	1.050
Ideal-4	0.844	0.947	0.882	1.027	0.926	1.083	0.832	0.960
Ideal-6	0.966	0.899	0.847	0.832	1.190	0.890	0.941	0.778
Ideal-8	0.985	0.961	0.873	0.921	0.945	0.906	1.045	1.018

Table A.1: Antlr on Opteron8

Solver	Replications							
Implementation	1	2	3	4	5	6	7	8
E-IFDS	287.265	295.413	274.480	260.580	276.809	277.441	286.731	295.708
Worklist-A	852.362	540.614	571.397	571.961	568.088	561.197	541.320	536.788
IFDS-A-1	757.337	856.845	781.768	748.138	779.294	851.608	750.182	767.941
IFDS-A-2	426.475	416.305	628.033	406.956	465.250	467.478	432.896	427.784
IFDS-A-4	255.967	313.008	253.850	259.994	256.791	306.002	273.368	240.906
IFDS-A-6	244.705	190.131	353.716	222.739	458.035	192.327	182.033	187.468
IFDS-A-8	183.657	194.450	242.065	264.517	166.739	206.817	164.868	164.463
IFDS-AD-1	727.319	724.835	747.467	720.989	871.432	789.315	805.668	741.938
IFDS-AD-2	495.896	476.648	476.638	470.418	490.577	447.277	464.617	476.092
IFDS-AD-4	278.275	324.929	279.923	284.875	2012.522	296.955	287.934	319.667
IFDS-AD-6	209.989	263.918	238.580	246.730	240.893	241.177	245.191	311.144
IFDS-AD-8	185.086	232.264	444.421	363.352	208.379	201.125	278.687	201.522
Ideal-1	0.769	0.571	0.502	0.670	0.898	0.464	0.449	0.739
Ideal-2	1.121	1.010	3.489	0.886	0.831	0.868	0.697	0.861
Ideal-4	0.780	0.847	0.646	0.897	0.658	0.677	0.692	0.757
Ideal-6	0.818	0.804	0.707	0.604	0.738	0.846	0.758	0.680
Ideal-8	0.767	0.921	0.848	0.746	0.736	3.073	0.695	0.785

Table A.2: Jython on Opteron8

Solver	Replications							
Implementation	1	2	3	4	5	6	7	8
E-IFDS	31.835	29.295	28.989	30.597	30.521	29.759	31.927	31.105
Worklist-A	57.820	55.092	58.544	58.446	65.446	68.912	62.586	61.336
IFDS-A-1	67.752	73.029	65.848	65.543	68.199	62.121	64.963	74.485
IFDS-A-2	42.428	39.276	38.767	38.823	41.073	38.085	41.763	39.629
IFDS-A-4	24.131	22.744	23.645	23.210	21.576	24.186	22.097	24.832
IFDS-A-6	18.457	18.813	20.785	24.590	19.237	20.089	22.226	23.401
IFDS-A-8	41.773	23.415	31.988	37.548	49.279	42.458	37.828	43.955
IFDS-AD-1	70.038	70.430	68.643	65.864	79.302	72.742	74.892	73.804
IFDS-AD-2	39.443	36.865	38.266	39.777	38.291	40.086	40.478	38.288
IFDS-AD-4	23.961	24.136	23.717	25.282	24.792	23.033	23.742	24.071
IFDS-AD-6	19.924	20.448	21.113	21.763	19.915	20.667	20.761	17.998
IFDS-AD-8	35.121	24.697	39.858	36.340	32.818	37.269	26.742	24.460
Ideal-1	0.821	0.660	1.256	0.871	1.117	1.088	0.919	0.880
Ideal-2	0.929	1.136	1.377	0.841	1.002	0.686	0.796	0.911
Ideal-4	0.884	0.832	1.155	0.851	0.922	0.950	0.922	0.871
Ideal-6	0.932	0.966	0.962	0.875	1.458	1.062	0.946	1.066
Ideal-8	1.021	1.037	0.942	0.978	1.341	0.949	0.940	1.030

Table A.3: Luindex on Opteron8

Solver	Replications					
Implementation	1	2	3	4	5	6
E-IFDS	1133.760	1119.000	1125.879	1122.295	1136.917	1120.009
Worklist-A	1749.923	1745.709	1746.785	1750.224	1746.153	1748.640
IFDS-A-1	2155.183	2148.884	2146.169	2168.832	2141.611	2147.903
IFDS-A-2	1081.015	1094.226	1085.663	1084.180	1079.718	1087.013
IFDS-A-8	289.037	292.011	290.523	290.399	286.610	293.431
IFDS-A-16	156.763	173.232	166.211	166.445	169.400	167.423
IFDS-A-32	110.085	112.586	110.994	120.122	106.623	115.893
IFDS-A-48	99.021	105.561	99.802	113.928	103.518	101.240
IFDS-A-64	176.727	103.440	95.526	137.622	105.617	99.177
IFDS-AD-1	2141.660	2160.317	2137.353	2172.106	2160.862	2141.978
IFDS-AD-2	1089.207	1090.732	1089.035	1093.834	1091.368	1088.746
IFDS-AD-8	294.241	309.212	290.812	295.175	293.295	294.547
IFDS-AD-16	235.389	171.635	169.409	178.688	222.839	166.332
IFDS-AD-32	112.314	116.617	111.541	110.106	114.838	114.552
IFDS-AD-48	101.845	102.315	108.536	108.708	104.209	108.526
IFDS-AD-64	132.930	101.585	111.673	137.123	101.609	149.730
Ideal-1	3.822	3.903	3.831	3.845	3.747	3.739
Ideal-2	3.875	3.871	3.851	3.868	3.782	3.763
Ideal-8	3.895	3.914	3.900	3.901	3.810	3.839
Ideal-16	4.226	4.080	4.115	4.085	3.979	3.991
Ideal-32	4.728	4.690	4.682	4.777	4.587	4.751
Ideal-48	5.518	5.439	5.531	5.536	5.761	5.429
Ideal-64	14.212	30.774	31.646	31.036	6.492	22.265

Table A.4: Antlr on Sparc64

Solver	Replications				
Implementation	1	2	3	4	5
E-IFDS	1266.852	1261.177	1245.767	1257.678	1264.439
Worklist-A	2106.569	2018.644	2254.169	2043.044	2373.160
IFDS-A-1	4609.300	4620.185	4596.172	4609.855	4556.902
IFDS-A-2	2354.222	2359.907	2299.896	2418.937	2349.710
IFDS-A-8	679.869	669.430	646.302	647.306	630.161
IFDS-A-16	380.547	393.278	390.178	396.344	397.260
IFDS-A-32	645.102	308.870	813.731	682.764	662.578
IFDS-A-48	876.473	786.107	683.791	951.098	1055.722
IFDS-A-64	728.127	754.890	732.518	823.150	1087.598
IFDS-AD-1	4588.204	4576.350	4609.596	4608.330	4562.790
IFDS-AD-2	2357.918	2372.692	2342.913	2327.696	2369.132
IFDS-AD-8	662.333	665.686	660.805	661.575	678.685
IFDS-AD-16	354.837	415.297	419.530	408.550	421.912
IFDS-AD-32	297.000	453.589	722.841	418.547	577.815
IFDS-AD-48	816.260	836.571	804.552	993.905	902.646
IFDS-AD-64	755.422	822.999	866.923	856.363	1258.727
Ideal-1	2.651	2.501	2.505	2.509	2.578
Ideal-2	2.505	2.506	2.533	2.517	2.615
Ideal-8	2.533	2.725	2.577	2.533	2.764
Ideal-16	2.694	3.133	2.652	2.706	2.771
Ideal-32	3.035	3.037	3.064	3.058	3.134
Ideal-48	3.468	3.456	3.457	3.453	3.586
Ideal-64	3.991	35.232	34.018	29.674	4.112

Table A.5: Jython on Sparc64

Solver	Replications			
Implementation	1	2	3	4
E-IFDS	186.965	183.617	181.753	182.109
Worklist-A	351.019	348.083	354.877	348.119
IFDS-A-1	431.323	433.712	431.720	431.873
IFDS-A-2	232.861	233.222	234.272	249.930
IFDS-A-8	73.990	61.601	67.540	69.880
IFDS-A-16	53.792	42.336	48.334	35.994
IFDS-A-32	35.699	35.496	29.203	34.338
IFDS-A-48	40.608	40.402	41.272	56.883
IFDS-A-64	58.313	56.404	62.356	42.467
IFDS-AD-1	434.330	431.029	439.744	432.409
IFDS-AD-2	235.571	249.309	232.739	250.010
IFDS-AD-8	63.993	67.695	66.248	65.155
IFDS-AD-16	41.638	41.810	41.236	39.011
IFDS-AD-32	33.504	33.678	33.923	34.676
IFDS-AD-48	49.393	36.948	36.074	38.950
IFDS-AD-64	48.325	50.522	65.295	77.535
Ideal-1	3.885	3.887	3.798	3.812
Ideal-2	3.833	3.898	3.823	3.827
Ideal-8	3.888	3.942	3.884	3.889
Ideal-16	4.078	4.071	4.049	4.065
Ideal-32	6.199	4.688	4.681	4.703
Ideal-48	5.510	5.463	5.431	5.464
Ideal-64	6.572	6.544	6.516	6.536

Table A.6: Luindex on Sparc64

References

[ABB$^+$07] Shivali Agarwal, Rajkishore Barik, Dan Bonachea, Vivek Sarkar, Rudrapatna K. Shyamasundar, and Katherine Yelick. Deadlock-free scheduling of X10 computations with bounded resources. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 229–240, New York, NY, USA, 2007. ACM.

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

[BB09] Alexandro Baldassin and Sebastian Burckhardt. Lightweight software transactions for games. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism (HotPar '09)*, 2009.

[BDS$^+$92] P. A. Buhr, Glen Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke. μC++: Concurrency in the object-oriented language C++. *Software - Practice and Experience*, 22(2):137–172, 1992.

[BGH$^+$06] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[BL99] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[BS06] Saisanthosh Balakrishnan and Gurindar S. Sohi. Program demultiplexing: Data-flow based speculative parallelization of methods in sequential programs. In *ISCA '06: Proceedings of the 33rd annual international symposium on*

Computer Architecture, pages 302–313, Washington, DC, USA, 2006. IEEE Computer Society.

[DLCO09] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: deterministic shared memory multiprocessing. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2009. ACM.

[DRT+09] Danny Dig, Cosmin Radoi, Mihai Tarce, Marius Minea, and Ralph Johnson. ReLooper: Refactoring for loop parallelism. Technical report, University of Illinois at Urbana-Champaign, 09 2009.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM.

[Gro07] Dan Grossman. The transactional memory / garbage collection analogy. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 695–706, New York, NY, USA, 2007. ACM.

[HB77] Carl Hewitt and Henry Baker. Laws for communicating parallel processes. In *1977 IFIP Congress Proceedings*, pages 987–992, 1977.

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

[Hew09] Carl Hewitt. ActorScript(TM): Industrial strength integration of local and nonlocal concurrency for client-cloud computing. *CoRR*, abs/0907.3330, 2009.

[HO09] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[Kaf90] Dennis Kafura. ACT++: building a concurrent C++ with actors. *J. Object Oriented Program.*, 3(1):25–37, 1990.

[KBI+09] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How much parallelism is there in irregular applications? In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 3–14, New York, NY, USA, 2009. ACM.

[Kno09] Kathleen Knobe. Ease of use with concurrent collections (CnC). In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism (HotPar '09)*, 2009.

[KPW$^+$07] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, New York, NY, USA, 2007. ACM.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lee06] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[Mur89] Tadao Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541 – 580, 1989.

[NLR10] Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez. Practical extensions to the IFDS algorithm. In *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, Berlin, 2010.

[OAA09] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 97–108, New York, NY, USA, 2009. ACM.

[OZ05] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 41–57, New York, NY, USA, 2005. ACM.

[PKA96] R. Panwar, W. Kim, and Gul Agha. Parallel implementations of irregular problems using high-level actor language. In *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*, pages 857–862, Washington, DC, USA, 1996. IEEE Computer Society.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, New York, NY, USA, 1995. ACM.

[RVDB07] Sean Rul, Hans Vandierendonck, and Koen De Bosschere. Function level parallelism driven by data dependencies. *SIGARCH Comput. Archit. News*, 35(1):55–62, 2007.

[SM98] J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. *High-Performance Computer Architecture, International Symposium on*, 0:2, 1998.

[Smo95] Gert Smolka. The Oz programming model. In *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[Ste99] Lynn Andrea Stein. Challenging the computational metaphor: Implications for how we think. *Cybernetics and Systems*, 30(6):473–507, 1999.

[Ste06] Guy L. Steele, Jr. Parallel programming and code selection in Fortress. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–1, New York, NY, USA, 2006. ACM.

[Swe06] Tim Sweeney. The next mainstream programming language: A game developer's perspective. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 269–269, New York, NY, USA, 2006. ACM.

[VRCG$^+$99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[VWW96] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in ERLANG (2nd ed.).* Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1996.

[YNW$^+$08] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 265–274, New York, NY, USA, 2008. ACM.

[ZGU$^+$09] Ferad Zyulkyarov, Vladimir Gajinov, Osman S. Unsal, Adrián Cristal, Eduard Ayguadé, Tim Harris, and Mateo Valero. Atomic quake: Using transactional memory in an interactive multiplayer game server. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 25–34, New York, NY, USA, 2009. ACM.